Unit 3

Introduction to Cortex-M0 Programming

This chapter covers

- Introduction to Cortex-M0 Programming
    - Introduction to Embedded System Programming
        - What happens when a microcontroller starts?
        - Designing Embedded programs
    - Input and outputs
    - Development Flow
    - C programming and Assembly Programming
    - What is in a Programming Image
        - Vector Table
        - C start up code
        - C library code
        - Data in RAM
    - C programming Data types
    - Accessing Peripherals in C
    - Cortex Microcontroller Software Interface(CMSIS)
        - Introduction to CMSIS
        - What is Standardized in CMSIS
        - Organization of the CMSIS
        - Using CMSIS
        - Benefits of CMSIS
- Instruction Set
    - Background of ARM and Thumb Instruction set
    - Assembly Basics
        - Quick Glance at Assembly Syntax
        - Use of a Suffix
        - Thumb code and unified Assembler Language(UAL)
        - Instruction List
        - Moving Data within processor

        - Memory Accesses
        - Stack Memory Accesses
        - Arithmetic operations
        - Logic Operations
        - Shift and Rotate operations
        - Extend and Reverse ordering
        - Program Flow Control
        - Memory Barrier Instructions
        - Exception related instructions
            - ✓ Sleep mode Feature- Related Instructions
        - Other Instructions
    - Pseudo Instructions
- ➢ Instruction Usage Examples
    - Overview
    - Program control
        - If Then Else
        - Loop
        - More on Branch Instructions
        - Typical usage of Branch conditions
        - Function calls and Returns
        - Branch Table
    - Data Accesses
        - Simple Data Accesses
        - Example of using Memory Access Instruction
    - Data Type Conversion
        - Conversion of Data Size
        - Endian Conversion
    - Data Processing
        - 64 Bit/128 Bit Add
        - 64 Bit/128 Bit Sub
        - Integer Divide

        - Unsigned Integer Square root
        - Bit and bit field computations
- implementation of various structures like loop, switch, functions, subroutines
- **Self-Study**
    - **Programs with subroutines**

3.1 *Introduction to Cortex-M0 Programming*

3.1.1 Introduction to Embedded System Programming

All microcontrollers need program code to enable them to perform their intended tasks. If your only experience comes from developing programs for personal computers, you might find the software development for microcontrollers very different. Many embedded systems do not have any operating systems (sometimes these systems are referred as bare metal targets) and do not have the same user interface as a personal computer. If you are completely new to microcontroller programming, do not worry. Programming the Cortex-M0 is easy. As long as you have a basic understanding of the C language, you will soon be able develop simple applications on the Cortex-M0.

3.1.1.1 What Happens When a Microcontroller Starts?

Most modern microcontrollers have on-chip flash memory to hold the compiled program. The flash memory holds the program in binary machine code format, and therefore programs written in C must be compiled before programmed to the flash memory. Some of these microcontrollers might also have a separate boot ROM, which contains a small boot loader program that is executed when the microcontroller starts, before executing the user program in the flash memory. In most cases, only the program code in the flash memory can be changed and the boot loader is fixed.

After the flash memory (or other types of program memory) is programmed, the program is then accessible by the processor. After the processor is reset, it carries out the reset sequence, as outlined at the end of the previous chapter (Figure 4.1).

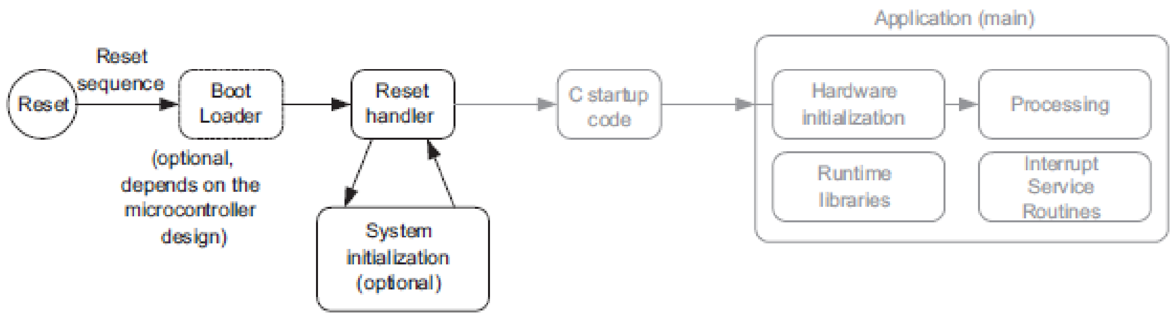


Figure 4.1:
What happens when a microcontroller starts—the Reset handler.

In the reset sequence, the processor obtains the initial MSP value and reset vector, and then it executes the reset handler. All of this required information is usually stored in a program file called startup code. The reset handler in the startup code might also perform system initialization (e.g., clock control circuitry and Phase Locked Loop [PLL]), although in some cases system initialization is carried out later when the C program "main()" starts. Example startup code can usually be found in the installation of the development suite or from software packages available from the microcontroller vendors. For example, if the Keil Microcontroller Development Kit (MDK) is used for development, the project creation wizard can optionally copy a default startup code file into your project that matches the microcontroller you selected.

For applications developed in C, the C startup code is executed before entering the main application code. The C startup code initializes variables and memory used by the application and they are inserted to the program image by the C development suite (Figure 4.2).

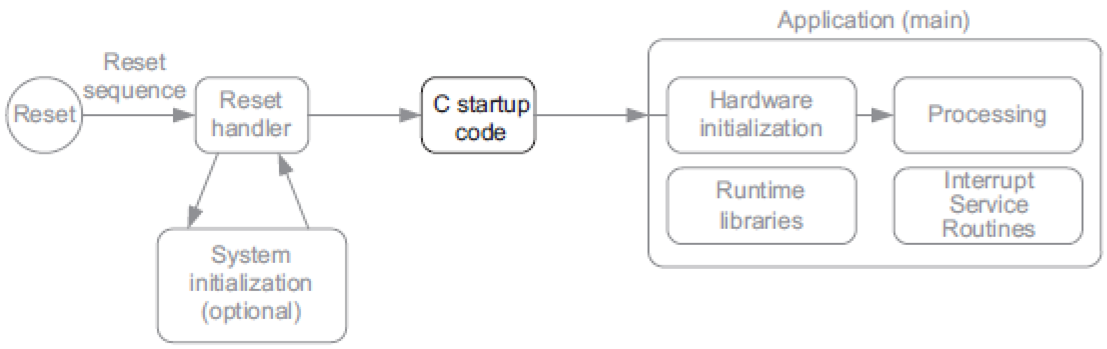


Figure 4.2:
What happens when a microcontroller starts—C startup code.

After the C startup code is executed, the application starts. The application program often contains the following elements:

• Initialization of hardware (e.g., clock, PLL, peripherals)

• The processing part of the application

• Interrupt service routines

In addition, the application might also use C library functions (Figure 4.3). In such cases, the C compiler/linker will include the required library functions into the compiled program image. The hardware initialization might involve a number of peripherals, some system control registers, and interrupt control registers inside the Cortex-M0 processors. The initialization of the system clock control and the PLL might also take place if this were not carried out in the reset handler. After the peripherals are initialized, the program execution can then proceed to the application processing part.

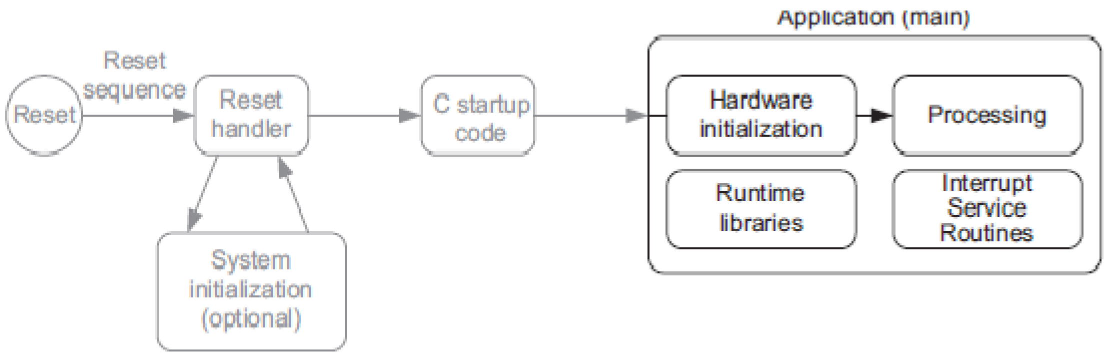


Figure 4.3:
What happens when a microcontroller starts—application.

3.1.1.2 Designing Embedded Programs

There are many ways to structure the flow of the application processing. Here we will cover a few fundamental concepts.

Polling

For simple applications, polling (sometimes also called super loop) is easy to set up and works fairly well for simple tasks (Figure 4.4).

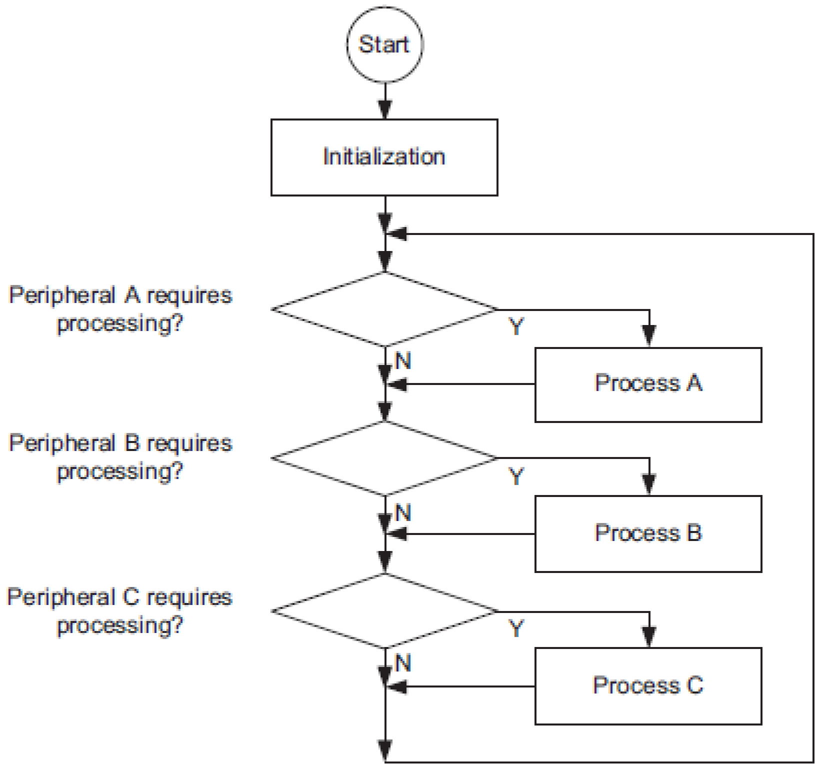

Start
Initialization
Peripheral A requires processing?
Y
N
Process A
Peripheral B requires processing?
Y
N
Process B
Peripheral C requires processing?
Y
N
Process C

Figure 4.4:
Polling method for simple application processing.

However, when the application gets complicated and demands higher processing performance, polling is not suitable. For example, if one of the processes takes a long time, other peripherals will not receive any service for some time.Another disadvantage of using the pollingmethod is that the processor has to run the polling program all the time, even if it requires no processing.

Interrupt Driven

In applications that require lower power, processing can be carried out in interrupt service routines so that the processor can enter sleep mode when no processing is required. Interrupts are usually generated by external sources or on chip peripherals to wake up the processor.

In interrupt-driven applications (Figure 4.5), the interrupts from different devices can be set at different priorities. In this way a high-priority interrupt request can obtain service even when a lower-priority interrupt service is running, which will be temporarily stopped. As a result, the latency for the higher-priority interrupt is reduced.

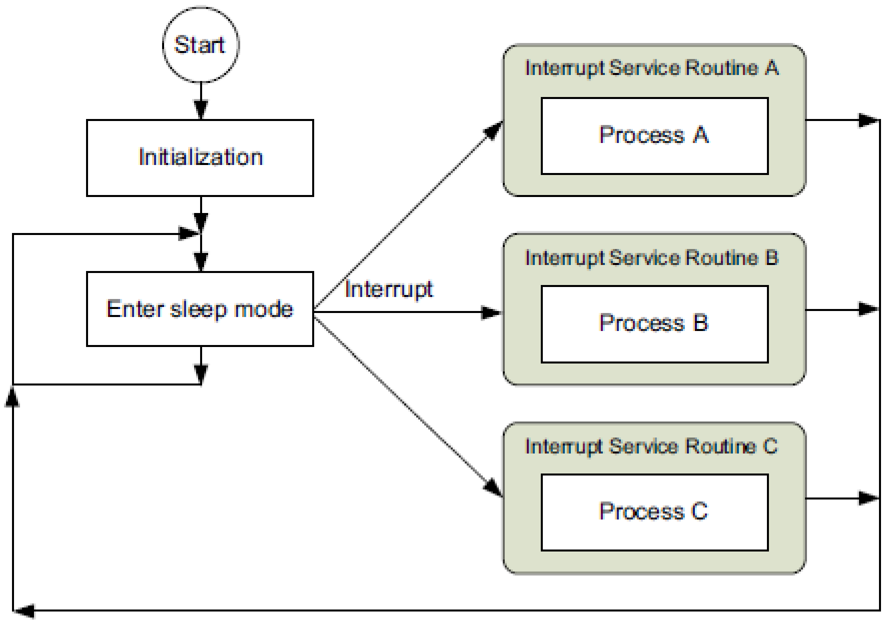
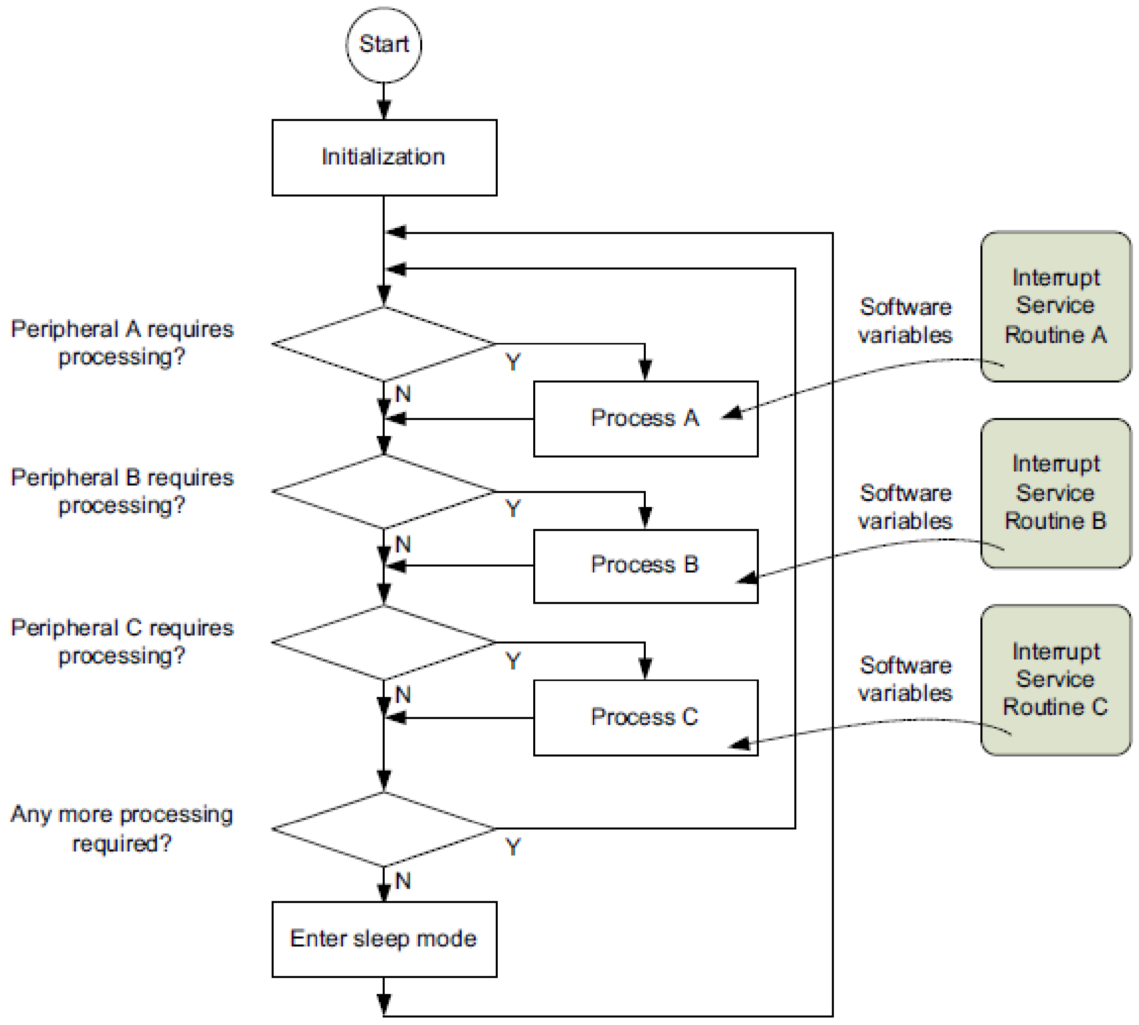

Start
Initialization
Enter sleep mode
Interrupt
Interrupt Service Routine A
Process A
Interrupt Service Routine B
Process B
Interrupt Service Routine C
Process C

Figure 4.5:
An interrupt-driven application.

Combination of Polling and Interrupt Driven

In many cases, applications can use a combination of polling and interrupt methods (Figure 4.6). By using software variables, information can be transferred between interrupt service routines and the application processes.

By dividing a peripheral processing task into an interrupt service routine and a process running in the main program, we can reduce the duration of interrupt services so that even lowerpriority interrupt services gain a better chance of getting serviced. At the same time, the system



Start
Initialization
Peripheral A requires processing?
Y
N
Process A
Peripheral B requires processing?
Y
N
Process B
Peripheral C requires processing?
Y
N
Process C
Any more processing required?
Y
N
Enter sleep mode
Software variables
Interrupt Service Routine A
Software variables
Interrupt Service Routine B
Software variables
Interrupt Service Routine C

can still enter sleep mode when no processing task is required. In Figure 4.6, the application is partitioned into processes A, B, and C, but in some cases, an application cannot be partitioned into individual parts easily and needs to be written as a large combined process.

Handling Concurrent Processes

In some cases, an application process could take a significant amount of time to complete and therefore it is undesirable to handle it in a big loop as shown in Figure 4.6. If process A takes too long to complete, processes B and C will not able to respond to peripheral requests fast enough, resulting in system failure. Common solutions are as follows:

1. Breaking down a long processing task to a sequence of states. Each time the process is accessed, only one state is executed.

2. Using a real-time operating system (RTOS) to manage multiple tasks.

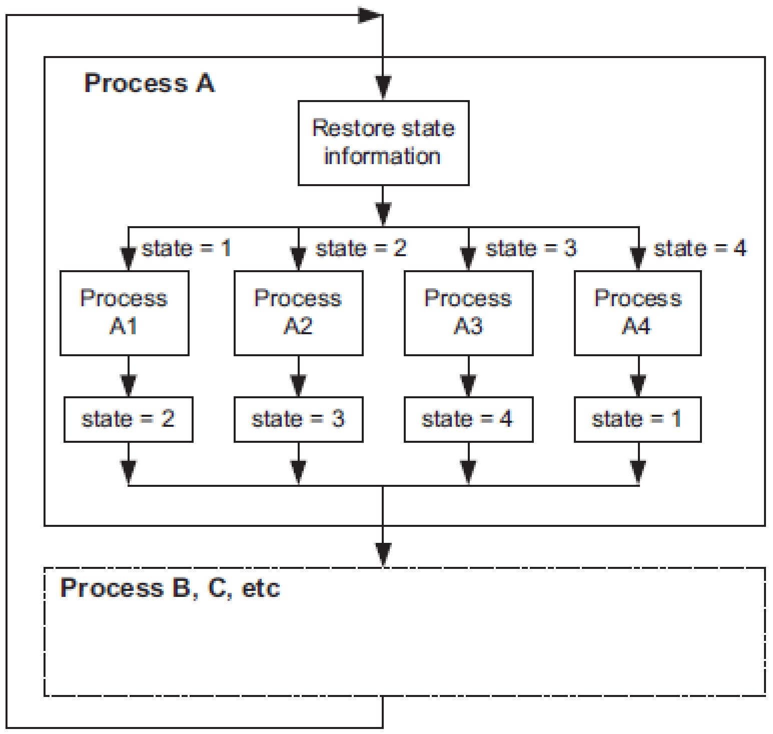


Figure 4.7:
Partitioning a process into multiple parts in the application loop.

For method 1, a process is divided into a number of parts, and software variables are used to track the state of the process (Figure 4.7). Each time the process is executed, the state information is updated so that next time the process is executed, the processing can resume correctly.

Because the execution path of the process is shortened, other processes in the main loop can be reached quicker inside the big loop. Although the total processing time required for the processing remains unchanged (or increases slightly because of the overhead of state saving and restoring), the system is more responsive. However, when the application tasks become more complex, partitioning the application task manually can become impractical.

For more complex applications, a real-time operating system (RTOS) can be used (Figure 4.8). An RTOS allows multiple application processes to be executed by dividing processor execution time into time slots and allocating one to each task. To use an RTOS, a timer is needed to generate regular interrupt requests. When each time slot ends, the timer generates an interrupt that triggers the RTOS task scheduler, which determines if context switching should be carried out. If context switching should be carried out, the task schedule suspends the current executing task and then switches to the next task that is ready to be executed.

Using an RTOS improves the responsiveness of a system by ensuring that all tasks are reached within a certain amount of time. Examples of using an RTOS are covered in Chapter 18.

3.1.2 Inputs and Outputs

On many embedded systems, the available inputs and outputs can be limited to simple electronic interfaces like digital and analog inputs and outputs (I/Os), UARTs, I2C, SPI, and so on. Many microcontrollers also offer USB, Ethernet, CAN, graphics LCD, and SD card interfaces. These interfaces are handled by peripherals in the microcontrollers.

On Cortex-M0 microcontrollers, peripherals are controlled by memory-mapped registers (examples of accessing peripherals are presented later in this chapter). Some of these peripherals are more sophisticated than peripherals available on 8-bit and 16-bit microcontrollers, and there might be more registers to program during the peripheral setup.

Typically, the initialization process for peripherals may consist of the following steps:

1. Programming the clock control circuitry to enable the clock signal to the peripheral and the corresponding I/O pins if necessary. In many low-power microcontrollers, the clock signals reaching different parts of the chip can be turned on or off individually to save power. By default, most of the clock signals are usually turned off and need to be enabled before the peripherals are programmed. In some cases you also need to enable the clock signals for the peripherals bus system.
2. Programming of I/O configurations. Most microcontrollers multiplex their I/O pins for multiple uses. For a peripheral interface to work correctly, the I/O pin assignment might need to be programmed. In addition, some microcontrollers also offer configurable electrical

characteristics for the I/O pins. This can result in additional steps in I/O configurations.

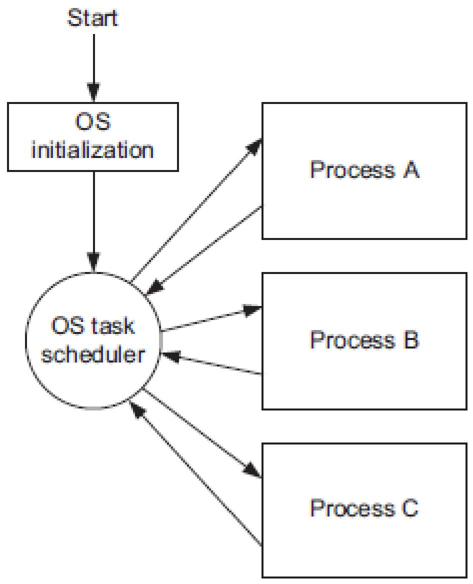


Figure 4.8:
Using an RTOS to handle multiple concurrent application processes.

3. Peripheral configuration. Most interface peripherals contain a number of programmable registers to control their operations, and therefore a programming sequence is usually needed to allow the peripheral to work correctly.
4. Interrupt configuration. If a peripheral operation requires interrupt processing, additional steps are required for the interrupt controller (e.g., the NVIC in the Cortex-M0).

Most microcontroller vendors provide device driver libraries for peripheral programming to simplify software development. Unlike programming on personal computers, you might need to develop your own user interface functions to design a user-friendly standalone embedded system. However, the device driver libraries provided by the microcontroller vendors will make the development of your user interface easier.

For the development of most deeply embedded systems, it is not necessary to have a rich user interface. However, basic interfaces like LEDs, DIP switches, and push buttons can deliver only a limited amount of information. For debugging software, a simple text input/output console is often sufficient. This can be handled by a simple RS-232 connection through a UART interface on the microcontroller to a UART interface on a personal computer (or via a USB adaptor) so that we can display the text messages and enter user inputs using a terminal application (Figure 4.9).

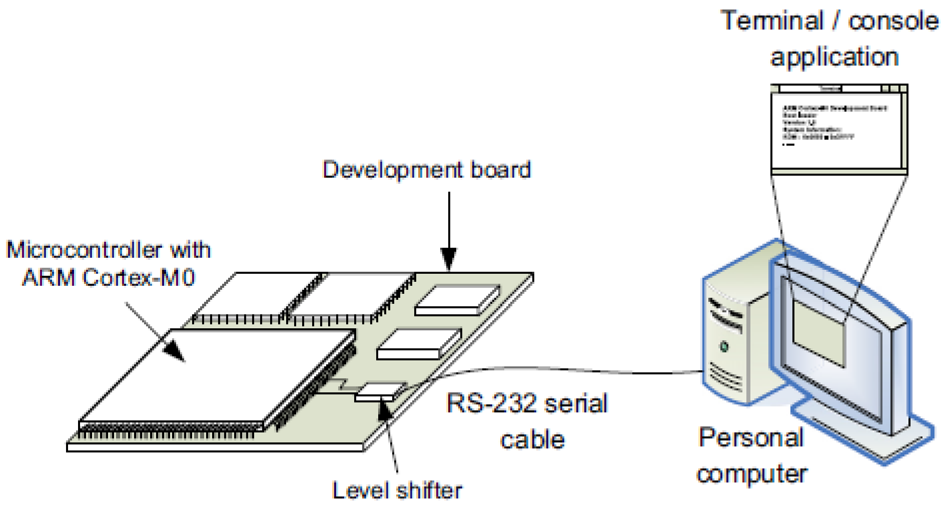


Figure 4.9:
Using UART interface for user input and output.

The technique to redirect text messages from a “printf” (in C language) to a UART (or another interface) is commonly referred to as “retargeting.” Retargeting can also handle user inputs and system functions. Examples of simple retargeting will be presented in later chapters of this
book.

Typically, microcontrollers also provide a number of general-purpose input and output ports (GPIOs) that are suitable for simple control, user buttons or switches, LEDs, and the like. You can also develop an embedded system with a full feature graphics display using a microcontroller with built-in LCD controllers or using an external LCD module with a parallel or SPI interface. Although microcontroller vendors usually provide device driver libraries for the peripheral blocks, you might still need to develop your own user input and output functions.

3.1.3 Development Flow

Many development tool chains are available for ARM microcontrollers. The majority of them support C and assembly language. Embedded projects can be developed in either C or assembly language, or a mixture of both. In most cases, the program-generation flow can be summarized in a diagram, as shown in Figure 4.10.

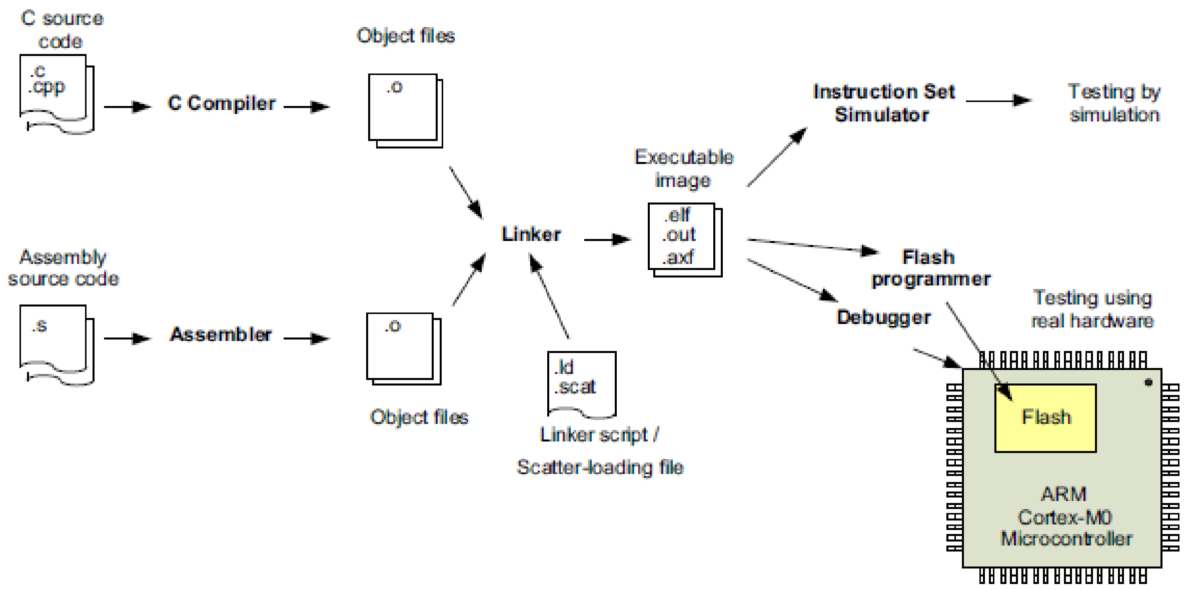


Figure 4.10:
Typical program-generation flow.

In most simple applications, the programs can be completely written in the C language. The C compiler compiles the C program code into object files and then generates the executable program image file using the linker. In the case of GNU C compilers, the compile and linking stages are often merged into one step.

Projects that require assembly programming use the assembler to generate object code from assembly source code. The object files can then be linked with other object files in the project to produce an executable image. Besides the program code, the object files and the executable image may also contain various debug information.

Depending on the development tools, it is possible to specify the memory layout for the linker using command line options. However, in projects using GNU C compilers, a linker script is normally required to specify the memory layout. A linker script is also required for other development tools when the memory layout gets complicated. In ARM development tools, the linker scripts are often called scatter-loading files. If you are using the Keil Microcontroller Development Kit (MDK), the scatter-loading file is generated automatically from the memory layout window. You can use your own scatter file if you prefer.

After the executable image is generated, we can test it by downloading it to the flash memory or internal RAM of the microcontroller. The whole process can be quite easy; most development suites come with a user-friendly integrated development environment (IDE).

When working together with an in-circuit debugger (sometimes referred to as an in-circuit emulator [ICE], debug probe, or USB-JTAG adaptor), you can create a project, build your application, and download your embedded application to the microcontroller in a few steps (Figure 4.11).

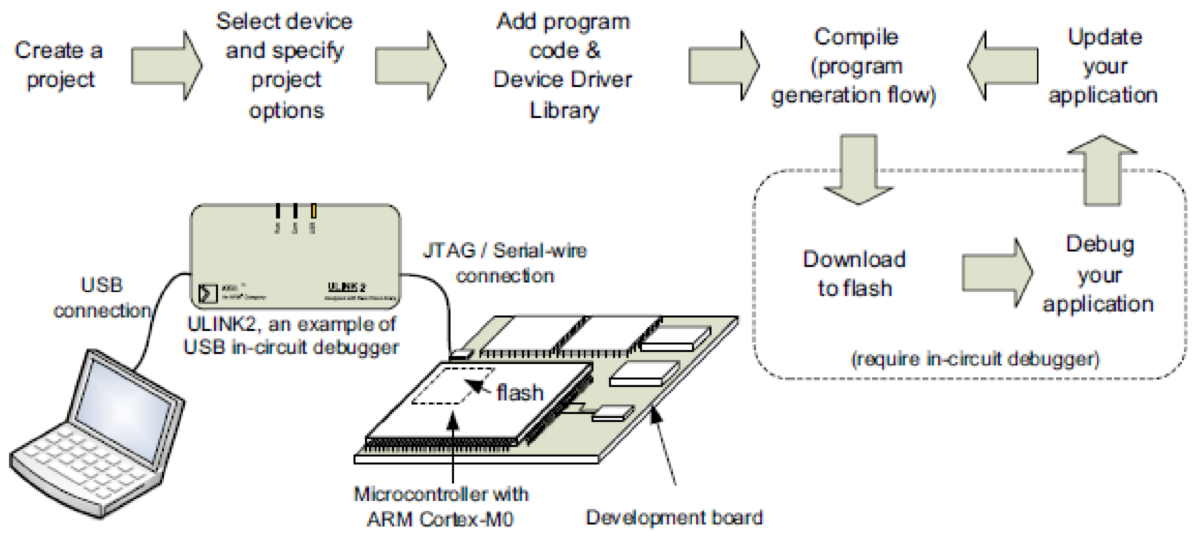

Create a project
Select device and specify project options
Add program code & Device Driver Library
Compile (program generation flow)
Update your application
Download to flash
Debug your application
(require in-circuit debugger)
USB connection
ULINK2, an example of USB in-circuit debugger
JTAG / Serial-wire connection
flash
Microcontroller with ARM Cortex-M0
Development board

Figure 4.11:
An example of development flow.

