ARM Software Development Toolkit Version 2.0

Programming Techniques

Document Number: ARM DUI 0021A
Issued: June 1995
Copyright Advanced RISC Machines Ltd (ARM) 1995

EUROPE
Advanced RISC Machines Limited
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
Telephone: +44 1223 400400
Facsimile: +44 1223 400410
Email: info@armltd.co.uk

JAPAN
Advanced RISC Machines K.K.
KSP West Bldg, 3F 300D, 3-2-1 Sakado,
Takatsu-ku, Kawasaki-shi
Kanagawa, 213 Japan
Telephone: +81 44 850 1301
Facsimile: +81 44 850 1308
Email: info@armltd.co.uk

USA
ARM USA
Suite 5, 985 University Avenue
Los Gatos
California 95030
Telephone: +1 408 399 5199
Facsimile: +1 408 399 8854
Email: info@arm.com

Proprietary Notice

ARM, the ARM Powered logo and EmbeddedICE are trademarks of Advanced RISC Machines Ltd.

Neither the whole nor any part of the information contained in, or the product described in, this datasheet may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.
The product described in this datasheet is subject to continuous developments and improvements. All particulars of the product and its use contained in this datasheet are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties or merchantability, or fitness for purpose, are excluded.

This datasheet is intended only to assist the reader in the use of the product. ARM Ltd shall not be liable for any loss or damage arising from the use of any information in this datasheet, or any error or omission in such information, or any incorrect use of the product.

Change Log

Issue	Date	By	Change
A	June 95	PB/BH/EH/AP	Created

TOC

Contents

Contents

Contents

1 Introduction

This chapter introduces the Programming Techniques manual.

Introduction

1.1 About this manual

1.1.1 Overview

This manual is designed to help programmers rapidly exploit the power of the ARM processor for embedded applications. The material has been written by ARM staff who have accumulated considerable experience with software for the ARM and Thumb microprocessors.

We have targeted this manual at embedded systems programmers who have some experience with other architectures, and who wish to quickly learn how to use an ARM chip.

A broad spectrum of topics is covered, from introductory illustrations through to complex examples. It has been organised by theme, for example:

- Programmer's Model
  This chapter describes the ARM architecture. It includes details for system programmers writing supervisor code and exception handlers.
- Programming in C for the ARM
  This chapter is essential reading for developers who wish to optimise their code for high performance and minimum size. It describes how to write C which compiles efficiently. This approach can yield considerable gains without resorting to assembly language.
- Writing Code for ROM
  This explains the issues involved in preparing code for ROM. It describes how the linker can be used to link an image into a fragmented memory map, with RAM areas initialised automatically on startup.
- ARMulator
  This describes how the ARM emulator (ARMulator) can be modified to emulate an entire system. The ARMulator can be used to develop and debug software while hardware design proceeds in parallel.
- Writing SWI Handlers
  This chapter describes how to use the SWI (Software Interrupt) instruction to interface user code with an operating system (or other code which runs in Supervisor-mode).
- Benchmarking
  This describes ways of obtaining high performance and minimum code size when evaluating the ARM processor.

You should use this book in conjunction with the ARM Software Development Toolkit, as most of the example programs are available on-line in the toolkit's `examples` directory

You will need to refer to the ARM Software Development Toolkit Reference Manual (ARM DUI 0020) for full details of the software tools. Also, the relevant ARM Datasheet will give you specific details about the device with which you are working.

1.1.2 Conventions

Typographical conventions

The following typographical conventions are used in this manual:

`typewriter`	denotes text that may be entered at the keyboard: commands, file and program names and assembler and C source.
`typewriter-italic`	shows text which must be substituted with user-supplied information: this is most often used in syntax descriptions
Oblique	is used to highlight important notes and ARM-specific terminology.

Filenames

Unless otherwise stated, filenames are quoted in Unix format—for example:

```
examples/basicasm/gcd1.s
```

If you are using the PC platform, you must translate them into their DOS equivalent:

```
EXAMPLES\BASICASM\GCD1.S
```

1.2 Feedback

1.2.1 Feedback on the Software Development Toolkit

If you have feedback on the Software Development Toolkit, please contact either your supplier or ARM Ltd. You can send feedback via e-mail to: xdevt@armltd.co.uk.

In order to help us give a rapid and useful response, please give:

- details of which hosting and release of the ARM software tools you are using
- a small sample code fragment which reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened

1.2.2 Feedback on this manual

If you have feedback on this manual, please send it via e-mail to: documentation@armltd.co.uk, giving:

- the manual's revision number
- the page number(s) to which your comments refer
- a concise explanation of the problem

General suggestions for additions and improvements are also welcome.

2 Getting Started

This chapter introduces the components of the ARM software development toolkit, and takes you through compiling, linking and running a simple ARM program.

2.1 Introducing the Toolkit

The ARM software development toolkit is a collection of utilities for producing programs written in ARM code. The tools include emulators so that programs can be run even when real ARM hardware is unavailable to the developer.

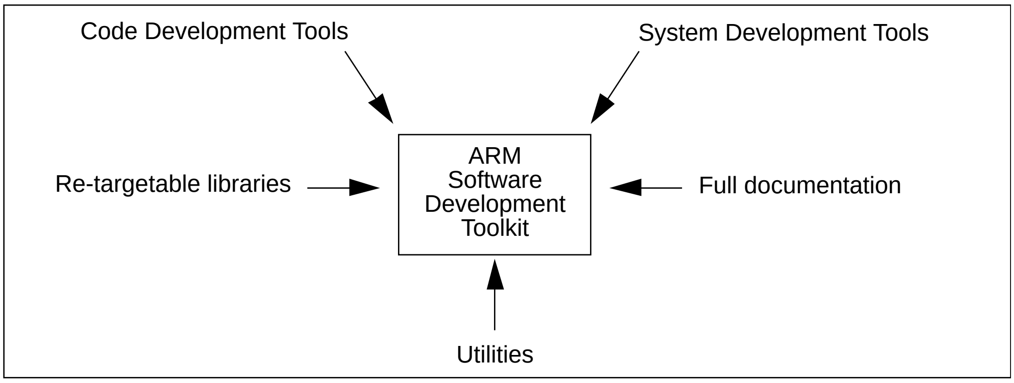


Figure 2-1: ARM Software Development Toolkit

The toolkit supports two platforms:

- IBM compatible PCs
- Sun workstations

It comprises a set of command line tools and, in the case of the IBM PC platform, a pair of applications which provide an interactive development environment in the Windows desktop.

The tools are used for two main purposes:

- Software development

  This involves building either C, C++, or ARM assembler source code into ARM object code, which is then debugged using the ARM source level debugger. The debugger has facilities which include single stepping, setting breakpoints and watchpoints, and viewing registers. Testing and debugging can be carried out on code running in a real ARM processor, or using the integrated ARM processor emulator.

- Benchmarking

  Once application code has been built, it can be benchmarked either on an ARM processor attached to the host system, or under software emulation.
  The ARM emulator can also be used to simulate the memory environment.

2.1.1 Tools

The ARM software development toolkit consists of the following core command-line tools:

`armcc` The ARM C cross compiler. This is a mature, industrial-strength compiler, tested against Plum Hall C Validation Suite for ANSI conformance. It supports both Unix and PCC compatible modes. It is highly optimising, with options to optimise for code size or execution speed. The compiler is very fast, compiling 500 lines per second on a SPARC 10/41. The compiler can also produce ARM assembly language source.

`tcc` The Thumb C cross compiler. This is based on the ARM C compiler but produces 16-bit Thumb instructions instead of 32-bit ARM instructions.

`armasm` The ARM cross assembler. This compiles ARM assembly language source into ARM object format object code.

`tasm` The Thumb and ARM cross assembler. This compiles both ARM assembly and Thumb assembly language source into object code. An assembler directive dictates whether the code following is ARM (32-bits) or Thumb (16-bits).

`armlink` The Thumb and ARM linker. This combines the contents of one or more object files (the output of a compilation of assembler) with selected parts of one or more object libraries, to produce an executable program.

`decAOF` The Thumb and ARM object file decoder/disassembler. This is used to extract information from object files, such as the code size.

`armsd` The Thumb and ARM symbolic debugger. This is used to emulate ARM processors, allowing ARM and Thumb executable programs to be run on non-ARM hardware. It also allows source level debugging of programs that have been compiled with debug information. This consists of single stepping either C source or assembler source, setting break points/ watchpoints, etc. armsd can also connect to real hardware and allow source level debugging on that hardware.

These tools are documented in *The ARM Software Development Toolkit Reference Manual: Chapter 1, Introduction*.

On the IBM PC platform, the toolkit also comprises:

`APM` The ARM Project Manager. This is an integrated development environment, which provides all the functions of a traditional make file, along with source editing facilities and a link to the ARM debugger.

`Windbg` The ARM windowed debugger. This is the Windows version of armsd which integrates with the ARM Project Manager.

These applications are documented in the ARM Windows Toolkit Guide (ARM DUI 0022).

2.2 The Hello World Example

This example shows you how to write, compile, link and execute a simple C program that prints “Hello World” and a carriage return on the screen. The code will be generated on a text editor, compiled and linked using armcc, and run on armsd.

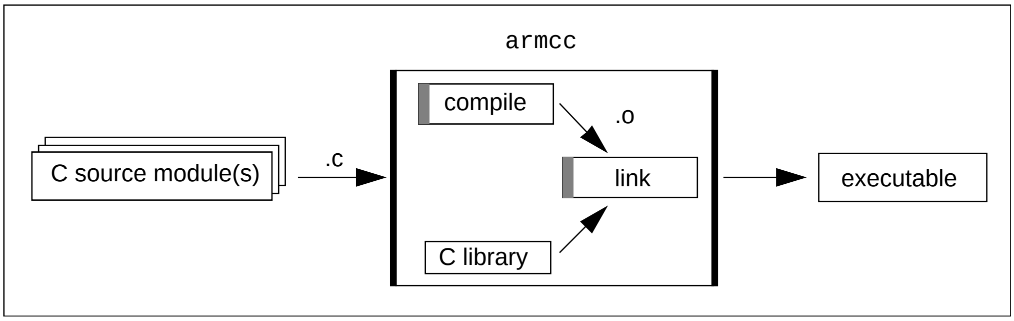


Figure 2-2: Compiling and linking C

2.2.1 Create, compile, link, and run

Generate the following code using any text editor, and save the file as `hello.c`.

```
#include <stdio.h>

int main(void)
{
  printf("Hello World\n");
  return 0;
}
```

Use the following command to compile and link the code:

```
armcc hello.c -o hello
```

The argument to the `-o` flag gives the name of the file which will hold the final output of the link step. The linker is automatically called after compilation (because in this instance the `-c` flag has not been specified). Note that flags are case-sensitive.

To execute the code under software emulation, enter:

```
armsd hello
```

at the system prompt. armsd will start, load in the file, and display the `armsd:` prompt to indicate that it is waiting for a command. Type

```
go
```

and press Return. The debugger should respond with “Hello World”, followed by a message indicating that the program terminated normally.

2.2.2 Timing

To find out how many microseconds this would take to run on real hardware, type the following:

```
print $clock
```

You can change the memory model and clock speed of the hardware being simulated—for more information, see ➲*Chapter 13, Benchmarking, Performance Analysis, and Profiling*.

To load and run the program again, enter:

```
reload
go
```

To quit the debugger, enter:

```
quit
```

2.2.3 Debugging

Next, re-compile the program to include high-level debugging information, and use the debugger to examine the code. Compile the program using:

```
armcc -g hello.c -o hello2
```

where the `-g` option instructs the compiler to add debug information.

Load `hello2` into armsd:

```
armsd hello2
```

and set a breakpoint on the first statement in `main` by entering:

```
break main
```

at the `armsd:` prompt.

To execute the program up to the breakpoint, enter:

```
go
```

The debugger reports that it has stopped at breakpoint #1, and displays the source line. To view the ARM registers, enter:

```
reg
```

To list the C source, enter:

```
type
```

This displays the whole source file. `type` can also display sections of code: for example if you enter:

```
type 1,6
```

lines 1 to 6 of the source will be displayed.

To show the assembly code rather than the C source, type:

```
list
```

This will produce the assembly around the current position in the program. You can also list memory at a given address:

```
list 0x8080
```

2.2.4 Separating the compile and link stages

To separate the compile and link stages, use the `-c` option when running armcc. Quit the debugger and then type the following:

```
armcc -c hello.c
```

This will produce the object file `hello.o`, but no executable. To link the object file with a library, and so generate an executable program, issue the command:

```
armlink hello.o libpath/armlib.32l -o hello3
```

replacing *`libpath`* with the pathname of the toolkit's `lib` directory on your system. The `armlib.32l` file is the version of the library which uses the 32-bit ARM instruction set and runs in a little endian memory model.

Run the program:

```
armsd hello3
```

`hello3` contains no C source because `hello.o` was compiled without the `-g` option, so attempting to view the source statements with the `type` command will fail. However, it is still possible to reference program locations and set breakpoints on them using the `@` character to reference the low-level symbols.

For example, to set a breakpoint on the first location in `main`, type:

```
break @main
```

2.2.5 Generating assembly language from C

The compiler can also generate assembly language from C. Quit the debugger and enter:

```
armcc -S hello.c
```

at the system prompt.

The `-S` flag instructs armcc to write out an assembly language listing of the instructions that would normally be compiled into executable code. By default the output file will have the same name as the C source file, but with the extension `.s`.

To view the assembly language which was output by armcc, display the file `hello.s` on screen using the appropriate operating system command, or load it into a text editor. You should see the following:

```
; generated by Norcroft  ARM C vsn 4.65 (Advanced RISC Machines) [May
23 1995]

        AREA |C$$code|, CODE, READONLY
|x$codeseg| DATA

main
        MOV      ip,sp
        STMDB    sp!,{fp,ip,lr,pc}
        SUB      fp,ip,#4
        CMP      sp,sl
        BLMI     __rt_stkovf_split_small
        ADD      a1,pc,#L000024-.-8
        BL       _printf
        MOV      a1,#0
        LDMDB    fp,{fp,sp,pc}
L000024
        DCB     0x48,0x65,0x6c,0x6c
        DCB     0x6f,0x20,0x77,0x6f
        DCB     0x72,0x6c,0x64,0x0a
        DCB     00,00,00,00

        AREA |C$$data|,DATA

|x$dataseg|

        EXPORT main

        IMPORT _printf
        IMPORT __rt_stkovf_split_small

        END
```

Note *Your code may differ slightly from the above, depending on the version of armcc in use.*

2.2.6 For more information

For a description of the ARM C compiler options, see ➲*The ARM Software Development Toolkit Reference Manual: Chapter 2, C Compiler.*

For a description of the ARM linker options, see ➲*The ARM Software Development Toolkit Reference Manual: Chapter 6, Linker.*

3 Programmer’s Model

This chapter describes the features of the ARM Processor which are of special interest to the programmer.

Programmer's Model

3.1 Introduction

This chapter gives an overview of the ARM from the programmer's point of view, and is designed to provide you with some general background for the discussions in this book.

3.1.1 The ARM Architecture—a brief overview

ARM architecture has evolved considerably since its first development. There are four major versions:

Architectures 1 and 2

The original architecture—Version 1—was implemented only by ARM1, and was never used in a commercial product.

ARM Architecture Version 2 was the first to be used commerially. It extended Version 1 by adding:

- the multiply and multiply accumulate instructions (`MUL` and `MLA`)
- support for coprocessors
- a further two banked registers for FIQ mode

Version 2a introduced an Atomic Load and Store instruction (`SWP`) and the use of Coprocessor 15 as a system control coprocessor. Versions 1, 2 and 2a all supported a 26-bit address bus and combined in register 15 a 24-bit Program Counter (PC) and 8 bits of processor status.

Architecture 3

Version 3 of the architecture extended the addressing range to 32 bits, defining a 30-bit Program Counter value in register 15. The status information was moved from register 15 to a new 11-bit status register (the *Current Program Status Register* or CPSR). Version 3 also added two new privileged processing modes (Version 2 has just three: Supervisor, IRQ and FIQ). The new modes, Undefined and Abort, allowed coprocessor emulation and virtual memory support in Supervisor mode. In addition, a further five status registers (the *Saved Program Status Registers* or SPSRs) were defined, one for each privileged processor mode, in which the CPSR contents is preserved when the corresponding exception is taken.

A variant of the Version 3 architecture—Version 3M—added multiply and multiply accumulate instructions that produce a 64 bit result (`SMULL`, `UMULL`, `SMLAL`, `UMLAL`).

Architecture 4

Version 4 added halfword load and store instructions and sign extended byte and halfword load instructions. It also reserved some SWI instruction space for architecturally defined operations, added a new privileged processor mode called *System* (that uses the User mode registers) and defined several new undefined instructions.

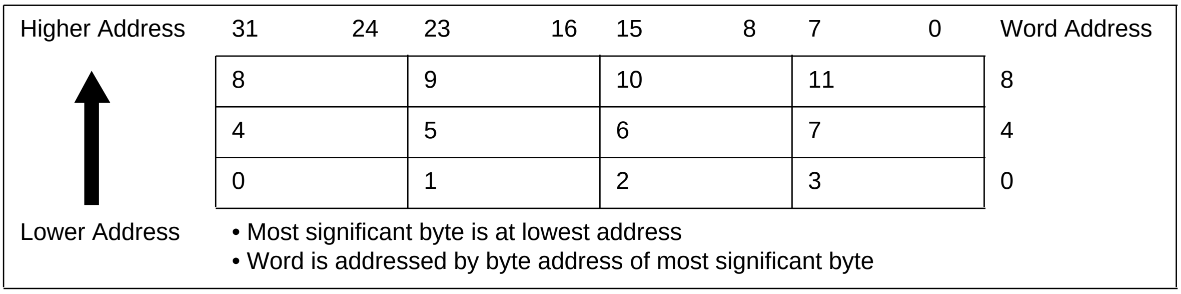
A variant of Version 4 called 4T incorporates an instruction decoder for a 16-bit subset of the ARM instruction set (known as Thumb). Processors which have this decoder are referred to as being *Thumb-aware*.

3.2 Memory Formats

The ARM views memory as a linear collection of bytes numbered upwards from zero. Bytes 0 to 3 hold the first stored word, bytes 4 to 7 the second and so on. The ARM can treat words in memory as being stored either in *Big Endian* or *Little Endian* format.

3.2.1 Big endian format

In big endian format, the most significant byte of a word is stored at the lowest numbered byte and the least significant byte at the highest numbered byte. Byte 0 of the memory system is therefore connected to data lines 31 through 24.



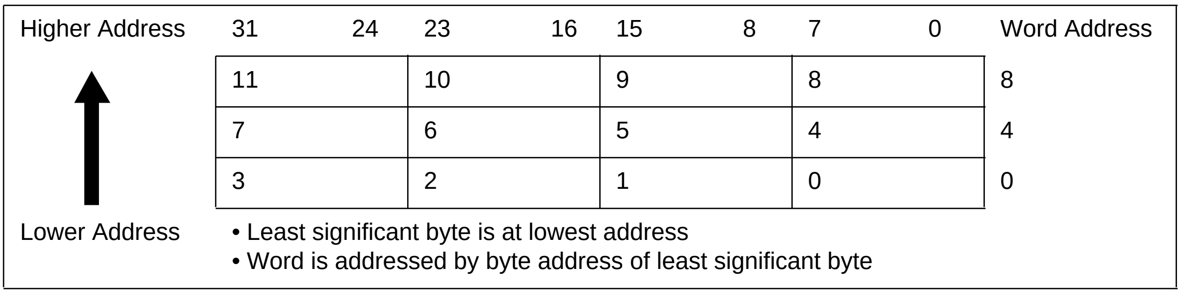
Higher Address	31 24	23 16	15 8	7 0	Word Address
	8	9	10	11	8
	4	5	6	7	4
	0	1	2	3	0
Lower Address					

- Most significant byte is at lowest address
- Word is addressed by byte address of most significant byte

Figure 3-1: Big endian addresses of bytes within words

3.2.2 Little endian format

In little endian format, the lowest numbered byte in a word is considered the word's least significant byte, and the highest numbered byte the most significant. Byte 0 of the memory system is therefore connected to data lines 7 through 0.



Higher Address	31 24	23 16	15 8	7 0	Word Address
	11	10	9	8	8
	7	6	5	4	4
	3	2	1	0	0
Lower Address					

- Least significant byte is at lowest address
- Word is addressed by byte address of least significant byte

Figure 3-2: Little endian addresses of bytes within words

3.3 Instruction Length

ARM instructions are exactly one word (32 bits), and are aligned on a four-byte boundary. Thumb instructions are exactly one halfword, and are aligned on a two-byte boundary.

3.4 Data Types

The ARM supports the following data types:

Byte	8 bits
Halfword	16 bits halfwords must be aligned to 2-byte boundaries (Architecture 4 only)
Word	32 bits words must be aligned to four-byte boundaries

Load and store operations can transfer bytes, halfwords and words to and from memory.

Signed operands are in two's complement format.

3.5 Processor Modes

There are a number of different processor modes. These are shown in the following table:

Processor mode			Description
1	User	(usr)	the normal program execution mode
2	FIQ	(fiq)	designed to support a high-speed data transfer or channel process
3	IRQ	(irq)	used for general-purpose interrupt handling
4	Supervisor	(svc)	a protected mode for the operating system
5	Abort	(abt)	used to implement virtual memory and/or memory protection
6	Undefined	(und)	used to support software emulation of hardware coprocessors
7	System	(sys)	used to run privileged operating system tasks (Architecture Version 4 only)

Table 3-1: ARM processor modes

Mode changes may be made under software control or may be caused by external interrupts or exception processing. Most application programs will execute in User mode. The other modes, known as *privileged* modes, will be entered to service interrupts or exceptions or to access protected resources: see ➲*3.10 Exceptions* on page 3-12.

3.6 Processor States

Note *This section applies to Architecture 4T only.*

Thumb-aware processors can be in one of two *processor states*:

ARM state — which executes 32-bit word-aligned ARM instructions

Thumb state — which executes 16-bit halfword-aligned Thumb instructions. In this state, the PC uses bit 1 to select between alternate halfwords.

3.6.1 Switching state

Entering Thumb state

Entry into Thumb state occurs on execution of a BX instruction with the state bit (bit 0) set in the operand register.

Transition to Thumb state also occurs automatically on return from an exception (IRQ, FIQ, RESET, UNDEF, ABORT, SWI etc) if the exception was entered from Thumb state.

Entering ARM state

Entry into ARM state happens:

1. On execution of the BX instruction with the state bit clear in the operand register.
2. On the processor taking an exception (IRQ, FIQ, RESET, UNDEF, ABORT, SWI etc.).

   In this case, the PC is placed in the exception mode’s link register, and execution commences at the exception’s vector address. See ➲*3.10 Exceptions* on page 3-12 and ➲*Chapter 11, Exceptions*.

3.7 The ARM Register Set

The ARM processor has a total of 37 registers, comprising:

- 30 general-purpose registers
- 6 status registers
- a program counter

However, not all of these registers can be seen at once. Depending on the processor mode, fifteen general-purpose registers (R0 to R14), one or two status registers and the program counter will be visible. The registers are arranged in partially overlapping banks with a different register bank for each processor mode: ➲*Table 3-2: The ARM register set* on page 3-7 shows how the registers are arranged, with the banked registers shaded. ➲*Table 3-4: The mode bits* on page 3-11 lists which registers are visible in which mode.

3.7.1 Register roles

Registers 0-12 are always free for general-purpose use. Registers 13 and 14, although available for general use, also have specific roles:

Register 13 (also known as the *Stack Pointer* or SP) is banked across all modes to provide a private Stack Pointer for each mode (except System mode which shares the user mode R13).

Register 14 (also known as the *Link Register* or LR) is used as the subroutine return address link register. R14 is also banked across all modes (except System mode which shares the user mode R14).

When a Subroutine call (Branch and Link instruction) is executed, R14 is set to the subroutine return address. The banked registers R14_SVC, R14_IRQ, R14_FIQ, R14_ABORT and R14_UNDEF are used similarly to hold the return address when exceptions occur (or a subroutine return address if subroutine calls are executed within interrupt or exception routines). R14 may be treated as a general-purpose register at all other times.

Register 15 is used specifically to hold the *Program Counter* (PC). When R15 is read, bits [1:0] are zero and bits [31:2] contain the PC. When R15 is written bits[1:0] are ignored and bits[31:2] are written to the PC. Depending on how it is used, the value of the PC is either the address of the instruction plus *n* (where *n* is 8 for ARM state and 4 for Thumb state) or is unpredictable.

CPSR is the Current Program Status Register. This is accessible in all processor modes, and contains the condition code flags, interrupt enable flags, and current processor mode. In Architecture 4T, the CPSR also holds the processor state. See ➲*3.9 Program Status Registers* on page 3-10 for more information.

3.7.2 The FIQ banked registers

FIQ mode has banked registers R8 to R12 (as well as R13 and R14). Regusters R8_FIQ, R9_FIQ, R10_FIQ, R11_FIQ and R12_FIQ are provided to allow very fast interrupt processing (without the need to preserve register contents by storing them to memory), and to preserve values across interrupt calls (so that register contents do not need to be restored from memory).

User/ System	Supervi-sor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8_FIQ
R9	R9	R9	R9	R9	R9_FIQ
R10	R10	R10	R10	R10	R10_FIQ
R11	R11	R11	R11	R11	R11_FIQ
R12	R12	R12	R12	R12	R12_FIQ
R13	R13_SVC	R13_ABORT	R13_UNDEF	R13_IRQ	R13_FIQ
R14	R14_SVC	R14_ABORT	R14_UNDEF	R14_IRQ	R14_FIQ
PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_SVC	SPSR_ABORT	SPSR_UNDEF	SPSR_IRQ	SPSR_FIQ

Table 3-2: The ARM register set

3.8 The Thumb Register Set

Note *This section applies to Architecture 4T only.*

The Thumb state register set is a subset of the ARM state set. The programmer has direct access to eight general registers, R0-R7, as well as the Program Counter (PC), a stack pointer register (SP), a link register (LR), and the CPSR. There are banked Stack Pointers, Link Registers and Saved Process Status Registers (SPSRs) for each privileged mode. This is shown in *Table 3-3: The Thumb register set*.

User/ System	Supervi-sor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
SP	SP_SVC	SP_ABORT	SP_UNDEF	SP_IRQ	SP_FIQ
LR	LR_SVC	LR_ABORT	LR_UNDEF	LR_IRQ	LR_FIQ
PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_SVC	SPSR_ABORT	SPSR_UNDEF	SPSR_IRQ	SPSR_FIQ

Table 3-3: The Thumb register set

The Thumb state registers relate to the ARM state registers in the following way:

- Thumb state R0-R7 and ARM state R0-R7 are identical
- Thumb state CPSR and SPSRs and ARM state CPSR and SPSRs are identical
- Thumb state SP maps onto ARM state R13
- Thumb state LR maps onto ARM state R14
- The Thumb state Program Counter maps onto the ARM state Program Counter (R15)

This relationship is shown in *Figure 3-3: Mapping of Thumb state registers onto ARM state registers*.

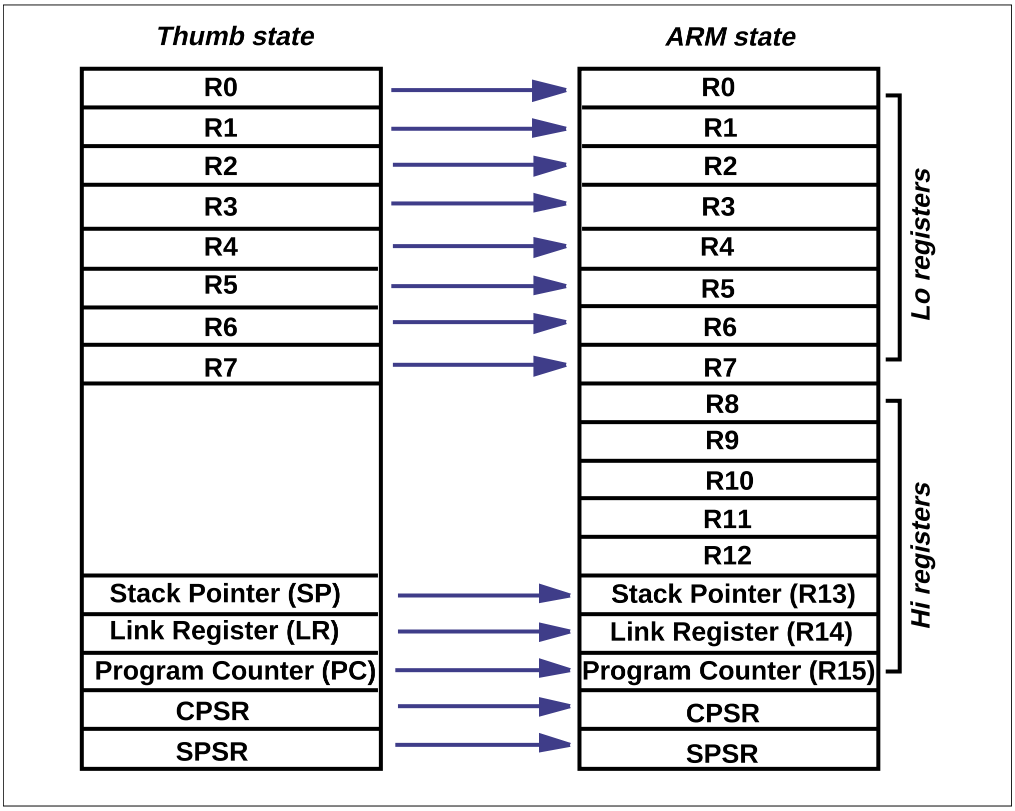


Figure 3-3: Mapping of Thumb state registers onto ARM state registers

3.8.1 Accessing Hi registers in Thumb state

In Thumb state, registers R8-R15 (the *Hi registers*) are not part of the standard register set. However, the assembly language programmer has limited access to them, and can use them for fast temporary storage.

A value may be transferred from a register in the range R0-R7 (a *Lo register*) to a Hi register, and from a Hi register to a Lo register, using special variants of the `MOV` instruction. Hi register values can also be compared against or added to Lo register values with the `CMP` and `ADD` instructions. For more details on the use of Thumb instructions, see *The ARM Software Development Toolkit Reference Manual: Chapter 5, Thumb Instruction Set*.

3.9 Program Status Registers

The ARM contains a *Current Program Status Register* (CPSR), plus five *Saved Program Status Registers* (SPSRs) for use by exception handlers. The CPSR:

- holds information about the most recently performed ALU operation
- controls the enabling and disabling of interrupts
- sets the processor operating mode
- sets the processor state (Architecture 4T only)

The CPSR is saved to the appropriate SPSR when the processor enters an exception.

The arrangement of bits in these registers is shown in ➲*Figure 3-4: Program Status Register format*, below.

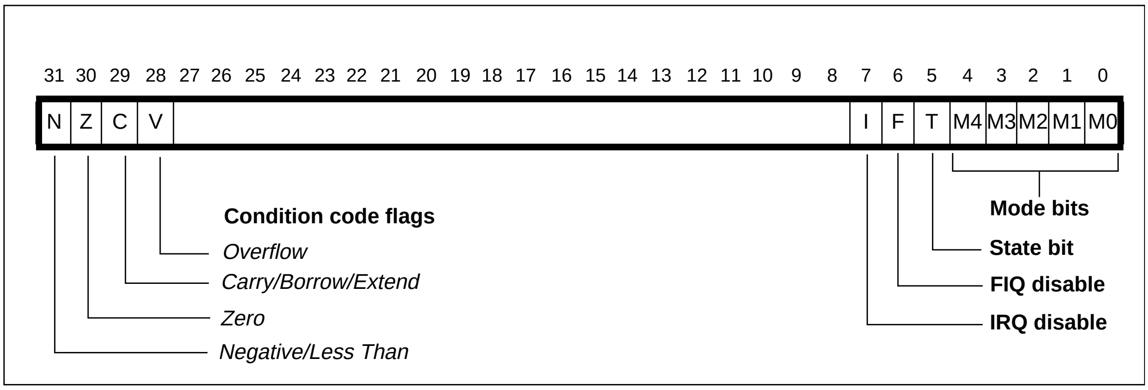


Figure 3-4: Program Status Register format

3.9.1 The condition code flags

The N, Z, C and V (Negative, Zero, Carry and oVerflow) bits are collectively known as the *condition code flags*. The condition code flags in the CPSR can be changed as a result of arithmetic and logical operations in the processor, and can be tested by all ARM instructions to determine if the instruction is to be executed. All ARM instructions may be executed conditionally in this way. Thumb instructions (only available in Architecture 4T) cannot be executed conditionally, with the exception of the Branch instruction—see ➲*The ARM Software Development Toolkit Reference Manual: Chapter 5, Thumb Instruction Set*.

3.9.2 The control bits

The bottom 8 bits of a PSR (incorporating I, F, T and M[4:0]) are known collectively as the *control bits.* These change when an exception arises, and can be altered by software only when the processor is in a privileged mode.

Interrupt disable bits — The I and F bits are the *interrupt disable bits*. When set, these disable the IRQ and FIQ interrupts respectively.

The state bit — Bit T is the processor *state bit*. When the state bit is set to 0, this indicates that the processor is in ARM state (ie. executing 32-bit ARM instructions). When it is set to 1, this indicates that the processor is in Thumb state (executing 16-bit Thumb instructions)

The state bit is only implemented on Thumb-aware processors (Architecture 4T). On non Thumb-aware processors the state bit will always be zero.

The mode bits — The M4, M3, M2, M1 and M0 bits (M[4:0]) are the *mode bits*. These determine the mode in which the processor operates, as shown in ➲*Table 3-4: The mode bits*, below. Not all combinations of the mode bits define a valid processor mode. Only those explicitly described can be used.

..

M[4:0]	Mode	Accessible Registers
10000	User	PC, R14 to R0, CPSR
10001	FIQ	PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq
10010	IRQ	PC, R14_irq, R13_irq,R12 to R0, CPSR, SPSR_irq
10011	SVC	PC, R14_svc, R13_svc,R12 to R0, CPSR, SPSR_svc
10111	Abort	PC, R14_abt, R13_abt,R12 to R0, CPSR, SPSR_abt
11011	Undef	PC, R14_und, R13_und,R12 to R0, CPSR, SPSR_und
11111	System	PC, R14 to R0, CPSR (Architecture 4 only)

Table 3-4: The mode bits

User mode and System mode do not have an SPSR, since they are not entered on any exception and therefore do not need a register in which to preserve the CPSR. In User mode or System mode, reads from the SPSR return an unpredictable value, and writes to the SPSR are ignored.

3.10 Exceptions

Note This section is a brief overview of the ARM's exceptions. For a detailed explanation of how they operate and how to handle them please refer to *Chapter 11, Exceptions*.

3.10.1 Exception types

Exceptions are generated by internal and external sources to divert the processor to handle an event—for example an externally generated interrupt or an attempt to execute an undefined instruction. The processor's internal state just prior to handling the exception must be preserved so that the original program can be resumed when the exception routine has completed. More than one exception may arise at the same time.

The ARM processor supports seven types of exception, and has a privileged processor mode for each type. *Table 3-5: Exception processing modes* lists each type of exception and the processor mode used to process that exception. When an exception occurs, execution is forced from a fixed memory address corresponding to the type of exception. ARM collectively names these fixed addresses the *hard vectors.*

Exception type	Exception mode	Vector address
Reset	Supervisor	0x00000000
Undefined instructions	Undefined	0x00000004
Software Interrupt (SWI)	Supervisor	0x00000008
Prefetch Abort (Instruction fetch memory abort)	Abort	0x0000000c
Data Abort (Data Access memory abort)	Abort	0x00000010
IRQ (Interrupt)	IRQ	0x00000018
FIQ (Fast Interrupt)	FIQ	0x0000001c

Table 3-5: Exception processing modes

Address 0x14 (omitted from the above table) holds the Address Exception vector, which is only used when the processor is configured for a 26-bit address space.

3.10.2 Action on entering an exception

When an exception occurs, the ARM makes use of the banked registers to save state, by:

1. copying the address of the next instruction into the appropriate Link Register
2. copying the CPSR into the appropriate SPSR
3. forcing the CPSR mode bits to a value corresponding to the exception
4. forcing the PC to fetch the next instruction from the relevant vector

It may also set the interrupt disable flags to prevent otherwise unmanageable nestings of exceptions from taking place.

If the processor is Thumb-aware (Architecture 4T) and is operating in Thumb state, it will automatically switch into ARM state.

3.10.3 Action on leaving an exception

On completion, the exception handler:

1. moves the Link Register, minus an offset where appropriate, to the PC. The offset will vary depending on the exception type.
2. copies the SPSR back to the CPSR
3. clears the interrupt disable flags, if they were set on entry

If the processor is Thumb-aware (Architecture 4T), it will restore the operating state (ARM or Thumb) which was in force at the time the exception occurred.

4

ARM Assembly Language Basics

This chapter explains the concepts behind, and the basics of programming in, ARM assembly language.

ARM Assembly Language Basics

4.1 Introduction

The ARM instruction set has the following key features, some of which are common to other processors, and some of which are not.

Load/store architecture

Only load and store instructions can access memory. This means that data processing operations have to use intermediate registers, loading the data from memory beforehand and storing it back again afterwards. However, this is not as inefficient as one might think. Most operations actually require several instructions to carry out the required calculation, and each instruction will run as fast as possible instead of being slowed down by external memory accesses.

32-bit instructions

All instructions are of the same length, so the processor can fetch every instruction from memory in one cycle. In addition, all instructions are stored word-aligned in memory, which means that the bottom two bits of the program counter (r15) are always set zero.

32-bit and 8-bit data

All ARM processors have load and store instructions that handle data as 32-bit words or 8-bit bytes. Words are always aligned on 4-byte boundaries.

Processors implementing Version 4 of the ARM Architecture also have instructions for loading and storing halfwords (16-bit values).

32-bit addresses

Processors implementing Versions 1 and 2 of the ARM Architecture only had a 26-bit addressing range. All later ARM processors have a 32-bit addressing. Those implementing ARM Architectures 3 and 4 (but *not* 4T) have retained the ability to perform 26-bit addressing backwards compatibility.

37 registers

These comprise:

- 30 general purpose registers, 15 of which are accessible at any one time
- 6 status registers, of which either one or two are accessible at any one time
- a program counter

The banking of registers gives rapid context switching for dealing with exceptions and privileged operations: see ➲*Chapter 3, Programmer’s Model* for a summary of the ARM register set.

Flexible load and store multiple instructions

The ARM’s multiple load and store instructions allow any set of registers from a single bank to be transferred to and from memory by a single instruction.

No single instruction to move an immediate 32-bit value to a register

In general, a literal value must be loaded from memory. However, a large set of common 32-bit values *can* be generated in a single instruction.

Conditional execution

All instructions are executed conditionally on the state of the Current Program Status Register (CPSR). Only data processing operations with the S bit set change the state of the current program status register.

Powerful barrel shifter

The second argument to all data-processing and single data-transfer operations can be shifted in quite a general way before the operation is performed. This supports— but is not limited to— scaled addressing, multiplication by a small constant, and the construction of constants, within a single instruction.

Co-processor instructions

These support a general way to extend the ARM's architecture in a customer-specific manner.

4.2 Structure of an Assembler Module

The assembler is described fully in ➲*The ARM Software Development Toolkit Reference Manual: Chapter 3, Assembler*. This section describes it in a simplified manner which gives you the basics required for writing simple assembler programs. ARM Instructions are described in ➲*The ARM Software Development Toolkit Reference Manual: Chapter 4, ARM Instruction Set*.

The following is a simple example which illustrates some of the core constituents of an ARM assembler module:

```
        AREA Example, CODE, READONLY      ; name this block of code
        ENTRY                             ; mark first instruction
                                          ; to execute
start
        MOV    r0, #15                    ; Set up parameters
        MOV    r1, #20
        BL     firstfunc                  ; Call subroutine
        SWI    0x11                       ; terminate
firstfunc                                 ; Subroutine firstfunc
        ADD    r0, r0, r1                 ; r0 = r0 + r1
        MOV    pc, lr                     ; Return from subroutine
                                          ; with result in r0
        END                               ; mark end of file
```

4.2.1 The AREA directive

Areas are chunks of data or code that are manipulated by the linker. A complete application will consist of one or more areas. The example above consists of a single area which contains code and is marked as being read-only. A single CODE area is the minimum required to produce an application.

4.2.2 The ENTRY directive

The first instruction to be executed within an application is marked by the ENTRY directive. An application can contain only a single entry point and so in a multi-source-module application, only a single module will contain an ENTRY directive. Note that when an application contains C code, the entry point will usually be contained within the C library.

4.2.3 General layout

The general form of lines in an assembler module is:

```
        label <whitespace> instruction <whitespace> ;comment
```

The important thing to note is that the three sections are separated by at least one whitespace character (such as a space or a tab). Actual instructions never start in the first column, since they must be preceded by whitespace, even if there is no label. All three sections are optional and the assembler will also accept blank lines to improve the clarity of the code.

4.2.4 Description of the module

The main routine of the program (labelled `start`) loads the values 15 and 20 into registers 0 and 1. This is done because up to four word-length parameters can be passed to a subroutine in registers r0 – r3. See *The ARM Software Development Toolkit Reference Manual: Chapter 19, ARM Procedure Call Standard* for more information on this.

The program then calls the subroutine `firstfunc` by using a branch with link instruction (`BL`). This adds the offset required to reach the subroutine to the program counter, `pc` (r15). It then stores in the link register, `lr` (r14), the address of the next instruction to be executed in the main routine.

The subroutine adds together the two parameters it has received and places the result back into r0, as required by the APCS. It then returns by simply restoring the program counter to the address which was stored in the link register on entry.

Upon return from the subroutine, the main program simply terminates using software interrupt 11. This instructs the program to exit cleanly and return control to the debugger.

4.2.5 Running the example

This module is available as `example.s` in directory `examples/basicasm`. To assemble this, first copy it to the current work directory and then issue the command:

```
armasm example.s
```

The object code can then be linked to produce an executable:

```
armlink example.o -o example
```

This can then be loaded into armsd and executed:

```
armsd example
go
```

Once the program has completed, you can check that it executed correctly by examining the value returned in r0 by `firstfunc`*:*

```
registers
```

Register r0 should be 0x23.

You can then exit from armsd by typing:

```
quit
```

4.3 Conditional Execution

4.3.1 The ARM’s ALU status flags

The ARM’s Program Status Register contains, among other flags, copies of the ALU status flags:

N	Negative result from ALU flag
Z	Zero result from ALU flag
C	ALU operation Carried out
V	ALU operation oVerflowed

See *Figure 3-4: Program Status Register format* on page 3-10 for details.

Data processing instructions change the state of the ALU’s N, Z, C and V status outputs, but these are latched in the PSR’s ALU flags only if a special bit (the S bit) is set in the instruction.

