Unit-2

Architecture

Cortex-M0 Programming model

This chapter covers

- Programming model
  - Operation Modes and States
- Registers and Special Registers
- Behaviors of the Application Program
- Status Register (APSR)
- Memory System Overview.
- **Self-Study:**
  - **Stack Memory operations**

Overview

The ARMv6-M architecture that the Cortex-M0 processor implemented covers a number of different areas. The complete details of the ARMv6-M architecture are documented in the ARMv6-M Architecture Reference Manual [reference 3]. This document is available from the ARM web site via a registration process. However, you do not have to know the complete details of the architecture to start using a Cortex-M0 microcontroller. To use a Cortex-M0 device with C language, you only need to know the memory map, the peripheral programming information, the exception handling mechanism, and part of the programmer's model. In this chapter, we will cover the programmer's model and a basic overview of the memory map and exceptions. Most users of the Cortex-M0 processor will work in C language; as a result, the underlying programmer's model will not be visible in the program code. However, it is still useful to know about the details, as this information is often needed during debugging and it will also help readers to understand the rest of this book.

2.1 Programming model

Operation Modes and States

The Cortex-M0 processor has two operation modes and two states (Figure 3.1).

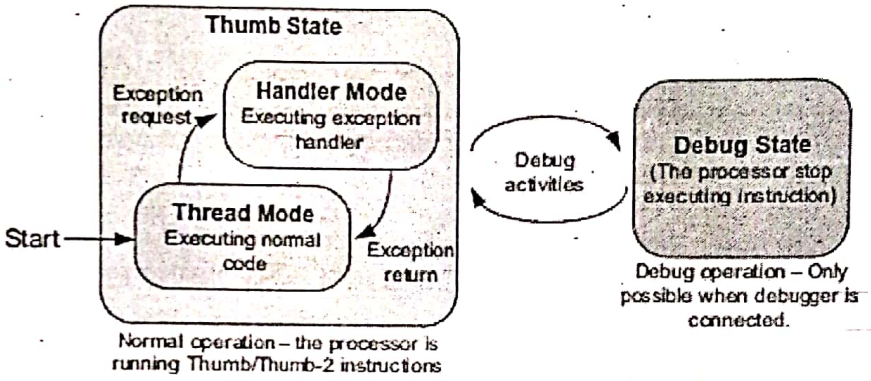


Figure 3.1:
Processor modes and states in the Cortex-M0 processor.

When the processor is running a program, it is in the Thumb state. In this state, it can be either in the Thread mode or the Handler mode. In the ARMv6-M architecture, the programmer's model of Thread mode and Handler mode are almost completely the same. The only difference is that Thread mode can use a shadowed stack pointer (Figure 3.7, presented later in the chapter) by configuring a special register called CONTROL. Details of stack pointer selection are covered later in this chapter.

The Debug state is used for debugging operation only. Halting the processor stops the instruction execution and enter debug state. This state allows the debugger to access or change the processor register values. The debugger can access system memory locations in either the Thumb state or the Debug state.

When the processor is powered up, it will be running in the Thumb state and Thread mode by default.

2.2 Registers and Special Registers

To perform data processing and controls, a number of registers are required inside the processor core. If data from memory are to be processed, they have to be loaded from the memory to a register in the register bank, processed inside the processor, and then written back to the memory if needed. This is commonly called a "load-store architecture." By having a sufficient number of registers in the register bank, this mechanism is easy to use and is C friendly. It is easy for C compilers to compile a C program into machine code with good performance. By using internal registers for short-term data storage, the amount of memory accesses can be reduced.

The Cortex-M0 processor provides a register bank of 13 general-purpose 32-bit registers and a number of special registers (Figure 3.2).

The register bank contains sixteen 32-bit registers. Most of them are general-purpose registers, but some have special uses. The detailed descriptions for these registers are as follows.

R0eR12

Registers R0 to R12 are for general uses. Because of the limited space in the 16-bit Thumb instructions, many of the Thumb instructions can only access R0 to R7, which are also called the low registers, whereas some instructions, like MOV (move), can be used on all registers.

When using these registers with ARM development tools such as the ARM assembler, you can use either uppercase (e.g., R0) or lowercase (e.g., r0) to specify the register to be used. The initial values of R0 to R12 at reset are undefined.

R13, Stack Pointer (SP)

R13 is the stack pointer. It is used for accessing the stack memory via PUSH and POP operations. There are physically two different stack pointers in Cortex-M0. The main stack

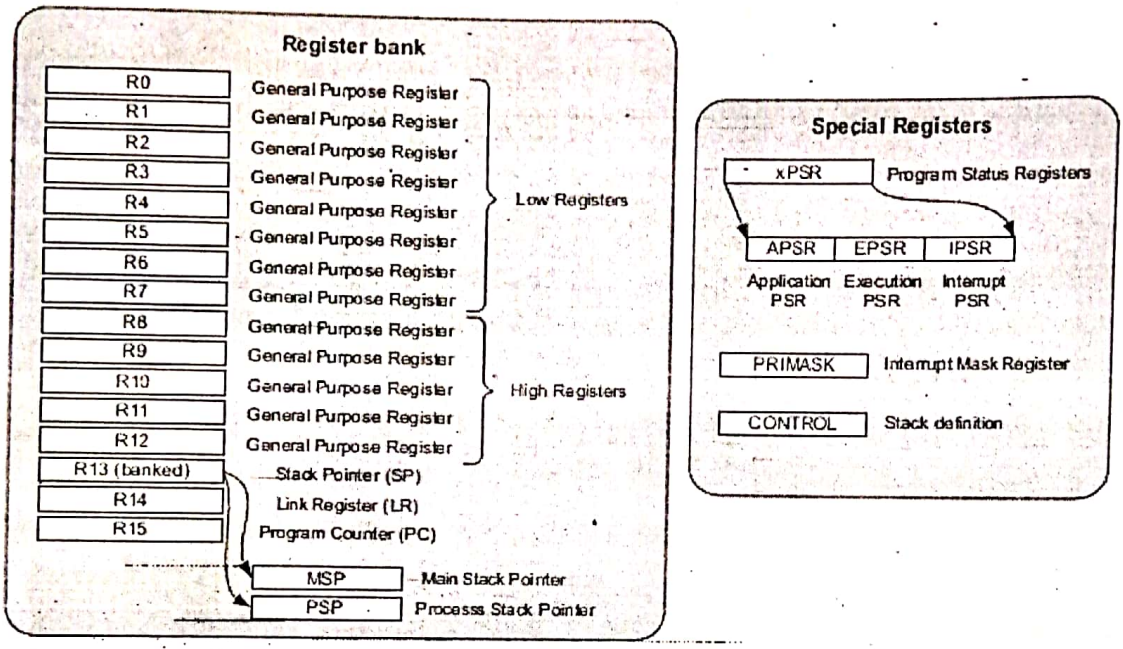


Figure 3.2:
Registers in the Cortex-M0 processor.

pointer (MSP, or SP_main in ARM documentation) is the default stack pointer after reset, and it is used when running exception handlers. The process stack pointer (PSP, or SP_process in ARM documentation) can only be used in Thread mode (when not handling exceptions). The stack pointer selection is determined by the CONTROL register, one of the special registers that will be introduced later.

