FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

The Definitive Guide to the ARM® Cortex-M3

Second Edition

Joseph Yiu

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Newnes is an imprint of Elsevier

CHAPTER

Instruction Sets

4

IN THIS CHAPTER

This chapter provides some insight into the instruction set in the Cortex™-M3 and examples for a number of instructions. You'll also find more information on the instruction set in Appendix A of this book. For complete details of each instruction, refer to the *ARM v7-M Architecture Application Level Reference Manual* [Ref. 2] or user guides from microcontroller vendors.

4.1 ASSEMBLY BASICS

Here, we introduce some basic syntax of ARM assembly to make it easier to understand the rest of the code examples in this book. Most of the assembly code examples in this book are based on the ARM assembler tools, with the exception of those in Chapter 19, which focus on the Gnu's Not Unix tool chain.

4.1.1 Assembler Language: Basic Syntax

In assembler code, the following instruction formatting is commonly used:

```
label
      opcode operand1, operand2, ...; Comments
```

The *label* is optional. Some of the instructions might have a label in front of them so that the address of the instructions can be determined using the label. Then, you will find the opcode (the instruction) followed by a number of operands. Normally, the first operand is the destination of the operation. The number of operands in an instruction depends on the type of instruction, and the syntax format of the

DOI: 10.1016/B978-0-12-382091-4.00010-4

operand can also be different. For example, immediate data are usually in the form *#number*, as shown here:

```
MOV R0, #0x12 ; Set R0 = 0x12 (hexadecimal)
MOV R1, #'A'  ; Set R1 = ASCII character A
```

The text after each semicolon (;) is a comment. These comments do not affect the program operation, but they can make programs easier for humans to understand.

You can define constants using EQU, and then use them inside your program code. For example,

```
NVIC_IRQ_SETEN0 EQU 0xE000E100
NVIC_IRQ0_ENABLE EQU 0x1
     ...
     LDR R0,=NVIC_IRQ_SETEN0; ; LDR here is a pseudo-instruction that
                              ; convert to a PC relative load by
                              ; assembler.
     MOV R1,#NVIC_IRQ0_ENABLE ; Move immediate data to register
     STR R1,[R0]              ; Enable IRQ 0 by writing R1 to address
                              ; in R0
```

A number of data definition directives are available for insertion of constants inside assembly code. For example, DCI (Define Constant Instruction) can be used to code an instruction if your assembler cannot generate the exact instruction that you want and if you know the binary code for the instruction.

```
DCI 0xBE00 ;  Breakpoint (BKPT 0),  a 16-bit instruction
```

We can use DCB (Define Constant Byte) for byte size constant values, such as characters, and Define Constant Data (DCD) for word size constant values to define binary data in your code.

```
     LDR R3,=MY_NUMBER  ; Get the memory address value of MY_NUMBER
     LDR R4,[R3]        ; Get the value code 0x12345678 in R4
     ...
     LDR R0,=HELLO_TXT  ; Get the starting memory address of
                        ; HELLO_TXT
     BL PrintText       ; Call a function called PrintText to
                        ; display string
     ...
MY_NUMBER
     DCD 0x12345678
HELLO_TXT
     DCB "Hello\n",0    ; null terminated string
```

Note that the assembler syntax depends on which assembler tool you are using. Here, the ARM assembler tools syntax is introduced. For syntax of other assemblers, it is best to start from the code examples provided with the tools.

4.1.2 Assembler Language: Use of Suffixes

In assembler for ARM processors, instructions can be followed by suffixes, as shown in Table 4.1.

For the Cortex-M3, the conditional execution suffixes are usually used for branch instructions. However, other instructions can also be used with the conditional execution suffixes if they are inside an IF-THEN instruction block. (This concept is introduced in a later part of this chapter.) In those

Table 4.1 Suffixes in Instructions

Suffix	Description
S	Update Application Program Status register (APSR) (flags); for example: `ADDS R0, R1 ; this will update APSR`
EQ, NE, LT, GT, and so on	Conditional execution; EQ = Equal, NE = Not Equal, LT = Less Than, GT = Greater Than, and so forth. For example: `BEQ <Label> ; Branch if equal`

cases, the *S* suffix and the conditional execution suffixes can be used at the same time. Fifteen condition choices are available, as described later in this chapter.

4.1.3 Assembler Language: Unified Assembler Language

To support and get the best out of the Thumb®-2 instruction set, the Unified Assembler Language (UAL) was developed to allow selection of 16-bit and 32-bit instructions and to make it easier to port applications between ARM code and Thumb code by using the same syntax for both. (With UAL, the syntax of Thumb instructions is now the same as for ARM instructions.)

```
ADD R0, R1     ; R0 = R0 + R1, using Traditional Thumb syntax
ADD R0, R0, R1 ; Equivalent instruction using UAL syntax
```

The traditional Thumb syntax can still be used. The choice between whether the instructions are interpreted as traditional Thumb code or the new UAL syntax is normally defined by the directive in the assembly file. For example, with ARM assembler tool, a program code header with "CODE16" directive implies the code is in the traditional Thumb syntax, and "THUMB" directive implies the code is in the new UAL syntax.

One thing you need to be careful with reusing traditional Thumb is that some instructions change the flags in APSR, even if the *S* suffix is not used. However, when the UAL syntax is used, whether the instruction changes the flag depends on the *S* suffix. For example,

```
AND  R0, R1     ; Traditional Thumb syntax
ANDS R0, R0, R1 ; Equivalent UAL syntax (S suffix is added)
```

With the new instructions in Thumb-2 technology, some of the operations can be handled by either a Thumb instruction or a Thumb-2 instruction. For example, R0 = R0 + 1 can be implemented as a 16-bit Thumb instruction or a 32-bit Thumb-2 instruction. With UAL, you can specify which instruction you want by adding suffixes:

```
ADDS   R0, #1 ; Use 16-bit Thumb instruction by default
              ; for smaller size
ADDS.N R0, #1 ; Use 16-bit Thumb instruction (N=Narrow)
ADDS.W R0, #1 ; Use 32-bit Thumb-2 instruction (W=wide)
```

The .W (wide) suffix specifies a 32-bit instruction. If no suffix is given, the assembler tool can choose either instruction but usually defaults to 16-bit Thumb code to get a smaller size. Depending on tool support, you may also use the .N (narrow) suffix to specify a 16-bit Thumb instruction.

Again, this syntax is for ARM assembler tools. Other assemblers might have slightly different syntax. If no suffix is given, the assembler might choose the instruction for you, with the minimum code size.

In most cases, applications will be coded in C, and the C compilers will use 16-bit instructions if possible due to smaller code size. However, when the immediate data exceed a certain range or when the operation can be better handled with a 32-bit Thumb-2 instruction, the 32-bit instruction will be used.

The 32-bit Thumb-2 instructions can be half word aligned. For example, you can have a 32-bit instruction located in a half word location.

```
0x1000 : LDR r0,[r1] ;a 16-bit instructions (occupy 0x1000-0x1001)
0x1002 : RBIT.W r0   ;a 32-bit Thumb-2 instruction (occupy
                     ;  0x1002-0x1005)
```

Most of the 16-bit instructions can only access registers R0–R7; 32-bit Thumb-2 instructions do not have this limitation. However, use of PC (R15) might not be allowed in some of the instructions. Refer to the *ARM v7-M Architecture Application Level Reference Manual* [Ref. 2] (section A4.6) if you need to find out more detail in this area.

4.2 INSTRUCTION LIST

The supported instructions are listed in Tables 4.2 through 4.9. The complete details of each instruction are available in the *ARM v7-M Architecture Application Level Reference Manual* [Ref. 2]. There is also information of the supported instruction sets in Appendix A.

Table 4.2 16-Bit Data Processing Instructions

Instruction	Function
ADC	Add with carry
ADD	Add
ADR	Add PC and an immediate value and put the result in a register
AND	Logical AND
ASR	Arithmetic shift right
BIC	Bit clear (Logical AND one value with the logic inversion of another value)
CMN	Compare negative (compare one data with two's complement of another data and update flags)
CMP	Compare (compare two data and update flags)
CPY	Copy (available from architecture v6; move a value from one high or low register to another high or low register); synonym of MOV instruction
EOR	Exclusive OR
LSL	Logical shift left
LSR	Logical shift right
MOV	Move (can be used for register-to-register transfers or loading immediate data)
MUL	Multiply
MVN	Move NOT (obtain logical inverted value)
NEG	Negate (obtain two's complement value), equivalent to RSB

Table 4.2 16-Bit Data Processing Instructions *Continued*

Instruction	Function
ORR	Logical OR
RSB	Reverse subtract
ROR	Rotate right
SBC	Subtract with carry
SUB	Subtract
TST	Test (use as logical AND; Z flag is updated but AND result is not stored)
REV	Reverse the byte order in a 32-bit register (available from architecture v6)
REV16	Reverse the byte order in each 16-bit half word of a 32-bit register (available from architecture v6)
REVSH	Reverse the byte order in the lower 16-bit half word of a 32-bit register and sign extends the result to 32 bits (available from architecture v6)
SXTB	Signed extend byte (available from architecture v6)
SXTH	Signed extend half word (available from architecture v6)
UXTB	Unsigned extend byte (available from architecture v6)
UXTH	Unsigned extend half word (available from architecture v6)

Table 4.3 16-Bit Branch Instructions

Instruction	Function
B	Branch
B<cond>	Conditional branch
BL	Branch with link; call a subroutine and store the return address in LR (this is actually a 32-bit instruction, but it is also available in Thumb in traditional ARM processors)
BLX	Branch with link and change state (BLX <reg> only)[1]
BX <reg>	Branch with exchange state
CBZ	Compare and branch if zero (architecture v7)
CBNZ	Compare and branch if nonzero (architecture v7)
IT	IF-THEN (architecture v7)

Table 4.4 16-Bit Load and Store Instructions

Instruction	Function
LDR	Load word from memory to register
LDRH	Load half word from memory to register
LDRB	Load byte from memory to register

Continued

[1]BLX with immediate is not supported because it will always try to change to the ARM state, which is not supported in the Cortex-M3. Attempts to use BLX <reg> to change to the ARM state will also result in a fault exception.

Table 4.4 16-Bit Load and Store Instructions *Continued*

Instruction	Function
LDRSH	Load half word from memory, sign extend it, and put it in register
LDRSB	Load byte from memory, sign extend it, and put it in register
STR	Store word from register to memory
STRH	Store half word from register to memory
STRB	Store byte from register to memory
LDM/LDMIA	Load multiple/Load multiple increment after
STM/STMIA	Store multiple/Store multiple increment after
PUSH	Push multiple registers
POP	Pop multiple registers

Table 4.5 Other 16-Bit Instructions

Instruction	Function
SVC	Supervisor call
SEV	Send event
WFE	Sleep and wait for event
WFI	Sleep and wait for interrupt
BKPT	Breakpoint; if debug is enabled, it will enter debug mode (halted), or if debug monitor exception is enabled, it will invoke the debug exception; otherwise, it will invoke a fault exception
NOP	No operation
CPSIE	Enable PRIMASK (CPSIE i)/FAULTMASK (CPSIE f) register (set the register to 0)
CPSID	Disable PRIMASK (CPSID i)/ FAULTMASK (CPSID f) register (set the register to 1)

Table 4.6 32-Bit Data Processing Instructions

Instruction	Function
ADC	Add with carry
ADD	Add
ADDW	Add wide (#immed_12)
ADR	Add PC and an immediate value and put the result in a register
AND	Logical AND
ASR	Arithmetic shift right
BIC	Bit clear (logical AND one value with the logic inversion of another value)
BFC	Bit field clear
BFI	Bit field insert
CMN	Compare negative (compare one data with two's complement of another data and update flags)

Table 4.6 32-Bit Data Processing Instructions *Continued*

Instruction	Function
CMP	Compare (compare two data and update flags)
CLZ	Count leading zero
EOR	Exclusive OR
LSL	Logical shift left
LSR	Logical shift right
MLA	Multiply accumulate
MLS	Multiply and subtract
MOV	Move
MOVW	Move wide (write a 16-bit immediate value to register)
MOVT	Move top (write an immediate value to the top half word of destination reg)
MVN	Move negative
MUL	Multiply
ORR	Logical OR
ORN	Logical OR NOT
RBIT	Reverse bit
REV	Byte reverse word
REV16	Byte reverse packed half word
REVSH	Byte reverse signed half word
ROR	Rotate right
RSB	Reverse subtract
RRX	Rotate right extended
SBC	Subtract with carry
SBFX	Signed bit field extract
SDIV	Signed divide
SMLAL	Signed multiply accumulate long
SMULL	Signed multiply long
SSAT	Signed saturate
SBC	Subtract with carry
SUB	Subtract
SUBW	Subtract wide (#immed_12)
SXTB	Sign extend byte
SXTH	Sign extend half word
TEQ	Test equivalent (use as logical exclusive OR; flags are updated but result is not stored)
TST	Test (use as logical AND; Z flag is updated but AND result is not stored)
UBFX	Unsigned bit field extract
UDIV	Unsigned divide
UMLAL	Unsigned multiply accumulate long
UMULL	Unsigned multiply long
USAT	Unsigned saturate

Continued

Table 4.6 32-Bit Data Processing Instructions *Continued*

Instruction	Function
UXTB	Unsigned extend byte
UXTH	Unsigned extend half word

Table 4.7 32-Bit Load and Store Instructions

Instruction	Function
LDR	Load word data from memory to register
LDRT	Load word data from memory to register with unprivileged access
LDRB	Load byte data from memory to register
LDRBT	Load byte data from memory to register with unprivileged access
LDRH	Load half word data from memory to register
LDRHT	Load half word data from memory to register with unprivileged access
LDRSB	Load byte data from memory, sign extend it, and put it to register
LDRSBT	Load byte data from memory with unprivileged access, sign extend it, and put it to register
LDRSH	Load half word data from memory, sign extend it, and put it to register
LDRSHT	Load half word data from memory with unprivileged access, sign extend it, and put it to register
LDM/LDMIA	Load multiple data from memory to registers
LDMDB	Load multiple decrement before
LDRD	Load double word data from memory to registers
STR	Store word to memory
STRT	Store word to memory with unprivileged access
STRB	Store byte data to memory
STRBT	Store byte data to memory with unprivileged access
STRH	Store half word data to memory
STRHT	Store half word data to memory with unprivileged access
STM/STMIA	Store multiple words from registers to memory
STMDB	Store multiple decrement before
STRD	Store double word data from registers to memory
PUSH	Push multiple registers
POP	Pop multiple registers

Table 4.8 32-Bit Branch Instructions

Instruction	Function
B	Branch
B<cond>	Conditional branch
BL	Branch and link
TBB	Table branch byte; forward branch using a table of single byte offset
TBH	Table branch half word; forward branch using a table of half word offset

Table 4.9 Other 32-Bit Instructions

Instruction	Function
LDREX	Exclusive load word
LDREXH	Exclusive load half word
LDREXB	Exclusive load byte
STREX	Exclusive store word
STREXH	Exclusive store half word
STREXB	Exclusive store byte
CLREX	Clear the local exclusive access record of local processor
MRS	Move special register to general-purpose register
MSR	Move to special register from general-purpose register
NOP	No operation
SEV	Send event
WFE	Sleep and wait for event
WFI	Sleep and wait for interrupt
ISB	Instruction synchronization barrier
DSB	Data synchronization barrier
DMB	Data memory barrier

Table 4.10 Unsupported Thumb Instructions for Traditional ARM Processors

Unsupported Instruction	Function
BLX label	This is branch with link and exchange state. In a format with immediate data, BLX always changes to ARM state. Because the Cortex-M3 does not support the ARM state, instructions like this one that attempt to switch to the ARM state will result in a fault exception called *usage fault*.
SETEND	This Thumb instruction, introduced in architecture v6, switches the endian configuration during run time. Since the Cortex-M3 does not support dynamic endian, using the SETEND instruction will result in a fault exception.

4.2.1 Unsupported Instructions

A number of Thumb instructions are not supported in the Cortex-M3; they are presented in Table 4.10.

A number of instructions listed in the *ARM v7-M Architecture Application Level Reference Manual* are not supported in the Cortex-M3. ARM v7-M architecture allows Thumb-2 coprocessor instructions, but the Cortex-M3 processor does not have any coprocessor support. Therefore, executing the coprocessor instructions shown in Table 4.11 will result in a fault exception (Usage Fault with No-Coprocessor "NOCP" bit in Usage Fault Status Register in NVIC set to 1).

Some of the change process state (CPS) instructions are also not supported in the Cortex-M3 (see Table 4.12). This is because the Program Status register (PSR) definition has changed, so some bits defined in the ARM architecture v6 are not available in the Cortex-M3.

Table 4.11 Unsupported Coprocessor Instructions

Unsupported Instruction	Function
MCR	Move to coprocessor from ARM processor
MCR2	Move to coprocessor from ARM processor
MCRR	Move to coprocessor from two ARM register
MRC	Move to ARM register from coprocessor
MRC2	Move to ARM register from coprocessor
MRRC	Move to two ARM registers from coprocessor
LDC	Load coprocessor; load memory data from a sequence of consecutive memory addresses to a coprocessor
STC	Store coprocessor; stores data from a coprocessor to a sequence of consecutive memory addresses

Table 4.12 Unsupported Change Process State Instructions

Unsupported Instruction	Function
CPS<IE|ID>.W A	There is no A bit in the Cortex-M3
CPS.W #mode	There is no mode bit in the Cortex-M3 PSR

Table 4.13 Unsupported Hint Instructions

Unsupported Instruction	Function
DBG	A hint instruction to debug and trace system
PLD	Preload data; this is a hint instruction for cache memory, however, since there is no cache in the Cortex-M3 processor, this instruction behaves as NOP
PLI	Preload instruction; this is a hint instruction for cache memory, however, since there is no cache in the Cortex-M3 processor, this instruction behaves as NOP
YIELD	A hint instruction to allow multithreading software to indicate to hardware that it is doing a task that can be swapped out to improve overall system performance.

In addition, the hint instructions shown in Table 4.13 will behave as NOP in the Cortex-M3.

All other undefined instructions, when executed, will cause the usage fault exception to take place.

4.3 INSTRUCTION DESCRIPTIONS

Here, we introduce some of the commonly used syntax for ARM assembly code. Some of the instructions have various options such as barrel shifter; these will not be fully covered in this chapter.

4.3.1 Assembler Language: Moving Data

One of the most basic functions in a processor is transfer of data. In the Cortex-M3, data transfers can be of one of the following types:

- Moving data between register and register
- Moving data between memory and register
- Moving data between special register and register
- Moving an immediate data value into a register

The command to move data between registers is MOV (move). For example, moving data from register R3 to register R8 looks like this:

```
MOV R8, R3
```

Another instruction can generate the negative value of the original data; it is called MVN (move negative).

The basic instructions for accessing memory are Load and Store. Load (LDR) transfers data from memory to registers, and Store transfers data from registers to memory. The transfers can be in different data sizes (byte, half word, word, and double word), as outlined in Table 4.14.

Multiple Load and Store operations can be combined into single instructions called LDM (Load Multiple) and STM (Store Multiple), as outlined in Table 4.15.

The exclamation mark (!) in the instruction specifies whether the register *Rd* should be updated after the instruction is completed. For example, if R8 equals 0x8000:

```
STMIA.W R8!, {R0-R3} ; R8 changed to 0x8010 after store
                     ; (increment by 4 words)
STMIA.W R8 , {R0-R3} ; R8 unchanged after store
```

ARM processors also support memory accesses with preindexing and postindexing. For preindexing, the register holding the memory address is adjusted. The memory transfer then takes place with the updated address. For example,

```
LDR.W R0,[R1, #offset]! ; Read memory[R1+offset], with R1
                        ; update to R1+offset
```

Table 4.14 Commonly Used Memory Access Instructions

Example	Description
`LDRB Rd, [Rn, #offset]`	Read byte from memory location Rn + offset
`LDRH Rd, [Rn, #offset]`	Read half word from memory location Rn + offset
`LDR Rd, [Rn, #offset]`	Read word from memory location Rn + offset
`LDRD Rd1,Rd2, [Rn, #offset]`	Read double word from memory location Rn + offset
`STRB Rd, [Rn, #offset]`	Store byte to memory location Rn + offset
`STRH Rd, [Rn, #offset]`	Store half word to memory location Rn + offset
`STR Rd, [Rn, #offset]`	Store word to memory location Rn + offset
`STRD Rd1,Rd2, [Rn, #offset]`	Store double word to memory location Rn + offset

Table 4.15 Multiple Memory Access Instructions

Example	Description
`LDMIA Rd!,<reg list>`	Read multiple words from memory location specified by *Rd*; address increment after (IA) each transfer (16-bit Thumb instruction)
`STMIA Rd!,<reg list>`	Store multiple words to memory location specified by *Rd*; address increment after (IA) each transfer (16-bit Thumb instruction)
`LDMIA.W Rd(!),<reg list>`	Read multiple words from memory location specified by *Rd*; address increment after each read (.W specified it is a 32-bit Thumb-2 instruction)
`LDMDB.W Rd(!),<reg list>`	Read multiple words from memory location specified by *Rd*; address Decrement Before (DB) each read (.W specified it is a 32-bit Thumb-2 instruction)
`STMIA.W Rd(!),<reg list>`	Write multiple words to memory location specified by *Rd*; address increment after each read (.W specified it is a 32-bit Thumb-2 instruction)
`STMDB.W Rd(!),<reg list>`	Write multiple words to memory location specified by *Rd*; address DB each read (.W specified it is a 32-bit Thumb-2 instruction)

Table 4.16 Examples of Preindexing Memory Access Instructions

Example	Description

`LDR.W  Rd,  [Rn, #offset]!`
`LDRB.W Rd,  [Rn, #offset]!`
`LDRH.W Rd,  [Rn, #offset]!`
`LDRD.W Rd1, Rd2,[Rn, #offset]!`
	Preindexing load instructions for various sizes (word, byte, half word, and double word)

`LDRSB.W Rd, [Rn, #offset]!` `LDRSH.W Rd, [Rn, #offset]!`	Preindexing load instructions for various sizes with sign extend (byte, half word)

`STR.W  Rd,  [Rn, #offset]!`
`STRB.W Rd,  [Rn, #offset]!`
`STRH.W Rd,  [Rn, #offset]!`
`STRD.W Rd1, Rd2,[Rn, #offset]!`
	Preindexing store instructions for various sizes (word, byte, half word, and double word)

The use of the "!" indicates the update of base register R1. The "!" is optional; without it, the instruction would be just a normal memory transfer with offset from a base address. The preindexing memory access instructions include load and store instructions of various transfer sizes (see Table 4.16).

Postindexing memory access instructions carry out the memory transfer using the base address specified by the register and then update the address register afterward. For example,

```
LDR.W R0,[R1], #offset ; Read memory[R1], with R1
                       ; updated to R1+offset
```

When a postindexing instruction is used, there is no need to use the "!" sign, because all postindexing instructions update the base address register, whereas in preindexing you might choose whether to update the base address register or not.

Similarly to preindexing, postindexing memory access instructions are available for different transfer sizes (see Table 4.17).

Table 4.17 Examples of Postindexing Memory Access Instructions

Example	Description

`LDR.W Rd, [Rn], #offset`
`LDRB.W Rd, [Rn], #offset`
`LDRH.W Rd, [Rn], #offset`
`LDRD.W Rd1, Rd2,[Rn], #offset`
	Postindexing load instructions for various sizes (word, byte, half word, and double word)

`LDRSB.W Rd, [Rn], #offset` `LDRSH.W Rd, [Rn], #offset`	Postindexing load instructions for various sizes with sign extend (byte, half word)

`STR.W Rd, [Rn], #offset`
`STRB.W Rd, [Rn], #offset`
`STRH.W Rd, [Rn], #offset`
`STRD.W Rd1, Rd2,[Rn], #offset`
	Postindexing store instructions for various sizes (word, byte, half word, and double word)

Two other types of memory operation are stack PUSH and stack POP. For example,

```
PUSH {R0, R4-R7, R9} ; Push R0, R4, R5, R6, R7, R9 into
                     ; stack memory
POP {R2,R3}          ; Pop R2 and R3 from stack
```

Usually a PUSH instruction will have a corresponding POP with the same register list, but this is not always necessary. For example, a common exception is when POP is used as a function return:

```
PUSH {R0-R3, LR} ; Save register contents at beginning of
                 ; subroutine
....             ; Processing
POP {R0-R3, PC}  ; restore registers and return
```

In this case, instead of popping the LR register back and then branching to the address in LR, we POP the address value directly in the program counter.