In many cases, an in-circuit debugger is needed to connect the debug host (personal computer) to the target board. The Keil U-LINK2 is one of the products available and can be used with Keil MDK and CodeSourcery gþþ (Figure 4.12).

The flash programming function can be carried out by the debugger software in the development suite (Figure 4.13) or in some cases by a flash programming utility downloadable from microcontroller vendor web site. The program can then be tested by running it on the microcontroller, and by connecting the debugger to the microcontroller, the program execution

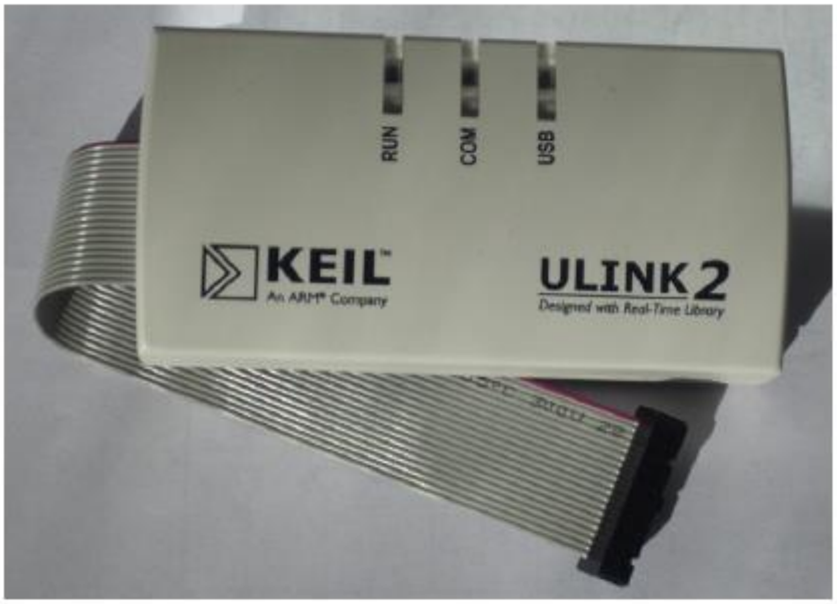


Figure 4.12:
ULINK 2 USB-JTAG adaptor.

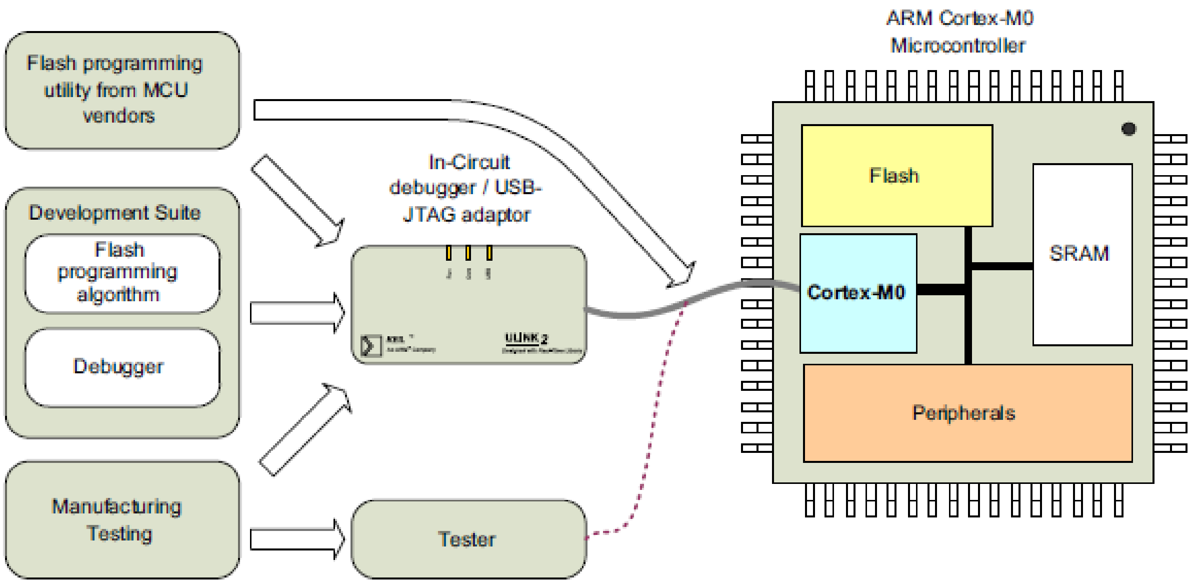

Flash programming utility from MCU vendors
Development Suite
Flash programming algorithm
Debugger
Manufacturing Testing
In-Circuit debugger / USB-JTAG adaptor
Tester
ARM Cortex-M0 Microcontroller
Flash
SRAM
Cortex-M0
Peripherals

Figure 4.13:
Various usages of the debug interface on the Cortex-M0 processor.

can be controlled and the operations can be observed. All these actions can be carried out via the debug interface of the Cortex-M0 processor.

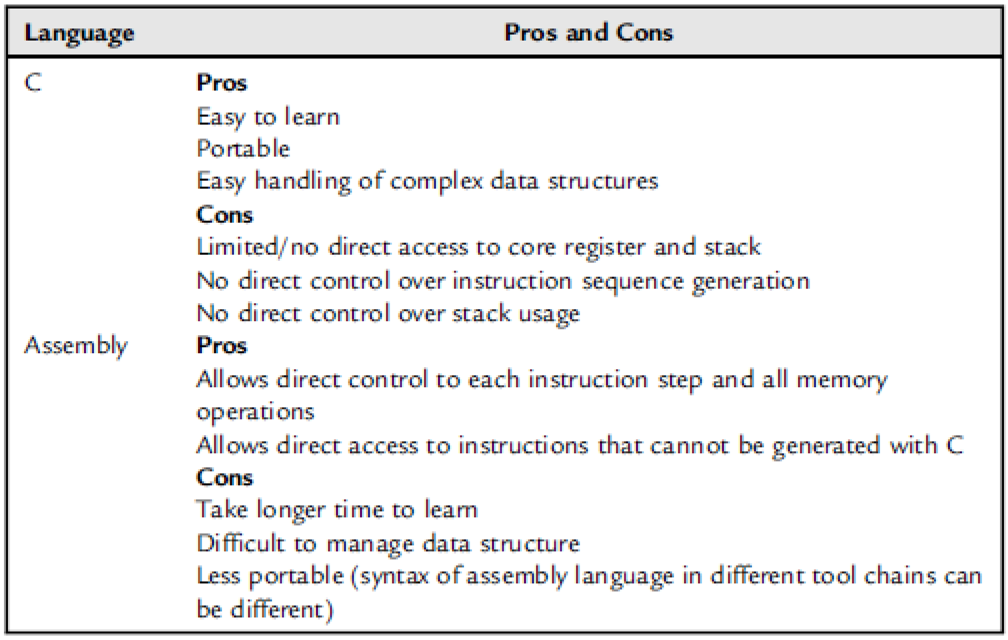
For simple program codes, we can also test the program using a simulator. This allows us to have full visibility to the program execution sequence and allows testing without actual hardware. Some development suites provide simulators that can also imitate peripheral behavior.

For example, Keil MDK provides device simulation for many ARM Cortex microcontrollers. Apart from the fact that different C compilers perform differently, different development suites also provide different C language extension features, as well as different syntax and directives in assembly programming. Chapters 5, 6, and 16 provide assembly syntax information for ARM development tools (including ARM RealView Development Suite [RVDS] and Keil MDK) and GNU C compilers. In addition, different development suites also provide different features in debug, utilities, and support different debug hardware product range.

3.1.4 C Programming and Assembly Programming

The Cortex-M0 processor can be programmed using C language, assembly language, or a mix of both. For beginners, C language is usually the best choice as it is easier to learn and most modern C compilers are very good at generating efficient code for the Cortex microcontrollers. Table 4.1 compares the use of C language and assembly language.

Table 4.1: Comparison between C Programming and Assembly Language Programming



Language	Pros and Cons
C	**Pros**
	Easy to learn
	Portable
	Easy handling of complex data structures
	Cons
	Limited/no direct access to core register and stack
	No direct control over instruction sequence generation
	No direct control over stack usage
Assembly	**Pros**
	Allows direct control to each instruction step and all memory operations
	Allows direct access to instructions that cannot be generated with C
	Cons
	Take longer time to learn
	Difficult to manage data structure
	Less portable (syntax of assembly language in different tool chains can be different)

Most C compilers provide workarounds to allow assembly code to be used within C program code. For example, ARM C compilers provide an Embedded Assembler so that assembly functions can be included in C program code easily. Similarly, most other C compilers provide an Inline Assembler for inlining assembly code within a C program file. However, the assembly syntax for using an Embedded Assembler and Inline Assembler are tool specific (not portable).

Note that the ARM C compiler has an Inline Assembler feature as well, but this is only available for 32-bit ARM instructions (e.g., for ARM7TDMI). Because the Cortex-M0 processor supports the Thumb instruction set only, the Embedded Assembler is used.

Some C compilers (including ARM C compilers in RealView Development Suite and Keil MDK) also provide intrinsic functions to allow special instructions to be used that cannot be generated using normal C code. Intrinsic functions are normally tool dependent. However, a tool-independent version of similar functions for Cortex-M0 is also available via the Cortex Microcontroller Software Interface Standard (CMSIS). This will be covered later in the chapter.

As Figure 4.10 shows, you can mix C and assembly code together in a project. This allows most parts of the program to be written in C, and some parts that cannot be handled in C can be written in assembly code. To do this, the interface between functions must be handled in a consistent manner to allow input parameters and returned results to be transferred correctly.

In ARM software development, the interface between functions is specified by a specification document called the ARM Architecture Procedure Call Standard (AAPCS, reference 4). The AAPCS is part of the Embedded Application Binary Interface (EABI). When using the Embedded Assembler, you should follow the guidelines set by the AAPCS. The AAPCS document and the EABI document can be downloaded from the ARM web site. More details in this area are covered in Chapter 16.

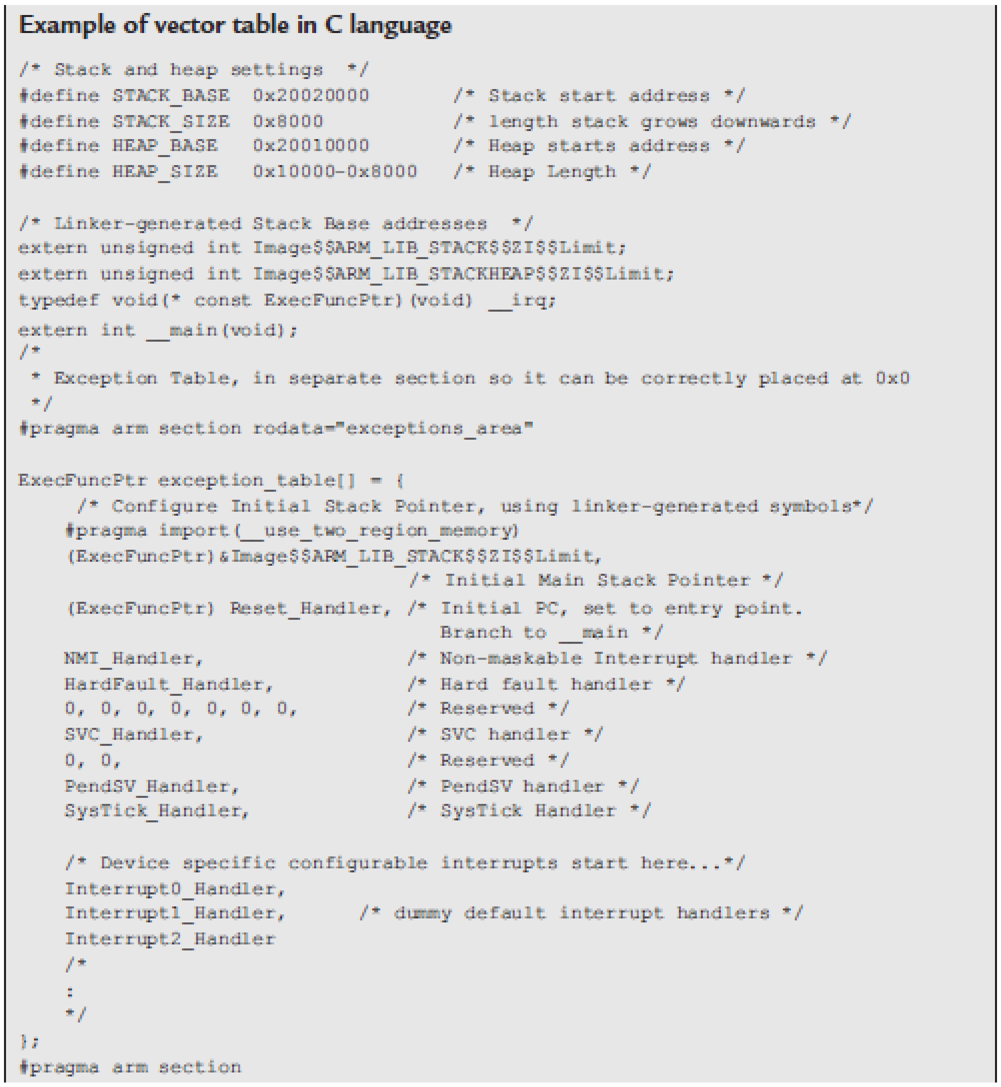
3.1.5 What Is in a Program Image?

At the end of Chapter 3 we covered the reset sequence of the Cortex-M0 and briefly introduced the vector table. Now we will look at the program image in more detail. A program image for the Cortex-M0 microcontroller often contains the following
components:

- Vector table
- C startup routine
- Program code (application code and data)
- C library code (program codes for C library functions, inserted at link time)

Vector Table

The vector table can be programmed in either C language or assembly language. The exact details of the vector table code are tool chain dependent because vector table entries require symbols created by the compiler and linker. For example, the initial stack pointer value is linked to stack region address symbols generated by the linker, and the reset vector is linked toC startup code address symbols, which are compiler dependent. For example, in the RealView Development Suite (RVDS), you can define the vector table with the following C code:



Example of vector table in C language

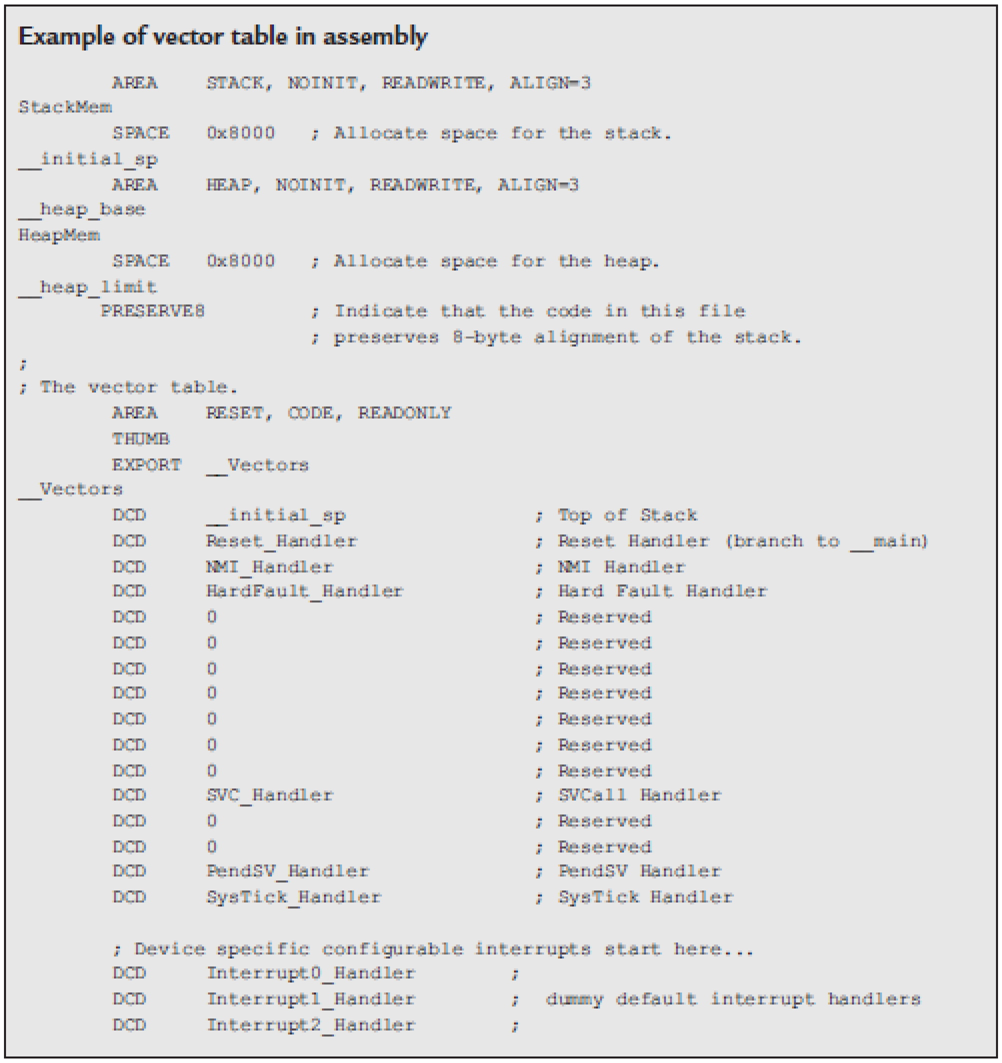
```
/* Stack and heap settings  */
#define STACK_BASE  0x20020000      /* Stack start address */
#define STACK_SIZE  0x8000          /* length stack grows downwards */
#define HEAP_BASE   0x20010000      /* Heap starts address */
#define HEAP_SIZE   0x10000-0x8000  /* Heap Length */

/* Linker-generated Stack Base addresses  */
extern unsigned int Image$$ARM_LIB_STACK$$ZI$$Limit;
extern unsigned int Image$$ARM_LIB_STACKHEAP$$ZI$$Limit;
typedef void(* const ExecFuncPtr)(void) __irq;
extern int __main(void);
/*
 * Exception Table, in separate section so it can be correctly placed at 0x0
 */
#pragma arm section rodata="exceptions_area"

ExecFuncPtr exception_table[] = {
     /* Configure Initial Stack Pointer, using linker-generated symbols*/
    #pragma import(__use_two_region_memory)
    (ExecFuncPtr)&Image$$ARM_LIB_STACK$$ZI$$Limit,
                                 /* Initial Main Stack Pointer */
    (ExecFuncPtr) Reset_Handler, /* Initial PC, set to entry point.
                                    Branch to __main */
    NMI_Handler,                 /* Non-maskable Interrupt handler */
    HardFault_Handler,           /* Hard fault handler */
    0, 0, 0, 0, 0, 0, 0,         /* Reserved */
    SVC_Handler,                 /* SVC handler */
    0, 0,                        /* Reserved */
    PendSV_Handler,              /* PendSV handler */
    SysTick_Handler,             /* SysTick Handler */

    /* Device specific configurable interrupts start here...*/
    Interrupt0_Handler,
    Interrupt1_Handler,      /* dummy default interrupt handlers */
    Interrupt2_Handler
    /*
    :
    */
};
#pragma arm section
```

Some development tools, including Keil MDK, create the vector table as part of the assembly startup code. In this case, the Define Constant Data (DCD) directive is used to create the vector table.



Example of vector table in assembly

```
        AREA    STACK, NOINIT, READWRITE, ALIGN=3
StackMem
        SPACE   0x8000    ; Allocate space for the stack.
__initial_sp
        AREA    HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
HeapMem
        SPACE   0x8000    ; Allocate space for the heap.
__heap_limit
       PRESERVE8          ; Indicate that the code in this file
                          ; preserves 8-byte alignment of the stack.
;
; The vector table.
        AREA    RESET, CODE, READONLY
        THUMB
        EXPORT  __Vectors
__Vectors
        DCD     __initial_sp                ; Top of Stack
        DCD     Reset_Handler               ; Reset Handler (branch to __main)
        DCD     NMI_Handler                 ; NMI Handler
        DCD     HardFault_Handler           ; Hard Fault Handler
        DCD     0                           ; Reserved
        DCD     0                           ; Reserved
        DCD     0                           ; Reserved
        DCD     0                           ; Reserved
        DCD     0                           ; Reserved
        DCD     0                           ; Reserved
        DCD     0                           ; Reserved
        DCD     SVC_Handler                 ; SVCall Handler
        DCD     0                           ; Reserved
        DCD     0                           ; Reserved
        DCD     PendSV_Handler              ; PendSV Handler
        DCD     SysTick_Handler             ; SysTick Handler

        ; Device specific configurable interrupts start here...
        DCD     Interrupt0_Handler        ;
        DCD     Interrupt1_Handler        ;  dummy default interrupt handlers
        DCD     Interrupt2_Handler        ;
```

You might notice that in both examples, the vector tables are given section names (exceptions_area in the C example and RESET in the assembly example). The vector table needs to be placed at the beginning of the system memory map (address 0x00000000). This can be done by a linker script or command line option, which requires a section name so that the contents of the vector table can be identified and mapped correctly by the linker.

In normal applications, the reset vector can point to the beginning of the C startup code. However, you can also define a reset handler to carry out additional initialization before branching to the C startup code.

C Startup Code

The C startup code is used to set up data memory such as global data variables. It also zero initializes part of the data memory for variables that are uninitialized at load time. For applications that use C functions like malloc(), the C startup code also needs to initialize the data variables controlling the heap memory. After this initialization, the C startup code branches to the beginning of the main() program.

The C startup code is inserted by the compiler/linker automatically and is tool chain specific; it might not be present if you are writing a program purely in assembly. For ARM compilers, the C startup code is labeled as "__main," whereas the startup code generated by GNU C compilers is normally labeled as "_start."

Program Code

The instructions generated from your application program code carry out the tasks you specify. Apart from the instruction sequence, there are also various types of data:

• Initial values of variables. Local variables in functions or subroutines need to be initialized, and these initial values are set up during program execution.

• Constants in program code. Constant data are used in application codes in many ways: data values, addresses of peripheral registers, constant strings, and so on. These data are sometimes grouped together within the program images as a number of data blocks called literal pools.

• Some applications can also contain additional constant data like lookup tables and graphics image data (e.g., bit map) that are merged into the program images.

C Library Code

C library code is injected in to the program image by the linker when certain C/Cþþ functions are used. In addition, C library code can also be included because of data processing tasks such as floating point operations and divide. The Cortex-M0 does not have a divide instruction, and this function typically needs to be carried out by a C library divide function.

Some development tools offer various versions of C libraries for different purposes. For example, in Keil MDK or ARM RVDS there is an option to use a special version of C library called Microlib. The Microlib is targeted for microcontrollers and is very small, but it does not offer all features of the standard C library. In embedded applications that do not require high data processing capability and have tight program memory requirement, the Microlib offers a good way to reduce code size.

Depending on the application, C library code might not be present in simple C applications (no C library function calls) or pure assembly language projects.

Apart from the vector table, which must be placed at the beginning of the memory map, there are no other constraints on the placement of the rest of the elements inside a program image. In some cases, if the layout of the items in the program memory is important, the layout of the program image can be controlled by a linker script.

Data in RAM

Like program ROM, the RAM of microcontrollers is used in different ways. Typically, the RAM usage is divided into data, stack, and heap regions (Figure 4.14).

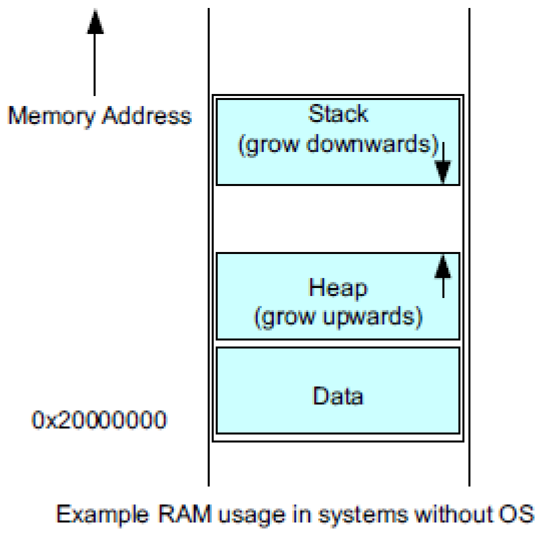


Figure 4.14:
Example of RAM usage in single task systems (without OS).

For microcontroller systems with an embedded OS (e.g., mClinux) or RTOS (e.g., Keil RTX), the stacks for each task are separate. Some OSs allow a user-defined stack for tasks that require larger stack memory. Some OSs divide the RAM into a number of segments, and each segment is assigned to a task, each containing individual data, stack, and heap regions (Figure 4.15).

So what is stored inside these data, stack, and heap regions?

• Data. Data stored in the bottom of RAM usually contain global variables and static variables. (Note: Local variables can be spilled onto the stack to reduce RAM usage. Local variables that belong to a function that is not in use do not take up memory space.)

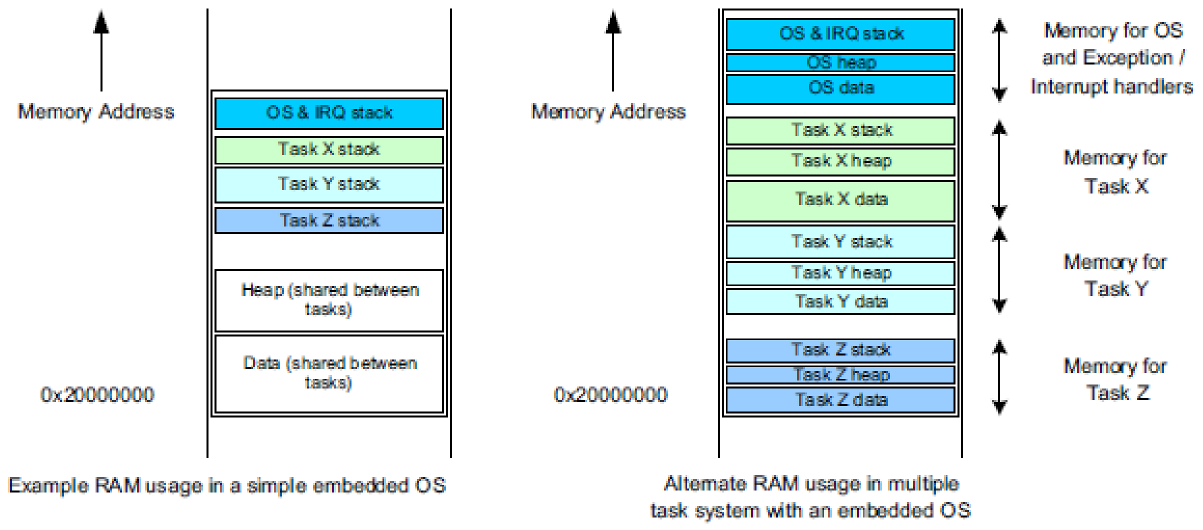


Figure 4.15:
Example of RAM usage in multiple task systems (with an OS).

• Stack. The role of stack memory includes temporary data storage (normal stack PUSH and POP operations), memory space for local variables, parameter passing in function calls, register saving during an exception sequence, and so on. The Thumb instruction set is very efficient in handling data accesses that use a stack pointer (SP) related addressing mode and allows data in the stack memory to be accessed with very low instruction overhead.

• Heap. The heap memory is used by C functions that dynamically reserve memory space, like “alloc(),” “malloc(),” and other function calls that use these functions. To allow these functions to allocate memory correctly, the C startup code needs to initialize the heap memory and its control variables.

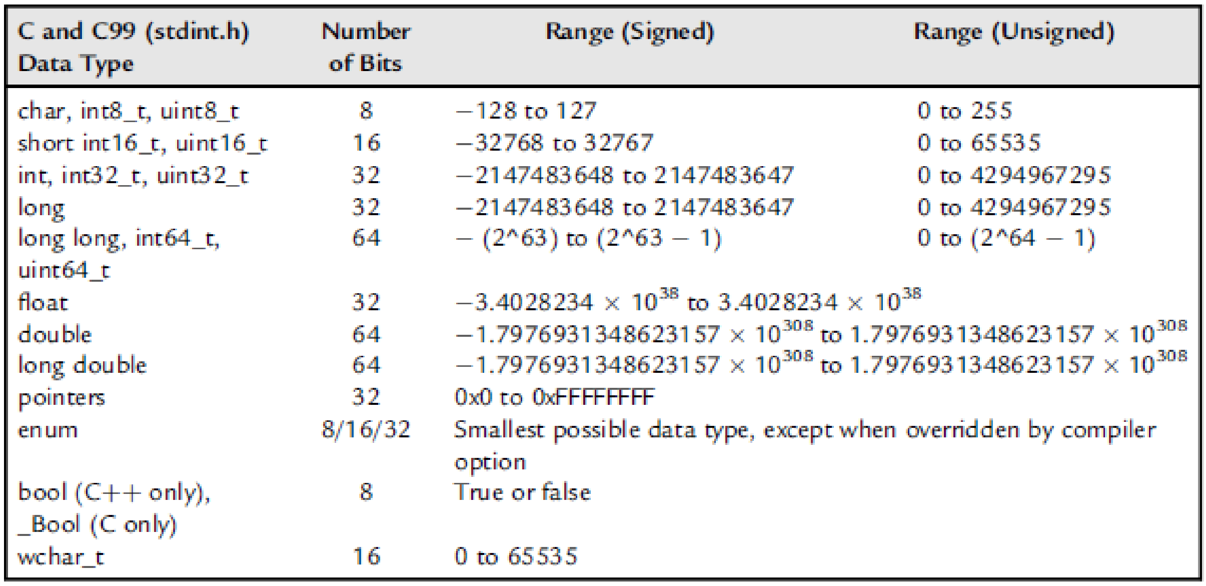
Usually, the stack is placed at the top of the memory space and the heap memory is placed underneath. This gives the best flexibility for the RAM usage. In an OS environment, there can be multiple regions of data, stack, and heap in the RAM.

3.1.6 C Programming: Data Types

The C language supports a number of “standard” data types. However, the implementation of data type can be processor architecture dependent and C compiler dependent. In ARM processors including the Cortex-M0, the data type implementations shown in Table 4.2 are supported by all C compilers.

When porting applications from other processor architectures to ARM processors, if the data types have different sizes, it might be necessary to modify the C program code in order to

Table 4.2: Size of Data Types in Cortex-M Processors



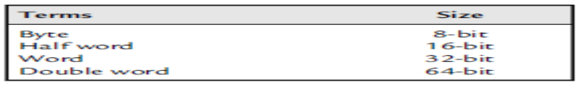
C and C99 (stdint.h) Data Type	Number of Bits	Range (Signed)	Range (Unsigned)
char, int8_t, uint8_t	8	−128 to 127	0 to 255
short int16_t, uint16_t	16	−32768 to 32767	0 to 65535
int, int32_t, uint32_t	32	−2147483648 to 2147483647	0 to 4294967295
long	32	−2147483648 to 2147483647	0 to 4294967295
long long, int64_t, uint64_t	64	− (2^63) to (2^63 − 1)	0 to (2^64 − 1)
float	32	$-3.4028234 \times 10^{38}$ to 3.4028234×10^{38}	
double	64	$-1.797693134862315 7 \times 10^{308}$ to $1.797693134862315 7 \times 10^{308}$	
long double	64	$-1.797693134862315 7 \times 10^{308}$ to $1.797693134862315 7 \times 10^{308}$	
pointers	32	0x0 to 0xFFFFFFFF	
enum	8/16/32	Smallest possible data type, except when overridden by compiler option	
bool (C++ only), _Bool (C only)	8	True or false	
wchar_t	16	0 to 65535	

ensure the program operates correctly. More details on porting software from 8-bit and 16-bit architecture are covered in Chapter 21.