4.3.2 Execution conditions

Every ARM instruction has a 4-bit field that encodes the conditions under which it will be executed. These conditions refer to the state of the ALU N, Z, C and V flags as shown in the *Table 4-1: Condition codes* on page 4-7.

If the condition field indicates that a particular instruction should not be executed given the current settings of the status flag, the instruction will simply soak up one cycle but will have no other effect.

If the current instruction is a data processing instruction, and the flags are to be updated by it, the instruction must be postfixed by an S. The exceptions to this are `CMP`, `CMN`, `TST` and `TEQ`, which always update the flags (since this is their only effect).

Examples

```
ADD    r0, r1, r2    ; ro = 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
```

Field mnemonic	Condition
EQ	Z set (equal)
NE	Z clear (not equal)
CS/HS	C set (unsigned >=)
CC/LO	C clear (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 (unsigned >)
LS	C clear and Z set (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 <=)
AL	Always execute (the default if none is specified)

Table 4-1: Condition codes

4.3.3 Using conditional execution

Most non-ARM processors only allow conditional execution of branch instructions. This means that small sections of code that should only be executed under certain conditions will need to be avoided by use of a branch statement. Consider Euclid’s Greatest Common Divisor algorithm:

```
function gcd (integer a, integer b) : result is integer
while (a<> b) do
       if (a > b) then
              a = a - b
       else
              b = b - a
       endif
endwhile
result = a
```

This might be coded as:

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

This will work correctly on an ARM, but every time a branch is taken, three cycles will be wasted in refilling the pipeline and continuing execution from the new location. Also, because of the number of branches in the code, the code will occupy seven words of memory. Using conditional execution, ARM code can improve both its execution time and code density:

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

Not only has code size been reduced from seven words to four, but execution time has also decreased, as can be seen from *Table 4-2: Only branches conditional* and *Table 4-3: All instructions conditional* on page 4-9. These show the execution times for the simple case where r0 equals 1 and r1 equals 2. In this case, replacing branches with conditional execution of all instructions has given a saving of three cycles. With all inputs to the gcd algorithm, the conditional version of the code will execute in the same number of cycles (when both inputs are the same), or fewer cycles.

4.3.4 Running the gcd examples

Both assembler versions of the gcd algorithm can be found in directory `examples/basicasm`. To assemble them, first copy them to your current work directory and then issue the commands:

```
armasm gcd1.s
armlink gcd1.o -o gcd1
```

This produces the version with conditional execution of branch statements only. To produce the version with full conditional execution of all instructions use:

```
armasm gcd2.s
armlink gcd2.o -o gcd2
```

Run these using the debugger and examine the difference in the way they execute.

r0:a	r1: b	Instruction	Cycles
1	2	`CMP r0, r1`	1
1	2	`BEQ end`	Not executed - 1
1	2	`BLT less`	3
1	2	`SUB r1, r1, r0`	1
1	1	`BAL gcd`	3
1	1	`CMP r0, r1`	1
1	1	`BEQ end`	3
			Total = 13

Table 4-2: Only branches conditional

r0:a	r1: b	Instruction	Cycles
1	2	`CMP r0, r1`	1
1	2	`SUBGT r0, r0, r1`	Not executed -1
1	1	`SUBLT r1, r1, r0`	1
1	1	`BNE gcd`	3
1	1	`CMP r0, r1`	1
1	1	`SUBGT r0, r0, r1`	Not executed -1
1	1	`SUBLT r1, r1, r0`	Not executed -1
1	1	`BNE gcd`	Not executed -1
			Total = 10

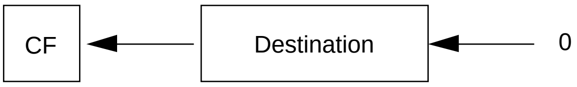
Table 4-3: All instructions conditional

4.4 The ARM’s Barrel Shifter

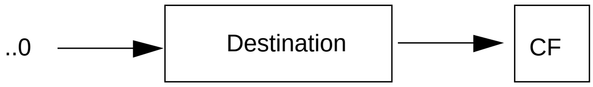
The ARM core contains a barrel shifter which takes a value to be shifted or rotated, an amount to shift or rotate by and the type of shift or rotate. This can be used by various classes of ARM instructions to perform comparatively complex operations in a single instruction. Instructions take no longer to execute by making use of the barrel shifter, unless the amount to be shifted is specified by a register, in which case the instruction will take an extra cycle to complete.

The barrel shifter can perform the following types of operation:

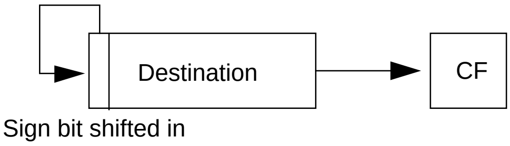
`LSL` shift left by n bits (multiplication by 2^n)



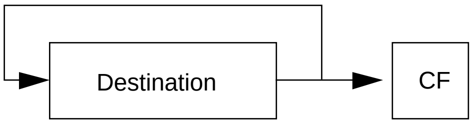
`LSR` logical shift right by n bits (unsigned division by 2^n)



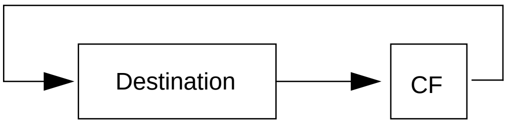
`ASR` arithmetic shift right by n bits. The bits fed into the top end of the operand are copies of the original top—or sign—bit. (signed division by 2^n)



`ROR` rotate right by n bits



`RRX`	rotate right extended by 1 bit. This is a 33-bit rotate, where the 33rd bit is the PSR Carry flag



The barrel shifter can be used in several of the ARM's instruction classes. The options available in each case are described below.

4.4.1 Data processing operations

The last operand (the second for binary operations, and the first for unary operations) may be:

an 8 bit constant rotated right (ROR) through an even number of positions

For example:

```
ADD    r0, r1, #0xC5, 10
MOV    r5, #0xFC000003
```

Note that in the second example the assembler is left to work out how to split the constant 0xFC000003 into an 8-bit constant and an even rotate (in this case `#0xFC000003` could be replaced by `#0xFF, 6`). For more information, see ➲*4.5 Loading Constants Into Registers* on page 4-14.

a register (optionally) shifted or rotated either by a 5-bit constant or by another register

For example:

```
ADD r0, r1, r2
SUB r0, r1, r2, LSR #10
CMP r1, r2, r1, ROR R5
MVN r3, r2, RRX
```

Note that in the last example, the rotate right extended does not take a parameter, but rather rotates right by only a single bit. `RRX` is actually encoded by the assembler as `ROR #0`.

Example: Constant multiplication

The ARM core provides a powerful multiplication facility in the `MUL` and `MLA` instructions (plus `UMULL`, `UMAL`, `SMULL` and `SMLAL` on processors that implement ARM Architectures 3M and 4M). These instructions make use of Booth's Algorithm to perform integer multiplication, taking up to 17 cycles to complete for `MUL` and `MLA` and up to 6 or 7 cycles to complete for `UMULL`, `UMAL`, `SMULL` and `SMLAL`. In cases where the multiplication is by a constant, it can be quicker to make use of the barrel shifter, as the operations it provides are effectively multiply / divide by powers of two.

For example:

r0 = r1 * 4	`MOV r0, r1, LSL #2`
r0 = r1 * 5 => r0 = r1 + (r1 * 4)	`ADD r0, r1, r1, LSL #2`
r0 = r1 * 7 => r0 = (r1 * 8) - r1	`RSB r0, r1, r1, LSL #3`

Using a move/add/subtract combined with a shift, all multiplications by a constant which are a power of two or a power of two +/– 1 can be carried out in a single cycle. See ➲*Chapter 5, Exploring ARM Assembly Language*.

4.4.2 Single data transfer instructions

The single data transfer instructions LDR and STR load and store the contents of a single register to and from memory. They make use of a base register (r0 in the examples below) plus an index (or offset) which can be a register shifted by any 5-bit constant or an unshifted 12-bit constant.

```
STR    r7, [r0], #24              ; Post-indexed
LDR    r2, [r0], r4, ASR #4       ; Post-indexed
STR    r3, [r0, r5, LSL #3]       ; Pre-indexed
LDR    r6, [r0, r1, ROR #6]!      ; Pre-indexed + Writeback
```

In *pre-indexed* instructions, the offset is calculated and added to the base, and the resulting address is used for the transfer. If writeback is selected, the transfer address is written back into the base register.

In *post-indexed* instructions the offset is calculated and added to the base after the transfer. The base register is always updated by post-indexed instructions.

Example: Addressing an entry in a table of words

The following fragment of code calculates the address of an entry in a table of words and then loads the desired word:

```
       ; r0 holds the entry number [0,1,2,...]
       LDR  r1, =StartOfTable
       MOV  r3, #4
       MLA  r1, r0, r3, r1
       LDR  r2, [r1]
       ...
StartOfTable
       DCD <table data>
```

It first loads the start address of the table, then moves the immediate constant 4 into a register, using the multiply and accumulate instruction to calculate the address, and finally loads the entry.

However, this operation can be performed more efficiently with the barrel shifter, as follows:

```
       ; r0 holds the entry number [0,1,2,...]
       LDR  r1, =StartOfTable
       LDR  r2, [r1, r0, LSL #2]
       ...
StartOfTable
       DCD <table data>
```

Here, the barrel shifter shifts r0 left 2 bits (so multiplying it by 4). This intermediate value is then used as the index for the LDR instruction. Thus a single instruction is used to perform the whole operation. Such significant savings can frequently be made by utilising the barrel shifter.

4.4.3 Program status register transfer instructions

It is possible to modify the N, Z, C and V flags of the PSRs by use of an MSR instruction of the form:

```
       MSR    cpsr_flg, #expression; OR spsr_flg in privileged mode
```

The assembler will attempt to generate a shifted 8-bit value to match the expression, the top four bits of which can be loaded into the top four bits of the PSR. This will not disturb the control bits. The flag bits are the only part of the CPSR which can be modified while in User mode (when no SPSRs are visible).

4.5 Loading Constants Into Registers

4.5.1 Why Is loading constants an issue?

Since all ARM instructions are precisely 32 bits long and since they do not use the instruction stream as data, there is no single instruction which will load a 32-bit immediate constant into a register without performing a data load from memory.

Although a data load will place any 32-bit value in a register, there are more direct—and therefore more efficient—ways to load many commonly used constants.

4.5.2 Direct loading with MOV/MVN

The `MOV` instruction allows 8-bit constant values to be loaded directly into a register, giving a range of 0x0 to 0xFF (255). The bitwise complement of these values can be constructed using `MVN`, giving the added ability to load values in the range 0xFFFFFF00 to 0xFFFFFFFF.

We can construct even more constants by using `MOV` and `MVN` in conjunction with the barrel shifter. These particular constants are 8-bit values rotated right through an even number of positions (giving rotate rights of 0, 2, 4...28, 30):

0 - 255	0 - 0xFF with no rotate
256, 260, 264, ..., 1016, 1020	0x100 - 0x3FC in steps of 4 by rotating right by 30 bits
1024, 1040, 1056, ..., 4080	0x400 - 0xFF0 in steps of 16 by rotating right by 28 bits
4096, 4160, 4224, ..., 16320	0x1000 - 0x3FC0 in steps of 64 by rotating right by 26 bits

and so on, plus their bitwise complements. We can therefore load constants directly into registers using instructions such as:

```
MOV    r0, #0xFF                 ; r0 = 255
MOV    r0, #0x1,30               ; r0 = 1020
MOV    r0, #0xFF, 28             ; r0 = 4080
MOV    r0, #0x1, 26              ; r0 = 4096
```

However, converting a constant into this form is an onerous task. The assembler therefore attempts the conversion itself. If the supplied constant cannot be expressed as a shifted 8-bit value or its bitwise complement, the assembler will report this as an error.

The following example illustrates how this works. The left-hand column lists the ARM instructions entered by the user, while the right-hand column shows the assembler's attempts to convert the supplied constants to an acceptable form.

```
MOV r0, #0         ;   =>     MOV r0, #0
MOV r1, #0xFF000000;   =>     MOV r1, #0xFF, 8
MOV r2, #0xFFFFFFFF;   =>     MVN r2, #0
MVN r0, #1         ;   =>     MVN r0, #1
MOV r1, #0xFC000003;   =>     MOV r1, #0xFF, 6
MOV r2, #0x03FFFFFC;   =>     MVN r2, #0xFF, 6
MOV r3, #0x55555555;   =>     Error (cannot be constructed)
```

The above code is available as `loadcon1.s` in directory `examples/basicasm`. To assemble it, first copy it into your current working directory and then issue the command:

```
armasm loadcon1.s -o loadcon1.o
```

To confirm that the assembler has produced the correct code, you can disassemble it using the ARM Object format decoder:

```
decaof -c loadcon1.o
```

4.5.3 Direct loading with LDR Rd, =numeric constant

The assembler provides a mechanism which, unlike `MOV` and `MVN`, can construct any 32-bit numeric constant, but which may not result in a data processing operation to do it. This is the `LDR Rd, =` instruction.

If the constant which is specified in an `LDR Rd, =` instruction can be constructed with either `MOV` or `MVN`, the assembler will use the appropriate instruction, otherwise it will produce an `LDR` instruction with a PC-relative address to read the constant from a literal pool.

Literal pools

A *literal pool* is a portion of memory set aside for constants. By default, a literal pool is placed at every END directive. However, for large programs, this may not be accessible throughout the program (due to the `LDR` offset being a 12-bit value, giving a 4Kbyte range), so further literal pools can be placed using the LTORG directive.

When an `LDR, Rd, =` instruction needs to access a constant in a literal pool, the assembler first checks previously encountered literal pools to see whether the desired constant is already available and addressable. If so, it addresses the existing constant, otherwise it will attempt to place the constant in the next available literal pool. If this is not addressable—because it does not exist or is further than 4Kbytes away—an error will result, and an additional LTORG should be placed close to (but after) the failed `LDR Rd,=` instruction.

To see how this works in practice, consider the following example. The instructions listed as comments are the ARM instructions which are generated by the assembler:

```
       AREA Loadcon2, CODE
       ENTRY                        ; Mark first instruction
       BL     func1                 ; Branch to first subroutine
       BL     func2                 ; Branch to second subroutine
       SWI    0x11                  ; Terminate
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 &55555555
func2
       LDR    r3, =0x55555555       ; => LDR R3, [PC, #offset to
                                    ; Literal Pool 1]
       ; LDR  r4, =0x66666666       ; If this is uncommented it
                                    ; will fail, as Literal Pool 2
                                    ; is not accessible (out of
                                    ; reach)
       MOV    pc, lr
 LargeTable   %   4200              ; Clears a 4200 byte area of
                                    ; memory,
                                    ; starting at the current location,
                                    ; to zero.
       END                          ; Literal Pool 2 is empty
```

Note that the literal pools must be placed outside sections of code, since otherwise they would be executed by the processor as instructions. This will typically mean placing them between subroutines as is done here if more pools than the default one at `END` is required.

The above code is available as `loadcon2.s` in directory `examples/basicasm`. To assemble this, first copy it into your current working directory and then issue the command:

```
armasm loadcon2.s
```

To confirm that the assembler has produced the correct code, the code area can be disassembled using the ARM Object format decoder:

```
decaof -c loadcon2.o
```

4.6 Loading Addresses Into Registers

It will often be necessary to load a register with an address—the location of a string constant within the code segment or the start location of a jump table, for example. However, because ARM code is inherently relocatable and because there are limitations on the values that can be directly moved into a register, absolute addressing cannot be used for this purpose. Instead, addresses must be expressed as offsets from the current PC. A register can either be directly set by combining the current PC with the appropriate offset, or the address can be loaded from a literal pool.

4.6.1 The ADR and ADRL pseudo instructions

Sometimes it is important for the purposes of efficiency that loading an address does not perform a memory access. The assembler provides two pseudo instructions, ADR and ADRL, which make it easier to do this. ADR and ADRL accept a PC-relative expression (a label within the same code area) and calculate the offset required to reach that location.

ADR will attempt to produce a single instruction (either an ADD or a SUB) to load an address into a register in the same way that the `LDR Rd, =` mechanism produces instructions. If the desired address cannot be constructed in a single instruction, an error will be raised. In typical usage 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.

ADRL will attempt to produce two data processing instructions to load an address into a register. Even if it is possible to produce a single data processing instruction to load the address, a second, redundant instruction will be produced (this is a consequence of the strict two-pass nature of the assembler). In cases where it is not possible to construct the address using two data processing instructions ADRL will produce an error, and in such cases the `LDR, =` mechanism is probably the best alternative. In typical usage the range of an ADRL is 64Kbytes for a non-word aligned address and 256Kbytes for a word-aligned address.

The following example shows how this works. The instruction listed in the comment is the ARM instruction which is generated by the assembler.

```
        AREA Loadcon3, CODE
        ENTRY                       ; Mark first instruction
Start
        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 as the offset is
                                    ; cannot be expressed by operand2
                                    ; of an ADD
        ADRL   r3, DataArea+4300    ; => ADD r2, PC, #offset1
                                    ;    ADD r2, r2, #offset2
        SWI    0x11                 ; Terminate
DataArea%      8000

       END
```

The above code is available as `loadcon3.s` in directory `examples/basicasm`. To assemble this, first copy it into your current working directory and then issue the command:

```
armasm loadcon3.s
```

To confirm that the assembler produced the correct code, the code area can be disassembled using the ARM Object format decoder:

```
decaof -c loadcon3.o
```

4.6.2 LDR Rd, =PC-relative expression

As well as numeric constants, the `LDR Rd, =` mechanism can cope with PC-relative expressions such as labels. Even if a PC-relative ADD or SUB could be constructed, an LDR will be generated to load the PC-relative expression. If a PC-relative ADD or SUB is desired, ADR should be used instead (see ➲*4.6.1 The ADR and ADRL pseudo instructions* on page 4-17). If no suitable literal is already available, the literal placed into the next literal pool will be the offset into the AREA, and an AREA-relative relocation directive will be added to ensure that the constant is appropriate wherever the containing AREA gets located by the linker.

The following example illustrates how this works. The instruction listed in the comment is the ARM instruction which is generated by the assembler.

```
        AREA Loadcon4, CODE
        ENTRY                      ; Mark first instruction
Start
        BL     func1               ; Branch to first subroutine
        BL     func2               ; Branch to second subroutine
        SWI    0x11                ; Terminate
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 contains 3 literals
func2
        LDR    r3, =Darea +6000    ; => LDR r3, [PC, #offset to
                                   ;Litpool 1]
                                   ; (sharing with previous
                                   ; literal)
```

```
       ; LDR  r4, =Darea +6004    ; If uncommented will produce an
                                  ; error as Litpool 2 is out of range
       MOV    pc, lr              ; Return
Darea         % 8000
      END                         ; Literal Pool 2 is out of
                                  ; range of the LDR instructions
                                  ; above
```

The above code is available as `loadcon4.s` in directory `examples/basicasm`. To assemble this, first copy it into your current working directory and then issue the command

```
armasm loadcon4.s
```

To confirm that the assembler produced the correct code, the code area can be disassembled using the ARM Object format decoder:

```
decaof -c loadcon4.o
```

4.6.3 Loading addresses into registers—an example routine

Strings

The following program contains a function, `strcopy`, which copies a string from one memory location to another. Two arguments are passed to the function: the address of the source string and the address of the destination. The last character in the string is a zero, and will be copied.

```
       AREA   StrCopy, CODE
       ENTRY              ; mark the first instruction
main   ADR    r1, srcstr  ; pointer to first string
       ADR    r0, dststr  ; pointer to second string
       BL     strcopy     ; copy the first into second
       SWI    0x11        ; and exit

srcstr DCB "This is my first (source) string",0
dststr DCB "This is my second (destination) string",0
       ALIGN              ; realign address to word boundary

strcopy
       LDRB   r2, [r1], #1 ; load byte, then update address
       STRB   r2, [r0], #1 ; store byte, then update address
       CMP    r2, #0       ; check for zero terminator
       BNE    strcopy      ; keep going if not
       MOV    pc, lr       ; return
       END
```

ADR is used to load the addresses of the two strings into registers r0 and r1, for passing to `strcopy`. These two strings have been stored in memory using the assembler directive `DCB` (Define Constant Byte). The first string is 33 bytes long, so the ADR offset to the second (as a non-word aligned offset) is limited to 255 bytes, which is therefore within reach.

Notice the use of an auto-indexing address mode to update the address registers in the LDR instructions. Thus:

```
LDRB   r2, [r1], #1
```

replaces a sequence like:

```
LDRB   r2, [r1]
ADD    r1, r1, #1
```

but takes only one cycle to execute rather than two.

The above code is available as `strcopy1.s` in directory `examples/basicasm`. Copy this into your current working directory and assemble it, with debug information included:

```
armasm strcopy1.s -g
```

Then link it and load it into the debugger

```
armlink strcopy1.o -o strcopy1 -d
armsd strcopy1
```

You can now view the source and destination strings using:

```
print/s @srcstr
print/s @dststr
```

Run the program and check that the destination string has been updated:

```
go
print/s @srcstr
print/s @dststr
```

Also in the `examples` directory is a version of this program called `strcopy2.s`, which uses `LDR Rd,=PC-relative expression` rather than ADR. Assemble this and compare the code and the code size with that of `strcopy1.s`, using the ARM Object format decoder:

```
armasm strcopy2.s
decaof -c strcopy2.o
decaof -c strcopy1.o
```

It is preferable to use ADR wherever possible, both because it results in shorter code (no storage space is required for addresses to be placed in the literal pool) and because the resulting code will run more quickly (a non-sequential fetch from memory to get the address from the literal pool is not required).

4.7 Jump Tables

Often it is necessary for an application to carry out one of a number of actions dependent upon a certain condition. In C, for instance, this will often be implemented as a `switch()` statement. In assembly language this can be done using a *jump table*.

Suppose we have a function that implements a simple set of arithmetic operations whose first argument controls a logic gate and whose second and third arguments are the gate's inputs. The gate's output is passed as the function's result.

The operations the gate function will respond to are:

0	result = argument1
1	result = argument2
2	result = argument1 + argument2
3	result = argument1 – argument2
4	result = argument2 – argument1

Values outside this range will have the same effect as value 0.

```
       AREA  ArithGate, CODE   ; name this block of code
       ENTRY                   ; mark the first instruction to call
main   MOV   r0, #2            ; set up three parameters
       MOV   r1, #5
       MOV   r2, #15
       BL    arithfunc         ; call the function
       SWI   0x11              ; terminate
arithfunc                      ; label the function
       CMP   r0, #4            ; Treat code as unsigned integer
       BHI   ReturnA1          ; If code > 4 then return first argument
       ADR   r3, JumpTable     ; Load address of the jump table
       LDR   pc,[r3,r0,LSL #2] ; Jump to appropriate routine
JumpTable
       DCD   ReturnA1
       DCD   ReturnA2
       DCD   DoAdd
       DCD   DoSub
       DCD   DoRsb
ReturnA1
       MOV   r0, r1            ; Operation 0, >4
       MOV   pc,lr
ReturnA2
       MOV   r0, r2            ; Operation 1
       MOV   pc,lr
```

```
DoAdd
        ADD     r0, r1, r2        ; Operation 2
        MOV     pc,lr
DoSub
        SUB     r0, r1, r2        ; Operation 3
        MOV     pc,lr
DoRsb
        RSB     r0, r1, r2        ; Operation 4
        MOV     pc,lr
        END                       ; mark the end of this file
```

The ADR pseudo instruction loads the address of the jump table into r3. The following LDR then multiplies the function code in r0 by 4 (using the barrel shifter) and adds this onto the address of the jump table to give the address of the required entry within the jump table. The jump table itself is set up using the DCD directive, which stores the address of the relevant routine (placed there by the linker).

The above code is available as `jump.s` in directory `examples/basicasm`. Copy this into your current working directory and assemble and link it:

```
        armasm jump.s
        armlink jump.o -o jump
```

Then load the resulting program into the debugger:

```
        armsd jump
```

If you now execute the program:

```
        go
```

and display the registers:

```
        reg
```

the value of r0 should be 0x14.

4.8 Using the Load and Store Multiple Instructions

4.8.1 Multiple versus single transfers

The load and store multiple instructions `LDM` and `STM` provide an efficient way of moving the contents of several registers to and from memory. The advantages of using a single load or store multiple instruction over a series of single data transfer instructions are:

- smaller code size
- there is only a single instruction fetch overhead, rather than many instruction fetches
- only one register writeback cycle is required for a load multiple, as opposed to one for every load single
- on uncached ARM processors, the first word of data transferred by a load or store multiple is always a non-sequential memory cycle, but all subsequent words transferred can be sequential (faster) memory cycles

4.8.2 The register list

The registers transferred by the load and store multiple instructions are encoded into the instruction by one bit for each of the registers r0 to r15. A set bit indicates that the register will be transferred, and a clear bit indicates that it will not be transferred. Thus it is possible to transfer any subset of the registers in a single instruction.

The subset of registers to be transferred is specified by listing them in curly brackets. For example:

```
{r1, r4-r6, r8, r10}
```

4.8.3 Increment/decrement, before/after

The base address for the transfer can either be incremented or decremented between register transfers, and this can happen either before or after each register transfer:

```
STMIA  r10, {r1, r3-r5, r8}
```

The suffix `IA` could also have been `IB`, `DA` or `DB`, where `I` indicates increment, `D` decrement, `A` after and `B` before.

In all cases the lowest numbered register is transferred to or from the lowest memory address, and the highest numbered register to or from the highest address. The order in which the registers appear in the register list makes no difference. Also, the ARM always performs sequential memory accesses in increasing memory address order. Therefore ‘decrementing’ transfers actually perform a subtraction first and then increment the transfer address register by register.

4.8.4 Base register writeback

Unless specifically requested, the base register will not be updated at the end of a multiple register transfer instruction. To specify register writeback, you must use the ! character:

```
LDMDB  r11!, {r9, r4-r7}
```

4.8.5 Stack notation

Since the load and store multiple instructions have the facility to update the base register (which for stack operations can be the stack pointer), these instructions provide single instruction push and pop operations for any number of registers (LDM being pop, and STM being push).

The Load and Store Multiple Instructions can be used with several types of stack:

- *ascending* or *descending*
  A stack is able to grow upwards, starting from a low address and progressing to a higher address—an *ascending* stack, or downwards, starting from a high address and progressing to a lower one—a *descending* stack.

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

As stated above, pop and push operations for these stacks can be implemented directly by load and store multiple instructions. To make it easier for the programmer, special stack suffixes can be added to the LDM and STM instructions (as an alternative to Increment/Decrement and Before/After suffixes) as follows:

```
STMFA  r13!, {r0-r5}; Push onto a Full Ascending Stack
LDMFA  r13!, {r0-r5}; Pop from a Full Ascending Stack
STMFD  r13!, {r0-r5}; Push onto a Full Descending Stack
LDMFD  r13!, {r0-r5}; Pop from a Full Descending Stack
STMEA  r13!, {r0-r5}; Push onto an Empty Ascending Stack
LDMEA  r13!, {r0-r5}; Pop from an Empty Ascending Stack
STMED  r13!, {r0-r5}; Push onto Empty Descending Stack
LDMED  r13!, {r0-r5}; Pop from an Empty Descending Stack
```

Note the use of r13 as the base pointer here. By convention r13 is used as the system stack pointer (sp). In addition, the system stack will usually be Full Descending.

The addressing modes are summarised in *Table 4-4: Stack addressing modes*, below.

Name	Stack	Other
pre-increment load	`LDMED`	`LDMIB`
post-increment load	`LDMFD`	`LDMIA`
pre-decrement load	`LDMEA`	`LDMDB`
post-decrement load	`LDMFA`	`LDMDA`
pre-increment store	`STMFA`	`STMIB`
post-increment store	`STMEA`	`STMIA`
pre-decrement store	`STMFD`	`STMDB`
post-decrement store	`STMED`	`STMDA`

Table 4-4: Stack addressing modes

5 Exploring ARM Assembly Language

This chapter presents some useful strategies for optimising the performance of your ARM assembly language programs.

Exploring ARM Assembly Language

5.1 Introduction

This chapter discusses some useful strategies for writing efficient ARM assembly language. It presents algorithms which you can apply in your own programs, and makes extensive use of the examples supplied with the Toolkit release.

It also shows you how to make of use the ARM assembler's more sophisticated features, and how to extract the maximum performance from ARM code.

The subject areas covered are:

- Making use of the stack and writing recursive routines: ➲*5.2 Integer to String Conversion* on page 5-3
- Fast multiplication and division using constants: ➲*5.3 Multiplication by a Constant* on page 5-8 and ➲*5.4 Division by a Constant* on page 5-12
- Manipulating 16-bit data on non Thumb-aware processors: ➲*5.5 Using 16-bit Data on the ARM* on page 5-17
- Generating pseudo-random numbers: ➲*5.6 Pseudo Random Number Generation* on page 5-25
- Loading non-aligned words: ➲*5.7 Loading a Word from an Unknown Alignment* on page 5-27
- Changing the endianness of a word:➲*5.8 Byte Order Reversal* on page 5-28
- Optimising performance: ➲*5.9 ARM Assembly Programming Performance Issues* on page 5-29

5.2 Integer to String Conversion

This section explains how to:

- convert an integer to a string in ARM assembly language
- use a stack in an ARM assembly language program
- write a recursive function in ARM assembly language

The example used can be found in file `utoa1.s` in directory `examples/explasm`. Its `dtoa` entry point converts a signed integer to a string of decimal digits (possibly with a leading '-'); its `utoa` entry point converts an unsigned integer to a string of decimal digits.

5.2.1 Algorithm

To convert a signed integer to a decimal string, generate a '-' and negate the number if it is negative; then convert the remaining unsigned value.

To convert a given unsigned integer to a decimal string, divide it by 10, yielding a quotient and a remainder. The remainder is in the range 0-9 and is used to create the last digit of the decimal representation. If the quotient is non-zero it is dealt with in the same way as the original number, creating the leading digits of the decimal representation; otherwise the process has finished.

5.2.2 Implementation

```
utoa
  STMFD  sp!, {v1, v2, lr}  ; function entry - save some v-registers
                            ; and the return address.
  MOV    v1, a1             ; preserve arguments over following
  MOV    v2, a2             ; function calls

  MOV    a1, a2
  BL     udiv10             ; a1 = a1 / 10

  SUB    v2, v2, a1, LSL #3 ; number - 8*quotient
  SUB    v2, v2, a1, LSL #1 ;        - 2*quotient = remainder

  CMP    a1, #0             ; quotient non-zero?
  MOVNE  a2, a1             ; quotient to a2...
  MOV    a1, v1             ; buffer pointer unconditionally to a1
  BLNE   utoa               ; conditional recursive call to utoa

  ADD    v2, v2, #'0'       ; final digit
  STRB   v2, [a1], #1       ; store digit at end of buffer

  LDMFD  sp!, {v1, v2, pc}  ; function exit - restore and return
```

The implementation of utoa employs the register naming and usage conventions of the ARM Procedure Call Standard:

a1-a4	are argument or scratch registers (a1 is the function result register)
v1-v5	are 'variable' registers (preserved across function calls)
sp	is the stack pointer
lr	holds the subroutine call return address at routine entry
pc	is the program counter

Explanation

On entry, a2 contains the unsigned integer to be converted and a1 addresses a buffer to hold the character representation of it.

On exit, a1 points immediately after the last digit written.

Both the buffer pointer and the original number have to be saved across the call to udiv10. This could be done by saving the values to memory. However, it turns out to be more efficient to use two 'variable' registers, v1 and v2 (which, in turn, have to be saved to memory).

Because utoa calls other functions, it must save its return link address passed in lr. The function therefore begins by stacking v1, v2 and lr using STMFD sp!, {v1,v2,lr}.

In the next block of code, a1 and a2 are saved (across the call to udiv10) in v1 and v2 respectively and the given number (a2) is moved to the first argument register (a1) before calling udiv10 with a BL (Branch with Link) instruction.

On return from udiv10, 10 times the quotient is subtracted from the original number (preserved in v2) by two SUB instructions. The remainder (in v2) is ready to be converted to character form (by adding ASCII '0') and to be stored into the output buffer.

But first, utoa has to be called to convert the quotient, unless that is zero. The next four instructions do this, comparing the quotient (in a1) with 0, moving the quotient to the second argument register (a2) if not zero, moving the buffer pointer to the first argument/result register (a1), and calling utoa if the quotient is not zero.

Note that the buffer pointer is moved to a1 unconditionally: if utoa is called recursively, a1 will be updated but will still identify the next free buffer location; if utoa is not called recursively, the next free buffer location is still needed in a1 by the following code which plants the remainder digit and returns the updated buffer location (via a1).

The remainder (in a2) is converted to character form by adding '0' and is then stored in the location addressed by a1. A post-incrementing STRB is used which stores the character and increments the buffer pointer in a single instruction, leaving the result value in a1.

Finally, the function is exited by restoring the saved values of v1 and v2 from the stack, loading the stacked link address into pc and popping the stack using a single multiple-load instruction:

```
LDMFD sp!, {v1,v2,pc}
```

5.2.3 Creating a runnable example

You can run the `utoa` routine described here under armsd. To do this, you must assemble the example and the `udiv10` function, compile a simple test harness written in C, and link the resulting objects together to create a runnable program.

Copy `utua1.s`, `udiv10.s` and `utuatest.c` from directory `examples/explasm` to your current working directory.Then issue the following commands:

```
armasm utoa1.s -o utoa1.o -li
armasm udiv10.s -o udiv10.o -li
armcc -c utoatest.c -apcs 3/32bit
armlink -o utoatest utoa1.o udiv10.o utoatest.o libpath/armlib.32l
```

where *`libpath`* is your release `lib` directory.

The first two armasm commands assemble the `utoa` function and the `udiv10` function, creating relocatable object files `utoa1.o` and `udiv10.o`. The `-li` flag tells armasm to assemble for a little-endian memory. You can omit this flag if your armasm has been configured to do this by default.

The `armcc` command compiles the test harness. The `-c` flag tells armcc not to link its output with the C library; the `-li` flag tells armcc to compile for a little-endian memory (as with armasm).

The `armlink` command links your three relocatable objects with the ARM C library to create a runnable program (here called `utoatest`).

If you have installed your ARM development tools in a standard way then you could use the following shorter command to do the compilation and linking:

```
armcc utoatest.c utoa1.o udiv10.o -apcs 3/32bit -li
```

5.2.4 Running the example

You can run your example program under armsd using:

```
armsd -li utoatest
```

Note *The `-li` and `-apcs 3/32bit` options can be omitted if the tools are configured appropriately.*

5.2.5 Stacks in assembly language

In this example, three words are pushed on to the stack on entry to `utoa` and popped off again on exit. By convention, ARM software uses r13, usually called `sp`, as a stack pointer pointing to the last-used word of a downward growing stack (a so-called 'full, descending' stack). However, this is only a convention and the ARM instruction set supports equally all four stacking possibilities: `FD`, `FA`, `ED`, `EA`.

The instruction used to push values on the stack was:

```
STMFD  sp!, {v1, v2, lr}
```

The action of this instruction is as follows:

1. Subtract 4 * number-of-registers from `sp`
2. Store the registers named in {...} in ascending register number order to memory at [`sp`], [`sp`,4], [`sp`,8] ...

The matching pop instruction was:

```
LDMFD  sp!, {v1, v2, pc}
```

Its action is:

1. Load the registers named in {...} in ascending register number order from memory at [`sp`], [`sp`,4], [`sp`,8] ...
2. Add 4 * number-of-registers to `sp`.

Many, if not most, register-save requirements in simple assembly language programs can be met using this approach to stacks.

A more complete treatment of run-time stacks requires a discussion of:

- stack-limit checking (and extension)
- local variables and stack frames

In the `utoa` program, you must assume the stack is big enough to deal with the maximum depth of recursion, and in practice this assumption will be valid. The biggest 32-bit unsigned integer is about four billion, or ten decimal digits. This means that at most 10 x 3 registers = 120 bytes have to be stacked. Because the ARM Procedure Call Standard guarantees that there are at least 256 bytes of stack available when a function is called, and because we can guess (or know) that `udiv10` uses no stack space, we can be confident that `utoa` is quite safe if called by an APCS-conforming caller such as a compiled C test harness.

The stacking technique illustrated here conforms to the ARM Procedure Call Standard *only if the function using it makes no function calls*. Since `utoa` calls both `udiv10` and itself, it really ought to establish a proper stack frame—see ➲*The ARM Software Development Toolkit Reference Manual: Chapter 19, ARM Procedure Call Standard*. If you really want to write functions that can 'plug and play together' you will have to follow the APCS exactly.

However, when writing a whole program in assembly language you often know much more than when writing a program fragment for general, robust service. This allows you to gently break the APCS in the following way:

- Any chain of function/subroutine calls can be considered compatible with the APCS provided it uses less than 256 bytes of stack space.

So the `utoa` example is APCS compatible, even though it is not APCS conforming.

Be aware however that if you call any function whose stack use is unknown (but which is believed to be APCS-conforming), you court disaster unless you establish a proper APCS call frame and perform APCS stack limit checking on function entry. Please refer to ➲*The ARM Software Development Toolkit Reference Manual: Chapter 19, ARM Procedure Call Standard* for further details.

5.3 Multiplication by a Constant

Note *Throughout the following discussion, registers are referred to using the names Rd, Rm, and Rs, but when trying the examples out for yourself you should use the default register names r0-r15, or names which have been declared using the RN assembler directive.*

This section explains how to construct a sequence of ARM instructions to multiply by a constant.

For some applications in which speed is essential—Digital Signal Pocessing, for example—multiply is used extensively.

In many cases where a multiply is used, one of the values is a constant (eg. weeks*7). A naive programmer would assume that the only way to calculate this would be to use the `MUL` instruction, but there is an alternative.

This section demonstrates how to improve the speed of multiply-by-constant by using a sequence of arithmetic instructions instead of the general-purpose multiplier.

5.3.1 Introduction

The `MUL` instruction has the following syntax:

```
MUL     Rd, Rm, Rs
```

The timing of this instruction depends on the value in Rs. The ARM6 datasheet specifies that for Rs between 2^(2m-3) and 2^(2m-1)-1 inclusive takes 1S + mI cycles.

Note ARM 7M family processors have a different implementation of `MUL`. This leads to a different relationship of cycle counts to values of Rs.

When multiplying by a constant value, it is possible to replace the general multiply with a fixed sequence of adds and subtracts that have the same effect. For instance, multiply by 5 could be achieved using a single instruction:

```
ADD     Rd, Rm, Rm, LSL #2        ; Rd = Rm + (Rm * 4) = Rm * 5
```

This is obviously better than the `MUL` version:

```
MOV     Rs, #5
MUL     Rd, Rm, Rs
```

The cost of the general multiply includes the instructions needed to load the constant into a register (up to four may be needed, or an `LDR` from a literal pool) as well as the multiply itself.

5.3.2 Finding the optimum sequence

The difficulty in using a sequence of arithmetic instructions is that the constant must be decomposed into a set of operations which can be done by one instruction each. Consider multiply by 105:

This could be achieved by decomposing, as follows:

```
105 == 128 - 23
    == 128 - (16 + 7)
    == 128 - (16 + (8 - 1))

RSB    Rd, Rm, Rm, LSL #3        ; Rd = Rm*7
ADD    Rd, Rd, Rm, LSL #4        ; Rd = Rm*7 + Rm*16 = Rm*23
RSB    Rd, Rd, Rm, LSL #7        ; Rd = -Rm*23 + Rm*128 = Rm*105
```

Or as follows:

```
105 == 15 * 7
    == (16 - 1) * (8 - 1)

RSB    Rt, Rm, Rm, LSL #4        ; Rt = Rm*15 (tmp reg)
RSB    Rd, Rt, Rt, LSL #3        ; Rd = Rt*7 = Rm*105
```

The second method is the optimal solution (fairly easy to find for small values such as 105). However, the problem of finding the optimum becomes much more difficult for larger constant values. A program can be written to search exhaustively for the optimum, but it may take a long time to execute. There are no known algorithms which solve this problem quickly.

Temporary registers can be used to store intermediate results to help achieve the shortest sequence. For a large constant, more than one temporary may be needed, otherwise the sequence will be longer.

The C compiler restricts the amount of searching it performs in order to minimise the impact on compilation time. The current version of armcc has a cut-off so that it uses a normal `MUL` if the number of instructions used in the multiply-by-constant sequence exceeds some number *N*. This is to avoid the sequence becoming too long.

5.3.3 Experimenting with armcc assembly output

When writing a speed-critical ARM assembler program, it is a good idea to code it in C first (to check the algorithm) before converting it to hand-tuned assembler. It is helpful to see the ARM code which the compiler generates as a starting point for your work.

Invoking armcc with the `-S` flag will generate an assembly file instead of an object file. For example, consider the following simple C code:

```
int mulby105( int num )
{
    return num * 105;
}
```

Compile this using:

```
armcc -li -S mulby105.c
```

Now, examine the file `mulby105.s` which has been created. (Your version of armcc may not produce precisely this output, although it should be very similar):

```
; generated by Norcroft ARM C vsn 4.41 (Advanced RISC Machines)
    AREA |C$$code|, CODE, READONLY
|x$codeseg|

    EXPORT  mulby105
mulby105
    RSB    a1,a1,a1,LSL #4
    RSB    a1,a1,a1,LSL #3
    MOV    pc,lr

    AREA |C$$data|,DATA

|x$dataseg|

    END
```

Notice that the compiler has found the short multiply-by-constant sequence.

5.3.4 Discussion of speed improvement

To evaluate the speed gains from using multiply-by-constant, consider multiplying by 11585 (which is 8192*sqr2):

In a normal multiply, the load-a-constant stage may take up to four instructions (in this case two) or an LDR,= and the multiply takes one instruction fetch plus a number of internal cycles to calculate the multiply (on ARM6 based processors):

```
MOV    Rs, #0x2D << 8           ; load constant
ORR    Rs, Rs, #0x41            ; load constant, now Rs = 11585
MUL    Rd, Rm, Rs               ; do the multiply
```

The optimal multiply-by-constant sequence consists of just four data-processing instructions:

```
ADD    Rd, Rm, Rm, LSL #1       ; Rd = Rm*3
RSB    Rd, Rd, Rd, LSL #4       ; Rd = Rd*15 = Rm*45
ADD    Rd, Rm, Rd, LSL #8       ; Rd = Rm + Rd*256 = Rm*11521
ADD    Rd, Rd, Rm, LSL #6       ; Rd = Rd + Rm*64 = Rm*11585
```

The following table shows a comparison of these methods:

Method	Cycles
MUL instruction	3 instructions + MUL internal cycles
Multiply by constant	4 instructions

Table 5-1: Simple comparison of performance

On ARM6 processors, the 2-bit Booth's Multiplier used by MUL takes a number of I-cycles depending on the value in Rs (in this case m=8, as Rs lies between 8192 and 32767). In this case, multiply-by-constant performs better. On the ARM60, an instruction fetch is an external memory S-cycle, or a cache F-cycle (if there is a cache hit) on cached processors.