As mentioned in Chapter 3, the Cortex-M3 has a number of special registers. To access these registers, we use the instructions MRS and MSR. For example,

```
MRS R0, PSR     ; Read Processor status word into R0
MSR CONTROL, R1 ; Write value of R1 into control register
```

Unless you're accessing the APSR, you can use MSR or MRS to access other special registers only in privileged mode.

Moving immediate data into a register is a common thing to do. For example, you might want to access a peripheral register, so you need to put the address value into a register beforehand. For small values (8 bits or less), you can use MOVS (move). For example,

```
MOVS R0, #0x12 ; Set R0 to 0x12
```

For a larger value (over 8 bits), you might need to use a Thumb-2 move instruction. For example,

```
MOVW.W R0, #0x789A ; Set R0 to 0x789A
```

Or if the value is 32-bit, you can use two instructions to set the upper and lower halves:

```
MOVW.W R0,#0x789A ; Set R0 lower half to 0x789A
MOVT.W R0,#0x3456 ; Set R0 upper half to 0x3456. Now
                  ; R0=0x3456789A
```

Alternatively, you can also use LDR (a pseudo-instruction provided in ARM assembler). For example,

```
LDR R0, =0x3456789A
```

This is not a real assembler command, but the ARM assembler will convert it into a PC relative load instruction to produce the required data. To generate 32-bit immediate data, using LDR is recommended rather than the MOVW.W and MOVT.W combination because it gives better readability and the assembler might be able to reduce the memory being used if the same immediate data are reused in several places of the same program.

4.3.2 LDR and ADR Pseudo-Instructions

Both LDR and ADR pseudo-instructions can be used to set registers to a program address value. They have different syntaxes and behaviors. For LDR, if the address is a program address value, the assembler will automatically set the LSB to 1. For example,

```
      LDR R0, =address1 ; R0 set to 0x4001
      ...
address1        ; address here is 0x4000
      MOV R0, R1 ; address1 contains program code
      ...
```

You will find that the LDR instruction will put 0x4001 into R1; the LSB is set to 1 to indicate that it is Thumb code. If *address1* is a data address, LSB will not be changed. For example,

```
      LDR R0, =address1 ; R0 set to 0x4000
      ...
address1     ; address here is 0x4000
      DCD 0x0 ; address1 contains data
      ...
```

For ADR, you can load the address value of a program code into a register without setting the LSB automatically. For example,

```
      ADR R0, address1
      ...
address1        ; (address here is 0x4000)
      MOV R0, R1 ; address1 contains program code
      ...
```

You will get 0x4000 in the ADR instruction. Note that there is no equal sign (=) in the ADR statement.

LDR obtains the immediate data by putting the data in the program code and uses a PC relative load to get the data into the register. ADR tries to generate the immediate value by adding or subtracting instructions (for example, based on the current PC value). As a result, it is not possible to create all immediate values using ADR, and the target address label must be in a close range. However, using ADR can generate smaller code sizes compared with LDR.

The 16-bit version of ADR requires that the target address must be word aligned (address value is a multiple of 4). If the target address is not word aligned, you can use the 32-bit version of ADR instruction "ADR.W." If the target address is more than 4095 bytes of current PC, you can use "ADRL" pseudo-instruction, which gives 1 MB range.

4.3.3 Assembler Language: Processing Data

The Cortex-M3 provides many different instructions for data processing. A few basic ones are introduced here. Many data operation instructions can have multiple instruction formats. For example, an ADD instruction can operate between two registers or between one register and an immediate data value:

```
ADD   R0, R0, R1    ; R0 = R0 + R1
ADDS  R0, R0, #0x12 ; R0 = R0 + 0x12
ADD.W R0, R1, R2    ; R0 = R1 + R2
```

These are all ADD instructions, but they have different syntaxes and binary coding.

With the traditional Thumb instruction syntax, when 16-bit Thumb code is used, an ADD instruction can change the flags in the PSR. However, 32-bit Thumb-2 code can either change a flag or keep it unchanged. To separate the two different operations, the *S* suffix should be used if the following operation depends on the flags:

```
ADD.W  R0, R1, R2 ; Flag unchanged
ADDS.W R0, R1, R2 ; Flag change
```

Aside from ADD instructions, the arithmetic functions that the Cortex-M3 supports include subtract (SUB), multiply (MUL), and unsigned and signed divide (UDIV/SDIV). Table 4.18 shows some of the most commonly used arithmetic instructions.

Table 4.18 Examples of Arithmetic Instructions

Instruction	Operation

`ADD Rd, Rn, Rm ; Rd = Rn + Rm`
`ADD Rd, Rd, Rm ; Rd = Rd + Rm`
`ADD Rd, #immed ; Rd = Rd + #immed`
`ADD Rd, Rn, # immed ; Rd = Rn + #immed`
	ADD operation

`ADC Rd, Rn, Rm ; Rd = Rn + Rm + carry`
`ADC Rd, Rd, Rm ; Rd = Rd + Rm + carry`
`ADC Rd, #immed ; Rd = Rd + #immed + carry`
	ADD with carry

`ADDW Rd, Rn,#immed ; Rd = Rn + #immed`	ADD register with 12-bit immediate value

`SUB Rd, Rn, Rm ; Rd = Rn – Rm`
`SUB Rd, #immed ; Rd = Rd – #immed`
`SUB Rd, Rn,#immed ; Rd = Rn – #immed`
	SUBTRACT

`SBC Rd, Rm ; Rd = Rd – Rm – borrow`
`SBC.W Rd, Rn, #immed ; Rd = Rn – #immed – borrow`
`SBC.W Rd, Rn, Rm ; Rd = Rn – Rm – borrow`
	SUBTRACT with borrow (not carry)

`RSB.W Rd, Rn, #immed ; Rd = #immed -Rn` `RSB.W Rd, Rn, Rm ; Rd = Rm – Rn`	Reverse subtract
`MUL Rd, Rm ; Rd = Rd * Rm` `MUL.W Rd, Rn, Rm ; Rd = Rn * Rm`	Multiply
`UDIV Rd, Rn, Rm ; Rd = Rn/Rm` `SDIV Rd, Rn, Rm ; Rd = Rn/Rm`	Unsigned and signed divide

These instructions can be used with or without the "S" suffix to determine if the APSR should be updated. In most cases, if UAL syntax is selected and if "S" suffix is not used, the 32-bit version of the instructions would be selected as most of the 16-bit Thumb instructions update APSR.

The Cortex-M3 also supports 32-bit multiply instructions and multiply accumulate instructions that give 64-bit results. These instructions support signed or unsigned values (see Table 4.19).

Another group of data processing instructions are the logical operations instructions and logical operations such as AND, ORR (or), and shift and rotate functions. Table 4.20 shows some of the most commonly used logical instructions. These instructions can be used with or without the "S" suffix to determine if the APSR should be updated. If UAL syntax is used and if "S" suffix is not used, the 32-bit version of the instructions would be selected as all of the 16-bit logic operation instructions update APSR.

The Cortex-M3 provides rotate and shift instructions. In some cases, the rotate operation can be combined with other operations (for example, in memory address offset calculation for load/store instructions). For standalone rotate/shift operations, the instructions shown in Table 4.21 are provided. Again, a 32-bit version of the instruction is used if "S" suffix is not used and if UAL syntax is used.

Table 4.19 32-Bit Multiply Instructions

Instruction	Operation
`SMULL RdLo, RdHi, Rn, Rm ; {RdHi,RdLo} = Rn * Rm` `SMLAL RdLo, RdHi, Rn, Rm ; {RdHi,RdLo} += Rn * Rm`	32-bit multiply instructions for signed values
`UMULL RdLo, RdHi, Rn, Rm ; {RdHi,RdLo} = Rn * Rm` `UMLAL RdLo, RdHi, Rn, Rm ; {RdHi,RdLo} += Rn * Rm`	32-bit multiply instructions for unsigned values

Table 4.20 Logic Operation Instructions

Instruction	Operation

`AND Rd, Rn ; Rd = Rd & Rn`
`AND.W Rd, Rn,#immed ; Rd = Rn & #immed`
`AND.W Rd, Rn, Rm ; Rd = Rn & Rd`
	Bitwise AND

`ORRRd, Rn ; Rd = Rd | Rn`
`ORR.W Rd, Rn,#immed ; Rd = Rn | #immed`
`ORR.W Rd, Rn, Rm ; Rd = Rn | Rd`
	Bitwise OR

`BIC Rd, Rn ; Rd = Rd & (~Rn)`
`BIC.W Rd, Rn,#immed ; Rd = Rn &(~#immed)`
`BIC.W Rd, Rn, Rm ; Rd = Rn &(~Rd)`
	Bit clear

`ORN.W Rd, Rn,#immed ; Rd = Rn | (~#immed)` `ORN.W Rd, Rn, Rm ; Rd = Rn | (~Rd)`	Bitwise OR NOT

`EOR Rd, Rn ; Rd = Rd ^ Rn`
`EOR.W Rd, Rn,#immed ; Rd = Rn | #immed`
`EOR.W Rd, Rn, Rm ; Rd = Rn | Rd`
	Bitwise Exclusive OR

Table 4.21 Shift and Rotate Instructions

Instruction	Operation

`ASR Rd, Rn,#immed ; Rd = Rn » immed`
`ASRRd, Rn ; Rd = Rd » Rn`
`ASR.W Rd, Rn, Rm ; Rd = Rn » Rm`
	Arithmetic shift right

`LSLRd, Rn,#immed ; Rd = Rn « immed`
`LSLRd, Rn ; Rd = Rd « Rn`
`LSL.W Rd, Rn, Rm ; Rd = Rn « Rm`
	Logical shift left

`LSRRd, Rn,#immed ; Rd = Rn » immed`
`LSRRd, Rn ; Rd = Rd » Rn`
`LSR.W Rd, Rn, Rm ; Rd = Rn » Rm`
	Logical shift right

`ROR Rd, Rn ; Rd rot by Rn`
`ROR.W Rd, Rn,#immed ; Rd = Rn rot by immed`
`ROR.W Rd, Rn, Rm ; Rd = Rn rot by Rm`
	Rotate right

`RRX.W Rd, Rn ; {C, Rd} = {Rn, C}`	Rotate right extended

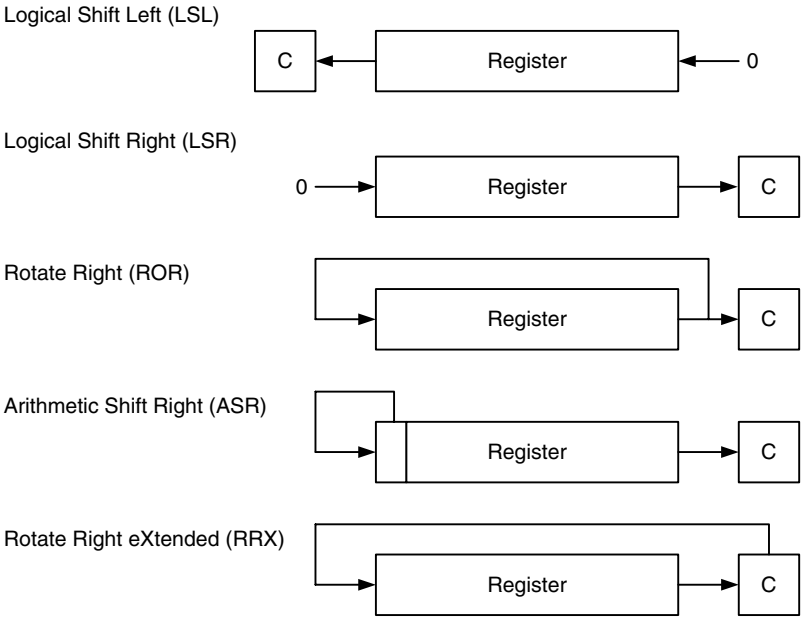

FIGURE 4.1

Shift and Rotate Instructions.

In UAL syntax, the rotate and shift operations can also update the carry flag if the *S* suffix is used (and always update the carry flag if the 16-bit Thumb code is used). See Figure 4.1.

If the shift or rotate operation shifts the register position by multiple bits, the value of the carry flag *C* will be the last bit that shifts out of the register.

WHY IS THERE ROTATE RIGHT BUT NO ROTATE LEFT?

The rotate left operation can be replaced by a rotate right operation with a different rotate offset. For example, a rotate left by 4-bit operation can be written as a rotate right by 28-bit instruction, which gives the same result and takes the same amount of time to execute.

Table 4.22 Sign Extend Instructions

Instruction	Operation
`SXTB Rd, Rm ; Rd = signext(Rm[7:0])`	Sign extend byte data into word
`SXTH Rd, Rm ; Rd = signext(Rm[15:0])`	Sign extend half word data into word

Table 4.23 Data Reverse Ordering Instructions

Instruction	Operation
`REV Rd, Rn ; Rd = rev(Rn)`	Reverse bytes in word
`REV16 Rd, Rn ; Rd = rev16(Rn)`	Reverse bytes in each half word
`REVSH Rd, Rn ; Rd = revsh(Rn)`	Reverse bytes in bottom half word and sign extend the result

For conversion of signed data from byte or half word to word, the Cortex-M3 provides the two instructions shown in Table 4.22. Both 16-bit and 32-bit versions are available. The 16-bit version can only access low registers.

Another group of data processing instructions is used for reversing data bytes in a register (see Table 4.23). These instructions are usually used for conversion between little endian and big endian data. See Figure 4.2. Both 16-bit and 32-bit versions are available. The 16-bit version can only access low registers.

The last group of data processing instructions is for bit field processing. They include the instructions shown in Table 4.24. Examples of these instructions are provided in a later part of this chapter.

4.3.4 Assembler Language: Call and Unconditional Branch

The most basic branch instructions are as follows:

```
B label ; Branch to a labeled address
BX reg  ; Branch to an address specified by a register
```

In BX instructions, the LSB of the value contained in the register determines the next state (Thumb/ARM) of the processor. In the Cortex-M3, because it is always in Thumb state, this bit should be set to 1. If it is zero, the program will cause a usage fault exception because it is trying to switch the processor into ARM state (See Figure 4.2.).

To call a function, the branch and link instructions should be used.

```
BL label  ; Branch to a labeled address and save return
          ; address in LR
```

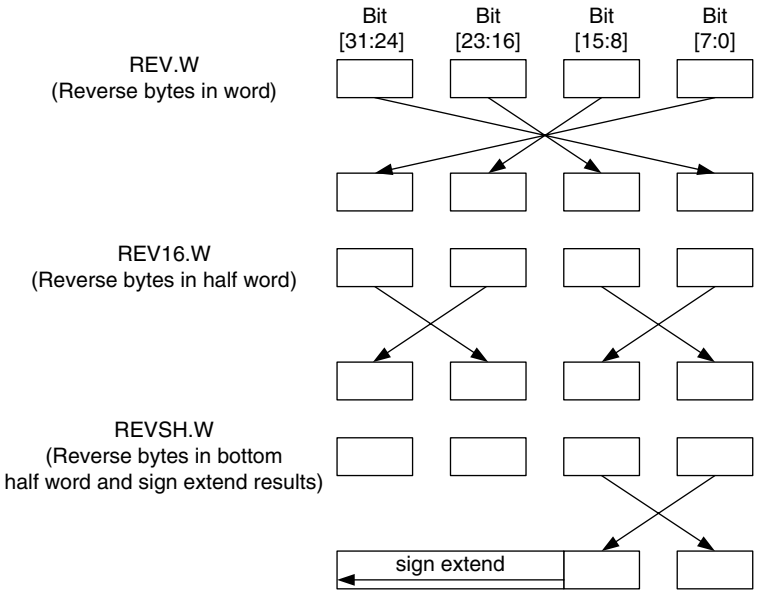

FIGURE 4.2

Operation of Reverse instructions.

Table 4.24 Bit Field Processing and Manipulation Instructions

Instruction	Operation
`BFC.W Rd, Rn, #<width>`	Clear bit field within a register
`BFI.W Rd, Rn, #<lsb>, #<width>`	Insert bit field to a register
`CLZ.W Rd, Rn`	Count leading zero
`RBIT.W Rd, Rn`	Reverse bit order in register
`SBFX.W Rd, Rn, #<lsb>, #<width>`	Copy bit field from source and sign extend it
`UBFX.W Rd, Rn, #<lsb>, #<width>`	Copy bit field from source register

```
BLX reg  ; Branch to an address specified by a register and
         ; save return
         ; address in LR.
```

With these instructions, the return address will be stored in the link register (LR) and the function can be terminated using BX LR, which causes program control to return to the calling process. However, when using BLX, make sure that the LSB of the register is 1. Otherwise the processor will produce a fault exception because it is an attempt to switch to the ARM state.

You can also carry out a branch operation using MOV instructions and LDR instructions. For example,

```
MOV R15, R0   ; Branch to an address inside R0
LDR R15, [R0] ; Branch to an address in memory location
              ; specified by R0
```

```
POP {R15}       ; Do a stack pop operation, and change the
                ; program counter value
                ; to the result value.
```

When using these methods to carry out branches, you also need to make sure that the LSB of the new program counter value is 0x1. Otherwise, a usage fault exception will be generated because it will try to switch the processor to ARM mode, which is not allowed in the Cortex-M3 redundancy.

SAVE THE LR IF YOU NEED TO CALL A SUBROUTINE

The BL instruction will destroy the current content of your LR. So, if your program code needs the LR later, you should save your LR before you use BL. The common method is to push the LR to stack in the beginning of your subroutine. For example,

```
main
      ...
      BL functionA
      ...
functionA
      PUSH {LR} ; Save LR content to stack
      ...
      BL functionB
      ...
      POP {PC} ; Use stacked LR content to return to main
functionB
      PUSH {LR}
      ...
      POP {PC} ; Use stacked LR content to return to functionA
```

In addition, if the subroutine you call is a C function, you might also need to save the contents in R0–R3 and R12 if these values will be needed at a later stage. According to *AAPCS* [Ref. 5], the contents in these registers could be changed by a C function.

4.3.5 Assembler Language: Decisions and Conditional Branches

Most conditional branches in ARM processors use flags in the APSR to determine whether a branch should be carried out. In the APSR, there are five flag bits; four of them are used for branch decisions (see Table 4.25).

There is another flag bit at bit[27], called the *Q flag*. It is for saturation math operations and is not used for conditional branches.

Table 4.25 Flag Bits in APSR that Can Be Used for Conditional Branches

Flag	PSR Bit	Description
N	31	Negative flag (last operation result is a negative value)
Z	30	Zero (last operation result returns a zero value)
C	29	Carry (last operation returns a carry out or borrow)
V	28	Overflow (last operation results in an overflow)

FLAGS IN ARM PROCESSORS

Often, data processing instructions change the flags in the PSR. The flags might be used for branch decisions, or they can be used as part of the input for the next instruction. The ARM processor normally contains at least the *Z, N, C*, and *V* flags, which are updated by execution of data processing instructions.

- Z (Zero) flag: This flag is set when the result of an instruction has a zero value or when a comparison of two data returns an equal result.
- N (Negative) flag: This flag is set when the result of an instruction has a negative value (bit 31 is 1).
- C (Carry) flag: This flag is for unsigned data processing—for example, in add (ADD) it is set when an overflow occurs; in subtract (SUB) it is set when a borrow did not occur (borrow is the invert of carry).
- V (Overflow) flag: This flag is for signed data processing; for example, in an add (ADD), when two positive values added together produce a negative value, or when two negative values added together produce a positive value.

These flags can also have special results when used with shift and rotate instructions. Refer to the *ARM v7-M Architecture Application Level Reference Manual* [Ref. 2] for details.

With combinations of the four flags (*N*, *Z*, *C*, and *V*), 15 branch conditions are defined (see Table 4.26). Using these conditions, branch instructions can be written as, for example,

```
BEQ label ; Branch to address 'label' if Z flag is set
```

You can also use the Thumb-2 version if your branch target is further away. For example,

```
BEQ.W label ; Branch to address 'label' if Z flag is set
```

Table 4.26 Conditions for Branches or Other Conditional Operations

Symbol	Condition	Flag
EQ	Equal	Z set
NE	Not equal	Z clear
CS/HS	Carry set/unsigned higher or same	C set
CC/LO	Carry clear/unsigned lower	C clear
MI	Minus/negative	N set
PL	Plus/positive or zero	N clear
VS	Overflow	V set
VC	No overflow	V clear
HI	Unsigned higher	C set and Z clear
LS	Unsigned lower or same	C clear or Z set
GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
LT	Signed less than	N set and V clear, or N clear and V set (N != V)
GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0, N == V)
LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
AL	Always (unconditional)	—

The defined branch conditions can also be used in IF-THEN-ELSE structures. For example,

```
CMP R0, R1    ; Compare R0 and R1
ITTEE GT      ; If R0 > R1 Then
              ; if true, first 2 statements execute,
              ; if false, other 2 statements execute
MOVGT R2, R0 ;      R2 = R0
MOVGT R3, R1 ;      R3 = R1
MOVLE R2, R0 ; Else R2 = R1
MOVLE R3, R1 ;      R3 = R0
```

APSR flags can be affected by the following:

- Most of the 16-bit ALU instructions
- 32-bit (Thumb-2) ALU instructions with the *S* suffix; for example, ADDS.W
- Compare (e.g., CMP) and Test (e.g., TST, TEQ)
- Write to APSR/xPSR directly

Most of the 16-bit Thumb arithmetic instructions affect the *N*, *Z*, *C*, and *V* flags. With 32-bit Thumb-2 instructions, the ALU operation can either change flags or not change flags. For example,

```
ADDS.W R0, R1, R2 ; This 32-bit Thumb instruction updates flag
ADD.W  R0, R1, R2 ; This 32-bit Thumb instruction does not
                  ; update flag
```

Be careful when reusing program code from old projects. If the old project is in tradition Thumb syntax; for example, "CODE16" directive is used with ARM assembler, then

```
ADD R0, R1   ; This 16-bit Thumb instruction updates flag
ADD R0, #0x1 ; This 16-bit Thumb instruction updates flag
```

However, if you used the same code in UAL syntax; that is "THUMB" directive is used with ARM assembler, then

```
ADD R0, R1   ; This 16-bit Thumb instruction does not
             ; update flag
ADD R0, #0x1 ; This will become a 32-bit Thumb instruction
             ; that does not update flag
```

To make sure that the code works correctly with different tools, you should always use the *S* suffix if the flags need to be updated for conditional operations such as conditional branches.

The compare (CMP) instruction subtracts two values and updates the flags (just like SUBS), but the result is not stored in any registers. CMP can have the following formats:

```
CMP R0, R1    ; Calculate R0 - R1 and update flag
CMP R0, #0x12 ; Calculate R0 - 0x12 and update flag
```

A similar instruction is the CMN (compare negative). It compares one value to the negative (two's complement) of a second value; the flags are updated, but the result is not stored in any registers:

```
CMN R0, R1    ; Calculate R0 - (-R1) and update flag
CMN R0, #0x12 ; Calculate R0 - (-0x12) and update flag
```

The TST (test) instruction is more like the AND instruction. It ANDs two values and updates the flags. However, the result is not stored in any register. Similarly to CMP, it has two input formats:

```
TST R0, R1    ; Calculate R0 AND R1 and update flag
TST R0, #0x12 ; Calculate R0 AND 0x12 and update flag
```

4.3.6 Assembler Language: Combined Compare and Conditional Branch

With ARM architecture v7-M, two new instructions are provided on the Cortex-M3 to supply a simple compare with zero and conditional branch operations. These are CBZ (compare and branch if zero) and CBNZ (compare and branch if nonzero).