In Cortex-M0 programming, the data variables stored in memory need to be stored at an address location that is a multiple of its size. More details on this area are covered in Chapter 7 (the data alignment section).

In ARM programming, we also refer to data size as word, half word, and byte (Table 4.3).

Table 4.3: Data Size Definition in ARM Processor



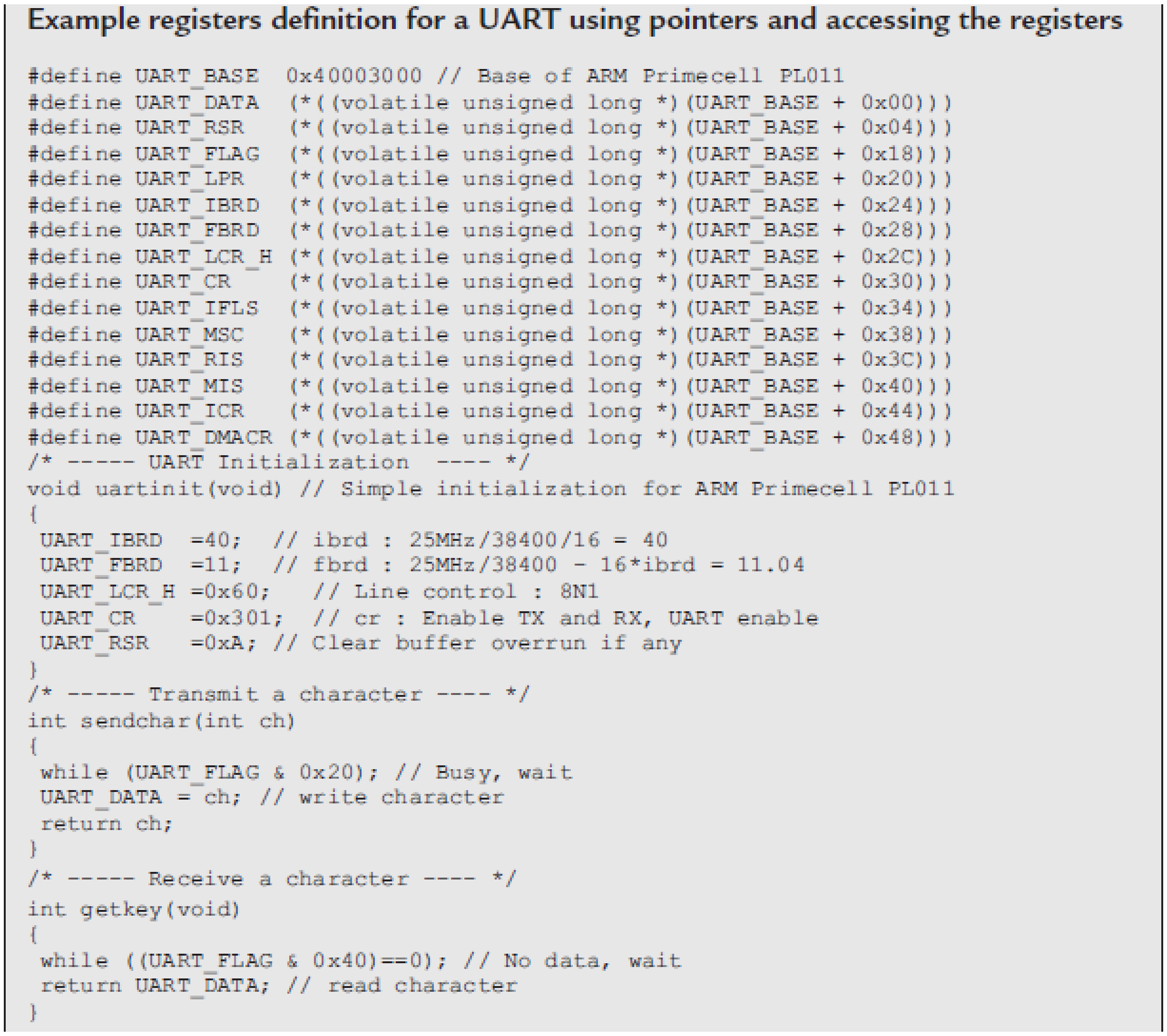
Terms	Size
Byte	8-bit
Half word	16-bit
Word	32-bit
Double word	64-bit

These terms are commonly found in ARM documentation, such as in the instruction set details.

3.1.7 *Accessing Peripherals in C*

Apart from data variables, a C program for microcontroller applications normally needs to access peripherals. In ARM Cortex-M0 microcontrollers, peripheral registers are memory mapped and can be accessed by memory pointers. In most cases, you can use the device drivers provided by the microcontroller vendors to simplify the software development task and make it easier to port software between different microcontrollers. If it is necessary to access the peripheral registers directly, the following methods can be used.

In simple cases of accessing a few registers, you can define a peripheral register as a pointer as
follows:

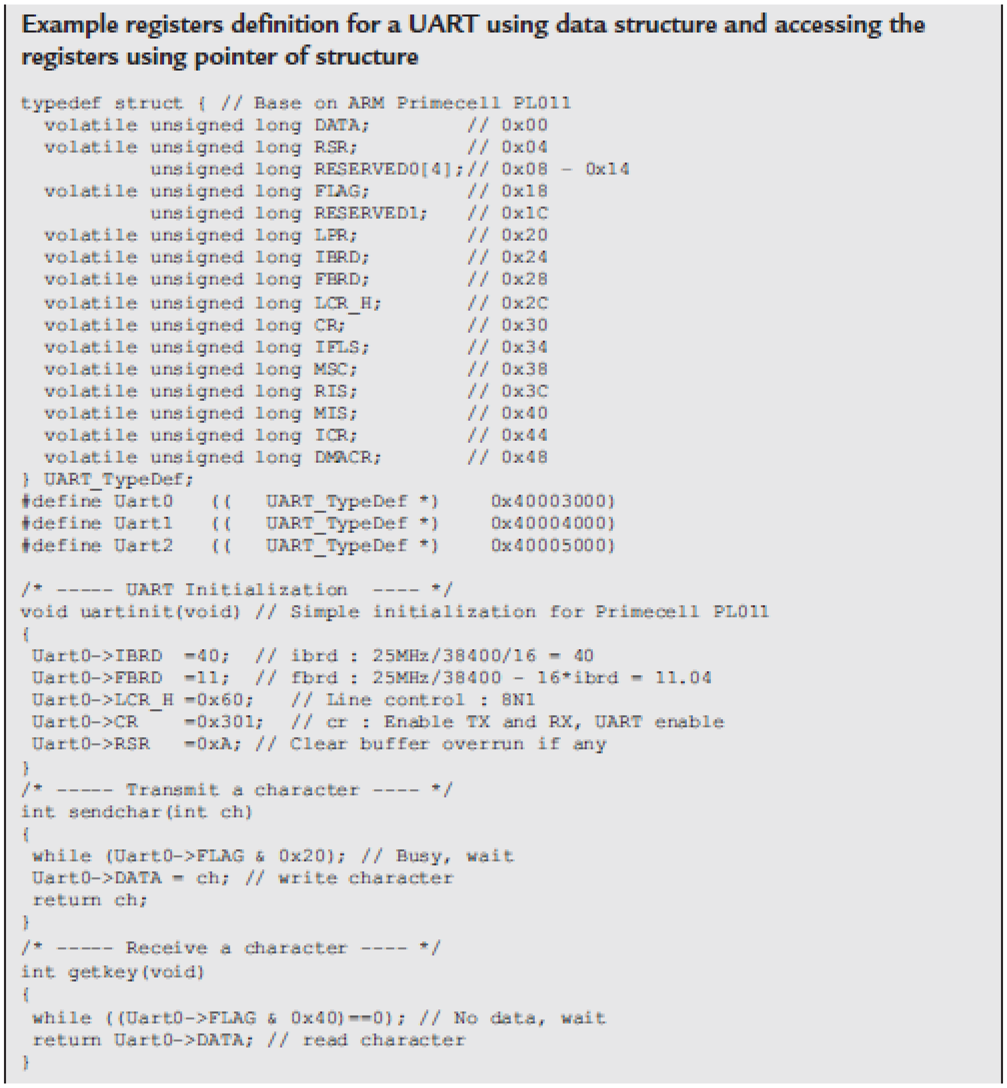


Example registers definition for a UART using pointers and accessing the registers

```
#define UART_BASE  0x40003000 // Base of ARM Primecell PL011
#define UART_DATA  (*((volatile unsigned long *)(UART_BASE + 0x00)))
#define UART_RSR   (*((volatile unsigned long *)(UART_BASE + 0x04)))
#define UART_FLAG  (*((volatile unsigned long *)(UART_BASE + 0x18)))
#define UART_LPR   (*((volatile unsigned long *)(UART_BASE + 0x20)))
#define UART_IBRD  (*((volatile unsigned long *)(UART_BASE + 0x24)))
#define UART_FBRD  (*((volatile unsigned long *)(UART_BASE + 0x28)))
#define UART_LCR_H (*((volatile unsigned long *)(UART_BASE + 0x2C)))
#define UART_CR    (*((volatile unsigned long *)(UART_BASE + 0x30)))
#define UART_IFLS  (*((volatile unsigned long *)(UART_BASE + 0x34)))
#define UART_MSC   (*((volatile unsigned long *)(UART_BASE + 0x38)))
#define UART_RIS   (*((volatile unsigned long *)(UART_BASE + 0x3C)))
#define UART_MIS   (*((volatile unsigned long *)(UART_BASE + 0x40)))
#define UART_ICR   (*((volatile unsigned long *)(UART_BASE + 0x44)))
#define UART_DMACR (*((volatile unsigned long *)(UART_BASE + 0x48)))
/* ----- UART Initialization  ---- */
void uartinit(void) // Simple initialization for ARM Primecell PL011
{
 UART_IBRD  =40;  // ibrd : 25MHz/38400/16 = 40
 UART_FBRD  =11;  // fbrd : 25MHz/38400 - 16*ibrd = 11.04
 UART_LCR_H =0x60;   // Line control : 8N1
 UART_CR    =0x301;  // cr : Enable TX and RX, UART enable
 UART_RSR   =0xA; // Clear buffer overrun if any
}
/* ----- Transmit a character ---- */
int sendchar(int ch)
{
 while (UART_FLAG & 0x20); // Busy, wait
 UART_DATA = ch; // write character
 return ch;
}
/* ----- Receive a character ---- */
int getkey(void)
{
 while ((UART_FLAG & 0x40)==0); // No data, wait
 return UART_DATA; // read character
}
```

This solution is fine for simple applications. However, when multiple units of the same peripherals are available in the system, defining registers will be required for each of these peripherals, which can make code maintenance difficult. In addition, defining each register as a separated pointer might result in larger program size, as each register access requires a 32-bit address constant to be stored in the program flash memory.

To simplify the code, we can define the peripheral register set as a data structure and define the peripheral as a memory pointer to this data structure.



Example registers definition for a UART using data structure and accessing the registers using pointer of structure

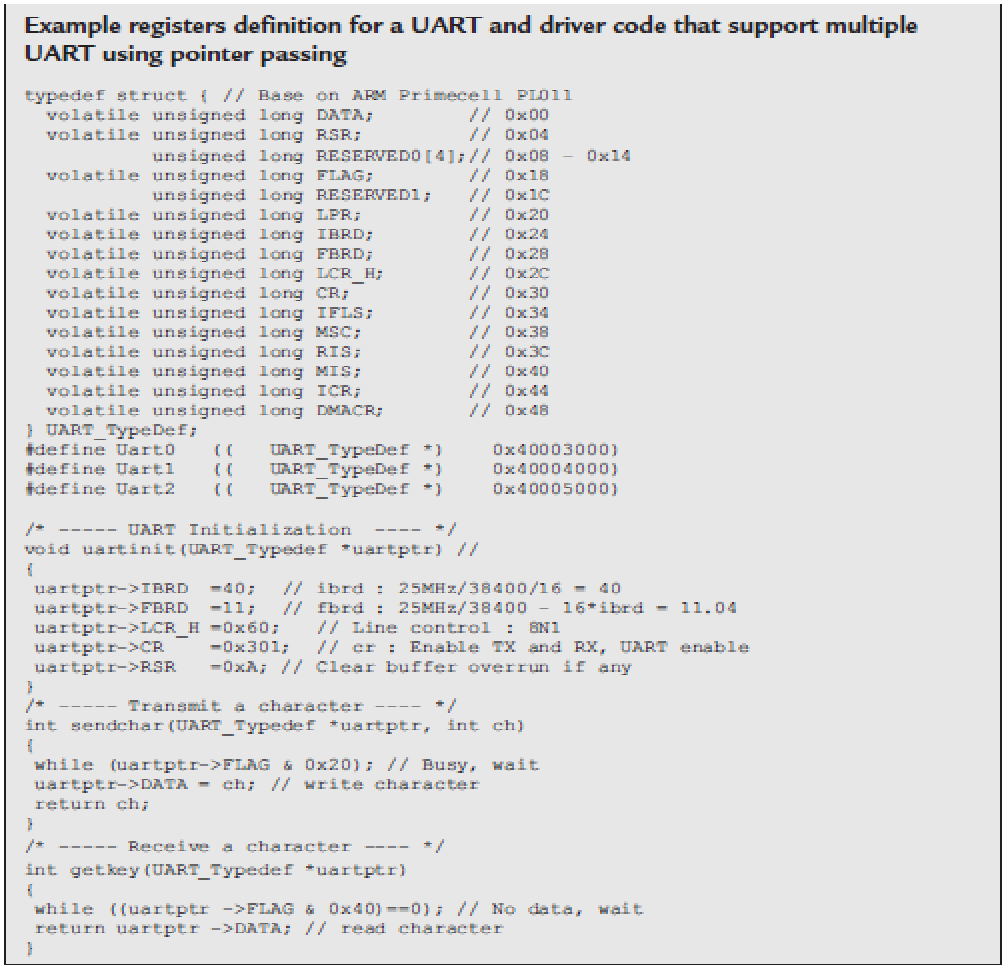
```
typedef struct { // Base on ARM Primecell PL011
  volatile unsigned long DATA;         // 0x00
  volatile unsigned long RSR;          // 0x04
           unsigned long RESERVED0[4];// 0x08 - 0x14
  volatile unsigned long FLAG;         // 0x18
           unsigned long RESERVED1;    // 0x1C
  volatile unsigned long LPR;          // 0x20
  volatile unsigned long IBRD;         // 0x24
  volatile unsigned long FBRD;         // 0x28
  volatile unsigned long LCR_H;        // 0x2C
  volatile unsigned long CR;           // 0x30
  volatile unsigned long IFLS;         // 0x34
  volatile unsigned long MSC;          // 0x38
  volatile unsigned long RIS;          // 0x3C
  volatile unsigned long MIS;          // 0x40
  volatile unsigned long ICR;          // 0x44
  volatile unsigned long DMACR;        // 0x48
} UART_TypeDef;
#define Uart0   ((   UART_TypeDef *)     0x40003000)
#define Uart1   ((   UART_TypeDef *)     0x40004000)
#define Uart2   ((   UART_TypeDef *)     0x40005000)

/* ----- UART Initialization  ---- */
void uartinit(void) // Simple initialization for Primecell PL011
{
 Uart0->IBRD  =40;  // ibrd : 25MHz/38400/16 = 40
 Uart0->FBRD  =11;  // fbrd : 25MHz/38400 - 16*ibrd = 11.04
 Uart0->LCR_H =0x60;   // Line control : 8N1
 Uart0->CR    =0x301;  // cr : Enable TX and RX, UART enable
 Uart0->RSR   =0xA; // Clear buffer overrun if any
}
/* ----- Transmit a character ---- */
int sendchar(int ch)
{
 while (Uart0->FLAG & 0x20); // Busy, wait
 Uart0->DATA = ch; // write character
 return ch;
}
/* ----- Receive a character ---- */
int getkey(void)
{
 while ((Uart0->FLAG & 0x40)==0); // No data, wait
 return Uart0->DATA; // read character
}
```

In this example, the Integer Baud Rate Divider (IBRD) register for UART #0 is accessed by the symbol Uart0->IBRD, and the same register for UART #1 is accessed by Uart1->IBRD.

With this arrangement, the same register data structure for the peripheral can be shared between multiple instantiations, making code maintenance easier. In addition, the compiled code could be smaller because of the reduced requirement of immediate data storage.

With further modification, a function developed for the peripherals can be shared between multiple units by passing the base pointer to the function:



Example registers definition for a UART and driver code that support multiple UART using pointer passing

```
typedef struct { // Base on ARM Primecell PL011
  volatile unsigned long DATA;          // 0x00
  volatile unsigned long RSR;           // 0x04
           unsigned long RESERVED0[4];// 0x08 - 0x14
  volatile unsigned long FLAG;          // 0x18
           unsigned long RESERVED1;     // 0x1C
  volatile unsigned long LPR;           // 0x20
  volatile unsigned long IBRD;          // 0x24
  volatile unsigned long FBRD;          // 0x28
  volatile unsigned long LCR_H;         // 0x2C
  volatile unsigned long CR;            // 0x30
  volatile unsigned long IFLS;          // 0x34
  volatile unsigned long MSC;           // 0x38
  volatile unsigned long RIS;           // 0x3C
  volatile unsigned long MIS;           // 0x40
  volatile unsigned long ICR;           // 0x44
  volatile unsigned long DMACR;         // 0x48
} UART_TypeDef;
#define Uart0   ((   UART_TypeDef *)    0x40003000)
#define Uart1   ((   UART_TypeDef *)    0x40004000)
#define Uart2   ((   UART_TypeDef *)    0x40005000)

/* ----- UART Initialization  ---- */
void uartinit(UART_Typedef *uartptr) //
{
 uartptr->IBRD  =40;  // ibrd : 25MHz/38400/16 = 40
 uartptr->FBRD  =11;  // fbrd : 25MHz/38400 - 16*ibrd = 11.04
 uartptr->LCR_H =0x60;   // Line control : 8N1
 uartptr->CR    =0x301;  // cr : Enable TX and RX, UART enable
 uartptr->RSR   =0xA; // Clear buffer overrun if any
}
/* ----- Transmit a character ---- */
int sendchar(UART_Typedef *uartptr, int ch)
{
 while (uartptr->FLAG & 0x20); // Busy, wait
 uartptr->DATA = ch; // write character
 return ch;
}
/* ----- Receive a character ---- */
int getkey(UART_Typedef *uartptr)
{
 while ((uartptr ->FLAG & 0x40)==0); // No data, wait
 return uartptr ->DATA; // read character
}
```

In most cases, peripheral registers are defined as 32-bit words. This is because most peripherals are connected to a peripheral bus (using APB protocol; see Chapter 7) that handles all transfers as 32 bit. Some peripherals might be connected to the processor bus (with AHB protocol that supports various transfer sizes; see Chapter 7). In such cases, the registers might be accessed in other transfer sizes. Please refer to the user manual of the microcontroller to determine the supported transfer size for each peripheral.

Note that when defining memory pointers for peripheral accesses, the “volatile” keyword should be used.

3.1.8 *Cortex Microcontroller Software Interface Standard (CMSIS)*

3.1.8.1 Introduction to CMSIS

As the complexity of embedded systems increase, the compatibility and reusability of software code becomes more important. Having reusable software often reduces development time for subsequent projects and hence speeds up time to market, and software compatibility helps the use of third-party software components. For example, an embedded system project might involve the following software components:

• Software from in-house software developers

• Software reused from other projects

• Device driver libraries from microcontroller vendors

• Embedded OS

• Other third-party software products like a communication protocol stack and codec (compressor/decompressor)

The use of the third-party software components is becoming more and more common.With all these software components being used in one project, compatibility is becoming critical for many large-scale software projects. To allow a high level of compatibility between these software products and improve software portability, ARM worked with various microcontroller vendors and software solution providers to develop the CMSIS, a common software framework covering most Cortex-M processors and Cortex-M microcontroller products (Figure 4.16).

The CMSIS is implemented as part of device driver library from microcontroller vendors. It provides a standardized software interface to the processor features like NVIC control and system control functions. Many of these processors feature access functions are available in CMSIS for the Cortex-M0, Cortex-M3 and Cortex-M4, allowing easy software porting between these processors.

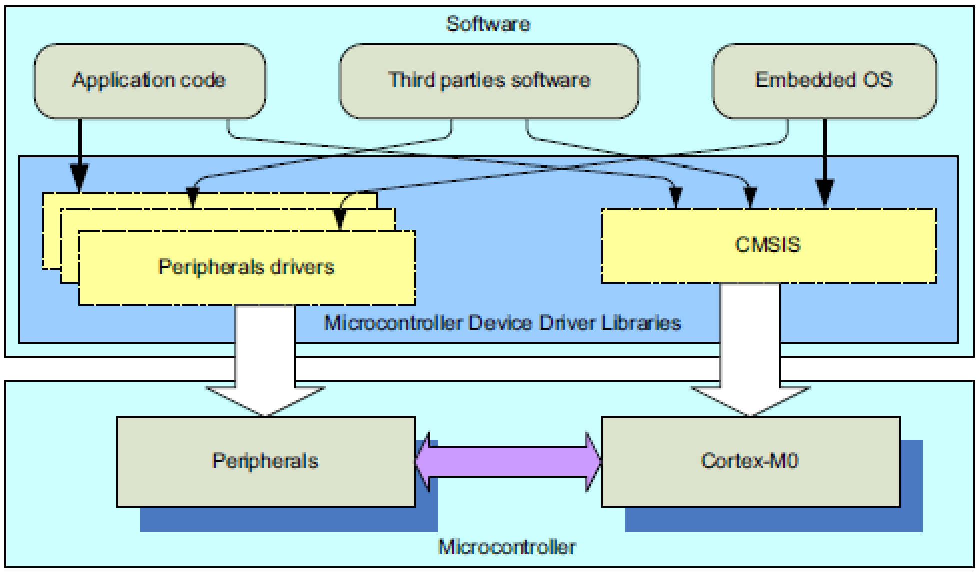


Figure 4.16:
CMSIS provides standardized access functions for processor features.

The CMSIS is standardized across multiple microcontroller vendors and is supported by multiple C compiler vendors. For example, it can be used with the Keil MDK, the ARM RealView Development Suite (RVDS), the IAR Embedded Workbench, the TASKING compiler, and various GNU-based C compiler suites including the CodeSourcery Gþþ tool chain.

3.1.8.2 What is standardized in CMSIS

The CMSIS standardized the following areas for embedded software:

• Standardized access functions for accessing NVIC, System Control Block (SCB), and System Tick timer (SysTick) such as interrupt control and SysTick initialization. These functions will be covered in various chapters of this book and in the CMSIS functions quick reference in Appendix C.

• Standardized register definitions for NVIC, SCB, and SysTick registers. For best software portability, we should use the standardized access functions. However, in some cases we need to directly access the registers in NVIC, SCB, or the SysTick. In such cases, the standardized register definitions help the software to be more portable.

• Standardized functions for accessing special instructions in Cortex-M microcontrollers. Some instructions on the Cortex-M microcontroller cannot be generated by normal C code. If they are needed, they can be generated by these functions provided in CMSIS. Otherwise, users will have to use intrinsic functions provided by the C compiler or embedded/inline assembly language, which are tool chain specific and less portable.

• Standardized names for system exceptions handlers. An embedded OS often requires system exceptions. By having standardized system exception handler names, supporting different device driver libraries in an embedded OS is much easier.

• Standardized name for the system initialization function. The common system initialization function "void SystemInit(void)" makes it easier for software developers to set up their system with minimum effort.

• Standardize variable for clock speed information. A standardized software variable called "SystemFreq" (CMSIS v1.00 to v1.20) or "SystemCoreClock" (CMSIS v1.30 or newer). This is used to determine the processor clock frequency.

The CMSIS also provides the following:

• A common platform for device driver librariesdeach device driver library has the same look and feel, making it easier for beginners to learn and making it easier for software
porting.

• In future release of CMSIS, it could also provide a set of common communication access functions so that middleware that has been developed can be reused on different devices without porting.

The CMSIS is developed to ensure compatibility for the basic operations. Microcontroller vendors can add functions to enhance their software solution so that CMSIS does not restrict the functionality and the capability of the embedded products.

3.1.8.3 Organization of CMSIS

The CMSIS is divided into multiple layers:

Core Peripheral Access Layer

• Name definitions, address definitions, and helper functions to access core registers and core peripherals like the NVIC, SCB, and SysTick

Middleware Access Layer (work in progress)

• Common method to access peripherals for typical embedded systems

• Targeted at communication interfaces including UART, Ethernet, and SPI

• Allows embedded software to be used on any Cortex microcontrollers that support the required communication interface

Device Peripheral Access Layer (MCU specific)

• Register name definitions, address definitions, and device driver code to access peripherals

Access Functions for Peripherals (MCU specific)

• Optional helper functions for peripherals

The role of these layers is summarized in Figure 4.17.

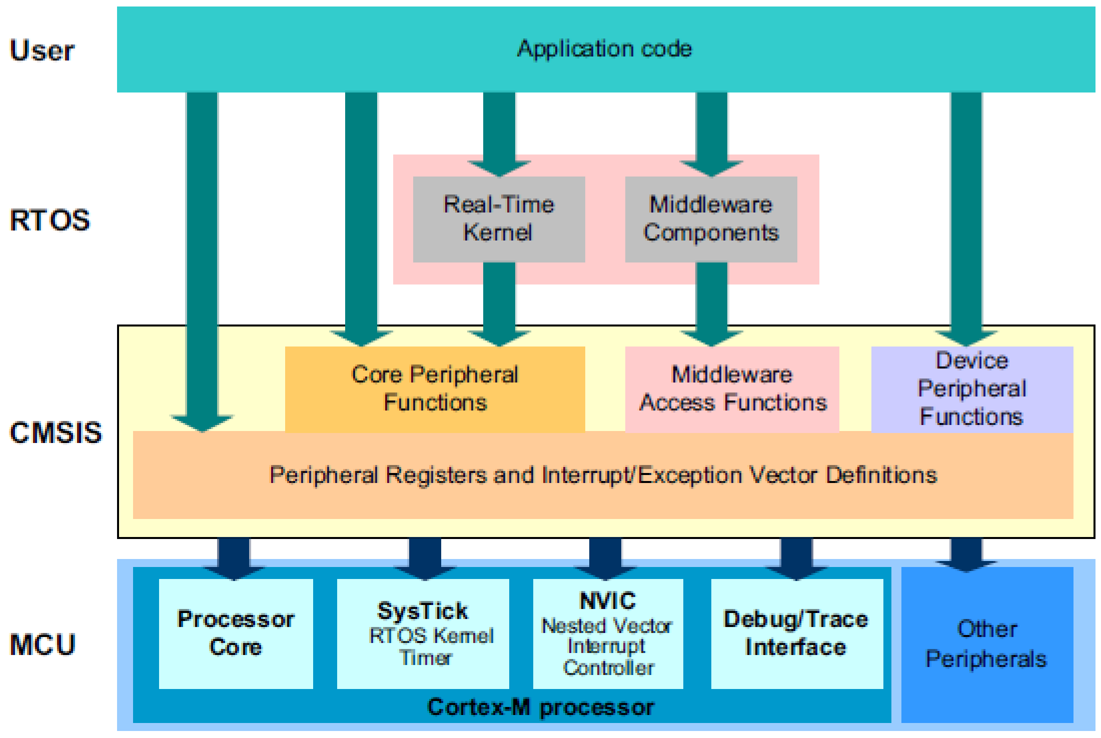

User
Application code
RTOS
Real-Time Kernel
Middleware Components
CMSIS
Core Peripheral Functions
Middleware Access Functions
Device Peripheral Functions
Peripheral Registers and Interrupt/Exception Vector Definitions
MCU
Processor Core
SysTick RTOS Kernel Timer
NVIC Nested Vector Interrupt Controller
Debug/Trace Interface
Other Peripherals
Cortex-M processor

Figure 4.17:
CMSIS structure.

3.1.8.4 Using CMSIS

The CMSIS is an integrated part of the device driver package provided by the microcontroller vendors. If you are using the device driver libraries for software development, you are already using the CMSIS. If you are not using device driver libraries from microcontroller vendors, you can still use CMSIS by downloading the CMSIS package from

OnARM web site (www.onarm. com), unpacking the files, and adding the required files for your project.

For C program code, normally you only need to include one header file provided in the device driver library from your microcontroller vendor. This header file then pulls in the all the required header files for CMSIS features as well as peripheral drivers. You also need to include the CMSIS-compliant startup code, which can be either in C or assembly code. CMSIS provides various versions of startup code customized for different tool chains.

Figure 4.18 shows a simple project setup using the CMSIS package. The name of some the files depends on the actual microcontroller device name (indicated as <device> in Figure 4.18). When you use the header file provided in the device driver library, it automatically includes the other required header files for you (Table 4.4).

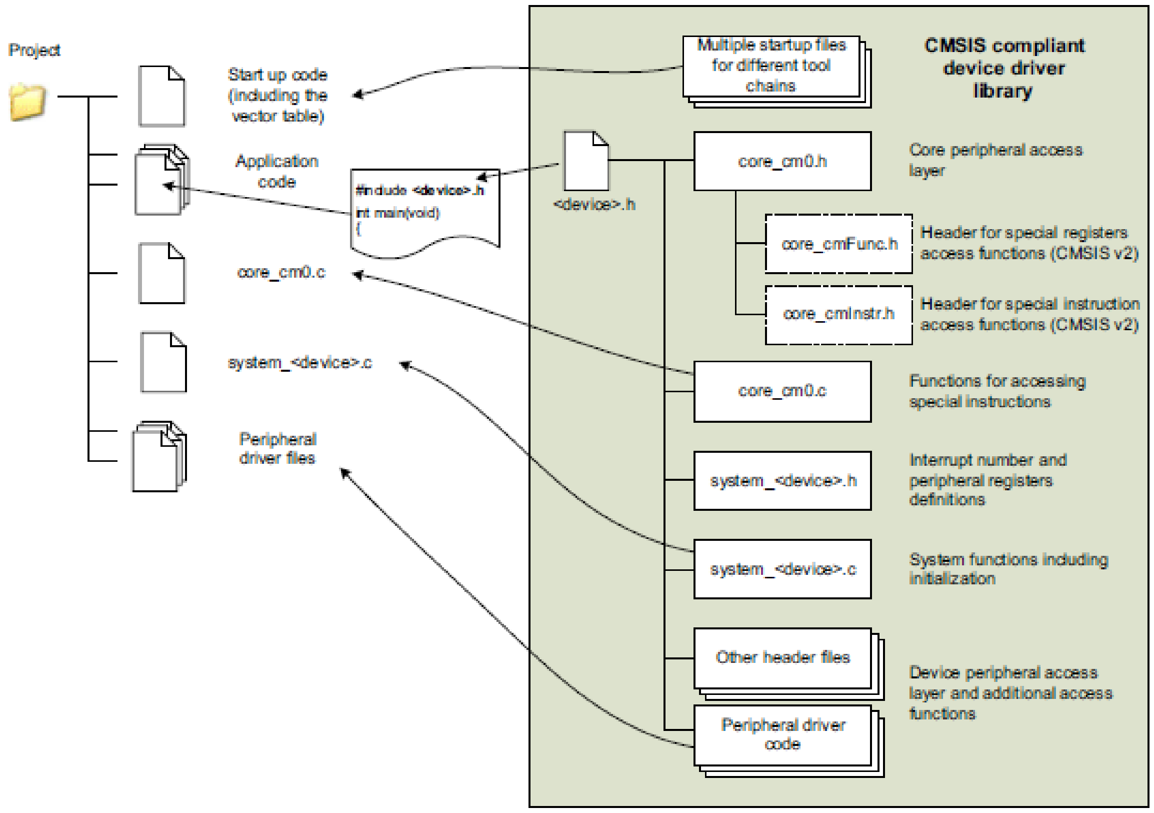

Project
Start up code (including the vector table)
Application code
#include <device>.h
int main(void)
{
core_cm0.c
system_<device>.c
Peripheral driver files
<device>.h
Multiple startup files for different tool chains
CMSIS compliant device driver library
core_cm0.h
Core peripheral access layer
core_cmFunc.h
Header for special registers access functions (CMSIS v2)
core_cmInstr.h
Header for special instruction access functions (CMSIS v2)
core_cm0.c
Functions for accessing special instructions
system_<device>.h
Interrupt number and peripheral registers definitions
system_<device>.c
System functions including initialization
Other header files
Peripheral driver code
Device peripheral access layer and additional access functions

Figure 4.18:
Using CMSIS in a project.

Figure 4.19 shows a simple example of using CMSIS.