With slow memory systems and non-cached processors, I-cycles can be much faster than other cycles because they are internal to the ARM core. So, the general multiply can sometimes be the fastest option (for large constants where an efficient solution cannot be found). It should also use less memory. If the load-a-constant stage could be moved outside a loop, this favours the general multiply, as there is only the MUL to execute.

Method	Cycles on ARM60	Cycles on ARM610
MUL instruction	3S + 8I	11F
Multiply by constant	4S	4F

Table 5-2: Comparison of performance on cached and uncached processors

5.4 Division by a Constant

The ARM instruction set was designed following a RISC philosophy. One of the consequences of this is that the ARM core has no divide instruction, so divides must be performed using a subroutine. This means that divides can be quite slow, but this is not a major issue as divide performance is rarely critical for applications.

It is possible to do better than the general divide in the special case when the divisor is a constant. This section shows how the divide-by-constant technique works, and how to generate ARM assembler code for divide-by-constant.

This section explains:

- how to improve on the general divide code for the case when the divisor is a constant
- the simple case for divide-by-2^n using the barrel shifter
- how to use `divc.c` to generate ARM code for divide-by-constant

5.4.1 Special case for divide-by-2^n

In the special case when dividing by 2^n, a simple right shift is all that is required.

There is a small caveat which concerns the handling of signed and unsigned numbers.
For signed numbers, an *arithmetic* right shift is required, as this performs sign extension (to handle negative numbers correctly). In contrast, unsigned numbers require a 0-filled *logical* shift right:

```
    MOV    a2, a1, lsr #5           ; unsigned division by 32
    MOV    a2, a1, asr #10          ; signed division by 1024
```

5.4.2 Explanation of divide-by-constant ARM code

The divide-by-constant technique basically does a multiply in place of the divide, but is somewhat more complicated than the multiply technique described in ➲*5.3 Multiplication by a Constant*. Given that:

```
  x/y == x * (1/y)
```

consider the underlined portion as a 0.32 fixed-point number (truncating any bits past the most significant 32). 0.32 means 0 bits before the decimal point and 32 after it.

```
  == (x * (2^32/y)) / 2^32
```

the underlined portion here is a 32.0 bit fixed-point number:

```
  == (x * (2^32/y)) >> 32
```

This is effectively returning the top 32-bits of the 64-bit product of x and (2^32/y).

If y is a constant, then (2^32/y) is also a constant.

For certain y, the reciprocal (2^32/y) is a repeating pattern in binary:

```
  y                (2^32/y)

  2      10000000000000000000000000000000    #
  3      01010101010101010101010101010101    *
  4      01000000000000000000000000000000    #
  5      00110011001100110011001100110011    *
  6      00101010101010101010101010101010    *
  7      00100100100100100100100100100100    *
  8      00100000000000000000000000000000    #
  9      00011100011100011100011100011100    *
 10      00011001100110011001100110011001    *
 11      00010111010001011101000101110100
 12      00010101010101010101010101010101    *
 13      00010011101100010011101100010011
 14      00010010010010010010010010010010    *
 15      00010001000100010001000100010001    *
 16      00010000000000000000000000000000    #
 17      00001111000011110000111100001111    *
 18      00001110001110001110001110001110    *
 19      00001101011110010100001101011110
 20      00001100110011001100110011001100    *
 21      00001100001100001100001100001100    *
 22      00001011101000101110100010111010
 23      00001011001000010110010000101100
 24      00001010101010101010101010101010    *
 25      00001010001111010111000010100011
```

The lines marked with a ’#’ are the special cases 2^n, which have already been dealt with.
The lines marked with a ’*’ have a simple repeating pattern.

Note how regular the patterns are for y=2^n+2^m or y=2^n-2^m (for n>m):

n	m	(2^n+2^m)	n	m	(2^n-2^m)
1	0	3	1	0	1
2	0	5	2	1	2
2	1	6	2	0	3
3	0	9	3	2	4
3	1	10	3	1	6
3	2	12	3	0	7
4	0	17	4	3	8
4	1	18	4	2	12
4	2	20	4	1	14
4	3	24	4	0	15
5	0	33	5	4	16
5	1	34	5	3	24
5	2	36	5	2	28
5	3	40	5	1	30
5	4	48	5	0	31

For the repeating patterns, it is a relatively easy matter to calculate the product by using a multiply-by-constant method.

The result can be calculated in a small number of instructions by taking advantage of the repetition in the pattern. This corresponds to the optimal solution in the multiply-by-constant problem (see ➲*5.3 Multiplication by a Constant* on page 5-8).

The actual multiply is slightly unusual due to the need to return the top 32 bits of the 64-bit result. It efficient to calculate just the top 32 bits. This can be achieved by modifying the multiply-by-constant sequence so that the input value is shifted right rather than left.

Consider this fragment of the divide-by-ten code (x is the input dividend as used in the above equations):

```
SUB  a1,  x,  x, lsr #2   ; a1 = x*%0.11000000000000000000000000000000
ADD  a1, a1, a1, lsr #4   ; a1 = x*%0.11001100000000000000000000000000
ADD  a1, a1, a1, lsr #8   ; a1 = x*%0.11001100110011000000000000000000
ADD  a1, a1, a1, lsr #16  ; a1 = x*%0.11001100110011001100110011001100
MOV  a1, a1, lsr #3       ; a1 = x*%0.00011001100110011001100110011001
```

The SUB calculates (for example):

```
a1 = x - x/4
   = x - x*%0.01
   = x*%0.11
```

Therefore, just five instructions are needed to perform the multiply.

A small problem is caused by calculating just the top 32 bits, as this ignores any carry from the low 32 bits of the 64-bit product. Fortunately, this can be corrected. A correct divide would round down, so the remainder can be calculated by:

```
x - (x/10)*10 = 0..9
```

By making good use of the ARM's barrel shifter, it takes just two ARM instructions to perform this multiply-by-10 and subtract. In the case when (x/10) is too small by 1 (if carry has been lost), the remainder will be in the range 10..19, in which case corrections must be applied. This test would require a compare-with-10 instruction, but this can be combined with other operations to save an instruction (see below).

When a lost carry is detected, both the quotient and remainder must be fixed up (one instruction each).

The following fragment should explain the full divide-by-10 code:

```
div10
; takes argument in a1
; returns quotient in a1, remainder in a2
; cycles could be saved if only divide or remainder is required
    SUB    a2, a1, #10              ; keep (x-10) for later
    SUB    a1, a1, a1, lsr #2
    ADD    a1, a1, a1, lsr #4
    ADD    a1, a1, a1, lsr #8
    ADD    a1, a1, a1, lsr #16
    MOV    a1, a1, lsr #3
    ADD    a3, a1, a1, asl #2
    SUBS   a2, a2, a3, asl #1       ; calc (x-10) - (x/10)*10
    ADDPL  a1, a1, #1               ; fix-up quotient
    ADDMI  a2, a2, #10              ; fix-up remainder
    MOV    pc, lr
```

The optimisation which eliminates the compare-with-10 instruction is to keep (x-10) for use in the subtraction to calculate the remainder. This means that compare-with-0 is required instead, which is easily achieved by adding an S (to set the flags) to the SUB opcode. This also means that the subtraction has to be undone if no rounding error occurred (which is why the ADDMI instruction is used).

5.4.3 How to generate divide-by-constant sequences

For suitable numbers, the details of the divide-by-constant technique can be avoided completely by using the `divc` program. This is supplied in ANSI C source in directory `examples/explasm`. You can compile it either with your host system's C compiler, or with armcc in which case the executable must be run using armsd.

You can get command-line help by running `divc` with no arguments.

A Thumb version of `divc` may be found in directory `examples/thumb`.

```
Usage: divc <n>
Generates optimal ARM code for divide-by-constant
where <n> is one of (2^n-2^m) or (2^n+2^m) eg. 10
Advanced RISC Machines [01 Jul 92]
```

To generate the ARM assembler code for divide-by-10, type:

```
divc 10
```

The output is suitable for immediate use as an armasm source file.

The routine is called `udiv10` for unsigned divide-by-10 (for example). It takes the unsigned argument in `a1`, and returns the quotient in `a1` and the remainder in `a2`. It conforms fully to the APCS, but the remainder may not be available when called from C.

The range of values covered by (2^n-2^m) and (2^n+2^m) contains some useful numbers such as 7, 10, 24, 60.

5.5 Using 16-bit Data on the ARM

Note *This section will only be of interest to designers working with Architecture 3 ARM cores and devices (eg. ARM6, ARM60, ARM610).*

ARM processors designed using ARM Architecture 4 have instructions for loading and storing halfword values. ARM processors designed using version 3 of the architecture, while lacking halfword support, are still capable of handling16-bit data efficiently, as this section will demonstrate.

This section covers several different approaches to 16-bit data manipulation on ARM processors which do not have halfword support:

- Converting the 16-bit data to 32-bit data, and from then on treating it as 32-bit data
- Converting 16-bit data into 32-bit data when loading and storing, but using 32-bit data within ARM's registers
- Loading 16-bit data into the top 16-bits of ARM registers, and processing it as 16-bit data (ie. keeping the bottom 16-bits clear at all times)

Useful code fragments are given which can be used to help implement these different approaches efficiently.

5.5.1 16-bit values in 32-bit words

Because data is 16-bit in size does not mean that it cannot be considered as 32-bit data and thus be manipulated using the ARM instruction set in the normal way.

Any unsigned 16-bit value can be held as a 32-bit value in which the top 16 bits are all zero. Similarly any signed 16-bit value can be held as a 32-bit value with the top 16 bits sign extended (ie. copied from the top bit of the 16-bit value).

The main disadvantage of storing 16-bit data as 32-bit data in this way for ARM-based systems is that it takes up twice as much space in memory or on disk. If the amount of memory taken up by the 16-bit data is small, then simply treating it as 32-bit data is likely to be the easiest and most efficient technique (ie. converting the data to 32-bit format and from then on treating it as 32-bit data.)

When the space taken by 16-bit data in memory or on disk is not small, an alternative method can be used. The 16-bit data is loaded and converted to be 32-bit data for use within the ARM, and then when processed, can either be output as 32-bit or 16-bit data. Useful code fragments are given to perform the necessary conversions for this approach in ➲*5.5.2 Little-endian loading* on page 5-18 to section ➲*5.5.5 Big-endian storing* on page 5-22.

Detecting 16-bit data

An issue which may arise when 16-bit data is converted to 32-bit data for use in the ARM and then stored back out as 16-bit data is detecting whether the data is still 16-bit data, ie. whether it has 'overflowed' into the top 16 bits of the ARM register. Code fragments which detect this are given in section ➲*5.5.6 Detecting overflow into the top 16 bits* on page 5-23.

Another approach which avoids having to use explicit code to check whether results have overflowed into the top 16-bits is to keep 16-bit data as 16-bit data all the time, by loading it into the top half of ARM registers, and ensuring that the bottom 16 bits are always 0. Useful code sequences, and the issues involved when taking this approach are described in *5.5.7 Using ARM registers as 16-bit registers* on page 5-24.

5.5.2 Little-endian loading

Code fragments in this section which transfer a single 16-bit data item transfer it to the least significant 16 bits of an ARM register. The byte offset referred to is the byte offset within a word at the load address. eg. the address 0x4321 has a byte offset of 1.

One data item - any alignment (byte offsets 0,1,2 or 3)

The following code fragment loads a 16-bit value into a register, whether the data is byte, halfword or word-aligned in memory, by using the ARM's load byte instruction.

This code is also optimal for the common case where the 16-bit data is half word-aligned, ie. at either byte offset 0 or 2 (but the same code is required to deal with both cases). Optimisations can be made when it is known that the data is at byte offset 0, and also when it is known to be at byte offset 2 (but not when it could be at either offset).

```
    LDRB   R0, [R2, #0]             ; 16-bit value is loaded from the
    LDRB   R1, [R2, #1]             ; address in R2, and put in R0
    ORR    R0, R0, R1, LSL #8       ; R1 is required as a
;   MOV    R0, R0, LSL #16          ; temporary register
;   MOV   R0, R0, ASR #16
```

The two MOV instructions are only required if the 16-bit value is signed, and it may be possible to combine the second MOV with another data-processing operation by specifying the second argument as R0, ASR, #16 rather than just R0.

One data item - byte offset 2

If the data is aligned on a half word boundary, but not a word boundary (ie. the byte offset is 2), then the following code fragment can be used (which is clearly much more efficient than the general case given above):

```
    LDR    R0, [R2, #-2]   ; 16-bit data is loaded from
                           ; address in R2 into R0
    MOV    R0, R0, LSR #16 ; (R2 has byte offset 2)
```

The LSR should be replaced with ASR if the data is signed. Note that as in the previous example it may be possible to combine the MOV with another data processing operation.

One data item - byte offset 0

If the data is on a word boundary, the following code fragment will load a 16-bit value (again a significant improvement over the general case):

```
LDR    R0, [R2, #0]          ; 16-bit value is loaded from the
MOV    R0, R0, LSL #16       ; word-aligned address in R2
MOV    R0, R0, LSR #16       ; into R0
```

As before, LSR should be replaced with ASR if the data is signed. Also, it may be possible to combine the second MOV with another data processing operation.

This code can be further optimised if non-word-aligned word-loads are permitted (ie. alignment faults are not enabled). This makes use of the way ARM rotates data into a register for non-word-aligned word-loads (see the appropriate ARM Datasheet for more information):

```
LDR    R0, [R2, #2]          ; 16-bit value is loaded from the
MOV    R0, R0, LSR #16       ; word-aligned address in R2
                             ; into R0.
```

Two data items - byte offset 0

Two 16-bit values stored in one word can be loaded more efficiently than two separate values. The following code loads two unsigned 16-bit data items into two registers from a word-aligned address:

```
LDR    R0, [R2, #0]          ; 2 unsigned 16-bit values are
MOV    R1, R0, LSR #16       ; loaded from one word of memory
BIC    R0, R0, R1, LSL #16   ; [R2]. The 1st is put in R0, and
                             ; the 2nd in R1.
```

The version of this for signed data is:

```
LDR    R0, [R2, #0]          ; 2 signed 16-bit values are
MOV    R1, R0, ASR #16       ; loaded from one word of memory
MOV    R0, R0, LSL #16       ; [R2]. The 1st is put in R0, and
MOV    R0, R0, ASR #16       ; the 2nd in R1.
```

The address in R2 should be word-aligned (byte offset 0), in which case these code fragments load the data item in bytes 0-1 into R0, and the data item in bytes 2-3 into R1.

5.5.3 Little-endian storing

The code fragment in this section transfers a single 16-bit data item from the least-significant 16 bits of an ARM register. The byte offset referred to is the byte offset within a word of the store address. For example, the address 0x4321 has a byte offset of 1.

One data item - any alignment (byte offsets 0,1,2 or 3)

The following code fragment saves a 16-bit value to memory, whatever the alignment of the data address, by using the ARM's byte-saving instructions:

```
    STRB   R0, [R2, #0]          ; 16-bit value is stored to the
    MOV    R0, R0, ROR #8        ; address in R2.
    STRB   R0, [R2, #1]
;   MOV    R0, R0, ROR #24
```

The second `MOV` instruction can be omitted if the data is no longer needed after being stored.

Unlike load operations, knowing the alignment of the destination address does not make optimisations possible.

Two data items - byte offset 0

Two unsigned 16-bit values in two registers can be packed into a single word of memory very efficiently, as the following code fragment demonstrates:

```
    ORR    R3, R0, R1, LSL #16   ; Two unsigned 16-bit values
    STR    R3, [R2, #0]          ; in R0 and R1 are packed into
                                 ; the word addressed by R2
                                 ; R3 is a temporary register
```

If the values in R0 and R1 are not needed after they are saved, R3 need not be used as a temporary register (one of R0 or R1 can be used instead).

The version for signed data is:

```
    MOV    R3, R0, LSL #16       ; Two signed 16-bit values
    MOV    R3, R3, LSR #16       ; in R0 and R1 are packed into
    ORR    R3, R3, R1, LSL #16   ; the word addressed by R2
    STR    R3, [R2, #0]          ; R3 is a temporary register
```

Again, if the values in R0 and R1 are not needed after they are saved, R3 need not be used as a temporary register (R0 can be used instead).

5.5.4 Big-endian loading

Code fragments in this section transfer a single 16-bit data item to the least-significant 16 bits of an ARM register. The byte offset referred to is the byte offset within a word at the load address. Foe example, the address 0x4321 has a byte offset of 1.

One data item - any alignment (byte offsets 0,1,2 or 3)

The following code fragment loads a 16-bit value into a register using the load byte instruction (`LDRB`). The data may be byte, halfword or word-aligned.

This code is also optimal for the common case where the 16-bit data is half word-aligned; ie. at either byte offset 0 or 2 (but the same code is required to deal with both cases). Optimisations can be made when it is known that the data is at byte offset 0, and also when it is known to be at byte offset 2 (but not when it could be at either offset).

```
    LDRB   R0, [R2, #0]          ; 16-bit value is loaded from the
    LDRB   R1, [R2, #1]          ; address in R2, and put in R0
    ORR    R0, R1, R0, LSL #8    ; R1 is a temporary register
;   MOV    R0, R0, LSL #16
;   MOV    R0, R0, ASR #16
```

The two `MOV` instructions are only required if the 16-bit value is signed, and it may be possible to combine the second `MOV` with another data-processing operation by specifying the second argument as `R0, ASR, #16` rather than simply `R0`.

One data item - byte offset 0

If the data is aligned on a word boundary, the following code fragment can be used (which is clearly much more efficient than the general case given above):

```
    LDR    R0, [R2, #0]          ; 16-bit value is loaded from the
    MOV    R0, R0, LSR #16       ; word-aligned address in R2
                                 ; into R0.
```

The `LSR` should be replaced with `ASR` if the data is signed. Note that as in the previous example it may be possible to combine the `MOV` with another data-processing operation.

One data item - byte offset 2

If the data is aligned on a halfword boundary, but not a word boundary (ie. the byte offset is 2) the following code fragment can be used (again a significant improvement over the general case):

```
    LDR    R0, [R2, #-2]         ; 16-bit value is loaded from the
    MOV    R0, R0, LSL #16       ; address in R2 into R0.  R2 is
    MOV    R0, R0, LSR #16       ; aligned to byte offset 2
```

As before, `LSR` should be replaced with `ASR` if the data is signed. Also, it may be possible to combine the second `MOV` with another data-processing operation.

This code can be further optimised if non-word-aligned word-loads are permitted (ie. alignment faults are not enabled). This makes use of the way in which the ARM rotates data into a register for non-word-aligned word loads:

```
    LDR    R0, [R2, #0]          ; 16-bit value is loaded from the
    MOV    R0, R0, LSR #16       ; address in R2 into R0.  R2 is
                                 ; aligned to byte offset 2
```

Two data items - byte offset 0

Two 16-bit values stored in one word can be loaded more efficiently than two separate values. The following code loads two unsigned 16-bit data items into two registers from a word-aligned address:

```
    LDR    R0, [R2, #0]         ; 2 unsigned 16-bit values are
    MOV    R1, R0, LSR #16      ; loaded from one word of memory.
    BIC    R0, R0, R1, LSL #16  ; The 1st in R0, the 2nd in R1.
```

The version of this for signed data is:

```
    LDR    R0, [R2, #0]         ; 2 signed 16-bit values are
    MOV    R1, R0, ASR #16      ; loaded from one word of memory.
    MOV    R0, R0, LSL #16      ; The 1st in R0, the 2nd in R1.
    MOV    R0, R0, ASR #16      ; into R1.
```

5.5.5 Big-endian storing

The code fragment in this section which transfers a single 16-bit data item, transfers it from the least-significant 16 bits of an ARM register. The byte offset referred to is the byte offset from a word address of the store address; eg. the address 0x4321 has a byte offset of 1.

One data item - any alignment (byte offsets 0,1,2 or 3)

The following code fragment saves a 16-bit value to memory, whatever the alignment of the data address:

```
    STRB   R0, [R2, #1]             ; 16-bit value is stored to the
    MOV    R0, R0, ROR #8           ; address in R2.
    STRB   R0, [R2, #0]
;   MOV    R0, R0, ROR #24
```

The second MOV instruction can be omitted if the data is no longer needed after being stored.

Unlike load operations, knowing the alignment of the destination address does not make optimisations possible.

Two data items - byte offset 0

Two unsigned 16-bit values in two registers can be packed into a single word of memory very efficiently, as the following code fragment demonstrates:

```
    ORR    R3, R0, R1, LSL #16       ; Two unsigned 16-bit values in
    STR    R3, [R2, #0]              ; R0 and R1 are packed into the
                                     ; word addressed by R2
    ; R3 is used as a temporary register
```

If the values in R0 and R1 are not needed after they are saved, R3 need not be used as a temporary register (one of R0 or R1 can be used instead).

The version for signed data is:

```
    MOV    R3, R0, LSL #16           ; Two signed 16-bit values in
    MOV    R3, R3, LSR #16           ; R0 and R1 are packed into the
    ORR    R3, R3, R1, LSL #16       ; word addressed by R2.
    STR    R3, [R2, #0]              ; R3 is a temporary register
```

Again, if the values in R0 and R1 are not needed after they are saved, R3 need not be used as a temporary register (R0 can be used instead).

5.5.6 Detecting overflow into the top 16 bits

If 16-bit data is converted to 32-bit data for use in the ARM, it may sometimes be necessary to check explicitly whether the result of a calculation has 'overflowed' into the top 16 bits of an ARM register. This is likely to be necessary because the ARM does not set its processor status flags when this happens.

The following instruction sets the Z flag if the value in R0 is a 16-bit unsigned value. R1 is used as a temporary register:

```
    MOVS   R1, R0, LSR #16
```

The following instructions set the Z flag if the value in R0 is a valid 16-bit signed value (ie. bit 15 is the same as the sign extended bits). R1 is used as a temporary register:

```
    MOVS   R1, R0, ASR #15
    CMNNE  R1, R1, #1
```

5.5.7 Using ARM registers as 16-bit registers

The final method of handling 16-bit data is to load it into the top 16 bits of the ARM registers, effectively making them 16-bit registers. This approach has several advantages:

- Some 16-bit data load instruction sequences are shorter. The loading and storing sequences shown above will have to be modified, and in some cases shorter instruction sequences will be possible. In particular, handling signed data will often be more efficient, as the top bit does not have to be copied into the top 16 bits of the register. However, note that the bottom 16 bits must be clear at all times.
- The ARM processor status flags will be set if the 'S' bit of a data processing instruction is set and overflow or carry occurs out of the 16-bit value. Thus, explicit 'overflow' checking instructions are not needed.
- Pairs of signed 16-bit integers can be saved more efficiently than in the previous approach, since the sign-extended bits do not have to be cleared out before the two values are combined.

There are also disadvantages:

- Instructions such as add with carry cannot be used. For example,. the instruction to increment R0 if Carry is set:

  ```
  ADC       R0, R0, #0
  ```

  must be replaced by:

  ```
  ADDCS     R0, R0, #&10000
  ```

  Using this form of instruction reduces the chances of being able to combine several data-processing operations into one by making use of the barrel shifter.
- Before two 16-bit values can be multiplied, they must be shifted into the bottom half of the register
- Before combining a 16-bit value with a 32-bit value, the 16-bit value must be shifted into the bottom half of the register. Note, however, that this may cost nothing if the barrel shifter can be used in parallel.

The examples given above for loading and storing 16-bit data into the bottom half of ARM registers can be easily adapted to load the data into the top half of the registers (and ensure the bottom half is all zero), or save the data from the top half of the registers.

5.6 Pseudo Random Number Generation

This section describes a 32-bit pseudo random number generator implemented efficiently in ARM Assembly Language.

It is often necessary to generate pseudo random numbers, and the most efficient algorithms are based on shift generators with exclusive-or feedback (rather like a cyclic redundancy check generator). Unfortunately, the sequence of a 32-bit generator needs more than one feedback tap to be maximal length (ie. 2^32-1 cycles before repetition), so this example uses a 33 bit register with taps at bits 33 and 20.

The basic algorithm is:

- newbit:=bit33 EOR bit20
- shift left the 33 bit number
- put in newbit at the bottom

This operation is performed for all the newbits needed (ie. 32 bits). The entire operation can be coded compactly by making maximal use of the ARM's barrel shifter:

```
; enter with seed in R0 (32 bits), R1 (1 bit in least significant bit)
; R2 is used as a temporary register.
; on exit the new seed is in R0 and R1 as before
; Note that a seed of 0 will always produce a new seed of 0.
; All other values produce a maximal length sequence.
;
    TST    R1, R1, LSR #1         ; top bit into Carry
    MOVS   R2, R0, RRX            ; 33 bit rotate right
    ADC    R1, R1, R1             ; carry into lsb of R1
    EOR    R2, R2, R0, LSL #12    ; (involved!)
    EOR    R0, R2, R2, LSR #20    ; (similarly involved!)
```

5.6.1 Using this example

This random number generation code is provided as `random.s` in directory `examples/explasm`. It is provided as ARM Assembly Language source which can be assembled and then linked with C modules (see ➲*Chapter 7, Interfacing C and Assembly Language* for more information).

The C test program `randtest.c` (also in directory `examples/explasm`) can be used to demonstrate this.

In the following commands:

`-li` indicates that the target ARM is little-endian

`-apcs 3/32bit` specifies that the 32-bit variant of the ARM Procedure Call Standard should be used.

These options can be omitted if the tools have already been configured appropriately.

First copy `random.s` and `randtest.c` from directory `examples/explasm` to your current directory, and enter the following commands to build an executable suitable for armsd:

```
armasm random.s -o random.o -li
armcc -c randtest.c -li -apcs 3/32bit
armlink randtest.o random.o -o randtest libpath/armlib.32l
```

Where *`libpath`* is the path to the toolkit’s `lib` directory on your system.

armsd can be used to run this program as follows:

```
> armsd -li randtest
A.R.M. Source-level Debugger, version 4.10 (A.R.M.) [Aug 26 1992]
ARMulator V1.20, 512 Kb RAM, MMU present, Demon 1.01, FPE, Little
endian.
Object program file randtest
armsd: go
randomnumber() returned 0b3a9965
randomnumber() returned ac0b1672
randomnumber() returned 6762ad4f
randomnumber() returned 1965a731
randomnumber() returned d6c1cef4
randomnumber() returned f78fa802
randomnumber() returned 8147fc15
randomnumber() returned 3f62adfc
randomnumber() returned b56e9da8
randomnumber() returned b36dc5e2
Program terminated normally at PC = 0x000082c8
      0x000082c8: 0xef000011 .... : >  swi     0x11
armsd: quit
Quitting
>
```

5.7 Loading a Word from an Unknown Alignment

This section describes a code sequence which loads a word from memory at any byte alignment. Although loading 32-bit data from non word-aligned addresses should be avoided whenever possible, it may sometimes be necessary.

5.7.1 Aligned and misaligned data

The ARM Load and Store (single and multiple) instructions are designed to load word-aligned data. Unless there is a very good reason for doing so, it is best not to have data at non word-aligned addresses, as neither the Load or Store instruction can access such data unaided.

To deal with misaligned word fetches, two words must be read and the required data extracted from these two words. The code below performs this operation for a little-endian ARM, making good use of the barrel shifter:

```
; enter with address in R0
; R2 and R3 are used as temporary registers
; the word is loaded into R1
;
    BIC    R2, R0, #3         ; Get word-aligned address
    LDMIA  R2, {R1, R3}       ; Get 64 bits containing data
    AND    R2, R0, #3         ; Get offset in bytes
    MOVS   R2, R2, LSL #3     ; Get offset in bits
    MOVNE  R1, R1, LSR R2     ; Extract data from bottom 32 bits
    RSBNE  R2, R2, #32        ; Get 32 - offset in bits
    ORRNE  R1, R1, R3, LSL R2 ; Extract data from top 32 bits
                              ; and combine with the other data
```

This code can easily be modified for use on a big-endian ARM; the `LSR R2` and `LSL R2` must be swapped over.

For details of what the Load and Store instructions do if used with non word-aligned addresses refer to the appropriate datasheet. Note that non-word-aligned word loads are also used in ➲*5.5 Using 16-bit Data on the ARM* on page 5-17.

5.8 Byte Order Reversal

Changing the endianness of a word can be a common operation in certain applications—for example when communicating word-sized data as a stream of bytes to a receiver of the opposite endianness.

This section describes a compact ARM Instruction Sequence to perform byte order reversal; ie. reversing the endianness of a word.

This operation can be performed efficiently on the ARM, using just four instructions. The word to be reversed is held in `a1` both on entry and exit of this instruction sequence. `ip` is used as a temporary register (For more information about these register names see ➲*Table 7-1: ACPS Registers* on page 7-4):

```
    EOR    ip, a1, a1, ror #16
    BIC    ip, ip, #&ff0000
    MOV    a1, a1, ror #8
    EOR    a1, a1, ip, lsr #8
```

A demonstration program which should help explain how this works has been provided in source form in directory `examples/explasm`. To compile this program and run it under armsd, first copy `bytedemo.c` from directory `examples/explasm` to your current working directory, and then use the following commands:

```
>armcc bytedemo.c -o bytedemo -li -apcs 3/32bit
>armsd -li bytedemo
A.R.M. Source-level Debugger, version 4.10 (A.R.M.) [Aug 26 1992]
ARMulator V1.20, 512 Kb RAM, MMU present, Demon 1.01, FPE, Little
endian.
Object program file bytedemo
armsd: go
```

Note *This program uses ANSI control codes, so should work on most terminal types under Unix and also on the PC. It will not work on HP-UX if the terminal emulator used is an HPTERM. An XTERM should be used to run this program on the HP-UX.*

5.9 ARM Assembly Programming Performance Issues

This section outlines many performance-related issues of which the ARM Assembly Language Programmer should be aware. It also provides useful background for C programmers using armcc, as some of these issues can also apply to programming in C.

Not all of the issues discussed here apply to every ARM-processor-based system. However, unless otherwise stated, all relate to processors based on ARM6 and ARM7, with or without cache and/or write buffer.

5.9.1 LDM / STM

Use `LDM` and `STM` instead of a sequence of `LDR` or `STR` instructions wherever possible. This provides several benefits:

- The code is smaller (and thus will cache better on an ARM processor with a cache)
- An instruction fetch cycle and a register copy back cycle is saved for each `LDR` or `STR` eliminated
- On an uncached ARM processor (for `LDM`) or an unbuffered ARM processor (for `STM`), non-sequential memory cycles can be turned into faster memory sequential cycles

5.9.2 Conditional execution

In many situations, branches around short pieces of code can be avoided by using conditionally executed instructions. This reduces the size of code and may avoid a pipeline break.

5.9.3 Using the barrel shifter

Combining shift operations with other operations can significantly increase the code density (and thus performance) of much ARM code.

5.9.4 Addressing modes

The ARM instruction set provides a useful selection of addressing modes, which can often be used to improve the performance of code; eg. using `LDR` or `STR` pre- or post-indexed with a non-zero offset increments the base register and performs the data transfer. For full details of the addressing modes available, refer to the appropriate ARM datasheet.

5.9.5 Multiplication

Be aware of the time taken by the ARM multiply and multiply accumulate instructions.

When multiplying by a constant value note that using the multiply instruction is often not the optimal solution. The issues involved are discussed in the ➲*5.3 Multiplication by a Constant* on page 5-8.

5.9.6 Optimising register usage

Examine your code and see if registers can be reused for another value during parts of a long calculation which uses many registers. Doing this will reduce the amount of 'register spillage' (ie. the number of times a value has to be reloaded or an intermediate value saved and then reloaded).

Because much can be achieved in a single data-processing instruction, keeping a calculated result in a register for use a considerable time later may be less efficient than recalculating it when it is next needed. This is because it may allow the freed register to be used for another purpose in the meantime, thus reducing the amount of register spillage.

5.9.7 Loop unrolling

Loop unrolling can be a useful technique, but detailed analysis is often necessary before using it. in some situations can reduce performance.

Loop unrolling involves using more than one copy of the inner loop of an algorithm. The following benefits may be gained by loop unrolling:

- the branch back to the beginning of the loop is executed less frequently
- it may be possible to combine some of one iteration with some of the next iteration, and thereby significantly reduce the cost of each iteration.
  A common case of this is combining `LDR` or `STR` instructions from two or more iterations into single `LDM` or `STM` instructions. This reduces code size, the number of instruction fetches, and in the case of `LDM`, the number of register writeback cycles.

As an example to illustrate the issues involved in loop unrolling, consider calculating the following over an array: x[i] = y[i] - y[i+1]. Below is a code fragment which performs this:

```
    LDR    R2, [R0]                 ; Preload y[i]
Loop
    LDR    R3, [R0, #4]!!           ; Load y[i+1]
    SUB    R2, R2, R3               ; x[i] = y[i] - y[i+1]
    STR    R2, [R1], #4             ; Store x[i]
    MOV    R2, R3                   ; y[i+1] is the next y[i]
    CMP    R0, R4                   ; Finished ?
    BLT    Loop
```

First examine the number of execution cycles this will take on an ARM6 based processor, where:

IF	stands for Instruction Fetch
WB	stands for Register Write Back
R	stands for Read
W	stands for Write

The loop will execute in the following cycles: 6 IF, 1 R, 1 WB, 1 W, and the branch costs an additional 2 IF cycles. Therefore the total cycle count for processing a 100 element x[] array is:

```
799 IF (198 caused by branching), 101 R, 101 WB, 100 W (1198 cycles)
Code size: 7 instructions
```

The effects of unrolling the loop

1. Branch overhead cycles
   In the above example there are 198 `IF`s caused by branching. Unrolling the loop can clearly reduce this, and the table below shows how progressively unrolling the loop gives reducing returns for the increase in code size. If code size is an issue of any importance, unrolling any more than around 3 times is unlikely to pay off with regard to branch overhead elimination:

Times unrolled	IFs caused by branching	IF saving
2	98	100
3	66	134
4	48	150
5	38	160
10	18	180
100	0	198

Table 5-3: Effects of loop unrolling on instruction fetches

2. Combining `LDR`s and `STR`s into `LDM` and `STM`
   The number of `LDR`s or `STR`s which can be combined into a single `LDM` or `STM` is restricted by the number of available registers. In this instance 10 registers is the most which are likely to be usable. This would result in unrolling the loop 10 times for the above example. Another case to consider is unrolling 3 times, as this seems to be a good compromise for branch overhead reduction.

3. Other optimisations
   Upon examining the unrolled code below, it can be seen that it is only necessary to execute the `MOV` once per loop, thus saving another 2 IF cycles per loop for the 3 times unrolled code, and another 9 IF cycles per loop for the 10 times unrolled code.

Unrolling the loop three times

Here is the code unrolled three times and then optimised as described above:

```
    LDR     R2, [R0], #4            ; Preload y[i]
Loop
    LDMIA   R0!, {R3-R5}            ; Load y[i+1] to y[i+3]
    SUB     R2, R2, R3              ; x[i]   = y[i]   - y[i+1]
    SUB     R3, R3, R4              ; x[i+1] = y[i+1] - y[i+2]
    SUB     R4, R4, R5              ; x[i+2] = y[i+2] - y[i+3]
    STMIA   R1!, {R2-R4}            ; Store x[i] to x[i+2]
    MOV     R2, R5                  ; y[i+3] is the next y[i]
    CMP     R0, R6                  ; Finished ?
    BLT     Loop
```

Analysing how this code executes for a y[] array of size 100, as described above for the unrolled code produces the following results:

```
339 IF (66 caused by branching), 101 R, 34 WB, 100 W (574 cycles)
Code size: 9 instructions
Saving over unrolled code: 460 IF, 67 WB
```

Unrolling the loop ten times

Here is the code unrolled ten times and then optimised in the same way:

```
    LDR     R2, [R0], #4            ; Preload y[i]
Loop
    LDMIA   R0!, {R3-R12}           ; Load y[i+1] to y[i+10]
    SUB     R2,  R2,  R3            ; x[i]   = y[i]   - y[i+1]
    SUB     R3,  R3,  R4            ; x[i+1] = y[i+1] - y[i+2]
    SUB     R4,  R4,  R5            ; x[i+2] = y[i+2] - y[i+3]
    SUB     R5,  R5,  R6            ; x[i+3] = y[i+3] - y[i+4]
    SUB     R6,  R6,  R7            ; x[i+4] = y[i+4] - y[i+5]
    SUB     R7,  R7,  R8            ; x[i+5] = y[i+5] - y[i+6]
    SUB     R8,  R8,  R9            ; x[i+6] = y[i+6] - y[i+7]
    SUB     R9,  R9,  R10           ; x[i+7] = y[i+7] - y[i+8]
    SUB     R10, R10, R11           ; x[i+8] = y[i+8] - y[i+9]
    SUB     R11, R11, R12           ; x[i+9] = y[i+9] - y[i+10]
    STMIA   R1!, {R2-R11}           ; Store x[i] to x[i+9]
    MOV     R2,  R12                ; y[i+10] is the next y[i]
    CMP     R0,  R13                ; Finished ?
    BLT     Loop
```

Analysing how this code executes for a y[] array of size 100, produces the following results:

```
169 IF (18 caused by branching), 101 R, 10 WB, 100 W (380 cycles)
Code size: 16 instructions
Saving over unrolled code: 630 IF, 91 WB
```

Thus for this problem, unless the extra seven instructions make the code too large unrolling ten times is likely to be the optimum solution.

Where loop unrolling is not appropriate

Loop unrolling is not always a good idea, especially when the optimisation between one iteration and the next is small. Consider the following loop which copies an area of memory:

```
Loop
    LDMIA  R0!,{R3-R14}
    STMIA  R1!,{R3-R14}
    CMP    R0, #LimitAddress
    BNE    Loop
```

This could be unrolled as follows:

```
Loop
    LDMIA  R0!,{R3-R14}
    STMIA  R1!,{R3-R14}
    LDMIA  R0!,{R3-R14}
    STMIA  R1!,{R3-R14}
    LDMIA  R0!,{R3-R14}
    STMIA  R1!,{R3-R14}
    LDMIA  R0!,{R3-R14}
    STMIA  R1!,{R3-R14}
    CMP    R0, #LimitAddress
    BLT    Loop
```

In this code the `CMP` and `BNE` will be executed only a quarter as often, but this will give only a small saving. However, other issues should be taken into account:

- If in the above case the amount of data to be transferred was not a multiple of 48, then this amount of loop unrolling will copy too much data. This may be catastrophic, or may merely be inefficient.
- On a cached ARM processor, the larger the inner loop, the more likely it is that the loop will not stay entirely in the cache. In this case, it is not obvious at what point the performance gain due to unrolling is offset by the performance loss due to cache misses, or the disadvantage of larger code.
- On an ARM processor with a write buffer, the loop unrolling in the above example is unlikely to help. If the data being copied is not in the cache, then every `LDMIA` will be stalled while the write buffer empties. Thus the time the `CMP` and `BNE` take is irrelevant, as the processor will be stalled on the following `LDMIA`.

5.9.8 The floating-point emulator

Note *This advice is not applicable to systems which use the ARM FPA co-processor nor to code using the software floating-point library.*

If the software-only floating-point emulator is being used, floating-point instructions should placed sequentially, as the floating-point emulator will detect that the next instruction is also a floating-point instruction, and will emulate it without leaving the undefined instruction code.

5.9.9 Stalling the write buffer

On ARM processors with a write buffer, performance can be maximised by writing code which avoids stalling due to the write buffer. For a write buffer with 2 tags and 8 words such as the ARM610, no three `STR` or `STM` instructions should be close together (as the third will be stalled until the first has finished). Similarly no two `STR` or `STM` instructions which together store more than 8 words should be close together, as the second will be stalled until there is space in the write buffer.

Rearranging code so that the write buffer does not cause a stall in this way is often hard, but is frequently worth the effort, and in any case it is always wise to be aware of this performance factor.

5.9.10 16-bit data

Note *The following applies to ARM Architecture 3 only.*

If possible treat 16-bit data as 32-bit data. However, if this cannot be done, be aware that you can make use of the barrel shifter and non-word-aligned `LDR`s in order to make working with 16-bit data more efficient. See *5.5 Using 16-bit Data on the ARM* on page 5-17 for a full discussion of this topic.

5.9.11 8-bit data

When processing a sequence of byte-sized objects (eg. strings), the number of loads and stores can be reduced if the data is loaded a word at a time and then processed a byte at a time by extracting the bytes using the barrel shifter.

5.9.12 Making full use of cache lines

In order to help the cache on a cached ARM processor maintain a high hit rate for data, place frequently accessed data values together so that they are loaded into the same cache line, rather scattering them around memory, as this will require more cache lines to be loaded and kept in the cache.

Commonly-used subroutines (especially short ones) will often run more quickly on a cached ARM processor if the entry address is aligned so that it will be loaded into the first word of a cache line. For example, on the ARM610 this means quad-word-aligned. This ensures that all four words of the first line fetch will be subsequently used by instruction fetches before another line fetch is caused by an instruction fetch.

This technique is most useful for large programs which do not cache well, as the number of times the code will have to be fetched from memory is not likely to be significant if the program does cache well.

5.9.13 Minimising non-sequential cycles

Note *This technique is only appropriate to uncached ARM processors, and is intended for memory systems in which non-sequential memory accesses take longer than sequential memory accesses.*

Consider a system where the length of memory bursts is B. That is, if executing a long sequence of data operations, the memory accesses which result are: one non-sequential memory cycle followed by B – 1 sequential memory cycles. An example of this is DRAM controlled by the ARM memory manager MEMC1a.

This sequence of memory accesses will be broken up by several ARM instruction types:

- Load or Store (single or multiple)
- Data Swap, Branch instructions
- SWIs
- Other instructions which modify the PC

By placing these instructions carefully, so that they break up the normal sequence of memory cycles only where a non-sequential cycle was about to occur anyway, the number of sequential cycles which are turned into longer non-sequential cycles can be minimised.

For a memory system which has memory bursts of length B, the optimal position for instructions which break up the memory cycle sequence is 3 words before the next B-word boundary.

To help explain this, consider a memory system with memory bursts of length 4 (ie. quad-word bursts), the optimal position for these break-up instructions is 16-12=4 bytes from a quad-word offset.