When using ARM development tools, you can access the stack pointer using either "R13" or "SP." Both uppercase and lowercase (e.g., "r13" or "sp") can be used. Only one of the stack pointers is visible at a given time. However, you can access to the MSP or PSP directly when using the special register access instructions MRS and MSR. In such cases, the register names "MSP" or "PSP" should be used.

The lowest two bits of the stack pointers are always zero, and writes to these two bits are ignored. In ARM processors, PUSH and POP are always 32-bit accesses because the registers are 32-bit, and the transfers in stack operations must be aligned to a 32-bit word boundary. The initial value of MSP is loaded from the first 32-bit word of the vector table from the program memory during the startup sequence. The initial value of PSP is undefined.

It is not necessary to use the PSP. In many applications, the system can completely rely on the MSP. The PSP is normally used in designs with an OS, where the stack memory for OS Kernel and the thread level application code must be separated.

R14, Link Register (LR)

R14 is the Link Register. The Link Register is used for storing the return address of a subroutine or function call. At the end of the subroutine or function, the return address stored in LR is loaded into the program counter so that the execution of the calling program can be resumed. In the case where an exception occurs, the LR also provides a special code value, which is used by the exception return mechanism. When using ARM development tools, you can access to the Link Register using either "R14" or "LR." Both upper and lowercase (e.g., "r14" or "lr") can be used. Although the return address in the Cortex-M0 processor is always an even address (bit[0] is zero because the smallest instructions are 16-bit and must be half-word aligned), bit zero of LR is readable and writeable. In the ARMv6-M architecture, some instructions require bit zero of a function address set to 1 to indicate Thumb state.

R15, Program Counter (PC)

R15 is the Program Counter. It is readable and writeable. A read returns the current instruction address plus four (this is caused by the pipeline nature of the design). Writing to R15 will cause a branch to take place (but unlike a function call, the Link Register does not get updated). In the ARM assembler, you can access the Program Counter, using either "R15" or "PC," in either upper or lower case (e.g., "r15" or "pc"). Instruction addresses in the Cortex-M0 processor must be aligned to half-word address, which means the actual bit zero of the PC should be zero all the time. However, when attempting to carry out a branch using the branch instructions (BX or BLX), the LSB of the PC should be set to 1.2 This is to indicate that the branch target is a Thumb program region. Otherwise, it can imply trying to switch the processor

to ARM state (depending on the instruction used), which is not supported and will cause a fault exception.

xPSR, combined Program Status Register

The combined Program Status Register provides information about program execution and the ALU flags. It is consists of the following three Program Status Registers (PSRs) (Figure 3.3):

- Application PSR (APSR)
- Interrupt PSR (IPSR)
- Execution PSR (EPSR)

The APSR contains the ALU flags: N (negative flag), Z (zero flag), C (carry or borrow flag), and V (overflow flag). These bits are at the top 4 bits of the APSR. The common use of these flags is to control conditional branches.

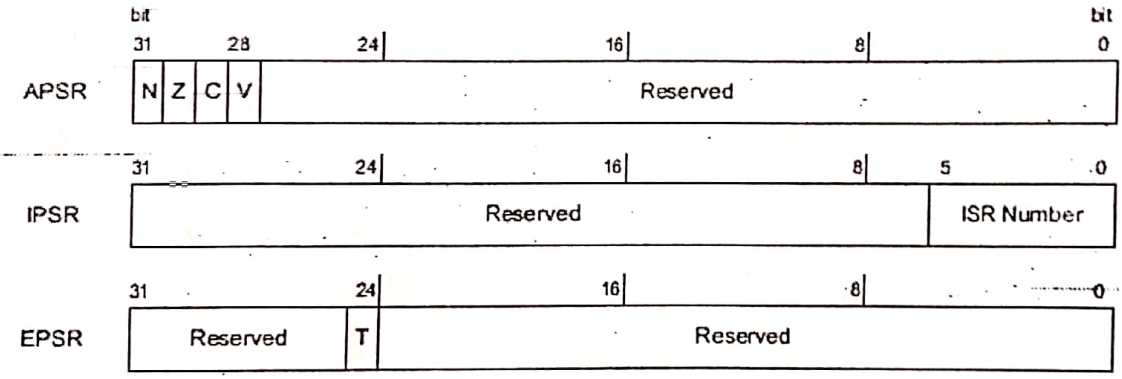


Figure 3.3:
APSR, IPSR, and EPSR.

The IPSR contains the current executing interrupt service routine (ISR) number. Each exception on the Cortex-M0 processor has a unique associated ISR number (exception type). This is useful for identifying the current interrupt type during debugging and allows an exception handler that is shared by several exceptions to know what exception it is serving. The EPSR on the Cortex-M0 processor contains the T-bit, which indicates that the processor is in the Thumb state. On the Cortex-M0 processor, this bit is normally set to 1 because the Cortex-M0 only supports the Thumb state. If this bit is cleared, a hard fault exception will be generated in the

next instruction execution. These three registers can be accessed as one register called xPSR (Figure 3.4). For example, when an interrupt takes place, the xPSR is one of the registers that is stored onto the stack memory automatically and is restored automatically after returning from an exception. During the stack store and restore, the xPSR is treated as one register.

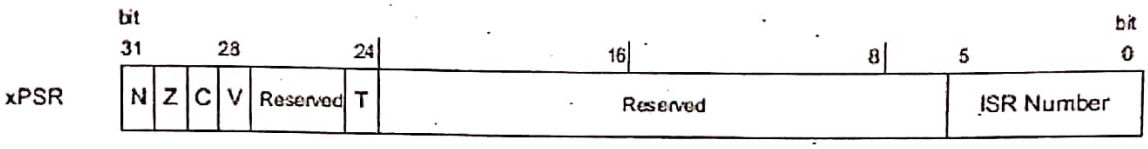


Figure 3.4:
xPSR.

Direct access to the Program Status Registers is only possible through special register access instructions. However, the value of the APSR can affect conditional branches and the carry flag in the APSR can also be used in some data processing instructions.

2.3 Behaviors of the Application Program Status Register (APSR)

Data processing instructions can affect destination registers as well as the APSR, which is commonly known as ALU status flags in other processor architectures. The APSR is essential for controlling conditional branches. In addition, one of the APSR flags, the C (Carry) bit, can also be used in add and subtract operations.

There are four APSR flags in the Cortex-M0 processor, and they are identified in Table 3.1.