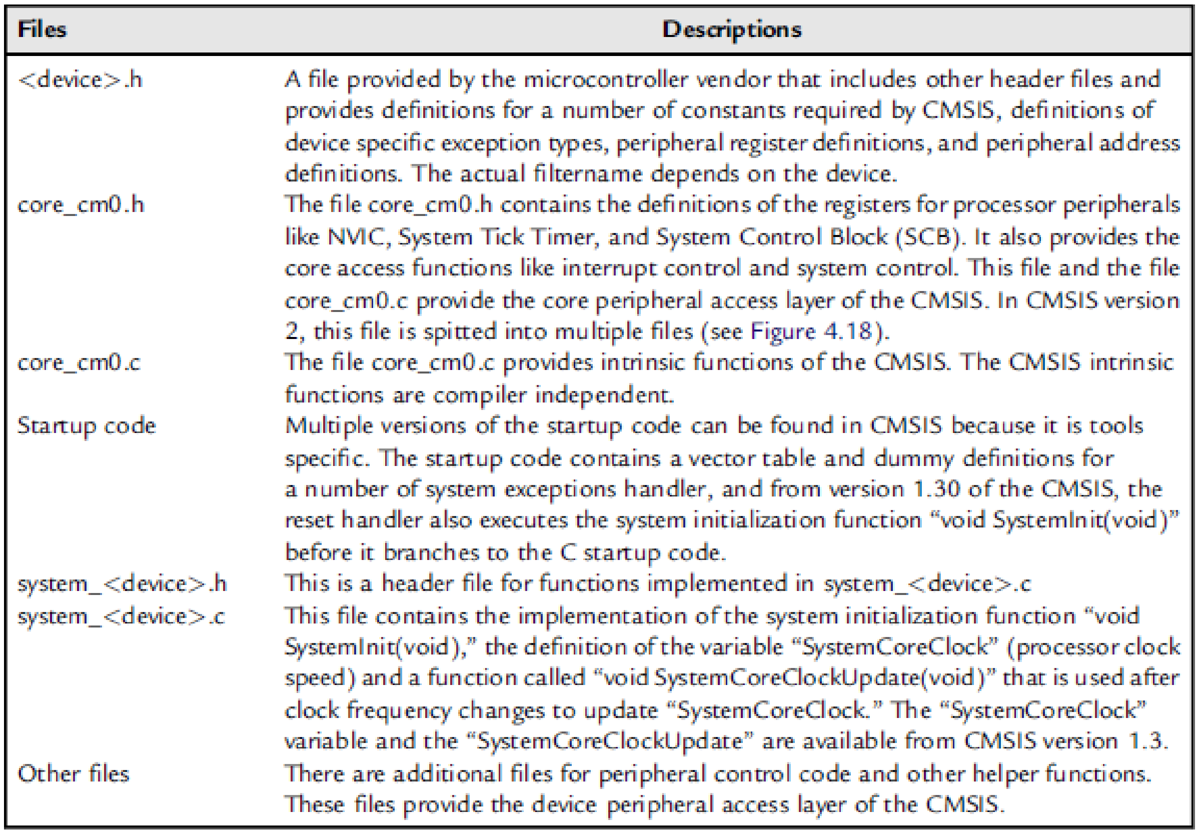
Typically, information and examples of using CMSIS can be found in the device driver libraries package from your microcontroller vendor. There are also some simple examples of using the CMSIS in the CMSIS package on the OnARM web site (www.onarm.com).

3.1.8.5 Benefits of CMSIS

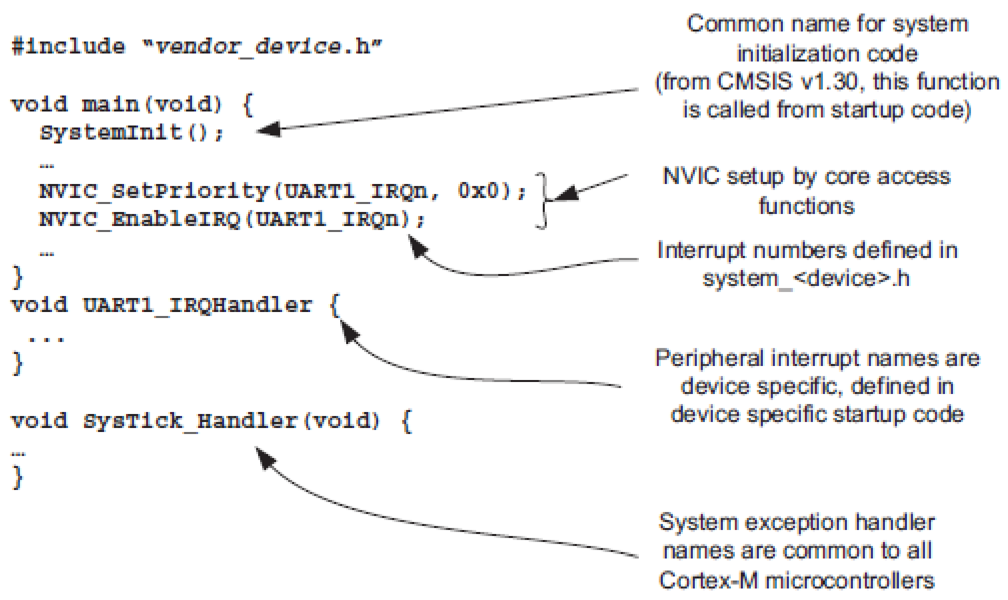
For most users, CMSIS offer a number of key advantages.

Porting of applications from one Cortex-M microcontroller to another Cortex-M microcontroller is much easier. For example, most of the interrupt control functions are available for Cortex-M0, Cortex-M3, and Cortex-M4 (only a few functions for Cortex-M3/M4 are not available for Cortex-M0 because of the extra functionality of the Cortex-M3/M4 processors). This makes it straightforward to reuse the same application code for a different project. You can migrate a Cortex-M3 project to Cortex-M0 for lower cost, or you can move a Cortex-M0 project to Cortex-M3 if higher performance is required.

Table 4.4: Files in CMSIS



Files	Descriptions
<device>.h	A file provided by the microcontroller vendor that includes other header files and provides definitions for a number of constants required by CMSIS, definitions of device specific exception types, peripheral register definitions, and peripheral address definitions. The actual filtername depends on the device.
core_cm0.h	The file core_cm0.h contains the definitions of the registers for processor peripherals like NVIC, System Tick Timer, and System Control Block (SCB). It also provides the core access functions like interrupt control and system control. This file and the file core_cm0.c provide the core peripheral access layer of the CMSIS. In CMSIS version 2, this file is spitted into multiple files (see Figure 4.18).
core_cm0.c	The file core_cm0.c provides intrinsic functions of the CMSIS. The CMSIS intrinsic functions are compiler independent.
Startup code	Multiple versions of the startup code can be found in CMSIS because it is tools specific. The startup code contains a vector table and dummy definitions for a number of system exceptions handler, and from version 1.30 of the CMSIS, the reset handler also executes the system initialization function "void SystemInit(void)" before it branches to the C startup code.
system_<device>.h	This is a header file for functions implemented in system_<device>.c
system_<device>.c	This file contains the implementation of the system initialization function "void SystemInit(void)," the definition of the variable "SystemCoreClock" (processor clock speed) and a function called "void SystemCoreClockUpdate(void)" that is used after clock frequency changes to update "SystemCoreClock." The "SystemCoreClock" variable and the "SystemCoreClockUpdate" are available from CMSIS version 1.3.
Other files	There are additional files for peripheral control code and other helper functions. These files provide the device peripheral access layer of the CMSIS.



```
#include "vendor_device.h"

void main(void) {
  SystemInit();
  ...
  NVIC_SetPriority(UART1_IRQn, 0x0);
  NVIC_EnableIRQ(UART1_IRQn);
  ...
}
void UART1_IRQHandler {
 ...
}

void SysTick_Handler(void) {
...
}
```

Common name for system initialization code (from CMSIS v1.30, this function is called from startup code)
NVIC setup by core access functions
Interrupt numbers defined in system_<device>.h
Peripheral interrupt names are device specific, defined in device specific startup code
System exception handler names are common to all Cortex-M microcontrollers

Figure 4.19:
CMSIS example.

Learning to use a new Cortex-M microcontroller is made easier. Once you have used one Cortex-M microcontroller, you can start using another quickly because all CMSIS device driver libraries have the same core functions and a similar look and feel.

The CMSIS also lowers the risk of incompatibility when integrating third-party software components. Because middleware and an embedded RTOS will be based on the same core peripheral register definitions and core access functions in CMSIS files, this reduces the chance of conflicting code. This can happen when multiple software components carry their own core access functions and register definitions. Without CMSIS, you might possibly find that different third-party software programs contain unique driver functions. This could lead to register name clashes, confusion because of multiple functions with similar names, and a waste of code space as a result of duplicated functions (Figure 4.20).

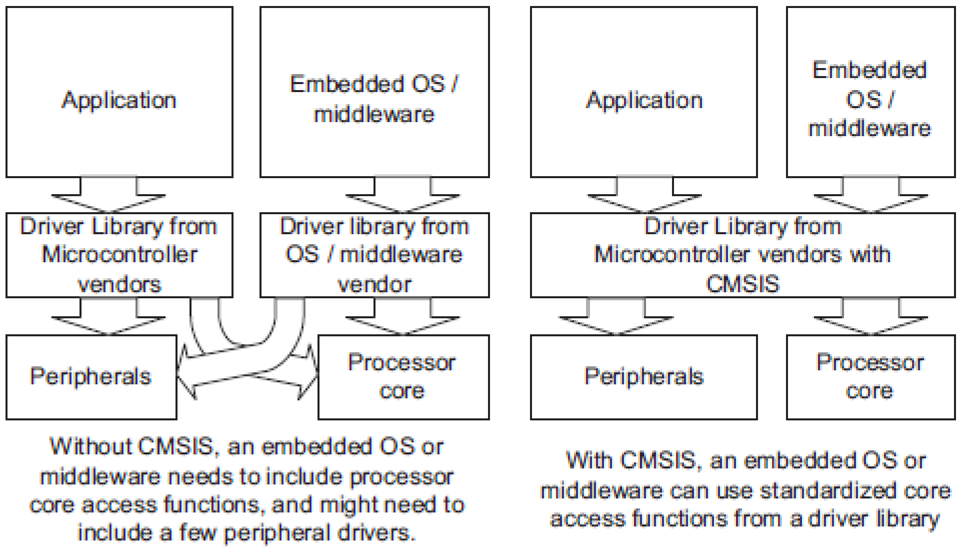

Application
Embedded OS / middleware
Application
Embedded OS / middleware
Driver Library from Microcontroller vendors
Driver library from OS / middleware vendor
Driver Library from Microcontroller vendors with CMSIS
Peripherals
Processor core
Peripherals
Processor core
Without CMSIS, an embedded OS or middleware needs to include processor core access functions, and might need to include a few peripheral drivers.
With CMSIS, an embedded OS or middleware can use standardized core access functions from a driver library

Figure 4.20:
CMSIS avoids overlapping of driver code.

CMSIS makes your software code future proof. Future Cortex-M microcontrollers will also have CMSIS support, so you can reuse your application code in future products.

The CMSIS core access functions have a small memory footprint. Multiple parties have tested CMSIS, and this helps reduce your software testing time. The CMSIS is Motor Industry Software Reliability Association (MISRA) compliant.

For companies developing an embedded OS or middleware products, the advantage of CMSIS is significant. Because CMSIS supports multiple compiler suites and is supported by multiple microcontroller vendors, the embedded OS or middleware developed with CMSIS can work on multiple complier products and can be used on multiple microcontroller families. Using CMSIS also means that these companies do not have to develop their own portable device drivers, which saves development time and verification efforts.

3.2 Instruction Set

3.2.1 Background of ARM and Thumb Instruction Set

The early ARM processors use a 32-bit instruction set called the ARM instructions. The 32-bit ARM instruction set is powerful and provides good performance, but at the same time it often requires larger program memory when compared to 8-bit and 16-bit processors. This was and still is an issue, as memory is expensive and could consume a considerable amount of power.

In 1995, ARM introduced the ARM7TDMI processor, adding a new 16-bit instruction set called the Thumb instruction set. The ARM7TDMI supports both ARM instructions and Thumb instructions, and a state-switching mechanism is used to allow the processor to decide which instruction decode scheme should be used (Figure 5.1). The Thumb instruction set provides a subset of the ARM instructions. By itself it can perform most of the normal functions, but interrupt entry sequence and boot code must still be in ARM state. Nevertheless, most processing can be carried out using Thumb instructions and interrupt handlers could switch themselves to use the Thumb state, so the ARM7TDMI processor provides excellent code density when compared to other 32-bit RISC architectures.

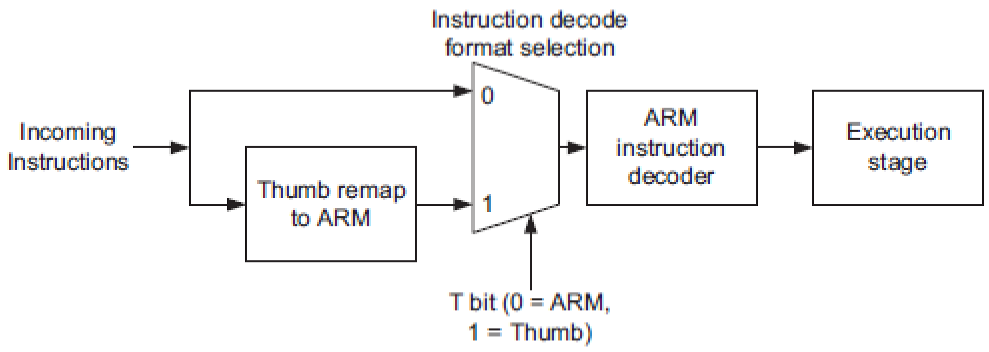


Figure 5.1:
ARM7TDMI design supports both ARM and the Thumb instruction set.

Thumb code provides a code size reduction of approximately 30% compared to the equivalent ARM code. However, it has some impact on the performance and can reduce the performance by 20%. On the other hand, in many applications, the reduction of program memory size and the low-power nature of the ARM7TDMI processor made it extremely popular with portable electronic devices like mobile phones and microcontrollers.

In 2003, ARM introduced Thumb-2 technology. This technology provides a number of 32-bit Thumb instructions as well as the original 16-bit Thumb instructions. The new 32-bit Thumb instructions can carry out most operations that previously could only be done with the ARM instruction set. As a result, program code compiled for Thumb-2 is typically 74% of the size of the same code compiled for ARM, but it maintains similar performance.

The Cortex-M3 processor is the first ARM processor that supports only Thumb-2 instructions. It can deliver up to 1.25 DMIPS per MHz (measured with Dhrystone 2.1), and various microcontroller vendors are already shipping microcontroller products based on the Cortex- M3 processor. By implementing only one instruction set, the software development is made simpler and at the same time improves the energy efficiency because only one instruction decoder is required (Figure 5.2).

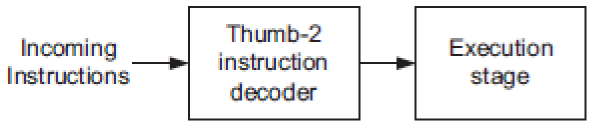


Figure 5.2:
Cortex-M processors do not have to remap instructions from Thumb to ARM.

In the ARMv6-M architecture used in the Cortex-M0 processor, in order to reduce the circuit size to a minimum, only the 16-bit Thumb instructions and a minimum subset of 32-bit Thumb instructions are supported. These 32-bit Thumb instructions are essential because the ARMv6-M architecture uses a number of features in the ARMv7-M architecture, which requires these instructions. For example, the accesses to the special registers require the MSR and MRS instructions. In addition, the Thumb-2 version of Branch and Link instruction (BL) is also included to provide a larger branch range.

Although the Cortex-M0 processor does not support many 32-bit Thumb instructions, the Thumb instruction set used in the Cortex-M0 processor is a superset of the original 16-bit Thumb instructions supported on the ARM7TDMI, which is based on ARMv4T architecture. Over the years, both ARM and Thumb instructions have gone through a number of enhancements as the architecture has evolved. For example, a number of instructions for data type conversions have been added to the Thumb instruction set for the ARMv6 and ARMv6-M architectures. These instruction set enhancements, along with various implementation optimizations, allow the

Cortex-M0 processor to deliver the same level of performance as an ARM7TDMI running ARM instructions.

Table 5.1: 16-Bit Thumb Instructions Supported on the Cortex-M0 Processor

16-Bit Thumb Instructions Supported on Cortex-M0									
ADC	ADD	ADR	AND	ASR	B	BIC	BLX	BKPT	BX
CMN	CMP	CPS	EOR	LDM	LDR	LDRH	LDRSH	LDRB	LDRSB
LSL	LSR	MOV	MVN	MUL	NOP	ORR	POP	PUSH	REV
REV16	REVSH	ROR	RSB	SBC	SEV	STM	STR	STRH	STRB
SUB	SVC	SXTB	SXTH	TST	UXTB	UXTH	WFE	WFI	YIELD

The Cortex-M0 processor also supports a number of 32-bit Thumb instructions from Thumb-2 technology (Table 5.2):

• MRS and MSR special register access instructions

• ISB, DSB, and DMB memory synchronization instructions

• BL instruction (BL was supported in traditional Thumb instruction set, but the bit field definition was extended in Thumb-2)

Table 5.2: 32-Bit Thumb Instructions Supported on the Cortex-M0 Processor

32-Bit Thumb Instructions Supported on Cortex-M0					
BL	DSB	DMB	ISB	MRS	MSR

3.2.2 *Assembly Basics*

This chapter introduces the instruction set of the Cortex-M0 processor. In most situations, application code can be written entirely in C language and therefore it is not necessary to know the details of the instruction set. However, it is still useful to know what instructions are available and their usages; for example, this information might be needed during debugging.

The complete details of each instruction are documented in the ARMv6-M Architecture Reference Manual (reference 3). Here, the basic syntax and usage are introduced. First of all, to help explain the assembly instructions covered in this chapter, some of the basic information about assembly syntax is introduced here.

3.2.2.1 Quick Glance at Assembly Syntax

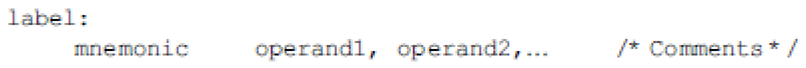
Most of the assembly examples in this book are written for the ARM assembler (armasm). Assembly tools from different vendors (e.g., GNU tool chain) have different assembly syntax. In most cases, the mnemonics of the assembly instructions are the same, but compile directives, definitions, labeling, and comment syntax can be different. For ARM assembly (applies to ARM RealView Development Suite and Keil Microcontroller Development Kit), the following instruction formatting is used:

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

The “label” is used as a reference to an address location. It is optional; some instructions might have a label in front of them so that the address of the instruction can be obtained using the label. Labels can also be used to reference data addresses. For example, you can put a label for a lookup table inside the program. After the “label,” you can find the “mnemonic,” which is the name of the instruction, followed by a number of operands. For data processing instructions written for the ARM assembler, the first operand is the destination of the operation. For a memory read or write, the first operand is the register that data are loaded into or the register that holds the write data (except for instructions that handle multiple loads and stores, which have a different syntax). The number of operands for each instruction depends on the instruction type. Some instructions do not need any operands, and some might need just one operand. Note that some mnemonics can use different types of operands and can result in different instruction encodings. For example, the MOV (move) instruction can be used to transfer data between two registers, or it can be used to put an immediate constant value into a register. The number of operands in an instruction depends on what type of instruction it is, and the syntax format can also be different. For example, immediate data are usually prefixed with “#”:

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

The text after each semicolon “;” is a comments. Comments do not affect the program operation but should make programs easier for humans to understand. With GNU tool chain, the common assembly syntax is



```
label:
     mnemonic     operand1, operand2,...     /* Comments */
```

The opcode and operands are the same as theARMassembler syntax, but the syntax for label and comments is different. For the same instructions as in the previous example, the GNU version is

```
MOVS R0, #0x12 /* Set R0 ¼ 0x12 (hexadecimal) */
MOVS R1, # ’A ’ /* Set R1 ¼ ASCII character A */
```

One of the commonly required features in assembly code is constant definitions. By using constant definitions, the program code can be more readable and can make code maintenance easier. In ARM assembly, an example of defining a constant is

```
NVIC_IRQ_SETEN EQU 0xE000E100
NVIC_IRQ0_ENABLE EQU 0x1
.
LDR R0,¼NVIC_IRQ_SETEN ; Put 0xE000E100 into R0
; LDR here is a pseudo instruction that will be converted
; to a PC relative literal data load by the assembler
MOVS R1, #NVIC_IRQ0_ENABLE ; Put immediate data (0x1) into
; register R1
STR R1, [R0] ; Store 0x1 to 0xE000E100, this enable external
; interrupt IRQ#0
```

Similarly, the same code can be written with GNU tool chain assembler syntax:

```
.equ NVIC_IRQ_SETEN, 0xE000E100
.equ NVIC_IRQ0_ENABLE, 0x1
.
LDR R0,¼NVIC_IRQ_SETEN /* Put 0xE000E100 into R0
LDR here is a pseudo instruction that will be
converted to a PC relative load by the assembler */
MOVS R1, #NVIC_IRQ0_ENABLE /* Put immediate data (0x1) into
register R1 */
STR R1, [R0] /* Store 0x1 to 0xE000E100, this enable
external interrupt IRQ#0 */
```

Another typical feature in most assembly tools is allowing data to be inserted inside programs. For example, we can define data in a certain location in the program memory and access it with memory read instructions. In the ARM assembler, an example is

```
LDR R3,¼MY_NUMBER ; Get the memory location of MY_NUMBER
LDR R4, [R3] ; Read the value 0x12345678 into R4
.
LDR R0,¼HELLO_TEXT ; Get the starting address of HELLO_TEXT
BL PrintText ; Call a function called PrintText to
; display string
.
ALIGN 4
MY_NUMBER DCD 0x12345678
HELLO_TEXT DCB “Hello\n”, 0 ; Null terminated string
```

In the preceding example, “DCD” is used to insert a word-size data, and “DCB” is used to insert byte-size data into the program. When inserting word-size data in program, we should use the “ALIGN” directive before the data. The number after the ALIGN directive determines the alignment size; in this case, the value is 4, which forces the following data to be aligned to a word boundary. Unaligned accesses are not supported in the Cortex-M0 processor. By ensuring the data following (MY_NUMBER) is word aligned, the program will be able to access the data correctly, avoiding any potential alignment faults. Again, this example can be rewritten into GNU tool chain assembler syntax:

```
LDR R3,¼MY_NUMBER /* Get the memory location of MY_NUMBER */
LDR R4, [R3] /* Read the value 0x12345678 into R4 */
.
LDR R0,¼HELLO_TEXT /* Get the starting address of
HELLO_TEXT */
BL PrintText /* Call a function called PrintText to
display string */
.
.align 4
MY_NUMBER:
.word 0x12345678
HELLO_TEXT:
.asciz “Hello\n” /* Null terminated string */
```

A number of different directives are available in both the ARM assembler and the GNU assembler for inserting data into a program. Table 5.3 presents a few commonly used examples.

Table 5.3: Commonly Used Directives for Inserting Data into a Program

Type of Data to Insert	ARM Assembler	GNU Assembler
Word	DCD (e.g., DCD 0x12345678)	.word / .4byte (e.g., .word 0x012345678)
Half word	DCW (e.g., DCW 0x1234)	.hword / .2byte (e.g., .hword 0x01234)
Byte	DCB (e.g., DCB 0x12)	.byte (e.g., .byte 0x012)
String	DCB (e.g., TXT DCB "Hello\n", 0)	.ascii /.asciz (with NULL termination) (e.g., .ascii "Hello\n" .byte 0 /* add NULL character */) (e.g., .asciz "Hello\n")
Instruction	DCI (e.g., DCI 0xBE00 ; Breakpoint-BKPT 0)	.word /.hword (e.g., .hword 0xBE00 /* Breakpoint (BKPT 0) */)

3.2.2.2 Use of a Suffix

In the assembler for ARM processors, some instructions can be followed by suffixes. For Cortex-M0, the available suffixes are shown in Table 5.4.

Table 5.4: Suffixes for Cortex-M0 Assembly Language

Suffix	Descriptions
S	Update APSR (flags); for example, `ADDS R0, R1; this ADD operation will update APSR`
EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE	Conditional execution. EQ = Equal, NE = Not Equal, LT = Less Than, GT = Greater Than, etc. On the Cortex-M0 processor, these conditions can only be applied to conditional branches. For example, `BEQ label; branch to label if equal`

For the Cortex-M0 processor, most of the data processing instructions always update the APSR (flags); only a few of the data operations do not update the APSR. For example, when moving a piece of data from one register to another, it is possible to use

```
MOVS R0, R1 ; Move R1 into R0 and update APSR
or
MOV R0, R1 ; Move R1 into R0
```

The second group of suffixes is for conditional execution of instructions. In the Cortex-M0 processor, the only instruction that can be conditionally executed is a conditional branch. By updating the APSR using data operations, or instructions like test (TST) or compare (CMP), the program flow can be controlled. More details of this instruction will be covered later in this chapter when the conditional branch is introduced.

3.2.2.3 *Thumb Code and Unified Assembler Language (UAL)*

Traditionally, programming of the ARM processors in Thumb state is done with the Thumb Assembly syntax. To allow better portability between architectures and to use a single assembly language syntax between different ARM processors with various architectures, recent ARM development tools have been updated to support the Unified Assembler Language (UAL). For users who have used ARM7TDMI in the past, the most noticeable differences are the following:

• Some data operation instructions use three operands even when the destination register is the same as one of the source registers. In the past (pre-UAL), syntax might only use two operands for the same instructions.

• The “S” suffix becomes more explicit. In the past, when an assembly program file was assembled into Thumb code, most data operations were implied as instructions that updated the APSR; as a result, the “S” suffix was not essential. With the UAL syntax, instructions that update the APSR should have the “S” suffix to clearly indicate the expected operation. This prevents program code from failing when being ported from one architecture to another.

For example, a pre-UAL ADD instruction for 16-bit Thumb code is

```
ADD R0, R1 ; R0 ¼ R0 þ R1, update APSR
```

With UAL syntax, this should be written as

```
ADDS R0, R0, R1 ; R0 ¼ R0 þ R1, update APSR
```

But in most cases (dependent on tool chain being used), you can still write the instruction with a pre-UAL style (only two operands), but the use of “S” suffix will be more explicit:

```
ADDS R0, R1 ; R0 ¼ R0 þ R1, update APSR
```

Most development tools still accept the pre-UAL syntax, including the ARM RealView Development Suite (RVDS) and the Keil Microcontroller Development Kit for ARM (MDK). However, the use of UAL is recommended for new projects. For assembly development with

RVDS or Keil MDK, you can specify using UAL syntax with “THUMB” directives and pre-UAL syntax with “CODE16” directives. The choice of assembler syntax depends on which tool you use. Please refer to the documentation for your development suite to determine the suitable syntax.

3.2.2.4 Instruction List

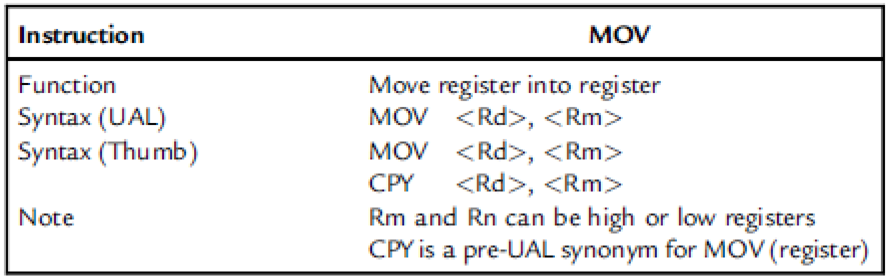
The instructions in the Cortex-M0 processor can be divided into various groups based on functionality:

• Moving data within the processor
• Memory accesses
• Stack memory accesses
• Arithmetic operations
• Logic operations
• Shift and rotate operations
• Extend and reverse ordering operations
• Program flow control (branch, conditional branch, and function calls)
• Memory barrier instructions
• Exception-related instructions
• Other functions

In this section, the instructions are discussed in more detail. The syntax illustrated here uses symbols “Rd,” “Rm,” and the like. In real program code, these need to be substituted with register names R0, R1, R2, and so on.

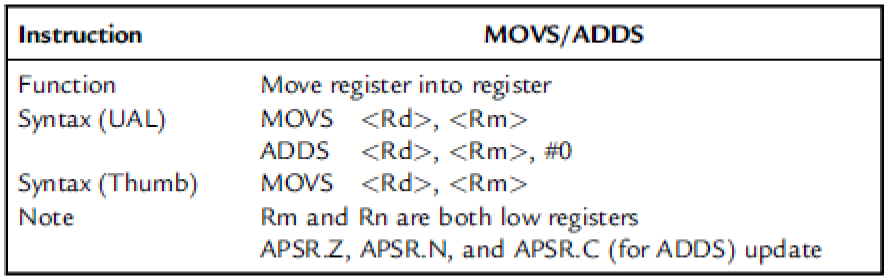
3.2.2.5 ***Moving Data within the Processor***

Transferring data is one of the most common tasks in a processor. In Thumb code, the instruction mnemonic for moving data is MOV. There are several types of MOV instructions, based on the operand type and opcode suffix.



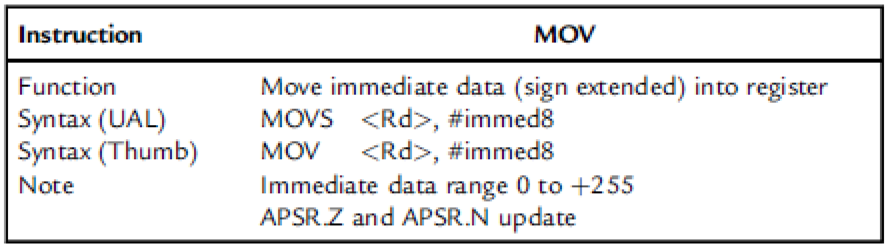
Instruction	MOV
Function	Move register into register
Syntax (UAL)	MOV <Rd>, <Rm>
Syntax (Thumb)	MOV <Rd>, <Rm>
	CPY <Rd>, <Rm>
Note	Rm and Rn can be high or low registers
	CPY is a pre-UAL synonym for MOV (register)

If we want to copy a register value to another and update the APSR at the same time, we could use MOVS/ADDS.



Instruction	MOVS/ADDS
Function	Move register into register
Syntax (UAL)	MOVS <Rd>, <Rm>
	ADDS <Rd>, <Rm>, #0
Syntax (Thumb)	MOVS <Rd>, <Rm>
Note	Rm and Rn are both low registers
	APSR.Z, APSR.N, and APSR.C (for ADDS) update

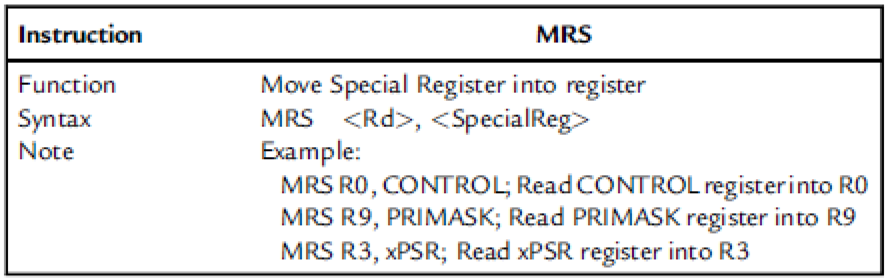
We can also load an immediate data element into a register using the MOV instruction.



Instruction	MOV
Function	Move immediate data (sign extended) into register
Syntax (UAL)	MOVS <Rd>, #immed8
Syntax (Thumb)	MOV <Rd>, #immed8
Note	Immediate data range 0 to +255
	APSR.Z and APSR.N update

If we want to load an immediate data element into a register that is out of the 8-bit value range, we need to store the data into a program memory space and then use a memory access instruction to read the data into the register. This can be written using a pseudo instruction LDR, which the assembler converts into a real instruction. This process will be covered later in this chapter.