The following code demonstrates this:

```
0x0000  Data Instr 1
0x0004  STR
0x0008  Data Instr 2
0x000C  Data Instr 3
0x0010  Data Instr 4
```

Taking into account the ARM instruction pipeline, the memory cycles executing this code will produce:

```
Instruction Fetch 0x0000 (Non Seq)
Instruction Fetch 0x0004 (Seq)
Instruction Fetch 0x0008 (Seq)      + Execute Data Instr 1
Instruction Fetch 0x000C (Seq)      + Execute STR
Data Write               (Non Seq)
Instruction Fetch 0x0010 (Non Seq)  + Execute Data Instr 2
Instruction Fetch 0x0014 (Seq)      + Execute Data Instr 3
```

The instruction fetch after the Data Write cycle had to be non-sequential cycle, but since the instruction fetch was of a quad-word-aligned address, it had to be non-sequential anyway. Therefore, the STR is optimally positioned to avoid changing sequential instruction fetches into non-sequential instruction fetches.

6 Programming in C for the ARM

This chapter explains some techniques for optimising the output of the ARM C compiler, and gives details of how to build code for deeply embedded applications.

Programming in C for the ARM

6.1 Introduction

This chapter:

- explains how you can maximise the efficiency of your applications by writing your C source in such a way as to make the compiler generate fast and compact machine code
- gives advice on which command line switches to use with the C compiler to optimise the resulting program, and shows you how to identify and eliminate unused sections of code
- describes how to compile and link C code for deeply embedded applications, using the components of the standalone C runtime system

You can find related information in ➲*Chapter 13, Benchmarking, Performance Analysis, and Profiling*.

A full description of the ARM C compiler is given in ➲*The ARM Software Development Toolkit Reference Manual: Chapter 2, C Compiler*.

6.2 Writing Efficient C for the ARM

The ARM C compiler can generate very good machine code for if you present it with the right sort of input. In this section we explain:

- what the C compiler compiles well and why
- how to help the C compiler generate efficient machine code

Some of the rules presented here are quite general; some are quite specific to the ARM or the ARM C compiler. It should be clear from context which rules are portable.

When writing C, there are a number of considerations which, if handled intelligently, will result in more compact and efficient ARM code:

- The way functions are written, their size, and the way in which they call each other.

  This is discussed in ➲*6.2.1 Function design*, below.
- The distribution of variables within functions, and their scoping.

  This affects the register allocation of variables, and the frequency with which they are spilled to memory: see ➲*6.2.2 Register allocation and how to help it* on page 6-6.
- The use of alternatives to the `switch()` statement

  Under certain circumstances, reductions in code size can be achieved by avoiding the use of `switch()`, as discussed in ➲*6.2.4 The switch() statement* on page 6-8.

6.2.1 Function design

Function call overhead on the ARM is small, and is often in proportion to the work done by the called function. Several features contribute to this:

- the minimal ARM call-return sequence is `BL... MOV pc, lr`, which is extremely economical
- the multiple load and store instructions, `STM` and `LDM,` which reduce the cost of entry to and exit from functions that must create a stack frame and/or save registers
- the ARM Procedure Call Standard, which has been carefully designed to allow two very important types of function call to be optimised so that the entry and exit overheads are minimal.

In general, it is a good idea to keep functions small, because this will help keep function calling overheads low. This section describes the conditions under which function call overhead is minimised, how small functions help the ARM C compiler, and explains how to assist the C compiler when functions cannot be kept small.

Leaf functions

In 'typical' programs, about half of all function calls made are to leaf functions (a *leaf function* is one that makes no calls from within its body).

Often, a leaf function is rather simple. On the ARM, if it is simple enough to compile using just five registers (`a1-a4` and `ip`), it will carry no function entry or exit overhead. A surprising proportion of useful leaf functions can be compiled within this constraint.

Once registers have to be saved, it is efficient to save them using STM. In fact, the more you can save at one go the better. In a leaf function, all the registers which need to be saved will be saved by a single STMFD sp!,{regs,lr} on entry and a matching LDMFD sp!,{regs,pc} on exit.

In general, the cost of pushing some registers on entry and popping them on exit is very small compared to the cost of the useful work done by a leaf function that is complicated enough to need more than five registers.

Overall, you should expect a leaf function to carry virtually no function entry and exit overhead, and at worst, a small overhead, most likely in proportion to the useful work done by it.

Veneer Functions (Simple Tail Continued Functions)

Historically, abstraction veneers have been relatively expensive. The kind of veneer function which merely changes the types of its arguments, or which calls a low-level implementation with an extra argument (say), has often cost much more in entry and exit overhead than it was worth in useful work.

On the ARM, if a function ends with a call to another function, that call can be converted to a *tail continuation*. In functions that do not need to save any registers, the effect can be dramatic. Consider, for example:

```
extern void *__sys_alloc(unsigned type, unsigned n_words);
#define  NOTGCable   0x80000000
#define  NOTMovable  0x40000000

void *malloc(unsigned n_bytes)
{   return __sys_alloc(NOTGCable+NOTMovable, n_bytes/4);
}
```

From this input, armcc generates:

```
malloc
    MOV     a2,a1,LSR #2
    MOV     a1,#&c0000000
    B       |__sys_alloc|
```

(Note that your version of armcc may produce slightly different code.)

Here there is no function entry or exit overhead, and the function return has disappeared entirely —return is direct from __sys_alloc to malloc's caller. In this case, the basic call-return cost for the function pair has been reduced from:

```
 BL + BL + MOV pc,lr + MOV pc,lr
```

to:

```
 BL + B  +             MOV pc,lr
```

which works out as a saving of 25%.

More complicated functions in which the only function calls are immediately before a return, collapse equally well. An artificial example is:

```
extern int f1(int), int f2(int, int);

int f(int a, int b)
{   if (b == 0)
        return a;
    else if (b < 0)
        return f2(a, -b);
    else
        return f2(b, a);  /* argument order swapped */
}
```

armcc generates the following, extremely efficient code (the version of armcc supplied with your release may produce slightly different output):

```
f   CMP     a2,#0
    MOVEQS  pc,lr
    RSBLT   a2,a2,#0
    BLT     f2
    MOV     a3,a1
    MOV     a1,a2
    MOV     a2,a3
    B       f2
```

Function arguments and argument passing

The final aspect of function design which influences low-level efficiency is argument passing.

Under the ARM Procedure Call Standard, up to four argument words can be passed to a function in registers. Functions of up to four integral (not floating point) arguments are particularly efficient and incur very little overhead beyond that required to compute the argument expressions themselves (there may be a little register juggling in the called function, depending on its complexity).

If more arguments are needed, then the 5th, 6th, etc., words will be passed on the stack. This incurs the cost of an STR in the calling function and an LDR in the called function for each argument word beyond four.

To minimise argument passing:

- Try to ensure that small functions take four or fewer arguments. These will compile particularly well.
- If a function needs many arguments, try to ensure that it does a significant amount of work on every call, so that the cost of passing arguments is amortised.
- Factor out read-mostly global control state and make this static. If it has to be passed as an argument (to support multiple clients, for example), wrap it up in a struct and pass a pointer to it.

Such control state is:

- logically global to the compilation unit or program
- read-mostly, often read-only except in response to user input, and for almost all functions cannot be changed by them or any function called from them
- referenced throughout the program, but relatively rarely in any given function. Frequent references inside a function should be replaced by references to a local, non-static copy.

Note *Don't confuse control state with computational arguments, the values of which differ on every call.*

- Collect related data into structs. Decide whether to pass pointers or struct values based on the use of each struct in the called function:
    - If few fields are read or written, passing a pointer is best.
    - The cost of passing a struct via the stack is typically a share in an `LDM-STM` pair for each word of the struct. This can be better than passing a pointer if on average each field is used at least once, and the register pressure in the function is high enough to force a pointer to be repeatedly re-loaded.

  As a general rule, you cannot lose much efficiency if you pass pointers to structs rather than struct values. To gain efficiency by passing struct values rather than pointers usually requires careful study of a function's machine code.

6.2.2 Register allocation and how to help it

It is well known that register allocation is critical to the efficiency of code compiled for RISC processors. It is particularly critical for the ARM, which has only 16 registers rather than the 'traditional' 32.

The ARM C compiler has a highly efficient register allocator which operates on complete functions and which tries to allocate the most frequently used variables to registers (taking loop nesting into account). It produces very good results unless the demand for registers seriously outstrips supply.

As code generation proceeds for a function, new variables are created for expression temporaries. These are never reused in later expressions and cannot be spilled to memory. Usually, this causes no problems. However, a particularly pathological expression could, in principle, occupy most of the allocatable registers, forcing almost all program variables to memory. Because the number of registers required to evaluate an expression is a logarithmic function of the number terms in it, it takes an expression of more than 32 terms to threaten the use of any variable register.

As a general rule, avoid very large expressions (more than 30 terms).

6.2.3 Static and extern variables—minimising access costs

A variable in a register costs nothing to access: it is simply there, waiting to be used. A local (auto) variable is addressed via the `sp` register, which is always available for the purpose.

A static variable, on the other hand, can only be accessed after the static base for the compilation unit has been loaded. So the first such use in a function always costs two `LDR` instructions or an `LDR` and an `STR`. However, if there are many uses of static variables within a function, there is a good chance that the static base will become a global common subexpression (CSE) and that, overall, access to static variables will be no more expensive than to auto variables on the stack.

Extern variables are fundamentally more expensive: each has its own base pointer. Thus each access to an extern is likely to cost two `LDR` instructions or an `LDR` and an `STR`. It is much less likely that a pointer to an extern will become a global CSE—and almost certain that there cannot be several such CSEs—so if a function accesses lots of extern variables, it is bound to incur significant access costs.

A further cost occurs when a function is called: the compiler has to assume—in the absence of inter-procedural data flow analysis—that *any* non- const static or extern variable *could* be side-effected by the call. This severely limits the scope across which the value of a static or extern variable can be held in a register.

Sometimes a programmer can do better than a compiler could do, even a compiler that did interprocedural data flow analysis. An example in C is given by the standard streams: stdin, stdout and stderr. These are not pointers to const objects (the underlying `FILE` structs are modified by I/O operations), nor are they necessarily const pointers (they may be assignable in some implementations). Nonetheless, a function can almost always safely slave a reference to a stream in a local `FILE *` variable.

It is a common practice to mimic the standard streams in applications. Consider, for example, the shape of a typical non-leaf printing function:

```
extern FILE *out;                  extern FILE *out;
    /* the output stream */            /* the output stream */

void print_it(Thing *t)            void print_it(Thing *t)
{                                  {   FILE *f = out;
    fprintf(out, ...);                 fprintf(f, ...);
    print_1(t->first);                 print_1(t->first);
    fprintf(out, ...);                 fprintf(f, ...);
    print_2(t->second);                print_2(t->second);
    fprintf(out, ...);                 fprintf(f, ...);
    ...                                ...
}                                  }
```

In the left-hand case, out has to be re-computed or re-loaded after each call to print_... (and after each fprintf...). In the right-hand case, f can be held in a register throughout the function (and probably will be).

Uniform application of this transformation to the disassembly module of the ARM C compiler saved more than 5% of its code space.

In general, it is difficult and potentially dangerous to assert that no function you call (or any functions they in turn call) can affect the value of any static or extern variables of which you currently have local copies. However, the rewards can be considerable so it is usually worthwhile to work out at the program design stage which global variables are slavable locally and which are not. Trying to retrofit this improvement to existing code is usually hazardous, except in very simple cases like the above.

6.2.4 The switch() statement

The switch() statement can be used:

1. to transfer control to one of several destinations—effectively implementing an indexed transfer of control
2. to generate a value related to the controlling expression—in effect computing an in-line function of the controlling expression

In the first role, switch() is hard to improve upon: the ARM C compiler does a good job of deciding when to compile jump tables and when to compile trees of if-then-elses. It is rare for a programmer to be able to improve upon this by writing if-then-else trees explicitly in the source.

In the second role, however, use of switch() is often mistaken. You can probably do better by being more aware of what is being computed and how.

The example below is a simplified version of a function taken from an early version of the ARM C compiler’s disassembly module. Its purpose is to map a 4-bit field of an ARM instruction to a 2-character condition code mnemonic. We will use it to demonstrate:

- the cost of implementing an in-line function using switch()
- how to implement the same function more economically

Here is the source:

```
char *cond_of_instr(unsigned instr)
{   char *s;‘
    switch (instr & 0xf0000000)
    {
case 0x00000000:  s = "EQ";  break;
case 0x10000000:  s = "NE";  break;
     ...          ...        ...
case 0xF0000000:  s = "NV";  break;
```

```
    }
    return s;
}
```

The compiler handles this code fragment well, generating 276 bytes of code and string literals. But we could do better. If performance is not critical (and in disassembly it never is) we could look up the code in a table, using something like:

```
char *cond_of_instr(unsigned instr)
{
    static struct {char name[3];  unsigned code;}
        conds[] = {
            "EQ", 0x00000000,
            "NE", 0x10000000,
            ....
            "NV", 0xf0000000,
        };
    int j;
    for (j = 0;  j < sizeof(conds)/sizeof(conds[0]);  ++j)
        if ((instr & 0xf0000000) == conds[j].code)
            return conds[j].name;
    return "";
}
```

This fragment compiles to 68 bytes of code and 128 bytes of table data. Already this is a 30% improvement on the `switch()` case, but this schema has other advantages: it copes well with a random code-to-string mapping and, if the mapping is not random, admits further optimisation. For example, if the code is stored in a byte (char) instead of an unsigned and the comparison is with `(instr >> 28)` rather than `(instr & 0xF0000000)`then only 60 bytes of code and 64 bytes of data are generated for a total of 124 bytes.

Another advantage for table lookup is that is possible to share the same table between a disassembler and an assembler—the assembler looks up the mnemonic to obtain the code value, rather than the code value to obtain the mnemonic. Where performance is not critical, the symmetric property of lookup tables can sometimes be exploited to yield significant space savings.

Finally, by exploiting the denseness of the indexing and the uniformity of the returned value it is possible to do better again, both in size and performance, by direct indexing:

```
char *cond_of_instr(unsigned instr)
{
    return "\
EQ\0\0NE\0\0CC\0\0CS\0\0MI\0\0PL\0\0VS\0\0VC\0\0\
HI\0\0LS\0\0GE\0\0LT\0\0GT\0\0LE\0\0AL\0\0NV" + (instr >> 28)*4;
}
```

This expression of the problem causes a miserly 16 bytes of code and 64 bytes of string literal to be generated and is probably close to what an experienced assembly language programmer would naturally write if asked to code this function. This was the solution finally adopted in the ARM C compiler's disassembler module.

The uniform application of this transformation to the disassembler module of the ARM C compiler saved between 5% and 10% of its code space.

You should therefore think hard before using `switch()` to compute an in-line function, especially if code size is an important consideration. Although `switch()` compiles to high-performance code, table lookup will often be smaller; where the function's domain is dense (or piecewise dense) direct indexing into a table will often be both faster and smaller.

6.3 Improving Code Size and Performance

Once you have optimised your source code, you may be able to obtain further performance benefits by using the appropriate command line options when you come to compile and link your program.

This section gives advice on which options to use and which to avoid, and explains how to identify and remove unused functions from your C source.

6.3.1 Compiler options

The ARM C compiler has a number of command line options which control the way in which code is generated. You can find a full list in ➲*The ARM Software Development Toolkit Reference Manual: Chapter 2, C Compiler*. There are a number of compiler options which can affect the size and/or the performance of generated code.

–g

`-g` severely impacts the size and performance of generated code, since it turns off all compiler optimisations. You should use it only when actually debugging your code, and it should never be enabled for a release build.

–Ospace -Otime

These options are complementary:

`-Ospace`	optimises for code size at the expense of performance
`-Otime`	optimises for performance at the expense of size

They can be used together on different parts of a build. For example, `-Otime` could be used on time critical source files, with `-Ospace` being used on the remainder.

If neither is specified, the compiler will attempt a balance between optimising for code size and optimising for performance.

–zpj0

This disables *crossjump optimisation*. Crossjump optimisation is a space-saving optimisation whereby common sections of code at the end of each element in a `switch()` statement are identified and commoned together, each occurrence of the code section being replaced with a branch to the commoned code section.

However, this optimisation can lead to extra branches being executed which may decrease performance, especially in interpreter-like applications which typically have large `switch()` statements.

Use the `-zpj0` option to disable this optimisation if you have a time-critical `switch()` statement.

Alternatively, you can use:

```
#pragma nooptimise_crossjump
```

before the function containing the `switch()` and:

```
#pragma optimise_crossjump
```

after the function to re-enable the optimisation for the remainder of the functions in the file.

–apcs /nofp

By default, armcc generates code which uses a dedicated frame pointer register. This register holds a pointer to the stack frame and is used by the generated code to access a function's arguments.

A dedicated frame pointer can make the code slightly larger. By specifying `-apcs/nofp` on the command line, you can force armcc to generate code which does not use a frame pointer, but which accesses the function's arguments via offsets from the stack pointer instead.

Note tcc never uses a frame pointer, so this option does not apply when compiling Thumb code.

–apcs /noswst

By default, armcc generates code which checks that the stack has not overflowed at the head of each function. This code can contribute several percent to the code size, so it may be worthwhile disabling this option with `-apcs /noswst`.

Be careful, however, to ensure that your program's stack is not going to overflow, or that you have an alternative stack checking mechanism such as an MMU-based check.

Note tcc has stack checking disabled by default.

–ARM7T

This option applies to armcc only.

By default, armcc generates code which is suitable for running on processors that implement ARM Architecture 3 (eg. ARM6, ARM7). If you know that the code is going to be run on a processor with halfword support, you can use the `-ARM7T` option to instruct the compiler to use the ARM Architecture 4 halfword and signed byte instructions. This can result in significantly improved code density and performance when accessing 16-bit data.

–pcc

The code generated by the compiler can be slightly larger when compiling with the `-pcc` switch. This is because of extra restrictions on the C language in the ANSI standard which the compiler can take advantage of when compiling in ANSI mode.

Note If your code will compile in ANSI mode, do not use the `-pcc` switch.

6.3.2 Identifying and eliminating unused code sections

During program development it can happen that functions which were used at an earlier stage of development are no longer called, and may therefore be eliminated.

The compiler and linker provide facilities to enable you to identify and eliminate such functions. This section explains how these facilities work, taking as an example the `unused.c` program in directory `examples/unused.`

Stage 1—allocating a code segment to each function

The first stage in eliminating unused functions is to compile all your sources with the `-zo` option. This instructs the compiler to place each function in a separate code segment:

```
armcc -c -zo unused.c
```

Stage 2—removing unreferenced segments

The second stage is to instruct the linker to remove those segments which are unreferenced by code in any other segment. This is done with the `-Remove` command line option. You can also specify `-info unused` to get the linker to tell you which code segments are unused:

```
armlink -info unused -remove -o unused unused.o armlib.32l
```

Note If armlib.32l is not in the current directory, you will need to specify the full pathname.

In this instance, the linker will produce the following output:

```
ARM Linker: Unreferenced AREA unused.o(C$$code) (file unused.o)
omitted from output.
```

Stage 3—identifying unused functions

The linker has removed the unused function from the output file. If you wish to find the name of this function so that you can remove it from your source, instruct the compiler to generate assembler output with the `-S` option.

```
armcc -c -zo -S unused.c
```

Edit the assembler output file `unused.s` and search for the AREA name which was given in the linker output (in this case `C$$code`). This will give the name of the unused function, as shown in the following extract from the file:

```
        AREA |C$$code|, CODE, READONLY
|x$codeseg| DATA

unused_function                       <<<< Name of unused function
        ADD      a1,pc,#L000008-.-8
        B        _printf
L000008
```

6.4 Choosing a Division Implementation

In some applications it is important that a general-purpose divide executes as quickly as possible. This section shows how to choose between different divide implementations for the ARM.

This section describes:

- how to select a divide implementation for the C library
- how to use the fast divide routines from the `examples` directory
- a comparison of the speeds of the divide routines

6.4.1 Divide implementations in the C library

The C library offers a choice between two divide variants. This choice is basically a speed vs. space tradeoff. The options are:

- unrolled
- small

In the C library build directory (eg. directory `semi` for the semi-hosted library), the `options` file is used to select variants of the C library.

The supplied file contains the following:

```
memcpy = fast
divide = unrolled
stdfile_redirection = off
fp_type = module
stack = contiguous
backtrace = off
thumb = false
```

Unrolled divide

The default divide implementation 'unrolled' is fast, but occupies a total of 416 bytes (55 instructions for the signed version plus 49 instructions for the unsigned version). This is an appropriate default for most Toolkit users who are interested in obtaining maximum performance.

Small divide

Alternatively you can change this file to select 'small' divide which is more compact at 136 bytes(20 instructions for signed plus 14 instructions for unsigned) but somewhat slower, as there is considerable looping overhead.

For a comparison of the speed difference between these two routines, see *Table 6-1: Signed division example timings* on page 6-15 (the speed of divide is data-dependent).

Signed division example timings

Cycle times are F-cycles on a cached ARM6 series processor excluding call and return

Calc	Unrolled cycles	Rolled cycles
0/100	22	19
9/7	22	19
4096/2	70	136
1000000/3	99	240
1000000000/1	148	370

Table 6-1: Signed division example timings

If you have a specific requirement, you can modify the supplied routines to suit your application. For instance, you could write an unrolled-2-times version or you could combine the signed and unsigned versions to save more space.

6.4.2 Guaranteed-performance divide routines for real-time applications

The C library also contains two fully unwound divide routines. These have been carefully implemented for maximum speed. They are useful when a guaranteed performance is required, eg. for real-time feedback control routines or DSP. The long maximum division time of the standard C library functions may make them unsuitable for some real-time applications.

The supplied routines implement signed division only; it would be possible to modify them for unsigned division if required. The routines are described by the standard header file "`stdlib.h`" which contains the following declarations:

ARM real-time divide functions for guaranteed performance

```
typedef struct __sdiv32by16 { int quot, rem; } __sdiv32by16;
/* used int so that values return in regs, although 16-bit */
typedef struct __sdiv64by32 { int rem, quot; } __sdiv64by32;

__value_in_regs extern __sdiv32by16 __rt_sdiv32by16(
     int /*numer*/,
     short int /*denom*/);
/*
 * Signed div: (16-bit quot), (16-bit rem) = (32-bit) / (16-bit)
 */
__value_in_regs extern __sdiv64by32 __rt_sdiv64by32(
     int /*numer_h*/, unsigned int /*numer_l*/,
     int /*denom*/);
/*
 * Signed div: (32-bit quot), (32-bit rem) = (64-bit) / (32-bit)
```

These routines both have guaranteed performance:

108 cycles for __rt_sdiv64by32 (excluding call & return)
44 cycles for __rt_sdiv32by16

Currently the C compiler does not automatically use these routines, as the default routines have early-termination which yields good performance for small values.

In order to use these new divide routines, you should explicitly call the relevant function. The __rt_div64by32 function is complicated by the fact that our C compiler does not currently support 64-bit integers, as you have to split a 64-bit value between two 32-bit variables.

The following example program shows how to use these routines. This is available as dspdiv.c in directory examples/progc. Once the program has been compiled and linked, type the following line to calculate 1000/3:

```
armsd dspdiv 1000 3
```

divdsp.c source code

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  if (argc != 3)
    puts("needs 2 numeric args");
  else
  {
    __sdiv32by16 result;

    result = __rt_sdiv32by16(atoi(argv[1]), atoi(argv[2]));

    printf("quotient %d\n", result.quot);
    printf("remainder %d\n", result.rem);
  }
  return 0;
}
```

6.4.3 Summary

The standard division routine used by the C library can be selected by using the options file in the C library build area. If the supplied variants are not suitable, you can write your own.

For real-time applications, the maximum division time must be as short as possible to ensure that the calculation can complete in time. In this case, the functions __rt_sdiv64by32 and __rt_sdiv32by16 are useful.

6.5 Using the C Library in Deeply Embedded Applications

This section discusses the Toolkit's standalone runtime support system for C programming in deeply embedded applications. In particular it explains:

- what `rtstand.s` supports
- how to make use of it
- how to extend it by adding extra functionality from the C library
- the size of the standalone run time library

6.5.1 The standalone runtime system

The semi hosted ANSI C library provides all the standard C library facilities and is thus quite large. This is acceptable when running under emulation with plenty of memory, or when running on development hardware with access to a real debugging channel. However, in a deeply embedded application many of its facilities—file access functions or time and date functions, for example—may no longer be relevant, and its size may be prohibitive if memory is severely limited.

For deeply embedded applications a minimal C runtime system is needed which takes up as little memory as possible, is easily portable to the target hardware, and only supports those functions that are required for such an application.

The ARM Software Development Toolkit comes with a minimal runtime system in source form. The 'behind the scenes' jobs which it performs are:

- setting up the initial stack and heap, and calling `main()`
- program termination—either automatic (returning from `main()`) or forced (explicitly calling `__rt_exit`)
- simple heap allocation (`__rt_alloc`)
- stack limit checking
- `setjmp` and `longjmp` support
- divide and remainder functions (calls to which can be generated by armcc)
- high level error handler support (`__err_handler`)
- optional floating point support, and a means to detect whether floating point support is available (`__rt_fpavailable`)

The source code `rtstand.s` documents the options which you may want to change for your target. These are not covered here. The header file `rtstand.h` documents the functions which `rtstand.s` provides to the C programmer.

A Thumb version of this file is located in `thumb/rtstand.s`.

Note No support is provided for outputting data down the debugging channel. This can be done, but is specific to the target application. The example C programs described below use the ARM Debug Monitor available under armsd to output messages using in-line SWIs. See ➲*The ARM*

Software Development Toolkit Reference Manual: Chapter 17, Demon for full details of the facilities provided by the Debug Monitor, and see also ➲*12.4 Calling SWIs from your Application* on page 12-11 for more information about in-line SWIs.

6.5.2 Using the standalone runtime system

In this section the main features of the standalone runtime system are demonstrated by example programs.

Before attempting any of the demonstrations below, proceed as follows:

1. Create a working directory, and make this your current directory.
2. Copy the contents of directory `examples/clstand` into your working directory.
3. Copy the files `fpe*.o` from directory `cl/fpe` into your working directory.

You are now ready to experiment with the C standalone runtime system.

In the examples below, the following options are passed to armcc, armasm, and in the first case armsd:

`-li` specifies that the target is a little endian ARM.

`-apcs 3/32bit/hardfp` specifies that the 32-bit variant of APCS 3 should be used. For armasm this is used to set the built-in variable {CONFIG} to 32.
ARM FPA instructions are used for floating point operations.

These arguments can be changed if the target hardware differs from this configuration, or omitted if your tools have been configured to have these options by default.

You may find it useful to study the sources to `rtstand.s`, `errtest.c` and `memtest.c` while working through the example programs.

6.5.3 A simple program

Let us first compile the example program `errtest.c`, and assemble the standalone runtime system. These can then be linked together to provide an executable image, `errtest`:

```
armcc -c errtest.c -li -apcs 3/32bit/hardfp
armasm rtstand.s -o rtstand.o -li -apcs 3/32bit
armlink -o errtest errtest.o rtstand.o
```

We can then execute this image using armsd as follows:

```
> armsd -li - size 512K errtest
A.R.M. Source-level Debugger, version 4.10 (A.R.M.) [Aug 26 1992]
ARMulator V1.20, 512 Kb RAM, MMU present, Demon 1.01, FPE, Little
endian.
Object program file errtest
armsd: go
(the floating point instruction-set is not available)
```

```
Using integer arithmetic ...
10000 / 0X0000000A = 0X000003E8
10000 / 0X00000009 = 0X00000457
10000 / 0X00000008 = 0X000004E2
10000 / 0X00000007 = 0X00000594
10000 / 0X00000006 = 0X00000682
10000 / 0X00000005 = 0X000007D0
10000 / 0X00000004 = 0X000009C4
10000 / 0X00000003 = 0X00000D05
10000 / 0X00000002 = 0X00001388
10000 / 0X00000001 = 0X00002710
Program terminated normally at PC = 0x00008550
      0x00008550: 0xef000011 .... : >  swi     0x11
armsd: quit
Quitting
>
```

The > prompt is the Operating System prompt, and the `armsd:` prompt is output by armsd to indicate that user input is required.

Already several of the standalone runtime system's facilities have been demonstrated:

- the C stack and heap have been set up
- `main()` has clearly been called
- the fact that floating point support is not available has been detected
- the integer division functions have been used by the compiler
- program termination was successful

6.5.4 Error handling

The same program, `errtest`, can also be used to demonstrate error handling, by recompiling `errtest.c` and predefining the DIVIDE_ERROR macro:

```
armcc -c errtest.c -li -apcs 3/32bit/hardfp -DDIVIDE_ERROR
armlink -o errtest errtest.o rtstand.o
```

Again, we can now execute this image under the armsd as follows:

```
> armsd -li -size 512K errtest
A.R.M. Source-level Debugger, version 4.10 (A.R.M.) [Aug 26 1992]
ARMulator V1.20, 512 Kb RAM, MMU present, Demon 1.01, FPE, Little
endian.
Object program file errtest
armsd: go
(the floating point instruction-set is not available)
Using integer arithmetic ...
10000 / 0X0000000A = 0X000003E8
10000 / 0X00000009 = 0X00000457
```

```
10000 / 0X00000008 = 0X000004E2
10000 / 0X00000007 = 0X00000594
10000 / 0X00000006 = 0X00000682
10000 / 0X00000005 = 0X000007D0
10000 / 0X00000004 = 0X000009C4
10000 / 0X00000003 = 0X00000D05
10000 / 0X00000002 = 0X00001388
10000 / 0X00000001 = 0X00002710
10000 / 0X00000000 = errhandler called: code = 0X00000001: divide by 0
caller's pc = 0X00008304
returning...

run time error: divide by 0
program terminated

Program terminated normally at PC = 0x0000854c
      0x0000854c: 0xef000011 .... : >  swi     0x11
armsd: quit
Quitting
>
```

This time an integer division by zero has been detected by the standalone runtime system, which called `__err_handler()`. `__err_hander()` output the first set of error messages in the above output. Control was then returned to the runtime system which output the second set of error messages and terminated execution.

6.5.5 longjmp and setjmp

A further demonstration can be made using `errtest` by predefining the macro LONGJMP to perform a `longjmp` out of `__err_handler` back into the user program, thus catching and dealing with the error. First recompile and link `errtest`:

```
armcc -c errtest.c -li -apcs 3/32bit hardfp -DDIVIDE_ERROR -DLONGJMP
armlink -o errtest errtest.o rtstand.o
```

Then rerun `errtest` under armsd. We expect the integer divide by zero to occur once again:

```
> armsd -li -size 512K errtest
A.R.M. Source-level Debugger, version 4.10 (A.R.M.) [Aug 26 1992]
ARMulator V1.20, 512 Kb RAM, MMU present, Demon 1.01, FPE, Little
endian.
Object program file errtest
armsd: go
(the floating point instruction-set is not available)
Using integer arithmetic ...
10000 / 0X0000000A = 0X000003E8
10000 / 0X00000009 = 0X00000457
10000 / 0X00000008 = 0X000004E2
```

```
10000 / 0X00000007 = 0X00000594
10000 / 0X00000006 = 0X00000682
10000 / 0X00000005 = 0X000007D0
10000 / 0X00000004 = 0X000009C4
10000 / 0X00000003 = 0X00000D05
10000 / 0X00000002 = 0X00001388
10000 / 0X00000001 = 0X00002710
10000 / 0X00000000 = errhandler called: code = 0X00000001: divide by 0
caller's pc = 0X00008310
returning...

Returning from __err_handler() with errnum = 0X00000001

Program terminated normally at PC = 0x00008558
      0x00008558: 0xef000011 .... : >  swi     0x11
armsd: quit
Quitting
>
```

The runtime system detected the integer divide by zero, and as before `__err_handler()` was called, which produced the error messages. However, this time `__err_handler()` used `longjmp` to return control to the program, rather than the runtime system.

6.5.6 Floating point support

Using `errtest` we can also demonstrate floating point support. You should already have copied the appropriate floating point emulator object code into your working directory. For the configuration used in this example `fpe_32l.o` is the correct object file.

However, in addition to this it is also necessary to link with an fpe *stub*, which we must compile from the source given (`fpestub.s`).

```
armasm fpestub.s -o fpestub.o -li -apcs 3/32bit
armlink -o errtest errtest.o rtstand.o fpestub.o fpe_32l.o -d
```

The resulting executable, `errtest`, can be run under armsd as before:

```
> armsd -li -size 512K errtest
A.R.M. Source-level Debugger, version 4.10 (A.R.M.) [Aug 26 1992]
ARMulator V1.20, 512 Kb RAM, MMU present, Demon 1.01, FPE, Little
endian.
Object program file errtest
armsd: go
(the floating point instruction-set is available)
Using Floating point, but casting to int ...
10000 / 0X0000000A = 0X000003E8
10000 / 0X00000009 = 0X00000457
10000 / 0X00000008 = 0X000004E2
```

```
10000 / 0X00000007 = 0X00000594
10000 / 0X00000006 = 0X00000682
10000 / 0X00000005 = 0X000007D0
10000 / 0X00000004 = 0X000009C4
10000 / 0X00000003 = 0X00000D05
10000 / 0X00000002 = 0X00001388
10000 / 0X00000001 = 0X00002710
10000 / 0X00000000 = errhandler called: code = 0X80000202: Floating
Point
Exception : Divide By Zero

caller's pc = 0XE92DE000
returning...

Returning from __err_handler() with errnum = 0X80000202

Program terminated normally at PC = 0x00008558 (__rt_exit + 0x10)
+0010 0x00008558: 0xef000011 .... : >  swi     0x11
armsd: quit
Quitting
>
```

This time the floating point instruction set is found to be available, and when a floating point division by zero is attempted, `__err_handler` is called with the details of the floating point divide by zero exception.

Note that if you have compiled `errtest.c` other than as in ➲*6.5.5 longjmp and setjmp* on page 6-20, you will not see precisely this dialogue with armsd.

6.5.7 Running out of heap

A second example program, memtest.c demonstrates how the standalone runtime system copes with allocating stack space, and also demonstrates the simple memory allocation function __rt_alloc. Let us first compile this program so that it should repeatedly request more memory, until there is none left:

```
armcc -li -apcs 3/32bit memtest.c -c
armlink -o memtest memtest.o rtstand.o
```

This can be run under armsd in the usual way:

```
> armsd -li -size 512K memtest
A.R.M. Source-level Debugger, version 4.10 (A.R.M.) [Aug 26 1992]
ARMulator V1.20, 512 Kb RAM, MMU present, Demon 1.01, FPE, Little
endian.
Object program file memtest
armsd: go
kernel memory management test
force stack to 4KB
request 0 words of heap - allocate 256 words at 0X000085A0
force stack to 8KB
..
force stack to 60KB
request 33211 words of heap - allocate 33211 words at 0X00049388
force stack to 64KB
request 49816 words of heap - allocate 5739 words at 0X00069A74
memory exhausted, 105376 words of heap, 64KB of stack
Program terminated normally at PC = 0x0000847c
      0x0000847c: 0xef000011 .... : >  swi     0x11
armsd: quit
Quitting
>
```

This demonstrates that allocating space on the stack is working correctly, and also that the __rt_alloc() routine is working as expected. The program terminated because in the end __rt_alloc() could not allocate the requested amount of memory.

6.5.8 Stack overflow checking

memtest can also be used to demonstrate stack overflow checking by recompiling with the macro STACK_OVERFLOW defined. In this case the amount of stack required is increased until there is not enough stack available, and stack overflow detection causes the program to be aborted.

To recompile and link memtest.c, issue the following commands:

```
armcc -li -apcs 3/32bit memtest.c -c -DSTACK_OVERFLOW
armlink -o memtest memtest.o rtstand.o
```

Running this program under armsd produces the following output:

```
> armsd -li -size 512K memtest
A.R.M. Source-level Debugger, version 4.10 (A.R.M.) [Aug 26 1992]
ARMulator V1.20, 512 Kb RAM, MMU present, Demon 1.01, FPE, Little
endian.
Object program file memtest
armsd: go
kernel memory management test
force stack to 4KB
...
force stack to 256KB
request 1296 words of heap - allocate 1296 words at 0X0000AE20
force stack to 512KB

run time error: stack overflow
program terminated

Program terminated normally at PC = 0x0000847c
      0x0000847c: 0xef000011 .... : >  swi     0x11
armsd: quit
Quitting
>
```

Clearly stack overflow checking did indeed catch the case where too much stack was required, and caused the runtime system to terminate the program after giving an appropriate diagnostic.

6.5.9 Extending the standalone runtime system

For a many applications it may be desirable to have access to more of the standard C library than is provided by the minimal runtime system. This section demonstrates how to take out a part of the standard C library and plug it into the standalone runtime system.

The function which we will add to `rtstand` is `memmove()`. Although this is small, and easily extracted from the C library source, the same methodology can be applied to larger sections of the C library, eg. the dynamic memory allocation system (`malloc()`, `free()`, etc.).

The source of the C library can be found in directory `cl`. The source for the `memmove()` function is in `cl/string.c`. The extracted source for `memmove()` has been put into `memmove.c`, and the compile time option `_copywords` has been removed. The function declaration for `memmove()` and a typedef for `size_t` (extracted from `include/stddef.h`) have been put into `examples/clstand/memmove.h`.

Our module can be compiled using:

```
armcc -c memmove.c -li -apcs 3/32bit
```

The output, `memmove.o` can be linked with the user's other object modules together with `rtstand.o` in the normal way (see previous examples in this section).

The files rtstand1.s and rtstand1.h are modified version of rtstand.s and rtstand.h respectively. rtstand1.s has the assembler code generated for __rt_memmore included in it. memmore.h has been merged with rtstand1.h to produce rtstand1.h.

6.5.10 The size of the standalone runtime library

rtstand.s has been separated into several code Areas. This allows armlink to detect any unreferenced Areas and then eliminate them from the output image.

The table below shows the typical size of the Areas in rtstand.o:

Area	Size in bytes	Functions
C$$data	4	
C$$code$$__main	96	__main, __rt_exit
C$$code$$__rt_fpavailable	8	__rt_fpavailable
C$$code$$__rt_trap	128	__rt_trap
C$$code$$__rt_alloc	68	__rt_alloc
C$$code$$__rt_stkovf	76	__rt_stkovf_split_*
C$$code$$__jmp	100	longjmp, setjmp
C$$code$$__divide	256	__rt_sdiv, __rt_udiv, __rt_udiv10,__rt_udiv10, __rt_divtest
TOTAL	736	

Table 6-2: Typical Area sizes in rtstand.o

If floating point support is definitely not required, then the EnsureNoFPSupport variable can be set to {TRUE}, and some extra space will be saved. After making any modifications to rtstand.s, the size of the various areas can be found using one of the following commands:

```
decaof -b rtstand.o
decaof -q rtstand.o
```

From the above table it is clear that for many applications the standalone runtime library will be roughly 0.5Kb.

7 Interfacing C and Assembly Language

This chapter explains how to write programs that contain routines written both in C and assembly language, and how to use the ARM Procedure Call Standard to pass arguments and results between them.

Interfacing C and Assembly Language

7.1 Introduction

In some situations you may find it necessary to mix C and assembly language in the same program. For example, a program may require particular routines which are performance-critical, and which must therefore be hand-coded in order to run at optimum speed.

The ARM Software Development Toolkit enables AOF object files to be generated from C and assembly language source by the appropriate tools (compiler and assembler, respectively), and then linked with one or more libraries to produce an executable file, as shown below:

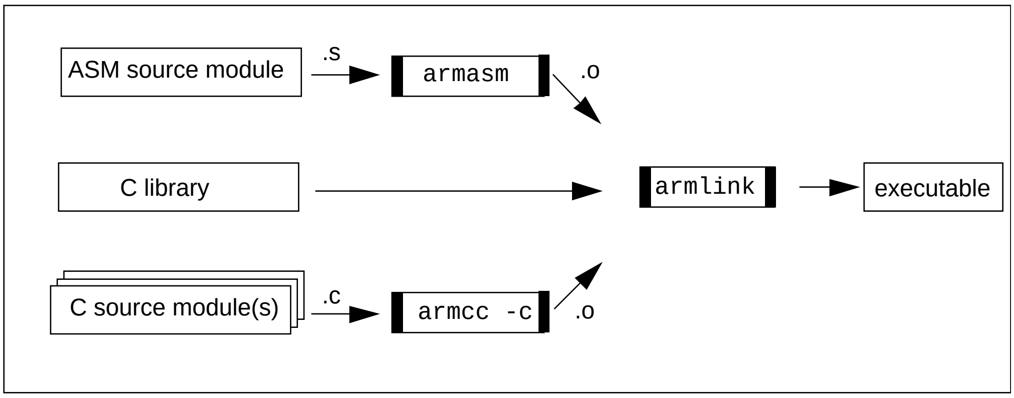


Figure 7-1: Mixing C and assembly language

Irrespective of the language in which they are written, routines that make cross-calls to other modules need to observe a common convention of argument and result passing. For the ARM, this convention is called the *ARM Procedure Call Standard*, or APCS. In this chapter, we introduce the APCS, and discuss its role in ARM assembly language for passing and returning values and pointers to structures for use by C routines.

7.2 Using the ARM Procedure Call Standard

The ARM Procedure Call Standard, or *APCS*, is a set of rules which govern calls between functions in separately compiled or assembled code fragments.

The APCS defines:

- constraints on the use of registers
- stack conventions
- the format of a stack backtrace data structure
- argument passing and result return
- support for the ARM shared library mechanism

Code which is produced by compilers is expected to adhere to the APCS at all times. Such code is said to be *strictly conforming.*

Handwritten code is expected to adhere to the APCS when making calls to externally visible functions. Such code is said to be *conforming*.

The ARM Procedure Call Standard comprises a family of variants. Each variant is exclusive, so that code which conforms to one cannot be used with code that conforms to another. Your choice of variant will depend on whether:

- the Program Counter is 32-bit or 26-bit
- stack limit checking is explicit (performed by code) or implicit (performed by memory management hardware)
- floating point values are passed in floating point registers
- code is reentrant or non-reentrant

For the full specification of the APCS, see ➲*The ARM Software Development Toolkit Reference Manual: Chapter 19, ARM Procedure Call Standard*.

7.2.1 Register names and usage

The following table summarises the names and uses allocated to the ARM and Floating Point registers under the APCS (note that not all ARM systems support floating point).

Register Number	APCS Name	APCS Role
0	`a1`	argument 1 / integer result / scratch register
1	`a2`	argument 2 / scratch register
2	`a3`	argument 3 / scratch register
3	`a4`	argument 4 / scratch register
4	`v1`	register variable
5	`v2`	register variable
6	`v3`	register variable
7	`v4`	register variable
8	`v5`	register variable
9	`sb/v6`	static base / register variable
10	`sl/v7`	stack limit / stack chunk handle / reg. variable
11	`fp`	frame pointer
12	`ip`	scratch register / new-sb in inter-link-unit calls
13	`sp`	lower end of current stack frame
14	`lr`	link address / scratch register
15	`pc`	program counter
f0	`0`	FP argument 1 / FP result / FP scratch register
f1	`1`	FP argument 2 / FP scratch register
f2	`2`	FP argument 3 / FP scratch register
f3	`3`	FP argument 4 / FP scratch register

Table 7-1: ACPS Registers

Simplistically:

`a1-a4`, `f0-f3`	are used to pass arguments to functions. `a1` is also used to return integer results, and `f0` to return FP results. These registers can be corrupted by a called function.
`v1-v5`, `f4-f7`	are used as register variables. They must be preserved by called functions.

`sb, sl, fp, ip, sp, lr, pc`

have a dedicated role in some APCS variants some of the time, ie. there are times when some of these registers can be used for other purposes even when strictly conforming to the APCS. In some variants of the APCS `sb` and `sl` are available as additional variable registers `v6` and `v7` respectively.

As stated previously, hand-coded assembler routines need not *conform strictly* to the APCS, but need only *conform*. This means that all registers which do not need to be used in their APCS role by an assembler routine (eg. `fp`) can be used as working registers provided that their value on entry is restored before returning.

7.2.2 An example of APCS register usage: 64-bit integer addition

This example illustrates how to code a small function in ARM assembly language, such that it can be used from C modules.

First, however, we will write the function in C, and examine the compiler’s output.

The function will perform a 64-bit integer addition using a two-word data structure to store each 64-bit operand. In assembler, the obvious way to code the addition of double-length integers would be to use the Carry flag from the low word addition in the high word addition. However, in C there is no way of specifying the Carry flag, so we have to use a workaround, as follows:

```
void add_64(int64 *dest, int64 *src1, int64 *src2)
{ unsigned hibit1=src1->lo >> 31, hibit2=src2->lo >> 31, hibit3;
  dest->lo=src1->lo + src2->lo;
  hibit3=dest->lo >> 31;
  dest->hi=src1->hi + src2->hi +
           ((hibit1 & hibit2) || (hibit1!= hibit3));
  return;
}
```

The highest bits of the low words in the two operands are calculated (shifting them into bit 0, while clearing the rest of the register). These are then used to determine the value of the carry bit (in the same way as the ARM itself does).

Examining the compiler's output

If the addition routine were to be used a great deal, a poor implementation such as this would probably be inadequate. To see just how good or bad it is, let us look at the actual code which the compiler produces.

Copy file `add64_1.c` from directory `examples/candasm` to your current working directory and compile it to ARM assembly language source as follows:

```
armcc -li -apcs 3/32bit -S add64_1.c
```

The -S flag tells the compiler to produce ARM assembly language source (suitable for armasm) instead of object code. The -li flag tells it to compile for little-endian memory and the -apcs option specifies the 32-bit version of APCS 3. You can omit these options if your compiler is configured to have them as defaults.

Looking at the output file, add64_1.s, we can see that this is indeed an inefficient implementation.