The compare and branch instructions only support forward branches. For example,

```
i = 5;
while (i != 0 ){
func1(); ; call a function
i--;
}
```

This can be compiled into the following:

```
      MOV R0, #5       ; Set loop counter
loop1 CBZ R0,loop1exit ; if loop counter = 0 then exit the loop
      BL  func1        ; call a function
      SUB R0, #1       ; loop counter decrement
      B   loop1        ; next loop
loop1exit
```

The usage of CBNZ is similar to CBZ, apart from the fact that the branch is taken if the *Z* flag is not set (result is not zero). For example,

```
status = strchr(email_address, '@');
if (status == 0){//status is 0 if @ is not in email_address
      show_error_message();
      exit(1);
      }
```

This can be compiled into the following:

```
      ...
      BL   strchr
      CBNZ R0, email_looks_okay ; Branch if result is not zero
      BL   show_error_message
      BL   exit
email_looks_okay
      ...
```

The APSR value is not affected by the CBZ and CBNZ instructions.

Assembler Language: Conditional Execution Using IT Instructions

The IT (IF-THEN) block is very useful for handling small conditional code. It avoids branch penalties because there is no change to program flow. It can provide a maximum of four conditionally executed instructions.

In IT instruction blocks, the first line must be the IT instruction, detailing the choice of execution, followed by the condition it checks. The first statement after the IT command must be

TRUE-THEN-EXECUTE, which is always written as *ITxyz*, where *T* means THEN and *E* means ELSE. The second through fourth statements can be either THEN (true) or ELSE (false):

```
IT<x><y><z> <cond>                    ; IT instruction (<x>, <y>,
                                      ; <z> can be T or E)
instr1<cond> <operands>               ; 1st instruction (<cond>
                                      ; must be same as IT)
instr2<cond or not cond> <operands>   ; 2nd instruction (can be
                                      ; <cond> or <!cond>
instr3<cond or not cond> <operands>   ; 3rd instruction (can be
                                      ; <cond> or <!cond>
instr4<cond or not cond> <operands>   ; 4th instruction (can be
                                      ; <cond> or <!cond>
```

If a statement is to be executed when <*cond*> is false, the suffix for the instruction must be the opposite of the condition. For example, the opposite of EQ is NE, the opposite of GT is LE, and so on. The following code shows an example of a simple conditional execution:

```
if (R1<R2) then
   R2=R2-R1
   R2=R2/2
else
   R1=R1-R2
   R1=R1/2
```

In assembly,

```
      CMP     R1, R2 ; If R1 < R2 (less then)
      ITTEE   LT     ; then execute instruction 1 and 2
                     ; (indicated by T)
                     ; else execute instruction 3 and 4
                     ; (indicated by E)
      SUBLT.W R2,R1  ; 1st instruction
      LSRLT.W R2,#1  ; 2nd instruction
      SUBGE.W R1,R2  ; 3rd instruction (notice the GE is
                     ; opposite of LT)
      LSRGE.W R1,#1  ; 4th instruction
```

You can have fewer than four conditionally executed instructions. The minimum is 1. You need to make sure the number of *T* and *E* occurrences in the IT instruction matches the number of conditionally executed instructions after the IT.

If an exception occurs during the IT instruction block, the execution status of the block will be stored in the stacked PSR (in the IT/Interrupt-Continuable Instruction [ICI] bit field). So, when the exception handler completes and the IT block resumes, the rest of the instructions in the block can continue the execution correctly. In the case of using multicycle instructions (for example, multiple load and store) inside an IT block, if an exception takes place during the execution, the whole instruction is abandoned and restarted after the interrupt process is completed.

4.3.7 Assembler Language: Instruction Barrier and Memory Barrier Instructions

The Cortex-M3 supports a number of barrier instructions. These instructions are needed as memory systems get more and more complex. In some cases, if memory barrier instructions are not used, race conditions could occur.

For example, if the memory map can be switched by a hardware register, after writing to the memory switching register you should use the DSB instruction. Otherwise, if the write to the memory switching register is buffered and takes a few cycles to complete, and the next instruction accesses the switched memory region immediately, the access could be using the old memory map. In some cases, this might result in an invalid access if the memory switching and memory access happen at the same time. Using DSB in this case will make sure that the write to the memory map switching register is completed before a new instruction is executed.

The following are the three barrier instructions in the Cortex-M3:

- DMB
- DSB
- ISB

These instructions are described in Table 4.27.

The memory barrier instructions can be accessed in C using Cortex Microcontroller Software Interface Standard (CMSIS) compliant device driver library as follows:

```
void __DMB(void); // Data Memory Barrier
void __DSB(void); // Data Synchronization Barrier
void __ISB(void); // Instruction Synchronization Barrier
```

The DSB and ISB instructions can be important for self-modifying code. For example, if a program changes its own program code, the next executed instruction should be based on the updated program. However, since the processor is pipelined, the modified instruction location might have already been fetched. Using DSB and then ISB can ensure that the modified program code is fetched again.

Architecturally, the ISB instruction should be used after updating the value of the CONTROL register. In the Cortex-M3 processor, this is not strictly required. But if you want to make sure your application is portable, you should ensure an ISB instruction is used after updating to CONTROL register.

DMB is very useful for multi-processor systems. For example, tasks running on separate processors might use shared memory to communicate with each other. In these environments, the order of memory accesses to the shared memory can be very important. DMB instructions can be inserted between accesses to the shared memory to ensure that the memory access sequence is exactly the same as expected.

Table 4.27 Barrier Instructions

Instruction	Description
DMB	Data memory barrier; ensures that all memory accesses are completed before new memory access is committed
DSB	Data synchronization barrier; ensures that all memory accesses are completed before next instruction is executed
ISB	Instruction synchronization barrier; flushes the pipeline and ensures that all previous instructions are completed before executing new instructions

More details about memory barriers can be found in the *ARM v7-M Architecture Application Level Reference Manual* [Ref. 2].

4.3.8 Assembly Language: Saturation Operations

The Cortex-M3 supports two instructions that provide signed and unsigned saturation operations: SSAT and USAT (for signed data type and unsigned data type, respectively). Saturation is commonly used in signal processing—for example, in signal amplification. When an input signal is amplified, there is a chance that the output will be larger than the allowed output range. If the value is adjusted simply by removing the unused MSB, an overflowed result will cause the signal waveform to be completely deformed (see Figure 4.3).

The saturation operation does not prevent the distortion of the signal, but at least the amount of distortion is greatly reduced in the signal waveform.

The instruction syntax of the SSAT and USAT instructions is outlined here and in Table 4.28.

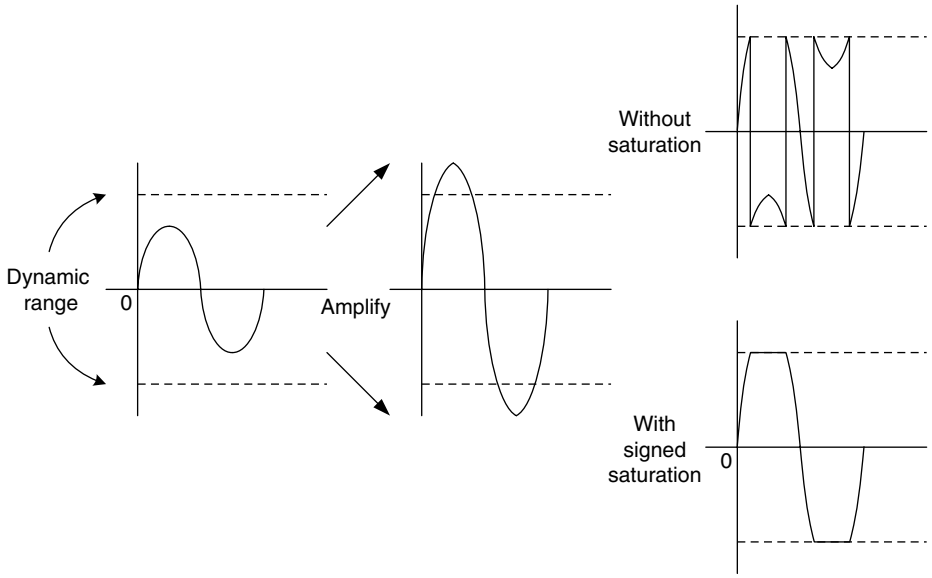

FIGURE 4.3

Signed Saturation Operation.

Table 4.28 Saturation Instructions

Instruction	Description
`SSAT.W <Rd>, #<immed>, <Rn>, {,<shift>}`	Saturation for signed value
`USAT.W <Rd>, #<immed>, <Rn>, {,<shift>}`	Saturation for a signed value into an unsigned value

Rn: Input value
Shift: Shift operation for input value before saturation; optional, can be #LSL N *or* #ASR N
Immed: Bit position where the saturation is carried out
Rd: Destination register

Besides the destination register, the Q-bit in the APSR can also be affected by the result. The Q flag is set if saturation takes place in the operation, and it can be cleared by writing to the APSR (see Table 4.29). For example, if a 32-bit signed value is to be saturated into a 16-bit signed value, the following instruction can be used:

```
SSAT.W R1, #16, R0
```

Similarly, if a 32-bit unsigned value is to saturate into a 16-bit unsigned value, the following instruction can be used:

```
USAT.W R1, #16, R0
```

This will provide a saturation feature that has the properties shown in Figure 4.4.

For the preceding 16-bit saturation example instruction, the output values shown in Table 4.30 can be observed.

Saturation instructions can also be used for data type conversions. For example, they can be used to convert a 32-bit integer value to 16-bit integer value. However, C compilers might not be able to directly use these instructions, so intrinsic function or assembler functions (or embedded/inline assembler code) for the data conversion could be required.

Table 4.29 Examples of Signed Saturation Results

Input (R0)	Output (R1)	Q Bit
0x00020000	0x00007FFF	Set
0x00008000	0x00007FFF	Set
0x00007FFF	0x00007FFF	Unchanged
0x00000000	0x00000000	Unchanged
0xFFFF8000	0xFFFF8000	Unchanged
0xFFFF7FFF	0xFFFF8000	Set
0xFFFE0000	0xFFFF8000	Set

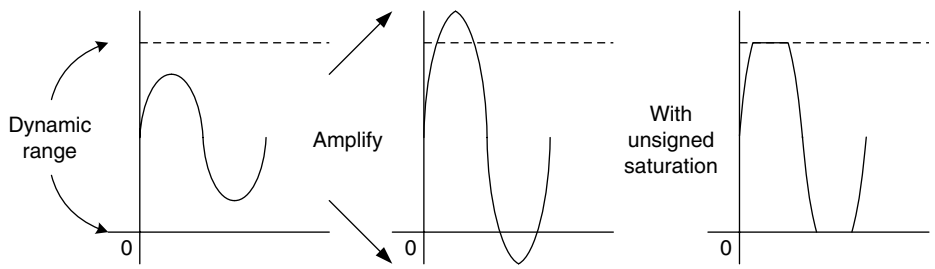

FIGURE 4.4

Unsigned Saturation Operation.

Table 4.30 Examples of Unsigned Saturation Results

Input (R0)	Output (R1)	Q Bit
0x00020000	0x0000FFFF	Set
0x00008000	0x00008000	Unchanged
0x00007FFF	0x00007FFF	Unchanged
0x00000000	0x00000000	Unchanged
0xFFFF8000	0x00000000	Set
0xFFFF8001	0x00000000	Set
0xFFFFFFFF	0x00000000	Set

4.4 SEVERAL USEFUL INSTRUCTIONS IN THE CORTEX-M3

Several useful Thumb-2 instructions from the architecture v7 and v6 are introduced here.

4.4.1 MSR and MRS

These two instructions provide access to the special registers in the Cortex-M3. Here is the syntax of these instructions:

```
MRS <Rn>, <SReg> ; Move from Special Register
MSR <SReg>, <Rn> ; Write to Special Register
```

where <*SReg*> could be one of the options shown in Table 4.31.

For example, the following code can be used to set up the process stack pointer:

```
LDR R0,=0x20008000 ; new value for Process Stack Pointer (PSP)
MSR PSP, R0
```

Unless accessing the APSR, the MRS and MSR instructions can be used in privileged mode only. Otherwise the operation will be ignored, and the returned read data (if MRS is used) will be zero.

After updating the value of the CONTROL register using MSR instruction, it is recommended to add an ISB instruction to ensure that the effect of the update takes place immediately. On the Cortex-M3 processor this is not strictly required, but for software portability (if the software code is to be used on other ARM processor) this is needed.

4.4.2 More on the IF-THEN Instruction Block

The IF-THEN instruction was introduced briefly in an earlier section in this chapter "Conditional Execution Using IT instruction." In here, we will cover more details about this instruction.

The IF-THEN (IT) instructions allow up to four succeeding instructions (called an *IT block*) to be conditionally executed. They are in the following formats as shown in Table 4.32, where,

- <*x*> specifies the execution condition for the second instruction
- <*y*> specifies the execution condition for the third instruction
- <*z*> specifies the execution condition for the fourth instruction
- <*cond*> specifies the base condition of the instruction block; the first instruction following IT executes if <*cond*> is true

Table 4.31 Special Register Names for MRS and MSR Instructions

Symbol	Description
IPSR	Interrupt status register
EPSR	Execution status register (read as zero)
APSR	Flags from previous operation
IEPSR	A composite of IPSR and EPSR
IAPSR	A composite of IPSR and APSR
EAPSR	A composite of EPSR and APSR
PSR	A composite of APSR, EPSR, and IPSR
MSP	Main stack pointer
PSP	Process stack pointer
PRIMASK	Normal exception mask register
BASEPRI	Normal exception priority mask register
BASEPRI_MAX	Same as normal exception priority mask register, with conditional write (new priority level must be higher than the old level)
FAULTMASK	Fault exception mask register (also disables normal interrupts)
CONTROL	Control register

Table 4.32 Various Length of IT Instruction Block

	IT Block (each of <x>, <y> and <z> can either be T [true] or E [else])	Examples
Only one conditional instruction	`IT <cond>` `instr1<cond>`	`IT EQ` `ADDEQ R0, R0, R1`

Two conditional instructions	`IT<x> <cond>`
	`instr1<cond>`
	`instr2<cond or ~(cond)>`
		`ITE GE`
		`ADDGE R0, R0, R1`
		`ADDLT R0, R0, R3`

Three conditional instructions	`IT<x><y> <cond>`
	`instr1<cond>`
	`instr2<cond or ~(cond)>`
	`instr3<cond or ~(cond)>`
		`ITET GT`
		`ADDGT R0, R0, R1`
		`ADDLE R0, R0, R3`
		`ADDGT R2, R4, #1`

Four conditional instructions	`IT<x><y><z> <cond>`
	`instr1<cond>`
	`instr2<cond or ~(cond)>`
	`instr3<cond or ~(cond)>`
	`instr4<cond or ~(cond)>`
		`ITETT NE`
		`ADDNE R0, R0, R1`
		`ADDEQ R0, R0, R3`
		`ADDNE R2, R4, #1`
		`MOVNE R5, R3`

The <*cond*> part uses the same condition symbols as conditional branch. If "AL" is used as <*cond*>, then you cannot use "*E*" in the condition control as it implies the instruction should never get executed.

Each of <*x*>, <*y*>, and <*z*> can be either *T* (THEN) or *E* (ELSE), which refers to the base condition <*cond*>, whereas <*cond*> uses traditional syntax such as EQ, NE, GT, or the like.

Here is an example of IT use:

```
if (R0 equal R1) then {
  R3 = R4 + R5
  R3 = R3/2
  } else {
  R3 = R6 + R7
  R3 = R3/2
  }
```

This can be written as follows:

```
CMP R0, R1        ; Compare R0 and R1
ITTEE EQ          ; If R0 equal R1, Then-Then-Else-Else
ADDEQ R3, R4, R5 ; Add if equal
ASREQ R3, R3, #1 ; Arithmetic shift right if equal
ADDNE R3, R6, R7 ; Add if not equal
ASRNE R3, R3, #1 ; Arithmetic shift right if not equal
```

Aside from using the IT instruction directly, the IT instruction also helps porting of assembly application codes from ARM7TDMI to Cortex-M3. When ARM assembler (including KEIL RealView Microcontroller Development Kit, which is covered in Chapter 20) is used, and if a conditional execution instruction is used in assembly code without IT instruction, the assembler can insert the required IT instruction automatically. An example is shown in Table 4.33. This feature allows existing assembly code to be reused on Cortex-M3 without modifications.

Table 4.33 Automatic Insertion of IT Instruction in ARM Assembler

Original Assembly Code	Disassembled Assembly Code from Generated Object File

`CMP R1, #2`
`ADDEQ R0, R1, #1`
`...`
	`CMP R1, #2`
	`IT EQ`
	`ADDEQ R0, R1, #1`

Note that 16-bit data processing instructions does not update APSR if they are used inside an IT instruction block. If you add the *S* suffix in the conditional executed instruction, the 32-bit version of the instruction would be used by the assembler.

4.4.3 SDIV and UDIV

The syntax for signed and unsigned divide instructions is as follows:

```
SDIV.W <Rd>, <Rn>, <Rm>
UDIV.W <Rd>, <Rn>, <Rm>
```

The result is Rd = Rn/Rm. For example,

```
LDR    R0,=300 ; Decimal 300
MOV    R1,#5
UDIV.W R2, R0, R1
```

This will give you an R2 result of 60 (0x3C).

You can set up the DIVBYZERO bit in the NVIC Configuration Control Register so that when a divide by zero occurs, a fault exception (usage fault) takes place. Otherwise, *<Rd>* will become 0 if a divide by zero takes place.

4.4.4 REV, REVH, and REVSH

REV reverses the byte order in a data word, and REVH reverses the byte order inside a half word. For example, if R0 is 0x12345678, in executing the following:

```
REV  R1, R0
REVH R2, R0
```

R1 will become 0x78563412, and R2 will be 0x34127856. REV and REVH are particularly useful for converting data between big endian and little endian.

REVSH is similar to REVH except that it only processes the lower half word, and then it sign extends the result. For example, if R0 is 0x33448899, running:

```
REVSH R1, R0
```

R1 will become 0xFFFF9988.

4.4.5 Reverse Bit

The RBIT instruction reverses the bit order in a data word. The syntax is as follows:

```
RBIT.W <Rd>, <Rn>
```

This instruction is very useful for processing serial bit streams in data communications. For example, if R0 is 0xB4E10C23 (binary value 1011_0100_1110_0001_0000_1100_0010_0011), executing:

```
RBIT.W R0, R1
```

R0 will become 0xC430872D (binary value 1100_0100_0011_0000_1000_0111_0010_1101).

4.4.6 SXTB, SXTH, UXTB, and UXTH

The four instructions SXTB, SXTH, UXTB, and UXTH are used to extend a byte or half word data into a word. The syntax of the instructions is as follows:

```
SXTB <Rd>, <Rn>
SXTH <Rd>, <Rn>
UXTB <Rd>, <Rn>
UXTH <Rd>, <Rn>
```

For SXTB/SXTH, the data are sign extended using bit[7]/bit[15] of Rn. With UXTB and UXTH, the value is zero extended to 32-bit.

For example, if R0 is 0x55AA8765:

```
SXTB R1, R0 ; R1 = 0x00000065
SXTH R1, R0 ; R1 = 0xFFFF8765
UXTB R1, R0 ; R1 = 0x00000065
UXTH R1, R0 ; R1 = 0x00008765
```

4.4.7 Bit Field Clear and Bit Field Insert

Bit Field Clear (BFC) clears 1–31 adjacent bits in any position of a register. The syntax of the instruction is as follows:

```
BFC.W <Rd>, <#lsb>, <#width>
```

For example,

```
LDR   R0,=0x1234FFFF
BFC.W R0, #4, #8
```

This will give R0 = 0x1234F00F.

Bit Field Insert (BFI) copies 1–31 bits (#width) from one register to any location (#lsb) in another register. The syntax is as follows:

```
BFI.W <Rd>, <Rn>, <#lsb>, <#width>
```

For example,

```
LDR   R0,=0x12345678
LDR   R1,=0x3355AACC
BFI.W R1, R0, #8, #16 ; Insert R0[15:0] to R1[23:8]
```

This will give R1 = 0x335678CC.

4.4.8 UBFX and SBFX

UBFX and SBFX are the unsigned and signed bit field extract instructions. The syntax of the instructions is as follows:

```
UBFX.W <Rd>, <Rn>, <#lsb>, <#width>
SBFX.W <Rd>, <Rn>, <#lsb>, <#width>
```

UBFX extracts a bit field from a register starting from any location (specified by #lsb) with any width (specified by #width), zero extends it, and puts it in the destination register. For example,

```
LDR    R0,=0x5678ABCD
UBFX.W R1, R0, #4, #8
```

This will give R1 = 0x000000BC.

Similarly, SBFX extracts a bit field, but its sign extends it before putting it in a destination register. For example,

```
LDR    R0,=0x5678ABCD
SBFX.W R1, R0, #4, #8
```

This will give R1 = 0xFFFFFFBC.

4.4.9 LDRD and STRD

The two instructions LDRD and STRD transfer two words of data from or into two registers. The syntax of the instructions is as follows:

```
LDRD.W <Rxf>, <Rxf2>, [Rn, #+/-offset]{!} ; Pre-indexed
```

```
LDRD.W <Rxf>, <Rxf2>, [Rn], #+/-offset    ; Post-indexed
STRD.W <Rxf>, <Rxf2>, [Rn, #+/-offset]{!} ; Pre-indexed
STRD.W <Rxf>, <Rxf2>, [Rn], #+/-offset    ; Post-indexed
```

where *<Rxf>* is the first destination/source register and *<Rxf2>* is the second destination/source register. Avoid using same register for *<Rn>* and *<Rxf>* when using LDRD because of an erratum in Cortex-M3 revision 0 to 2.

For example, the following code reads a 64-bit value located in memory address 0x1000 into R0 and R1:

```
LDR    R2,=0x1000
LDRD.W R0, R1, [R2] ; This will gives R0 = memory[0x1000],
                    ; R1 = memory[0x1004]
```

Similarly, we can use STRD to store a 64-bit value in memory. In the following example, preindexed addressing mode is used:

```
LDR    R2,=0x1000          ; Base address
STRD.W R0, R1, [R2, #0x20] ; This will gives memory[0x1020] = R0,
                           ; memory[0x1024] = R1
```

4.4.10 Table Branch Byte and Table Branch Halfword

Table Branch Byte (TBB) and Table Branch Halfword (TBH) are for implementing branch tables. The TBB instruction uses a branch table of byte size offset, and TBH uses a branch table of half word offset. Since the bit 0 of a program counter is always zero, the value in the branch table is multiplied by two before it's added to PC. Furthermore, because the PC value is the current instruction address plus four, the branch range for TBB is (2 × 255) + 4 = 514, and the branch range for TBH is (2 × 65535) + 4 = 131074. Both TBB and TBH support forward branch only.

TBB has this general syntax:

```
TBB.W [Rn, Rm]
```

where *Rn* is the base memory offset and *Rm* is the branch table index. The branch table item for TBB is located at Rn + Rm. Assuming we used PC for Rn, we can see the operation as shown in Figure 4.5.

For TBH instruction, the process is similar except the memory location of the branch table item is located at Rn + 2 x Rm and the maximum branch offset is higher. Again, we assume that Rn is set to PC, as shown in Figure 4.6.