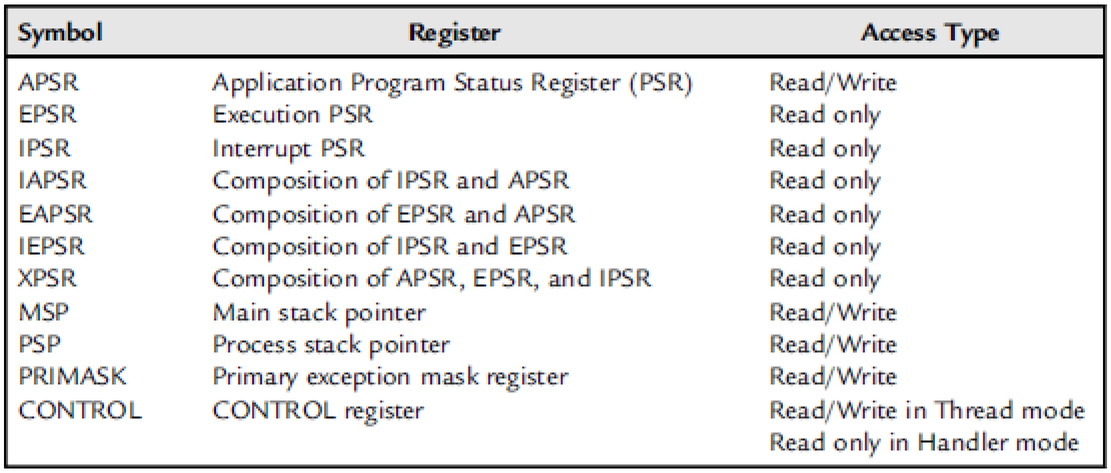
The MOV instructions can cause a branch to happen if the destination register is R15 (PC). However, generally the BX instruction is used for this purpose. Another type of data transfer in the Cortex-M0 processor is Special Registers accesses. To access the Special Registers (CONTROL, PRIMASK, xPSR, etc.), the MRS and MSR instructions are needed. These two instructions cannot be generated in C language. However, they can be created using inline assembler or Embedded Assembler,3 or another C compilere specific feature like the named register variables feature in ARM RVDS or Keil MDK.



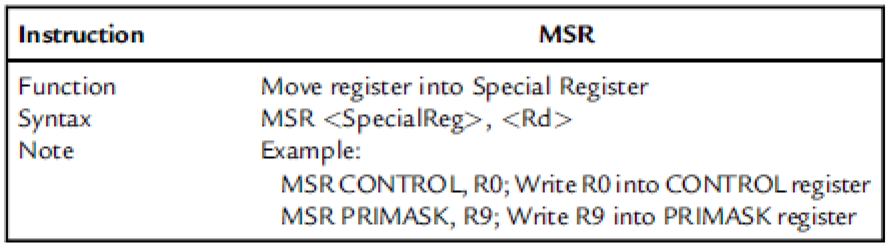
Instruction	MRS
Function	Move Special Register into register
Syntax	MRS <Rd>, <SpecialReg>
Note	Example: MRS R0, CONTROL; Read CONTROL register into R0 MRS R9, PRIMASK; Read PRIMASK register into R9 MRS R3, xPSR; Read xPSR register into R3

Table 5.5 shows the complete list of special register symbols that are available on the Cortex-M0 processor when MSR and MRS instructions are used.

Table 5.5: Special Register Symbols for MRS and MSR Instructions



Symbol	Register	Access Type
APSR	Application Program Status Register (PSR)	Read/Write
EPSR	Execution PSR	Read only
IPSR	Interrupt PSR	Read only
IAPSR	Composition of IPSR and APSR	Read only
EAPSR	Composition of EPSR and APSR	Read only
IEPSR	Composition of IPSR and EPSR	Read only
XPSR	Composition of APSR, EPSR, and IPSR	Read only
MSP	Main stack pointer	Read/Write
PSP	Process stack pointer	Read/Write
PRIMASK	Primary exception mask register	Read/Write
CONTROL	CONTROL register	Read/Write in Thread mode Read only in Handler mode



Instruction	MSR
Function	Move register into Special Register
Syntax	MSR <SpecialReg>, <Rd>
Note	Example: MSR CONTROL, R0; Write R0 into CONTROL register MSR PRIMASK, R9; Write R9 into PRIMASK register

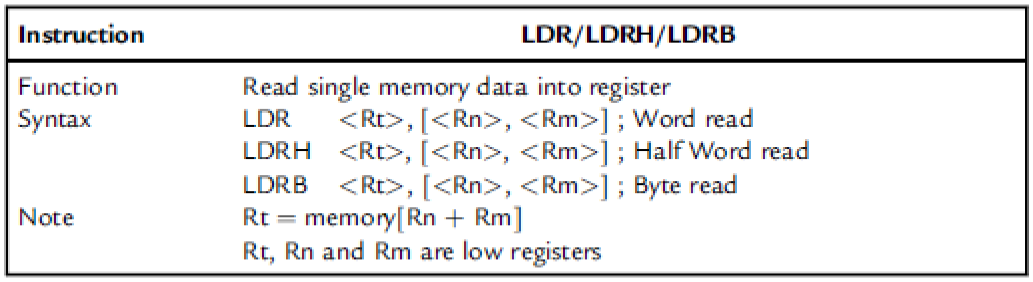
3.2.2.6 ***Memory Accesses***

The Cortex-M0 processor supports a number of memory access instructions, which support various data transfer sizes and addressing modes. The supported data transfer sizes are Word, HalfWord and Byte. In addition, there are separate instructions to support signed and unsigned data. Table 5.6 summarizes the memory address instruction mnemonics. Most of these instructions also support multiple addressing modes. When the instruction is used with different operands, the assembler will generate different instruction encoding.

Important

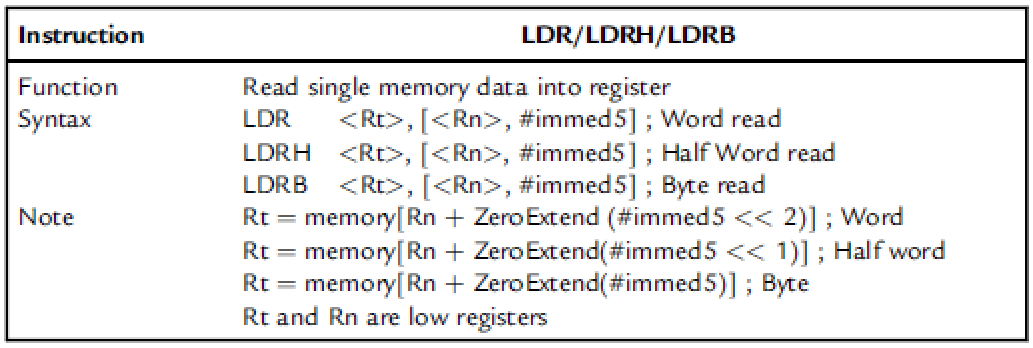
It is important to make sure the memory address accessed is aligned. For example, a word size access can only be carried out on address locations when address bits[1:0] are set to zero, and a half word size access can only be carried out on address locations when an address bit[0] is set to zero. The Cortex-M0 processor does not support unaligned transfers. Any attempt at unaligned memory access results in a hard fault exception. Byte-size transfers are always aligned on the Cortex-M0 processor.

For memory read operations, the instruction to carry out single accesses is LDR (load):



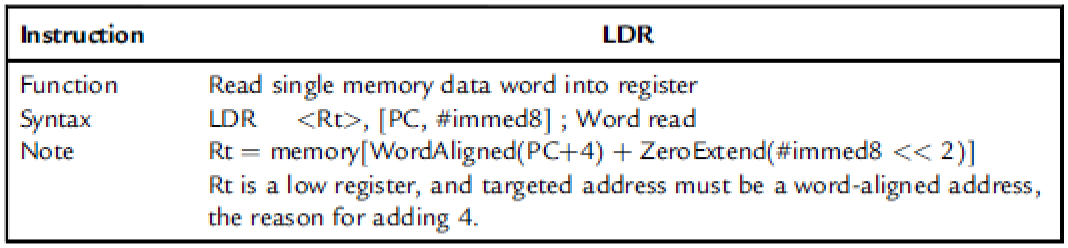
Instruction	LDR/LDRH/LDRB
Function	Read single memory data into register
Syntax	LDR <Rt>, [<Rn>, <Rm>] ; Word read
	LDRH <Rt>, [<Rn>, <Rm>] ; Half Word read
	LDRB <Rt>, [<Rn>, <Rm>] ; Byte read
Note	Rt = memory[Rn + Rm]
	Rt, Rn and Rm are low registers

The Cortex-M0 processor also supports immediate offset addressing modes:

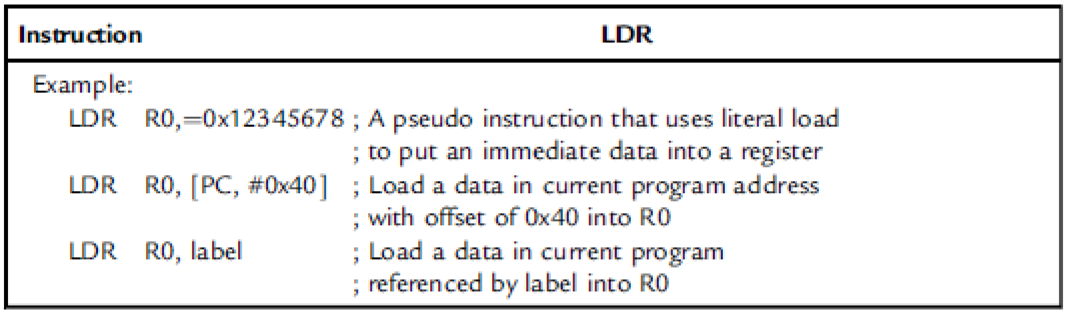


Instruction	LDR/LDRH/LDRB
Function	Read single memory data into register
Syntax	LDR <Rt>, [<Rn>, #immed5] ; Word read
	LDRH <Rt>, [<Rn>, #immed5] ; Half Word read
	LDRB <Rt>, [<Rn>, #immed5] ; Byte read
Note	Rt = memory[Rn + ZeroExtend (#immed5 << 2)] ; Word
	Rt = memory[Rn + ZeroExtend(#immed5 << 1)] ; Half word
	Rt = memory[Rn + ZeroExtend(#immed5)] ; Byte
	Rt and Rn are low registers

The Cortex-M0 processor supports a useful PC relative load instruction for allowing efficient literal data accesses. This instruction can be generated when we use the LDR pseudo instruction for putting an immediate data value into a register. These data are stored alongside the instructions, called literal pools.



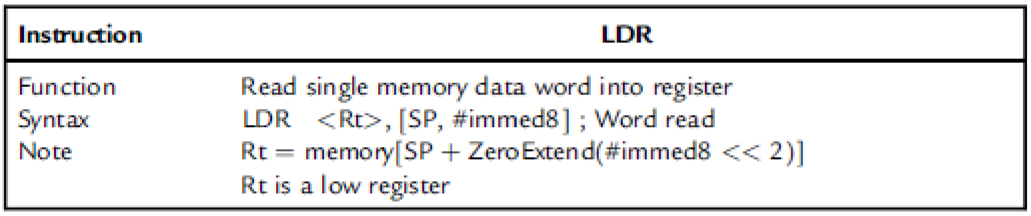
Instruction	**LDR**
Function	Read single memory data word into register
Syntax	LDR <Rt>, [PC, #immed8] ; Word read
Note	Rt = memory[WordAligned(PC+4) + ZeroExtend(#immed8 << 2)]
	Rt is a low register, and targeted address must be a word-aligned address, the reason for adding 4.



Instruction	**LDR**

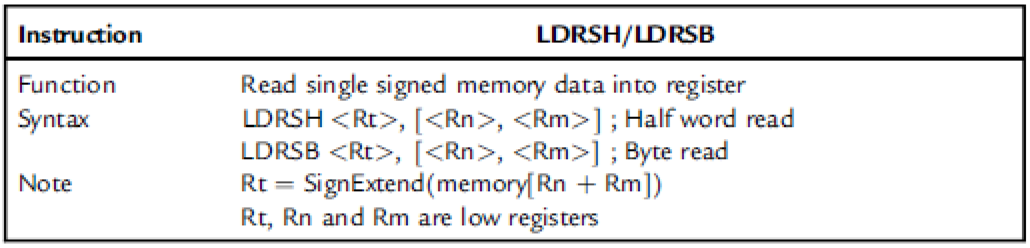
```
Example:
    LDR    R0,=0x12345678 ; A pseudo instruction that uses literal load
                          ; to put an immediate data into a register
    LDR    R0, [PC, #0x40]  ; Load a data in current program address
                          ; with offset of 0x40 into R0
    LDR    R0, label      ; Load a data in current program
                          ; referenced by label into R0
```

There is also an SP-related load instruction, which supports a wider offset range. This instruction is useful for accessing local variables in C functions because often the local variables are stored on the stack.



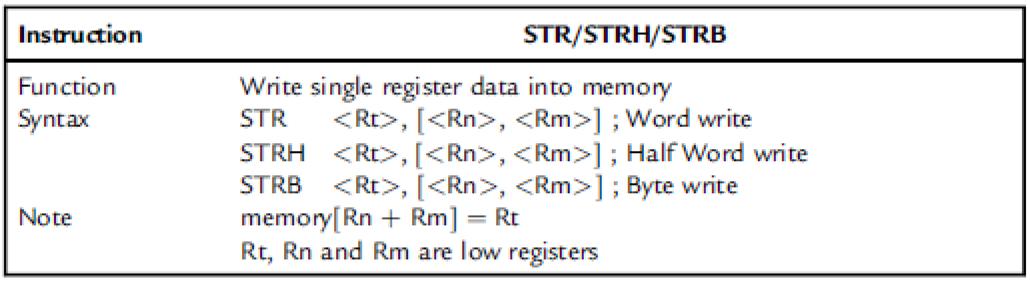
Instruction	**LDR**
Function	Read single memory data word into register
Syntax	LDR <Rt>, [SP, #immed8] ; Word read
Note	Rt = memory[SP + ZeroExtend(#immed8 << 2)]
	Rt is a low register

The Cortex-M0 processor can also sign extends the read data automatically using the LDRSB and LDRSH instructions. This is useful when a signed 8-bit/16-bit data type is used, which is common in C programs.



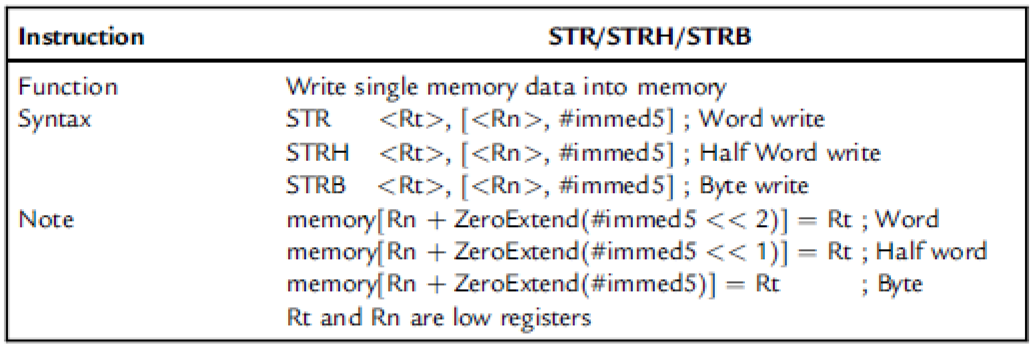
Instruction	**LDRSH/LDRSB**
Function	Read single signed memory data into register
Syntax	LDRSH <Rt>, [<Rn>, <Rm>] ; Half word read
	LDRSB <Rt>, [<Rn>, <Rm>] ; Byte read
Note	Rt = SignExtend(memory[Rn + Rm])
	Rt, Rn and Rm are low registers

For single data memory writes, the instruction is STR (store):



Instruction	STR/STRH/STRB
Function	Write single register data into memory
Syntax	STR <Rt>, [<Rn>, <Rm>] ; Word write
	STRH <Rt>, [<Rn>, <Rm>] ; Half Word write
	STRB <Rt>, [<Rn>, <Rm>] ; Byte write
Note	memory[Rn + Rm] = Rt
	Rt, Rn and Rm are low registers

Like the load operation, the store operation supports an immediate offset addressing mode:

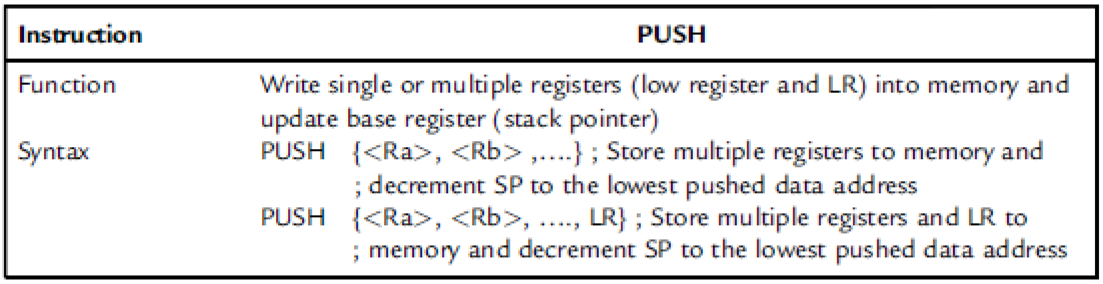


Instruction	STR/STRH/STRB
Function	Write single memory data into memory
Syntax	STR <Rt>, [<Rn>, #immed5] ; Word write
	STRH <Rt>, [<Rn>, #immed5] ; Half Word write
	STRB <Rt>, [<Rn>, #immed5] ; Byte write
Note	memory[Rn + ZeroExtend(#immed5 << 2)] = Rt ; Word
	memory[Rn + ZeroExtend(#immed5 << 1)] = Rt ; Half word
	memory[Rn + ZeroExtend(#immed5)] = Rt ; Byte
	Rt and Rn are low registers

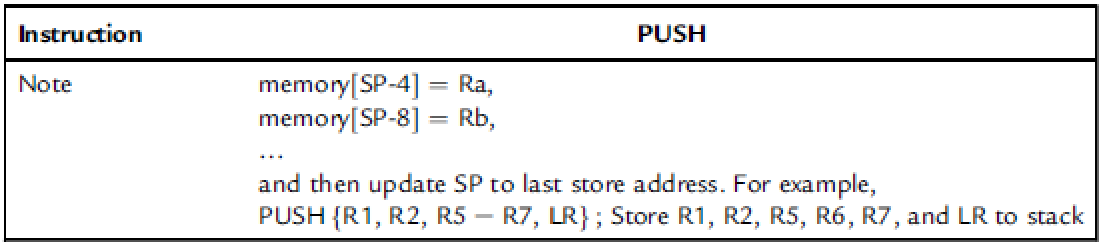
An SP-relative store instruction, which supports a wider offset range, is also available. This instruction is useful for accessing local variables in C functions because often the local variables are stored on the stack.

3.2.2.7 Stack Memory Access

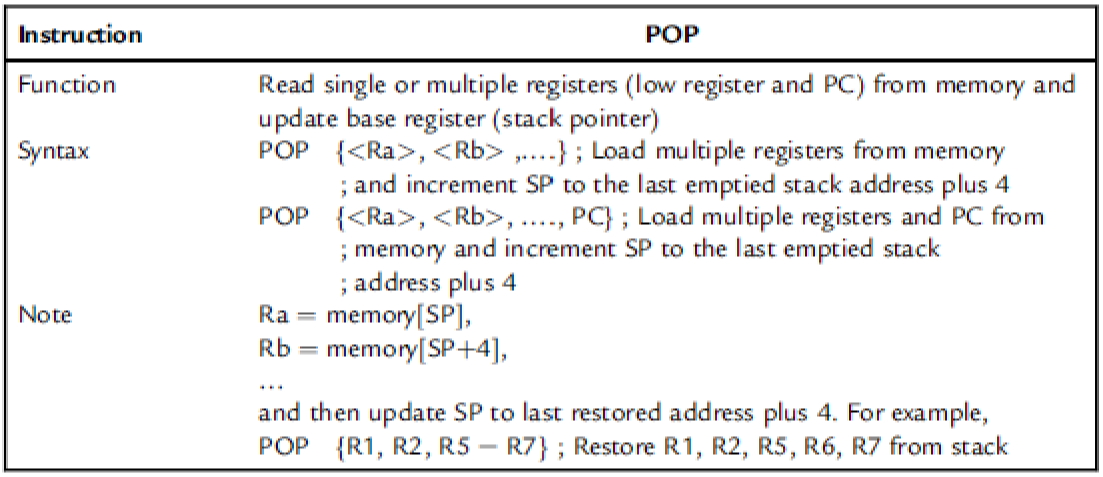
Two memory access instructions are dedicated to stack memory accesses. The PUSH instruction is used to decrement the current stack pointer and store data to the stack. The POP instruction is used to read the data from the stack and increment the current stack pointer. Both PUSH and POP instructions allow multiple registers to be stored or restored. However, only low registers, LR (for PUSH operation) and PC (for POP operation), are supported.



Instruction	**PUSH**
Function	Write single or multiple registers (low register and LR) into memory and update base register (stack pointer)
Syntax	PUSH {<Ra>, <Rb> ,....} ; Store multiple registers to memory and ; decrement SP to the lowest pushed data address PUSH {<Ra>, <Rb>, ..., LR} ; Store multiple registers and LR to ; memory and decrement SP to the lowest pushed data address



Instruction	**PUSH**
Note	memory[SP-4] = Ra, memory[SP-8] = Rb, ... and then update SP to last store address. For example, PUSH {R1, R2, R5 – R7, LR} ; Store R1, R2, R5, R6, R7, and LR to stack



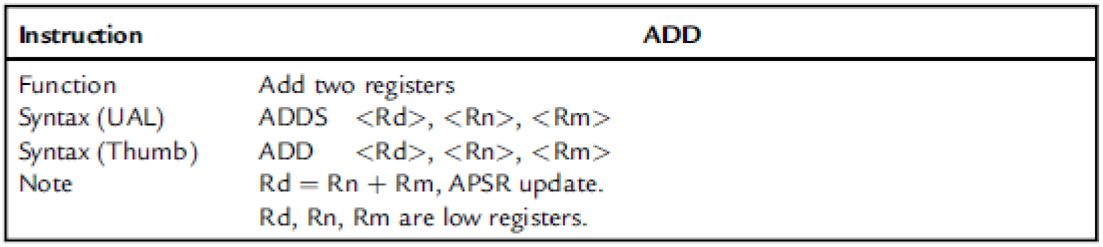
Instruction	**POP**
Function	Read single or multiple registers (low register and PC) from memory and update base register (stack pointer)
Syntax	POP {<Ra>, <Rb> ,....} ; Load multiple registers from memory ; and increment SP to the last emptied stack address plus 4 POP {<Ra>, <Rb>, ..., PC} ; Load multiple registers and PC from ; memory and increment SP to the last emptied stack ; address plus 4
Note	Ra = memory[SP], Rb = memory[SP+4], ... and then update SP to last restored address plus 4. For example, POP {R1, R2, R5 – R7} ; Restore R1, R2, R5, R6, R7 from stack

By allowing the Link Register (LR) and Program Counter (PC) to be used with the PUSH and the POP instructions, a function call can combine the register restore and function return operations into a single instruction. For example,

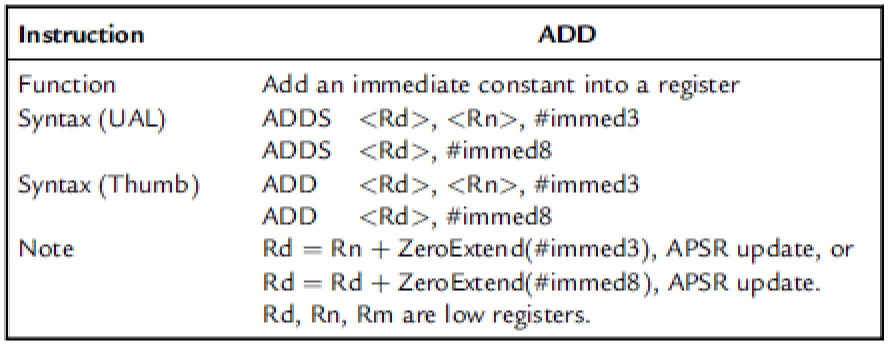
```
my_function
PUSH {R4, R5, R7, LR} ; Save R4, R5, R7 and LR (return address)
. ; function body
POP {R4, R5, R7, PC} ; Restore R4, R5, R7 and return
```

3.2.2.8 ***Arithmetic Operations***

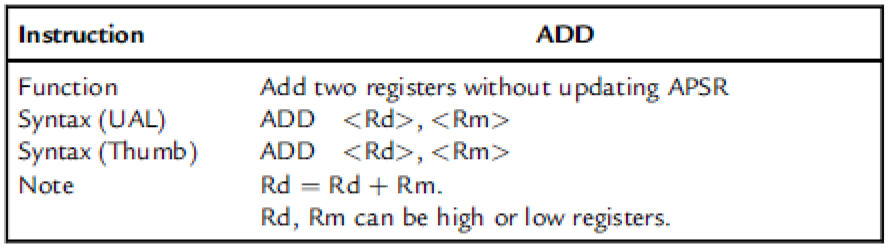
The Cortex-M0 processor supports a number of arithmetic operations. The most basic are add, subtract, twos complement, and multiply. For most of these instructions, the operation can be carried out between two registers, or between one register and an immediate constant.



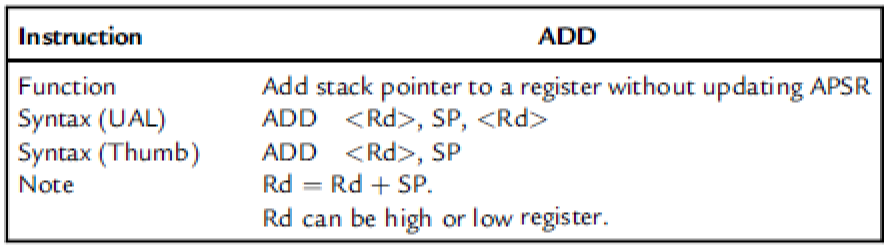
Instruction	ADD
Function	Add two registers
Syntax (UAL)	ADDS <Rd>, <Rn>, <Rm>
Syntax (Thumb)	ADD <Rd>, <Rn>, <Rm>
Note	Rd = Rn + Rm, APSR update.
	Rd, Rn, Rm are low registers.



Instruction	ADD
Function	Add an immediate constant into a register
Syntax (UAL)	ADDS <Rd>, <Rn>, #immed3
	ADDS <Rd>, #immed8
Syntax (Thumb)	ADD <Rd>, <Rn>, #immed3
	ADD <Rd>, #immed8
Note	Rd = Rn + ZeroExtend(#immed3), APSR update, or
	Rd = Rd + ZeroExtend(#immed8), APSR update.
	Rd, Rn, Rm are low registers.



Instruction	ADD
Function	Add two registers without updating APSR
Syntax (UAL)	ADD <Rd>, <Rm>
Syntax (Thumb)	ADD <Rd>, <Rm>
Note	Rd = Rd + Rm.
	Rd, Rm can be high or low registers.



Instruction	ADD
Function	Add stack pointer to a register without updating APSR
Syntax (UAL)	ADD <Rd>, SP, <Rd>
Syntax (Thumb)	ADD <Rd>, SP
Note	Rd = Rd + SP.
	Rd can be high or low register.

Instruction	**ADD**
Function	Add stack pointer to a register without updating APSR
Syntax (UAL)	ADD SP, <Rm>
Syntax (Thumb)	ADD SP, <Rm>
Note	SP = SP + Rm. Rm can be high or low register.

Instruction	**ADD**
Function	Add stack pointer to a register without updating APSR
Syntax (UAL)	ADD <Rd>, SP, #immed8
Syntax (Thumb)	ADD <Rd>, SP, #immed8
Note	Rd = SP + ZeroExtend(#immed8 <<2). Rd is a low register.

Instruction	**ADD**
Function	Add an immediate constant to stack pointer
Syntax (UAL)	ADD SP, SP, #immed7
Syntax (Thumb)	ADD SP, #immed7
Note	SP = SP + ZeroExtend(#immed7 <<2). This instruction is useful for C functions to adjust the SP for local variables.

Instruction	**ADR (ADD)**
Function	Add an immediate constant with PC to a register without updating APSR
Syntax (UAL)	ADR <Rd>, <label> (normal syntax) ADD <Rd>, PC, #immed8 (alternate syntax)
Syntax (Thumb)	ADR <Rd>, (normal syntax) ADD <Rd>, PC, #immed8 (alternate syntax)
Note	Rd = (PC[31:2]<<2) + ZeroExtend(#immed8 <<2). This instruction is useful for locating a data address within the program memory near to the current instruction. The result address must be word aligned. Rd is a low register.

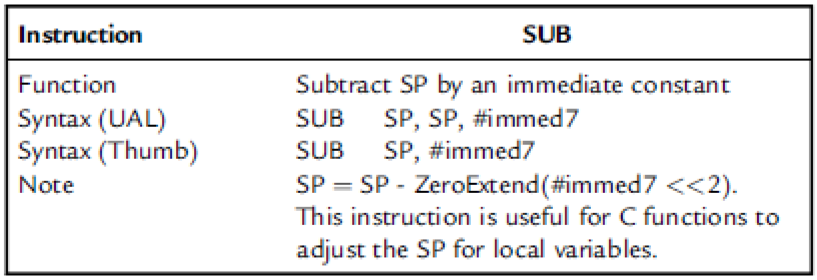
Instruction	**ADC**
Function	Add with carry and update APSR
Syntax (UAL)	ADCS <Rd>, <Rm>
Syntax (Thumb)	ADC <Rd>, <Rm>
Note	Rd = Rd + Rm + Carry Rd and Rm are low registers.

Instruction	ADC
Function	Add with carry and update APSR
Syntax (UAL)	ADCS <Rd>, <Rm>
Syntax (Thumb)	ADC <Rd>, <Rm>
Note	Rd = Rd + Rm + Carry Rd and Rm are low registers.

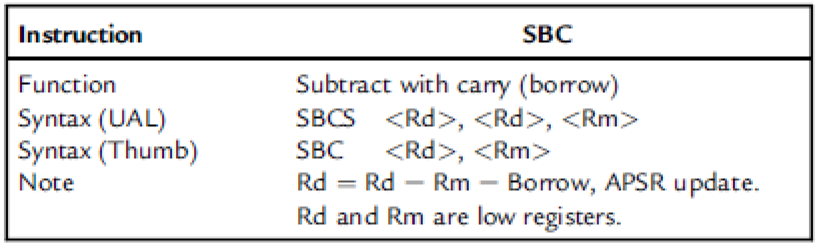
Instruction	SUB
Function	Subtract two registers
Syntax (UAL)	SUBS <Rd>, <Rn>, <Rm>
Syntax (Thumb)	SUB <Rd>, <Rn>, <Rm>
Note	Rd = Rn − Rm, APSR update. Rd, Rn, Rm are low registers.

Instruction	SUB
Function	Subtract a register with an immediate constant
Syntax (UAL)	SUBS <Rd>, <Rn>, #immed3 SUBS <Rd>, #immed8
Syntax (Thumb)	SUB <Rd>, <Rn>, #immed3 SUB <Rd>, #immed8

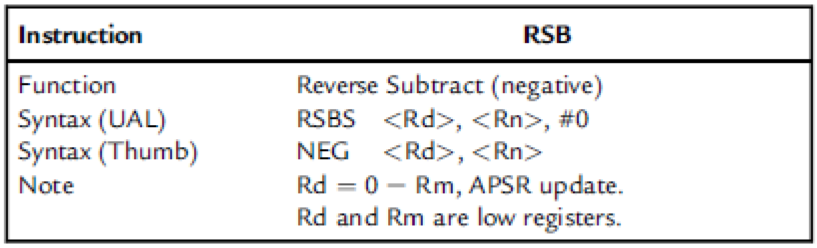
Note	Rd = Rn − ZeroExtend(#immed3), APSR update, or
	Rd = Rd − ZeroExtend(#immed8), APSR update.
	Rd, Rn are low registers.



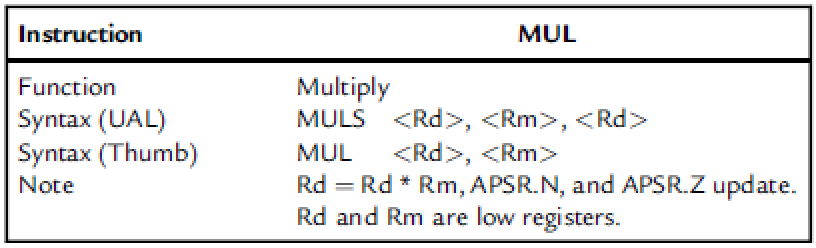
Instruction	SUB
Function	Subtract SP by an immediate constant
Syntax (UAL)	SUB SP, SP, #immed7
Syntax (Thumb)	SUB SP, #immed7
Note	SP = SP - ZeroExtend(#immed7 <<2). This instruction is useful for C functions to adjust the SP for local variables.