```
add_64
        STMDB    sp!,{v1,lr}
        LDR      v1,[a2,#0]
        MOV      a4,v1,LSR #31
        LDR      ip,[a3,#0]
        MOV      lr,ip,LSR #31
        ADD      ip,v1,ip
        STR      ip,[a1,#0]
        MOV      ip,ip,LSR #31
        LDR      a2,[a2,#4]
        LDR      a3,[a3,#4]
        ADD      a2,a2,a3
        TST      a4,lr
        TEQEQ    a4,ip
        MOVNE    a3,#1
        MOVEQ    a3,#0
        ADD      a2,a2,a3
        STR      a2,[a1,#4]!
        LDMIA    sp!,{v1,pc}
```

Modifying the compiler's output

Let us return to our original intention of coding the 64-bit integer addition using the Carry flag. Since use of the Carry flag cannot be specified in C, we must get the compiler to produce almost the right code, and then modify it by hand. Let us start with (incorrect) code which does not perform the carry addition:

```
void add_64(int64 *dest, int64 *src1, int64 *src2)
{ dest->lo=src1->lo + src2->lo;
  dest->hi=src1->hi + src2->hi;
  return;
}
```

You will find this in file examples/candasm/add64_2.c. Copy it to your current working directory, and then compile it to assembler source with the command:

```
armcc -li -apcs 3/32bit -S add64_2.c
```

This produces source in add64_2.s, which will include something like the following code (though yours may be slightly different, depending on the version of armcc supplied with your release):

```
add_64
    LDR    a4,[a2,#0]
    LDR    ip,[a3,#0]
    ADD    a4,a4,ip
    STR    a4,[a1,#0]
    LDR    a2,[a2,#4]
    LDR    a3,[a3,#4]
    ADD    a2,a2,a3
    STR    a2,[a1,#4]
    MOV    pc,lr
```

Comparing this to the C source we can see that the first ADD instruction produces the low order word, and the second produces the high order word. All we need to do to get the carry from the low to high word right is change the first ADD to ADDS (add and set flags), and the second ADD to an ADC (add with carry). This modified code is available in directory `examples/candasm` as `add64_3.s`.

What effect did the APCS have?

The most obvious way in which the APCS has affected the above code is that the registers have all been given APCS names.

`a1` holds a pointer to the destination structure, while `a2` and `a3` hold pointers to the operand structures. Both `a4` and `ip` are used as temporary registers which are not preserved. The conditions under which `ip` can be corrupted will be discussed later in this chapter.

This is a simple leaf function, which uses few temporary registers, so none are saved to the stack and restored on exit. Therefore a simple `MOV pc,lr` can be used to return.

If we wished to return a result—perhaps the carry out from the addition—this would be loaded into `a1` prior to exit. We could do this by changing the second ADD to ADCS (add with carry and set flags), and adding the following instructions to load `a1` with `1` or `0` depending on the carry out from the high order addition.

```
    MOV    a1, #0
    ADC    a1, a1, #0
```

Back to the first implementation

Although the first C implementation is inefficient, it shows us more about the APCS than the hand-modified version.

We have already seen `a4` and `ip` being used as non-preserved temporary registers. However, here `v1` and `lr` are also used as temporary registers. `v1` is preserved by being stored (together with `lr`) on entry. `lr` is corrupted, but a copy is saved onto the stack and then reloaded into `pc` when `v1` is restored.

Thus there is still only a single exit instruction, but now it is:

```
    LDMIA  sp!,{v1,pc}
```

7.2.3 A more detailed look at APCS register usage

We stated initially that `sb`, `sl`, `fp`, `ip`, `sp` and `lr` are dedicated registers, but the example showed `ip` and `lr` being used as temporary registers. Indeed, there are times when these registers are not used for their APCS roles. It will be useful for you to know about these situations, so that you can write efficient (but safe) code which uses as many of the registers as possible and so avoids unnecessary saving and restoring of registers.

`ip` is used only during function calls. It is conventionally used as a local code generation temporary register. At other times it can be used as a corruptible temporary register.

`lr` holds the address to which control must return on function exit. It can be (and often is) used as a temporary register after pushing its contents onto the stack. This value can then be reloaded straight into the PC.

`sp` is the stack pointer, which is always valid in *strictly conforming* code, but need only be preserved in handwritten code. Note, however, that if any use of the stack is to be made by handwritten code, `sp` must be available.

`sl` is the stack limit register. If stack limit checking is explicit (ie. is performed by code when stack pushes occur, rather than by memory management hardware causing a trap on stack overflow), `sl` must be valid whenever `sp` is valid. If stack checking is implicit `sl` is instead treated as `v7`, an additional register variable (which must be preserved by called functions).

`fp` is the frame pointer register. It contains either zero, or a pointer to the most recently created stack backtrace data structure. As with the stack pointer this must be preserved, but in handwritten code does not need to be available at every instant. However It should be valid whenever any *strictly conforming* functions are called.

`sb` is the static base register. If the variant of the APCS in use is reentrant, this register is used to access an array of static data pointers to allow code to access data reentrantly. However, if the variant being used is not reentrant, `sb` is instead available as an additional register variable, `v6` (which must be preserved by called functions).

`sp`, `sl`, `fp` and `sb` must all be preserved on function exit for APCS *conforming* code.

For more information refer to ➲*The ARM Software Development Toolkit Reference Manual: Chapter 19, ARM Procedure Call Standard*.

7.3 Passing and Returning Structures

This section covers:

- the default method for passing structures to and from functions
- cases in which this is automatically optimised
- telling the compiler to return a struct value in several registers

7.3.1 The default method

Unless special conditions apply (detailed in following sections), C structures are passed in registers which if necessary overflow onto the stack and are returned via a pointer to the memory location of the result

For struct-valued functions, a pointer to the location where the struct result is to be placed is passed in `a1`(the first argument register). The first argument is then passed in `a2`, the second in `a3`, and so on.

It is as if:

```
struct s f(int x)
```

were compiled as:

```
void f(struct s *result, int x)
```

Consider the following code:

```
typedef struct two_ch_struct
{ char ch1;
  char ch2;
} two_ch;

two_ch max( two_ch a, two_ch b )
{ return (a.ch1>b.ch1) ? a : b;
}
```

This is available in the directory `examples/candasm` as `two_ch.c.` It can be compiled to produce assembly language source using:

```
armcc -S two_ch.c -li -apcs 3/32bit
```

where `-li` and `-apcs 3/32bit` can be omitted if armcc has been configured appropriately.

Here is the code which armcc produced (the version of armcc supplied with your release may produce slightly different output to that listed here):

```
max
    MOV    ip,sp
    STMDB  sp!,{a1-a3,fp,ip,lr,pc}
    SUB    fp,ip,#4
    LDRB   a3,[fp,#-&14]
    LDRB   a2,[fp,#-&10]
    CMP    a3,a2
    SUBLE  a2,fp,#&10
    SUBGT  a2,fp,#&14
    LDR    a2,[a2,#0]
    STR    a2,[a1,#0]
    LDMDB  fp,{fp,sp,pc}
```

The `STMDB` instruction saves the arguments onto the stack, together with the frame pointer, stack pointer, link register and current pc value (this sequence of values is the stack backtrace data structure).

`a2` and `a3` are then used as temporary registers to hold the required part of the structures passed, and `a1` is a pointer to an area in memory in which the resulting struct is placed—all as expected.

7.3.2 Returning integer-like structures

The ARM Procedure Call Standard specifies different rules for returning *integer-like* structures. An integer-like structure:

- is no larger than one word in size
- exclusively has sub-fields whose byte offset is 0

The following structures are integer-like:

```
struct

{ unsigned a:8, b:8, c:8, d:8;
}

union polymorphic_ptr
{ struct A *a;
  struct B *b;
  int      *i;
}
```

whereas the structure used in the previous example is not:

```
struct { char ch1, ch2; }
```

An integer-like structure has its *contents* returned in `al`. This means that `a1` is not needed to pass a pointer to a result struct in memory, and is instead used to pass the first argument.

For example, consider the following code:

```
typedef struct half_words_struct
{ unsigned field1:16;
  unsigned field2:16;
} half_words;

half_words max( half_words a, half_words b )
{ half_words x;
  x= (a.field1>b.field1) ? a : b;
  return x;
}
```

Arguments a and b will be passed in registers a1 and a2, and since half_word_struct is integer-like we expect a1 to return the result structure directly, rather than a pointer to it.

The above code is available in directory examples/candasm as half_str.c. It can be compiled to produce assembly language source using:

```
armcc -S half_str.c -li -apcs 3/32bit
```

where -li and -apcs 3/32bit can be omitted if armcc has been configured appropriately. Here is the code which armcc produced (your version may produce slightly different output to that listed here):

```
max
    MOV    a3,a1,LSL #16
    MOV    a3,a3,LSR #16
    MOV    a4,a2,LSL #16
    MOV    a4,a4,LSR #16
    CMP    a3,a4
    MOVLE  a1,a2
    MOV    pc,lr
```

From this we can see that the contents of the half_words structure is returned directly in a1 as expected.

7.3.3 Returning non integer-like structures in registers

There are occasions when a function needs to return more than one value. The normal way to achieve this is to define a structure which holds all the values to be returned, and return this.

This will result in a pointer to the structure being passed in a1, which will then be dereferenced to store the values returned.

For some applications in which such a function is time-critical, the overhead involved in “wrapping” and then “unwrapping” the structure can be significant. However, there is a way to tell the compiler that a structure should be returned in the argument registers a1 - a4. Clearly this is only useful for returning structures that are no larger than four words.

The way to tell the compiler to return a structure in the argument registers is to use the keyword __value_in_regs.

Example: returning a 64-bit Result

To illustrate how to use `__value_in_regs`, let us consider writing a function which multiplies two 32-bit integers together and returns a 64-bit result.

The way this function must work is to split the two 32-bit numbers (a, b) into high and low 16-bit parts (`a_hi`, `a_lo`, `b_hi`, `b_lo`). The four multiplications `a_lo` * `b_lo`, `a_hi` * `b_lo`, `a_lo` * `b_hi`, `a_hi` * `b_lo` must be performed and the results added together, taking care to deal with carry correctly.

Since the problem involves manipulation of the Carry flag, writing this function in C will not produce optimal code (see *7.2.2 An example of APCS register usage: 64-bit integer addition* on page 7-5). We will therefore have to code the function in ARM assembly language. The following performs the algorithm just described:

```
; On entry a1 and a2 contain the 32-bit integers to be multiplied (a, b)
; On exit a1 and a2 contain the result (a1 bits 0-31, a2 bits 32-63)

mul64
    MOV    ip, a1, LSR #16        ; ip = a_hi
    MOV    a4, a2, LSR #16        ; a4 = b_hi
    BIC    a1, a1, ip, LSL #16    ; a1 = a_lo
    BIC    a2, a2, a4, LSL #16    ; a2 = b_lo
    MUL    a3, a1, a2             ; a3 = a_lo * b_lo        (m_lo)
    MUL    a2, ip, a2             ; a2 = a_hi * b_lo        (m_mid1)
    MUL    a1, a4, a1             ; a1 = a_lo * b_hi        (m_mid2)
    MUL    a4, ip, a4             ; a4 = a_hi * b_hi        (m_hi)
    ADDS   ip, a2, a1             ; ip = m_mid1 + m_mid2    (m_mid)
    ADDCS  a4, a4, #&10000        ; a4 = m_hi + carry       (m_hi')
    ADDS   a1, a3, ip, LSL #16    ; a1 = m_lo + (m_mid<<16)
    ADC    a2, a4, ip, LSR #16    ; a2 = m_hi' + (m_mid>>16) + carry
    MOV    pc, lr
```

This code is fine for use with assembly language modules, but in order to use it from C we need to tell the compiler that this routine returns its 64-bit result in registers. This can be done by making the following declarations in a header file:

```
typedef struct int64_struct

{ unsigned int lo;
  unsigned int hi;
} int64;

__value_in_regs extern int64 mul64(unsigned a, unsigned b);
```

The above assembly language code and declarations, together with a test program, are all in directory `examples/candasm` as the files `mul64.s`, `mul64.h`, `int64.h` and `multest.c`. To compile, assemble and link these to produce an executable image suitable for armsd, first copy them to your current directory, and then execute the following commands:

```
armasm mul64.s -o mul64.o -li
armcc -c multest.c -li -apcs 3/32bit
armlink mul64.o multest.o libpath/armlib.32l -o multest
```

where *libpath* is the directory in which the semi-hosted C libraries reside (eg. the lib directory of the ARM Software Tools Release).

Note that -li and -apcs 3/32bit can be omitted if armcc and armasm (and armsd, below) have been configured appropriately.

multest can then be run under armsd as follows:

```
> armsd -li multest
A.R.M. Source-level Debugger, version 4.10 (A.R.M.) [Aug 26 1992]
ARMulator V1.20, 512 Kb RAM, MMU present, Demon 1.01, FPE, Little
endian.
Object program file multest
armsd: go
Enter two unsigned 32-bit numbers in hex eg.(100 FF43D)
12345678 10000001
Least significant word of result is 92345678
Most  significant word of result is  1234567
Program terminated normally at PC = 0x00008418
      0x00008418: 0xef000011 .... : >  swi     0x11
armsd: quit
Quitting
>
```

To convince yourself that __value_in_regs is being used, try removing it from mul64.h, recompile multest.c, relink multest, and re-run armsd. This time the answers returned will be incorrect, since the result is no longer being returned in registers, but in a block of memory instead (ie. the code now has a bug).

8 Advanced Linking

This chapter explains how to generate programs which use overlays, and the linker's scatter loading facility, and describes the use of the ARM shared libraries.

Advanced Linking

8.1 Using Overlays

The ARM linker has two different methods of generating applications that use overlays. These are selectable from the command line using the `-OVERLAY` and `-SCATTER` linker options. Note that these options are mutually exclusive.

`-OVERLAY` causes the linker to compute the size of the overlay segments automatically, and to abut distinct memory partitions.

The linker generates a set of files in a directory specified by the `-OUTPUT` option. Overlay segments to be forced to specific memory addresses in a simple form of scatter loading. However, PCIT entries will be generated even for non-clashing overlays, producing extra overheads in terms of code size and execution speed. For this reason the `-OVERLAY` option is not recommended for generating scatter loaded images, and `-SCATTER` should be used instead.

`-SCATTER` instructs the linker to create either an extended AIF file or a directory of files. The overlays will be placed into load regions and the linker will add information to the executable to allow the overlay manager to copy the overlay segments from the correct load region. The directory of output files will be suitable for use in a ROM-based system.

All the overlay segments must have an execution address specified in the scatter load description file. The linker will not place overlay segments automatically. The scatter loading scheme does not support dynamic overlays. With scatter loading, PCIT information is not generated for execution regions not marked as overlays, so these regions do not have any overlay overhead associated with them.

8.1.1 Segment clash detection

In both cases, clash detection relies on the *name* of an overlay segment rather than its base address and size. The linker will attempt to find an underscore character ('_') in the name, and on finding one will take the preceding string to be the partition name. Two segments are deemed to clash if they have the same partition names: for example, `seg_test` and `seg_eval` will clash as the partition name is `seg` in both cases, while the names `alt_reset` and `pri_reset` will not.

Note The overlay system's underlying mechanisms such as the PCIT rely on the processor executing in ARM state. Therefore Thumb-aware processors cannot call overlays while operating in Thumb state.

8.1.2 The overlay manager

The overlay manager for scatter loaded overlays and the conventional overlay scheme are very similar. Indeed, only the code for loading a segment need be different. For a scatter loaded application, the code will be of the form:

```
Retry
;
; Use the overlay table generated by the linker. The table format
: is as follows:
; The first word in the table is contains the number of entries in
; the table.
; There follows that number of table entries. Each entry is 3 words
: long:
;      Word 1   Length of the segment in bytes.
;      Word 2   Execution address of the PCIT section address. This is
;               compared against the value in R8. If the values are
;               equal we have found the entry for the called overlay.
;      Word 3  Load address of the segment.
; Segment names are not used.
;
       IMPORT  |Root$$OverlayTable|
       LDR     r0,=|Root$$OverlayTable|
       LDR     r1,[r0],#4
search_loop
       CMP     r1,#0
       MOVEQ   r0,#2               ; The end the table has been reached
       BEQ     SevereErrorHandler   ; and the segment has not been found

       LDMIA   r0!,{r2,r3,r4}
       CMP     r8,r3
       SUBNE   r1,r1,#1
       BNE     search_loop

       LDR     r0,[ r8, #PCITSect_Base ]
       MOV     r1,r4
       MOV     r4,r2
       BL      MemCopy
```

where:

- `Root$$OverlayTable` is a symbol bound to the address of the linker generated overlay information table.
- `SevereErrorHandler` is a routine called when the overlay manager detects an error.
- `MemCopy` is a system-specific memory copy routine where r0 points to the destination area, r1 points to the source area, and r2 is the block size in bytes.

In the scatter loaded example supplied with the toolkit (in the `scatter.s` file in directory `examples/scatter`), the overlay manager initialisation routine has no work to do, as all memory copying and initialisation is done as part of the scatter loaded image initialisation.

When using the `-OVERLAY` option, the overlay manager code would be:

```
Retry
;
;       Call a routine to load the overlay segment.
;       First parameter is the length of the segment name.
;       The second parameter is the address of the segment name
;       The third parameter is the base address of the segment.
;       The routine returns the segment length in r0.
;
        MOV     r0,#12
        ADD     r1, r8, #PCITSect_Name
        LDR     r2, [ r8, #PCITSect_Base]
        BL      LoadOverlaySegment

        TEQ     r0,#0
        MOVEQ   r0,#2
        BEQ     SevereErrorHandler
```

`LoadOverlaySegment` loads the named segment. In a nonembedded environment, this routine would be implemented to load the segment from a file somewhere on the file system. This is the case in the `overlmgrs.s` file in directory `examples/overlay`. In an embedded environment where the code is in some form of nonvolatile memory, the overlay segments would need to be packaged up with sufficient information for a `LoadOverlaySegment` implementation to load the segments correctly.

For example, the overlay could be put into a pseudo file system in nonvolatile memory and the segments accessed by name. This 'packaging up' operation would need to be carried out after linking. The ARM software development toolkit does not do this, as it will be highly specific to the application’s run time environment. In the overlay example supplied with the toolkit, the overlay manager initialisation routine is used to copy read/write data from the load address to the execution address.

8.1.3 Scatter loading initialisation

The linker generates sufficient information for an initialisation routine to be written which will initialise all the execution regions that have base addresses not equal to their load addresses.

The linker will generate symbols specifying the length, execution addresses and load addresses. A simple initialisation routine is listed below. This uses two tables to control the initialisation process. The first table lists the lengths and execution addresses of zero initialised data. The second specifies the lengths, load and execution addresses of the execution regions that need to be copied. Both tables are terminated by a word containing zero.

```
       AREA InitApp, CODE , READONLY
       EXPORT InitialiseApp
InitialiseApp
       ADR    r0,ziTable
       MOV    R3,#0
ziLoop
       LDR    r1,[r0],#4
       CMP    r1,#0
       BEQ    initLoop
       LDR    r2,[r0],#4
ziFillLoop
       STR    r3,[r2],#4
       SUBS   r1,r1,#4
       BNE    ziFillLoop
       B      ziLoop

initLoop
       LDR    r1,[r0],#4
       CMP    r1,#0
       MOVEQ  pc,lr
       LDMIA  r0!,{r2,r3}
       CMP    r1,#16
       BLT    copyWords
copy4Words
       LDMIA  r3!,{r4,r5,r6,r7}
       STMIA  r2!,{r4,r5,r6,r7}
       SUBS   r1,r1,#16
       BGT    copy4Words
       BEQ    initLoop
copyWords
       SUBS r1,r1,#8
       LDMIAGE r3!,{r4,r5}
       STMIAGE r2!,{r4,r5}
       BEQ   initLoop

       LDR    r4,[r3]
       STR    r4,[r2]

       B      initLoop

;
; A couple of MACROS to make the table entries easier to add.
; The execname parameter is the name of execution to initialise or copy.
;
```

```
       MACRO
       ZIEntry  $execname
       LCLS   lensym
       LCLS   basesym
       LCLS   namecp
namecp SETS   "$execname"
lensym SETS   "|Image$$":CC:namecp:CC:"$$ZI$$Length|"
basesym SETS  "|Image$$":CC:namecp:CC:"$$ZI$$Base|"
       IMPORT $lensym
       IMPORT $basesym
       DCD    $lensym
       DCD    $basesym
       MEND

       MACRO
       InitEntry  $execname
       LCLS   lensym
       LCLS   basesym
       LCLS   loadsym
       LCLS   namecp
namecp SETS   "$execname"
lensym SETS   "|Image$$":CC:namecp:CC:"$$Length|"
basesym SETS   "|Image$$":CC:namecp:CC:"$$Base|"
loadsym SETS   "|Load$$":CC:namecp:CC:"$$Base|"
       IMPORT $lensym
       IMPORT $basesym
       IMPORT $loadsym
       DCD    $lensym
       DCD    $basesym
       DCD    $loadsym
       MEND
ziTable
       ZIEntry root    ; Zero initialised data from the root read/write
                     ; region
       DCD    0

InitTable
      InitEntry root ; Initialised data from the root read/write region
       DCD    0
       END
```

Each execution region that needs zero-initialised data to be initialised must have an entry in `ziTable` of the form:

```
        ZIEntry   name
```

where *name* is the name of the execution region. Similarly, each execution region that needs to be copied must have an entry in `InitTable` of the form:

```
        InitEntry name
```

The `InitialiseApp` routine is not called automatically at startup: it must be called explicitly before the application main program is entered.

This code can be found in the `initapp.s` file in directory `examples/scatter`.

8.2 ARM Shared Libraries

This section explains:

- what an ARM shared library is
- how the shared library mechanism works
- how to instruct the ARM linker to make a shared library
- how to make a toy shared library from the string section of the ANSI C library

8.2.1 About ARM shared libraries

ARM shared libraries support the sharing of utility, service or library functions between several concurrently executing client applications in a single address space. Such shared code is necessarily reentrant.

If a function is reentrant, each of its concurrently active clients must have a separate copy of the data it manipulates for them. The data cannot be associated with the code itself unless the data is read-only. In the ARM shared library architecture, a dedicated register called `sb` is used to address (indirectly) the static data associated with a client.

An ARM shared library is read only, reentrant and usually position-independent. A shared library made exclusively from object code compiled by the ARM C compiler will have all three of these attributes. Library components implemented in ARM assembly Language do not need to be reentrant and position-independent, but in practice only position independence is inessential.

A library with all three of these attributes in an ideal candidate for packing into a system ROM.

Some shared library mechanisms associate a shared library's data with the library itself and put only a place holder in the stub. At run time, a copy of the library's initialised static data is copied into the client's place holder by the dynamic linker or by library initialisation code.

The ARM shared library mechanism supports these ways of working provided the data is free of values which require link time (or run time) relocation. In other words, it can be supported provided the input data areas are free of relocation directives.

8.2.2 How ARM shared libraries work

Stubs and proxy functions

When a client application is linked with a shared library, it is linked not with the library itself but with a *stub object* containing:

- an *entry vector*
- a copy of the library's static data or a place holder for it

Each member of the entry vector is a *proxy* for a function in the matching shared library.

When a client first calls a proxy function, the call is directed to a *dynamic linker*. This is a small function (typically about 50-60 ARM instructions) which:

- locates the matching shared library
- if required, copies an initial image of the library's static data from the library to the place holding area in the stub
- patches the entry vector so each proxy function points at the corresponding library function
- resumes the call

Once an entry vector has been patched, all future proxy calls proceed directly to the target library function with only minimal indirection delay and no intervention by the dynamic linker.

Making an inter-link-unit call like this *is* more expensive than making a straightforward local procedure call, but not by much. It is also the only supported way to call a function more than 32MBytes away.

8.2.3 Locating a library which matches the stub

Locating a matching shared library is specific to a target system and you must provide code to do the location, but the remainder of the dynamic linking process is generic to all target systems. Consequently, in order to use ARM shared libraries, you have to design and implement a library location mechanism and adapt the dynamic linker to it. In practice this is quite straightforward, since:

- the ARM Linker provides support for parameterising a location mechanism
- a basic dynamic linker with neither location nor failure reporting mechanisms is a mere 42 ARM instructions

Please refer to ➲*The ARM Software Development Toolkit Reference Manual: Chapter 6, Linker* for a full explanation of parameter blocks.

8.2.4 How the dynamic linker works

The dynamic linker is entered via a proxy call with r0 pointing at the dynamic linker's 16-byte entry stub. Following this stub code is a copy of the parameter block for the shared library.

Stored in the parameter block is the identity of the library—this will be a 32-bit unique identifier or perhaps a string name. Either way, this can be passed to the library location mechanism. It is up to you to decide how to identify your shared libraries and, hence, what to put in their parameter blocks.

The library location function is required to return the address of the start of the library's offset table.

A primitive location mechanism might be to search a ROM for a matching string. This would identify the start of the parameter block of the matching shared library. Immediately preceding it will be negative offsets to library entry points and a non-negative count word containing the number of entry points. By working backwards through memory and counting, you can be sure you have found the entry vector and can return the address of its count word to the dynamic linker.

More sophisticated location schemes are possible, for example:

- You might include in your library a header containing code to be executed when the library is first loaded (into RAM) or initialised (in ROM) which registers the library's name with a library manager. Since the library manager has to be locatable without using the library manager, either its address has to be known or its function has to be supported by an underlying system call.
- You might adopt a scheme similar to that which is used by Acorn's RISC OS operating system. This supports a *module* mechanism which is often used to implement shared libraries. A RISC OS module may, by declaring so in its module header, be called when software interrupts (SWIs) in a specified range occur. When such a module is loaded, it extends the range of SWIs interpreted by the operating system. This mechanism can be used to locate a shared library by storing the identity of a library location SWI in the library's parameter block, and by implementing this SWI in the library module's header.

8.2.5 Instructing the linker to make a shared library

Prerequisites

You can make a shared library from any number of object files, including *reentrant stubs* of other shared libraries, provided that:

- each object file conforms to a reentrant version of the ARM Procedure Call Standard and each code area has the REENTRANT attribute
- there are no unresolved references resulting from the linking together of the component objects

An immediate consequence of the second rule is that it is impossible to make two shared libraries which refer to one another: to make the second library and its stub would require the stub of the first, but to make the first and its stub would require the stub of the second.

The first rule is not 100% necessary, and is difficult to enforce. The linker warns you if it finds a non-reentrant code area in the list of objects to be linked into a shared library, but will build the library and its matching stub anyway. You must decide whether the warning is real, or merely a formality.

Linker outputs

The ARM linker generates a shared library as two files:

- a plain binary file containing the read-only, reentrant, usually position independent, shared code
- an AOF format stub file with which client applications can be linked.

The linker can also generate a reentrant stub suitable for inclusion in another shared library.

The library image file contains, in order:

- read only code sections from your input objects
- if requested, a read only copy of the initialised static data from the input objects
- a table of (negative) offsets from the end of the library to its entry points
- if requested, the size and offset of the static data image
- a copy of the library's *parameter block*

You request a copy of the initialised static data to be included in a library when you describe to the linker how to make a shared library. If you request this, the linker writes the length and offset of the data image immediately after the entry vector. During linking, armlink defines symbols SHL$$data$$Size and SHL$$data$$Base to have these values; components of your library may refer to these symbols. Instead of including the static data in the stub, armlink includes a zero initialised place holding area of the same size. It also writes the length and (relocatable) address of this area immediately after the dynamic linker's entry veneer, giving the dynamic linker sufficient information to initialise the place holder at run time. During linking, the linker symbols SHL$$data$$Size and $$0$$Base describe this length and relocatable address.

Any data included in your shared library must be free of relocation directives. Please refer to ➲*The ARM Software Development Toolkit Reference Manual: Chapter 6, Linker* for a full explanation of what kind of data can be included in a shared library.

You specify a parameter block when you describe to the linker how to make a shared library. You might, for example, include the name of the library in its parameter block, to aid its location. An identical copy of the parameter block is included in the library's entry vector in the stub file.

Describing a shared library to the linker

To describe a shared library to the linker you have to prepare a file which describes:

- the name of the library
- the library parameter block
- what data areas to include
- what entry points to export

For precise details of how to do this, please refer to ➲*The ARM Software Development Toolkit Reference Manual: Chapter 6, Linker*. Below is an intuitive example you can work with and adapt:

```
; First, give the name of the file to contain the library -
; strlib - and its parameter block - the single word 0x40000...
> strlib \
  0x40000
; ...then include all suitable data areas...
+ ()
; ... finally export all the entry points...
; ... mostly omitted here for brevity of exposition.
memcpy
...
strtok
```

The name of this file is passed to armlink as the argument to the `-SHL` command line option: see ➲*The ARM Software Development Toolkit Reference Manual: Chapter 6, Linker* for further details.

8.2.6 Making a toy string library

This section refers to the files collected in directory `examples/reent`.

The header files `config.h` and `interns.h` let you compile `cl/string.c` locally. Little-endian code is assumed. If you want to make a big-endian string library you should edit `config.h`. Similarly, if you want to alter which functions are included or whether static data is initialised by copying from the library, you should edit `config.h`. You do not need to edit `interns.h`. If you use `config.h` unchanged you will build a little-endian library which includes a data image and which exports all of its functions.

Compiling the string library

To compile `string.c`, use the following command:

```
armcc -li -apcs /reent -zps1 -c -I. ../../cl/string.c
```

where:

`-li` tells armcc to compile for a little-endian ARM.

`-apcs /reent`tells armcc to compile reentrant code.

`-zps1` turns off software stack limit checking and allows the string library to be independent of all other objects and libraries. With software stack limit checking turned on, the library would depend on the stack limit checking functions which, in turn, depend on other sections of the C run time library. While such dependencies do not much obstruct the construction of full scale, production quality shared libraries, they are major impediments to a simple demonstration of the underlying mechanisms.

`-I.` tells armcc to look for needed header files in the current directory.

Linking the string library

To make a shared library and matching stub from string.o, use

```
       armlink -o strstub.o -shl strshl -s syms string.o
```

where:

-o instructs the linker to put strlib's stub in strstub.o

-shl points to the file which contains instructions for making a shared library called strlib

-s asks for a listing of symbol values in a file called syms

You may later need to look up the value of EFT$$Offset. As supplied, the dynamic linker expects a library's *external function table* (EFT) to be at address 0x40000. So, unless you extend the dynamic linker with a library location mechanism, you will have to load strlib at the address 0x40000-EFT$$Offset.

Making the test program and dynamic linker

You should now assemble the dynamic linker and compile the test code:

```
       armasm -li dynlink.s dynlink.o
       armcc -li -c strtest.c
```

To make the test program you must link together the test code, the dynamic linker, the string library stub and the appropriate ARM C library (so that references to library members other than the string functions can be resolved):

```
armlink -d -o strtest strtest.o dynlink.o strstub.o ../../lib/
armlib.32l
```

Running the test program with the shared string library

Now you are ready to try everything under the control of command-line armsd. For this example the value of EFT$$offset is assumed to be 0xa38.

```
>armsd strtest
A.R.M. Source-level Debugger version ...
ARMulator V1.30, 4 Gb memory, MMU present, Demon 1.1,...
Object program file strtest
armsd: getfile strlib 0x40000-0xa38
armsd: go

strerror(42) returns unknown shared string-library error 0x0000002A

Program terminated normally at PC = 0x00008354 (__rt_exit + 0x24)
+0024 0x00008354: 0xef000011 .... :    swi      0x11
armsd: q
Quitting
>
```

Before starting strtest you must load the shared string library with the command:

```
getfile strlib 0x40000-0xa38
```

where strlib is the name of the file containing the library, 0x40000 is the hard-wired address at which the dynamic linker expects to find the external function table, and 0xa38 is the value of EFT$$Offset, the offset of the external function table from the start of the library.

When strtest runs, it calls strerror(42) which causes the dynamic linker to be entered, the static data to be copied, the stub vector to be patched and the call to be resumed. You can watch this is more detail by setting a breakpoint on __rt_dynlink and single stepping.

9 Writing Code for ROM

This chapter describes how to construct simple ROM images which contain C code.

Writing Code for ROM

9.1 Introduction

This chapter describes how to construct simple ROM images which contain C code. It presents the necessary assembler glue to initialise the memory system before any C code can be called.

The following subjects are covered:

- how your application in ROM is initialised and how it passes control to C
- how to include standalone functions from the C library in your ROM
- troubleshooting hints and tips

9.2 Application Startup

One of the main considerations with C code in ROM is the way in which the application initialises itself and starts executing. If there is an operating system present this causes no problem as the application is entered automatically via the `main()` function.

In an embedded system there are a number of ways an image may be entered:

- via the RESET vector at location 0
- at the base address of the image

Applications entered via the RESET vector

The simplest case is where the application ROM is located at address 0 in the address map. The first instruction of your application will then be a branch instruction to the real entry point.

Applications entered at the base address

An application may be entered at its base address in one of two ways:

- The hardware can fake a branch at address 0 to the base address of the ROM.
- On RESET the ROM is mapped to address 0 by the memory management. When the application initialises the MMU it remaps the ROM to its correct address and performs a jump to the copy of itself running at the correct address.

9.2.1 Initialisation on RESET

Nothing is initialised on RESET so the entry point will have to perform some initialisation before it can call any C code.

Typically, the initialisation code may perform some or all of the following:

- define the entry point

  The assembler directive `ENTRY` marks the entry point.

- setup interrupt/exception vectors

  If the ROM is located at address 0, this will consist of a sequence of hard-coded branch instructions to the handler for each interrupt/exception.

  If the ROM is located elsewhere, the vectors will have to be dynamically initialised by the initialisation code. Some standard code for doing this is given in the first example later in the chapter.

- initialise the Stack Pointer registers

  Some or all of the following Stack pointers may require initialisation depending on which interrupts and exceptions are used.

`SP_irq`	if interrupt requests are used
`SP_fiq`	if fast interrupt requests are used

  The above must be initialised before interrupts are enabled.

`SP_abt`	for data abort handling
`SP_und`	for undefined instruction handling

  Generally, the above two will not be used in a simple embedded system, however you may wish to initialise them for debugging purposes.

`SP_svc`	must always be initialised

- initialise the memory system

  If your system has an MMU, the memory mapping must be initialised at this point before interrupts are enabled and before any code is called which might rely on RAM being present at particular address, either explicitly, or implicitly via the use of stack space.

- initialise any critical IO devices

  Critical IO devices are any devices which must be initialised before interrupts are enabled. Typically these devices will need to be initialised at this point. If they are not, they may cause spurious interrupts when interrupts are enabled.

- initialise any RAM variables required by the interrupt system

  For example, if your interrupt system has buffer pointers to read data into memory buffers, the pointers must be initialised at this point before interrupts are enabled.

- enable interrupts and change processor mode/state if necessary

  At this stage the processor will be in Supervisor mode. If your application runs in User mode, you should change to User mode at this point. You will also need to initialise the User mode SP register.

- initialise memory required by C code

  The initial values for any initialised variables must be copied from ROM to RAM. All other variables must be initialised to zero.

If the application uses scatter loading, see *8.1.3 Scatter loading initialisation* on page 8-4 for details of how to initialise these areas.

If scatter loading is not being used, the code to do this is given below:

```
    IMPORT |Image$$RO$$Limit|    ; End of ROM code (=start of ROM data)
    IMPORT |Image$$RW$$Base|     ; Base of RAM to initialise
    IMPORT |Image$$ZI$$Base|     ; Base and limit of area
    IMPORT |Image$$ZI$$Limit|    ; to zero initialise

    LDR    r0, =|Image$$RO$$Limit|; Get pointer to ROM data
    LDR    r1, =|Image$$RW$$Base| ; and RAM copy
    LDR    r3, =|Image$$ZI$$Base| ; Zero init base => top of
                                  ; initialised data
    CMP    r0, r1                 ; Check that they are different
    BEQ    %1
0   CMP    r1, r3                 ; Copy init data
    LDRCC  r2, [r0], #4
    STRCC  r2, [r1], #4
    BCC    %0
1   LDR    r1, =|Image$$ZI$$Limit|; Top of zero init segment
    MOV    r2, #0
2   CMP    r3, r1                 ; Zero init
    STRCC  r2, [r3], #4
    BCC    %2
```

- enter C code

  If your application runs in Thumb state, you should change to Thumb state using:

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

  It is now safe to call C code provided that it does not rely on any memory being initialised. For example:

  ```
  IMPORT  C_Entry
  BL      C_Entry
  ```

9.2.2 Example 1 - Building a ROM to be entered at its base address

The following example shows how to construct a simple piece of code suitable for ROM. In a real example much more would have to go into the initialisation section, but this is very hardware specific and is therefore not appropriate for this example.

The following sections of code may be found in the files `init.s` and `ex.c` in directory `examples/rom`.

The commands necessary to build the image are given at the end of the code.

```
--- init.s -----------------------------------------------------------
;
; The AREA must have the attribute READONLY, otherwise the linker will
; not place it in ROM.
;
; The AREA must have the attribute CODE, otherwise the assembler will
; not let us put any code in this AREA
;
; Note the '|' character is used to surround any symbols which contain
; non standard characters like '!'.

       AREA    Init, CODE, READONLY

; Now some standard definitions...

Mode_IRQ      EQU    0x12
Mode_SVC      EQU    0x13

I_Bit         EQU    0x80
F_Bit         EQU    0x40

SWI_Exit      EQU    0x11

; Locations of various things in our memory system

RAM_Base      EQU    0x10000000          ; 64k RAM at this base
RAM_Limit     EQU    0x10010000

IRQ_Stack     EQU    RAM_Limit           ; 1K IRQ stack at top of memory
SVC_Stack     EQU    RAM_Limit-1024      ; followed by SVC stack

; --- Define entry point
       EXPORT  __main; The symbol '__main' is defined here to ensure
__main              ; the C runtime system is not linked in.
       ENTRY

; --- Setup interrupt / exception vectors
       IF :DEF: ROM_AT_ADDRESS_ZERO
; If the ROM is at address 0 this is just a sequence of branches
              B      Reset_Handler
              B      Undefined_Handler
              B      SWI_Handler
              B      Prefetch_Handler
              B      Abort_Handler
```

```
             NOP           ; Reserved vector
             B      IRQ_Handler
             B      FIQ_Handler
       ELSE
; Otherwise we copy a sequence of LDR PC instructions over the vectors
; (Note: We copy LDR PC instructions because branch instructions
; could not simply be copied, the offset in the branch instruction
; would have to be modified so that it branched into ROM. Also, a
; branch instructions might not reach if the ROM is at an address
; > 32M).
                    MOV    R8, #0
                    ADR    R9, Vector_Init_Block
                    LDMIA  R9!, {R0-R7}
                    STMIA  R8!, {R0-R7}
                    LDMIA  R9!, {R0-R7}
                    STMIA  R8!, {R0-R7}

; Now fall into the LDR PC, Reset_Addr instruction which will continue
; execution at 'Reset_Handler'

Vector_Init_Block
                    LDR    PC, Reset_Addr
                    LDR    PC, Undefined_Addr
                    LDR    PC, SWI_Addr
                    LDR    PC, Prefetch_Addr
                    LDR    PC, Abort_Addr
                    NOP
                    LDR    PC, IRQ_Addr
                    LDR    PC, FIQ_Addr

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
       ENDIF

; The following handlers do not do anything useful in this example.
;
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

; The RESET entry point
Reset_Handler

; --- Initialise stack pointer registers
; Enter IRQ mode and set up the IRQ stack pointer
              MOV     R0, #Mode_IRQ:OR:I_Bit:OR:F_Bit ; No interrupts
              MSR     CPSR, R0
              LDR     R13, =IRQ_Stack

; Set up other stack pointers if necessary
; ...

; Set up the SVC stack pointer last and return to SVC mode
              MOV     R0, #Mode_SVC:OR:I_Bit:OR:F_Bit ; No interrupts
              MSR     CPSR, R0
              LDR     R13, =SVC_Stack

; --- Initialise memory system
; ...

; --- Initialise critical IO devices
; ...

; --- Initialise interrupt system variables here
; ...