Table 3.1: ALU Flags on the Cortex-M0 Processor

Flag	Descriptions
N (bit 31)	Set to bit [31] of the result of the executed instruction. When it is "1", the result has a negative value (when interpreted as a signed integer). When it is "0", the result has a positive value or equal zero.
Z (bit 30)	Set to "1" if the result of the executed instruction is zero. It can also be set to "1" after a compare instruction is executed if the two values are the same.
C (bit 29)	Carry flag of the result. For unsigned addition, this bit is set to "1" if an unsigned overflow occurred. For unsigned subtract operations, this bit is the inverse of the borrow output status.
V (bit 28)	Overflow of the result. For signed addition or subtraction, this bit is set to "1" if a signed overflow occurred.

A few examples of the ALU flag results are shown in Table 3.2.

Table 3.2: ALU Flags Example

Operation	Results, Flags
0x70000000 + 0x70000000	Result = 0xE0000000, N = 1, Z = 0, C = 0, V = 1
0x90000000 + 0x90000000	Result = 0x30000000, N = 0, Z = 0, C = 1, V = 1
0x80000000 + 0x80000000	Result = 0x00000000, N = 0, Z = 1, C = 1, V = 1
0x00001234 – 0x00001000	Result = 0x00000234, N = 0, Z = 0, C = 1, V = 0
0x00000004 – 0x00000005	Result = 0xFFFFFFFF, N = 1, Z = 0, C = 0, V = 0
0xFFFFFFFF – 0xFFFFFFFC	Result = 0x00000003, N = 0, Z = 0, C = 1, V = 0
0x80000005 – 0x80000004	Result = 0x00000001, N = 0, Z = 0, C = 1, V = 0
0x70000000 – 0xF0000000	Result = 0x80000000, N = 1, Z = 0, C = 0, V = 1
0xA0000000 – 0xA0000000	Result = 0x00000000, N = 0, Z = 1, C = 1, V = 0

In the Cortex-M0, almost all of the data processing instructions modify the APSR; however, some of these instructions do not update the V flag or the C flag. For example, the MULS (multiply) instruction only changes the N flag and the Z flag. The ALU flags can be used for handling data that is larger than 32 bits. For example, we can perform a 64-bit addition by splitting the operation into two 32-bit additions. The pseudo form of the operation can be written as follows:

```
// Calculating Z ¼ X þ Y, where X, Y and Z are all 64-bit
Z[31:0] ¼ X[31:0] þ Z[31:0]; // Calculate lower word addition, carry flag get updated
Z[63:32] ¼ X[63:32] þ Z[63:32] þ Carry; // Calculate upper word addition
```

An example of carry out such 64-bit add operation in assembly code can be found in Chapter 6.

2.4 Memory System Overview

The Cortex-M0 processor has 4 GB of memory address space (Figure 3.8). The memory space is architecturally defined as a number of regions, with each region having a recommended usage to help software porting between different devices.

The Cortex-M0 processor contains a number of built-in components like the NVIC and a number of debug components. These are in fixed memory locations within the system region of the memory map. As a result, all the devices based on the Cortex-M0 have the same programming model for interrupt control and debug. This makes it convenient for software porting and helps debug tool vendors to develop debug solutions for the Cortex-M0 based microcontroller or system-on-chip (SoC) products.

In most cases, the memories connected to the Cortex-M0 are 32-bits, but it is also possible to connect memory of different data widths to the Cortex-M0 processor with suitable memory interface hardware. The Cortex-M0 memory system supports memory transfers of different sizes such as byte (8-bit), half word (16-bit), and word (32-bit). The Cortex-M0 design can be

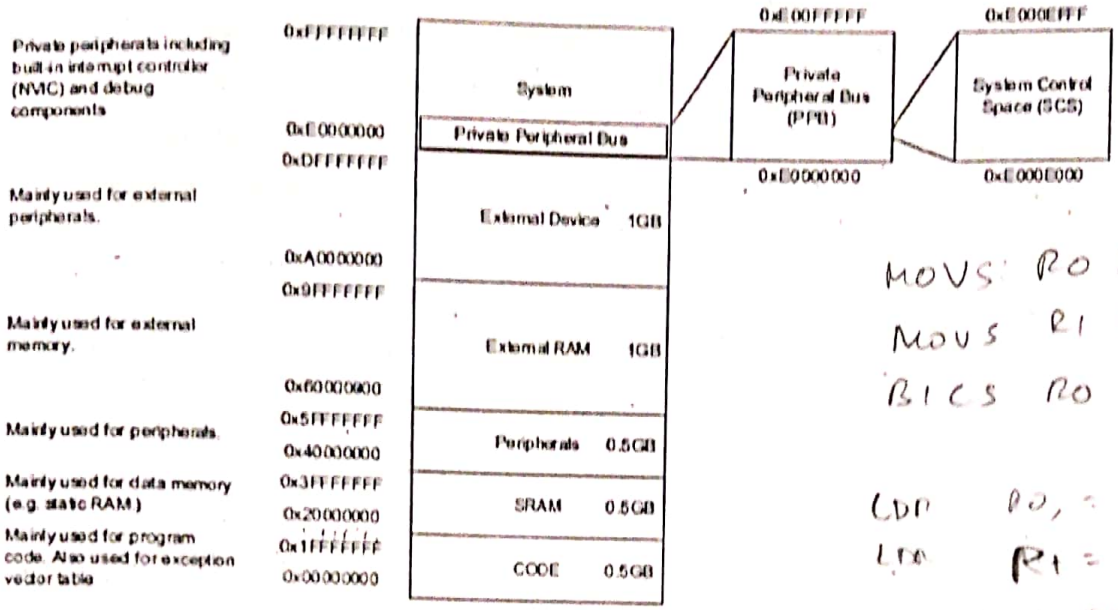


Figure 3.8:
Memory map.

configured to support either little endian or big endian memory systems, but it cannot switch from one to another in an implemented design.

Because the memory system and peripherals connected to the Cortex-M0 are developed by microcontroller vendors or system-on-chip (SoC) designers, different memory sizes and memory types can be found in different Cortex-M0 based products.

2.5 Stack Memory Operations

Stack memory is a memory usage mechanism that allows the system memory to be used as temporary data storage that behaves as a first-in, last-out buffer. One of the essential elements of stack memory operation is a register called the stack pointer. The stack pointer is adjusted automatically each time a stack operation is carried out. In the Cortex-M0 processor, the stack pointer is register R13 in the register bank. Physically there are two stack pointers in the Cortex-M0 processor, but only one of them is used at one time, depending on the current value of the CONTROL register and the state of the processor (see Figure 3.7).