If *Rn* in the table branch instruction is set to R15, the value used for *Rn* will be PC + 4 because of the pipeline in the processor. These two instructions are more likely to be used by a C compiler to generate code for switch (case) statements. Because the values in the branch table are relative to the current program counter, it is not easy to code the branch table content manually in assembler as the address offset value might not be able to be determined during assembly/compile stage, especially if the branch target is in a separate program code file. The coding syntax for calculating TBB/TBH branch table content could be dependent on the development tool. In ARM assembler (*armasm*), the TBB branch table can be created in the following way:

```
        TBB.W [pc, r0] ; when executing this instruction, PC equal
                       ; branchtable
```

```
branchtable
     DCB ((dest0 – branchtable)/2) ; Note that DCB is used because
                                   ; the value is 8-bit
     DCB ((dest1 – branchtable)/2)
     DCB ((dest2 – branchtable)/2)
     DCB ((dest3 – branchtable)/2)
dest0
     ... ; Execute if r0 = 0
dest1
     ... ; Execute if r0 = 1
dest2
     ... ; Execute if r0 = 2
dest3
     ... ; Execute if r0 = 3
```

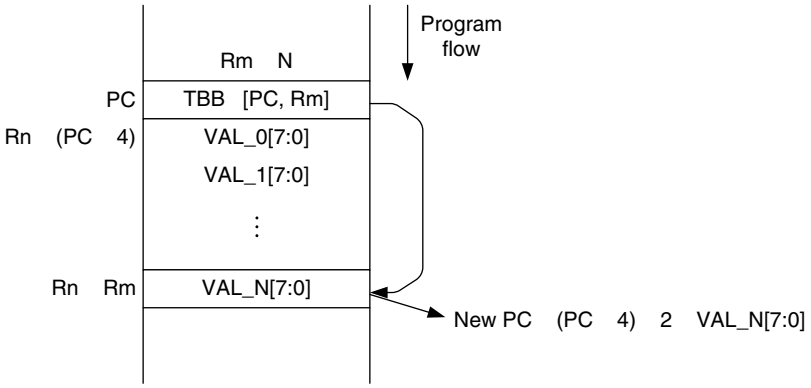

FIGURE 4.5

TBB Operation.

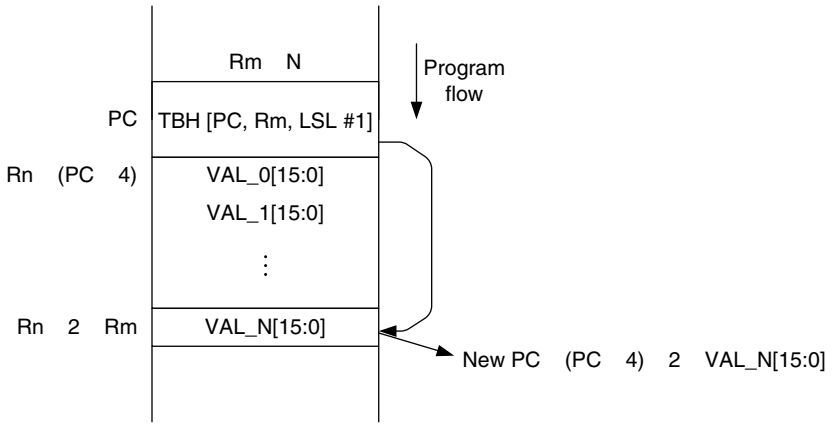

FIGURE 4.6

TBH Operation.

When the TBB instruction is executed, the current PC value is at the address labeled as *branchtable* (because of the pipeline in the processor). Similarly, for TBH instructions, it can be used as follows:

```
     TBH.W [pc, r0, LSL #1]
branchtable
     DCI ((dest0 – branchtable)/2) ; Note that DCI is used because
                                   ; the value is 16-bit
     DCI ((dest1 – branchtable)/2)
     DCI ((dest2 – branchtable)/2)
     DCI ((dest3 – branchtable)/2)
dest0
     ... ; Execute if r0 = 0
dest1
     ... ; Execute if r0 = 1
dest2
     ... ; Execute if r0 = 2
dest3
     ... ; Execute if r0 = 3
```

This page intentionally left blank

CHAPTER

Memory Systems

5

IN THIS CHAPTER

5.1 MEMORY SYSTEM FEATURES OVERVIEW

The Cortex™-M3 processor has different memory architecture from that of traditional ARM processors. First, it has a predefined memory map that specifies which bus interface is to be used when a memory location is accessed. This feature also allows the processor design to optimize the access behavior when different devices are accessed.

Another feature of the memory system in the Cortex-M3 is the bit-band support. This provides atomic operations to bit data in memory or peripherals. The bit-band operations are supported only in special memory regions. This topic is covered in more detail later in this chapter.

The Cortex-M3 memory system also supports unaligned transfers and exclusive accesses. These features are part of the v7-M architecture. Finally, the Cortex-M3 supports both little endian and big endian memory configuration.

5.2 MEMORY MAPS

The Cortex-M3 processor has a fixed memory map (see Figure 5.1). This makes it easier to port software from one Cortex-M3 product to another. For example, components described in previous sections, such as Nested Vectored Interrupt Controller (NVIC) and Memory Protection Unit (MPU), have the

DOI: 10.1016/B978-0-12-382091-4.00011-6

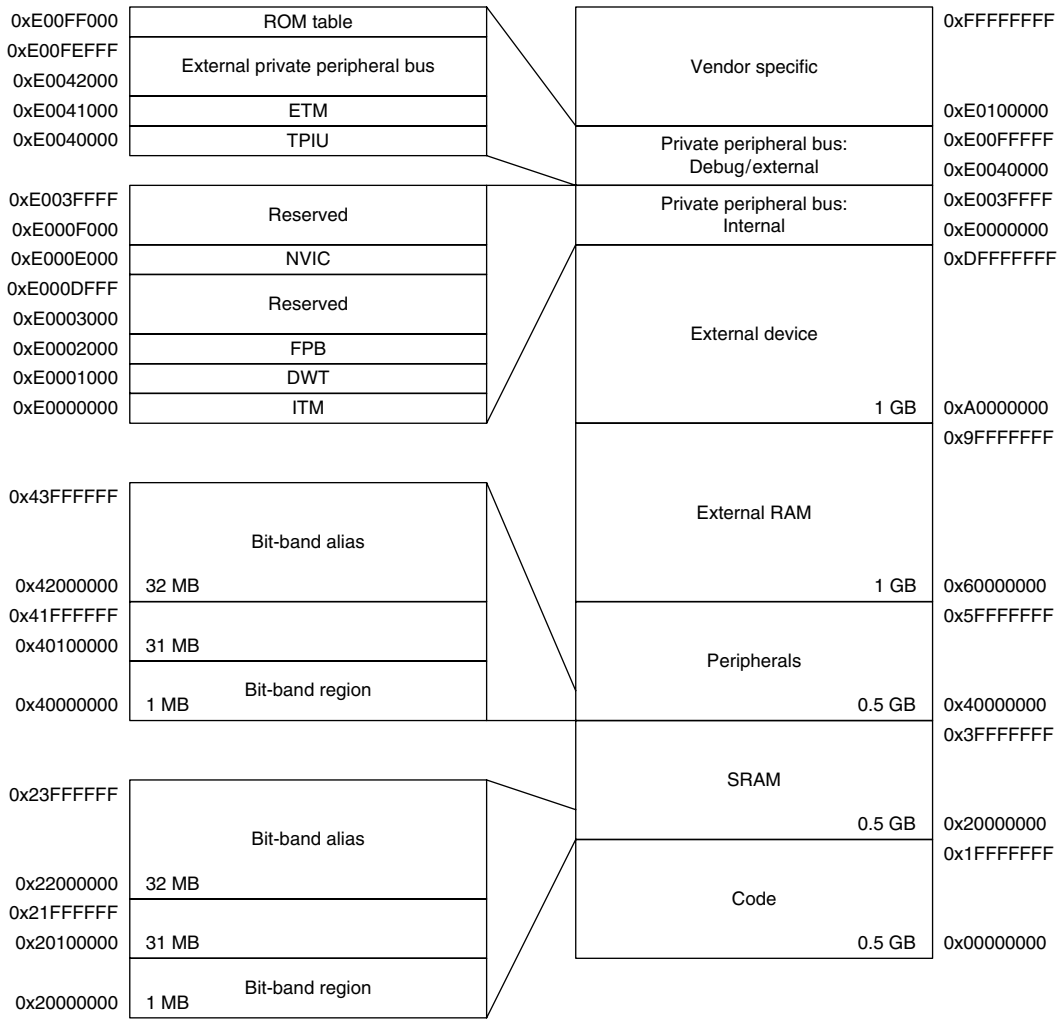

FIGURE 5.1

Cortex-M3 Predefined Memory Map.

same memory locations in all Cortex-M3 products. Nevertheless, the memory map definition allows great flexibility so that manufacturers can differentiate their Cortex-M3-based product from others.

Some of the memory locations are allocated for private peripherals such as debugging components. They are located in the private peripheral memory region. These debugging components include the following:

- Fetch Patch and Breakpoint Unit (FPB)
- Data Watchpoint and Trace Unit (DWT)

- Instrumentation Trace Macrocell (ITM)
- Embedded Trace Macrocell (ETM)
- Trace Port Interface Unit (TPIU)
- ROM table

The details of these components are discussed in later chapters on debugging features.

The Cortex-M3 processor has a total of 4 GB of address space. Program code can be located in the code region, the Static Random Access Memory (SRAM) region, or the external RAM region. However, it is best to put the program code in the code region because with this arrangement, the instruction fetches and data accesses are carried out simultaneously on two separate bus interfaces.

The SRAM memory range is for connecting internal SRAM. Access to this region is carried out via the system interface bus. In this region, a 32-MB range is defined as a bit-band alias. Within the 32-bit-band alias memory range, each word address represents a single bit in the 1-MB bit-band region. A data write access to this bit-band alias memory range will be converted to an atomic READ-MODIFY-WRITE operation to the bit-band region so as to allow a program to set or clear individual data bits in the memory. The bit-band operation applies only to data accesses not instruction fetches. By putting Boolean information (single bits) in the bit-band region, we can pack multiple Boolean data in a single word while still allowing them to be accessible individually via bit-band alias, thus saving memory space without the need for handling READ-MODIFY-WRITE in software. More details on bit-band alias can be found later in this chapter.

Another 0.5-GB block of address range is allocated to on-chip peripherals. Similar to the SRAM region, this region supports bit-band alias and is accessed via the system bus interface. However, instruction execution in this region is not allowed. The bit-band support in the peripheral region makes it easy to access or change control and status bits of peripherals, making it easier to program peripheral control.

Two slots of 1-GB memory space are allocated for external RAM and external devices. The difference between the two is that program execution in the external device region is not allowed, and there are some differences with the caching behaviors.

The last 0.5-GB memory is for the system-level components, internal peripheral buses, external peripheral bus, and vendor-specific system peripherals. There are two segments of the private peripheral bus (PPB):

- Advanced High-Performance Bus (AHB) PPB, for Cortex-M3 internal AHB peripherals only; this includes NVIC, FPB, DWT, and ITM
- Advance Peripheral Bus (APB) PPB, for Cortex-M3 internal APB devices as well as external peripherals (external to the Cortex-M3 processor); the Cortex-M3 allows chip vendors to add additional on-chip APB peripherals on this private peripheral bus via an APB interface

The NVIC is located in a memory region called the system control space (SCS) (see Figure 5.2). Besides providing interrupt control features, this region also provides the control registers for SYSTICK, MPU, and code debugging control.

The remaining unused vendor-specific memory range can be accessed via the system bus interface. However, instruction execution in this region is not allowed.

The Cortex-M3 processor also comes with an optional MPU. Chip manufacturers can decide whether to include the MPU in their products.

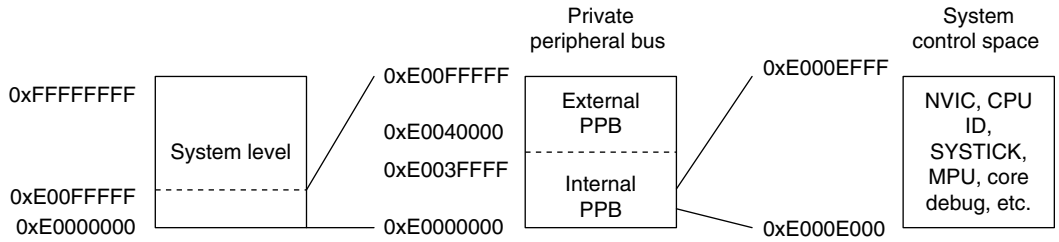

FIGURE 5.2

The System Control Space.

What we have shown in the memory map is merely a template; individual semiconductor vendors provide detailed memory maps including the actual location and size of ROM, RAM, and peripheral memory locations.

5.3 MEMORY ACCESS ATTRIBUTES

The memory map shows what is included in each memory region. Aside from decoding which memory block or device is accessed, the memory map also defines the memory attributes of the access. The memory attributes you can find in the Cortex-M3 processor include the following:

- *Bufferable*: Write to memory can be carried out by a write buffer while the processor continues on next instruction execution.
- *Cacheable*: Data obtained from memory read can be copied to a memory cache so that next time it is accessed the value can be obtained from the cache to speed up the program execution.
- *Executable*: The processor can fetch and execute program code from this memory region.
- *Sharable*: Data in this memory region could be shared by multiple bus masters. Memory system needs to ensure coherency of data between different bus masters in shareable memory region.

The Cortex-M3 bus interfaces output the memory access attributes information to the memory system for each instruction and data transfer. The default memory attribute settings can be overridden if MPU is present and the MPU region configurations are programmed differently from the default. Though the Cortex-M3 processor does not have a cache memory or cache controller, a cache unit can be added on the microcontroller which can use the memory attribute information to define the memory access behaviors. In addition, the cache attributes might also affect the operation of memory controllers for on-chip memory and off-chip memory, depending on the memory controllers used by the chip manufacturers.

The memory access attributes for each memory region are as follows:

- *Code memory region* (0x00000000–0x1FFFFFFF): This region is executable, and the cache attribute is write through (WT). You can put data memory in this region as well. If data operations are carried out for this region, they will take place via the data bus interface. Write transfers to this region are bufferable.

- *SRAM memory region* (0x20000000–0x3FFFFFFF): This region is intended for on-chip RAM. Write transfers to this region are bufferable, and the cache attribute is write back, write allocated (WB-WA). This region is executable, so you can copy program code here and execute it.
- *Peripheral region* (0x40000000–0x5FFFFFFF): This region is intended for peripherals. The accesses are noncacheable. You cannot execute instruction code in this region (Execute Never, or XN in ARM documentation, such as the Cortex-M3 TRM).
- *External RAM region* (0x60000000–0x7FFFFFFF): This region is intended for either on-chip or off-chip memory. The accesses are cacheable (WB-WA), and you can execute code in this region.
- *External RAM region* (0x80000000–0x9FFFFFFF): This region is intended for either on-chip or off-chip memory. The accesses are cacheable (WT), and you can execute code in this region.
- *External devices* (0xA0000000–0xBFFFFFFF): This region is intended for external devices and/or shared memory that needs ordering/nonbuffered accesses. It is also a nonexecutable region.
- *External devices* (0xC0000000–0xDFFFFFFF): This region is intended for external devices and/or shared memory that needs ordering/nonbuffered accesses. It is also a nonexecutable region.
- *System region* (0xE0000000–0xFFFFFFFF): This region is for private peripherals and vendor-specific devices. It is nonexecutable. For the PPB memory range, the accesses are strongly ordered (noncacheable, nonbufferable). For the vendor-specific memory region, the accesses are bufferable and noncacheable.

Note that from Revision 1 of the Cortex-M3, the code region memory attribute export to external memory system is hardwired to cacheable and nonbufferable. This cannot be overridden by MPU configuration. This update only affects the memory system outside the processor (e.g., level 2 cache and certain types of memory controllers with cache features). Within the processor, the internal write buffer can still be used for write transfers accessing the code region.

5.4 DEFAULT MEMORY ACCESS PERMISSIONS

The Cortex-M3 memory map has a default configuration for memory access permissions. This prevents user programs (non-privileged) from accessing system control memory spaces such as the NVIC. The default memory access permission is used when either no MPU is present or MPU is present but disabled.

If MPU is present and enabled, the access permission in the MPU setup will determine whether user accesses are allowed.

The default memory access permissions are shown in Table 5.1.

Table 5.1 Default Memory Access Permissions

Memory Region	Address	Access in User Program
Vendor specific	0xE0100000–0xFFFFFFFF	Full access
ROM table	0xE00FF000–0xE00FFFFF	Blocked; user access results in bus fault
External PPB	0xE0042000–0xE00FEFFF	Blocked; user access results in bus fault
ETM	0xE0041000–0xE0041FFF	Blocked; user access results in bus fault

Continued

Table 5.1 Default Memory Access Permissions *Continued*

Memory Region	Address	Access in User Program
TPIU	0xE0040000–0xE0040FFF	Blocked; user access results in bus fault
Internal PPB	0xE000F000–0xE003FFFF	Blocked; user access results in bus fault
NVIC	0xE000E000–0xE000EFFF	Blocked; user access results in bus fault, except Software Trigger Interrupt Register that can be programmed to allow user accesses
FPB	0xE0002000–0xE0003FFF	Blocked; user access results in bus fault
DWT	0xE0001000–0xE0001FFF	Blocked; user access results in bus fault
ITM	0xE0000000–0xE0000FFF	Read allowed; write ignored except for stimulus ports with user access enabled
External device	0xA0000000–0xDFFFFFFF	Full access
External RAM	0x60000000–0x9FFFFFFF	Full access
Peripheral	0x40000000–0x5FFFFFFF	Full access
SRAM	0x20000000–0x3FFFFFFF	Full access
Code	0x00000000–0x1FFFFFFF	Full access

When a user access is blocked, the fault exception takes place immediately.

5.5 BIT-BAND OPERATIONS

Bit-band operation support allows a single load/store operation to access (read/write) to a single data bit. In the Cortex-M3, this is supported in two predefined memory regions called bit-band regions. One of them is located in the first 1 MB of the SRAM region, and the other is located in the first 1 MB of the peripheral region. These two memory regions can be accessed like normal memory, but they can also be accessed via a separate memory region called the bit-band alias (see Figure 5.3). When the bit-band alias address is used, each individual bit can be accessed separately in the least significant bit (LSB) of each word-aligned address.

For example, to set bit 2 in word data in address 0x20000000, instead of using three instructions to read the data, set the bit, and then write back the result, this task can be carried out by a single instruction (see Figure 5.4). The assembler sequence for these two cases could be like the one shown in Figure 5.5.

Similarly, bit-band support can simplify application code if we need to read a bit in a memory location. For example, if we need to determine bit 2 of address 0x20000000, we use the steps outlined in Figure 5.6. The assembler sequence for these two cases could be like the one shown in Figure 5.7.

Bit-band operation is not a new idea; in fact, a similar feature has existed for more than 30 years on 8-bit microcontrollers such as the 8051. Although the Cortex-M3 does not have special instructions for bit operation, special memory regions are defined so that data accesses to these regions are automatically converted into bit-band operations.

Note that the Cortex-M3 uses the following terms for the bit-band memory addresses:

- *Bit-band region*: This is a memory address region that supports bit-band operation.
- *Bit-band alias*: Access to the bit-band alias will cause an access (a bit-band operation) to the bit-band region. (*Note:* A memory remapping is performed.)

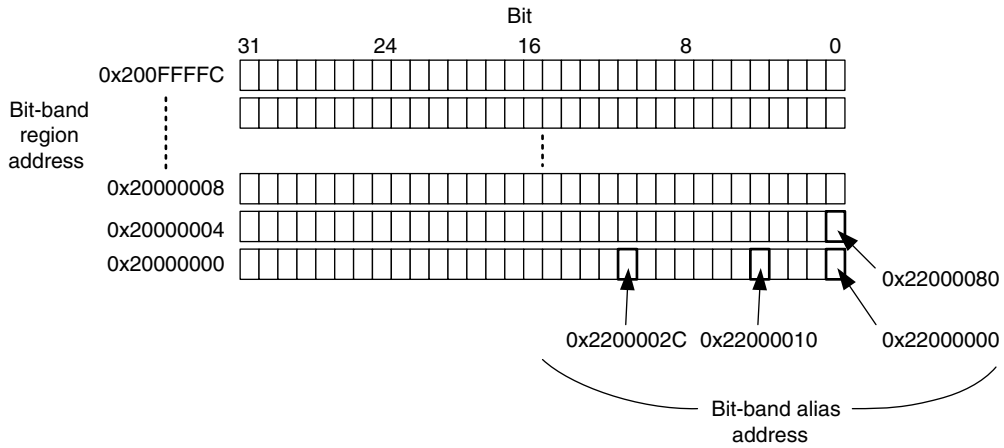

FIGURE 5.3

Bit Accesses to Bit-Band Region via the Bit-Band Alias.

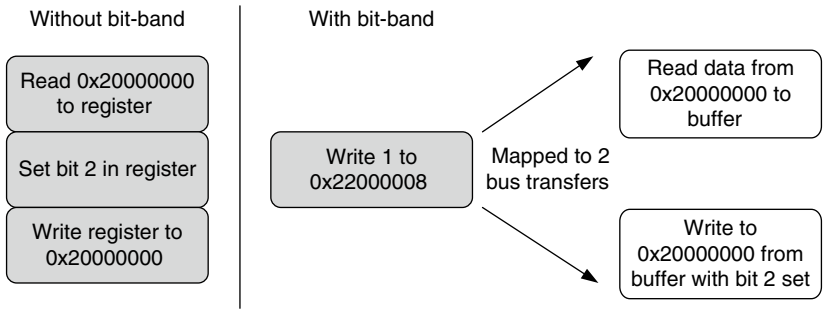

FIGURE 5.4

Write to Bit-Band Alias.

Without bit-band

```
LDR   R0,=0x20000000 ; Setup address
LDR   R1, [R0]       ; Read
ORR.W R1, #0x4       ; Modify bit
STR   R1, [R0]       ; Write back result
```

With bit-band

```
LDR   R0,=0x22000008 ; Setup add
MOV   R1, #1         ; Setup dat
STR   R1, [R0]       ; Write
```

FIGURE 5.5

Example Assembler Sequence to Write a Bit with and without Bit-Band.

Within the bit-band region, each word is represented by an LSB of 32 words in the bit-band alias address range. What actually happens is that when the bit-band alias address is accessed, the address is remapped into a bit-band address. For read operations, the word is read and the chosen bit location is shifted to the LSB of the read return data. For write operations, the written bit data are shifted to the required bit position, and a READ-MODIFY-WRITE is performed.

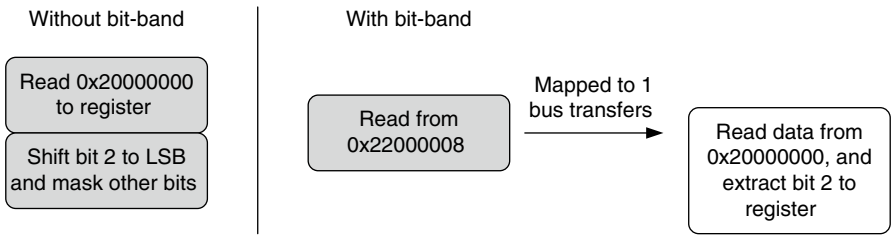

FIGURE 5.6

Read from the Bit-Band Alias.

Without bit-band

```
LDR     R0,=0x20000000 ; Setup address
LDR     R1, [R0]       ; Read
UBFX.W R1,  R1, #2, #1 ; Extract bit[2]
```

With bit-band

```
LDR   R0,=0x22000008 ; Setup address
LDR   R1, [R0]       ; Read
```

FIGURE 5.7

Read from the Bit-Band Alias.

Table 5.2 Remapping of Bit-Band Addresses in SRAM Region

Bit-Band Region	Aliased Equivalent
0x20000000 bit[0]	0x22000000 bit[0]
0x20000000 bit[1]	0x22000004 bit[0]
0x20000000 bit[2]	0x22000008 bit[0]
…	…
0x20000000 bit[31]	0x2200007C bit[0]
0x20000004 bit[0]	0x22000080 bit[0]
…	…
0x20000004 bit[31]	0x220000FC bit[0]
…	…
0x200FFFFC bit[31]	0x23FFFFFC bit[0]

There are two regions of memory for bit-band operations:

- 0x20000000–0x200FFFFF (SRAM, 1 MB)
- 0x40000000–0x400FFFFF (peripherals, 1 MB)

For the SRAM memory region, the remapping of the bit-band alias is shown in Table 5.2.