Instruction	SBC
Function	Subtract with carry (borrow)
Syntax (UAL)	SBCS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	SBC <Rd>, <Rm>
Note	Rd = Rd − Rm − Borrow, APSR update. Rd and Rm are low registers.

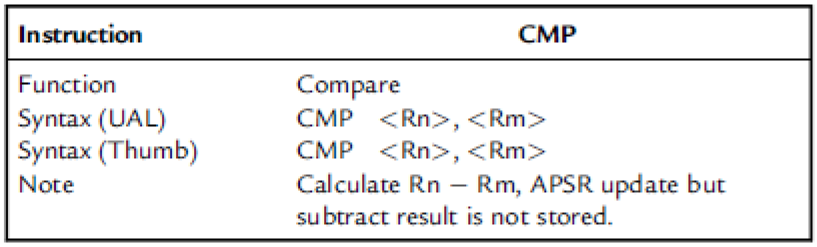


Instruction	RSB
Function	Reverse Subtract (negative)
Syntax (UAL)	RSBS <Rd>, <Rn>, #0
Syntax (Thumb)	NEG <Rd>, <Rn>
Note	Rd = 0 − Rm, APSR update. Rd and Rm are low registers.

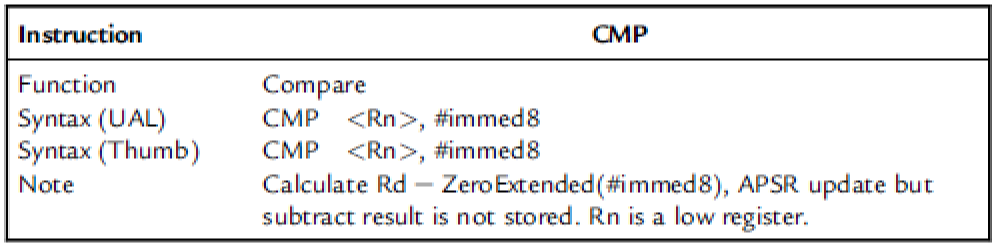


Instruction	MUL
Function	Multiply
Syntax (UAL)	MULS <Rd>, <Rm>, <Rd>
Syntax (Thumb)	MUL <Rd>, <Rm>
Note	Rd = Rd * Rm, APSR.N, and APSR.Z update. Rd and Rm are low registers.

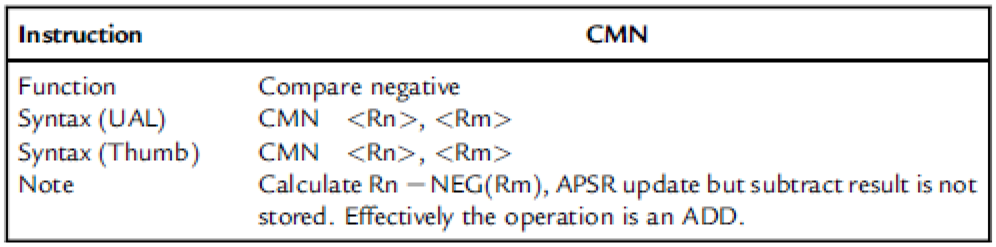
There are also a few compare instructions that compare (using subtract) values and update flags (APSR), but the result of the comparison is not stored.



Instruction	CMP
Function	Compare
Syntax (UAL)	CMP <Rn>, <Rm>
Syntax (Thumb)	CMP <Rn>, <Rm>
Note	Calculate Rn − Rm, APSR update but subtract result is not stored.



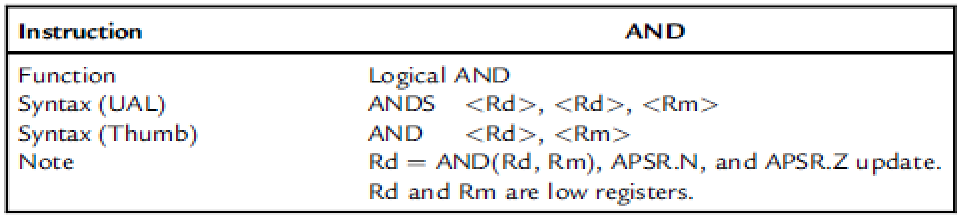
Instruction	CMP
Function	Compare
Syntax (UAL)	CMP <Rn>, #immed8
Syntax (Thumb)	CMP <Rn>, #immed8
Note	Calculate Rd − ZeroExtended(#immed8), APSR update but subtract result is not stored. Rn is a low register.



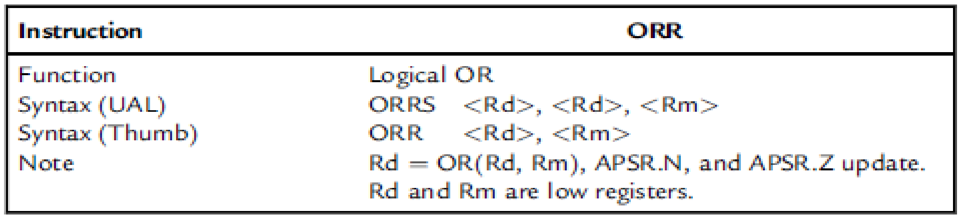
Instruction	CMN
Function	Compare negative
Syntax (UAL)	CMN <Rn>, <Rm>
Syntax (Thumb)	CMN <Rn>, <Rm>
Note	Calculate Rn − NEG(Rm), APSR update but subtract result is not stored. Effectively the operation is an ADD.

3.2.2.9 Logic Operations

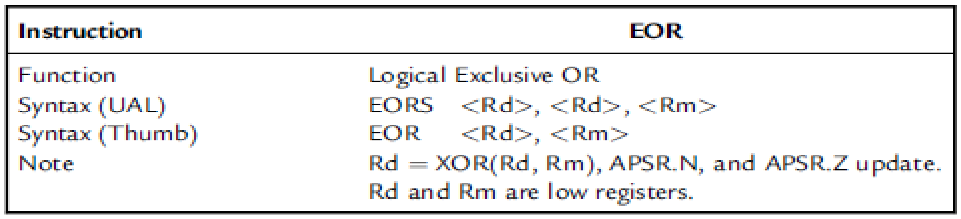
Another set of essential operations in most processors is made up of logic operations. For logical operations, the Cortex-M0 processor has a number of instructions available, including basic features like AND, OR, and the like. In addition, it has a number of instructions for compare and testing.



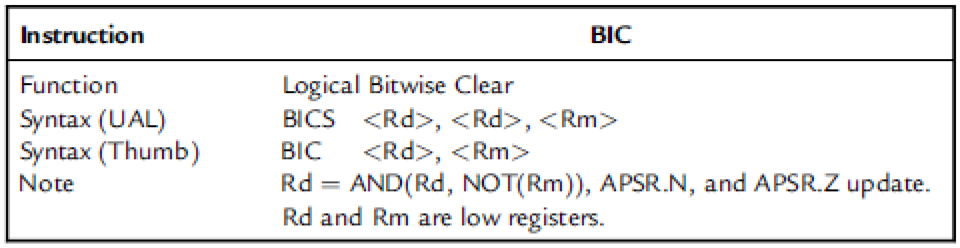
Instruction	AND
Function	Logical AND
Syntax (UAL)	ANDS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	AND <Rd>, <Rm>
Note	Rd = AND(Rd, Rm), APSR.N, and APSR.Z update. Rd and Rm are low registers.



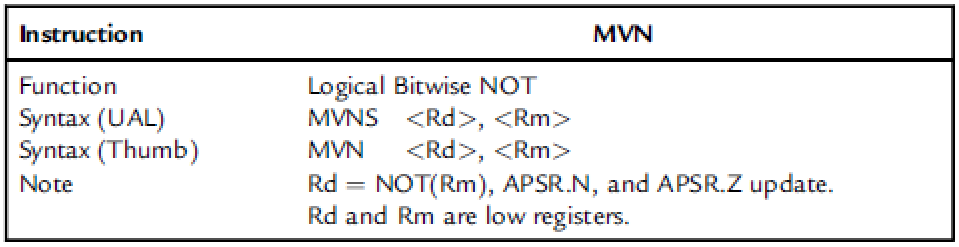
Instruction	ORR
Function	Logical OR
Syntax (UAL)	ORRS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	ORR <Rd>, <Rm>
Note	Rd = OR(Rd, Rm), APSR.N, and APSR.Z update. Rd and Rm are low registers.



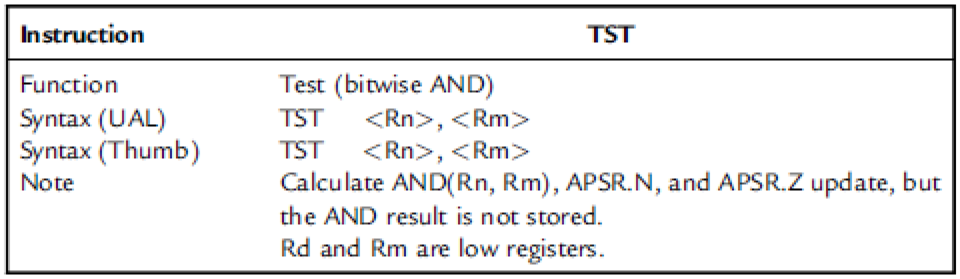
Instruction	EOR
Function	Logical Exclusive OR
Syntax (UAL)	EORS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	EOR <Rd>, <Rm>
Note	Rd = XOR(Rd, Rm), APSR.N, and APSR.Z update. Rd and Rm are low registers.



Instruction	BIC
Function	Logical Bitwise Clear
Syntax (UAL)	BICS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	BIC <Rd>, <Rm>
Note	Rd = AND(Rd, NOT(Rm)), APSR.N, and APSR.Z update. Rd and Rm are low registers.



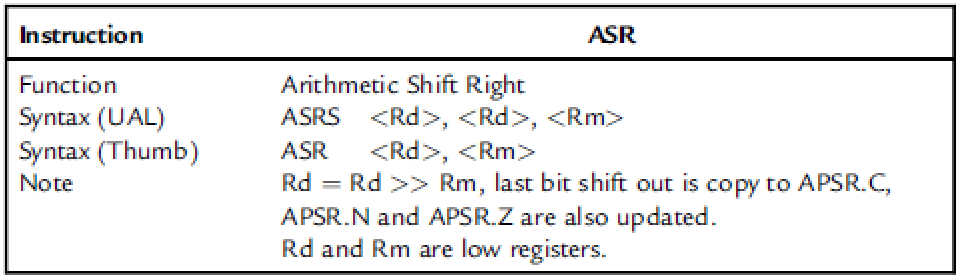
Instruction	MVN
Function	Logical Bitwise NOT
Syntax (UAL)	MVNS <Rd>, <Rm>
Syntax (Thumb)	MVN <Rd>, <Rm>
Note	Rd = NOT(Rm), APSR.N, and APSR.Z update. Rd and Rm are low registers.



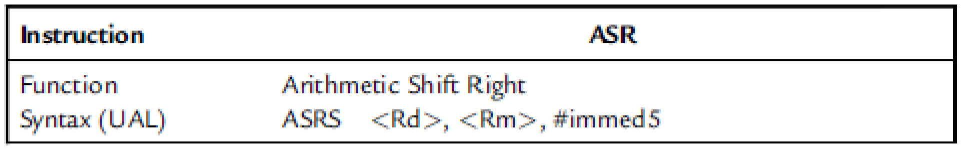
Instruction	TST
Function	Test (bitwise AND)
Syntax (UAL)	TST <Rn>, <Rm>
Syntax (Thumb)	TST <Rn>, <Rm>
Note	Calculate AND(Rn, Rm), APSR.N, and APSR.Z update, but the AND result is not stored. Rd and Rm are low registers.

3.2.2.10 Shift and Rotate operations

The Cortex-M0 also supports shift and rotate instructions. It supports both arithmetic shift operations (the datum is a signed integer value where MSB needs to be reserved) as well as logical shift.



Instruction	ASR
Function	Arithmetic Shift Right
Syntax (UAL)	ASRS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	ASR <Rd>, <Rm>
Note	Rd = Rd >> Rm, last bit shift out is copy to APSR.C, APSR.N and APSR.Z are also updated. Rd and Rm are low registers.



Instruction	ASR
Function	Arithmetic Shift Right
Syntax (UAL)	ASRS <Rd>, <Rm>, #immed5

Instruction	**ASR**
Syntax (Thumb)	ASR <Rd>, <Rm>, #immed5
Note	Rd = Rm >> immed5, last bit shifted out is copied to APSR.C, APSR.N and APSR.Z are also updated. Rd and Rm are low registers.

When ASR is used, the MSB of the result is unchanged, and the Carry flag is updated using the last bit shifted out (Figure 5.3).

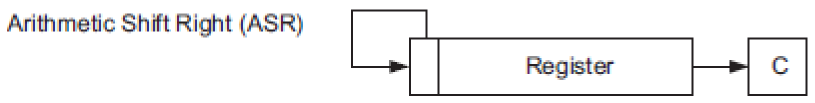


Figure 5.3:
Arithmetic Shift Right.

For logical shift operations, the instructions are LSL (Figure 5.4) and LSR (Figure 5.5).

Instruction	**LSL**
Function	Logical Shift Left
Syntax (UAL)	LSLS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	LSL <Rd>, <Rm>
Note	Rd = Rd << Rm, last bit shifted out is copied to APSR.C, APSR.N and APSR.Z are also updated. Rd and Rm are low registers.

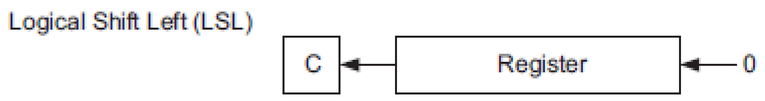


Figure 5.4:
Logical Shift Left.

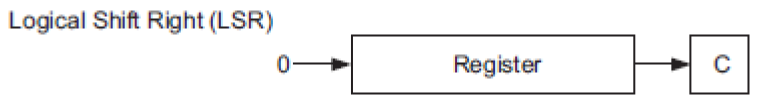


Figure 5.5:
Logical Shift Right.

Instruction	LSL
Function	Logical Shift Left
Syntax (UAL)	LSLS <Rd>, <Rm>, #immed5
Syntax (Thumb)	LSL <Rd>, <Rm>, #immed5
Note	Rd = Rm << #immed5, last bit shifted out is copied to APSR.C, APSR.N and APSR.Z are also updated. Rd and Rm are low registers.

Instruction	LSR
Function	Logical Shift Right
Syntax (UAL)	LSRS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	LSR <Rd>, <Rm>
Note	Rd = Rd >> Rm, last bit shifted out is copied to APSR.C, APSR.N and APSR.Z are also updated. Rd and Rm are low registers.

Instruction	LSR
Function	Logical Shift Right
Syntax (UAL)	LSRS <Rd>, <Rm>, #immed5
Syntax (Thumb)	LSR <Rd>, <Rm>, #immed5
Note	Rd = Rm >> #immed5, last bit shifted out is copied to APSR.C, APSR.N and APSR.Z are also updated. Rd and Rm are low registers.

There is only one rotate instruction, ROR (Figure 5.6).

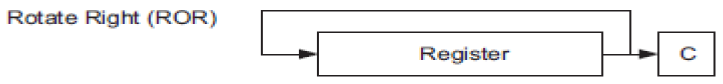


Figure 5.6:
Rotate Right.

Instruction	ROR
Function	Rotate Right
Syntax (UAL)	RORS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	ROR <Rd>, <Rm>
Note	Rd = Rd rotate right by Rm bits, last bit shifted out is copied to APSR.C, APSR.N and APSR.Z are also updated. Rd and Rm are low registers.

If a rotate left operation is needed, this can be done using a ROR with a different offset:

$$\mathrm{Rotate\_Left(Data,\ offset)} == \mathrm{Rotate\_Right(Data,\ (32 - offset))}$$

3.2.2.11 Extend and Reverse ordering

The Cortex-M0 processor supports a number of instructions that can perform data reordering or extraction (Figures 5.7, 5.8, and 5.9).

Instruction	REV (Byte-Reverse Word)
Function	Byte Order Reverse
Syntax	REV <Rd>, <Rm>
Note	Rd = {Rm[7:0] , Rm[15:8], Rm[23:16], Rm[31:24]} Rd and Rm are low registers.

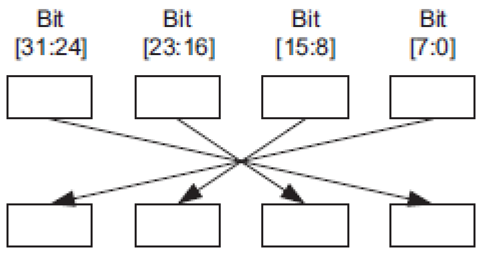


Figure 5.7:
REV operation.

Instruction	REV16 (Byte-Reverse Packed Half Word)
Function	Byte Order Reverse within half word
Syntax	REV16 <Rd>, <Rm>
Note	Rd = {Rm[23:16], Rm[31:24], Rm[7:0] , Rm[15:8]} Rd and Rm are low registers.

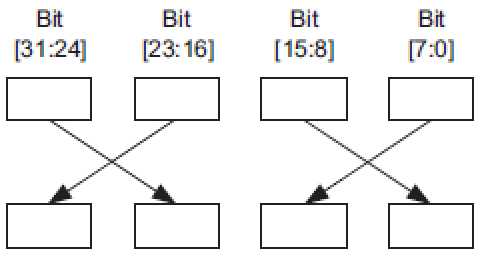


Figure 5.8:
REV16 operation.

Instruction	REVSH (Byte-Reverse Signed Half Word)
Function	Byte order reverse within lower half word, then sign extend result
Syntax	REVSH <Rd>, <Rm>
Note	Rd = SignExtend({Rm[7:0] , Rm[15:8]})
	Rd and Rm are low registers.

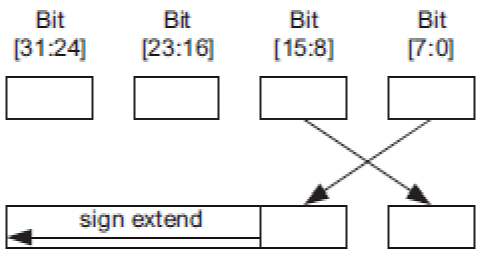


Figure 5.9:
REVSH operation.

These reverse instructions are usually used for converting data between little endian and big endian systems.

The SXTB, SXTH, UXT, and UXTH instructions are used for extending a byte or half word data into a word. They are usually used for data type conversions.

Instruction	SXTB (Signed Extended Byte)
Function	SignExtend lowest byte in a word of data
Syntax	SXTB <Rd>, <Rm>
Note	Rd = SignExtend(Rm[7:0])
	Rd and Rm are low registers.

Instruction	SXTH (Signed Extended Half Word)
Function	SignExtend lower half word in a word of data
Syntax	SXTH <Rd>, <Rm>
Note	Rd = SignExtend(Rm[15:0])
	Rd and Rm are low registers.

Instruction	UXTB (Unsigned Extended Byte)
Function	Extend lowest byte in a word of data
Syntax	UXTB <Rd>, <Rm>
Note	Rd = ZeroExtend(Rm[7:0])
	Rd and Rm are low registers.

Instruction	UXTH (Unsign Extended Half Word)
Function	Extend lower half word in a word of data
Syntax	UXTH <Rd>, <Rm>
Note	Rd = ZeroExtend(Rm[15:0]) Rd and Rm are low registers.

With SXTB or SXTH, the data are extended using bit[7] or bit[15] of the input data, whereas for UXTB and UXTH, the data are extended using zeros. For example, if R0 is 0x55AA8765, the result of these extended instructions is

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

3.2.2.12 *Program Flow Control*

There are five branch instructions in the Cortex-M0 processor. They are essential for program flow control like looping and conditional execution, and they allow program code to be partitioned into functions and subroutines.

Instruction	B (Branch)
Function	Branch to an address (unconditional)
Syntax	B <label>
Note	Branch range is +/− 2046 bytes of current program counter

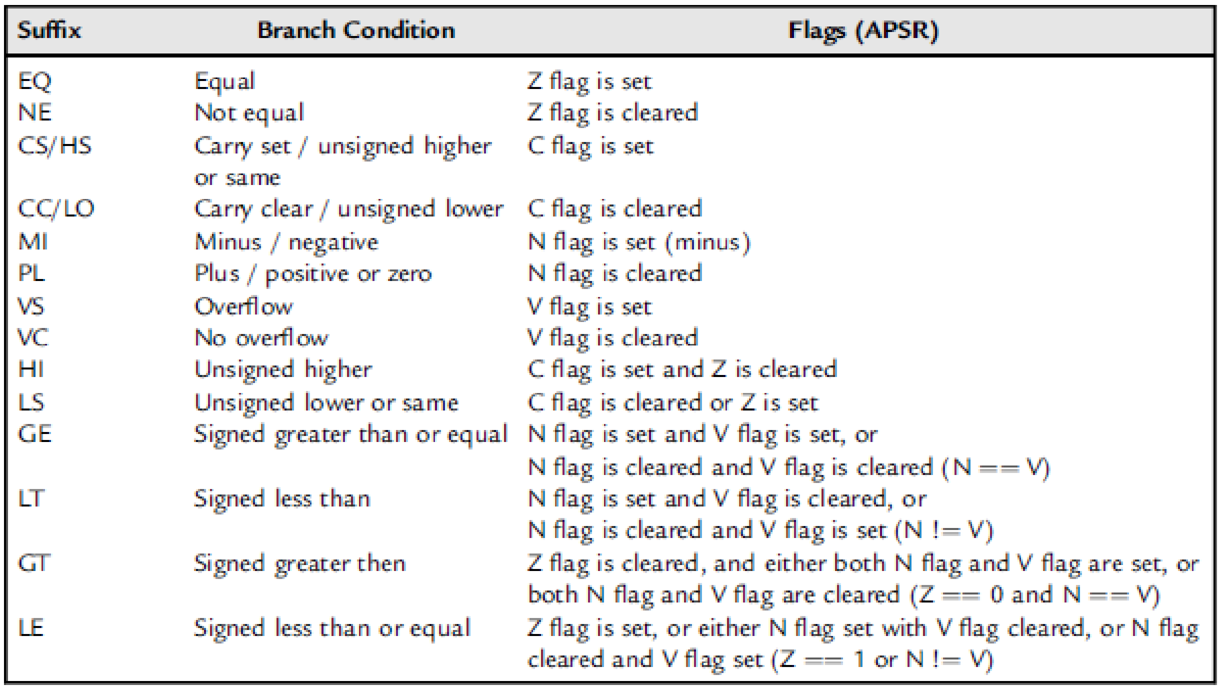
Instruction	B<cond> (Conditional Branch)
Function	Depending of APSR, branch to an address
Syntax	B<cond> <label>
Note	Branch range is +/− 254 bytes of current program counter. For example, CMP R0, #0x1 ; Compare R0 with 0x1 BEQ process1 ; Branch to process1 if R0 equal 1

The <cond> is one of the 14 possible condition suffixes (Table 5.7).

For example, a simple loop that runs three times could be

```
MOVS R0, #3 ; Loop counter starting value is 3
loop ; “loop” is an address label
SUBS R0, #1 ; Decrement by 1 and update flag
BGT loop ; branch to loop if R0 is Greater Than (GT) 1
```

Table 5.7: Condition Suffixes for Conditional Branch



Suffix	Branch Condition	Flags (APSR)
EQ	Equal	Z flag is set
NE	Not equal	Z flag is cleared
CS/HS	Carry set / unsigned higher or same	C flag is set
CC/LO	Carry clear / unsigned lower	C flag is cleared
MI	Minus / negative	N flag is set (minus)
PL	Plus / positive or zero	N flag is cleared
VS	Overflow	V flag is set
VC	No overflow	V flag is cleared
HI	Unsigned higher	C flag is set and Z is cleared
LS	Unsigned lower or same	C flag is cleared or Z is set
GE	Signed greater than or equal	N flag is set and V flag is set, or N flag is cleared and V flag is cleared (N == V)
LT	Signed less than	N flag is set and V flag is cleared, or N flag is cleared and V flag is set (N != V)
GT	Signed greater then	Z flag is cleared, and either both N flag and V flag are set, or both N flag and V flag are cleared (Z == 0 and N == V)
LE	Signed less than or equal	Z flag is set, or either N flag set with V flag cleared, or N flag cleared and V flag set (Z == 1 or N != V)

The loop will execute three times. The third time, R0 is 1 before the SUBS instruction. After the SUBS instruction, the zero flag is set, so the condition for the branch failed and the program continues execution after the BGT instruction.

Instruction	BL (Branch and Link)
Function	Branch to an address and store return address to LR. Usually use for function calls, and can be used for long-range branch that is beyond the branch range of branch instruction (B <label>).
Syntax	BL <label>
Note	Branch range is +/− 16MB of current program counter. For example, BL functionA ; call a function called functionA

Instruction	BX (Branch and Exchange)
Function	Branch to an address specified by a register, and change processor state depending on bit[0] of the register.
Syntax	BX <Rm>
Note	Because the Cortex-M0 processor only supports Thumb code, bit[0] of the register content (Rm) must be set to 1, otherwise it means it is trying to switch to the ARM state and this will generate a fault exception.

BL is commonly used for calling a subroutine or function. When it is executed, the address of the next instruction will be stored to the Link Register (LR), with the LSB set to 1. When the subroutine or function completes the required task, it can then return to the calling program by executing a "BX LR" instruction (Figure 5.10).

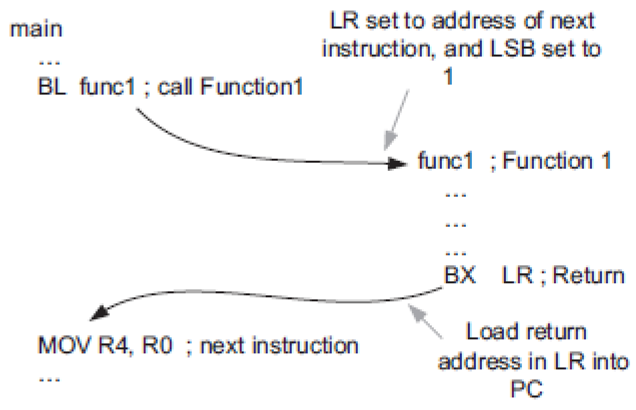
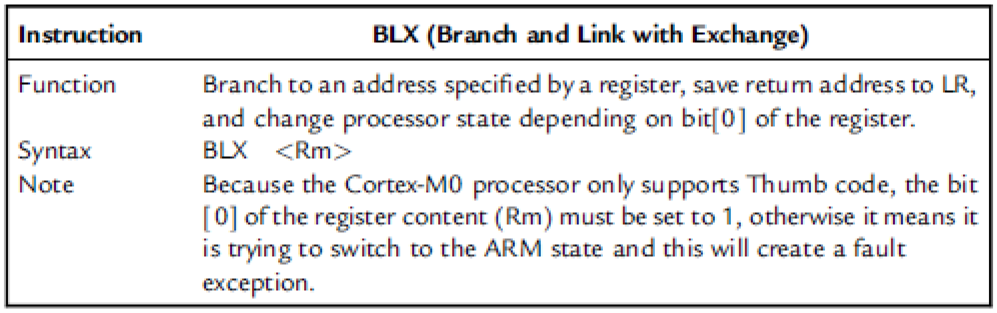

main
...
BL func1 ; call Function1
LR set to address of next instruction, and LSB set to 1
func1 ; Function 1
...
...
...
BX LR ; Return
MOV R4, R0 ; next instruction
...
Load return address in LR into PC

Figure 5.10:
Function call and return using BL and BX instructions.

BX can also be used to branch to an address that has an offset that is more than the normal branch instruction. Because the target is specified by a 32-bit register, it can branch to any address in the memory map.



Instruction	BLX (Branch and Link with Exchange)
Function	Branch to an address specified by a register, save return address to LR, and change processor state depending on bit[0] of the register.
Syntax	BLX <Rm>
Note	Because the Cortex-M0 processor only supports Thumb code, the bit [0] of the register content (Rm) must be set to 1, otherwise it means it is trying to switch to the ARM state and this will create a fault exception.

BLX is used when a function call is required but the address of the function is held inside a register (e.g., when working with function pointers).

3.2.2.13 ***Memory Barrier Instructions***

Memory barrier instructions are often needed when the memory system is complex. In some cases, if the memory barrier instruction is not used, race conditions could occur and cause system failures. For example, in some ARM processors that support simultaneous bus transfers (as a processor can have multiple memory interfaces), the transfer sequence of these transfers might overlap. If the software code relies on strict ordering of memory access sequences, it could result in software errors in corner cases. The memory barrier instructions allow the processor to stop executing the next instruction, or stop starting a new transfer, until the current memory access has completed.

Because the Cortex-M0 processor only has a single memory interface to the memory system and does not have a write buffer in the system bus interface, the memory barrier instruction is rarely needed. However, memory barriers may be necessary on other ARM processors that have more complex memory systems. If the software needs to be portable to other ARM processors, then the uses of memory barrier instructions could be essential. Therefore, the memory barrier instructions are supported on the Cortex-M0 to provide better compatibility between the Cortex-M0 processor and other ARM processors. There are three memory barrier instructions that support on the Cortex-M0 processor:

- DMB
- DSB
- ISB

Instruction	**DMB**
Function	Data Memory Barrier
Syntax	DMB
Note	Ensures that all memory accesses are completed before new memory access is committed

Instruction	**DSB**
Function	Data Synchronization Barrier
Syntax	DSB
Note	Ensures that all memory accesses are completed before the next instruction is executed

Instruction	**ISB**
Function	Instruction Synchronization Barrier
Syntax	ISB
Note	Flushes the pipeline and ensures that all previous instructions are completed before executing new instructions

Architecturally, there are various cases where these instructions are needed. Although in practice omitting the memory barrier instruction might not cause any issue on the Cortex-M0, it could be an issue when the same software is used on another ARM processor. For example, after changing the CONTROL register with MSR instruction, architecturally an ISB should be used after writing to the CONTROL register to ensure subsequent instructions use the updated settings. Although the Cortex-M0 omits the ISB instruction in this case, the omission does not cause an issue.

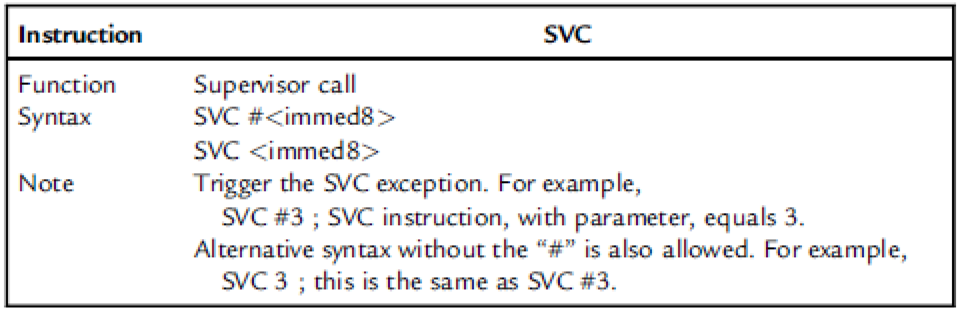
Another example is memory remap control. In some microcontrollers, a hardware register can change the memory map. After writing to the memory map switching register, you need to use the DSB instruction to ensure the write has been completed and memory configuration has been updated before carrying out the next step. Otherwise, if the memory switching is delayed, possibly because of a write buffer in the system bus interface (e.g., the Cortex-M3 has a write buffer in the system bus interface to allow higher performance), and the processor starts to access the switched memory region immediately, the access could be using the old memory mapping, or the transfer could become corrupted by the memory map switching.

Memory barrier instruction is also needed when the program contains self-modifying code. For example, if an application changes its own program code, the instruction execution that follows should use the updated program code. However, if the processor is pipelined or has a fetch buffer, the processor may have already fetched an old copy of the modified instruction. In this case, the program should use a DSB operation to ensure the write to the memory is completed; then it should use an ISB instruction to ensure the instruction fetch buffer is updated with the new instructions.

More details about memory barriers can be found in the ARMv6-M Architecture Reference manual (reference 3).