In common terms, storing data to the stack is called pushing (using the PUSH instruction) and restoring data from the stack is called popping (using the POP instruction). Depending on processor architecture, some processors perform storing of new data to stack memory using incremental address indexing and some use decrement address indexing. In the Cortex-M0 processor, the stack operation is based on a "full-descending" stack model. This means the

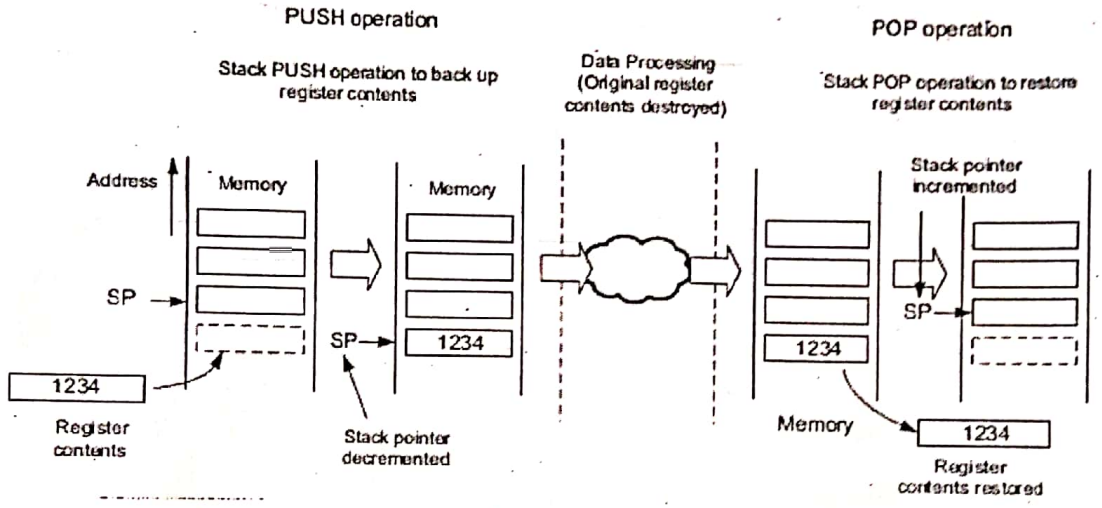


Figure 3.9:
Stack PUSH and POP in the Cortex-M0 processor.

stack pointer always points to the last filled data in the stack memory, and the stack pointer predecrements for each new data store (PUSH) (Figure 3.9).

PUSH and POP are commonly used at the beginning and end of a function or subroutine. At the beginning of a function, the current contents of the registers used by the calling program are stored onto the stack memory using a PUSH operation, and at the end of the function, the data on the stack memory is restored to the registers using a POP operation. Typically, each register PUSH operation should have a corresponding register POP operation, otherwise the stack pointer will not be able to restore registers to their original values. This can result in unpredictable behavior, for example, stack overflow.

The minimum data size to be transferred for each push and pop operations is one word (32-bit), and multiple registers can be pushed or popped in one instruction. The stack memory accesses in the Cortex-M0 processor are designed to be always word aligned (address values must be a multiple of 4, for example, 0x0, 0x4, 0x8, etc.), as this gives the best efficiency for

minimum design complexity. For this reason, bits [1:0] of both stack pointers in the Cortex-M0 processor are hardwired to zeros and read as zeros.

The stack pointer can be accessed as either R13 or SP. Depending on the processor state and the CONTROL register value, the stack pointer accessed can either be the main stack pointer (MSP) or the process stack pointer (PSP). In many simple applications, only one stack pointer is needed and by default the main stack pointer (MSP) is used. The process stack pointer (PSP) is usually only required when an operating system (OS) is used in the embedded application (Table 3.3).

Table 3.3: Stack Pointer Usage Definition

Processor State	CONTROL[1] = 0 (Default Setting)	CONTROL[1] = 1 (OS Has Started)
Thread mode	Use MSP (R13 is MSP)	Use PSP (R13 is PSP)
Handler mode	Use MSP (R13 is MSP)	Use MSP (R13 is MSP)

In a typical embedded application with an OS, the OS kernel uses the MSP and the application processes use the PSP. This allows the stack for the kernel to be separate from stack memory for the application processes. This allows the OS to carry out context switching quickly (switching from execution of one application process to another). Even though the OS kernel only uses the MSP as its stack pointer, it can still access the value in PSP by using special register access instructions (MRS and MSR).

Because the stack grows downward (full-descending), it is common for the initial value of the stack pointer to be set to the upper boundary of SRAM. For example, if the SRAM memory range is from 0x20000000 to 0x20007FFF, we can start the stack pointer at 0x20008000. In this case, the first stack PUSH will take place at address 0x20007FFC, the top word of the SRAM.

The initial value of MSP is stored at the beginning of the program memory. Here we will find the exception vector table, which is introduced in the next section. The initial value of PSP is undefined, and therefore the PSP must be initialized by software before using it.

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).

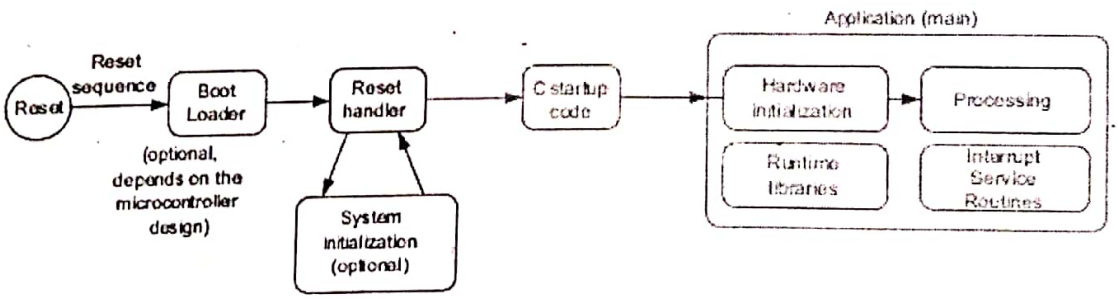


Figure 4.1:
What happens when a microcontroller starts—the Reset handler.

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.

C start up code:

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).

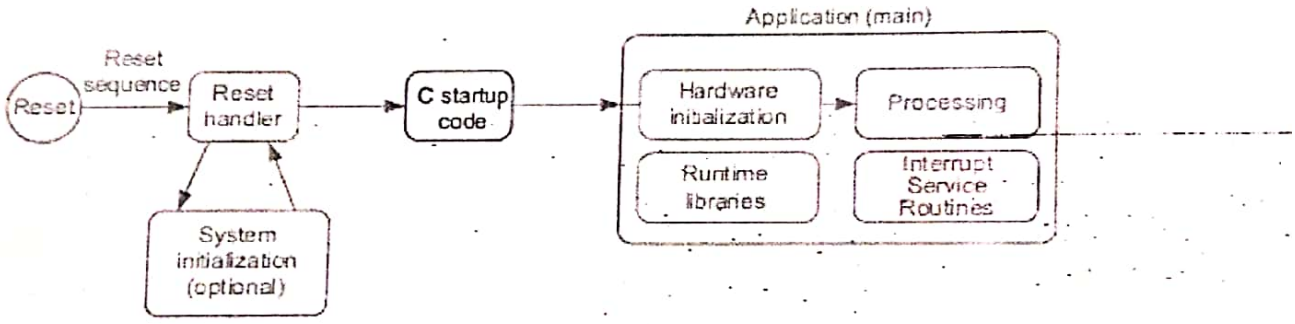


Figure 4.2:
What happens when a microcontroller starts—C startup code.