; --- Enable interrupts
; Now safe to enable interrupts, so do this and remain in SVC mode
              MOV     R0, #Mode_SVC:OR:F_Bit  ; Only IRQ enabled
              MSR     CPSR, R0

; --- Initialise memory required by C code

       IMPORT |Image$$RO$$Limit|  ; End of ROM code (=start of ROM data)
       IMPORT |Image$$RW$$Base|   ; Base of RAM to initialise
       IMPORT |Image$$ZI$$Base|         ; Base and limit of area
       IMPORT |Image$$ZI$$Limit|        ; to zero initialise

       LDR    r0, =|Image$$RO$$Limit|    ; Get pointer to ROM data
       LDR    r1, =|Image$$RW$$Base|     ; and RAM copy
       LDR    r3, =|Image$$ZI$$Base|     ; Zero init base => top of
                                         ; initialised data
       CMP    r0, r1                     ; Check that they are different
       BEQ    %1
0      CMP    r1, r3                     ; Copy init data
       LDRCC  r2, [r0], #4
       STRCC  r2, [r1], #4
       BCC    %0
1      LDR    r1, =|Image$$ZI$$Limit|    ; Top of zero init segment
       MOV    r2, #0
2      CMP    r3, r1                     ; Zero init
       STRCC  r2, [r3], #4
       BCC    %2

; --- Now we enter the C code

              IMPORT  C_Entry
       [ :DEF:THUMB
              ORR     lr, pc, #1
              BX      lr
              CODE16              ; Next instruction will be Thumb
       ]
       BL     C_Entry
; In a real application we wouldn't normally expect to return, however
; this example does so the debug monitor swi SWI_Exit is used to halt the
; application.
       SWI    SWI_Exit

       END
--- ex.c ------------------------------------------------------------
/* We use the following Debug Monitor SWIs to write things out
* in this example
*/
extern __swi(0) WriteC(char c);          /* Write a character */
extern __swi(2) Write0(char *s);         /* Write a string */

/* The following symbols are defined by the linker and define
* various memory regions which may need to be copied or initialised
*/
extern char Image$$RO$$Limit[];
extern char Image$$RW$$Base[];

/* We define some more meaningful names here */
#define rom_data_base Image$$RO$$Limit
#define ram_data_base Image$$RW$$Base

/* This is an example of a pre-initialised variable. */
static unsigned factory_id = 0xAA55AA55;  /* Factory set ID */

/* This is an example of an uninitialised (or 0 initialised) variable */
static char display[8][40];               /* Screen buffer */

static const char hex[16] = "0123456789ABCDEF";

static void pr_hex(unsigned n)
{
       int i;

       for (i = 0; i < 8; i++) {
       WriteC(hex[n >> 28]);
       n <<= 4;
       }
}

void C_Entry(void)
{
if (rom_data_base == ram_data_base) {
Write0("Warning: Image has been linked as an application. To link as a
ROM image\r\n");
Write0("         link with the options -RO <rom-base> -RW <ram-
base>\r\n");
}

Write0("'factory_id' is at address ");
pr_hex((unsigned)&factory_id);
Write0(", contents = ");
pr_hex((unsigned)factory_id);
Write0("\r\n");

Write0("'display' is at address ");
pr_hex((unsigned)display);
Write0("\r\n");
}
---------------------------------------------------------------------
```

To build the ROM image

1. Compile the C file `ex.c` with the following command. The compiler will generate one warning which may be ignored.

```
armcc -c -fc -apcs 3/noswst/nofp ex.c
```

`-fc` Tells the compiler to allow the `$` character in variables.

`-apcs 3/noswst/nofp` Tells the compiler not to include code to do software stack checking (`noswst`) and not to use a frame pointer (`nofp`).

2. Assemble the initialisation code `init.s`.

```
armasm -apcs 3/noswst init.s
```

`-apcs 3/noswst` Tells the assembler that this code is only suitable for use with other code which does not have software stack checking. Code which uses software stack checking cannot generally be mixed with code which does not. The assembler will mark the object file as containing code which does not perform software stack checking so that the linker can give an error if it is mixed with code which does.

3. Build the ROM image using armlink.

```
armlink -o ex1_rom -Bin -RO 0xf0000000 -RW 0x10000000 -First
init.o(Init) -Remove -NoZeroPad -Map -Info Sizes init.o ex.o
```

`-Bin` Tells the linker to produce a plain binary image with no header. This is the most suitable form of image for putting in ROM.

`-RO 0xf0000000` Tells the linker that the ReadOnly or code segment will be placed at `0xf0000000` in the address map.This is the base of the ROM in this example.

`-RW 0x10000000` Tells the linker that the ReadWrite or data segment will be placed at `0x10000000` in the address map. This is the base of the RAM in this example.

`-First init.o(Init)` Tells the linker to place this area first in the image. Note that on Unix systems you may need to put a backslash `\` before each bracket.

`-Remove` Tells the linker to remove any unused code areas. In this example there are no unused areas, however this is a useful option for larger ROM builds.

`-NoZeroPad` Tells the linker not to pad the end of the image with zeros to make space for the variables. This option should always be used when building ROM images.

`-Map`

`-Info Sizes` These two options tell the linker to output various sorts of information during the link process. Neither of these options are necessary to build the ROM but are included here as an example. The output generated by each option is given below.

`-Map` tells the linker to print an AREA map or listing showing where each code or data section will be placed in the address space. The output from the above example is given below.

```
AREA map of ex1_rom:

Base          Size   Type   RO?    Name
f0000000      e4     CODE   RO     !!! from object file init.o
f00000e4      238    CODE   RO     C$$code from object file ex.o
f000031c      10     CODE   RO     C$$constdata from object file ex.o
10000000      4      DATA   RW     C$$data from object file ex.o
10000004      140    ZERO   RW     C$$zidata from object file ex.o
```

This shows that the linker places three code areas at successive locations starting from `0xf0000000` (where our ROM is based) and two data areas starting at address `0x10000000` (where our RAM is based).

`-Info Sizes` tells 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.

```
object file  code  inline   inline  'const'      RW  0-Init   debug
             size    data  strings     data    data    data    data
init.o        228       0        0        0       0       0       0
ex.o          184      28      356       16       4     320       0

             code  inline   inline  'const'      RW  0-Init   debug
             size    data  strings     data    data    data    data
Object totals 412      28      356       16       4     320       0
```

The required ROM size will be the sum of the code size (412), the inline data size (28), the inline strings (356), the `const` data (16) and the RW data (4). In this example the required ROM size would be 816 bytes. This should be exactly the same as the size of the `ex1_rom` image produced by the linker.

The required RAM size will be the sum of the RW data (4) and the 0-Init data (320), in this case 324 bytes. Note that the RW data is included in both the ROM and the RAM counts. This is because the ROM contains the initialisation values for the RAM data.

Running the ROM image

You can now run the ROM image using the ARMulator.

Start up armsd by typing `armsd` and enter the following commands at the `armsd:` prompt.

```
getfile ex1_rom 0xf0000000
```

This tells armsd to load the `ex1_rom` file at address `0xf0000000` in the ARMul memory map.

Check that the ROM has indeed been loaded correctly by disassembling the first section of it:

```
l 0xf0000000
```

This should produce output like the following, which is a disassembly of the first part of `init.s`. If it produces output which has each word reversed (ie. the word at `0xf0000000` is `0x0080a0e3` instead of `0xe3a08000`) then there is a problem with endianness. Check than your compiler and armsd are both configured for the same endianness.

```
0xf0000000: 0xe3a08000  .... :mov       r8,#0
0xf0000004: 0xe28f900c  .... :add       r9,pc,#0xc
0xf0000008: 0xe8b900ff  .... :ldmia     r9!,{r0-r7}
0xf000000c: 0xe8a800ff  .... :stmia     r8!,{r0-r7}
0xf0000010: 0xe8b900ff  .... :ldmia     r9!,{r0-r7}
0xf0000014: 0xe8a800ff  .... :stmia     r8!,{r0-r7}
0xf0000018: 0xe59ff018  .... :ldr       pc,0xf0000038 ; = #0xf0000070
0xf000001c: 0xe59ff018  .... :ldr       pc,0xf000003c ; = #0xf0000058
0xf0000020: 0xe59ff018  .... :ldr       pc,0xf0000040 ; = #0xf000005c
0xf0000024: 0xe59ff018  .... :ldr       pc,0xf0000044 ; = #0xf0000060
0xf0000028: 0xe59ff018  .... :ldr       pc,0xf0000048 ; = #0xf0000064
0xf000002c: 0xe1a00000  .... :nop
0xf0000030: 0xe59ff018  .... :ldr       pc,0xf0000050 ; = #0xf0000068
0xf0000034: 0xe59ff018  .... :ldr       pc,0xf0000054 ; = #0xf000006c
0xf0000038: 0xf0000070  ...p :andnv     r0,r0,r0,ror r0
0xf000003c: 0xf0000058  ...X :andnv     r0,r0,r8,asr r0
```

You should now be able to execute the ROM image. Set the PC to the base of the ROM image, then run it.

```
pc=0xf0000000
go
```

This should produce the following output:

```
'factory_id' is at address 10000000, contents = AA55AA55
'display' is at address 10000004
Program terminated normally at PC = 0xf00000c8
0xf00000c8: 0xef000011  .... :    swi      0x11
```

9.2.3 Example 2 - Building a ROM to be loaded at address 0

Using the same files as in example 1 (`ex.c` and `init.s`) reassemble the `init.s` file using the following command:

```
armasm -apcs 3/noswst -PD 'ROM_AT_ADDRESS_ZERO SETL {TRUE}' init.s
```

`-PD 'ROM_AT_ADDRESS_ZERO SETL {TRUE}'`

This option tells the assembler to PreDefine the symbol ROM_AT_ADDRESS_ZERO and to give it the logical (or boolean) value TRUE.

The assembler file `init.s` tests this symbol and generates different code depending on whether or not the symbol is set.

If the symbol is set, it generates a sequence of branches to be loaded directly over the vector area at address 0.

If you have not already compiled the C file `ex.c` do that now (see *To build the ROM image* on page 9-10) then relink the image using the following command:

```
armlink -o ex2_rom -Bin -RO 0 -RW 0x10000000 -First init.o(Init)
      -Remove -NoZeroPad -Map -Info Sizes init.o ex.o
```

The only difference between this and the command used in example 1 is that here we use `-RO 0` to specify the ROM is based at address 0.

Load and execute the ROM image under ARMul/armsd as follows.

```
armsd
getfile ex2_rom 0
pc=0
go
```

9.2.4 Example 3 - Building a ROM using scatter loading

Examples 1 and 2 are simple examples of scatter loading. Hence they can be modified to use the scatter loading mechanisms provided by the linker. The initialisation code `init.s` would be modified to use the scatter loading initialisation code described in *8.1.3 Scatter loading initialisation* on page 8-4.

The `Image$$` symbols used in `ex.c` will still be bound to the same values as in the non scatter loading case.

The linker command would be modified to be:

```
armlink -o ex3_rom -Bin  -Scatter scatdes -First init.o(Init)  -Remove
-Map -info Sizes init.o ex.o
```

In this case, the `-o` option will create a subdirectory called `ex3_rom` containing a single binary file called `root`.

Note that `-Scatter` tells the linker not to pad the end of the output binary files with zeros. Hence the `-NoZeroPad` option is not required when using `-Scatter`.

`scatdes` is the scatter loading description file. For example 1, this file would be:

```
ROOT 0xf0000000
ROOT-DATA 0x10000000
```

For example 2, the file would be:

```
ROOT 0x0
ROOT-DATA 0x10000000
```

The procedure for running the scatter loaded versions of these examples is identical to the non scatter loaded versions, except that the filename used in the armsd getfile commands would be `ex3_rom/root`.

In these cases, scatter loading does not offer a significant advantage over the non scatter loading case. However if we have more than one execution region that needs to be initialised, scatter loading is the easiest method to use.

9.3 Using the C Library in ROM

Although it is possible to link all of the C library into a ROM application there are several reasons why you may not wish to do this.

- The C library relies on the Debug Monitor (Demon) SWIs for its operation. Unless your ROM supports these SWIs in its SWI handler, the C library will not work in ROM.
- For the C library to run, the memory system must be configured in the way in which the C library expects. This may not be easy to support in your system.
- There is a minimum overhead of about 10K when the C library is included.

You are more likely to want to include particular standalone functions from the C library in your ROM.

Note Standalone functions are functions which do not rely on any part of the operating system environment. The functions `memcpy()` and `strcpy()` are examples of standalone functions. `fopen()` is not standalone, since it relies on being able to open files which are part of the operating system. Only standalone functions can be included easily in ROM. See ➲*9.3.2 Standalone C functions* on page 9-17 for a list of which functions in the C library are standalone.

No special code is necessary in your C code to use a standalone C function, just use the function as normal.

See ➲*6.5 Using the C Library in Deeply Embedded Applications* on page 6-17 for further details of runtime support for deeply embedded applications.

9.3.1 Example 4 - Using sprintf in ROM

Example 4 involves the following steps:

1. Compile the C source file using the Thumb compiler.
2. Assemble the `init.s` file using the Thumb/ARM assembler tasm.

3 Build the ROM image.

4 Run the ROM image under ARMul/armsd.

Compiling the C source file

Compile the C source file shown below using the Thumb compiler. This file may be found in directory `examples/rom`.

```
tcc -c -fc -apcs 3/noswst/nofp sprintf.c

--- sprintf.c ---------------------------------------------------------
#include <stdio.h>

/* We use the following Debug Monitor SWI to write things out
* in this example
*/
extern __swi(2) Write0(char *s);        /* Write a character */

/* The following symbols are defined by the linker and define
* various memory regions which may need to be copied or initialised
*/
extern char Image$$RO$$Limit[];
extern char Image$$RW$$Base[];


/* We define some more meaningful names here */
#define rom_data_base Image$$RO$$Limit
#define ram_data_base Image$$RW$$Base

void C_Entry(void)
{
       char s[80];

       if (rom_data_base == ram_data_base) {
              Write0("Warning: Image has been linked as an application.
              To link as a ROM image\r\n");
              Write0("        link with the options -RO <rom-base> -RW
              <ram-base>\r\n");
       }

       sprintf(s, "ROM is at address %p, RAM is at address %p\n",
rom_data_base, ram_data_base);
       Write0(s);
}
----------------------------------------------------------------------
```

Assembling the init.s file

Assemble the `init.s` file using the Thumb/ARM assembler tasm:

```
tasm -32 -apcs 3/noswst/nofp -PD 'THUMB SETL {TRUE}' init.s
```

`-32` By default the Thumb assembler assembles Thumb code. `init.s` contains a mixture of ARM and Thumb code, however it starts with ARM code so we tell the assembler to expect ARM code to start with.

`-PD 'THUMB SETL {TRUE}'` This sets the THUMB variable to TRUE. The variable is tested by `init.s` which generates extra code to switch to Thumb state before calling the C entry point.

Building the ROM image

Build the ROM image with the following armlink command:

```
armlink -o ex4_rom -Bin -RO 0xf0000000 -RW 0x10000000 -First init.o(Init)
-Remove -NoZeroPad -Info Sizes init.o sprintf.o armlib.16l
```

If `armlib.16l` is not in the current directory, you will need to specify the directory on the command line.

This will produce the following output:

```
object file   code  inline   inline   'const'      RW 0-Init   debug
              size    data  strings      data    data   data    data
init.o         236       0        0         0       0      0       0
sprintf.o       40      12      184         0       0      0       0

library membercode  inline   inline   'const'      RW 0-Init   debug
              size    data  strings      data    data   data    data
_sprintf.o      56       8        0         0       0      0       0
_sputc.o        16       0        0         0       0      0       0
nofpdisp.o       4       0        0         0       0      0       0
__vfpntf.o    1828       4       68         0       0      0       0
rtudiv10.o      40       0        0         0       0      0       0
strlen.o        68       0        0         0       0      0       0
ctype.o          0       0        0         0     260      0       0
ferror.o         8       0        0         0       0      0       0

              code  inline   inline   'const'      RW 0-Init   debug
              size    data  strings      data    data   data    data
Object totals  276      12      184         0       0      0       0
Library totals2020      12       68         0     260      0       0
Grand totals  2296      24      252         0     260      0       0
```

Running the ROM image

Run the ROM image under ARMul/armsd as follows:

```
armsd
getfile ex4_rom 0xf0000000
pc=0xf0000000
go
```

This should produce the following output:

```
ROM is at address f0000000, RAM is at address 10000000
```

9.3.2 Standalone C functions

The following functions are standalone functions and may be safely used in standalone ROM code.

```
<string.h>

memcpy  memmove memset  memcmp  strcpy  strncpy strcat  strncat
strcmp  strncmp strcoll strxfrm memchr  strchr  strcspn strpbrk
strrchr strspn  strstr  strtok

<ctype.h>
```

You must call the `_ctype_init()` function in your initialisation if you wish to use any of the `ctype.h` functions.

```
isalnum isalpha iscntrl isdigit isgraph islower isprint ispunct
isspace tolower toupper isxdigit

<math.h>
acos    asin    atan    atan2   cos     sin     tan     cosh
sinh    tanh    exp     frexp   ldexp   log     log10   modf
pow     sqrt    ceil    fabs    floor   fmod

<setjmp.h>
setjmp  longjmp

<stdlib.h>
atof    atoi     atol   strtod  strtol  strtoul rand    srand
bsearch qsort    abs    div     labs    ldiv    mblen   mbtowc
wctomb  mbstowcs wcstombs

<locale.h>
```

You must call the `_locale_init()` function in your initialisation if you wish to use any of the `locale.h` functions.

```
setlocale       localeconv
```

```
<stdio.h>
sprintf sscanf  vsprintf

<time.h>
mktime  asctime ctime   gmtime  difftime localtime strftime
```

9.4 Troubleshooting Hints and Tips

Problem

The linker reports one of the symbols `__rt_stkovf_split_big` or `__rt_stkovf_split_small` as being undefined.

Cause

You have compiled your C code with stack checking enabled. The C compiler generates code which calls one of the above functions when stack overflow is detected.

Solution

This problem may be fixed in one of the following ways:

- Recompile your C code with the `-apcs 3/noswst` option to disable stack checking.
- Link with a C library which provides support for stack limit checking (of the pre-built C libraries provided with the release only `armlib_n.32x` does not support stack limit checking)

  Note: This is usually only possible in an application environment as the C libraries stack overflow handling code relies heavily on the application environment.
- Write a pair of functions `__rt_stkovf_split_big` and `__rt_stkovf_split_small`. This will usually just generate an error for debugging purposes.

  The code might look similar to the following:

```
    EXPORT  __rt_stkovf_split_big
    EXPORT  __rt_stkovf_split_small
__rt_stkovf_split_big
__rt_stkovf_split_small
    ADR     R0, stack_overflow_message
    SWI     Debug_Message ; System dependent SWI to
                          ; write a debugging message
    B       .             ; and loop forever.
stack_overflow_message
DCB     "Stack overflow", 0
```

Problem

The linker generates an error similar to the following:

```
ARM Linker: (Warning) Attribute conflict between AREA
test2.o(C$$code) and image code.
ARM Linker: (attribute difference = {NO_SW_STACK_CHECK}).
ARM Linker: (Warning) Attribute conflict within AREA C$$code
(conflict first found with test2.o(C$$code)).
ARM Linker: (attribute difference = {NO_SW_STACK_CHECK}).
```

Cause

Parts of your code have been compiled or assembled with software stack checking enabled and parts without. Alternatively, you have linked with a library which has software stack checking enabled whereas your code has it disabled or vice versa.

Solution

Make sure all your code is compile/assembled with either `-apcs 3/noswst` or `-apcs 3/swst`.

Link with the correct library, of the pre-built libraries provided with the release the libraries `armlib.16x` and `armlib_i.32x` have stack checking disabled, all others have stack checking enabled.

Problem

The linker reports `__main` as being undefined.

Cause

When the compiler compiles the function `main` it generates a reference to the symbol `__main` to force the linker to include the basic C run time system from the C library. If you are not linking with a C library and have a function `main` you may get this error.

Solution

This problem may be fixed in one of the following ways:

- If the `main` function is only used when building an application version of your ROM image for debugging purposes, you should comment it out with a `#ifdef` when building a ROM image.

  Usually when building a ROM image you will call the C entry point something other than `main` such as `C_Entry` or `ROM_Entry` to avoid confusion.

- If you do need to have a function called `main` simply define a symbol `__main` in your ROM initialisation code.

  Usually this is defined to be the entry point of the ROM image so you should define it just before the ENTRY directive as follows.

  ```
              EXPORT __main
  __main
              ENTRY
  ```

- If you are building an application simply link with the appropriate C library.

Problem

The linker reports a number of undefined symbols of the form:

```
__rt_... or __16__rt_...
```

Cause

These are run time support functions which are called by code generated by the compiler to perform tasks which cannot be performed simply in ARM or Thumb code such as integer division or floating point operations.

For example, the following code will generate a call to the run time support function `__rt_sdiv` to perform a division.

```
int test(int a, int b)
{
      return a/b;
}
```

Solution

You should assemble file `examples/clstand/rtstand.s` and link this in. A Thumb version of this file is available in the `thumb` subdirectory.

Note The divide routines in `rtstand.s` use Demon SWIs to report division be zero. You may need to edit `rtstand.s` to change these SWIs if your system does not support them.

Problem

The linker produces the error message:

```
ARM Linker: (Fatal) No entry point for image.
ARM Linker: garbage output file aif removed
```

Cause

You have not defined an entry point. You must define the entry point even if the entry point is the start of the ROM image.

Solution

To define an entry point, use the assemblers `ENTRY` directive as shown in the example file `init.s` previously in this chapter.

Problem

The compiler produces errors of the form:

```
Serious error: illegal character (0x24 = '$') in source
```

Cause

The `$` character is not allowed in variable names as standard by ANSI although many compilers allow this.

Solution

Use the `-fc` option on the C compiler to tell it to allow `$` in variable names.

Problem

When loading an image into the ARMulator and trying to run it, the following error occurs:

```
*** Error: Can't go
```

Cause

armsd does not know the location at which it should begin executing your image.

Solution

Tell armsd where to start executing using the command:

```
pc = <address in hex>
```

Re-enter the `go` command.

If your image is to be executed from its base address, the address you specify above should be the same address as that used in the `getfile` command with which you loaded the image.

Problem

The image is bigger than expected (bigger than the size given by `-info sizes`).

This problem may also be caused by the image having a large section of zeros on the end of it.

Cause

By default, when generating a plain binary image, the linker expands zero initialised areas with zero bytes in the image.

The area will then be zero initialised when the image is loaded directly into memory.

Solution

Use the `-NoZeroPad` option to tell the linker not to expand the zero init area.

Problem

The image compiles and links without problem, but when loaded and disassembled from the base address, no initialisation code is present.

Causes

There are a number of possible causes:

- If the hex words look as though they are reversed instruction words, armsd may be using the wrong endianness.

  Solution
  Reconfigure your copy of armsd to the opposite endianness and try again.

- You may have linked it as an application image instead of a plain binary image. If the disassembly looks something like the following, then this is the case.

  ```
  0x10000000: 0xe1a00000  .... :    nop
  0x10000004: 0xe1a00000  .... :    nop
  0x10000008: 0xeb00000c  .... :    bl       0x10000040
  0x1000000c: 0xeb00001b  .... :    bl       0x10000080
  0x10000010: 0xef000011  .... :    swi      0x11
  ```

  Solution
  Relink with the `-bin` flag and without any `-aif` flag.

- The initialisation code may not be at the start of the image because you have omitted the `-First` option.

  Solution
  Try relinking with the `-First` option to see if this resolves the problem.

Problem

The image loads without problem but when trying to run, it crashes/hangs immediately.

Causes

Any of the causes in the previous problem may also apply here.

Another possibility is that it has been linked or loaded at the wrong address.

Solution

Check that the address is the same on each of the following:

- The linker's `-RO` option
- The `GetFile` command in armsd
- The `PC=` command in armsd

If all this is correct, try setting the PC to the start and using the `Step In` command to step through all the initialisation code to see if it is going wrong in the initialisation.

10 The ARMulator

This chapter describes the ARM processor software emulator, ARMulator.

10.1 The ARMulator

The ARMulator is a software emulator of the ARM processor forming part of ARM's debugger. It allows you to debug ARM programs on an emulated system. It can emulate any current ARM processor at the instruction level, including Thumb-aware processors.

The ARMulator consists of four parts:

- the model of the ARM processor, together with various utility functions to support system modelling

  This part of the ARMulator is not customisable and handles all communication with the debugger. It is supplied in object form only, on Unix hosts, and is built into the debugger in the Windows Toolkit.

- a memory interface which transfers data between the ARM model and the memory model or memory management unit model

  The memory model is fully customisable. Example implementations are provided with the ARMulator. Features such as models of memory-mapped I/O can be provided through the memory interface.

  Three memory models are provided with the ARMulator. `armfast` (a fast model of 512Kb of RAM) and `armvirt` (a slower model which models a full 4Gb of physical memory) use the full-blown memory interface (as described in the Software Development Toolkit Reference Manual). `armproto` provides a simpler memory interface allowing more rapid prototyping of memory models. An example of building such a model is provided in ➲*10.4 Rebuilding the ARMulator* on page 10-13.

- a coprocessor interface to optional ARM coprocessor models

  Although the ARM floating-point instruction set is implemented using ARM coprocessors, the ARMulator does not use a coprocessor model to emulate these on the host. The coprocessor model does not handle such instructions, so they are passed through the undefined instruction vector, and the ARM-code floating-point emulator (FPE400) emulates the operations.

  The default coprocessor model (`armproto.c`) provides a cut-down coprocessor #15 model, allowing software control of the processor's endianness, abort behaviour, etc. This model is fully customisable, and models of other co-processors can be added easily.

- an operating system interface to provide an execution environment

  The operating system model (supplied in `armos.c`) directly implements some operating system calls (such as open file, read the clock etc.) on the debugger host. These calls form the basis for the library calls (eg. `fopen()` and `time()`) provided by the ANSI C library.

  This part of the ARMulator is also fully customisable. Extra SWIs can be added to provide more host system functionality to the debuggee. SWIs that are not handled by this model take the SWI trap and can be handled by ARM SWI handler code running on the ARMulator.

By modifying or rewriting the supplied default models, you can make a model of almost any ARM system, and use it to debug code.

For simple modelling systems with different RAM types and access speeds, the `armvirt.c` memory model supports memory map files. See ➲*The ARM Software Development Toolkit Reference Manual: Chapter 14, ARMulator* for further details. User supplied memory models can also support map files using `armvirt.c` as a template.

A complete description of the API between the ARM debugger and the memory model, coprocessor model and operating system, and some additional calls for setting timed callbacks etc., can be found in ➲*The ARM Software Development Toolkit Reference Manual: Chapter 14, ARMulator*.

10.2 Using the ARMulator Rapid Prototype Memory Model

10.2.1 Overview

This section gives an example implementation of a memory system using the rapid prototype ARMulator memory model. The starting point for this model is the ARMulator `armproto.c` model.

It gives the implementation of an example `ARMul_MemAccess` function, and discusses methods for improving the efficiency of this model.

10.2.2 The memory model

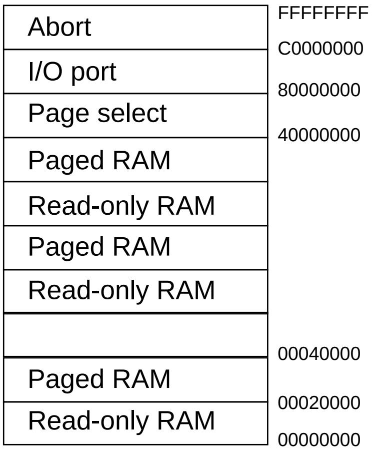
This example considers a device where memory is split into two 128Kb pages. The bottom page is read-only, and the top page has one of eight 128Kb memory pages mapped into it, page 0 being the low page. This type of system might be used to implement a small number of user tasks.

Addresses wrap around above 256Kb for the first 1gigabyte of memory, as if bits 29:18 of the address bus are ignored.

Bits 31:30 are statically decoded as follows:

bit 31	**bit 30**	
0	0	is a memory access
0	1	bits 18:16 of the address select the physical page mapped in to the top page
1	0	single byte I/O port (bits 23:16 of the address are written to the debugger's display)
1	1	generates an abort

This produces the following memory map:



10.2.3 Model data structures

There are eight banks of 128Kb of RAM, one of which is currently mapped in to the top page. The memory model has two pieces of information describing the state:

- An array representing the model of memory
- The number of the page currently mapped into the top page resulting in the use of a simple datastructure to store the memory model state. This datastructure is attached to the `MemDataPtr` in the ARMulator's state.

```
#define PAGESIZE (1<<17)

typedef union {
         char byte[PAGESIZE];
         ARMword word[PAGESIZE/4];
} page;

typedef struct {
         page *p[8];       /* eight pages of memory */
         int mapped_in;
} ModelState;
```

The example does not consider different endian modes. It assumes that the ARM is configured to be the same endianness as the host architecture.

```
#define OFFSET(addr) ((addr) & 0x7fff)
#define WORDOFF(addr) (OFFSET(addr)>>2)
unsigned ARMul_MemoryInit(ARMul_State *state,
                           unsigned long initmemsize)
```

```
{
  ModelState *s;
  int i;

  s=(ModelState *)malloc(sizeof(ModelState));
  if (memory==NULL) return FALSE;

  for (i=0;i<8;i++) {
        s->p[i]=(page *)malloc(sizeof(page));
        if (s->p[i]==NULL) return FALSE;
        memset(s->p[i], 0, sizeof(page));
   }

  state->MemDataPtr=(unsigned char *)s;
  s->mapped_in=0;

  state->MemSize=8*PAGESIZE;  /* ignore initmemsize */
  ARMul_ConsolePrint(state, ", 1Mb memory");

  /* Ask ARMulator to clear aborts for us regularly */
  state->clearAborts=TRUE;

  return TRUE;
}
```

The Exit function is shown below:

```
void ARMul_MemoryExit(ARMul_State *state)
{
  free(state->MemDataPtr);
}
```

Finally you need to write the generic access function:

```
ARMword ARMul_MemAccess(ARMul_State *state,
                        ARMword address,
                        ARMword dataOut,
                        ARMword mas1,
                        ARMword mas0,
                        ARMword Nrw,
                        ARMword seq,
                        ARMword Nmreq,
                        ARMword Nopc,
                        ARMword lock,
                        ARMword trans,
                        ARMword account)
{
```

```
int highpage=(address & (1<<17));
ModelState *s=(ModelState *)(state->MemDataPtr);
page *mem;

if (highpage)
  mem=s->p[s->mapped_in];
else
  mem=s->p[0];

if (Nmreq==LOW) {     /* memory request */
```

The memory models must track the numbers of N, S, I and C cycles that occur in the ARMulator. These counts are used to provide the `$statistics` and `$statistics_inc` variables in armsd.

```
  if (account) {      /* an ARMulator request */
    if (seq==LOW) state->NumNcycles++;
    else state->NumScycles++;
  }

  switch ((address>>30)&0x3) {
  case 0:             /* 00 - memory access */
    if (Nrw==LOW)
      return mem->word[WORDOFF(address)];
```

You do not need to extract the relevant byte or halfword presented on the data bus for byte or half-word loads, as the ARM will do this for you. Note that this is not true of the high speed memory interface.

```
   else             /* write - need to do right width access */
    /* Ignore writes out of supervisor mode to the "low" page */
     if (highpage || account==FALSE || state->NtransSig==LOW) {
```

Note The `trans` value supplied is not correct. Use the `NtransSig` in the `ARMul_State` instead.

```
        if (mas0==LOW) {      /* byte or word */
          if (mas1==LOW)      /* byte */
            mem->byte[OFFSET(address)]=dataOut;
          else
            mem->word[WORDOFF(address)]=dataOut;
        } else {              /* half-word */
          ARMword offset=OFFSET(address) & ~1;
          mem->byte[offset]=dataOut>>8;
          mem->byte[offset+1]=dataOut;
        }
      }
    break;
```

```
case 1:             /* 01 - page select in SVC mode */
```

To change the mapped in page:

```
       if (state->NtransSig==LOW || account==FALSE) {
         s->mapped_in=(address>>16) & 7;
       }
       break;

case 2:             /* 10 - single byte I/O */
       ARMul_ConsolePrint(state,"%c",(address>>16) & 0xff);
       break;

case 3:             /* 11 - generate an abort */
```

There are two types of abort:

- prefetch abort
- data abort

Use the appropriate macro. (A real ARM has only one abort pin.)

```
       if (Nopc==LOW) {
         ARMul_PREFETCHABORT(address);
       } else {
         ARMul_DATAABORT(address);
       }
       return ARMul_ABORTWORD;
       break;
     }
   } else {             /* not a memory request */
```

`MemAccess` is called for all ARM cycles, not just memory cycles, and must keep count of these I and C cycles.

```
     if (seq==LOW)       /* I-cycle */
       state->NumIcycles++;
     else
       state->NumCcycles++;
   }

   return 0;
 }
```

10.2.4 Improving performance

Whilst running an emulation of the ARM, the ARMulator spends a lot of time in the memory access functions. Small improvements in the efficiency of the memory model can give significant performance boosts.

In the simple case of this memory model, removing the need to look up the number of the currently mapped in top page on each access optimises the code. Instead, you can retain a pointer to it in the `ARMul_State`, on the `MemSparePtr`.

The modified `MemoryInit` is shown below:

```
unsigned ARMul_MemoryInit(ARMul_state *state,
                          unsigned long initmemsize)
{
  page *memory;

  memory=(page *)calloc(8, sizeof(page));
  if (memory==NULL) return FALSE;

  state->MemDataPtr=(unsigned char *)memory;

  /* attach page zero to the top page pointer */
  state->MemSparePtr=(unsigned char *)(&memory[0]);

  ...
}
```

`MemAccess` is similar to the following:

```
...
{
  int highpage=(address & (1<<17));
  page *mem;

  mem=(page *)(highpage ? state->MemDataPtr    /* high */
                        : state->MemSparePtr); /* low */
  ...
```

Only one load from memory is required to get the address of the page giving great improvement in performance.

In other memory models, you can also improve performance by:

- Using a form of tree to model the memory map.

  The memory-model performs access checks on memory cycles and these access permissions can be encoded by using multiple trees: one that maps the real physical memory, one that maps pages that are readable, and one that maps pages that are writable.

  Using such a system, the model need not look up whether a page is read/write-able before an access takes place. It can gain a handle onto the page from the appropriate tree structure, and test it's validity. This may improve performance of the common case, where there is no exception, at the cost of slowing down the uncommon cases where an exception occurs, or where a page has yet to be allocated in the memory model.

  A similar trick can be used for distinguishing between cached and uncached memory, or for modelling memory-mapped I/O devices.

- Using the distinction between S- and N-cycles, and caching a pointer into the memory model on non-sequential cycles which is used on sequential cycles. This is similar to the page-mode access provided by DRAM hardware.

  Take care because in some circumstances the ARM will perform an I-cycle followed by an S cycle and not the expected N cycle. On ARM6 and ARM7 these merged I-S cycles occur for the N-cycle instruction prefetch following a data load. The ARM (and ARMulator) will perform a cycle marked as sequential where the address is not actually sequential from the previous memory access. However the address will be sequential from the previous instruction fetch, so a sequential instruction fetch will always be sequential from the previous instruction fetch, on current processors.

  On a model which uses, for example, a deep tree structure or a hash table, this trick can be used to remove the need to search the model, instead allowing the program to immediately find the appropriate page.

Using these and similar techniques ARM have developed a complete model of the ARM610 memory system (physical memory, cache and MMU based translation unit) which only has a 25% overhead over the standard `armvirt` memory model.

10.3 Writing Custom Serial Drivers for ARM Debuggers

In addition to the ARMulator target, the ARM debuggers can be used to debug a remote target, using the remote debug protocol (RDP). RDP is a byte-stream protocol, allowing communication to take place over any kind of channel, providing that drivers are written for both ends of the link.

At the DEMON end, the drivers should be written in C or ARM assembler as part of the port of DEMON to your target hardware.

At the debugger end of the link, the new drivers must be included as part of armsd/windbg. This involves adding a module into armsd/windbg, using the supplied prebuilt objects and makefile. For Win32 tools (Windows95 and Windows NT), the remote drivers are packaged into a DLL so they can be easily replaced with a user-supplied DLL. This is because the debugger (`armsd.exe` or `windbg.exe`) will not need to be rebuilt, and the same DLL will be used for both debuggers. Under Unix/DOS the entire tool needs to be rebuilt. See ➲*10.4 Rebuilding the ARMulator* on page 10-13 for further details.

It is possible to rebuild the RDP drivers for the following hosts:

SunOS

HP/UX

Macintosh

Windows 95 & Windows NT

DOS

It is NOT possible to rebuild the RDP drivers for:

Windows 3.1	the Windows 3.1 serial/parallel drivers are complicated by extra 'thunking' between the 32-bit application and 16-bit Windows. Users wishing to rebuild the RDP drivers should upgrade to Windows 95 or Windows NT.

10.3.1 Supplied serial/parallel driver source files

In the Software Development Toolkit, sources are supplied (in the `armsd/source` directory) for serial and parallel drivers. These are described below.

In the Windows Toolkit, sources are supplied for Win32 serial and parallel drivers. The source for the supplied combined 16-bit and 32-bit drivers is not supplied.

The API to the drivers is the same on both systems:

`serdrive.h`	This is a header file for the RDP I/O interface. It defines a `DriverDesc` struct which contains function pointers for routines to open, close, read, write the comms link. This is the official interface between the debugger and the RDP comms link.

`serdrive.c` This is an implementation of the serial driver.

`spdrive.c` This is an implementation of the serial/parallel driver.

`drivers.c` New drivers are added to the debugger by including a pointer to their `DriverDesc` in the `drivers` array defined by `drivers.c`

`pirdi.h` Defines functions used by `hostos.c` to read and write bytes across the RDP link.

In addition the DOS serial drivers include code for interrupt-driven serial I/O (in the file `comsisr.c`).

Any new driver must provide a `DriverDesc` structure, defined in `serdrive.h` and add this structure to the `DriverList` defined in `drivers.h`. The elements of this structure are:

`name` This should be a unique name for the driver, which is used by armsd as a command-line argument. For example the standard serial driver defines the name "SERIAL".

`OpenProc` This function opens a connection, and returns a handle onto it. This handle is passed into the other driver functions. On the standard Unix serial port drivers it merely opens the appropriate serial port and returns the (Unix) file handle.

`ConfigProc` The configuration function is used to set the linespeed on the connection and it initialise it.

`ReadProc WriteProc` These functions read and write data across the link. The RDP assumes that the link is error free, and that characters are not lost (must have flow-control or adequate buffering).

`CloseProc` The `CloseProc` is called when the RDP wishes to close the connection.

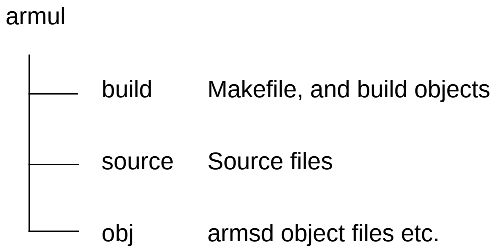
`LoggingProc` This is called when the level of RDP logging is changed. (e.g. when the `$rdi_log` variable is changed in armsd). For a description of the meaning of the logging values, see ➲*The ARM Software Development Toolkit Reference Manual: Chapter 7, Symbolic Debugger*.

10.4 Rebuilding the ARMulator

Under Unix, DOS, and on the Macintosh the ARMulator is linked onto a core debugger, to produce an armsd executable image. Under Windows, the ARMulator exists as a Windows DLL.

10.4.1 Rebuilding armsd under Unix/DOS

The following diagram shows the ARMulator source tree for Unix and DOS:



New sources, e.g. memory models, serial drivers, etc., should be placed in the `source` directory.

Any new memory models can be added to the ARMulator `Makefile` by adding a rule to it for the model. For example:

```
example.o: $(SRC)example.c $(HFILES)
        $(CC) $(CFLAGS) -c $(SRC)example.c
```

Any new serial drivers also need rules adding (similar to the above), but also need to be added to the OFILES list of object files at the top of the Makefile to be linked into the resulting armsd. You also need to declare the driver in `drivers.c`.

An ARMulator can then be built using `make`:

```
make MODEL=example
```

This compiles to an ARMulator based armsd which uses the `example` memory model. By default, armsd will be rebuilt with the `armvirt` memory model.

10.4.2 Rebuilding ARMULATE.DLL under Windows

The process required to rebuild the `ARMULATE.DLL` component is very similar to that described above.