Similarly, the bit-band region of the peripheral memory region can be accessed via bit-band aliased addresses, as shown in Table 5.3.

Table 5.3 Remapping of Bit-Band Addresses in Peripheral Memory Region

Bit-Band Region	Aliased Equivalent
0x40000000 bit[0]	0x42000000 bit[0]
0x40000000 bit[1]	0x42000004 bit[0]
0x40000000 bit[2]	0x42000008 bit[0]
…	…
0x40000000 bit[31]	0x4200007C bit[0]
0x40000004 bit[0]	0x42000080 bit[0]
…	…
0x40000004 bit[31]	0x420000FC bit[0]
…	…
0x400FFFFC bit[31]	0x43FFFFFC bit[0]

Here's a simple example:

1. Set address 0x20000000 to a value of 0x3355AACC.
2. Read address 0x22000008. This read access is remapped into read access to 0x20000000. The return value is 1 (bit[2] of 0x3355AACC).
3. Write 0x0 to 0x22000008. This write access is remapped into a READ-MODIFY-WRITE to 0x20000000. The value 0x3355AACC is read from memory, bit 2 is cleared, and a result of 0x3355AAC8 is written back to address 0x20000000.
4. Now, read 0x20000000. That gives you a return value of 0x3355AAC8 (bit[2] cleared).

When you access bit-band alias addresses, only the LSB (bit[0]) in the data is used. In addition, accesses to the bit-band alias region should not be unaligned. If an unaligned access is carried out to bit-band alias address range, the result is unpredictable.

5.5.1 Advantages of Bit-Band Operations

So, what are the uses of bit-band operations? We can use them to, for example, implement serial data transfers in general-purpose input/output (GPIO) ports to serial devices. The application code can be implemented easily because access to serial data and clock signals can be separated.

BIT-BAND VERSUS BIT-BANG

In the Cortex-M3, we use the term *bit-band* to indicate that the feature is a special memory band (region) that provides bit accesses. *Bit-bang* commonly refers to driving I/O pins under software control to provide serial communication functions. The bit-band feature in the Cortex-M3 can be used for bit-banging implementations, but the definitions of these two terms are different.

Bit-band operation can also be used to simplify branch decisions. For example, if a branch should be carried out based on 1 single bit in a status register in a peripheral, instead of

- Reading the whole register
- Masking the unwanted bits
- Comparing and branching

you can simplify the operations to

- Reading the status bit via the bit-band alias (get 0 or 1)
- Comparing and branching

Besides providing faster bit operations with fewer instructions, the bit-band feature in the Cortex-M3 is also essential for situations in which resources are being shared by more than one process. One of the most important advantages or properties of a bit-band operation is that it is atomic. In other words, the READ-MODIFY-WRITE sequence cannot be interrupted by other bus activities. Without this behavior in, for example, using a software READ-MODIFY-WRITE sequence, the following problem can occur: consider a simple output port with bit 0 used by a main program and bit 1 used by an interrupt handler. A software-based READ-MODIFY-WRITE operation can cause data conflicts, as shown in Figure 5.8.

With the Cortex-M3 bit-band feature, this kind of race condition can be avoided because the READ-MODIFY-WRITE is carried out at the hardware level and is atomic (the two transfers cannot be pulled apart) and interrupts cannot take place between them (see Figure 5.9).

Similar issues can be found in multitasking systems. For example, if bit 0 of the output port is used by Process A and bit 1 is used by Process B, a data conflict can occur in software-based READ-MODIFY-WRITE (see Figure 5.10).

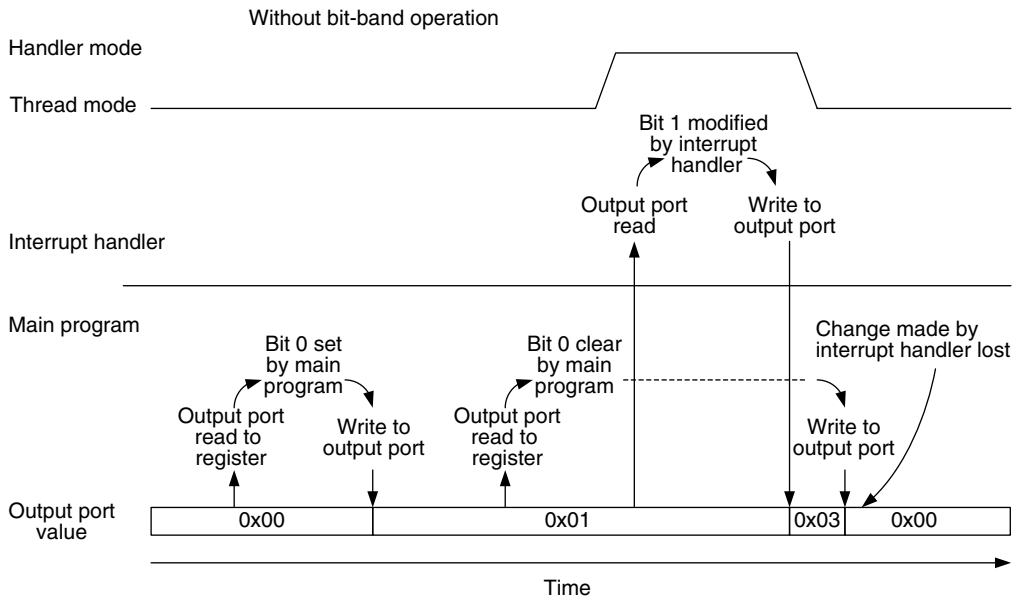

FIGURE 5.8

Data Are Lost When an Exception Handler Modifies a Shared Memory Location.

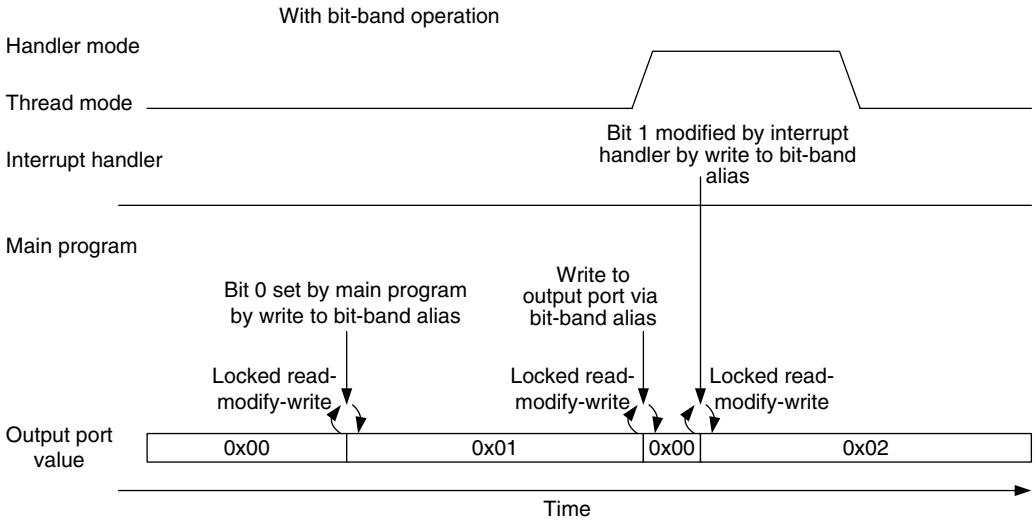

FIGURE 5.9

Data Loss Prevention with Locked Transfers Using the Bit-Band Feature.

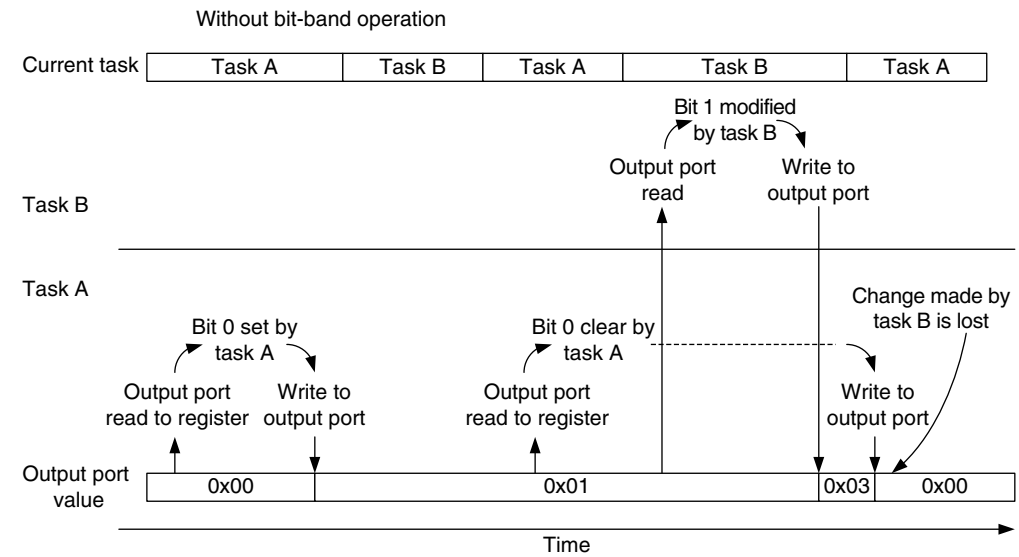

FIGURE 5.10

Data Are Lost When a Different Task Modifies a Shared Memory Location.

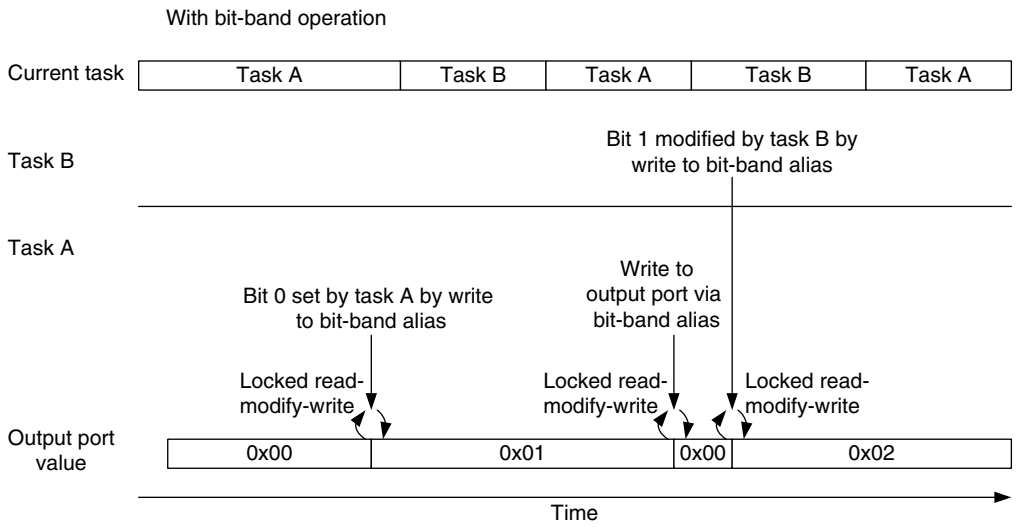

FIGURE 5.11

Data Loss Prevention with Locked Transfers Using the Bit-Band Feature.

Again, the bit-band feature can ensure that bit accesses from each task are separated so that no data conflicts occur (see Figure 5.11).

Besides I/O functions, the bit-band feature can be used for storing and handling Boolean data in the SRAM region. For example, multiple Boolean variables can be packed into one single memory location to save memory space, whereas the access to each bit is still completely separated when the access is carried out via the bit-band alias address range.

For system-on-chip (SoC) designers designing a bit-band-capable device, the device's memory address should be located within the bit-band memory, and the lock (HMASTLOCK) signal from the AHB interface must be checked to make sure that writable register contents will not be changed except by the bus when a locked transfer is carried out.

5.5.2 Bit-Band Operation of Different Data Sizes

Bit-band operation is not limited to word transfers. It can be carried out as byte transfers or half word transfers as well. For example, when a byte access instruction (LDRB/STRB) is used to access a bit-band alias address range, the accesses generated to the bit-band region will be in byte size. The same applies to half word transfers (LDRH/STRH). When you use nonword transfers to bit-band alias addresses, the address value should still be word aligned.

5.5.3 Bit-Band Operations in C Programs

There is no native support of bit-band operation in most C compilers. For example, C compilers do not understand that the same memory can be accessed using two different addresses, and they do not know that accesses to the bit-band alias will only access the LSB of the memory location. To use the

bit-band feature in C, the simplest solution is to separately declare the address and the bit-band alias of a memory location. For example:

```
#define  DEVICE_REG0       *((volatile unsigned long *)  (0x40000000))
#define  DEVICE_REG0_BIT0  *((volatile unsigned long *)  (0x42000000))
#define  DEVICE_REG0_BIT1  *((volatile unsigned long *)  (0x42000004))
...
DEVICE_REG0 = 0xAB; // Accessing the hardware register by normal
                    // address
...
DEVICE_REG0 = DEVICE_REG0 | 0x2; // Setting bit 1 without using
                                 // bitband feature
...
DEVICE_REG0_BIT1 = 0x1; // Setting bit 1 using bitband feature
                        // via the bit band alias address
```

It is also possible to develop C macros to make accessing the bit-band alias easier. For example, we could set up one macro to convert the bit-band address and the bit number into the bit-band alias address and set up another macro to access the memory location by taking the address value as a pointer:

```
// Convert bit band address and bit number into
// bit band alias address
#define BITBAND(addr,bitnum) ((addr & 0xF0000000)+0x2000000+((addr &
   0xFFFFF)<<5)+(bitnum <<2))
// Convert the address as a pointer
#define MEM_ADDR(addr) *((volatile unsigned long *) (addr))
```

Based on the previous example, we rewrite the code as follows:

```
#define DEVICE_REG0 0x40000000
#define BITBAND(addr,bitnum) ((addr & 0xF0000000)+0x02000000+((addr &
   0xFFFFF)<<5)+(bitnum<<2))
#define MEM_ADDR(addr) *((volatile unsigned long *) (addr))
...
MEM_ADDR(DEVICE_REG0) = 0xAB; // Accessing the hardware
                              // register by normal address
...
// Setting bit 1 without using bitband feature
MEM_ADDR(DEVICE_REG0) = MEM_ADDR(DEVICE_REG0) | 0x2;
...
// Setting bit 1 with using bitband feature
MEM_ADDR(BITBAND(DEVICE_REG0,1)) = 0x1;
```

Note that when the bit-band feature is used, the variables being accessed might need to be declared as volatile. The C compilers do not know that the same data could be accessed in two different addresses, so the volatile property is used to ensure that each time a variable is accessed, the memory location is accessed instead of a local copy of the data inside the processor.

Starting from ARM RealView Development Suite version 4.0 and Keil MDK-ARM 3.80, bit band support is provided by `__attribute__((bitband))` language extension and `__bitband` command line option (see reference 6). You can find further examples of bit-band accesses with C macros using ARM RealView Compiler Tools in the *ARM Application Note 179* [Ref. 7].

5.6 UNALIGNED TRANSFERS

The Cortex-M3 supports unaligned transfers on single accesses. Data memory accesses can be defined as aligned or unaligned. Traditionally, ARM processors (such as the ARM7/ARM9/ARM10) allow only aligned transfers. That means in accessing memory, a word transfer must have address bit[1] and bit[0] equal to 0, and a half word transfer must have address bit[0] equal to 0. For example, word data can be located at 0x1000 or 0x1004, but it cannot be located in 0x1001, 0x1002, or 0x1003. For half word data, the address can be 0x1000 or 0x1002, but it cannot be 0x1001.

So, what does an unaligned transfer look like? Figures 5.12 through 5.16 show some examples. Assuming that the memory infrastructure is 32-bit (4 bytes) wide, an unaligned transfer can be any word size read/write such that the address is not a multiple of 4, as shown in Figures 5.12–5.14, or when the transfer is in half word size, and the address is not a multiple of 2, as shown in Figures 5.15 and 5.16.

All the byte-size transfers are aligned on the Cortex-M3 because the minimum address step is 1 byte.

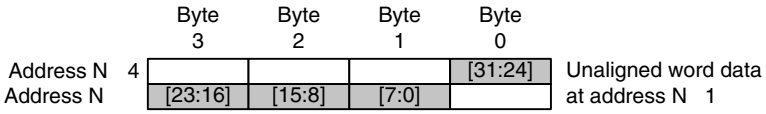

FIGURE 5.12

Unaligned Transfer Example 1.

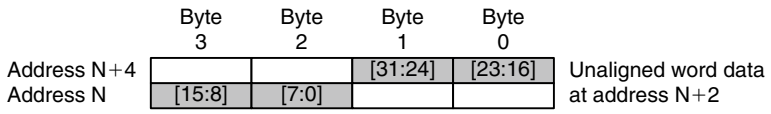

FIGURE 5.13

Unaligned Transfer Example 2.

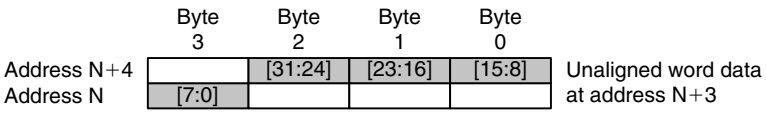

FIGURE 5.14

Unaligned Transfer Example 3.

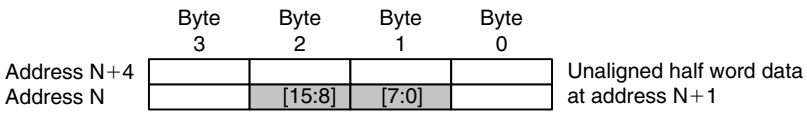

FIGURE 5.15

Unaligned Transfer Example 4.

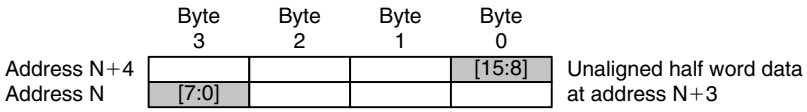

FIGURE 5.16

Unaligned Transfer Example 5.

In the Cortex-M3, unaligned transfers are supported in normal memory accesses (such as LDR, LDRH, STR, and STRH instructions). There are a number of limitations:

- Unaligned transfers are not supported in Load/Store multiple instructions.
- Stack operations (PUSH/POP) must be aligned.
- Exclusive accesses (such as LDREX or STREX) must be aligned; otherwise, a fault exception (usage fault) will be triggered.
- Unaligned transfers are not supported in bit-band operations. Results will be unpredictable if you attempt to do so.

When unaligned transfers are used, they are actually converted into multiple aligned transfers by the processor's bus interface unit. This conversion is transparent, so application programmers do not have to worry about it. However, when an unaligned transfer takes place, it is broken into separate transfers, and as a result, it takes more clock cycles for a single data access and might not be good for situations in which high performance is required. To get the best performance, it's worth making sure that data are aligned properly.

It is also possible to set up the NVIC so that an exception is triggered when an unaligned transfer takes place. This is done by setting the UNALIGN_TRP (unaligned trap) bit in the configuration control register in the NVIC (0xE000ED14). In this way, the Cortex-M3 generates usage fault exceptions when unaligned transfers take place. This is useful during software development to test whether an application produces unaligned transfers.

5.7 EXCLUSIVE ACCESSES

You might have noticed that the Cortex-M3 has no SWP instruction (swap), which was used for semaphore operations in traditional ARM processors like ARM7TDMI. This is now being replaced by exclusive access operations. Exclusive accesses were first supported in architecture v6 (for example, in the ARM1136).

Semaphores are commonly used for allocating shared resources to applications. When a shared resource can only service one client or application processor, we also call it Mutual Exclusion (MUTEX). In such cases, when a resource is being used by one process, it is locked to that process and cannot serve another process until the lock is released. To set up a MUTEX semaphore, a memory location is defined as the lock flag to indicate whether a shared resource is locked by a process. When a process or application wants to use the resource, it needs to check whether the resource has been locked first. If it is not being used, it can set the lock flag to indicate that the resource is now locked. In traditional ARM processors, the access to the lock flag is carried out by the SWP instruction. It allows

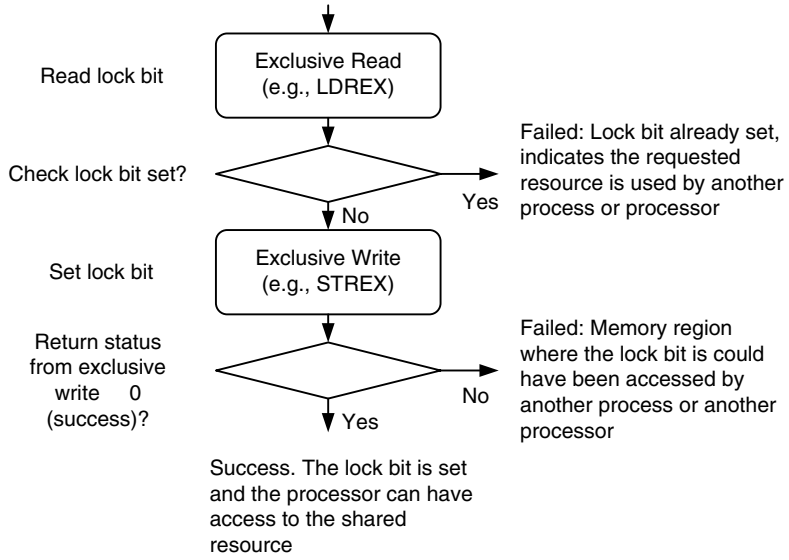

FIGURE 5.17

Using Exclusive Access in MUTEX Semaphores.

the lock flag read and write to be atomic, preventing the resource from being locked by two processes at the same time.

In newer ARM processors, the read/write access can be carried out on separated buses. In such situations, the SWP instructions can no longer be used to make the memory access atomic because the read and write in a locked transfer sequence must be on the same bus. Therefore, the locked transfers are replaced by exclusive accesses. The concept of exclusive access operation is quite simple but different from SWP; it allows the possibility that the memory location for a semaphore could be accessed by another bus master or another process running on the same processor (see Figure 5.17).

To allow exclusive access to work properly in a multiple processor environment, an additional hardware called "exclusive access monitor" is required. This monitor checks the transfers toward shared address locations and replies to the processor if an exclusive access is success. The processor bus interface also provides additional control signals[1] to this monitor to indicate if the transfer is an exclusive access.

If the memory device has been accessed by another bus master between the exclusive read and the exclusive write, the exclusive access monitor will flag an exclusive failed through the bus system when the processor attempts the exclusive write. This will cause the return status of the exclusive write to be 1. In the case of failed exclusive write, the exclusive access monitor also blocks the write transfer from getting to the exclusive access address.

[1]Exclusive access signals are available on the system bus and the D-Code bus of the Cortex-M3 processor. They are EXREQD and EXRESPD for the D-Code bus and EXREQS and EXRESPS for the system bus. The I-Code bus that is used for instruction fetch cannot generate exclusive accesses.

Exclusive access instructions in the Cortex-M3 include LDREX (word), LDREXB (byte), LDREXH (half word), STREX (word), STREXB (byte), and STREXH (half word). A simple example of the syntax is as follows:

```
LDREX  <Rxf>, [Rn, #offset]
STREX  <Rd>, <Rxf>,[Rn, #offset]
```

Where *Rd* is the return status of the exclusive write (0 = success and 1 = failure).

Example code for exclusive accesses can be found in Chapter 10. You can also access exclusive access instructions in C using intrinsic functions provided in Cortex Microcontroller Software Interface Standard (CMSIS) compliant device driver libraries from microcontroller vendors: `__LDREX`, `__LEDEXH`, `__LDREXB`, `__STREX`, `__STREXH`, `__STREXB`. More details of these functions are covered in Appendix G.

When exclusive accesses are used, the internal write buffers in the Cortex-M3 bus interface will be bypassed, even when the MPU defines the region as bufferable. This ensures that semaphore information on the physical memory is always up to date and coherent between bus masters. SoC designers using Cortex-M3 on multiprocessor systems should ensure that the memory system enforces data coherency when exclusive transfers occur.