Application

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.

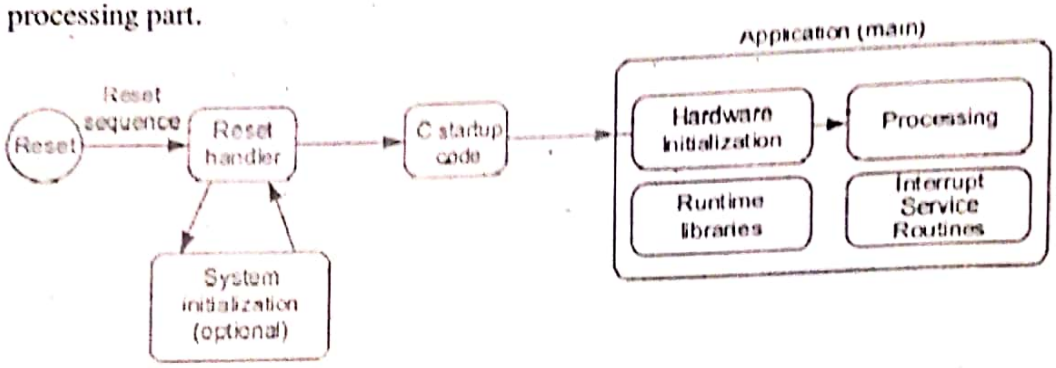


Figure 4.3:
What happens when a microcontroller starts—application.

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).

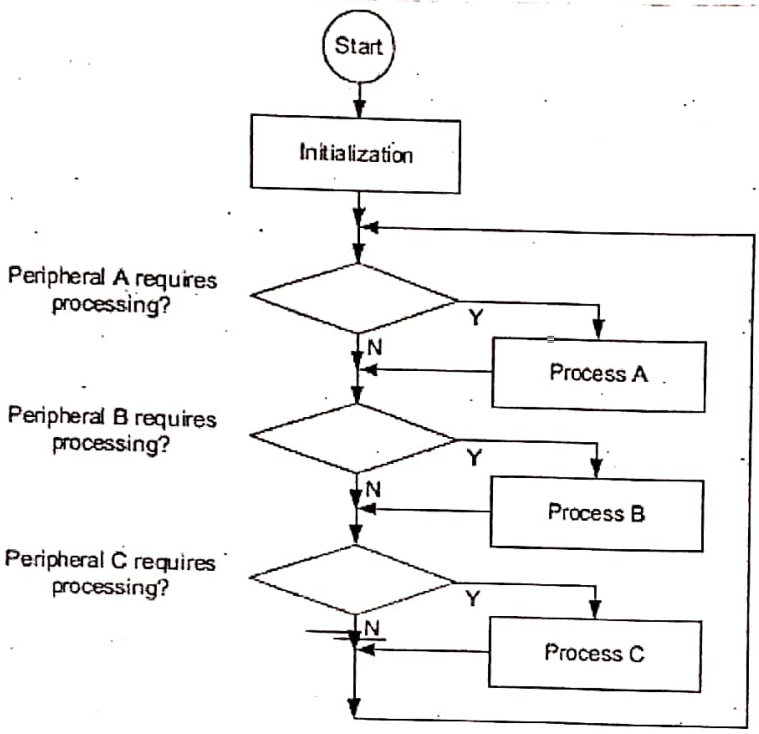


Figure 4.4:
Polling method for simple application processing.

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.

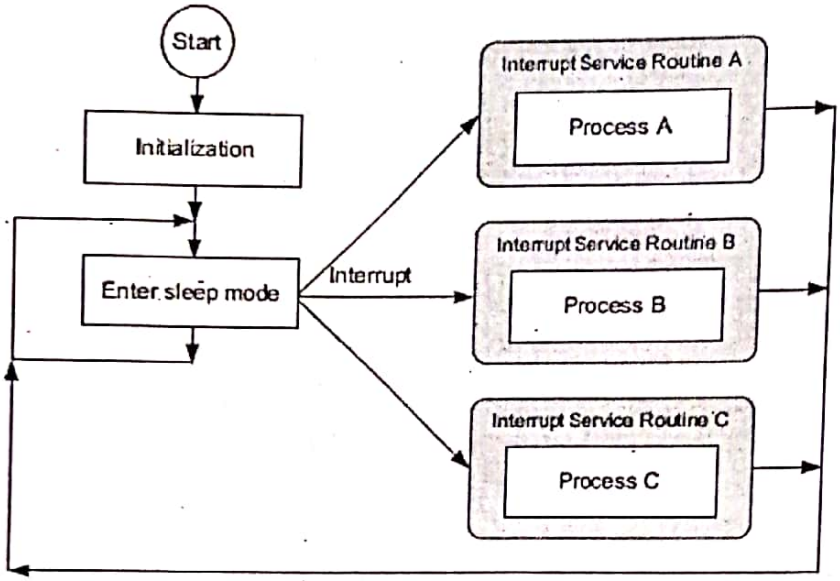
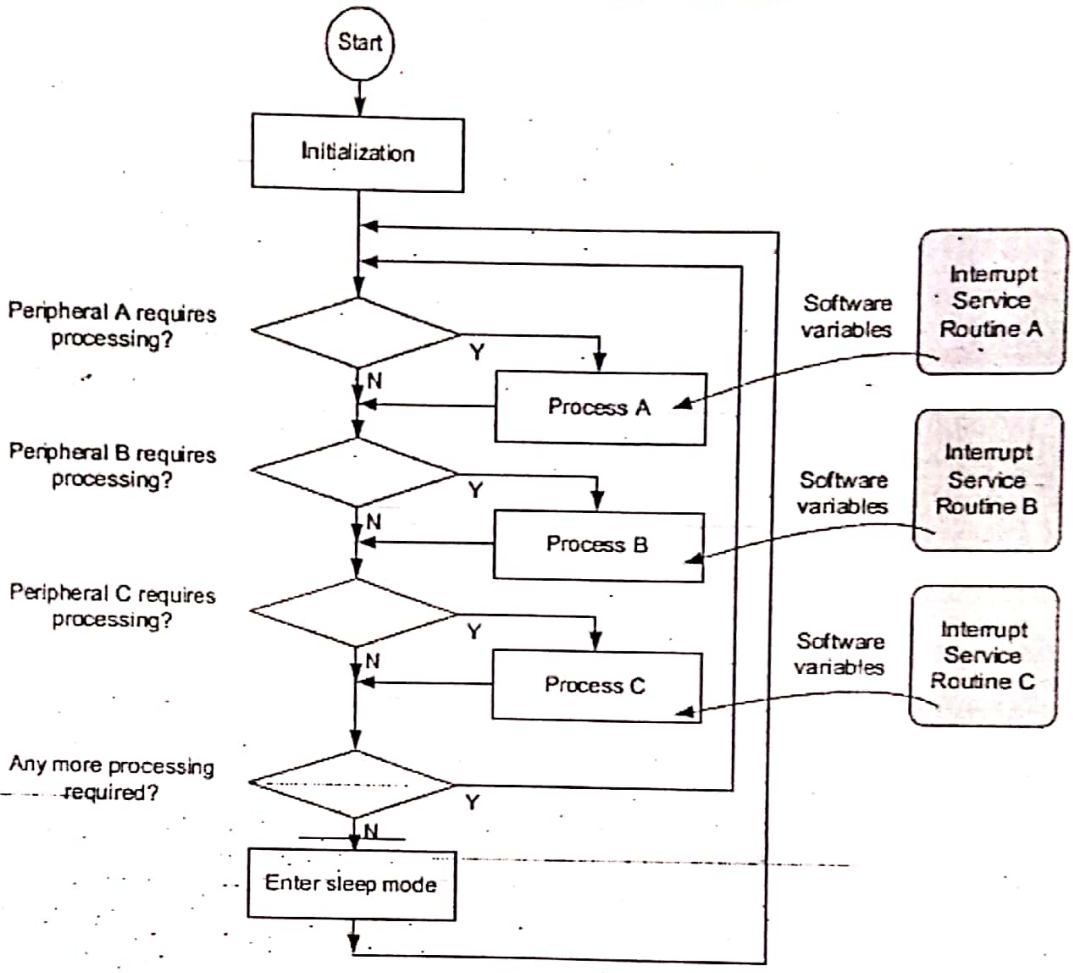