```
WIN32                              Rebuild parent directory for WIN32 components
  |
  |---- ARMBG                      ARM Debugger toolbox components
  |
  |---- ARMUL                      ARMulator components
  |       |
  |       |---- MSVC20
  |       |       |___ DLL         Microsoft Visual C++ V2.0+ DLL rebuild directory
  |       |
  |       |____ WATCOM10
  |               |___ DLL         Watcom C/C++ V10.0a DLL rebuild directory
  |
  |---- CLX
  |
  |---- CUSTOM                     Customisable resources for Windows Tools
  |
  |____ REMOTE                     WIN32 serial driver components
          |
          |---- MSVC20
          |       |___ DLL         Microsoft Visual C++ V2.0+ DLL rebuild directory
          |
          |____ WATCOM10
                  |___ DLL         Watcom C/C++ V10.0a DLL rebuild directory
```

There is a choice of compilers for rebuilding the ARMulator DLL, Microsoft Visual C++ (produces the fastest code, but only runs under 32-bit Windows - Windows NT or Windows 95) and Watcom C/C++ V10.0a; the rebuild kit provides makefiles for both these compilers.

As described above, new rules may be added to the ARMulator DLL makefile `ARMULATE.MAK`.

When complete, the make procedure will produce a file called ARMULATE.DLL, this should be placed in the BIN sub-directory of the ARM Tools200 installation directory, typically `C:\ARM200\BIN`.

The new DLL will automatically be used on the next invocation of either the ARM Debugger for Windows or armsd (on 32-bit Windows operating systems).

Rebuilding the serial driver DLL `REMOTE.DLL` is accomplished in the same way (using `REMOTE.MAK`). However, it should be noted that the rebuild kit provided is for 32-bit Windows operating systems only, i.e. Windows NT or Windows 95.

11 Exceptions

This chapter explains how the ARM deals with exceptions, and discusses the issues involved in writing exception handlers.

Exceptions

11.1 Overview

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

Exceptions occur when this normal flow of execution is diverted, so that the processor can 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

It is necessary for the handling of such exceptions to preserve the previous processor state so that execution of the original user program can resume once the appropriate exception routine has been completed.

The ARM recognises seven different types of exception, as shown below:

Exception	Description
Reset	Occurs when the CPU reset pin is asserted. Only expected to occur for signalling power-up, or for resetting as if the CPU has just powered up. It can therefore be useful for producing soft resets.
Undefined Instruction	Occurs if neither the CPU nor any attached coprocessor recognises the currently executing instruction.
Software Interrupt (SWI)	Is a user-defined synchronous interrupt instruction, so that a program running in User mode can request privileged operations which need to be run in Supervisor mode.
Prefetch Abort	Occurs when the CPU attempts to execute an instruction which has prefetched from an *illegal* address, ie. an address that the memory management subsystem has determined as inaccessible to the CPU 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 CPU's external interrupt request pin is asserted (low) and the I bit in the CPSR is clear.
FIQ	Occurs when the CPU's external fast interrupt request pin is asserted (low) and the F bit in the CPSR is clear.

Table 11-1: Exception types

11.1.1 The vector table

Exception handling is controlled by a *vector table.* This is a reserved area of 32 bytes at the bottom of the memory map with one word of space allocated to each exception type (plus one word currently reserved for handling address exceptions when the processor is configured for a 26-bit address space). 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 or load PC instruction to continue execution with the appropriate handler.

11.1.2 Use of modes and registers by exceptions

As a rule, the user program runs in *User mode*, but the servicing of the exceptions requires privileged (ie. 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>)
- and in the case of FIQ, five other general purpose registers (r8_FIQ to r12_FIQ)

Each exception handler must ensure that other registers are restored to their original state upon exit. This can be done by storing the contents of any registers the handler needs to use onto its stack and restoring them before returning.

Note *You must ensure that the required stacks have been set up. If you are using Demon or ARMulator, this is done for you.*

11.1.3 Exception priorities

Several exceptions can occur simultaneously, and they are serviced in a fixed order of priority. Each exception is handled in turn before execution of the user program continues. However, it is not possible for all exceptions to occur concurrently. For instance, the undefined instruction and SWI exceptions are mutually exclusive as they both correspond to particular decodings of the current instruction.

Placing the Data Abort exception above the FIQ exception in the priority list ensures that 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. Once 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 11-2: The exception vectors* on page 11-4 shows the processor modes, the exceptions that give rise to them and the priority in which the exceptions are handled.

Vector Address	Exception Type	Exception Mode	Priority (1=High, 6=Low)
0x0	Reset	svc	1
0x4	Undefined Instruction	undef	6
0x8	Software Interrupt (SWI)	svc	6
0xC	Prefetch Abort	abort	5
0x10	Data Abort	abort	2
0x14	*Reserved*	*Not applicable*	*Not applicable*
0x18	Interrupt (IRQ)	irq	4
0x1C	Fast Interrupt (FIQ)	fiq	3

Table 11-2: The exception vectors

11.2 Entering and Leaving an Exception

11.2.1 The processor’s response to an exception

When an exception is generated, the processor:

1. Copies the *Current Program Status Register* (CPSR) into the *Saved Program Status Register* (SPSR) for the mode in which the exception will be handled.

   This saves the current mode, interrupt mask and condition flags.

2. Sets the appropriate CPSR mode bits:

   a) to change to the appropriate mode, also mapping in the appropriate banked registers for that mode.

   b) to disable interrupts.
   IRQs are disabled once any other exception occurs, and FIQs are also disabled when a FIQ occurs.

3. Stores the *return address* (PC – 4) in LR_<mode>.

4. Sets the PC to the appropriate vector address.

   This forces the branch to the appropriate exception handler.

11.2.2 Returning from an exception handler

Two actions need to take place to return execution to the place where the exception occurred:

1. Restore the CPSR from the SPSR_<mode>.

2. Restore the PC using the *return address* stored in LR_<mode>.

These can be achieved in a single instruction, because adding the S flag (update condition codes) to a data-processing instruction when in a privileged mode with the PC as the destination register, also transfers the SPSR to CPSR as required. This also applies to the Load Multiple instruction (using the ^ qualifier).

11.3 The Return Address and Return Instruction

The actual value in the PC which causes a return from a handler varies with the exception type. When an exception is taken, the PC may or may not have been updated, and the return address may not necessarily be the next instruction pointed to by the PC, because of the way the ARM loads its instructions.

When loading the instructions it needs to execute a program, the ARM uses a pipeline with a fetch, decode and execute stage.

At any one time, there will be one instruction in each stage of the pipeline. The PC actually points to the instruction being *fetched*. Since each instruction is a word long, the instruction being decoded is at address PC – 4 and the instruction being executed is at PC – 8.

11.3.1 Returning from SWI and undefined instruction

The SWI and Undefined Instruction exceptions are generated by the instruction itself, so the PC has not been 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 PC from the LR returns control from the handler:

```
MOVS pc, lr
```

11.3.2 Returning from FIQ and IRQ

After executing each instruction, the CPU checks the interrupt pins and interrupt disable bits in the CPSR. This means that an IRQ or FIQ exception is only ever generated after the PC has been updated, and consequently 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 (or four bytes) less than that in LR_<mode>, so the return instruction is:

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

11.3.3 Returning from prefetch abort

A prefetch abort is not generated at the time the CPU attempts to fetch an instruction from an illegal address. Instead, the instruction is flagged as invalid, and the execution of instructions already in the pipeline continues until the invalid instruction reaches the execute stage, when the exception is generated.

The handler gets the MMU to load the appropriate virtual memory locations into physical memory. Having done this, it must return to the offending address and reload the instruction, which should now load and execute correctly.

As the PC will not have been updated at the time the prefetch abort was issued, LR_ABORT will point to the instruction following the one that caused the exception. The handler must therefore return to LR_ABORT – 4 using:

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

11.3.4 Returning from data abort

When a load or store instruction tries to access memory, the PC has already been updated, so that storing PC – 4 in LR_ABORT makes it point to two instructions beyond the address where the exception was generated. Once the MMU has loaded the appropriate address into physical memory, the handler should return to the original, aborted instruction so a second attempt can be made to execute it. The return address is therefore two words (or eight bytes) less than that in LR_ABORT, making the return instruction:

```
SUBS pc, lr, #8
```

11.4 Writing an Exception Handler

This section explains the functions performed by the code that handles each type of exception.

11.4.1 SWI handler

When the SWI handler is entered, it decides which SWI is being called. This information is stored in bits 0-23 of the instruction itself (see *Figure 12-2: The SWI instruction* on page 12-4).

The handler must load the SWI instruction that caused the exception so that it can examine these bits. It does this using the address stored in LR_SVC, which is the address of the instruction which follows the SWI. Therefore the SWI is loaded into a register (in this case r0) using:

```
LDR r0, [lr,#-4]
```

and the SWI number is extracted by clearing the top eight bits of the opcode:

```
BIC r0, r0, #0xff000000
```

The resulting value can then be used in a C `switch()` statement or an assembly language lookup table to branch to the routine which implements the relevant SWI.

Note *Because of the need to access the link register and load in the actual SWI instruction, the top-level SWI handler must be written in assembly language. However, the individual routines that implement each SWI can be written in C if required: see Chapter 12, Implementing SWIs.*

11.4.2 Interrupt handlers

The ARM has two levels of external interrupt:

- FIQ
- IRQ

FIQs have higher priority than IRQs in two ways:

1. FIQs are serviced first when multiple interrupts occur.
2. Servicing an FIQ causes IRQs to be disabled, preventing them from being serviced until after the FIQ handler has re-enabled them (usually by restoring the CPSR from the SPSR at the end of the handler).

You can set up C functions as interrupt handlers by using the special function declaration keyword `__irq`.

This keyword:

- preserves all registers (excluding floating-point)
- exits the function by setting the PC to LR – 4 and restoring the CPSR to its original value.

The simple example handler below reads a byte from location 0xc0000000 and writes it to location 0xc0000004:

```
void __irq IRQHandler (void)
{      volatile char *base = (char *) 0xc0000000;
       *(base+4) = *base;
}
```

Installing the FIQ handler

The FIQ vector is the last entry in the vector table, at address 0x1C. It is situated there 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 simplest way to place the FIQ handler at 0x1c is to copy it there. ARM code is inherently relocatable, but note that:

- the code should not use any absolute addresses
- PC-relative addresses are allowable as long as the data is copied as well as the code (so that it remains in the same relative place)

The five extra FIQ mode banked registers mean that status can be held between calls to the handler

A simple FIQ handler is shown below. This takes some data and copies it out to an i/o port, using the following registers:

r8	points to the i/o port (with an interrupt flag at r8 + 4)
r9	points to the current word in the data
r10	points to the end of data
r11	is used as temporary storage
r12	points to a semaphore which is set when the copy is complete

```
FIQ_Start; Note no stack usage - banked registers
       STR r8, [r8,#4];set int_flag in port
       CMP r9, r10;End of data reached?
       LDRNE r11,[r9],#4;Read in next word
       STRNE r11, [r8];Copy it to port
       STREQ r8,[r12];Set semaphore when finished
       SUBS pc,lr,#4; Return
FIQ_End
```

This can be copied to the bottom of the vector table with:

```
memcpy (0x1c, FIQ_Start, FIQ_End-FIQ_Start);
```

11.4.3 Reset handler

The operations carried out by the Reset handler depend upon the system that the software is part of. It might, for example:

- do a hardware self test
- detect how much memory is available
- initialise stacks and registers
- initialise peripheral hardware such as i/o ports
- initialise the MMU if one is being used (ARM cached processors)
- call the main body of code (`__main()` if using C)

See ➲*Chapter 9, Writing Code for ROM* for a detailed example.

11.4.4 Undefined instruction handler

Any instructions that are not recognised by the CPU are first of all offered to any coprocessors attached to the system. If the instruction remains unrecognised, an undefined instruction exception is generated. It is still possible that the instruction is intended for a coprocessor, but the relevant coprocessor (eg. 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, storing the old contents.
2. Examine the undefined instruction to see if it should be emulated.

   This is similar way to the way a SWI handler extracts the number of a SWI, but rather than extracting the bottom 24 bits, the emulator must extract bits 24 to 27, which determine if the instruction is a coprocessor operation:

   - If bits 27-24 = 1110 or 110x, 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.
   - 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.

Once any 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.

11.4.5 Prefetch abort handler

If the system contains no MMU, the Prefetch Abort handler can simply report the error and quit. If there is an MMU, the address that caused the abort needs to be restored into physical memory. LR_ABORT points to the instruction at the address following the one that caused the abort, so the address that needs restoring is at LR_ABORT – 4. Thus the virtual memory fault for that address can be dealt with and the instruction fetch re-attempted. The handler should therefore return to the same instruction, rather than the following one.

11.4.6 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 virtual memory fault needs dealing with.

The instruction which caused the abort is at LR_ABORT – 8 (since LR_ABORT points two instructions beyond the instruction that caused the abort).

There are three possible cases of instruction that can cause this abort:

1. Single Register Load or Store:

   a) If the CPU is in early abort mode (ARM6 only), the address register will not have been updated (if writeback was requested).

   b) If the CPU is in late abort mode, if writeback was requested the address register will have been updated. The change will need to be undone.

2. Swap:

   There is no address register update involved with this instruction.

3. Load / Store Multiple:

   If 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 will handle replacing the overwritten value with the modified base value in such a way that recovery is possible.) The number of registers involved will therefore need to be used to recalculate the original base address.

In each case, the MMU can load the required virtual memory into physical memory (the address which caused the abort being stored in the MMU’s *Fault Address Register* (FAR)). Once this is done, the handler can return and retry executing the instruction.

11.5 Installing an Exception Handler

Once a handler for a particular exception has been written, it must be installed in the vector table so that it will be executed when the exception occurs.

11.5.1 Branching to the handler

The simplest method of installing a handler is to place a branch to it in the vector table. The limitation of this method is that the branch instruction only has a range of 32Mbytes.

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. Shift the result right by two to give a word offset, rather than a byte offset.
5. Test that the top eight bits of this are clear, thus ensuring that the result is only 24 bits long (as the offset for the branch is limited to this).
6. Logically OR this with 0xea000000 (the opcode for the BAL instruction) to produce the value to be placed in the vector.

A C function which implements this algorithm is provided below. This takes as its arguments the address of the handler and the address of the vector in which the handler is to be to installed. The function installs the handler and returns the original contents of the vector. This result might be used for creating a chain of handlers for a particular exception.

```
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 32Mbytes from 'vector'.*/
{      unsigned vec, oldvec;
       vec = ((routine - (unsigned)vector - 0x8)>>2);
       if (vec & 0xff000000)
       {      printf ("Installation of Handler failed");
              exit (0);
       }
       vec = 0xea000000 | vec;
       oldvec = *vector;
       *vector = vec;
       return (oldvec);
}
```

Code to call this to install an IRQ handler might be:

```
unsigned *irqvec = (unsigned *)0x18;
Install_Handler ((unsigned)IRQHandler,(unsigned)irqvec);
```

In this case the returned, original contents of the IRQ vector are discarded.

11.5.2 Long-distance branches

In most circumstances, the branch instruction's 32Mbyte range will be sufficient to reach the appropriate handler from the vector table. In cases where the handler routine is further than 32Mbytes from the vector table, the PC can be forced to the required address by:

1. Storing the address of the handler in a suitable memory location.
2. Placing in the vector, the encoding of an instruction to load the PC with the contents of the chosen memory location.

The following C routine implements this:

```
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) | 0xe59ff000

       oldvec = *vector;
       *vector = vec;
       return (oldvec);
}
```

Code to call this to install an IRQ handler might be:

```
unsigned *irqvec = (unsigned *)0x18;
unsigned *irqaddr = (unsigned *)0x38; /* For example */
*irqaddr = (unsigned)IRQHandler;
Install_Handler (irqaddr,irqvec);
```

Again in this case the returned, original contents of the IRQ vector are discarded.

11.6 Exception Handling on Thumb-Aware Processors

Note *This section only applies to processors that implement ARM Architecture 4T.*

When writing exception handlers suitable for use on Thumb-aware processors, there are some further considerations to those already described in this chapter.

The basic exception handling mechanism on Thumb-aware processors is the same as that of non-Thumb-aware processors, where an exception causes the next instruction to be fetched from the appropriate vector table entry.

The same vector table is used for both Thumb state and ARM state exceptions. This means that an initial step must be added at the top of the exception handling procedure described in ➲*11.2.1 The processor's response to an exception* on page 11-5. The procedure now reads:

1. Check the processor's state. If it is operating in Thumb state, switch to ARM state.
2. Copy the CPSR into SPSR_<mode>
3. Set the CPSR mode bits:
4. Store the return address (PC – 4) in LR_<mode>.
5. Set the PC to the appropriate vector address

The switch from Thumb state to ARM state in step 1ensures that the ARM instruction installed at the appropriate vector (either a branch or a PC-relative load) is correctly fetched, decoded and executed. Execution then moves to a top-level veneer, also written in ARM code, which saves the processor status and any registers. The programmer then has two choices.

1. Write the whole exception handler in ARM code.
2. Make the top-level veneer store any necessary status, and then perform a `BX` (branch and exchange) to a Thumb code routine which handles the exception.

   Such a routine will need to return to an ARM code veneer in order to return from the exception, since the Thumb instruction set does not have the instructions required for restoring the CPSR from the SPSR.

This second strategy is shown in ➲*Figure 11-1: Handling an exception in Thumb state* on page 11-15.

11.6.1 Returning from exceptions

When an exception is taken, the processor computes the LR_<mode> value in such a way that the instruction used to exit the handler will be independent of the state in which the exception occurred.

For example, the following instruction always exits from a FIQ handler correctly, regardless of whether the FIQ was raised in ARM state (with 32-bit instructions in the pipeline) or in Thumb state (with 16-bit instructions in the pipeline).

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

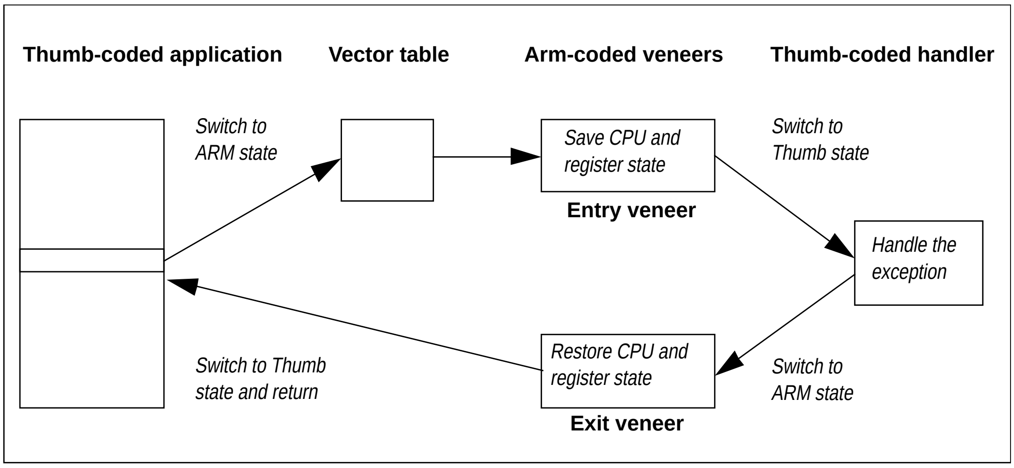


Figure 11-1: Handling an exception in Thumb state

11.6.2 Handling SWIs in Thumb state

Some handlers need to decide what state the processor was in when the exception occurred. This can be done by examining the T bit in the SPSR:

```
T_bit  EQU    0x20      ; Thumb bit of CPSR/SPSR, ie. bit 5.
       :
       :
       MRS    r0, spsr  ; move SPSR into general-purpose register
       TST    r0, #T_bit; Test if bit 5 is set
       BEQ    T_handle  ; T bit set - exception occurred in Thumb
                        ; state
       BNE    A_handle  ; T bit clear - exception occurred in ARM
                        ; state
```

An example of where this would be needed would be in a SWI handler. Both ARM and Thumb instruction sets contain SWI instructions. See *11.4.1 SWI handler* on page 11-8 and in *Chapter 12, Implementing SWIs* for more information.

When handling a Thumb SWI instruction, three things need to be taken into account:

1. The address of the instruction will be at LR – 2, rather than LR – 4.
2. A halfword load is required to fetch the instruction.
3. There are only 8 bits available for the SWI number instead of the ARM version’s 24 bits.

The following fragment of ARM code will handle a SWI from either source:

```
MRS     r0, spsr         ; move SPSR into general purpose register
TST     r0, #T_bit       ; Test if bit 5 is set
LDRHEQ  r0,[lr,#-2]      ; T_bit set so load halfword (Thumb)
BICEQ   r0,r0,#0xff00    ; and clear top 8 bits of halfword
                         ; (LDRH clears top 16 bits of word)
LDRNE   r0,[lr,#-4]      ; T_bit clear so load word (ARM)
BICNE   r0,r0,#0xff000000; and clear top 8 bits of word
```

12 Implementing SWIs

This chapter explains how to implement SWIs and how to call them from your programs.

12.1 Introduction

This chapter explains the steps involved in writing and installing a Software Interrupt (SWI) handler that is able to deal with SWIs in your application code.

It also examines the additions required to allow a user SWI handler to cooperate with the Debug Monitor (Demon) SWI handler when developing on a PIE card.

For additional information, refer to:

- *Chapter 3, Programmer's Model*, which explains the ARM's usage of modes and banked registers
- *Chapter 11, Exceptions*, which gives a general guide to writing exception handlers

12.1.1 What is a SWI?

A SWI is a user-defined synchronous interrupt instruction. It provides the means for a program running in User mode to request privileged operations which need to be run in Supervisor mode. The services provided by an operating system to the user for input/output are examples of such operations. To the CPU, a SWI is an exception.

12.1.2 What happens when a SWI instruction is executed?

When the CPU executes a SWI, it:

1. copies the *Current Program Status Register* (CPSR) into the Supervisor mode *Saved Program Status Register* (SPSR_SVC)

   This saves the current mode, interrupt mask and condition flags.

2. sets the CPSR mode bits to cause a change to Supervisor mode

   This maps in the banked *Stack Pointer* (SP_SVC) and *Link Register* (LR_SVC).

3. sets the CPSR IRQ disable bit

   This means that the SWI handler will execute without ordinary interrupts being taken. The FIQ disable bit is not set, so fast interrupts can still be taken. You can choose to turn IRQs back on, or disable FIQs, within the handler itself.

4. stores the value (PC – 4) into LR_SVC

   This means that the link register now points to the next instruction to be executed when the SWI has been handled. The `SWI` itself is located at PC – 8.

5. forces the PC to 0x8

   Address 0x8 is the SWI entry in the vector table. Typically, this will contain a branch instruction to the handler.

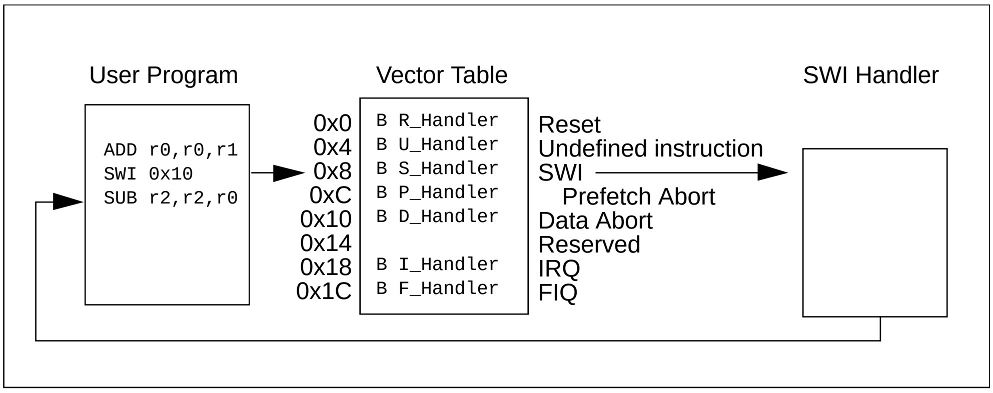


Figure 12-1: SWI execution path

12.1.3 Returning from a SWI

Once it has finished, the SWI handler returns control to the calling program by:

- copying SPSR_SVC into the CPSR

  This restores the condition codes, interrupt mask and the mode which were in force when the SWI was encountered.

- restoring the PC from LR_SVC

  This resumes execution of the program from the instruction following the SWI.

These two operations must be carried out in one instruction. If the return address is held in the link register (r14), the instruction:

```
MOVS pc, lr
```

not only moves the address to the PC, but also restores the processor status. This is because in privileged modes the `MOVS` instruction, with the PC as the destination register, causes the SPSR to be copied to the CPSR.

If the return address was previously saved on the stack, the SWI handler can use an instruction of the form:

```
LDMFD sp! {r0-r12, pc}^
```

to exit the SWI. The use of **^** in a privileged mode, with the PC as a destination register as shown above, causes the SPSR to be copied to the CPSR.

12.1.4 Decoding the SWI instruction

The handler's first task is to decode the SWI number to decide which function to perform. The SWI number is stored within the SWI instruction itself as a 24-bit field, giving a range of 0 to 0xFFFFFF. This is shown in ➲*Figure 12-2: The SWI instruction*, below.

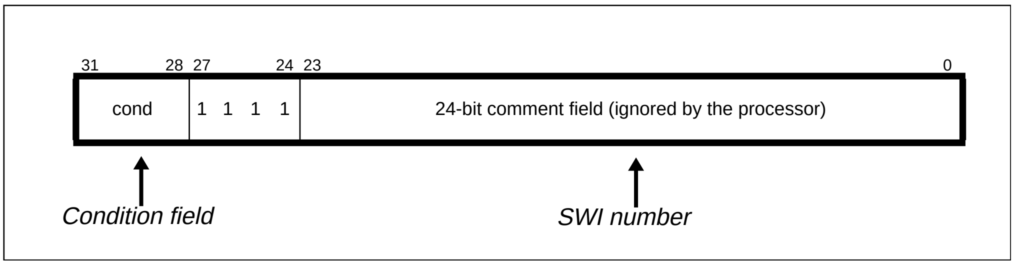


Figure 12-2: The SWI instruction

The handler's first task is therefore to locate the instruction, so it can read the SWI number. When a SWI instruction is executed, LR_SVC is set to PC – 4, so the instruction is located at LR_SVC –4.

Because LR_SVC can only be accessed via assembly language, so the code that obtains the SWI number must be written in assembler. The following two lines extract the SWI number and place it in r0:

```
LDR r0,[lr,#-4]            ; Load the SWI instruction into r0
BIC r0, r0, #0xff000000    ; Mask out the top 8 bits.
```

The part of the handler that actually implements the SWIs can be written in assembly language (using r0 to control execution through a jump table) or as a C subroutine (with r0 being passed as a parameter that controls a `switch()` statement). The last part of the handler will again need to be in assembly language because of the need to restore the CPSR from SPSR_SVC.

12.1.5 Stack and register usage

The system's SWI calling standard (defined by the system designer) will specify which registers have to be preserved across calls. The SWI handler must adhere to the standard, and avoid corrupting the specified registers, otherwise the behaviour of the user program will be unpredictable.

Since the handler will require working registers, it needs to save all the non-banked registers which must be preserved (Supervisor mode has banked SP and LR only). This description assumes that all registers must be preserved except for those which pass back a result. On a real system this might result in too much of a performance overhead, and the number of saved registers would have to be reduced accordingly.

12.1.6 Re-entrant SWI handling

Unless your SWI handler is written to be *re-entrant*, it will be unable to use SWIs itself because taking another exception while in Supervisor mode will corrupt both SPSR_SVC and LR_SVC. The second exception will return correctly to the first, but the first will be unable to return to the calling program.

If the handler stores SPSR_SVC and LR_SVC, along with the non-banked registers each time it is called and then retrieves them again each time it exits, this problem will not arise, as each instance will have access to the correct return address and status information.

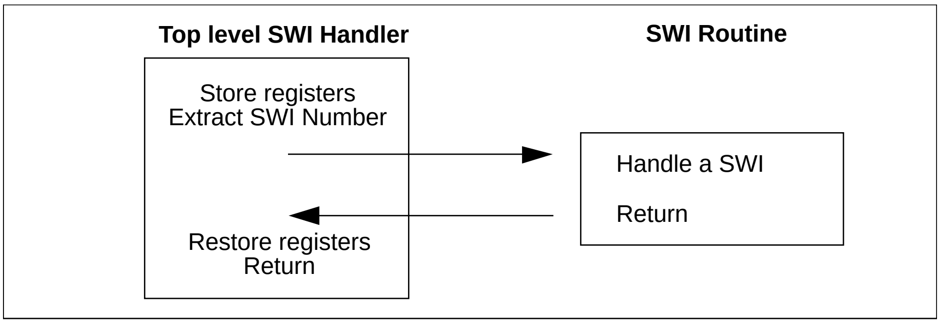


Figure 12-3: Ensuring re-entrancy

Supervisor mode has its own banked SP pointing to its own stack, so the registers can all be stored there. However, note that the SPSR cannot be stored on the stack directly, but must be copied onto the stack using an intermediate general purpose register. The following fragment of code shows how this is done:

```
SUB sp, sp, #4             ; Leave room on stack for storing SPSR
STMFD sp!, {r0-r12,lr}     ; Store gp registers
MOV r1,sp                  ; Pointer to parameters on stack
MRS r2, spsr               ; Get SPSR into gp register
;
; Extract SWI number and place in r0 as above
;
STR r2, [sp,#14*4]; Store SPSR above gp registers
;
; Handle this particular SWI
;
LDR r2, [sp,#14*4]         ; Restore SPSR from stack
MSR spsr, r2
LDMFD sp!, {r0-r12,lr}     ; Unstack registers
ADD sp, sp, #4             ; Remove space used for storing SPSR
MOVS pc, lr                ; Return from handler.
```

Passing parameters via the stack

Storing the registers has another advantage. A SWI will often be executed with a set of parameters, passed in registers. These can be accessed easily in assembly language, but you may have written your SWI handler in C. In this case, you can pass the SWI number in r0 (as extracted above), with r1 pointing to the location of the registers on the stack. See ➲*12.2.1 Implementing a SWI handler in C* on page 12-7.

Note that the stack used here is the Supervisor stack so before the SWI Handler is called, it must have been set up to point to a dedicated area of memory. In a final system this might be done in the `__main` routine within `rtstand.s`, the standalone C library. It might also be necessary to add code for stack overflow checking to the top level SWI Handler, as follows:

```
SVCStackBase     EQU 0xA00    ; Full descending stack so base
SVCStackEnd      EQU 0x800    ; is higher in memory than end.
SVCStackHeadroom EQU 0x40     ;Allow headroom of 16 words, even
                              ; though maximum handler places on
                              ; stack is 15 words, because can then
                              ; use (8 bit shifted) immediate value
                              ; in the CMP.
SVCStackLimit    EQU SVCStackEnd + SVCStackHeadroom
       :
       :
       :
       MOV sp, #SVCStackBase  ; Set up SVC stack pointer
       :                      ; (in rtstand.s, say)
       :
       :
SWIHandler
       CMP sp, #SVCStackLimit ; Check if enough room on stack to
       BLS stack_overflow     ; store registers, if not report error
       ;
       ; Rest of SWI Handler code
```

12.2 Implementing a SWI Handler

12.2.1 Implementing a SWI handler in C

The easiest way of implementing the SWI handling mechanism is to write it in C, using a `switch()` statement. Suppose we have the following function:

```
void C_SWI_handler (unsigned number, unsigned *reg)
{    /* Handle the SWIs */
}
```

the actual body of which is in the format:

```
switch (number)
{    case 0 : /* SWI number 0 code */
             break;
     case 1 : /* SWI number 1 code */
             break;
     /* Rest of SWI routines */
}
```

The code implementing each SWI must be kept as short as possible and in particular should not call routines from the C library, as these can make many nested procedure calls which can exhaust the stack space and cause the application to crash.

This C function is called from the top-level assembly language routine, which places the number of the SWI to be handled into r0, and a pointer to the registers as they were when the SWI was encountered (ie. the Supervisor stack) in r1. It then invokes the C function with a branch with link:

```
   BL C_SWI_Handler
```

Passing arguments from the top-level routine

The APCS ensures that when `C_SWI_Handler` is called, r0 is allocated to its first argument and r1 to the second. To read the values in registers r0 to r12 from `C_SWI_Handler`, access the integer values pointed to by `reg`, for example:

```
value_in_reg_0 = reg [0];
value_in_reg_1 = reg [1];
value_in_reg_2 = reg [2];
value_in_reg_3 = reg [3];
:
:
value_in_reg_12 = reg [12];
```

How reg relates to the stack is shown in ➲*Figure 12-4: Accessing the Supervisor stack.*

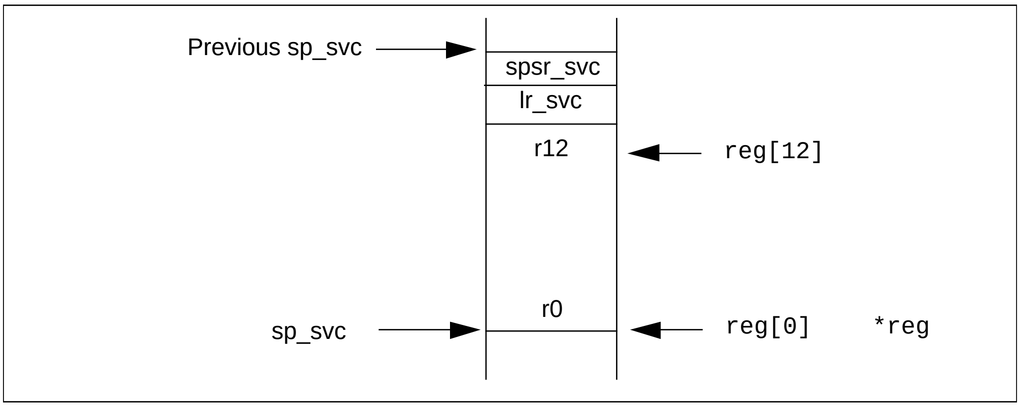


Figure 12-4: Accessing the Supervisor stack

Returning a result to the top-level routine

To return a result from the function to the calling assembly language routine, place the result value back on the stack with a simple assignment, for example:

```
reg [0] = result_of_SWI_call;
```

12.2.2 Implementing a SWI handler in assembly language

In assembly language, choosing which SWI to execute can be done with a jump table:

```
          ADR r2, SWIJumpTable
          LDR pc, [r2,r0,LSL #2]

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 go back to user program
```

12.3 Loading the Vector Table

Finally, having written your SWI handler, install an instruction in the vector table so that encountering a SWI causes it to be called. To do this, place a branch instruction in the table. The instruction can be generated using the following method:

1. Take the address of the top level SWI handler.
2. Subtract the address of the SWI vector (ie. 0x8).
3. Subtract 0x8 to allow for the pipeline.
4. Shift the result right by two to give a word offset rather than a byte offset.
5. Test that the top eight bits of this offset are clear to ensure that the offset is only 24 bits long (as the branch is limited to this).
6. Logically OR this with 0xea000000 (BAL instruction) to produce the complete instruction for placing in the vector.

In C this could be coded as:

```
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 32Mbytes    */
{    unsigned vec, oldvec;
     vec = ((routine - (unsigned)vector - 0x8)>>2);
     if (vec & 0xff000000)
     {  printf ("Installation of Handler failed");
        exit (0);
     }
     vec = 0xea000000 | vec;
     oldvec = *vector;
     *vector = vec;
     return (oldvec);
}
```

Notice that the contents of the vector are updated by the routine itself; the return value is the previous contents of the vector. The reason for returning this value will be examined shortly. For now, as no use is made of the previous contents, this could be called from your C program with:

```
Install_Handler((unsigned)SWIHandler,swivec);
```

where

```
unsigned *swivec = (unsigned *)0x8;
```

Implementing SWIs

In most circumstances, the branch instruction’s 32 Mbyte range will be sufficient to reach the SWI handler from the vector table. Sometimes, however, an alternate method is needed. This is to directly force the PC to the handler’s start address. For this to work:

- the address of the handler must be stored in a suitable memory location
- the vector must contain the encoding of an instruction to load the PC with the contents of that memory location

This can be implemented as:

```
unsigned Install_LDR_Handler (unsigned *vector, unsigned address)
/* Updates contents of 'vector' to contain 'LDR pc,[pc,#offset]'*/
/* to cause branch from vector to location contained within     */
/* 'address'. Function return value is original contents of     */
/* 'vector'.                                                    */
/* NB: 'address' must be within 4k of vector                    */
{    unsigned vec, oldvec;
     vec = ((address - (unsigned)vector - 0x8);
     if (vec & 0xfffff000)
     {  printf ("Installation of Handler failed");
        exit (0);
     }
     vec = 0xe59ff000 | vec;  /* LDR pc, [pc,#offset] */
     oldvec = *vector;
     *vector = vec;
     return (oldvec);
  }
```

This again returns the original contents of the vector. Temporarily ignoring this returned value, this routine could be called from the user’s C program by:

```
Install_LDR_Handler(swivec,(unsigned)swiaddr);
```

where

```
unsigned *swivec = (unsigned *) 0x8;
unsigned *swiaddr= (unsigned*)0x38; /*An address<=4k from vector*/
*swiaddr = (unsigned)SWIHandler;
```

12.4 Calling SWIs from your Application

This is very simple in assembly language. Set up your registers as required and then call the relevant SWI:

```
SWI 700
```

In C, things are slightly more complicated. Onto each SWI, you must map a call to a function in your code using the `__swi` compiler directive.This allows a SWI to be compiled in-line, without additional calling overhead, provided that:

- its arguments (if any) are passed in r0-r3 only
- its results (if any) are returned in r0-r3 only

The following sections demonstrate how to use the compiler's in-line SWI facility for a variety of different SWIs that conform to these rules. These SWIs are taken from the ARM Debug Monitor interface. For more information see *The ARM Software Development Toolkit Reference Manual: Chapter 17, Demon*.

In the examples below, the following options are used with armcc:

`-li`	specifies that the target is a little endian ARM
`-apcs 3/32bit`	specifies that the 32-bit variant of APCS 3 should be used

12.4.1 Calling SWIs that return no result

Consider an example `SWI_WriteC`. This SWI, which you can declare to be SWI number 0, writes a byte to the debugging channel, with the byte to be written being passed in r0.

The following C code writes a Carriage Return / Line Feed sequence to the debugging channel. You can find it in directory `examples/swi` as `newline.c`:

```
void __swi(0) SWI_WriteC(int ch);

void output_newline(void)
{ SWI_WriteC(13);
  SWI_WriteC(10);
}
```

In the declaration of `SWI_WriteC`, notice how `__swi(0)` declares the `SWI_WriteC` 'function' to be in-line SWI number 0.

Compile this to ARM assembly language source using:

```
armcc -S -li -apcs 3/32bit newline.c -o newline.s
```

This generates the following:

```
output_newline
    MOV    a1,#&d
    SWI    &0
    MOV    a1,#&a
    SWI    &0
    MOV    pc,lr
```

Note that your version of armcc may produce slightly different output to that listed here.

12.4.2 Calling SWIs that return one result

Consider SWI_ReadC, which you want to be SWI number 4. This reads a byte from the debug channel and returns it in r0.

The following C code, a naive routine for reading a line, can be found in directory examples/swi as readline.c:

```
char __swi(4) SWI_ReadC(void);

void readline(char *buffer)
{ char ch;
  do {
    *buffer++=ch=SWI_ReadC();
  } while (ch!=13);
  *buffer=0;
}
```

Note the declaration SWI_ReadC is a function which takes no arguments and returns a char, and is implemented as in-line SWI number 4.

Compile this code to produce ARM Assembler source using:

```
armcc -S -li -apcs 3/32bit readline.c -o readline.s
```

The code produced is:

```
readline
    STMDB  sp!,{lr}
    MOV    lr,a1
|L000008.J4.readline|
    SWI    &4
    STRB   a1,[lr],#1
    CMP    a1,#&d
    BNE    |L000008.J4.readline|
    MOV    a1,#0
    STRB   a1,[lr,#0]
    LDMIA  sp!,{pc}
```

Note that your version of armcc may produce slightly different output to that listed here.

12.4.3 Calling a SWI which returns 2-4 results

If a SWI returns two, three or four results, its declaration must specify that it is a struct-valued SWI, and the special keyword `__value_in_regs` must also be used. 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. See *7.3 Passing and Returning Structures* on page 7-9 for more details.

As an example, consider `SWI_InstallHandler`, which we want to be SWI number 0x70.

On entry r0 contains the exception number, r1 contains the workspace pointer and r2 contains the address of the handler.

On exit r0 is undefined, r2 contains the address of the previous handler and r1 the previous handler's workspace pointer.

The following fragment demonstrates how this SWI could be declared and used in C:

```
typedef struct SWI_InstallHandler_struct
{ unsigned exception;
  unsigned workspace;
  unsigned handler;
} SWI_InstallHandler_block;


SWI_InstallHandler_block
  __value_in_regs
    __swi(0x70) SWI_InstallHandler(unsigned r0, unsigned r1,
unsigned r2);

void InstallHandler(SWI_InstallHandler_block *regs_in,
                    SWI_InstallHandler_block *regs_out)
{ *regs_out=SWI_InstallHandler(regs_in->exception,
                               regs_in->workspace,
                               regs_in->handler);
}
```

This code is provided in directory `examples/swi` as `installh.c`, and can be compiled to produce ARM assembler source using:

```
armcc -S -li -apcs 3/32bit installh.c -o installh.s
```

The code which armcc produces is:

```
InstallHandler
    STMDB  sp!,{lr}
    MOV    lr,a2
    LDMIA  a1,{a1-a3}
    SWI    &70
    STMIA  lr,{a1-a3}
    LDMIA  sp!,{pc}
```

Note that your version of armcc may produce slightly different output to that listed here.

12.4.4 Dealing with a SWI whose number is not known until run time

If you need to call a SWI whose number is not known until run time, the mechanisms discussed above are not appropriate.

This situation might occur when there are a number of related operations that can be performed on an object, and each operation has its own SWI.

There are several ways of dealing with this. For example:

- constructing the SWI instruction from the SWI number, storing it somewhere and then executing it
- using a 'generic' SWI which takes as an extra argument a code for the actual operation to be performed on its arguments. This 'generic' SWI would then decode the operation and perform it.

A mechanism has been added to armcc to support the second method outlined here. The operation is specified by a value which is passed in r12 (`ip`). The arguments to the 'generic' SWI are passed in registers r0-r3, and values optionally returned in r0-r3 using the mechanisms described above. The operation number passed in r12 could be, but need not be, the number of the SWI to be called by the 'generic' SWI.

Here is an C fragment which uses a 'generic', or 'indirect' SWI:

```
unsigned __swi_indirect(0x80)
    SWI_ManipulateObject(unsigned operationNumber,
       unsigned object,unsigned parameter);

unsigned DoSelectedManipulation(unsigned object,
       unsigned parameter, unsigned operation)
{ return SWI_ManipulateObject(operation, object, parameter);
}
```

This code is provided in directory `examples/swi` as `swimanip.c`, and can be compiled to produce ARM Assembler source using:

```
armcc -S -li -apcs 3/32bit swimanip.c -o swimanip.s
```

This produces the following code:

```
DoSelectedManipulation
    MOV    ip,a3
    SWI    &80
    MOV    pc,lr
```

Note that the your version of armcc may produce output which is slightly different from that listed here.

12.5 Development Issues: SWI Handlers and Demon

When developing an application for ARM, the initial testing ground for the code is liable to be armsd using either the ARMulator or a PIE card.

The ARM debug monitor (Demon) reserves SWIs in the range 0–255. These implement many of the semi-hosted functions which, among other things, are needed for armsd to work correctly. You should therefore avoid defining SWIs that overlap this range.

If you are using an ARMulator, installing your own handler will not stop Demon’s SWIs still being accessible. However, if you are using a PIE card, Demon’s SWI facilities will disappear.

You can prevent this from happening by intercepting Demon’s SWI handler before installing your own. If your handler does not deal with a particular SWI, it can pass the SWI on to Demon’s. To do this, move Demon’s installation instruction out of the vector table and replace it with one pointing to your own handler, putting Demon’s instruction at an address where your own handler can call it if required. Bear in mind that Demon installs its SWI handler using the LDR method described above, since on a PIE card the SWI handler code is in ROM some 3 Gbytes above the vector table. You will therefore have to adjust the instruction’s PC-relative offset value to take account of its new location.

First you need a location in which to store the original Demon SWI vector instruction. This might be:

```
unsigned *Dswivec = (unsigned *) 0x20;
```

The `Install_Handler()` routine described in ➲*12.3 Loading the Vector Table* on page 12-9 returns the original contents of the vector being installed into, so the following call will store the original Demon SWI vector instruction at its new location as well as installing our own handler:

```
*Dswivec = Install_Handler ((unsigned)SWIHandler, swivec);
```

Next update the PC-relative offset in the original Demon instruction, to allow for the fact that it now occupies a different memory location. The following call will do this:

```
Update_Demon_Vec (swivec, Dswivec);
```

where:

```
void Update_Demon_Vec (unsigned *original, unsigned *Dvec)
/* Returns updated instruction 'LDR pc, [pc,#offset]' when   */
/* moved from 'original' to 'Dvec' (ie recalculates offset). */
/* Assumes 'Dvec' is higher in memory than 'original'.       */
{
      *Dvec = ((*Dvec &0xfff)
                     - ((unsigned) Dvec - (unsigned) original))
                     | (*Dvec & 0xfffff000);
}
```

Implementing SWIs

The C SWI handler function must be updated so that it can report whether or not it has handled this SWI:

```
unsigned C_SWI_Handler (unsigned number, unsigned *reg)
{      unsigned done = 1;
       switch (number)
       {             case 256: /* SWI number 256 code */
                            break;
                     case 257: /* SWI number 257 code */
                            break;
                     default: done = 0;
       }
       return (done);
}
```

The result passed back can be used by the top-level assembly language handler to determine whether the SWI has been handled, or whether it should be handed on to Demon's handler:

```
BL  C_SWI_Handler      ; Call C routine to handle SWI
CMP r0, #0             ; Has C routine handled SWI ?
                       ;  0 = no, 1 = yes
;
; Restore registers and cpsr from stack
;
; Now need to decide whether to return from handler or to
; call the next handler in the chain (the debugger's).
MOVNES pc,lr           ; return from handler if SWI handled
BEQ Dswivec            ; else jump to address containing
                       ; instruction to branch to address of
                       ; debugger's SWI handler.
```

Note The `BEQ Dswivec` instruction would not actually branch to the required stored vector, but would instead jump to the address where the location of that pointer is stored in the data area. It is cited here to illustrate the location to which the handler is attempting to branch. The easiest way to write it is as a branch to a known address, which in this case would be:

```
BEQ 0x20
```

However, this can be done more flexibly by importing the `Dswivec` label into the assembly language module. You can then store the address where the Demon vector is stored within the module and force the PC to that address.

This requires a short piece of code (`MakeChain`) which can be called from the main program after it has set up the new vector and stored the old vector.