5.8 ENDIAN MODE

The Cortex-M3 supports both little endian and big endian modes. However, the supported memory type also depends on the design of the rest of the microcontroller (bus connections, memory controllers, peripherals, and so on). Make sure that you check your microcontroller datasheets in detail before developing your software. In most cases, Cortex-M3-based microcontrollers will be little endian. With little endian mode, the first byte of a word size data is stored in the least significant byte of the 32-bit memory location (see Table 5.4).

There are some microcontrollers that use big endian mode. In such a case, the first byte of a word size data is stored in the most significant byte of the 32-bit address memory location (see Table 5.5).

The definition of big endian in the Cortex-M3 is different from the ARM7. In the ARM7TDMI, the big endian scheme is called *word-invariant big endian*, also referred as BE-32 in ARM documentation, whereas in the Cortex-M3, the big endian scheme is called *byte-invariant big endian*, also referred as BE-8 (byte-invariant big endian is supported on ARM architecture v6 and v7). The memory view of both schemes is the same, but the byte lane usage on the bus interface during data transfers is different (see Tables 5.6 and 5.7).

Note that the data transfer on the AHB bus in BE-8 mode uses the same data byte lanes as in little endian. However, the data byte inside the half word or word data is reversely ordered compared to little endian (see Table 5.8).

Table 5.4 The Cortex-M3 Little Endian Memory View Example

Address	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x1003 – 0x1000	Byte – 0x1003	Byte – 0x1002	Byte – 0x1001	Byte – 0x1000
0x1007 – 0x1004	Byte – 0x1007	Byte – 0x1006	Byte – 0x1005	Byte – 0x1004
···	Byte – 4xN+3	Byte – 4xN+2	Byte – 4xN+1	Byte – 4xN

Table 5.5 The Cortex-M3 Big Endian Memory View Example

Address	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x1003 – 0x1000	Byte – 0x1000	Byte – 0x1001	Byte – 0x1002	Byte – 0x1003
0x1007 – 0x1004	Byte – 0x1004	Byte – 0x1005	Byte – 0x1006	Byte – 0x1007
…	Byte – 4xN	Byte – 4xN+1	Byte – 4xN+2	Byte – 4xN+3

Table 5.6 The Cortex-M3 (Byte-Invariant Big Endian, BE-8)—Data on the AHB Bus

Address, Size	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x1000, word	Data bit [7:0]	Data bit [15:8]	Data bit [23:16]	Data bit [31:24]
0x1000, half word	—	—	Data bit [7:0]	Data bit [15:8]
0x1002, half word	Data bit [7:0]	Data bit [15:8]	—	—
0x1000, byte	—	—	—	Data bit [7:0]
0x1001, byte	—	—	Data bit [7:0]	—
0x1002, byte	—	Data bit [7:0]	—	—
0x1003, byte	Data bit [7:0]	—	—	—

Table 5.7 ARM7TDMI (Word-Invariant Big Endian, BE-32)—Data on the AHB Bus

Address, Size	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x1000, word	Data bit [7:0]	Data bit [15:8]	Data bit [23:16]	Data bit [31:24]
0x1000, half word	Data bit [7:0]	Data bit [15:8]	—	—
0x1002, half word	—	—	Data bit [7:0]	Data bit [15:8]
0x1000, byte	Data bit [7:0]	—	—	—
0x1001, byte	—	Data bit [7:0]	—	—
0x1002, byte	—	—	Data bit [7:0]	—
0x1003, byte	—	—	—	Data bit [7:0]

Table 5.8 The Cortex-M3 Little Endian—Data on the AHB Bus

Address, Size	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x1000, word	Data bit [31:24]	Data bit [23:16]	Data bit [15:8]	Data bit [7:0]
0x1000, half word	—	—	Data bit [15:8]	Data bit [7:0]
0x1002, half word	Data bit [15:8]	Data bit [7:0]	—	—
0x1000, byte	—	—	—	Data bit [7:0]
0x1001, byte	—	—	Data bit [7:0]	—
0x1002, byte	—	Data bit [7:0]	—	—
0x1003, byte	Data bit [7:0]	—	—	—

In the Cortex-M3 processor, the endian mode is set when the processor exits reset. The endian mode cannot be changed afterward. (There is no dynamic endian switching, and the SETEND instruction is not supported.) Instruction fetches are always in little endian as are data accesses in the system control memory space (such as NVIC and FPB) and the external PPB memory range (memory range from 0xE0000000 to 0xE00FFFFF is always little endian).

In case your SoC does not support big endian but one or some of the peripherals you are using contain big endian data, you can easily convert the data between little endian and big endian using some of the data type conversion instructions in the Cortex-M3. For example, REV and REV16 are very useful for this kind of conversion.

This page intentionally left blank

CHAPTER

Cortex-M3 Programming 10

IN THIS CHAPTER

10.1 OVERVIEW

The Cortex™-M3 can be programmed using either assembly language, C language, or other high-level languages like National Instruments LabVIEW. For most embedded applications using the Cortex-M3 processor, the software can be written entirely in C language. There are of course some people who prefer to use assembly language or a combination of C and assembly language in their projects. The procedure of building and downloading the resultant image files to the target device is largely dependent on the tool chain used. Although this is not the main focus of this book, some simple examples showing how to use the Gnu's Not Unix (GNU) and Keil tool chains are provided in Chapters 19 and 20, and an introduction of using LabVIEW on Cortex-M3 is covered in Chapter 21.

10.2 A TYPICAL DEVELOPMENT FLOW

Various software programs are available for developing Cortex-M3 applications. The concepts of code generation flow in terms of these tools are similar. For the most basic uses, you will need assembler, a C compiler, a linker, and binary file generation utilities. For ARM solutions, the RealView Development Suite (RVDS) or RealView Compiler Tools (RVCT) provide a file generation flow, as shown in

DOI: 10.1016/B978-0-12-382091-4.00016-5

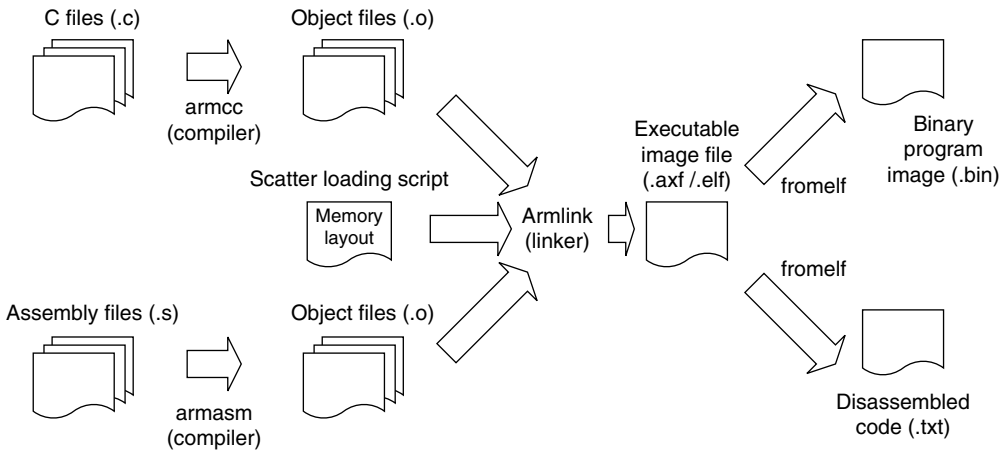

FIGURE 10.1

Example Flow Using ARM Development Tools.

Figure 10.1. The scatter-loading script is optional but often required when the memory map becomes more complex.

Besides these basic tools, RVDS also contains a large number of utilities, including an Integrated Development Environment (IDE) and debuggers. Please visit the ARM web site (*www.arm.com*) for details.

10.3 USING C

For beginners in embedded programming, using C language for software development on the Cortex-M3 processor is the best choice. Programming in C with the Cortex-M3 processor is made even easier as most microcontroller vendors provide device driver libraries written in C to control peripherals. These can then be included into your project. Since modern C compilers can generate very efficient code, it is better to program in C than spending a lot of time to try to develop complex routines in assembly language, which is error prone and less portable.

In this chapter, we will have a quick look at a simple example of using C language to create a simple program image. Then, we will have a look at some C language development areas including using device driver libraries and the Cortex Microcontroller Software Interface Standard (CMSIS).

C has the advantage of being portable and easier for implementing complex operations, compared with assembly language. Since it's a generic computer language, C does not specify how the processor is initialized. For these areas, tool chains can have different approaches. The best way to get started is to look at example codes. For users of ARM C compiler products, such as RVDS or Keil RealView Microcontroller Development Kit (MDK-ARM), a number of Cortex-M3 program examples are already included in the installation. For users of the GNU tool chain, Chapter 19 provides a simple C example based on the CodeSourcery GNU tool chain for ARM.

10.3.1 Example of a Simple C Program Using RealView Development Site

A normal program for the Cortex-M3 contains at least the "main" program and a vector table. Let's start with the most basic main program that toggles an Light Emitting Diode (LED):

```
#define LED *((volatile unsigned int *)(0xDFFF000C))

int main (void)
{
int i;          /* loop counter for delay function */
volatile int j; /* dummy volatile variable to prevent
          C compiler from optimize the delay away */

  while (1) {
    LED = 0x00; /* toogle LED */
    for (i=0;i<10;i++) {j=0;} /* delay */
    LED = 0x01; /* toogle LED */
    for (i=0;i<10;i++) {j=0;} /* delay */
    }
  return 0;
}
```

This file is named "blinky.c." For the vector table, we create a separate C program called "vectors.c." The file "vectors.c" contains the vector table, as well as a number of dummy exception handlers (these can be customized for target application later on):

```
typedef void(* const ExecFuncPtr)(void) __irq;
extern int __main(void);

/*
 * Dummy handlers Exception Handlers
 */
__irq void NMI_Handler(void)
{   while(1); }
__irq void HardFault_Handler(void)
{   while(1); }
__irq void SVC_Handler(void)
{   while(1); }
__irq void DebugMon_Handler(void)
{   while(1); }
__irq void PendSV_Handler(void)
{   while(1); }
__irq void SysTick_Handler(void)
{   while(1); }
__irq void ExtInt0_IRQHandler(void)
{   while(1); }
__irq void ExtInt1_IRQHandler(void)
{   while(1); }
__irq void ExtInt2_IRQHandler(void)
{   while(1); }
__irq void ExtInt3_IRQHandler(void)
{   while(1); }

#pragma arm section rodata="exceptions_area"
```

```
ExecFuncPtr exception_table[] = { /* vector table */
    (ExecFuncPtr)0x20002000,
    (ExecFuncPtr)__main,
    NMI_Handler, /* NMI */
    HardFault_Handler,
    0, /* MemManage_Handler in Cortex-M3 */
    0, /* BusFault_Handler in Cortex-M3 */
    0, /* UsageFault_Handler in Cortex-M3 */

    0, /* Reserved */
    0, /* Reserved */
    0, /* Reserved */
    0, /* Reserved */
    SVC_Handler,
    0, /* DebugMon_Handler in Cortex-M3 */
    0, /* Reserved */
    PendSV_Handler,
    SysTick_Handler,

    /* External Interrupts*/
    ExtInt0_IRQHandler,
    ExtInt1_IRQHandler,
    ExtInt2_IRQHandler,
    ExtInt3_IRQHandler
};
#pragma arm section
```

Assuming you are using RVDS, you can compile the program using the following command line:

```
$> armcc -c -g -W blinky.c -o blinky.o
$> armcc -c -g -W vectors.c -o vectors.o
```

Then the linker can be used to generate the program image. A scatter loading file "led.scat" is used to tell the linker the memory layout and to put the vector table in the starting of the program image. The "led.scat" is

```
#define HEAP_BASE 0x20001000
#define STACK_BASE 0x20002000
#define HEAP_SIZE ((STACK_BASE-HEAP_BASE)/2)
#define STACK_SIZE ((STACK_BASE-HEAP_BASE)/2)

LOAD_REGION 0x00000000 0x00200000
{
 VECTORS 0x0 0xC0
 {
  ; Provided by the user in vectors.c
  * (exceptions_area)
 }

 CODE 0xC0 FIXED
 {
  * (+RO)
 }

 DATA 0x20000000 0x00010000
 {
  * (+RW, +ZI)
 }
```

```
    ;; Heap starts at 4KB and grows upwards
    ARM_LIB_HEAP HEAP_BASE EMPTY HEAP_SIZE
    {
    }

    ;; Stack starts at the end of the 8KB of RAM
    ;; And grows downwards for 2KB
    ARM_LIB_STACK STACK_BASE EMPTY -STACK_SIZE
    {
    }
}
```

And the command line for the linker is

```
$> armlink -scatter led.scat "--keep=vectors.o(exceptions_area)"
  blinky.o vectors.o -o blinky.elf
```

The executable image `blinky.elf` is now generated. We can convert it to binary file and disassembly file using fromelf.

```
/* create binary file */
$> fromelf --bin blinky.elf -output blinky.bin
/* Create disassembly output */
$> fromelf -c blinky.elf > list.txt
```

Previously in ARM processors, because there is a Thumb® state and an ARM state, the code for different states has to be compiled differently. In the Cortex-M3, there is no such need because everything is in the Thumb state, and project file management is much simpler.

When you're developing applications in C, it is recommended that you use the double word stack alignment function (configured by the STKALIGN bit in the Nested Vectored Interrupt Controller [NVIC] Configuration Control register). For users of Cortex-M3 revision 2 or future products, the STKALIGN bit is set by default at reset so there is no need to set up this bit in the software. Users of Cortex-M3 revision 1 can enable this feature by setting this bit in the beginning of their applications, for example. The details of STKALIGN feature are covered in Chapter 9.

```
SCB->CCR = SCB->CCR | 0x200; /* Set STKALIGN */
/* SCB->CCR is defined in device driver library. */
```

If you are not using a CMSIS compliant device driver, you can use the following code instead.

```
#define NVIC_CCR *((volatile unsigned long *)(0xE000ED14))
NVIC_CCR = NVIC_CCR | 0x200; /* Set STKALIGN */
```

Using this feature ensures that the system conforms to Procedure Call Standards for the ARM Architecture (AAPCS). Additional information on this subject is covered in Chapter 12.

10.3.2 Compile the Same Example Using Keil MDK-ARM

For users of Keil MDK-ARM, it is possible to compile the same program as in RVDS. However, the command line options and a few symbols in the linker script (scatter loading file) have to be modified. Based on the example in Section 10.3.1, scatter loading file "led.scat" needed to be modified to

```
#define HEAP_BASE 0x20001000
#define STACK_BASE 0x20002000
#define HEAP_SIZE ((STACK_BASE-HEAP_BASE)/2)
#define STACK_SIZE ((STACK_BASE-HEAP_BASE)/2)

LOAD_REGION 0x00000000 0x00200000
{
 VECTORS 0x0 0xC0
 {
   ; Provided by the user in vectors.c
   * (exceptions_area)
 }

 CODE 0xC0 FIXED
 {
   * (+RO)
 }

 DATA 0x20000000 0x00010000
 {
   * (+RW, +ZI)
 }

 ;; Heap starts at 4KB and grows upwards
 Heap_Mem HEAP_BASE EMPTY HEAP_SIZE
 {
 }

 ;; Stack starts at the end of the 8KB of RAM
 ;; And grows downwards for 2KB
 Stack_Mem STACK_BASE EMPTY -STACK_SIZE
 {
 }
}
```

And the compile sequence can be created in a DOS batch file

```
SET PATH=C:\Keil\ARM\BIN40\;%PATH%
SET RVCT40INC=C:\Keil\ARM\RV31\INC
SET RVCT40LIB=C:\Keil\ARM\RV31\LIB
SET CPU_TYPE=Cortex-M3
SET CPU_VENDOR=ARM
SET UV2_TARGET=Target 1
SET CPU_CLOCK=0x00000000
C:\Keil\ARM\BIN40\armcc -c -O3 -W -g -Otime --device DLM vectors.c
C:\Keil\ARM\BIN40\armcc -c -O3 -W -g -Otime --device DLM blinky.c
C:\Keil\ARM\BIN40\armlink --device DLM "--keep=Startup.o(RESET)"
  "--first=Startup.o(RESET)" -scatter led.scat --map vectors.o
  blinky.o -o blinky.elf
C:\Keil\ARM\BIN40\fromelf --bin blinky.elf -o blinky.bin
```

In general, it is much easier to use the µVision IDE to create and compile projects rather than using command lines. Chapter 20 is ideal for beginners who want to start using the Cortex-M3 microcontrollers with the Keil Microcontroller Development Kit for ARM (MDK-ARM).

10.3.3 Accessing Memory-Mapped Registers in C

There are various ways to access memory-mapped peripheral registers in C language. For illustration, we will use the System Tick (SYSTICK) Timer in the Cortex-M3 as an example peripheral to demonstrate different access methods in C language. The SYSTICK is a 24-bit timer which contains only four registers. The functionality of the SYSTICK will be covered in Chapter 14. In the previous examples, we have already illustrated the easiest method—defining each register as a pointer. To apply the same solution to the SYSTICK, we can define each register separately. This is illustrated in Figure 10.2.

Based on the same method, we can define a macro to convert address values to C pointer. The C-code looks a bit different, but the generated code is the same as previous implementation. This is illustrated in Figure 10.3.

Method 2 is to define the registers as a data structure, and then define a pointer of the defined structure. This is the method used in CMSIS compliant device driver libraries. This is illustrated in Figure 10.4.

Method 3 also uses data structure, but the base address of the peripheral is defined using a scatter loading file (or linker script) during linking stage. This is illustrated in Figure 10.5.

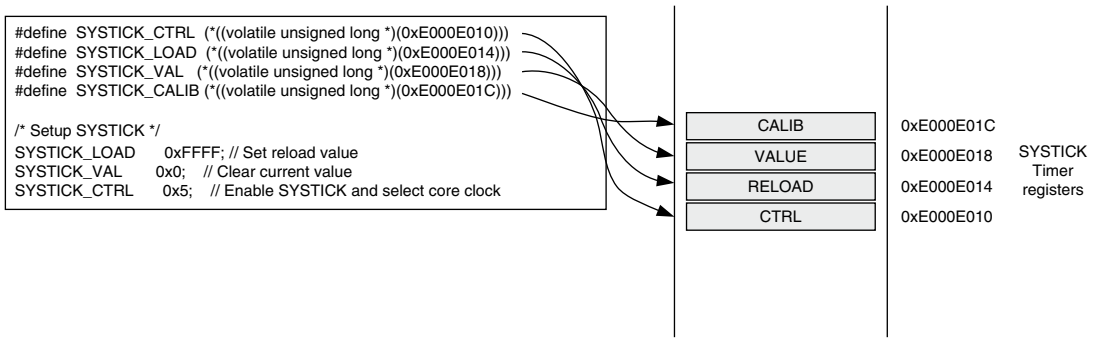

FIGURE 10.2

Accessing Peripheral Registers as Pointers.

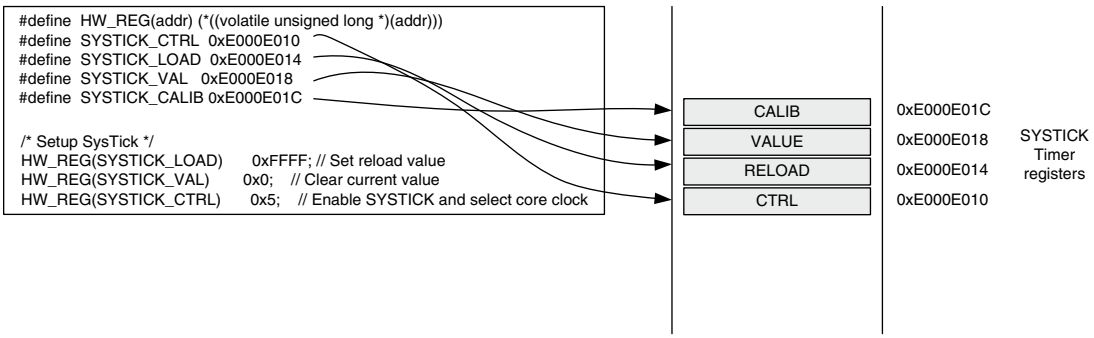

FIGURE 10.3

Alternative Way of Accessing Peripheral Registers as Pointers.

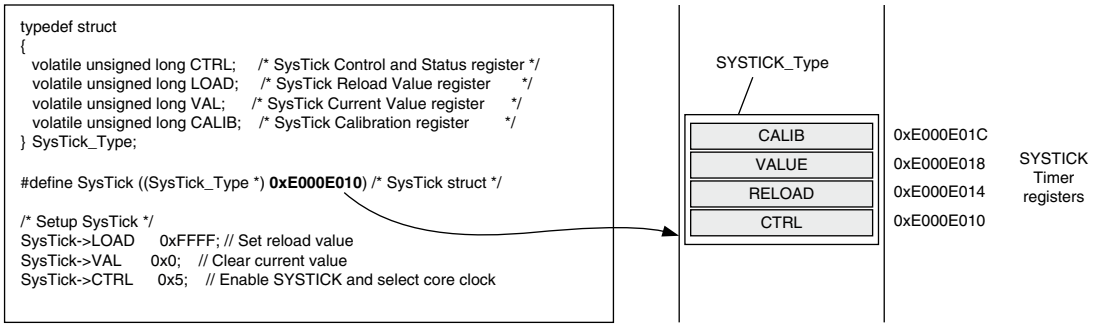

FIGURE 10.4

Accessing Peripheral Registers as Pointers to Elements in a Data Structure.

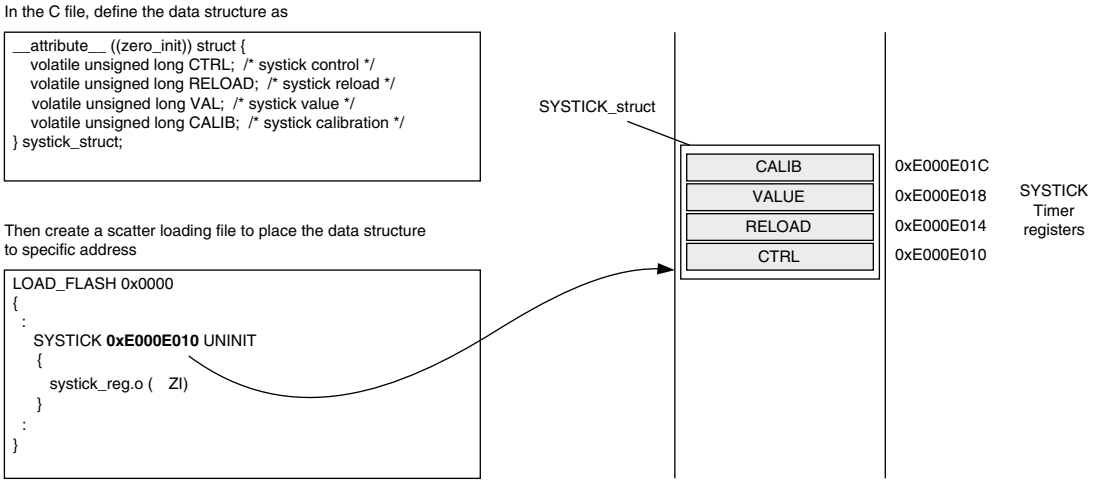

FIGURE 10.5

Defining Peripheral-Based Address Using Scatter Loading File.

In this case (method is shown in Figure 10.5), the program code using the peripheral has to define the peripheral as a C pointer in an external object. The code for accessing the register is the same as in the second method.

Method 1 (shown in Figures 10.2 and 10.3) is the simplest, however, it can result in less efficient code compared with the others as the address value for the registers are stored separately as constant. As a result, the code size can be larger and might be slower as it requires more accesses to the program memory to set up the address values. However, for peripheral control code that only access to one register, the efficiency of method 1 is identical to others.

Method 2 (using data structure and a pointer defined in the C-code) is possibly the most commonly used. It allows the registers in a peripheral to share just one constant for base address value. The immediate offset address mode can be used for access of each register. This is the method used in CMSIS, which will be covered later in this chapter.