Figure 4.5:
An interrupt-driven application.

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



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.

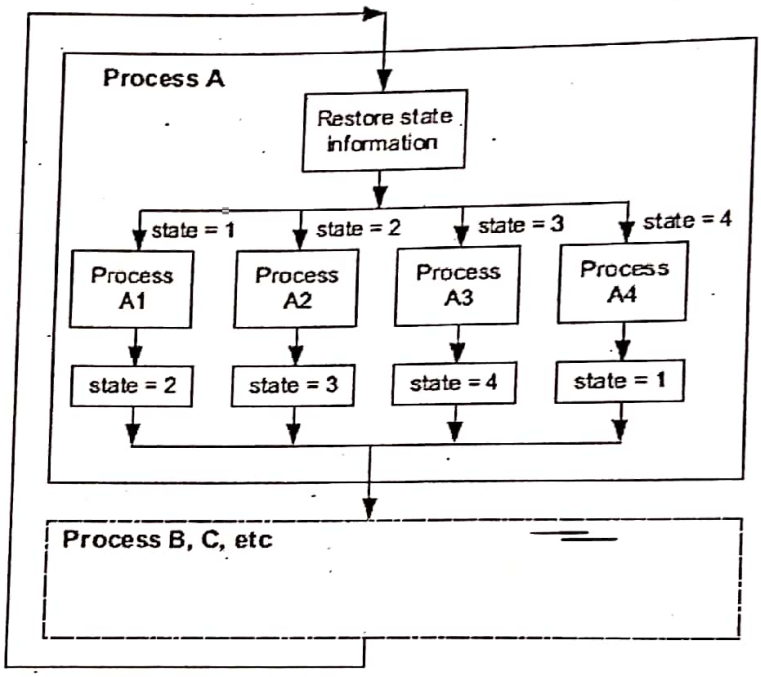


Figure 4.7:
Partitioning a process into multiple parts in the application loop.

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.

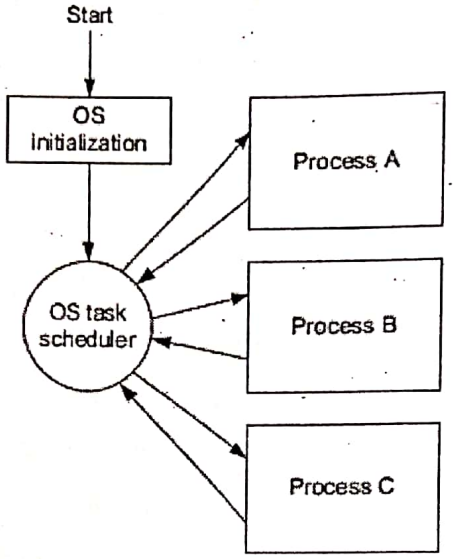


Figure 4.8:
Using an RTOS to handle multiple concurrent application processes.

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).

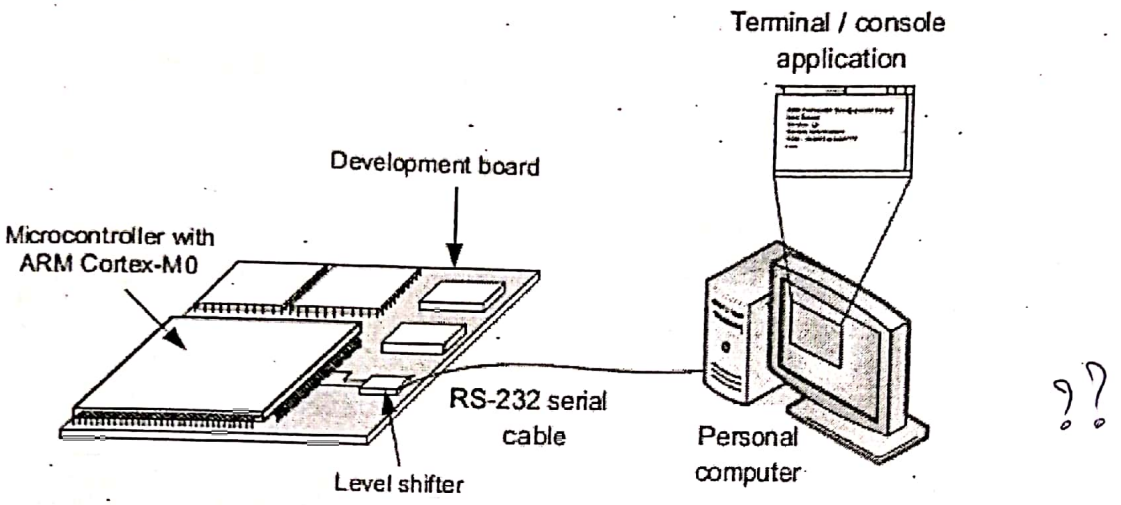


Figure 4.9:
Using UART interface for user input and output.