3.2.2.14 *Exception-Related Instructions*

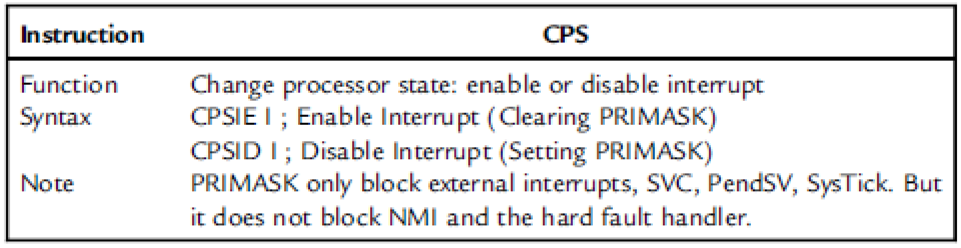
The Cortex-M0 processor provides an instruction called supervisor call (SVC). This instruction causes the SVC exception to take place immediately if the exception priority level of SVC is higher than current level.



Instruction	SVC
Function	Supervisor call
Syntax	SVC #<immed8> SVC <immed8>
Note	Trigger the SVC exception. For example, SVC #3 ; SVC instruction, with parameter, equals 3. Alternative syntax without the "#" is also allowed. For example, SVC 3 ; this is the same as SVC #3.

An 8-bit immediate data element is used with SVC instruction. This parameter does not affect the SVC exception directly, but it can be extracted by the SVC handler and be used as an input to the SVC function. Typically the SVC can be used to provide access to system service or the application programming interface (API), and this parameter can be used to indicate which system service is required.

If the SVC instruction is used in an exception handler that has the same or a higher priority than the SVC, this will cause a fault exception. As a result, the SVC cannot be used in the hard fault handler, the NMI handler, or the SVC handler itself. Besides using MSR instruction, the PRIMASK special register can also be changed using an instruction called CPS:

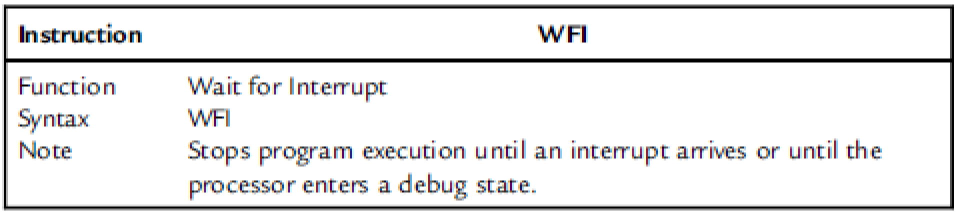


Instruction	CPS
Function	Change processor state: enable or disable interrupt
Syntax	CPSIE I ; Enable Interrupt (Clearing PRIMASK) CPSID I ; Disable Interrupt (Setting PRIMASK)
Note	PRIMASK only block external interrupts, SVC, PendSV, SysTick. But it does not block NMI and the hard fault handler.

The switching of PRIMASK to disable and enable the interrupt is commonly used for timing critical code.

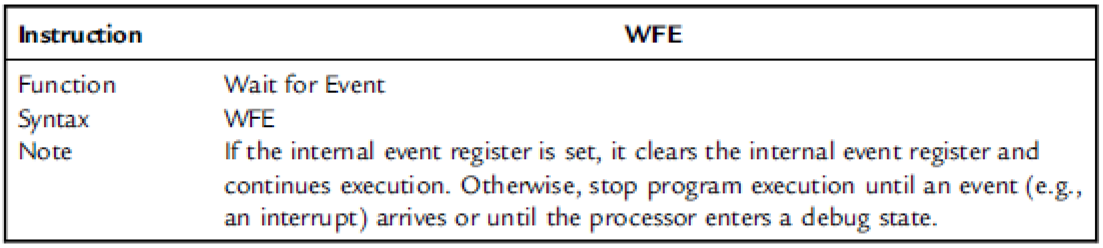
Sleep Mode FeatureeRelated Instructions

The Cortex-M0 processor can enter sleep mode by executing the Wait-for-Interrupt (WFI) and Wait-for-Event (WFE) instructions. Note that for the Cortex-M1 processor, as the design is implemented in a FPGA design, which does not have sleep mode, these two instructions execute as NOP and will not cause the processor to stop.



Instruction	WFI
Function	Wait for Interrupt
Syntax	WFI
Note	Stops program execution until an interrupt arrives or until the processor enters a debug state.

WFE is just like WFI, except that it can also be awakened by events. An event can be an interrupt, the execution of an SEV instruction (see next page), or the entering of a debug state. A previous event also affects a WFE instruction: Inside the Cortex-M0 processor, there is an event register that records whether an event has occurred (exceptions, external events, or the execution of an SEV instruction). If the event register is not set when the WFE is executed, the WFE instruction execution will cause the processor to enter sleep mode. If the event register is set when WFE is executed, it will cause the event register to be cleared and the processor proceeds to the next instruction.



Instruction	WFE
Function	Wait for Event
Syntax	WFE
Note	If the internal event register is set, it clears the internal event register and continues execution. Otherwise, stop program execution until an event (e.g., an interrupt) arrives or until the processor enters a debug state.

WFE can also be awakened by an external event input signal, which is normally used in a multiprocessing environment. The Send Event (SEV) instruction is normally used in multiprocessor systems to wake up other processors that are in sleep mode by means of the WFE instruction. For single-processor systems, where the processor does not have a multiprocessor communication interface or the multiprocessor communication interface is not used, the SEV can only affect the local event register inside the processor itself.

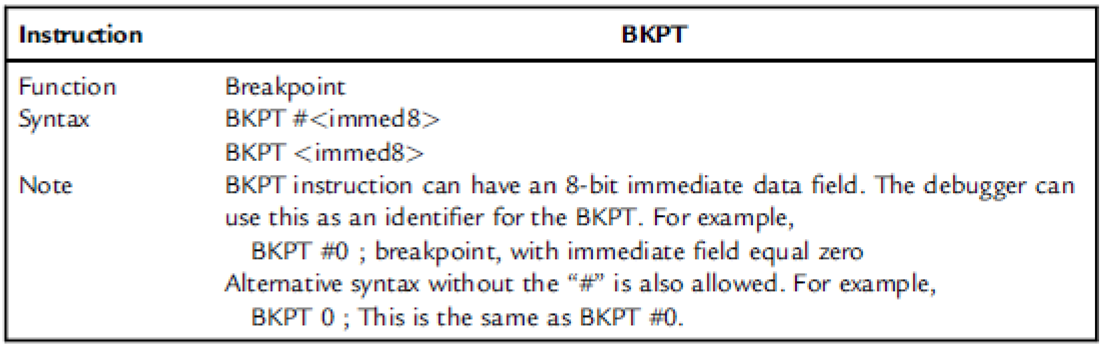
Instruction	SEV
Function	Send event to all processors in multiprocessing environment (including itself)
Syntax	SEV
Note	Set local event register and send out an event pulse to other microprocessor in a multiple processor system

3.2.2.15 Other Instructions

The Cortex-M0 processor supports an NOP instruction. This instruction can be used to produce instruction alignment or to introduce delay.

Instruction	NOP
Function	No operation
Syntax	NOP
Note	The NOP instruction takes one cycle minimum on Cortex-M0. In general, delay timing produced by NOP instruction is not guaranteed and can vary among different systems (e.g., memory wait states, processor type). If the timing delay needs to be accurate, a hardware timer should be used.

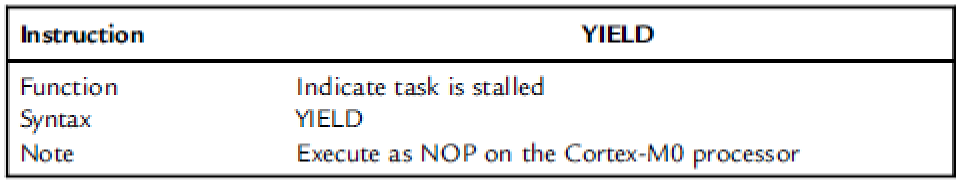
The breakpoint instruction is used to provide a breakpoint function during debug. Usually a debugger, replacing the original instruction, inserts this instruction. When the breakpoint is hit, the processor would be halted, and the user can then carry out the debug tasks through the debugger. The Cortex-M0 processor also has a hardware breakpoint unit. This is limited to four breakpoints. Because many microcontrollers use flash memory, which can be reprogrammed a number of times, using software breakpoint instruction allows more breakpoints to be set at no extra cost. The breakpoint instruction has an 8-bit immediate data field. This immediate value does not affect the breakpoint operation directly, but the debugger can extract this value and use it for debug operation.



Instruction	BKPT
Function	Breakpoint
Syntax	BKPT #<immed8> BKPT <immed8>
Note	BKPT instruction can have an 8-bit immediate data field. The debugger can use this as an identifier for the BKPT. For example, BKPT #0 ; breakpoint, with immediate field equal zero Alternative syntax without the "#" is also allowed. For example, BKPT 0 ; This is the same as BKPT #0.

The YIELD instruction is a hint instruction targeted for embedded operating systems. This is not implemented in the current releases of the Cortex-M0 processor and executes as NOP.

When used in multithread systems, YIELD can indicate that the current thread is delayed (e.g., waiting for hardware) and can be swapped out. In this case, the processor does not have to spend too much time on an idle task and can switch to other tasks earlier to get better system throughput. On the Cortex-M0 processor, this instruction is executed as an NOP (no operation) because it does not have special support for multithreading. This instruction is included for better software compatibility with other ARM processors.

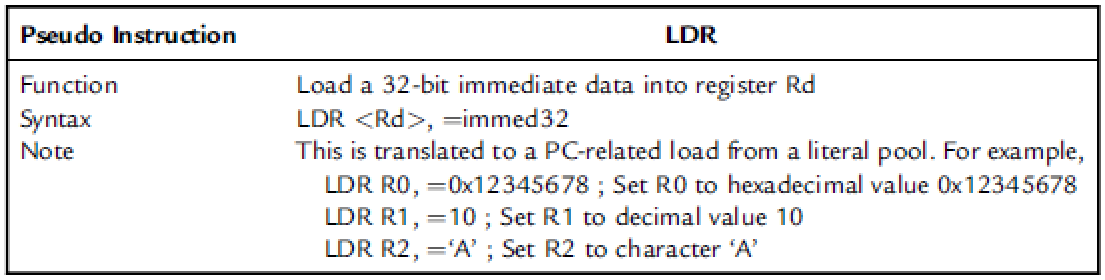


Instruction	YIELD
Function	Indicate task is stalled
Syntax	YIELD
Note	Execute as NOP on the Cortex-M0 processor

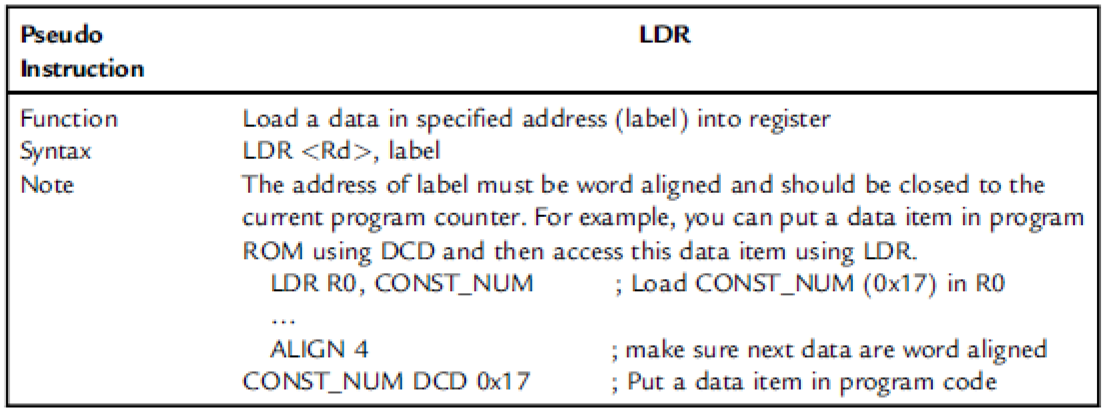
3.2.2.16 *Pseudo Instructions*

Apart from the instructions listed in the previous section, a few pseudo instructions are also available. The pseudo instructions are provided by the assembler tools, which convert them into one or more real instructions.

The most commonly used pseudo instruction is the LDR. This allows a 32-bit immediate data item to be loaded into a register.



Pseudo Instruction	LDR
Function	Load a 32-bit immediate data into register Rd
Syntax	LDR <Rd>, =immed32
Note	This is translated to a PC-related load from a literal pool. For example,
	LDR R0, =0x12345678 ; Set R0 to hexadecimal value 0x12345678
	LDR R1, =10 ; Set R1 to decimal value 10
	LDR R2, ='A' ; Set R2 to character 'A'



Pseudo Instruction	LDR
Function	Load a data in specified address (label) into register
Syntax	LDR <Rd>, label
Note	The address of label must be word aligned and should be closed to the current program counter. For example, you can put a data item in program ROM using DCD and then access this data item using LDR.

```
   LDR R0, CONST_NUM        ; Load CONST_NUM (0x17) in R0
   ...
   ALIGN 4                  ; make sure next data are word aligned
CONST_NUM DCD 0x17          ; Put a data item in program code
```

Other pseudo instructions depend on the tool chain being used. For more information, please refer to the tools documentation for details.

3.3 Instruction usage Examples

3.3.1 *Overview*

In the previous chapter we looked at the instruction set of the Cortex-M0 processor. In this chapter we will see how these instructions are used to carry out various operations. The examples in this chapter are useful for understanding the instruction set. Because most embedded programmers write their program in C, there is no need for most application to write code in assembly, as illustrated in these examples.

The following examples are written based on ARM assembly syntax. For the GNU assembler, the syntax is different in a number of ways, as highlighted in the previous chapter.

3.3.2 *Program Control*

3.3.2.1 If-Then-Else

One the most important functions of the instruction set is to handle conditional branches. For example, if we need to carry out the task

```
if (counter > 10) then
counter = 0
else
counter = counter + 1
```

Assume the R0 is used as a "counter" variable; the preceding operation can be implemented as

```
CMP R0, #10 ; compare to 10
BLE incr_counter ; if less or equal, then branch
MOVS R0, #0 ; counter = 0
B counter_done ; branch to counter_done
incr_counter
ADDS R0, R0, #1 ; counter = counter+1
counter_done
```

.

The program code first carries out a compare and then executes a conditional branch. The program then carries out a required task and finishes at the program address labeled as "counter_done."

3.3.2.3 Loop

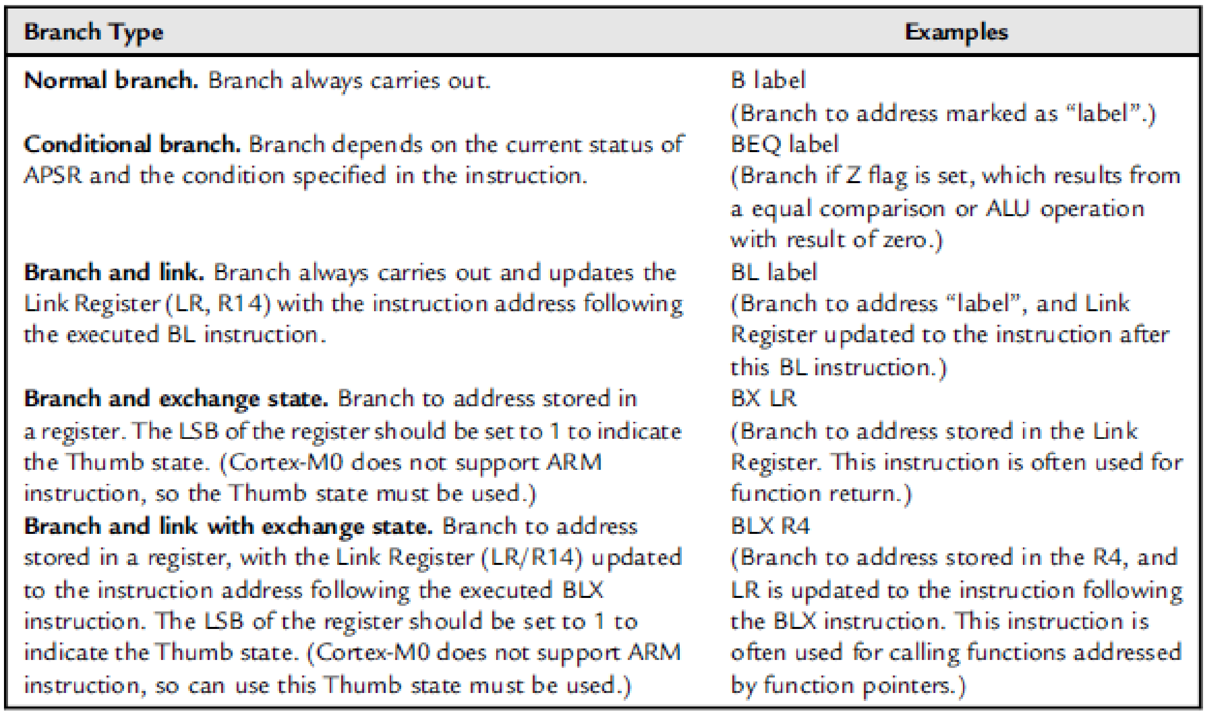
Another important program control operation is looping. For example,

```
Total = 0;
for (i=0;i<5;i=i+1)
        Total = Total + i;
Assume "Total" is R0 and "i" is R1; the program can be implemented as
        MOVS R0, #0 ; Total = 0
        MOVS R1, #0 ; i = 0
loop
        ADDS R0, R0, R1 ; Total = Total + i
        ADDS R1, R1, #1 ; i=i + 1
        CMP R1, #5 ; compare i to 5
        BLT loop ; if less than then branch to loop
```

3.3.2.4 *More on the Branch Instructions*

As Table 6.1 illustrates, there are various branch instructions.

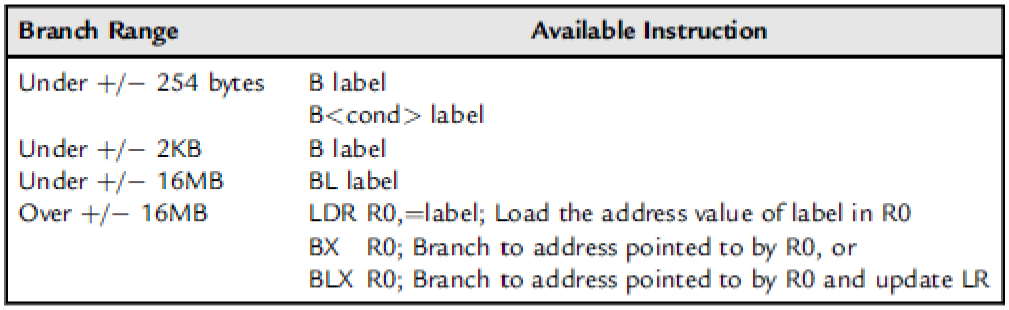
Table 6.1: Various Branch Instructions



Branch Type	Examples
Normal branch. Branch always carries out.	B label (Branch to address marked as "label".)
Conditional branch. Branch depends on the current status of APSR and the condition specified in the instruction.	BEQ label (Branch if Z flag is set, which results from a equal comparison or ALU operation with result of zero.)
Branch and link. Branch always carries out and updates the Link Register (LR, R14) with the instruction address following the executed BL instruction.	BL label (Branch to address "label", and Link Register updated to the instruction after this BL instruction.)
Branch and exchange state. Branch to address stored in a register. The LSB of the register should be set to 1 to indicate the Thumb state. (Cortex-M0 does not support ARM instruction, so the Thumb state must be used.)	BX LR (Branch to address stored in the Link Register. This instruction is often used for function return.)
Branch and link with exchange state. Branch to address stored in a register, with the Link Register (LR/R14) updated to the instruction address following the executed BLX instruction. The LSB of the register should be set to 1 to indicate the Thumb state. (Cortex-M0 does not support ARM instruction, so can use this Thumb state must be used.)	BLX R4 (Branch to address stored in the R4, and LR is updated to the instruction following the BLX instruction. This instruction is often used for calling functions addressed by function pointers.)

The BL instruction (Branch and Link) is usually used for calling functions. It can also be used for normal branch operations when a long branch range is required. If the branch target offset is more than 16MB, we can use the BX instruction instead. An example is illustrated in Table 6.2.

Table 6.2: Instruction for Branch Range

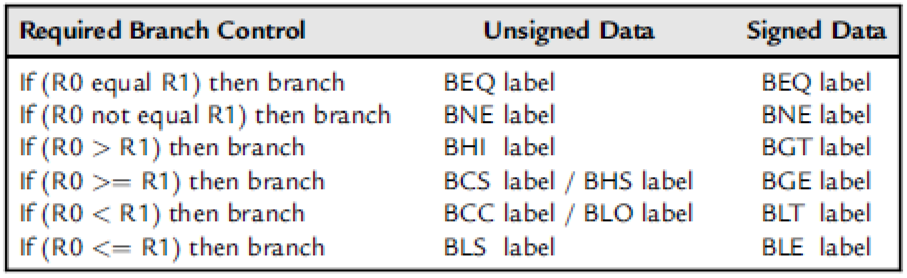


Branch Range	Available Instruction
Under +/− 254 bytes	B label
	B<cond> label
Under +/− 2KB	B label
Under +/− 16MB	BL label
Over +/− 16MB	LDR R0,=label; Load the address value of label in R0
	BX R0; Branch to address pointed to by R0, or
	BLX R0; Branch to address pointed to by R0 and update LR

3.3.2.5 *Typical Usages of Branch Conditions*

A number of conditions are available for the conditional branch. They allow the result of signed and unsigned data operations or compare operations to be used for branch control. For example, if we need to carry out a conditional branch after a compare operation "CMP R0, R1," we can use one of the conditional branch instructions shown in Table 6.3.

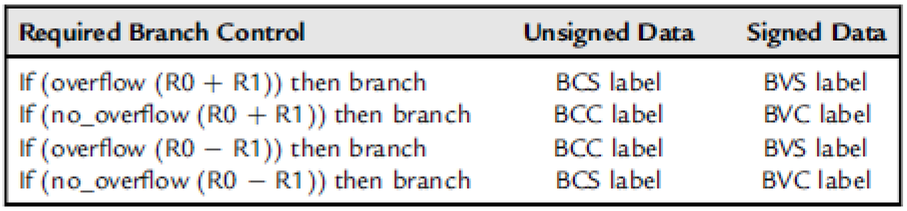
Table 6.3: Conditional Branch Instructions for Value Comparison



Required Branch Control	Unsigned Data	Signed Data
If (R0 equal R1) then branch	BEQ label	BEQ label
If (R0 not equal R1) then branch	BNE label	BNE label
If (R0 > R1) then branch	BHI label	BGT label
If (R0 >= R1) then branch	BCS label / BHS label	BGE label
If (R0 < R1) then branch	BCC label / BLO label	BLT label
If (R0 <= R1) then branch	BLS label	BLE label

To detect value overflow in add or subtract operations, we can use the instructions shown in Table 6.4.

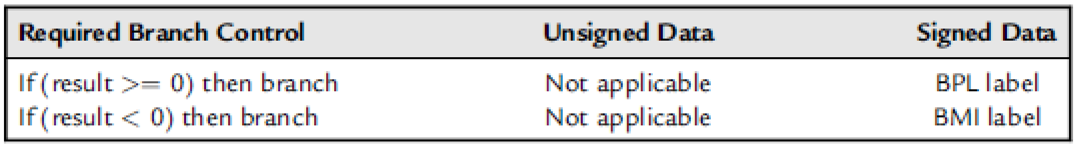
Table 6.4: Conditional Branch Instructions for Overflow Detection



Required Branch Control	Unsigned Data	Signed Data
If (overflow (R0 + R1)) then branch	BCS label	BVS label
If (no_overflow (R0 + R1)) then branch	BCC label	BVC label
If (overflow (R0 − R1)) then branch	BCC label	BVS label
If (no_overflow (R0 − R1)) then branch	BCS label	BVC label

To detect whether an operation result is a positive value or negative value (signed data), the “PL” and “MI” suffixes can be used for the conditional branch (Table 6.5).

Table 6.5: Conditional Branch Instructions for Positive or Negative Value Detection



Required Branch Control	Unsigned Data	Signed Data
If (result >= 0) then branch	Not applicable	BPL label
If (result < 0) then branch	Not applicable	BMI label

Apart from using the compare (CMP) instruction, conditional branches can also be controlled by the result of arithmetic operations and logical operations, or instructions like compare negative (CMN) and test (TST). For example, a simple loop that executes five times can be written as

```
        MOVS R0, #5 ; Loop counter
loop
        SUBS R0, R0, #1 ; Decrement loop counter
        BNE loop ; if result is not 0 then branch to loop
```

A polling loop that waits until a status register bit 3 to be set can be written as

```
        LDR R0, ¼Status ; Load address of status register in R0
        MOVS R2, #0x8 ; Bit 3 is set
loop
        LDR R1, [R0] ; Read the status register
        TST R1, R2 ; Compute “R1 AND 0x8”
        BEQ loop ; if result is 0 then try again
```

3.3.2.6 ***Function Calls and Function Returns***

When carrying out function call (or subroutine call), we need to save the return address, which is the address of the instruction following the call instruction, so that we can resume the execution of the current instruction sequence. There are two instructions that can be used for the function call, as shown in Table 6.6.

Table 6.6: Instructions for Function or Subroutine Calls

Instruction Example	Scenarios
BL function	Target function address is fixed, and the offset is within +/− 16MB.
LDR R0, =function; (other registers could also be used) BLX R0	Target function address can be changed during run time, or the offset is over +/− 16MB.

After executing the BL/BLX instructions, the return address is stored in the Link Register (LR/R14) for function return when the function completed. In the simple cases, the function executed will be terminated using "BX LR" (Figure 6.1).

If the value of LR can be changed during "FunctionA," we will need to save the return address to prevent it from being lost. This happens when the BL or BLX instruction is executed within FunctionA, for example, when a nested function call is required. For illustration, Figure 6.2 shows when FunctionA calls another function, called FunctionB. In the Cortex-M0 processor, you can push multiple low registers (R0 to R7) and the return address in LR onto the stack with just one instruction. Similarly, you can carry out the pop

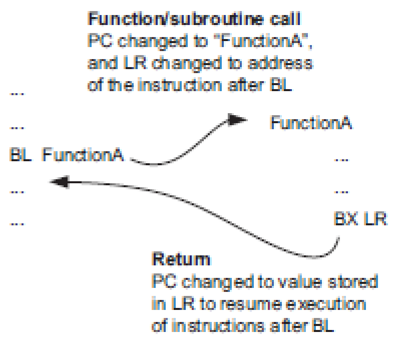

Function/subroutine call
PC changed to "FunctionA", and LR changed to address of the instruction after BL
...
...
BL FunctionA
...
...
FunctionA
...
...
BX LR
Return
PC changed to value stored in LR to resume execution of instructions after BL

Figure 6.1:
Simple function call and function return.

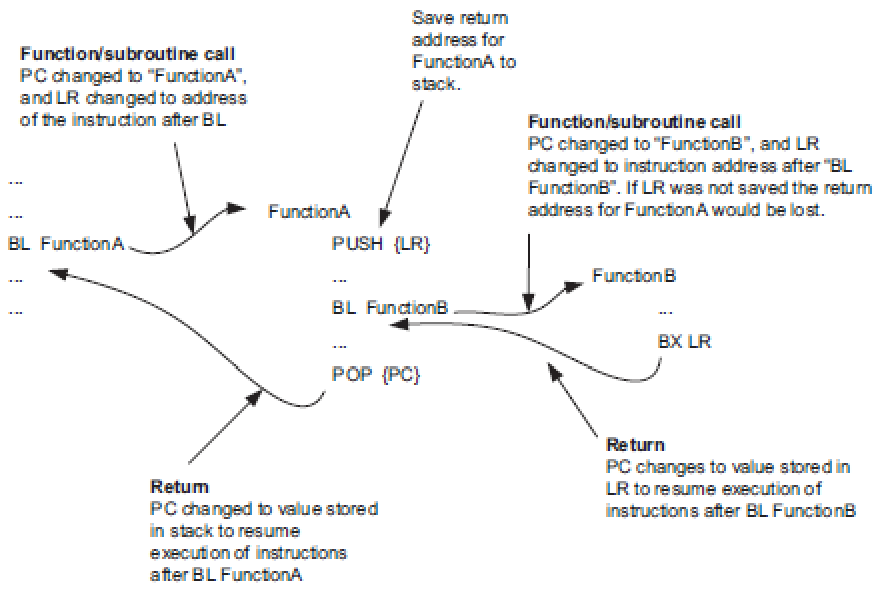

Function/subroutine call
PC changed to "FunctionA", and LR changed to address of the instruction after BL
Save return address for FunctionA to stack.
Function/subroutine call
PC changed to "FunctionB", and LR changed to instruction address after "BL FunctionB". If LR was not saved the return address for FunctionA would be lost.
...
...
BL FunctionA
...
...
FunctionA
PUSH {LR}
...
BL FunctionB
...
POP {PC}
FunctionB
...
BX LR
Return
PC changes to value stored in LR to resume execution of instructions after BL FunctionB
Return
PC changed to value stored in stack to resume execution of instructions after BL FunctionA

Figure 6.2:

operation to low registers and the Program Counter (PC) in one instruction. This allows you to combine register values restore and return with a single instruction. For example, if the registers R4 to R6 are being modified in "FunctionA," and needed to be saved to the stack, we can write "FunctionA," as shown in Figure 6.3.

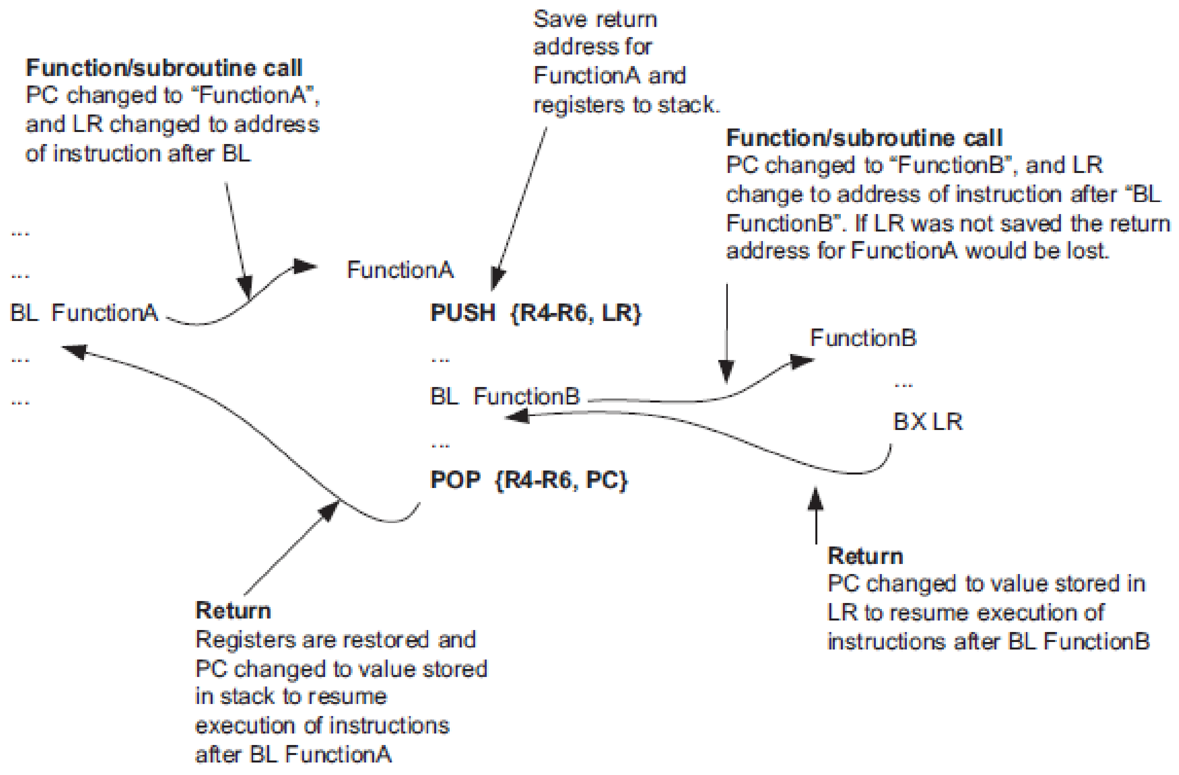

Function/subroutine call
PC changed to "FunctionA", and LR changed to address of instruction after BL
Save return address for FunctionA and registers to stack.
Function/subroutine call
PC changed to "FunctionB", and LR change to address of instruction after "BL FunctionB". If LR was not saved the return address for FunctionA would be lost.
...
...
BL FunctionA
...
...
FunctionA
PUSH {R4-R6, LR}
...
BL FunctionB
...
POP {R4-R6, PC}
FunctionB
...
BX LR
Return
Registers are restored and PC changed to value stored in stack to resume execution of instructions after BL FunctionA
Return
PC changed to value stored in LR to resume execution of instructions after BL FunctionB

Figure 6.3:
Using push and pop of multiple registers in functions.