Method 3 (using scatter loading file or linker script, as shown in figure 10.5) has the same efficiency as method 2, but it is less portable due to the use of a scatter loading file (scatter loading file syntax is tool chain specific). Method 3 is required when you are developing a device driver library for a peripheral that is used in multiple devices, and the base address of the peripheral is not known until in the linking stage.

10.3.4 Intrinsic Functions

Use of the C language can often speed up application development, but in some cases, we need to use some instructions that cannot be generated using normal C-code. Some C compilers provide intrinsic functions for accessing these special instructions. Intrinsic functions are used just like normal C functions. For example, ARM compilers (including RealView C Compilers and Keil MDK-ARM) provide the intrinsic functions listed in Table 10.1 for commonly used instructions.

10.3.5 Embedded Assembler and Inline Assembler

As an alternative to using intrinsic functions, we can also directly access assembly instructions in C-code. This is often necessary in low-level system control or when you need to implement a timing critical routine and decide to implement it in assembly for the best performance. Most ARM C compilers allow you to include assembly code in form of *inline assembler*.

Table 10.1 Intrinsic Functions Provided in ARM Compilers

Assembly Instructions	ARM Compiler Intrinsic Functions
CLZ	unsigned char __clz(unsigned int val)
CLREX	void __clrex(void)
CPSID I	void __disable_irq(void)
CPSIE I	void __enable_irq(void)
CPSID F	void __disable_fiq(void)
CPSIE F	void __enable_fiq(void)
LDREX/LDREXB/LDREXH	unsigned int __ldrex(volatile void *ptr)
LDRT/LDRBT/LDRSBT/LDRHT/LDRSHT	unsigned int __ldrt(const volatile void *ptr)
NOP	void __nop(void)
RBIT	unsigned int __rbit(unsigned int val)
REV	unsigned int __rev(unsigned int val)
ROR	unsigned int __ror(unsigned int val, unsigned int shift)
SSAT	int __ssat(int val, unsigned int sat)
SEV	void __sev(void)
STREX/STREXB/STREXH	int __strex(unsigned int val, volatile void *ptr)
STRT/STRBT/STRHT	void int __strt(unsigned int val, const volatile void *ptr)
USAT	int __usat(unsigned int val, unsigned int sat)
WFE	void __wfe(void)
WFI	void __wfi(void)
BKPT	void __breakpoint(int val)

In the ARM compiler, you can add assembly code inside the C program. Traditionally, inline assembler is used, but the inline assembler in RealView C Compiler does not support instructions in Thumb-2 technology. Starting with RealView C Compiler version 3.0, a new feature called the Embedded Assembler is included, and it supports the instruction set in Thumb-2. For example, you can insert assembly functions in your C programs this way:

```
__asm void SetFaultMask(unsigned int new_value)
{
  // Assembly code here
  MSR FAULTMASK, new_value // Write new value to FAULTMASK
  BX LR                    // Return to calling program
}
```

Detailed descriptions of Embedded Assembler in RealView C Compiler can be found in the *RVCT 4.0 Compilation Tools Compiler Guide* [Ref. 6].

For the Cortex-M3, Embedded Assembler is useful for tasks, such as direct manipulation of the stacks and timing critical processing task (codec software).

10.4 CMSIS

10.4.1 Background of CMSIS

The Cortex-M3 microcontrollers are gaining momentum in the embedded application market, as more and more products based on the Cortex-M3 processor and software that support the Cortex-M3 processor are emerging. At the end of 2008, there were more than five C compiler vendors, and more than 15 embedded Operating Systems (OS) supporting the Cortex-M3 processor. There are also a number of companies providing embedded software solutions, including codecs, data processing libraries, and various software and debug solutions. The CMSIS was developed by ARM to allow users of the Cortex-M3 microcontrollers to gain the most benefit from all these software solutions and to allow them to develop their embedded application quickly and reliably (see Figure 10.6).

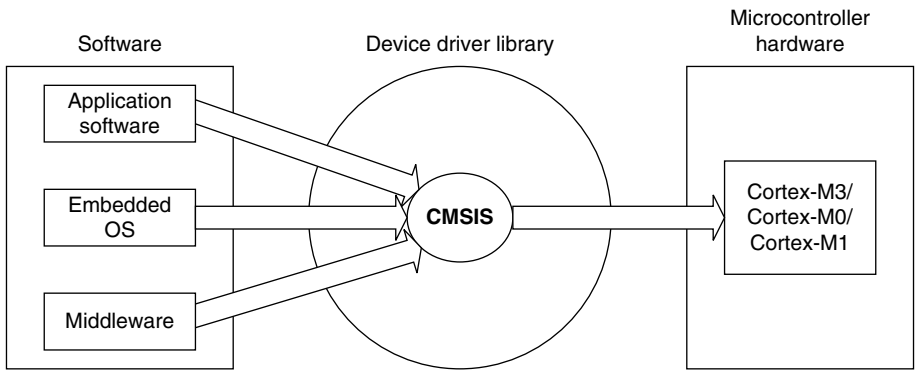

FIGURE 10.6

CMSIS Provides a Standardized Access Interface for Embedded Software Products.

The CMSIS was started in 2008 to improve software usability and inter-operability of ARM microcontroller software. It is integrated into the driver libraries provided by silicon vendors, providing a standardized software interface for the Cortex-M3 processor features, as well as a number of common system and I/O functions. The library is also supported by software companies including embedded OS vendors and compiler vendors.

The aims of CMSIS are to:

- improve software portability and reusability
- enable software solution suppliers to develop products that can work seamlessly with device libraries from various silicon vendors
- allow embedded developers to develop software quicker with an easy-to-use and standardized software interface
- allow embedded software to be used on multiple compiler products
- avoid device driver compatibility issues when using software solutions from multiple sources

The first release of CMSIS was available from fourth quarter of 2008 and has already become part of the device driver library from microcontroller vendors. The CMSIS is also available for Cortex-M0.

10.4.2 Areas of Standardization

The scope of CMSIS involves standardization in the following areas:

- *Hardware Abstraction Layer (HAL) for Cortex-M processor registers*: This includes standardized register definitions for NVIC, System Control Block registers, SYSTICK register, MPU registers, and a number of NVIC and core feature access functions.
- *Standardized system exception names*: This allows OS and middleware to use system exceptions easily without compatibility issues.
- *Standardized method of header file organization*: This makes it easier for users to learn new Cortex microcontroller products and improve software portability.
- *Common method for system initialization*: Each Microcontroller Unit (MCU) vendor provides a *SystemInit()* function in their device driver library for essential setup and configuration, such as initialization of clocks. Again, this helps new users to start to use Cortex-M microcontrollers and aids software portability.
- *Standardized intrinsic functions*: Intrinsic functions are normally used to produce instructions that cannot be generated by IEC/ISO C.* By having standardized intrinsic functions, software reusability and portability are considerably improved.
- *Common access functions for communication*: This provides a set of software interface functions for common communication interfaces including universal asynchronous receiver/transmitter (UART), Ethernet, and Serial Peripheral Interface (SPI). By having these common access functions in the device driver library, reusability and portability of embedded software are improved. At the time of writing this book, it is still under development.
- *Standardized way for embedded software to determine system clock frequency*: A software variable called *SystemFrequency* is defined in device driver code. This allows embedded OS to set up the SYSTICK unit based on the system clock frequency.

*C/C++ features are specified in a standard document "ISO/IEC 14882" prepared by the International Organization for Standards (ISO) and the International Electrotechnical Commission (IEC).

The CMSIS defines the basic requirements to achieve software reusability and portability. MCU vendors can include additional functions for each peripheral to enrich the features of their software solution. So using CMSIS does not limit the capability of the embedded products.

10.4.3 Organization of CMSIS

The CMSIS is divided into multiple layers as follows:

Core Peripheral Access Layer
- Name definitions, address definitions, and helper functions to access core registers and core peripherals

Middleware Access Layer
- Common method to access peripherals for the software industry (work in progress)
- Targeted communication interfaces include Ethernet, UART, and SPI.
- Allows portable software to perform communication tasks on any Cortex microcontrollers that support the required communication interface

Device Peripheral Access Layer (MCU specific)
- Name definitions, address definitions, and driver code to access peripherals

Access Functions for Peripherals (MCU specific)
- Optional additional helper functions for peripherals

The role of these layers is summarized in Figure 10.7.

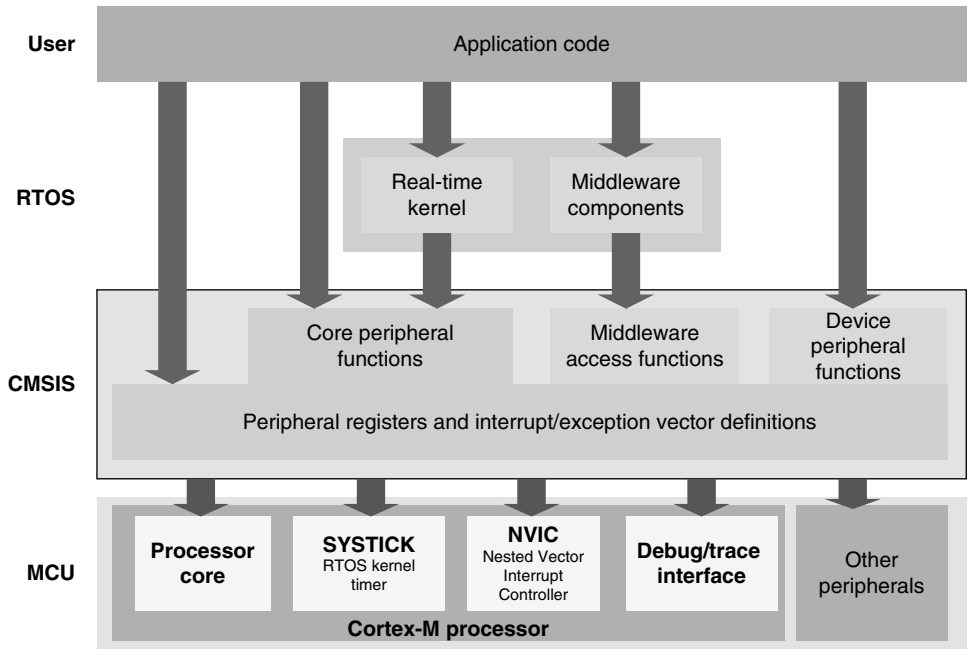

FIGURE 10.7

CMSIS Structure.

10.4.4 Using CMSIS

Since the CMSIS is incorporated inside the device driver library, there is no special setup requirement for using CMSIS in projects. For each MCU device, the MCU vendor provides a header file, which pulls in additional header files required by the device driver library, including the Core Peripheral Access Layer defined by ARM (see Figure 10.8).

The file *core_cm3.h* contains the peripheral register definitions and access functions for the Cortex-M3 processor peripherals like NVIC, System Control Block registers, and SYSTICK registers. The *core_cm3.h* file also contains declaration of CMSIS intrinsic functions to allow C applications to access instructions that cannot be generated using IEC/ISO C language. In addition, this file also contains a function for outputting a debug message via the Instrumentation Trace Module (ITM).

Note that in some cases, the intrinsic functions in CMSIS could have similar names compared with the intrinsic functions provided in the C compilers, whereas the CMSIS intrinsic functions are compiler independent.

The file *core_cm3.c* contains implementation of CMSIS intrinsic functions that cannot be implemented in core_cm3.h using simple definitions.

The *system_<device>.h* file contains microcontroller specific interrupt number definitions, and peripheral register definitions. The *system_<device>.c* file contains a microcontroller specific function called *SystemInit* for system initialization.

In addition, CMSIS compliant device drivers also contain start-up code (which contains the vector table) for various supported compilers, and CMSIS version of intrinsic functions to allow embedded software access to all processor core features on different C compiler products.

Examples of using CMSIS can be found on the microcontroller vendor's web site. You might also find examples in the device driver libraries itself. Alternatively, you can download the ARM CMSIS

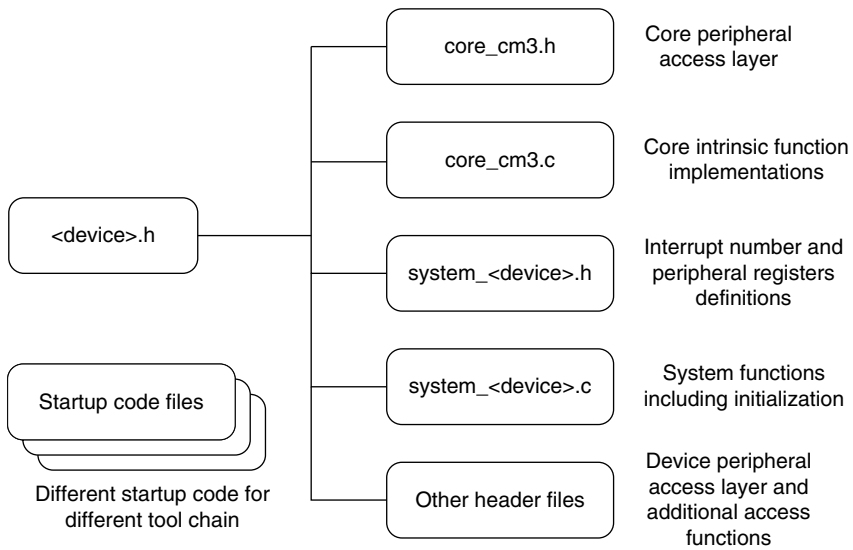

FIGURE 10.8

CMSIS Files.

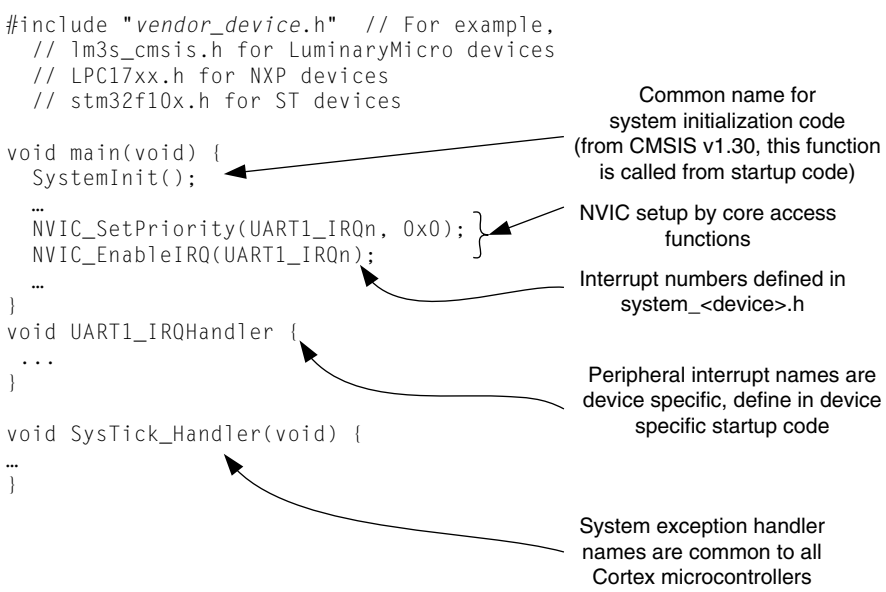

FIGURE 10.9

CMSIS Example.

package from *www.onarm.com*, which contains examples and documentation. Documentation of the common functions can also be found in this package.

A simple example of using CMSIS in your application development is shown in Figure 10.9. To use the CMSIS to set up interrupts and exceptions, you need to use the exception/interrupt constants defined in the *system_<device>.h*. These exception and interrupt constants are different from the exception number used in the core internal registers (e.g., Interrupt Program Status Register [IPSR]). For CMSIS, negative numbers are for system exceptions and positive numbers are for peripheral interrupts.

For development of portable code, you should use the core access functions to access core functionalities and middleware access functions to access peripheral. This allows the porting of software to be minimized between different Cortex microcontrollers.

Details of common CMSIS access functions and intrinsic functions can be found in Appendix G.

10.4.5 Benefits of CMSIS

So what does CMSIS mean to end users?

The main advantage is much better software portability and reusability. Besides easy migration between different Cortex-M3 microcontrollers, it also allows software to be quickly ported between Cortex-M3 and other Cortex-M processors, reducing time to market.

For embedded OS vendors and middleware providers, the advantages of the CMSIS are significant. By using the CMSIS, their software products can become compatible with device drivers from multiple microcontroller vendors, including future microcontroller products that are yet to be released (see Figure 10.10). Without the CMSIS, the software vendors either have to include a small library for

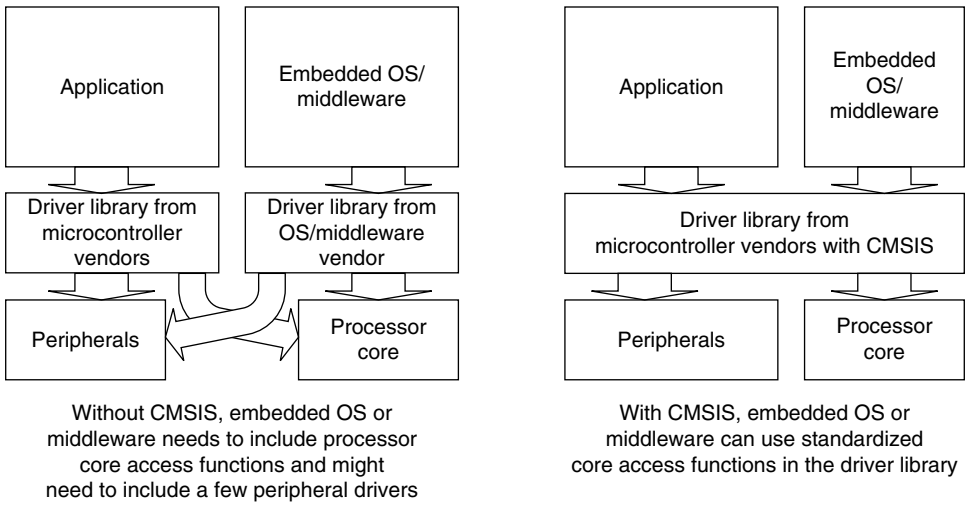

FIGURE 10.10

CMSIS Avoids Overlapping Driver Code.

Cortex-M3 core functions or develop multiple configurations of their product so that it can work with device libraries from different microcontroller vendors.

The CMSIS has a small memory footprint (less than 1 KB for all core access functions and a few bytes of RAM). It also avoids overlapping of core peripheral driver code when reusing software code from other projects.

Since CMSIS is supported by multiple compiler vendors, embedded software can compile and run with different compilers. As a result, embedded OS and middleware can be MCU vendor independent and compiler tool vendor independent. Before availability of CMSIS, intrinsic functions were generally compiler specific and could cause problems in retargetting the software in a different compiler.

Since all CMSIS compliant device driver libraries have a similar structure, learning to use different Cortex-M3 microcontrollers is even easier as the software interface has similar look and feel (no need to relearn a new application programming interface).

CMSIS is tested by multiple parties and is Motor Industry Software Reliability Association (MISRA) compliant, thus reducing the validation effort required for developing your own NVIC or core feature access functions.

10.5 USING ASSEMBLY

For small projects, it is possible to develop the whole application in assembly language. However, this is often much harder for beginners. Using assembler, you might be able to get the best optimization you want, though it might increase your development time, and it could be easy to make mistakes. In addition, handling complex data structures or function library management can be extremely difficult

in assembler. Yet even when the C language is used in a project, in some situations part of the program is implemented in assembly language as follows:

- Functions that cannot be implemented in C, such as direct manipulation of stack data or special instructions that cannot be generated by the C compiler in normal C-code
- Timing-critical routines
- Tight memory requirements, causing part of the program to be written in assembly to get the smallest memory size

10.5.1 The Interface between Assembly and C

In various situations, assembly code and the C program interact. For example,

- When embedded assembly (or inline assembler, in the case of the GNU tool chain) is used in C program code
- When C program code calls a function or subroutine implemented in assembler in a separate file
- When an assembly program calls a C function or subroutine

In these cases, it is important to understand how parameters and return results are passed between the calling program and the function being called. The mechanisms of these interactions are specified in the *ARM Architecture Procedure Call Standard* [*AAPCS*, [Ref. 5]].

For simple cases, when a calling program needs to pass parameters to a subroutine or function, it will use registers R0–R3, where R0 is the first parameter, R1 is the second, and so on. Similarly, R0 is used for returning a value at the end of a function. R0–R3 and R12 can be changed by a function or subroutine whereas the contents of R4–R11 should be restored to the previous state before entering the function, usually handled by stack PUSH and stack POP.

To make them easier to understand, the examples in this book do not strictly follow AAPCS practices. If a C function is called by an assembly code, the effect of a possible register change to R0–R3 and R12 will need to be taken into account. If the contents of these registers are needed at a later stage, these registers might need to be saved on the stack and restored after the C function completes. Since the example codes mostly only call assembly functions or subroutines that affect a few registers or restore the register contents at the end, it's not necessary to save registers R0–R3 and R12.

10.5.2 The First Step in Assembly Programming

This chapter reviews a few examples in assembly language. In most cases, you will be programming in C, but by looking into some assembler examples, we can gain a better understanding of how to use the Cortex-M3 processor. The examples here are based on ARM assembler tools (armasm) in RVDS. For users of Keil MDK-ARM, the command line options are slightly different. For other assembler tools, the file format and instruction syntax will also need to be modified. In addition, some development tools will actually do the startup code for you, so you might not need to worry about creating your assembly startup code.

The first simple program can be something like this

```
STACK_TOP EQU 0x20002000; constant for SP starting value

     AREA |Header Code |, CODE
     DCD STACK_TOP ; Stack top
```

```
        DCD Start      ; Reset vector
        ENTRY          ; Indicate program execution start here
Start ; Start of main program
        ; initialize registers
        MOV r0, #10    ; Starting loop counter value
        MOV r1, #0     ; starting result
        ; Calculated 10+9+8+...+1
loop
        ADD r1, r0     ; R1 = R1 + R0
        SUBS r0, #1    ; Decrement R0, update flag ("S" suffix)
        BNE loop       ; If result not zero jump to loop
        ; Result is now in R1
deadloop
        B deadloop     ; Infinite loop
        END            ; End of file
```

This simple program contains the initial stack pointer (SP) value, the initial program counter (PC) value, and setup registers and then does the required calculation in a loop.

Assuming you are using ARM RealView compilation tools, this program can be assembled using

```
$> armasm --cpu cortex-m3 -o test1.o test1.s
```

The *-o* option specifies the output file name. The test1.o is an object file. We then need to use a linker to create an executable image (ELF). This can be done by

```
$> armlink --rw_base 0x20000000 --ro_base 0x0 --map -o test1.elf test1.o
```

Here, *--ro-base 0x0* specifies that the read-only region (program ROM) starts at address 0x0; *--rw-base* specifies that the read/write region (data memory) starts at address 0x20000000. (In this example test1.s, we did not have any RAM data defined.) The *--map* option creates an image map, which is useful for understanding the memory layout of the compiled image.

Finally, we need to create the binary image

```
$> fromelf --bin --output test1.bin test1.elf
```

For checking that the image looks like what we wanted, we can also generate a disassembled code list file by

```
$> fromelf -c --output test1.list test1.elf
```

If everything works fine, you can then load your ELF image or binary image into your hardware or instruction set simulator for testing.

10.5.3 Producing Outputs

It is always more fun when you can connect your microcontroller to the outside world. The simplest way to do that is to turn on/off the LEDs. However, this practice is quite limiting because it can only represent very limited information. One of the most common output methods is to send text messages to a console. In embedded product development, this task is often handled by a UART interface connecting

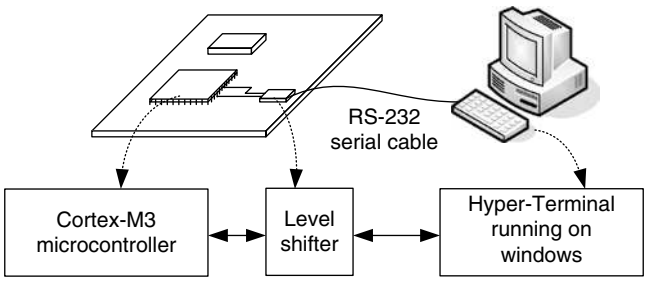

FIGURE 10.11

A Low-Cost Test Environment for Outputting Text Messages.

to a personal computer. For example, a computer running a Windows[1] system with the Hyper-Terminal program acting as a console can be a handy way to produce outputs (see Figure 10.11).