```
        LDR pc, swichain    ; else jump to address containing
                            ; instruction to branch to address of
                            ; debugger's SWI handler.
         :
         :
swichain
        DCD 0

MakeChain
        LDR r0, =swichain   ; Load address of swichain into r0.
        LDR r1, =Dswivec    ; Load address of Dswivec into r1.
        LDR r2, [r1]        ; Load contents of Dswivec, i.e. the
                            ; location of the stored Demon vector.
        STR r2, [r0]        ; Store vector location within range
                            ; of PC relative load.
        MOV pc,lr           ; Return from routine.
```

Note that while developing under Demon, you will not need to set up a stack in Supervisor mode, as Demon creates a 512 byte stack for you.

Once development is finished, you will need to do two further things before producing the code for your final system:

- set up the Supervisor mode stack (as described earlier)
- remove the additions that patch your handler in front of Demon’s

12.6 Example SWI Handler

The following two program listings implement an example SWI handler that can be run on a PIE card, via armsd. To produce this program, enter the listings, then type:

```
armcc -li -c install.c
armasm -li handle.s
armlink install.o handle.o /work/arm/lib/armlib.32l -o swi
armsd -serial swi
go
```

Note that you should use the pathname of the library on your system at the link stage. In addition, Demon only installs its vectors on start up of armsd, so the updating of the Demon SWI vector will only work correctly during the first execution of the application.

The C SWI handler actually stores the values it is passed in the memory locations pointed to by `called_256`, `param_257`, `param_258` and `param_259`. After running the program you can check that the parameters were passed correctly by examining these locations.

12.6.1 install.c

```
/***************************/
/* File: install.c         */
/* Author: Andy Beeson     */
/* Date: 7th February 1994 */
/***************************/
#include <stdio.h>
#include <stdlib.h>

extern void SWIHandler (void);
extern void MakeChain (void);

unsigned *Dswivec =(unsigned *)0x20; /*ie place to store old one*/

struct four_results
{ unsigned a;
  unsigned b;
  unsigned c;
  unsigned d;
};

__swi (256) void  my_swi_256 (void);
__swi (257) void my_swi_257 (unsigned);
__swi (258) unsigned my_swi_258
           (unsigned,unsigned,unsigned,unsigned);
__swi (259) __value_in_regs struct four_results
      my_swi_259 (unsigned, unsigned, unsigned,unsigned);

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 32Mbytes from 'vector'. */
{ unsigned vec, oldvec;
  vec = ((routine - (unsigned)vector - 0x8)>>2);
  if (vec & 0xff000000)
  { printf ("Installation of Handler failed");
    exit (0);
  }
  vec = 0xea000000 | vec;
  oldvec = *vector;
  *vector = vec;
  return (oldvec);
}

void Update_Demon_Vec (unsigned *original, unsigned *Dvec)
/* Returns updated instruction 'LDR pc, [pc,#offset]' when   */
/* moved from 'original' to 'Dvec' (ie recalculates offset). */
/* Assumes 'Dvec' is higher in memory than 'original'.       */
{
  *Dvec = ((*Dvec &0xfff)
    - ((unsigned) Dvec - (unsigned) original))
    | (*Dvec & 0xfffff000);
}

unsigned C_SWI_Handler (unsigned number, unsigned *reg)
{ unsigned done = 1;

  /* Set up parameter storage block pointers */
  unsigned *called_256 = (unsigned *) 0x24;
  unsigned *param_257 = (unsigned*) 0x28;
  unsigned *param_258 = (unsigned*) 0x2c; /* + 0x30,0x34,0x38 */
  unsigned *param_259 = (unsigned*) 0x3c; /* + 0x40,0x44,0x48 */
  switch (number)
  { case 256:
      *called_256 = 256; /* Store a value to show that */
      break;             /* SWI was handled correctly. */
    case 257:
      *param_257 = reg [0]; /* Store parameter */
      break;
    case 258:
      *param_258++ = reg [0]; /* Store parameters */
      *param_258++ = reg [1];
      *param_258++ = reg [2];
      *param_258 = reg [3];
      /* Now calculate result */
      reg [0] += reg [1] + reg [2] + reg [3];
      break;
    case 259:
      *param_259++ = reg [0]; /* Store parameters */
      *param_259++ = reg [1];
      *param_259++ = reg [2];
      *param_259 = reg [3];
      reg [0] *= 2; /* Calculate results */
      reg [1] *= 3;
      reg [2] *= 4;
      reg [3] *= 5;
      break;
    default: done = 0; /* SWI not handled */
  }
  return (done);
}

int main ()
{
  struct four_results r_259; /* Results from SWI 259 */
  unsigned *swivec = (unsigned *)0x8; /* Pointer to SWI vector */
  *Dswivec = Install_Handler ((unsigned)SWIHandler, swivec);
  Update_Demon_Vec (swivec, Dswivec);
  MakeChain ();

  printf("Hello 256\n");
  my_swi_256 ();
  printf("Hello 257\n");
  my_swi_257 (257);
  printf("Hello 258\n");
  printf(" Result = %u\n",my_swi_258 (1,2,3,4));
  printf ("Hello 259\n");
  r_259 = my_swi_259 (10,20,30,40);
  printf (" Results are: %u %u %u %u\n",
         r_259.a,r_259.b,r_259.c,r_259.d);
  printf("The end\n");
  return (0);
}
```

12.6.2 handle.s

```
/***************************/
/* File: handle.s          */
/* Author: Andy Beeson     */
/* Date: 7th February 1994 */
/***************************/
  AREA TopSwiHandler, CODE ; name this block of code

  EXPORT  SWIHandler
  EXPORT  MakeChain
  IMPORT  C_SWI_Handler
  IMPORT  Dswivec
SWIHandler
  SUB r13, r13, #4          ; leave space to store spsr
  STMFD r13!,{r0-r12,r14}   ; store registers
  MOV r1, r13               ; second parameter to C routine
                            ; is register values.
  LDR r0,[r14,#-4]          ; Calculate address of SWI instruction
                            ; and load it into r0
  BIC r0,r0,#0xff000000     ; mask off top 8 bits of instruction
  MRS r2, spsr
  STR r2,[r13,#14*4]        ; store spsr on stack at original r13
  BL  C_SWI_Handler         ; Call C routine to handle SWI
  CMP r0, #0                ; Has C routine handled SWI ?
                            ;  0 = no, 1 = yes
  LDR r2, [r13,#14*4]       ; extract spsr from stack
  MSR spsr,r2               ; and restore it
  LDMFD r13!, {r0-r12,lr}   ; Restore original registers
  ADD r13,r13,#4
  ; Now need to decide whether to return from handler or to call
  ; the next handler in the chain (the debugger's).
  MOVNES pc,lr              ; return from handler if SWI handled
  LDR pc, swichain          ; else jump to address containing
                            ; instruction to branch to address of
                            ; debugger's SWI handler.

swichain
  DCD 0

MakeChain
  LDR r0, =swichain         ; Load address of swichain into r0.
  LDR r1, =Dswivec          ; Load address of Dswivec into r1.
  LDR r2, [r1]              ; Load contents of Dswivec, i.e. the
                            ; location of the stored Demon vector.
  STR r2, [r0]              ; Store vector location within range
                            ; of PC relative load.
```

```
MOV pc,lr                  ; Return from routine.

END                        ; mark end of this file
```

13 Benchmarking, Performance Analysis, and Profiling

This chapter explains how to run benchmarks on the ARM processor, and how to use the profiling facilities to help improve the size and performance of your code.

Benchmarking, Performance Analysis, and Profiling

13.1 Introduction

It is often useful to obtain the following information about a piece of application software:

- code size
- overall execution time
- time spent in specific parts of an application

Such information can allow the you to:

- compare the ARM's performance against other processors in benchmark tests
- make decisions about required clock speed and memory configuration of a projected system
- pinpoint where an application can be streamlined, leading to a reduction in the system's memory requirements
- identify performance-critical sections of code which can then be optimised using a different algorithm, or by rewriting in assembler

This chapter shows you how to measure code size and execution time, and how to generate an execution profile to discover where the time is being spent in your application.

13.2 Measuring Code and Data size

To measure the code size of an ARM image, use armlink's `-info sizes` or `-info totals` option:

`-info sizes` gives a breakdown of the code and data sizes of each object file or library member making up an image.

`-info totals` gives a summary of the total code and data sizes of all object files and all library members making up an image.

13.2.1 Example: Measuring the code and data size of Dhrystone

To show how these options work in practice, the example uses the Dhrystone benchmark program. You can find this in directory `examples/dhry`. Compile it with the command:

```
armcc -c -Ospace -DMSC_CLOCK dhry_1.c dhry_2.c
```

As armcc carries out the compilation, it will issue a number of warning messages. You can ignore these, or, if you prefer, suppress them by including the `-w` option on the command line.

Notice the use of the `-Ospace` option, which optimises for space. You would normally use the `-Otime` option with Dhrystone, as it is primarily a performance benchmark.

Now link the program:

```
armlink -o dhry -info totals dhry_1.o dhry_2.o armlib.32l
```

Note If `armlib.32l` is not in the current directory, you need to specify its full path name.

The `-info totals` option causes armlink to produce output similar to the following—the exact figures may vary since these are dependent on the version of the compiler and library being used:

```
                code   inline  inline'const'    RW   0-Init  debug
                size     data strings   data   data     data   data
Object totals   2272       28    1540      0     48    10200      0
Library totals 34408      400     764    128    700     1176      0
Grand totals   36680      428    2304    128    748    11376      0
```

The columns in the table have the following meanings:

`code size` gives the code size, excluding any data which has been placed in the code segment (see `inline data`, below).

`inline data` reports the size of the data included in the code segment by the compiler. Typically, this data will contain the addresses of variables which are accessed by the code, plus any floating point immediate values or immediate values that are too big to load directly into a register. In does *not* include inlined strings, which are listed separately (see `inline strings`, below).

`inline strings` shows the size of read-only strings placed in the code segment. The compiler will put such strings here whenever possible, since this has the effect of reducing run-time RAM requirements.

`const` lists the size of any variables explicitly declared as `const`. These variables are guaranteed to be read only and so are placed in the code segment by the compiler.

`RW data` gives the size of Read/Write data. This is data which is read/write and also has an initialising value (which may be 0 if none is explicitly given). RW data will consume the displayed amount of RAM at run time, but will also require the same amount of ROM to hold the initialising values which are copied into RAM on image startup.

`0-init data` shows the size of Read/Write data which is zero-initialised at image startup. Typically this will contain arrays which are not initialised in the C source code. `0-init data` requires the displayed amount of RAM at run-time but does not require any space in ROM, since its initialising value is 0.

`debug data` reports the size of any debugging data if the files are compiled with the `-g` option.

The ROM and RAM requirements for the Dhrystone program would be:

ROM = code size + inline data + inline strings + const data + RW data
= 36680 + 428 + 2304 + 128 + 748
= 40278

RAM = RW data + 0-Init data
= 748 + 11376
= 12124

To repeat this experiment with the Thumb compiler, issue the command:

```
tcc -c -Ospace -DMSC_CLOCK dhry_1.c dhry_2.c
```

This time use armlink’s `-info sizes` option to give a complete breakdown of the code and data sizes:

```
armlink -o dhry -info sizes dhry_1.o dhry_2.o armlib.16l
```

13.3 Timing Program Execution Using the ARMulator

The ARMulator provides facilities for real time simulation. As it executes your program it counts the total number of clock ticks taken, and is then able to report this figure either directly through the debugger's `$clock` variable, or indirectly through a C library function such as `clock()`.

This section explains how to time a program on a simulated system. You can find a detailed description of the available real time simulation facilities in ➲*The ARM Software Development Toolkit Reference Manual: Chapter 14, ARMulator* and ➲*The ARM Software Development Toolkit Reference Manual: Chapter 7, Symbolic Debugger*.

13.3.1 Example 1: sorts.c

Before you can use ARMulator's real time simulation facilities of the ARMulator, you must tell armsd or the Windows Debugger:

- the processor speed
- the type and speed of memory attached to the processor

This information is conveyed via a file called `armsd.map`, which must be in the current directory when the debugger is run. You can find the following example map file in directory `examples/sorts`:

```
0 80000000 RAM 4 rw 135/80 135/80
```

This describes a single contiguous section of memory from 0 up to 0x80000000. The memory system is 32 bits wide, and has an N cycle access time of 135nS and an S cycle access time of 80nS. The cycle times for reads and writes are the same.

The following steps investigate how changing the `armsd.map` file parameters alters the processor's performance.

Compile the `sorts.c` example program in directory `examples/sorts`, as follows:

```
armcc -Otime -o sorts sorts.c
```

This program sorts 1000 strings using three different algorithms—insertion, shell and quick sort—and reports the time taken by each.

Run the program under armsd using the command:

```
armsd -clock 33MHz sorts
```

where `-clock 33MHz` specifies the processor speed. When armsd starts up, it will report the following:

```
Memory map ...
00000000..80000000, 32-Bit, rw,  R(N/S) = 135/80, W(N/S) = 135/80
Clock speed = 33.33Mhz
```

If this information does not appear, armsd has failed to read the map file—check that it is in the current directory.

Next, run the sorts example by entering:

```
       go
```

at the `armsd:` prompt.

The program may take a couple of minutes to run, depending on the speed of your machine. If it takes an inordinately long time, try changing the line:

```
#define N 1000
```

in `sorts.c` to define `N` as 500 or 100.

When the program has finished, it should print the following (where one clock tick is one centisecond):

```
Insertion sort took 162 clock ticks
Shell sort took 11 clock ticks
Quick sort took 12 clock ticks
```

Now find out how long the program took to execute in total using the `$clock` variable:

```
       print $clock
```

This should report a value of about 2072176 (the exact value may be different, depending on the versions of the ARM compiler and C libraries you are using).

The `$clock` variable reports the time in microseconds, so the obtained value is equivalent to about 207 centiseconds. Adding the time taken by the three sort algorithms we get 185 centiseconds. The remaining 22 centiseconds is spent initialising the array of strings before each sort and checking that the strings have been sorted correctly after calling each sort routine.

Try re-running the program on a 50MHz ARM to see what difference this makes to the performance. Use the command:

```
       armsd -clock 50MHz sorts
```

This time you should obtain the following result:

```
Insertion sort took 145 clock ticks
Shell sort took 10 clock ticks
Quick sort took 10 clock ticks
```

Modify `armsd.map` to simulate some fast on-chip memory: change it to specify an N and S cycle access time of 20nS:

```
0 80000000 RAM 4 rw 20/20 20/20
```

Re-run the sorts example using the command:

```
    armsd -clock 50MHz sorts
```

armsd should now report:

```
Insertion sort took 33 clock ticks
Shell sort took 3 clock ticks
Quick sort took 2 clock ticks
```

13.3.2 Example 2: Dhrystone

One of the main uses for real time simulation is running performance benchmarks such as Dhrystone.

Compile the Dhrystone program in directory `examples/dhry` with both ARM and Thumb compilers.

To compile the program for ARM, use:

```
armcc -o dhry_32 -Otime -DMSC_CLOCK dhry_1.c dhry_2.c
```

To compile it for Thumb, use:

```
tcc -o dhry_16 -Otime -DMSC_CLOCK dhry_1.c dhry_2.c
```

Create the following `armsd.map` file:

```
00000000 80000000 RAM 4 rw 135/85 135/85
```

Load the benchmark into armsd with:

```
armsd -clock 20MHz dhry_32
```

and enter

```
go
```

at the `armsd:` prompt to start execution.

When prompted for the number of Dhrystones enter:

```
35000
```

The program will report the number of Dhrystones per second. Record the value and repeat the simulation with the Thumb version of Dhrystone (`dhry_16`).

You may obtain slightly different figures depending on the version of compiler and library you are using. Try varying the clock speed, the memory access speeds and the data bus width to see the effect of these on performance.

When measuring Thumb on a 32-bit memory system, try placing a `*` after the memory access `rw` (ie. enter `rw*`) to see the performance gain from putting a 16-bit latch on such a system. Here are some example results. Your results may vary depending on compiler version, compiler options and the library version.

Note *These results are for Dhrystone version 2.1*

Clock speed = 20MHz, Memory access times (N = 135nS, S = 85nS)		
	ARM	Thumb
32-bit memory	14204.5	11876.5
32-bit memory	14204.5	13636.4 (with 16 bit latch)
16-bit memory	7894.7	10067.1
8-bit memory	4731.9	5703.4

Clock speed = 33MHz, Memory access times (N = 115nS, S = 85nS)		
	ARM	Thumb
32-bit memory	16759.8	14018.7
32-bit memory	16759.8	17142.9 (with 16 bit latch)
16-bit memory	9063.4	11718.7
8-bit memory	4724.4	6237.0

Clock speed = 33MHz, Memory access times (N = 30nS, S = 30nS)		
	ARM	Thumb
32-bit memory	52083.3	43478.3
32-bit memory	52083.3	43478.3 (with 16 bit latch)
16-bit memory	27624.3	35971.2
8-bit memory	14285.7	18939.4

13.4 Profiling Programs using the ARMulator

If you find that your application is running unacceptably slowly, you may wish to use the ARMulator's profiling facilities to determine exactly where your program is spending its time.

For a full description of the toolkit's profiling facilities see *The ARM Software Development Toolkit Reference Manual: Chapter 8, ARM Profiler*. This section shows you how to profile an application using armsd, and explains the format of the profile report.

13.4.1 An example profile

Compile the `sorts.c` example program in directory `examples/sorts` as follows. Note that no special options are needed to compile a program for profiling:

```
armcc -Otime -o sorts sorts.c
```

Now start up armsd:

```
armsd
```

Load the `sorts` program into armsd with the `/callgraph` option. `/callgraph` tells armsd to prepare an image for profiling by adding code that counts the number of function calls. Enter

```
load/callgraph sorts
```

at the `armsd:` prompt, and turn profiling on:

```
ProfOn
```

Next, run the program as normal:

```
go
```

and write the profile data to a file using the `ProfWrite` command:

```
ProfWrite sorts.prf
```

Exit armsd with the `Quit` command, and display a profile from the collected profile data by entering:

```
armprof -Parent sorts.prf > profile
```

at the system prompt, where `-Parent` instructs the profile displayer to include information about the callers of each function.

The debugger will generate the profile report and output it to the `profile` file, an extract from which is shown at the end of this section

13.4.2 Layout of the profile report

The report is divided into sections, each of which gives information about a function in the program. A section's function (called the *current function*) is indicated by having its name start at the left-hand edge of the `Name` column. For example, the current function in the first section is `main`. Functions listed below the current function are child functions, ie. functions called by the current function. Those listed above the current function are parents, ie. functions that call it.

The columns in the report have the following meanings:

`Name` Displays the function names. The current function in a section starts at the column's left-hand edge: parent and child functions are shown indented.

`cum%` Shows the total percentage time spent in the current function plus the time spent in any functions which it called. In the case of `main`, the program spent 96.04% of its time in `main` and its children.

`self%` Shows the percentage time spent in the current function on each parent function's behalf.

`desc%` Shows the percentage time spent in children of the current function on the current function's behalf. For example, in the case of `main` only 0.16% of the time is spent in `main` itself, whereas 95.88% of the time is spent in functions called by `main`.

`calls` Reports the number of times a function is called from the current function. The call count for `main` is 0 because `main` is the top-level function, and is not called by any other functions.

The section for `insert_sort` shows that it made 243432 calls to `strcmp,` and that this accounted for 59.44% of the time spent in `strcmp` (the `desc%` column shows 0 in this case because `strcmp` does not call any functions).

In the case of `strcmp`, `qs_string_compare` (which is called by `qsort`), `shell_sort` and `insert_sort` made respectively 13021, 14059 and 243432 calls to `strcmp` and the time spent in `strcmp` is shared out between the functions in the ratio 3.17% to 3.43% to 59.44%.

```
Name                            cum%     self%    desc%    calls
main                           96.04%    0.16%   95.88%        0
  qsort                                  0.44%    0.75%        1
  _printf                                0.00%    0.00%        3
  clock                                  0.00%    0.00%        6
  _sprintf                               0.34%    3.56%     1000
  check_order                            0.29%    5.28%        3
  randomise                              0.12%    0.69%        1
  shell_sort                             1.59%    3.43%        1
  insert_sort                           19.91%   59.44%        1
----------------------------------------------------------------
  main                                  19.91%   59.44%        1
insert_sort                    79.35%   19.91%   59.44%        1
  strcmp                                59.44%    0.00%   243432
----------------------------------------------------------------
  qs_string_compare                      3.17%    0.00%    13021
  shell_sort                             3.43%    0.00%    14059
  insert_sort                           59.44%    0.00%   243432
strcmp                         66.05%   66.05%    0.00%   270512
----------------------------------------------------------------
  main                                   0.29%    5.28%        3
check_order                     5.57%    0.29%    5.28%        3
  atoi                                   0.49%    4.78%     3000
----------------------------------------------------------------
  check_order                            0.49%    4.78%     3000
atoi                            5.28%    0.49%    4.78%     3000
  strtol                                 0.92%    3.86%     3000
----------------------------------------------------------------
  main                                   1.59%    3.43%        1
shell_sort                      5.02%    1.59%    3.43%        1
  __rt_sdiv                              0.00%    0.00%        6
  strcmp                                 3.43%    0.00%    14059
----------------------------------------------------------------
  atoi                                   0.92%    3.86%     3000
strtol                          4.78%    0.92%    3.86%     3000
  _strtoul                               2.44%    1.41%     3000
----------------------------------------------------------------
  main                                   0.34%    3.56%     1000
_sprintf                        3.91%    0.34%    3.56%     1000
  putc                                   0.09%    0.00%     1000
  __vfprintf                             0.95%    2.27%     1000
  memset                                 0.25%    0.00%     1000
----------------------------------------------------------------
```

```
  strtol                                 2.44%    1.41%     3000
_strtoul                        3.86%    2.44%    1.41%     3000
  _chval                                 1.41%    0.00%    18000
----------------------------------------------------------------
qs_string_compare               3.69%    0.51%    3.17%        0
  strcmp                                 3.17%    0.00%    13021
----------------------------------------------------------------
  _printf                                0.00%    0.00%        3
  _sprintf                               0.95%    2.27%     1000
__vfprintf                      3.24%    0.95%    2.28%     1003
  ferror                                 0.03%    0.00%     1003
  putc                                   0.00%    0.00%       91
  printf_display                         1.49%    0.75%     1003
----------------------------------------------------------------
  __vfprintf                             1.49%    0.75%     1003
printf_display                  2.24%    1.49%    0.75%     1003
  strlen                                 0.14%    0.00%     1003
  _kernel_udiv10                         0.00%    0.00%     2896
  putc                                   0.57%    0.03%     6007
----------------------------------------------------------------
  _strtoul                               1.41%    0.00%    18000
_chval                          1.41%    1.41%    0.00%    18000
----------------------------------------------------------------
  main                                   0.44%    0.75%        1
qsort                           1.19%    0.44%    0.75%        1
  partition_sort                         0.72%    0.03%        1
----------------------------------------------------------------
  main                                   0.12%    0.69%        1
randomise                       0.81%    0.12%    0.69%        1
  rand                                   0.21%    0.00%     1000
  __rt_sdiv                              0.48%    0.00%     1000
----------------------------------------------------------------
  qsort                                  0.72%    0.03%        1
partition_sort                  0.75%    0.72%    0.03%        1
  __rt_udiv                              0.03%    0.00%      167
----------------------------------------------------------------
```

Remainder of file omitted for brevity

14 Using the Thumb Instruction Set

This chapter offers advice on exploiting the features of the Thumb instruction set.

14.1 Working with Thumb Assembly Language

To assemble Thumb code you must use tasm, which is able to assemble ARM assembly language into 32-bit ARM instructions, and Thumb assembly language into 16-bit Thumb instructions. By default tasm assembles Thumb code, but it can be forced to produce ARM code with the `-32` command-line switch or the `CODE32` assembler directive. (Another directive, `CODE16,` enables you to switch between ARM and Thumb code in the same source file).

For a comprehensive description of the Thumb instruction set, see *The ARM Software Development Toolkit Reference Manual: Chapter 5, Thumb Instruction Set*.

For details of how to use armasm and tasm, please refer to *The ARM Software Development Toolkit Reference Manual: Chapter 3, Assembler*.

14.1.1 A simple example program

The following example shows how to write a simple program in Thumb which prints out some values in hex.

Create the following file and call it `hex.s`, or use the copy of this file in directory `examples/thumb`.

```
--- hex.s -----------------------------------------------------
; All assembler files must have at least one AREA directive at
; the start. The AREA directive passes information to the linker
; telling it where to place the code in memory (for example,
; READONLY areas would be placed in a ROM segment).
;
      AREA   hex, CODE, READONLY
; Some definitions for system calls or SWIs supported by
; 'armos'
;
OS_WriteC     EQU      0x00; Write a character
OS_Exit       EQU      0x11; Exit program
;
; ENTRY tells the linker where the entry point of an image is.
; The linker in turn encodes this information in the image
; header so the armulator knows where the application image
; starts.
;
      ENTRY
;
; All applications are entered initially in ARM state so we
; must write a couple of lines of ARM assembler to switch to
; Thumb state. We tell the assembler we are going to do this
; with the 'CODE32' directive.
```

```
;
      CODE32
;
; Use the ARM BX instruction to switch into Thumb state. The
; BX instruction will branch to the Thumb entry point. Add
; one to the entry point to force the switch into Thumb state.
;
      ADR    R0, Thumb_Entry+1 BXR0
; The next section of code is Thumb code so tell the assembler
; to assemble it as Thumb.
;
      CODE16
Thumb_Entry
; Now load several values in turn into R0 and call the Hex
; Print routine.
;
      MOV    R0, #0xED
      BL     Hex_Print
; The number 0xAA55AA55 is too big to load into a register
; using an immediate constant so we tell the assembler to
; load it from memory by placing an '=' symbol before the
; number. We do not need to tell the assembler where to
; store the number in memory, it will decide that for us.
;
      LDR    R0, =0xAA55AA55
      BL     Hex_Print
; To load a negative number it is often easier to load the
; positive value and then negate it as negative numbers cannot
; be loaded using immediate constants.
;
      MOV    R0, #10
      NEG    R0, R0
      BL     Hex_Print
; Now exit the program cleanly
;
      SWI    OS_Exit
;
Hex_Print
; Save registers used by this routine. Also save the link
; register which contains the return address as this may be
; destroyed by the call to SWI OS_WriteC if the code is
; executing in supervisor mode.
      PUSH   {R0, R1, R2, LR}
      MOV    R1, R0       ; Enter with value to print in R0
                          ; Mov it to R1 as R0 needed for call
```

```
                             ; to OS_WriteC.
       MOV    R2, #8         ; Loop counter for 8 digits
 Loop1 LSR    R0, R1, #28; Extract digits from R1 starting with
                             ; the most significant digit.
       LSL    R1, #4         ; Shift up the digit to be used next
                             ; time round the loop.
       CMP    R0, #10        ; Convert R0 to a hex digit
       BLT    Loop2
       ADD    R0, #'A'-'0'-10 ; Value >10 => add in extra to convert
                             ; to range 'A' to 'F'.
Loop2  ADD    R0, #'0'  ; Add in ASCII value for '0'.
       SWI    OS_WriteC ; Write the hex digit
       SUB    R2, #1         ; Loop for each digit
       BNE    Loop1          ; (SUB implicitly sets the condition codes)
       MOV    R0, #0x0D SWI OS_WriteC    ; Write a CR
       MOV    R0, #0x0A SWI OS_WriteC    ; Followed by a newline
       POP    {R0, R1, R2, PC}; Recover registers and return
; An END directive is always needed.
;
       END
```

Assemble this file using the command:

```
tasm hex.s
```

This will produce an object file called `hex.o`. Link this file using `armlink` as follows:

```
armlink -o hex hex.o
```

Now try running this using armsd or the Windows debugger. To run the program using armsd enter:

```
armsd -pr arm7tdm hex
```

and then enter:

```
go
```

at the `armsd:` prompt. The `-pr arm7tdm` option tells armsd to emulate an ARM7TDM processor: you must specify this option whenever you run Thumb code—see ➲*The ARM Software Development Toolkit Reference Manual: Chapter 7, Symbolic Debugger* for a full list of the command line options which you can use with armsd.

14.2 Hand-optimising the Thumb Compiler’s Output

The Thumb compiler, tcc, produces reasonably optimal code—so much so that re-coding a routine in assembler to improve performance is very rarely necessary. However, on some occasions it may be possible to make small improvements to the compiler’s output. Here we show what to look for when examining tcc’s assembler output, and how to go about making changes to it.

14.2.1 Optimising multiple loads and stores

First examine how tcc’s use of the multiple load and store instructions can be made more efficient, taking as an example a block copy routine.

This copies a block of memory, with R1 pointing to the start of the block, R0 pointing to the target area, and R2 holding the number of bytes to copy. On entry, R0 and R1 are assumed to be word-aligned, and R2 is assumed to be a multiple of 4.

In C, the routine looks like this:

```
void block_copy(void *dest, void *source, int n)
{
    int *dest_ip, *source_ip;

    dest_ip = (int *)dest;/* Move function arguments into integer  */
    source_ip = (int *)source;  /* integer for use in following loop*/
   while (n > 0) {
       *dest_ip++ = *source_ip++;
       n -= 4;
     }
}
```

You can find this in file `bcopy.c` in directory `examples/thumb`. To translate it into Thumb assembler, use the command:

```
       tcc -Otime -S bcopy.c
```

This produces the following in file `bcopy.s`:

```
; generated by Norcroft  Thumb C vsn 1.00 (Advanced RISC Machines)
; [Mar 31 1995]
|block_copy|
  CMP      __r2,#0
  BLE      F1L13
 F1L4
  LDMIA    __r1!,{__r3}
  STMIA    __r0!,{__r3}
  SUB      __r2,#4
  BGT      F1L4
 F1L13
  MOV      __pc,__lr
```

This is fairly optimal, but notice that the LDMIA/STMIA pair are being used to transfer only one word at a time. You can improve on this by making them handle four words at once, as follows:

```
block_copy
      PUSH   {R4, R5}     ; C expects these to be saved
      SUB    R2, #16      ; Fewer than 16 bytes to start with?
      BCC    %FT1
 0
      LDMIA  R1!, {R2, R3, R4, R5}; Transfer 16 bytes
      STMIA  R0!, {R2, R3, R4, R5}
      SUB    R2, #16      ; Decrement count
      BCS    %BT0
      ; R2 is now 16 less than number of bytes to go
 1    ADD    R2, #12      ; See if at least 4 bytes to go (-16+12)
      BCC    %FT3
 2    LDMIA  R1!, {R2}    ; Transfer 4 bytes
      STMIA  R0!, {R2}
      SUB    R2, #4
      BCS    %BT2
 3    POP    {R4, R5}     ; Recover C's variable registers
      MOV    PC, LR
```

14.2.2 Testing the carry flag

There is no way of expressing operations on the processor's Carry flag in C. This means that a C programmer is sometimes forced to use algorithms which, when translated into machine code, are clumsy when compared with those employed by the assembly language programmer to solve the same problem.

A good example can be found in the file bits.c in directory examples/thumb. This contains three bit manipulation routines, the last of which reverses the order of bits in a word:

```
unsigned reverse_bits(unsigned n)
{
    unsigned r, old_r;
    r = 1;
    do {
       old_r = r;
       r <<= 1;
       if (n & 1) r++;
       n >>= 1;
    } while (r > old_r);
    return r;
}
```

This routine, like the others in the file, make use of the (n & (-n)) paradigm, which evaluates to the first set bit in a word. For example, if n = 6 then (n & (-n)) evaluates to 2, which is the value of the first set bit in n.

We can obtain an assembly language listing from the file by issuing the command:

```
tcc -Otime -S bits.c
```

The code which has been generated for `reverse_bits()` is as follows:

```
  |reverse_bits|
  MOV      __r1,#1
 F3L4
  MOV      __r2,__r1
  LSL      __r1,#1
  LSR      __r3,__r0,#1
  BCC      F3L13
  ADD      __r1,#1
 F3L13
  LSR      __r0,#1
  CMP      __r1,__r2
  BHI      F3L4
  MOV      __r0,__r1
  MOV      __pc,__lr
```

By taking advantage of the ability to test the Carry flag directly, you can recode this in Thumb assembly language much more efficiently, reducing the main loop from eight instructions to just three:

```
reverse_bits
       MOV    R1, #1
0      LSR    R0, R0, #1
       ADC    R1, R1
       BCC    %BT0
       MOV    R0, R1
       MOV    PC, LR
```

14.3 ARM/Thumb Interworking

You may sometimes wish to compile parts of your program as ARM code to benefit from ARM code’s better performance in 32-bit systems, while leaving the bulk of your program compiled as Thumb code to take advantage of its better code density.

See the section on ARM/Thumb Interworking in *The ARM Software Development Toolkit Reference Manual: Chapter 2, C Compiler* for a detailed explanation of how to compile programs that mix Thumb code and ARM code.

14.3.1 A simple example

The following example shows a sort routine which you may wish to compile into ARM code, and a main program containing the rest of the application which we wish to compile into Thumb code.

The source for this example can be found in directory `examples/thumb/interwork`. The files are called `sort.c` and `main.c`.

```
--- sort.c--------------------------------------------------------

void sort(char *strings[], int n)
{   int h, i, j;
    char *v;
    strings--;        /* Make array 1 origin */
     h = 1;
     do {h = h * 3 + 1;} while (h <= n);
     do {
       h = h / 3;
         for (i = h + 1; i <= n; i++) {
           v = strings[i];
             j = i;
             while (j > h && strcmp(strings[j-h], v) > 0) {
             strings[j] = strings[j-h];
                 j = j-h;
             }
             strings[j] = v;
         }
     } while (h > 1);
 }
--- main.c --------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define N 1000
extern void sort(char **strings, int n);
static void randomise(char *strings[], int n)
```

```
{
     int i;
     int v;
     char *t;
    for (i = 0; i < N; i++) {
       v = rand() % N;
       t = strings[v];
       strings[v] = strings[i];
       strings[i] = t;
     }
}
int main(void)
{
    char *strings[N];
    char buffer[1000*(3+1)];
    char *p;
    int i;
    p = buffer;
    for (i = 0; i < 1000; i++) {
       sprintf(p, "%03d", i);
       strings[i] = p;
       p += 3+1;
     }
     randomise(strings, N);
     sort(strings, N);
}
```

Compile `sort.c` using the command:

```
armcc -arm7tm -apcs 3/noswst/nofp/interwork -Otime -c sort.c
```

and compile `main.c` with the command:

```
tcc -Ospace -c main.c
```

You do not need to use the `-apcs 3/interwork` flag when compiling the Thumb section of this example, as it is never called from ARM code. You must, however, specify the flag when compiling `sort.c`, since this function is called from Thumb code in `main.c`.

Next link the ARM and Thumb sections of code with a C library. Generally, you will want to link the interworked code with the Thumb C library, since this reduces the overall size of the program. In this particular case, however, the sort routine, which has been compiled as ARM code, uses the library function `strcmp()`. The interworking code required to call a Thumb library routine from ARM code will introduce an overhead in the routine’s main loop.

To avoid this overhead, you can force the linker to use the ARM C library version of `strcmp()` by specifying the name of module which contains it after the library name on the command line, as follows:

```
armlink -o sort main.o sort.o armlib.32l(strcmp.o) armlib.16l
```

14.3.2 Calling ARM instructions from Thumb code

There may be times when you want to make use of features in a Thumb program that are unavailable from its instruction set—the SWP instruction, or the long multiply facilities, for example. This section shows you how to access the ARM 64-bit multiply instructions from Thumb code.

There are four example functions, each of which corresponds to one of the long multiply instructions. The types Int64 and Unsigned64 are defined to hold 64-bit ints and unsigned ints respectively.

You can find the examples in directory examples/thumb/multiply.

```
--- mul.h -----------------------------------------------------------
/* This header file contains various definitions for the 64 bit mul
   instructions. It must be included in any C source wishing to use
   these functions. */

typedef struct Int64 { int lo, hi } Int64;
typedef struct Unsigned64 { unsigned lo, hi } Unsigned64;

/* Return 64 bit signed result of 'a' * 'b' */
extern __value_in_regs Int64 smull(int a, int b);
/* Return 64 bit signed result of 'a' * 'b' + 'acc' */
extern __value_in_regs Int64 smlal(Int64 acc, int a, int b);
/* Return 64 bit unsigned result of 'a' * 'b' */
extern __value_in_regs Unsigned64 umull(unsigned a, unsigned b);
/* Return 64 bit unsigned result of 'a' * 'b' + 'acc' */
extern __value_in_regs Unsigned64 umlal(Unsigned64 acc, unsigned a,
unsigned b);
--- mul.s ----------------------------------------------------------
       AREA   mul, CODE, READONLY, INTERWORK
       EXPORT smull
       EXPORT smlal
       EXPORT umull
       EXPORT umlal
smull  MOV    R2, R0       ; Ensure RdLo != RdHi != Rm
       SMULL  R0, R1, R2, R1
       BX     LR           ; Return using BX for interworking
smlal  SMLAL  R0, R1, R2, R3
       BX     LR
umull  MOV    R2, R0       ; Ensure RdLo != RdHi != Rm
       UMULL  R0, R1, R2, R1
       BX     LR
umlal  UMLAL  R0, R1, R2, R3
       BX     LR
       END
```

```
--- main.c-----------------------------------------------------
#include <stdio.h>
#include "mul.h"
int main(void)
{
   Unsigned64 u;
   Int64 s;
   u = umull(0xA0000000, 0x10101010);
   printf("umull (0xA0000000*0x10101010) = 0x%08x%08x\n", u.hi, u.lo);
   u = umlal(u, 0x00500000, 0x10101010);
    printf("umlal (+0x00500000*0x10101010) = 0x%08x%08x\n", u.hi,
u.lo);
   s = smull(0xA0000000, 0x10101010);
   printf("smull (0xA0000000*0x10101010) = 0x%08x%08x\n", s.hi, s.lo);
   s = smlal(s, 0x00500000, 0x10101010);
    printf("smlal (+0x00500000*0x10101010) = 0x%08x%08x\n", s.hi,
s.lo);
}
```

The presence of `INTERWORK` in the `AREA` declaration of the assembler section tells the linker that it is safe for functions in this code segment to be called from Thumb.

These functions are safe to call from Thumb state because they return with the `BX LR` instruction instead of the more usual `MOV PC, LR`. When a function is called from Thumb code, bit 0 of the link register will be set. `BX LR` will therefore return correctly to Thumb state, whereas `MOV PC,LR`, while returning to the correct address, would remain in ARM state.

These routines may also be called from ARM code. In this case, bit 0 of the link register will be clear on entry and the `BX LR` instruction will therefore correctly return to ARM state.

Compile the example as follows:

```
armasm -cpu arm7m mul.s n
tcc -c main.c
armlink -o mul main.o mul.o armlib.16l
```

Run the program using armsd or the Windows debugger. You should obtain the following output:

```
umull (0xA0000000*0x10101010) = 0x0a0a0a0a00000000
umlal (+0x00500000*0x10101010) = 0x0a0f0f0f05000000
smull (0xA0000000*0x10101010) = 0xf9f9f9fa00000000
smlal (+0x00500000*0x10101010) = 0xf9fefeff05000000
```

Recompile `main.c` with armcc and re-run it to check that you get the same results:

```
armcc -c main.c armlink -o mul main.o mul.o armlib.32l
```

14.4 Division by a Constant in Thumb Code

➲*5.4 Division by a Constant* on page 5-12 describes a method for performing fast division using constants, and this can also be coded in Thumb assembler. Directory `examples/thumb` contains a version of the program `divc.c` which will generate code to divide by a constant of the form (2^n – 2^m) or (2^n + 2^m).

Compile the `divc` example using your system's host compiler, using a command of the form:

```
cc -o divc divc.c
```

where *cc* is the appropriate compiler command.

Run the resulting the program, giving as the argument the constant you wish to divide by—for example, to generate code to divide by 3, enter

```
divc 3
```

This will produce the following assembly output.

```
; generated by Thumb divc 1.00 (Advanced RISC Machines) [4 Apr 95]
        CODE16
        AREA |div3$code|, CODE, READONLY
        EXPORT udiv3
udiv3
; takes argument in a1
; returns quotient in a1, remainder in a2
; cycles could be saved if only divide or remainder is required
        MOV     a2, a1
        LSR     a1, #1
        LSR     a3, a1, #2
        ADD     a1, a3
        LSR     a3, a1, #4
        ADD     a1, a3
        LSR     a3, a1, #8
        ADD     a1, a3
        LSR     a3, a1, #16
        ADD     a1, a3
        LSR     a1, #1
        ASL     a3, a1, #2
        SUB     a3, a3, a1
        ASL     a3, #0
        SUB     a2, a3
        CMP     a2, #3
        BLT     %FT0
        ADD     a1, #1
        SUB     a2, #3
0
```

```
        CMP     a2, #3
        BLT     %FT0
        ADD     a1, #1
        SUB     a2, #3
0
        MOV     pc, lr
        END
```

Index

Index

Index