3.3.2.7 Branch Table

In C programming, sometime we use the "switch" statement to allow a program to branch to multiple possible address locations based on an input. In assembly programming, we can handle the same operation by creating a table of branch destination addresses, issue a load (LDR) to the table with offset computed from the input, and then use BX to carry out the branch. In the following example, we have a selection input of 0 to 3 in R0, which allows the program to branch to Dest0 to Dest3. If the input value is larger than 3, it will cause a branch to the default case:

```
CMP R0, #3              ; Compare input to maximum valid choice
BHI default_case         ; Branch to default case if higher than 3
MOVS R2, #4             ; Multiply branch table offset by 4
MULS R0, R2, R0         ; (size of each entry)
LDR R1,¼BranchTable      ; Get base address of branch table
LDR R2,[R1,R0]           ; Get the actual branch destination
BX R2                   ; Branch to destination
ALIGN 4                 ; Alignment control. The table has
```

```
                                        ; to be word aligned to prevent unaligned read
        BranchTable                     ; table of each destination addresses
                DCD Dest0
                DCD Dest1
                DCD Dest2
                DCD Dest3
        default_case
                . ; Instructions for default case
        Dest0
                . ; Instructions for case ‘0’
        Dest1
                . ; Instructions for case ‘1’
        Dest2
                . ; Instructions for case ‘2’
        Dest3
                . ; Instructions for case ‘3’
```

Additional examples of complex branch conditional handling are presented in Chapter 16.

3.3.3 Data Accesses

Data accesses are vital to embedded applications. The Cortex-M0 processor provides a number of load (memory read) and store (memory write) instructions with various address modes. Here we will go through a number of typical examples of how these instructions can be applied.

3.3.3.1 Simple Data Accesses

Normally the memory locations (physical addresses) of software variables are defined by the linker. However, we can write the software code to access to the variables as long as we know the symbol of the variables. For example, if we need to calculate the sum of an integer array “DataIn” with 10 elements (32-bit each) and put the result in another variable called “Sum” (also 32-bit), we can use the following assembly code:

```
            LDR r0,=DataIn            ; Get the address of variable 'DataIn'
            MOVS r1, #10              ; loop counter
            MOVS r2, #0               ; Result - starting from 0
add_loop
            LDM r0!,{r3}              ; Load result and increment address
            ADDS r2, r3               ; add to result
            SUBS r1, #1               ; increment loop counter
            BNE add_loop
            LDR r0,=Sum                ; Get the address of variable 'Sum'
            STR r2,[r0]               ; Save result to Sum
```

In the preceding example, we use the LDMinstruction rather than a normal LDR instruction. This allows us to read the memory and increment the address to the next array element at the same time.

When using assembly to access data, we need to pay attention to a few things:

• Use correct instruction for corresponding data size. Different instructions are available for different data sizes.

• Make sure that the access is aligned. If an access is unaligned, it will trigger a fault exception. This can happen if an instruction of incorrect data size is used to access a data.

• Various addressing modes are available and can simplify your assembly codes. For example, when programming/accessing a peripheral, you can set a register to its base address value and then use an immediate offset addressing mode for accessing each register. In this way, you do not have to set up the register address every time a different register is accessed.

3.3.3.2 Example of using Memory Access Instruction

To demonstrate how different memory access instructions can be used, this section presents several simple examples of memory copying functions. The most basic approach is to copy the data byte by byte, thus allowing any number of bytes to be copied, and this approach does not have memory alignment issues:

```
        LDR r0,=0x00000000              ; Source address
        LDR r1,=0x20000000              ; Destination address
        LDR r2,=100                     ; number of bytes to copy
copy_loop
        LDRB r3, [r0]                   ; read 1 byte
        ADDS r0, r0, #1                 ; increment source pointer
        STRB r3, [r1]                   ; write 1 byte
        ADDS r1, r1, #1                 ; increment destination pointer
        SUBS r2, r2, #1                 ; decrement loop counter
        BNE copy_loop                   ; loop until all data copied
```

The program code uses a number of add and subtract instructions in the loop, which reduce the performance. We could modify the code to reduce the program size using a register offset address mode:

```
        LDR r0,=0x00000000              ; Source address
        LDR r1,=0x20000000              ; Destination address
        LDR r2,=100                     ; number of bytes to copy, also
copy_loop                               ; acts as loop counter
        SUBS r2, r2, #1                 ; decrement offset and loop counter
        LDRB r4,[r0, r2]                ; read 1 byte
        STRB r4,[r1, r2]                ; write 1 byte
        BNE copy_loop                   ; loop until all data copied
```

By using the loop counter as a memory offset, we have reduced the code size and improved execution speed. The only side effect is that the copying operation will start from the end of the memory block and finish at the start of the memory block.

For copying large amounts of data, we can use multiple load and store instructions to increase the performance. Because the load store multiple instructions can only be used with word accesses, we usually use them in memory-copying functions only when we know that the size of the memory being copied is large and the data are word aligned:

```
          LDR r0,=0x00000000                ; Source address
          LDR r1,=0x20000000                ; Destination address
          LDR r2,=128                        ; number of bytes to copy, also
copy_loop                                   ; acts as loop counter
          LDMIA r0!,{r4-r7}                 ; Read 4 words and increment r0
          STMIA r1!,{r4-r7}                  ; Store 4 words and increment r1
          LDMIA r0!,{r4-r7}                 ; Read 4 words and increment r0
          STMIA r1!,{r4-r7}                  ; Store 4 words and increment r1
          LDMIA r0!,{r4-r7}                  ; Read 4 words and increment r0
          STMIA r1!,{r4-r7}                 ; Store 4 words and increment r1
          LDMIA r0!,{r4-r7}                 ; Read 4 words and increment r0
          STMIA r1!,{r4-r7}                 ; Store 4 words and increment r1
          SUBS r2, r2, #64                  ; Each time 64 bytes are copied
          BNE copy_loop                     ; loop until all data copied
```

In the preceding code, each loop iteration copies 64 bytes. This greatly increases the performance of data transfer.

Another type of useful memory access instruction is the load and store instruction with stack pointer-related addressing. This is commonly used for local variables, as C compilers often store simple local variables on the stack memory. For example, let’s say we need to create two local variables in a function called “function1.” The code can be written as follows:

```
function1
          SUB SP, SP, #0x8          ; Reserve 2 words of stack
                                    ;(8 bytes) for local variables
                                    ;Data processing in function
          MOVS r0, #0x12            ; set a dummy value
          STR r0, [sp, #0]          ; Store 0x12 in 1st local variable
          STR r0, [sp, #4]          ; Store 0x12 in 2nd local variable
          LDR r1, [sp, #0]          ; Read from 1st local variable
          LDR r2, [sp, #4]          ; Read from 2nd local variable
          ADD SP, SP, #0x8          ; Restore SP to original position
          BX LX
```

In the beginning of the function, a stack pointer adjustment is carried out so that the data reserved will not be overwritten by further stack push operations. During the execution of the function, SP-related addressing with immediate offset allows the local variables to be accessed efficiently. The value of SP can also be copied to another register if further stack operations are required or if the some of the local variables are in byte or half-word size (in ARMv6-M, SP-related addressing mode only supports word-size data). In such cases, load/store instructions accessing the local variables would use the copied version of SP (Figure 6.4).

At the end of the function, the local variables can be discarded and we restore the SP value to the position as when the function started using an ADD instruction.

3.3.4 *Data Type Conversion*

The Cortex-M0 processor supports a number of instructions for converting data among different data types.

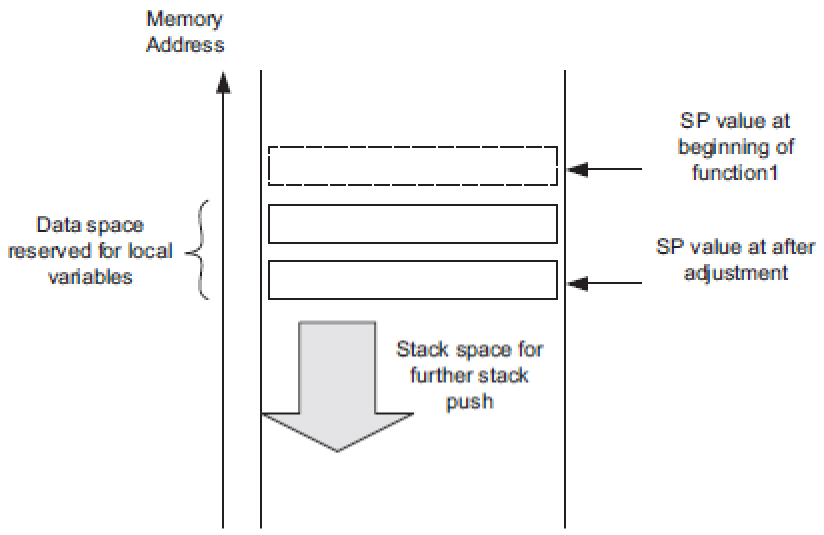

Memory
Address
SP value at beginning of function1
Data space reserved for local variables
SP value at after adjustment
Stack space for further stack push

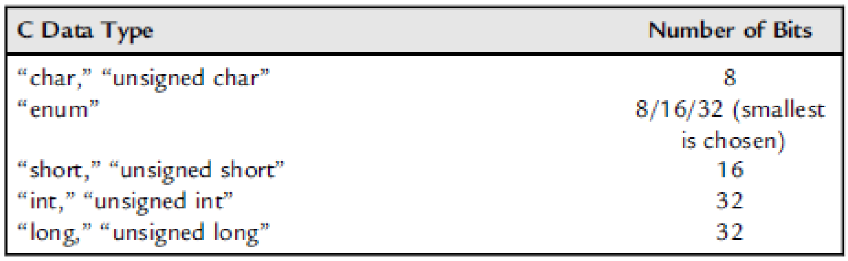
Figure 6.4:
Reserving two words of stack space for local variables.

3.3.4.1 Conversion of Data Size

On ARM compilers, different data types have different sizes. Table 6.7 shows a number of commonly used data types and their corresponding sizes on ARM compilers.

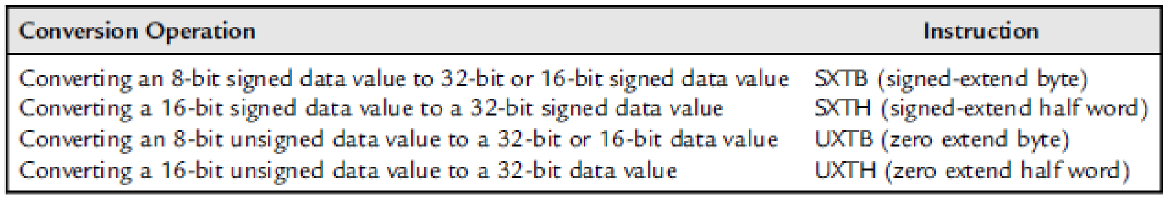
When converting a data value from one type to another type with a larger size, we need to signextend or zero-extend it. A number of instructions are available to handle this conversion (Table 6.8).

Table 6.7: Size of Commonly Used Data Types in C Compilers for ARM



C Data Type	Number of Bits
"char," "unsigned char"	8
"enum"	8/16/32 (smallest is chosen)
"short," "unsigned short"	16
"int," "unsigned int"	32
"long," "unsigned long"	32

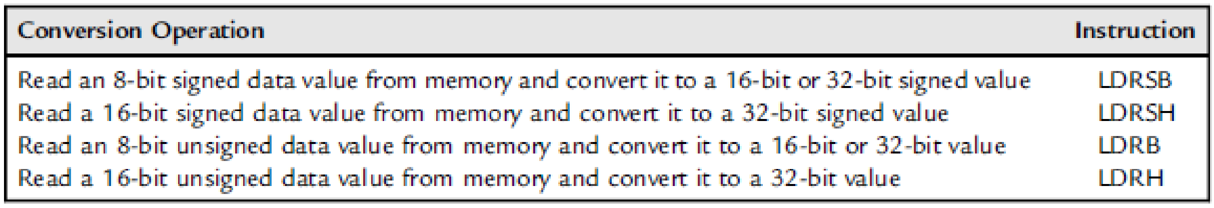
Table 6.8: Instructions for Signed-Extend and Zero-Extend Data Value Operations



Conversion Operation	Instruction
Converting an 8-bit signed data value to 32-bit or 16-bit signed data value	SXTB (signed-extend byte)
Converting a 16-bit signed data value to a 32-bit signed data value	SXTH (signed-extend half word)
Converting an 8-bit unsigned data value to a 32-bit or 16-bit data value	UXTB (zero extend byte)
Converting a 16-bit unsigned data value to a 32-bit data value	UXTH (zero extend half word)

If the data are in the memory, we can read the data and carry out the zero-extend or signed extend operation in a single instruction (Table 6.9).

Table 6.9: Memory Read Instructions with Signed-Extend and Zero Extend Data Value Operations

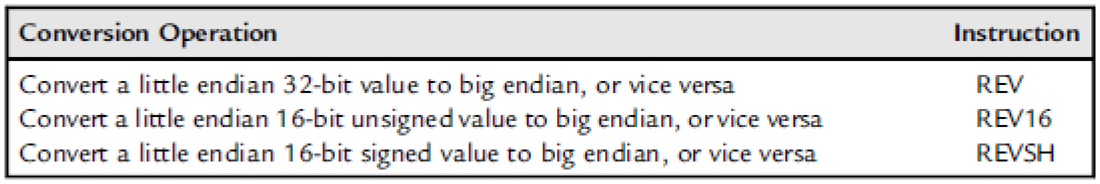


Conversion Operation	Instruction
Read an 8-bit signed data value from memory and convert it to a 16-bit or 32-bit signed value	LDRSB
Read a 16-bit signed data value from memory and convert it to a 32-bit signed value	LDRSH
Read an 8-bit unsigned data value from memory and convert it to a 16-bit or 32-bit value	LDRB
Read a 16-bit unsigned data value from memory and convert it to a 32-bit value	LDRH

3.3.4.2 Endian Conversion

The memory system of a Cortex-M0 microcontroller can be in either little endian configuration or big endian configuration. It is defined in hardware and cannot be changed by programming. Occasionally we might need to convert data between little endian and big endian format. Table 6.10 presents several instructions to handle this situation.

Table 6.10: Instructions for Conversion between Big Endian and Little Endian Data



Conversion Operation	Instruction
Convert a little endian 32-bit value to big endian, or vice versa	REV
Convert a little endian 16-bit unsigned value to big endian, or vice versa	REV16
Convert a little endian 16-bit signed value to big endian, or vice versa	REVSH

3.3.5 *Data Processing*

Most of the data processing operations can be carried out in a simple instruction sequence. However, there are situations when more steps are required. Here we will look at a number of examples.

3.3.5.1 64-Bit/128-Bit Add

Adding two 64-bit values together is fairly straightforward. Assume that you have two 64-bit values (X and Y) stored in four registers. You can add them together using ADDS followed up by ADCS instruction:

```
LDR r0,=0xFFFFFFFF          ; X_Low (X ¼ 0x3333FFFFFFFFFFFF)
LDR r1,=0x3333FFFF          ; X_High
LDR r2,=0x00000001          ; Y_Low (Y ¼ 0x3333000000000001)
LDR r3,=0x33330000          ; Y_High
ADDS r0,r0,r2               ; lower 32-bit
ADCS r1,r1,r3               ; upper 32-bit
```

In this example, the result is in R1, R0, which is 0x66670000 and 0x00000000. The operation can be extended to 96-bit values, 128-bit values, or more by increasing the number of ADCS instructions in the sequence (Figure 6.5).

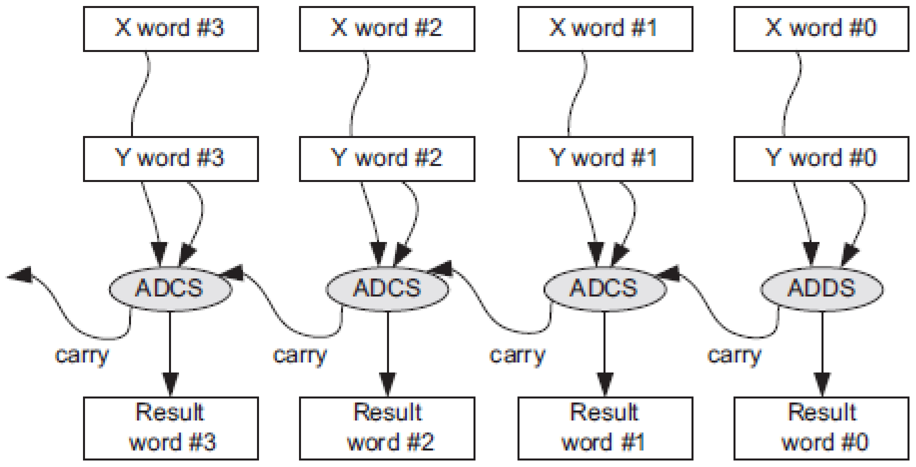


Figure 6.5:
Adding of two 128-bit numbers.

3.3.5.2 64-Bit/128-Bit Sub

The operation of 64-bit subtract is similar to the one for 64-bit add. Assume that you have two 64-bit values (X and Y) in four registers. You can subtract them (X _ Y) using SUBS followed by SBCS instruction:

```
LDR r0,=0x00000001              ; X_Low(X = 0x0000000100000001)
LDR r1,=0x00000001               ; X_High
LDR r2,=0x00000003              ; Y_Low(Y = 0x0000000000000003)
LDR r3,=0x00000000              ; Y_High
SUBS r0,r0,r2                   ; lower 32-bit
SBCS r1,r1,r3                   ; upper 32-bit
```

In this example, the result is in R1, R0, which is 0x00000000 and 0xFFFFFFFE. The operation can be extended to 96-bit values, 128-bit values, or more by increasing the number of SBCS instructions in the sequence (Figure 6.6).

3.3.5.3 Integer Divide

Unlike the Cortex-M3/M4 processor, the Cortex-M0 processor does not have integer divide instructions. For users who program their application in C language, the C compiler automatically inserts the required C library function that handles integer divide if required. Users who prefer to write their application entirely in assembly language can create an assembly function like that shown in Figure 6.7, which handles unsigned integer divide.

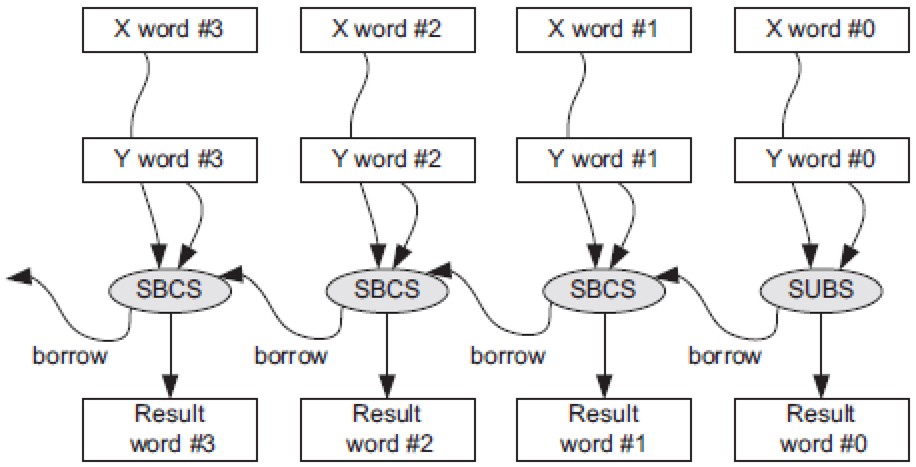


Figure 6.6:
Subtracting two 128-bit values.

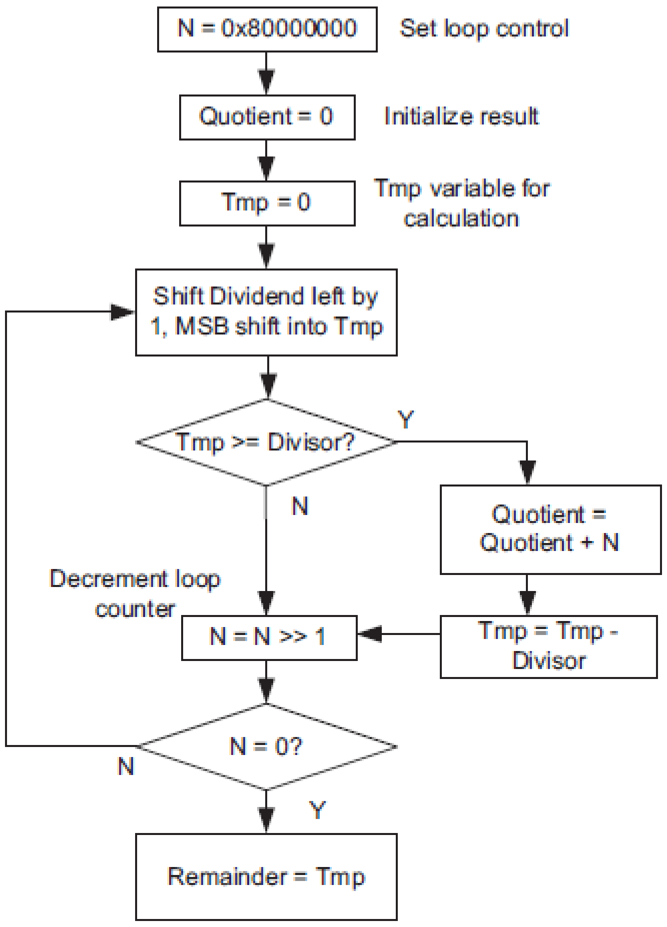


Figure 6.7:
Simple unsigned integer divide function.

The divide function contains a loop that iterates 32 times and computes 1 bit of the result each time. Instead of using an integer loop counter, the loop control is done by a value N, which has 1 bit set (one hot), and shifts right by 1 bit each time the loop is executed. The corresponding assembly code can be written as follows:

```
simple_divide
        ; Inputs
        ; R0¼ dividend
        ; R1¼ divider
        ; Outputs
        ; R0¼ quotient
        ; R1¼ remainder
        PUSH {R2-R4}             ; Save registers to stack
        MOV R2, R0               ; Save dividend to R2 as R0 will be changed
        MOVS R3, #0x1            ; loop control
        LSLS R3, R3, #31         ; N = 0x80000000
        MOVS R0, #0              ; initial Quotient
        MOVS R4, #0              ; initial Tmp
simple_divide_loop
        LSLS R2, R2, #1           ; Shift dividend left by 1 bit, MSB go into carry
        ADCS R4, R4, R4          ; Shift Tmp left by 1 bit, carry move into LSB
        CMP R4, R1
        BCC simple_divide_lessthan
        ADDS R0, R0, R3          ; Increment quotient
        SUBS R4, R4, R1
simple_divide_lessthan
        LSRS R3, R3, #1          ; N = N >> 1
        BNE simple_divide_loop
        MOV R1, R4               ; Put remainder in R1, Quotient is already in R0
        POP {R2-R4}              ; Restore used register
        BX LR ; Return
```

This simple example does not handle signed data and there is no special handling for divide-by-zero cases. If you need to handle signed data division, you can create a wrapper to convert the dividend and divisor into unsigned data first, and then run the unsigned divide and convert the result back to the signed value afterward.

3.3.5.4 Unsigned Integer Square Root

Another mathematical calculation that is occasionally needed in embedded systems is the square root. Because the square root can only deal with positive numbers (unless complex numbers are used), the following example only handles unsigned integers (Figure 6.8). For the following implementation, the result is rounded to the next lower integer. The corresponding assembly code can be written as follows:

```
simple_sqrt
        ; Input : R0
        ; Output : R0 (square root result)
        PUSH {R1-R3}                    ; Save registers to stack
        MOVS R1, #0x1                   ; Set loop control register
        LSLS R1, R1, #15                ; R1 = 0x00008000
        MOVS R2, #0 ; Initialize result
simple_sqrt_loop
        ADDS R2, R2, R1                 ; M = (M þ N)
        MOVS R3, R2                     ; Copy (M þ N) to R3
        MULS R3, R3, R3                 ; R3 = (M þ N)^2
        CMP R3, R0
        BLS simple_sqrt_lessequal
        SUBS R2, R2, R1                 ; M = (M _ N)
simple_sqrt_lessequal
        LSRS R1, R1, #1                 ; N = N >> 1
        BNE simple_sqrt_loop
        MOV R0, R2                      ; Copy to R0 and return
        POP {R1-R3}                     ;
        BX LR                           ; Return
```

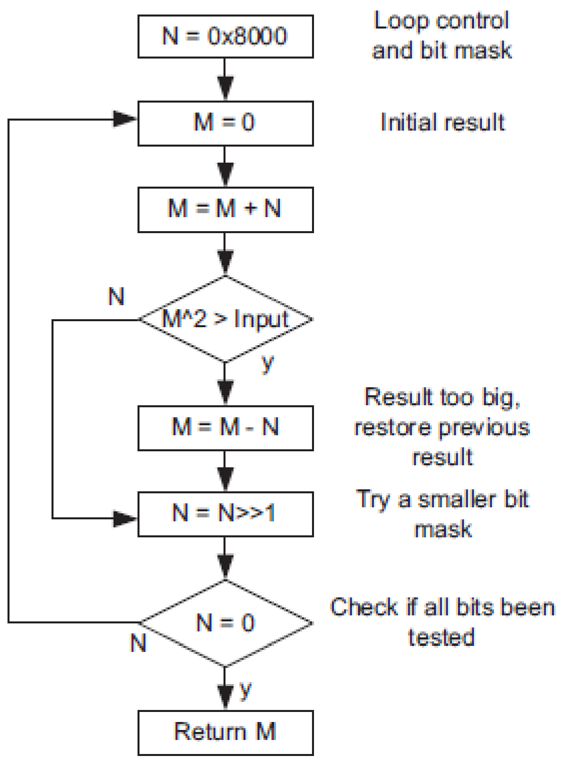


Figure 6.8:
Simple unsigned integer square root function.

3.3.5.5 Bit and Bit Field Computations

Bit data processing is common in microcontroller applications. From the previous divide example code, we have already seen some basic bit computation on the Cortex-M0 processor. Here we will cover a few more examples of bit and bit field processing. To extract a bit from a value stored in a register, we first need to determine how the result will be used. If the result is to be used for controlling a conditional branch, the best solution is to use shift or rotate instructions to copy the required bit in the Carry flag in the APSR, and then carry out the conditional branch using a BCC or BCS instruction. For example,

```
LSRS R0, R0, #<nþ1>          ; Shift bit “n” into carry flag in APSR
BCS <label>                  ; branch if carry is set
```

If the result is going to be used for other processing, then we could extract the bit by a logic shift operation. For example, if we need to extract bit 4 in the register R0, this can be carried out as follows:

```
LSLS R0, R0, #27           ; Remove un-needed top bits
LSRS R0, R0, #31           ; Move required bit into bit 0
```

This extraction method can be generalized to support the extraction of bit fields. For example, if we need to extract a bit field in a data value that is "W" bits wide, starting with bit position "P" (LSB of the bit field), we can extract the bit field using the following instruction:

```
LSLS R0, R0, #(32-W-P)     ; Remove un-needed top bits
LSRS R0, R0, #(32-W)       ; Align required bits to bit 0
```

For example, if we need to extract an 8-bit-width bit field from bit 4 to bit 11 (Figure 6.9), we can use this instruction sequence:

```
LSLS R0, R0, #(32-8-4)     ; Remove un-needed top bits
LSRS R0, R0, #(32-8)       ; Align required bits to bit 0
```

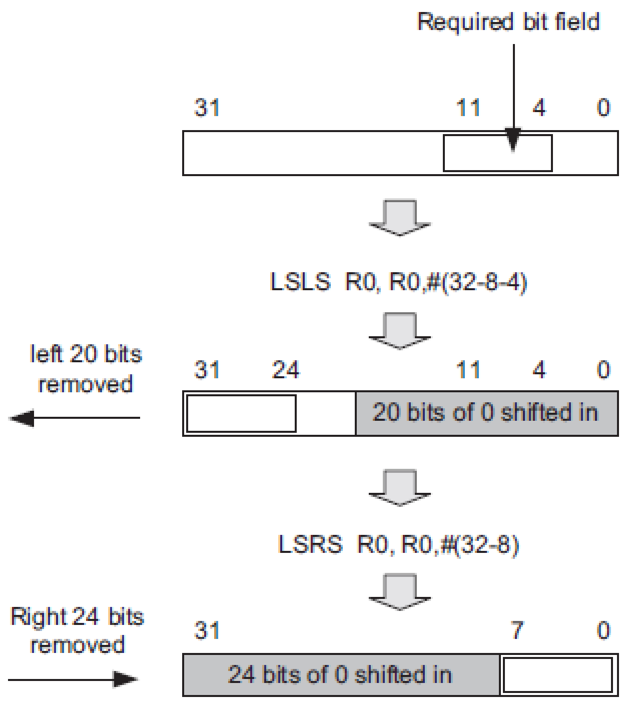


Figure 6.9:
Bit field extract operation.

In a similar way, we can clear the bit field in a register by a few shift and rotate instructions (Figure 6.10):

```
RORS R0, R0, #4            ; Shift unneeded bit to bit 0
LSRS R0, R0, #8            ; Align required bits to bit 0
RORS R0, R0, #(32-8-4)     ; store value to original position
```

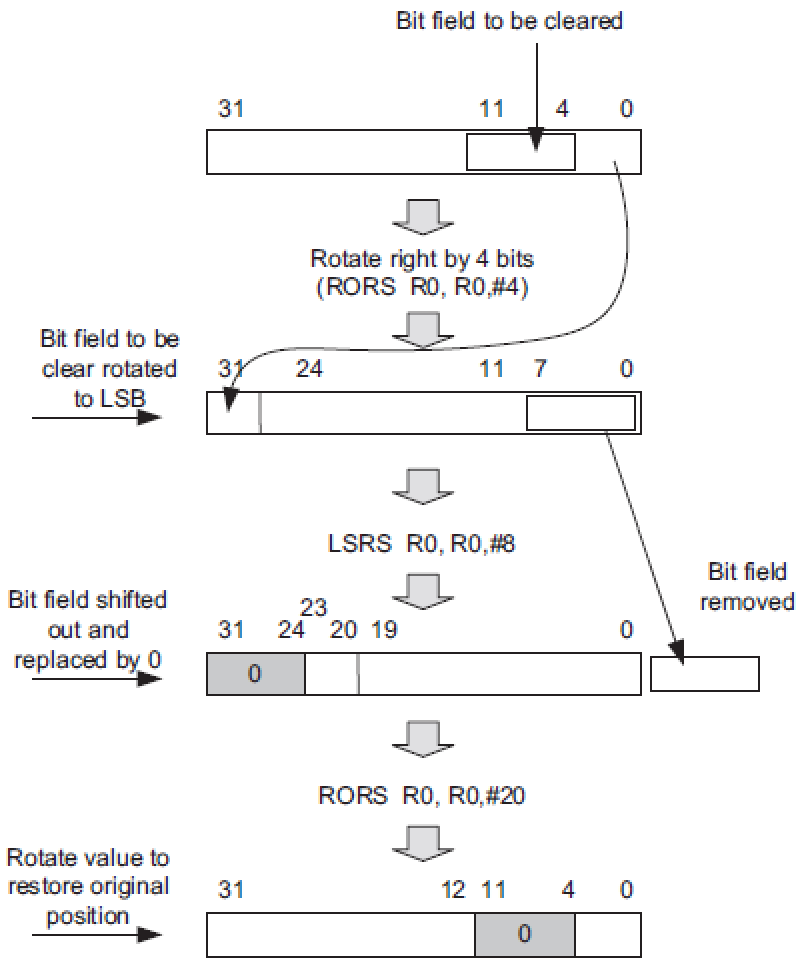


Figure 6.10:
Bit field clear operation.

For masking other bit patterns, we can use the Bit Clear (BICS) instruction. For example,

```
LDR R1, =Bit_Mask          ; Bit to clear
BICS R0, R0, R1            ; Clear bits that are not required
```

The “Bit_Mask” is a value that reflects the bit pattern you want to clear. The BICS instruction does not have any limitation of the bit pattern to be cleared, but it might require a slightly larger program size, as the program might need to store the value of the “Bit_Mask” pattern as a word-size constant.