The Cortex-M3 processor does not contain a UART interface, but most Cortex-M3 microcontrollers come with UART provided by the chip manufacturers. The specification of the UART can differ among various devices, so we won't attempt to cover the topic in this book. Our next example assumes that a UART is available and has a status flag to indicate whether the transmit buffer is ready for sending out new data. A level shifter is needed in the connection because RS-232 has a different voltage level than the microcontroller I/O pins.

UART is not the only solution to output text messages. A number of features are implemented on the Cortex-M3 processor to help output debugging messages:

- *Semihosting*: Depending on the debugger and code library support, *semihosting* (outputting *printf* messages via a debug probe device) can be done via debug register in the NVIC. More information on this topic is covered in Chapter 15. In these cases, you can use *printf* within your C program, and the output will be displayed on the console/standard output (STDOUT) of the debugger software.
- *Instrumentation trace*: If the Cortex-M3 microcontroller provides a trace port and an external Trace Port Analyzer (TPA) is available, instead of using UART to output messages, we can use the ITM. The trace port works much faster than UART and can offer more data channels.
- *Instrumentation trace via Serial-Wire Viewer (SWV)*: Alternatively, the Cortex-M3 processor (revision 1 and later) also provides an SWV operation mode on the Trace Port Interface Unit (TPIU). This interface allows outputs from ITM to be captured using low-cost hardware instead of a TPA. However, the bandwidth provided with the SWV mode is limited, so it is not ideal for large amounts of data (e.g., instruction trace operation).

10.5.4 The "Hello World" Example

Before we try to write a "Hello world" program, we should figure out how to send one character through the UART. The code used to send a character can be implemented as a subroutine, which can

[1]Windows and Hyper-Terminal are trademarks of Microsoft Corporation.

be called by other message output codes. If the output device changes, we only need to change this subroutine and all the text messages can be output by a different device. This modification is usually called retargetting.

A simple routine to output a character could be something like this

```
UART0_BASE   EQU   0x4000C000
UART0_FLAG   EQU   UART0_BASE+0x018
UART0_DATA   EQU   UART0_BASE+0x000

Putc       ; Subroutine to send a character via UART
           ; Input R0 = character to send
           PUSH {R1,R2, LR}   ; Save registers
           LDR R1,=UART0_FLAG

PutcWaitLoop
           LDR R2,[R1]        ; Get status flag
           TST R2, #0x20      ; Check transmit buffer full flag
                              ; bit
           BNE PutcWaitLoop   ; If busy then loop
           LDR R1,=UART0_DATA ; otherwise
           STRB R0, [R1]      ; Output data to transmit buffer
           POP {R1,R2, PC}    ; Return
```

The register addresses and bit definitions here are just examples; you might need to change the value for your device. In addition, some UART might require a more complex status-checking process before the character is output to the transmit buffer. Furthermore, another subroutine call (*Uart0Initialize* in the following example) is required to initialize the UART, but this depends on the UART specification and will not be covered in this chapter. An example of UART initialization in C for Luminary Micro LM3S811 devices is covered in Chapter 20.

Now, we can use this subroutine to build a number of functions to display messages:

```
Puts       ; Subroutine to send string to UART
           ; Input R0 = starting address of string.
           ; The string should be null terminated
           PUSH {R0 ,R1, LR}     ; Save registers
           MOV R1, R0            ; Copy address to R1, because R0 will
                                 ; be used
PutsLoop                         ; as input for Putc
           LDRB R0,[R1],#1       ; Read one character and increment
                                 ; address
           CBZ  R0, PutsLoopExit ; if character is null, goto end
           BL   Putc             ; Output character to UART
           B    PutsLoop         ; Next character
PutsLoopExit
           POP {R0, R1, PC}      ; Return
```

With this subroutine, we are ready for our first "Hello world" program:

```
STACK_TOP   EQU   0x20002000; constant for SP starting value
UART0_BASE  EQU   0x4000C000
UART0_FLAG  EQU   UART0_BASE+0x018
UART0_DATA  EQU   UART0_BASE+0x000
```

```
                AREA | Header Code|, CODE
                DCD STACK_TOP ; Stack Pointer initial value
                DCD Start ; Reset vector
                ENTRY
Start           ; Start of main program
                MOV r0, #0          ; initialize registers
                MOV r1, #0
                MOV r2, #0
                MOV r3, #0
                MOV r4, #0
                BL  Uart0Initialize ; Initialize the UART0
                LDR r0,=HELLO_TXT   ; Set R0 to starting address of string
                BL  Puts
deadend
                B deadend ; Infinite loop
                ;--------------------------------
                ; subroutines
                ;--------------------------------
Puts            ; Subroutine to send string to UART
                ;Input R0 = starting address of string.
                ; The string should be null terminated
                PUSH {R0 ,R1, LR}   ; Save registers
                MOV R1, R0          ; Copy address to R1, because R0 will
                                    ; be used
PutsLoop                            ; as input for Putc
                LDRB R0,[R1],#1     ; Read one character and increment
                                    ; address
                CBZ  R0, PutsLoopExit ; if character is null, goto end
                BL   Putc             ; Output character to UART
                B    PutsLoop         ; Next character
PutsLoopExit
                POP {R0, R1, PC}    ; Return
                ;--------------------------------
Putc            ; Subroutine to send a character via UART
                ; Input R0 = character to send
                PUSH {R1,R2, LR}     ; Save registers
                LDR   R1,=UART0_FLAG
PutcWaitLoop
                LDR   R2,[R1]        ; Get status flag
                TST   R2, #0x20      ; Check transmit buffer full flag bit
                BNE   PutcWaitLoop   ; If busy then loop
                LDR   R1,=UART0_DATA ; otherwise
                STR   R0, [R1]       ; Output data to transmit buffer
                POP  {R1,R2, PC}     ; Return
                ;--------------------------------
Uart0Initialize
                ; Device specific, not shown here
                BX    LR ; Return
                ;--------------------------------
HELLO_TXT
                DCB "Hello world\n",0 ; Null terminated Hello
                                      ; world string
                END     ; End of file
```

The only thing you need to add to this code is the details for the *Uart0Initialize* subroutine and modify the UART register address constants at the top of the file.

It will also be useful to have subroutines that output register values as well. To make things easier, they can all be based on *Putc* and *Puts* subroutines we have already done. The first subroutine is to display hexadecimal values.

```
PutHex ; Output register value in hexadecimal format
       ; Input R0 = value to be displayed
     PUSH {R0-R3,LR}
     MOV R3, R0    ; Save register value to R3 because R0 is used
                   ; for passing input parameter
      MOV R0,#'0' ; Starting the display with "0x"
      BL  Putc
      MOV R0,#'x'
      BL  Putc
      MOV R1, #8       ; Set loop counter
      MOV R2, #28      ; Rotate offset
PutHexLoop
     ROR R3,  R2       ; Rotate data value left by 4 bits
                       ; (right 28)
     AND R0,  R3,#0xF ; Extract the lowest 4 bit
     CMP R0,  #0xA     ; Convert to ASCII
     ITE GE
     ADDGE R0, #55     ; If larger or equal 10, then convert
                       ; to A-F
     ADDLT R0, #48     ; otherwise convert to 0-9
     BL Putc            ; Output 1 hex character
     SUBS R1,  #1      ; decrement loop counter
     BNE PutHexLoop    ; if all 8 hexadecimal character been
                       ; display then
     POP {R0-R3,PC}    ; return, otherwise process next 4-bit
```

This subroutine is useful for outputting register values. However, sometimes we also want to output register values in decimal. This sounds like a rather complex operation, but in the Cortex-M3 it is easy because of the hardware multiply and divide instructions. One of the other main problems is that during calculation, we will get output characters in reverse order, so we need to put the output results in a text buffer first, wait until the whole text is ready to display, and then use the *Puts* function to display the whole result. In this example, a part of the stack memory is used as the text buffer:

```
PutDec  ; Subroutine to display register value in decimal
            ; Input R0 = value to be displayed.
            ; Since it is 32 bit, the maximum number of character
            ; in decimal format, including null termination is 11
            PUSH {R0-R5, LR}  ; Save register values
            MOV R3, SP        ; Copy current Stack Pointer to R3
            SUB SP, SP, #12   ; Reserved 12 bytes as text buffer
            MOV R1, #0        ; Null character
            STRB R1,[R3, #-1]!; Put null character at end of text
                              ; buffer,pre-indexed
            MOV R5, #10     ; Set divide value
PutDecLoop
            UDIV R4, R0, R5 ; R4 = R0 / 10
```

```
MUL R1, R4, R5      ; R1 = R4 * 10
SUB R2, R0, R1      ; R2 = R0 - (R4 * 10) = remainder
ADD R2, #48         ; convert to ASCII (R2 can only be 0-9)
STRB R2,[R3, #-1]! ; Put ascii character in text
                    ; buffer, pre-indexed
MOVS R0, R4         ; Set R0 = Divide result and set Z flag
; if R4=0
BNE PutDecLoop      ; If R0(R4) is already 0, then there
                    ; is no more digit
MOV R0, R3          ; Put R0 to starting location of text
                    ; buffer
BL Puts             ; Display the result using Puts
ADD SP, SP, #12     ; Restore stack location
POP {R0-R5, PC}     ; Return
```

With various features in the Cortex-M3 instruction set, the processing to convert values into decimal format display can be implemented in a very short subroutine.

10.5.5 Using Data Memory

Back to our first example: When we were doing the linking stage, we specified the read/write memory region. How do we put data there? The method is to define a data region in your assembly file. Using the same example from the beginning, we can store the data in the data memory at 0x20000000 (the SRAM region). The location of the data section is controlled by a command-line option when you run the linker:

```
STACK_TOP   EQU 0x20002000   ; constant for SP starting value
            AREA | Header Code|, CODE
            DCD STACK_TOP    ; SP initial value
            DCD Start        ; Reset vector
            ENTRY
Start       ; Start of main program
            ; initialize registers
            MOV r0, #10 ; Starting loop counter value
            MOV r1, #0  ; starting result
            ; Calculated 10+9+8+…+1
loop
            ADD r1, r0      ; R1 = R1 + R0
            SUBS r0, #1     ; Decrement R0, update flag ("S"
                            ; suffix)
            BNE loop        ; If result not zero jump to loop
            ; Result is now in R1
            LDR r0,=MyData1 ; Put address of MyData1 into R0
            STR r1,[r0]     ; Store the result in MyData1
deadloop
            B deadloop      ; Infinite loop
            AREA | Header Data|, DATA
            ALIGN 4
MyData1     DCD 0           ; Destination of calculation result
MyData2     DCD 0
            END             ; End of file
```

During the linking stage, the linker will put the DATA region into read/write memory, so the address for *MyData1* will be 0x20000000 in this case.

10.6 USING EXCLUSIVE ACCESS FOR SEMAPHORES

Exclusive access instructions are used for semaphore operations—for example, a MUTEX (Mutual Exclusion) to make sure that a resource is used by only one task. For instance, let's say that a data variable DeviceALocked in memory can be used to indicate that Device A is being used. If a task wants to use Device A, it should check the status by reading the variable DeviceALocked. If it is zero, it can write a 1 to DeviceALocked to lock the device. After it's finished using the device, it can then clear the DeviceALocked to zero so that other tasks can use it.

What will happen if two tasks try to access Device A at the same time? In that case, possibly both tasks will read the variable DeviceALocked, and both will get zero. Then both of them will try writing back 1 to the variable DeviceALocked to lock the device, and we'll end up with both tasks believing that they have exclusive access to Device A. That is where exclusive accesses are used. The STREX instruction has a return status, which indicates whether the exclusive store has been successful. If two tasks try to lock a device at the same time, the return status will be 1 (exclusive failed) and the task can then know that it needs to retry the lock.

Chapter 5 provided some background on the use of exclusive accesses. The flowchart in that earlier discussion is shown in Figure 10.12.

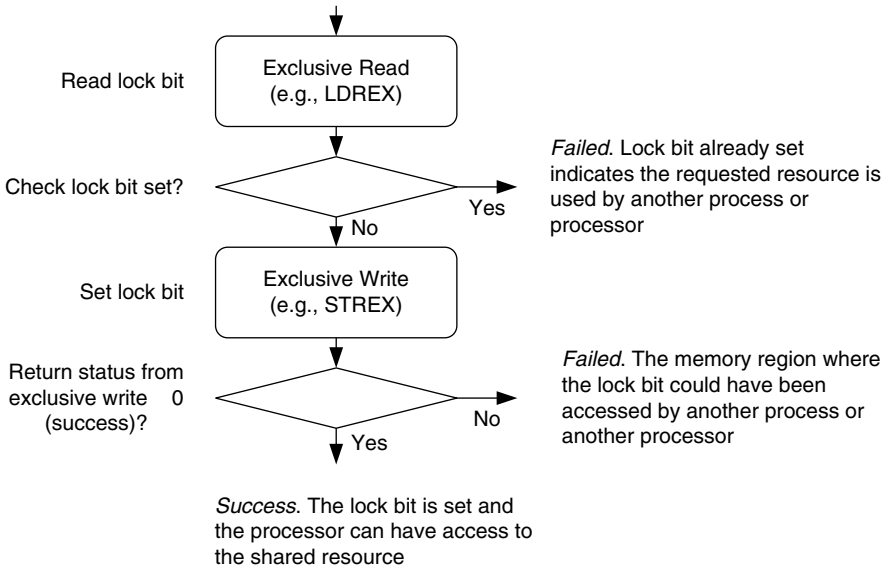

FIGURE 10.12

Using Exclusive Access for Semaphore Operations.

The operation can be carried out by the following C-code using intrinsic functions from CMSIS. Note that the data write operation of STREX will not be carried out if the exclusive monitor returns a fail status, preventing a lock bit being set when the exclusive access fails:

```
volatile unsigned int DeviceALocked; // lock variable

int LockDeviceA(void){
 unsigned int status; // variable to hold STREX status
 // Get the lock status and see if it is already locked
if (__LDREXW(&DeviceALocked) = 0) {
 // if not locked, try set lock to 1
 status = __STREXW(1, &DeviceALocked);
 if (status!=0) return (1); // return fail status
 else return(0); // return success status
} else {
 return(1); // return fail status
 }
}
```

The same operation can also be carried out by the following assembly code:

```
LockDeviceA
            ; A simple function to try to lock Device A
            ; Output R0 : 0 = Success, 1 = failed
            If successful, value of 1 will be written to variable
            ; DeviceALocked
            PUSH {R1, R2, LR}
TryToLockDeviceA
            LDR   R1,=DeviceALocked  ; Get the lock status
            LDREX R2,[R1]
            CMP   R2,#0              ; Check if it is locked
            BNE   LockDeviceAFailed
DeviceAIsNotLocked
            MOV    R0,#1             ; Try to write 1 to
                                     ; DeviceALocked
            STREX R2,R0,[R1]         ; Exclusive write
            CMP    R2, #0
            BNE LockDeviceAFailed    ; STREX Failed
LockDeviceASucceed
            MOV    R0,#0             ; Return success status
            POP {R1, R2, PC}         ; Return
LockDeviceAFailed
            MOV R0,#1                ; Return fail status
            POP {R1, R2, PC}         ; Return
```

If the return status of this function is 1 (exclusive failed), the application tasks should wait a bit and retry later. In single-processor systems, the common cause of an exclusive access failing is an Interrupt occurring between the exclusive load and the exclusive store. If the code is run in privileged mode, this situation can be prevented by setting an Interrupt Mask register, such as PRIMASK, for a short time to increase the chance of getting the resource locked successfully.

In multiprocessor systems, aside from interrupts, the exclusive store could also fail if another processor has accessed the same memory region. To detect memory accesses from different processors,

the bus infrastructure requires exclusive access monitor hardware to detect whether there is an access from a different bus master to a memory between the two exclusive accesses. However, in most low-cost Cortex-M3 microcontrollers, there is only one processor, so this monitor hardware is not required.

With this mechanism, we can be sure that only one task can have access to certain resources. If the application cannot gain the lock to the resource after a number of times, it might need to quit with a timeout error. For example, a task that locked a resource might have crashed and the lock remained set. In these situations, the OS should check which task is using the resource. If the task has completed or terminated without clearing the lock, the OS might need to unlock the resource.

If the process has started an exclusive access using LDREX and then found that the exclusive access is no longer needed, it can use the CLREX instruction to clear the local record in the exclusive access monitor. This can be done with CMSIS function:

```
void __CLREX(void);
```

If assembly language is used, the CLREX instruction can be used:

```
CLREX
```

or

```
CLREX.W
```

For the Cortex-M3 processor, all exclusive memory transfers must be carried out sequentially. However, if the exclusive access control code has to be reused on other ARM Cortex processors, the Data Memory Barrier (DMB) instruction might need to be inserted between exclusive transfers to ensure correct ordering of the memory accesses. Example code of using barrier instructions with exclusive accesses can be found in Section 14.3, Multiprocessor Communication.

10.7 USING BIT BAND FOR SEMAPHORES

It is possible to use the bit-band feature to carry semaphore operations, provided that the memory system supports locked transfers or only one bus master is present on the memory bus. With bit band, it is possible to carry out the semaphore in normal C-code, but the operation is different from using exclusive access. To use bit band as a resource allocation control, a memory location (such as word data) with a bit-band memory region is used, and each bit of this variable indicates that the resource is used by a certain task.

Since the bit-band alias writes are locked READ-MODIFY-WRITE transfers (the bus master cannot be switched to another one between the transfers), provided that all tasks only change the lock bit representing themselves, the lock bits of other tasks will not be lost, even if two tasks try to write to the same memory location at the same time. Unlike using exclusive accesses, it is possible for a resource to be "locked" simultaneously by two tasks for a short period of time until one of them detects the conflict and releases the lock (see Figure 10.13).

Using bit band for semaphores can work only if all the tasks in the system change only the lock bit they are assigned to using the bit-band alias. If any of the tasks change the lock variable using a normal write, the semaphore can fail because another task sets a lock bit just before the write to the lock variable, the previous lock bit set by the other task will be lost.

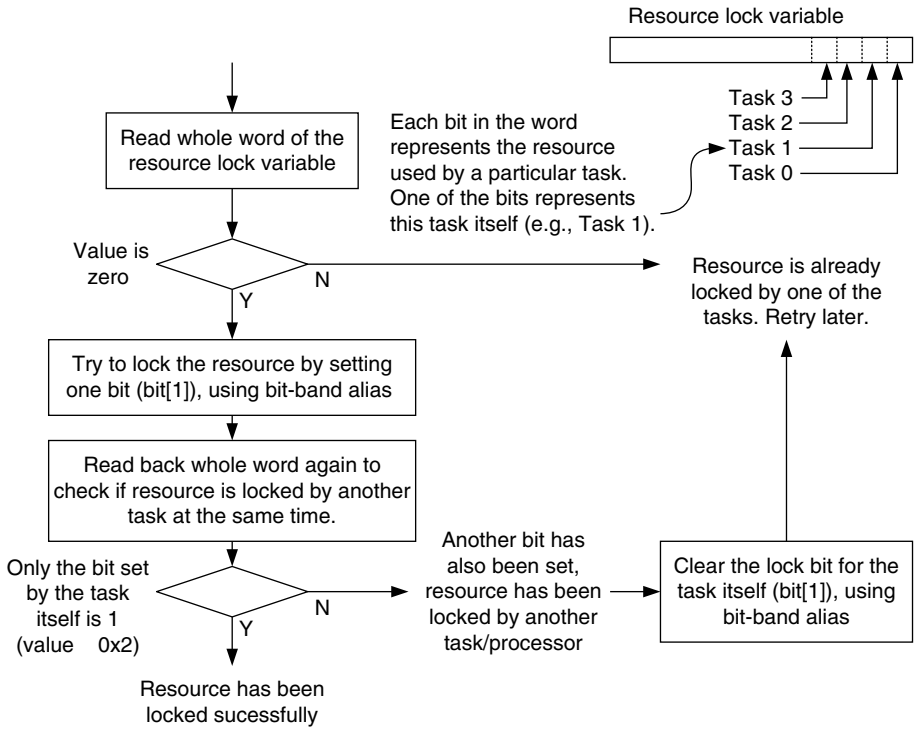

FIGURE 10.13

Mutex Implemented Using Bit Band as a Semaphore Control.

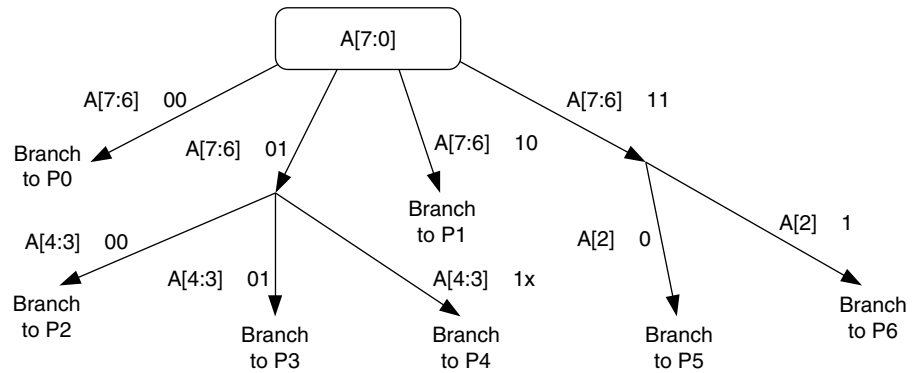

FIGURE 10.14

Bit Field Decoder: Example Use of UBFX and TBB Instructions.

10.8 WORKING WITH BIT FIELD EXTRACT AND TABLE BRANCH

We examined the unsigned bit field extract (UBFX) and Table Branch (TBB/TBH) instructions in Chapter 4. These two instructions can work together to form a very powerful branching tree. This capability is very useful in data communication applications where the data sequence can have different meanings with different headers. For example, let's say that the following decision tree based on Input A is to be coded in assembler (see Figure 10.14).

```
DecodeA
    LDR  R0,=A           ; Get the value of A from memory
    LDR  R0,[R0]
    UBFX R1, R0, #6, #2  ; Extract bit[7:6] into R1
    TBB  [PC, R1]
BrTable1
    DCB  ((P0      -BrTable1)/2) ; Branch to P0       if A[7:6] = 00
    DCB  ((DecodeA1-BrTable1)/2) ; Branch to DecodeA1 if A[7:6] = 01
    DCB  ((P1      -BrTable1)/2) ; Branch to P1       if A[7:6] = 10
    DCB  ((DecodeA2-BrTable1)/2) ; Branch to DecodeA1 if A[7:6] = 11
DecodeA1
    UBFX  R1, R0, #3, #2 ; Extract bit[4:3] into R1
    TBB   [PC, R1]
BrTable2
    DCB  ((P2 -BrTable2)/2) ; Branch to P2    if A[4:3] = 00
    DCB  ((P3 -BrTable2)/2) ; Branch to P3    if A[4:3] = 01
    DCB  ((P4 -BrTable2)/2) ; Branch to P4    if A[4:3] = 10
    DCB  ((P4 -BrTable2)/2) ; Branch to P4    if A[4:3] = 11
DecodeA2
    TST  R0, #4 ; Only 1 bit is tested, so no need to use UBFX
    BEQ  P5
    B    P6
P0 ... ; Process 0
P1 ... ; Process 1
P2 ... ; Process 2
P3 ... ; Process 3
P4 ... ; Process 4
P5 ... ; Process 5
P6 ... ; Process 6
```

This code completes the decision tree in a short assembler code sequence. If the branch target addresses are at a larger offset, some of the TBB instructions would have to be replaced by TBH instructions.

This page intentionally left blank