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.

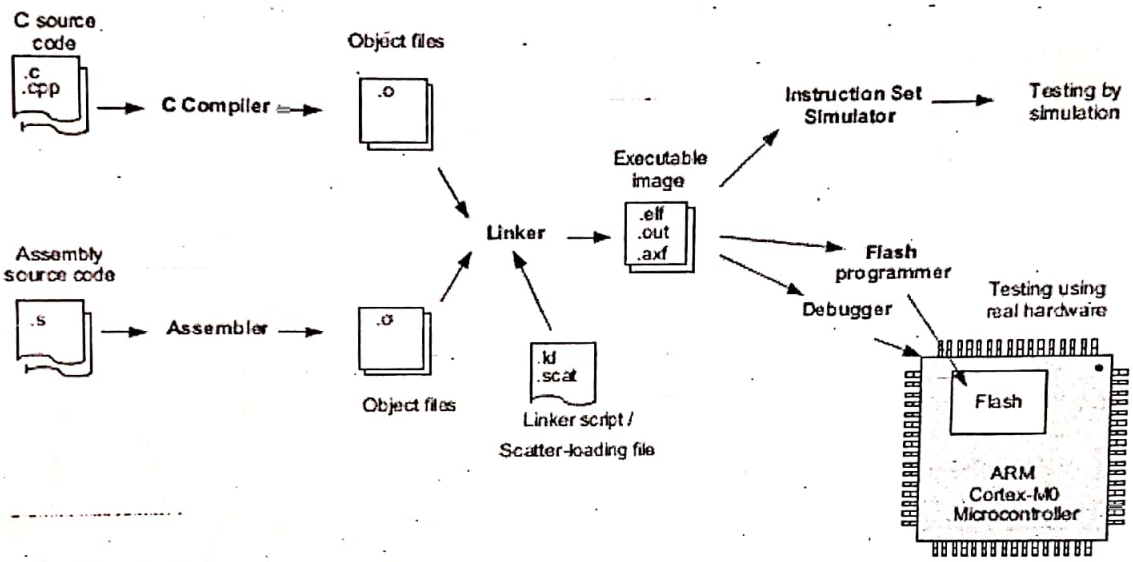


Figure 4.10:
Typical program-generation flow.

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).

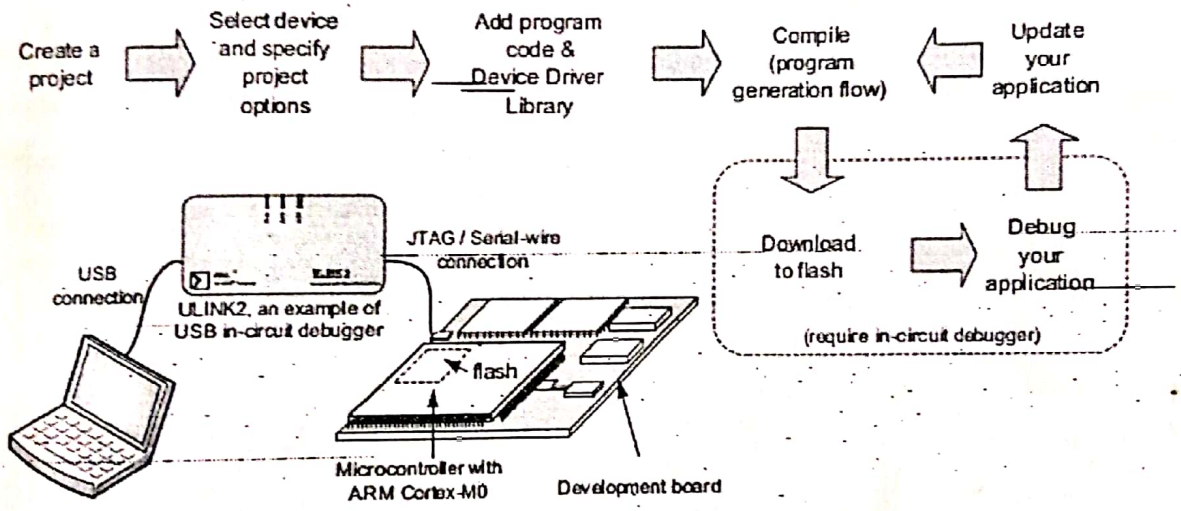


Figure 4.11:
An example of development flow.

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

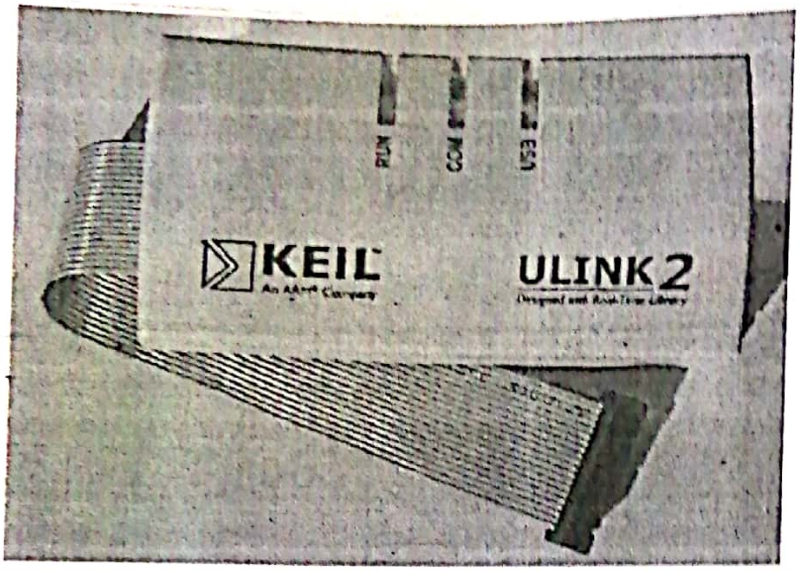


Figure 4.12:
ULINK 2 USB-JTAG adaptor.

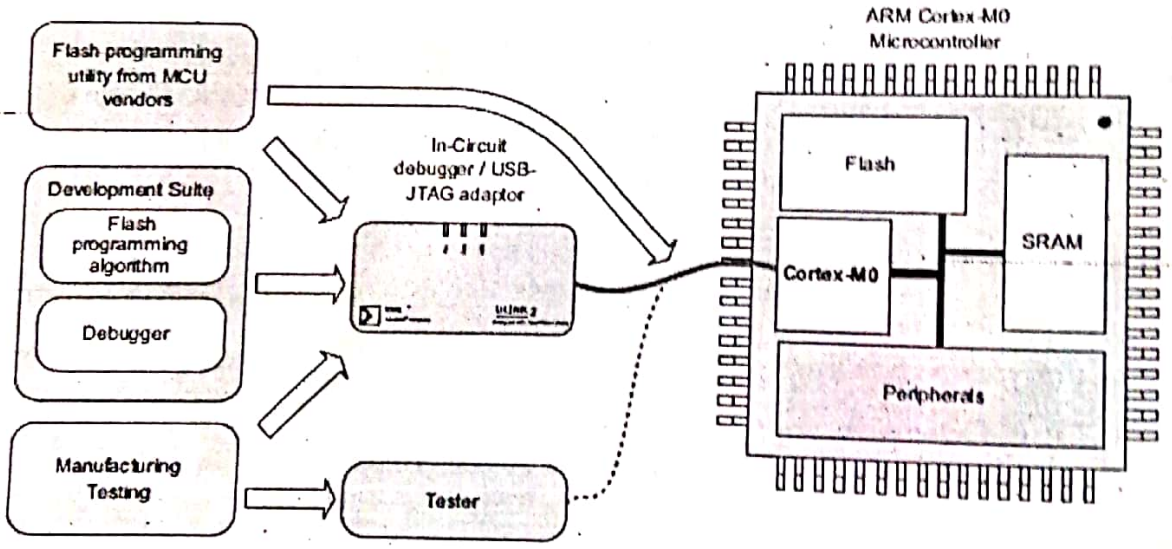


Figure 4.13:
Various usages of the debug interface on the Cortex-M0 processor.

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 (Theory)

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 to C 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:

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).

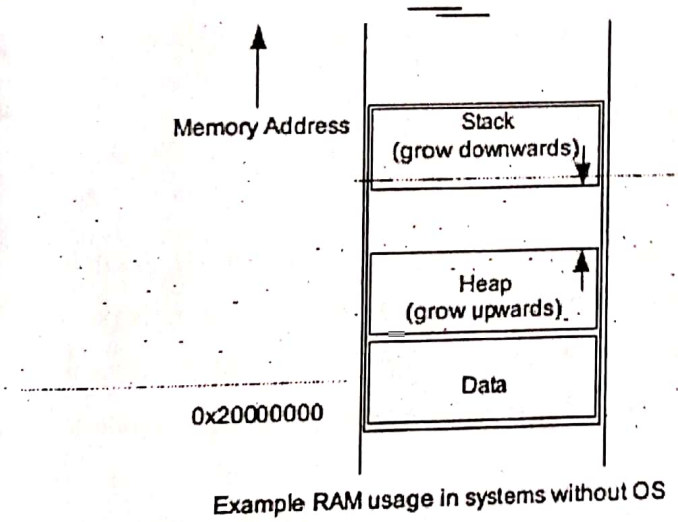


Figure 4.14:
Example of RAM usage in single task systems (without OS).

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.)

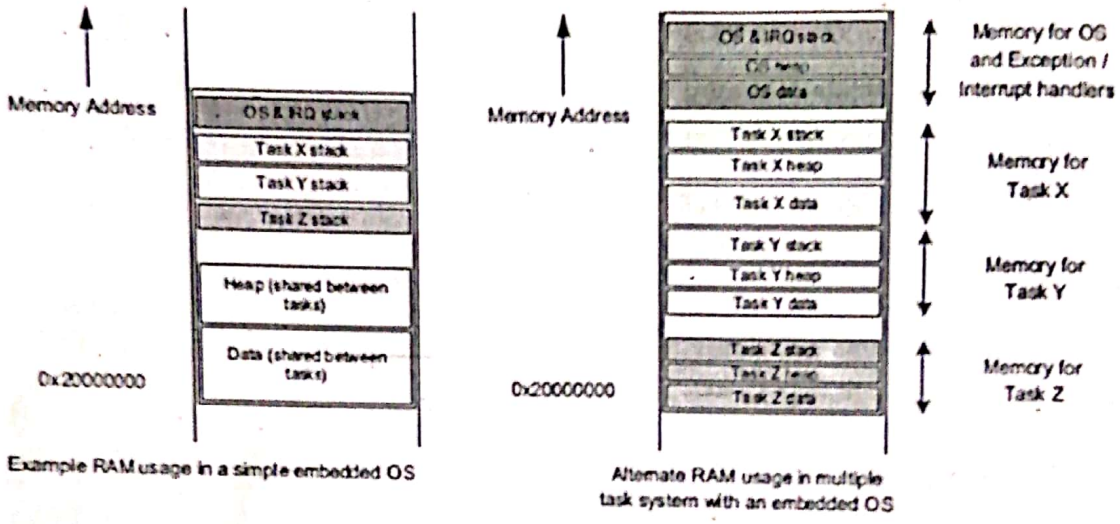


Figure 4.15:
Example of RAM usage in multiple task systems (with an OS).

• 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	$-1.7976931348623157 \times 10^{308}$ to $1.7976931348623157 \times 10^{308}$	
long double	64	$-1.7976931348623157 \times 10^{308}$ to $1.7976931348623157 \times 10^{308}$	
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.

3.1.8 *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.

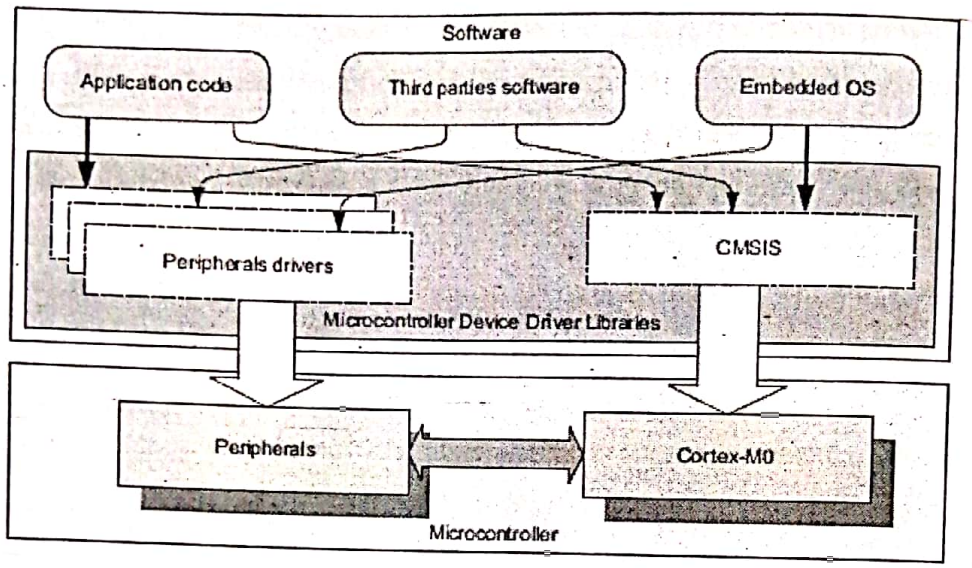


Figure 4.16:
CMSIS provides standardized access functions for processor features.

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 Gpp 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.

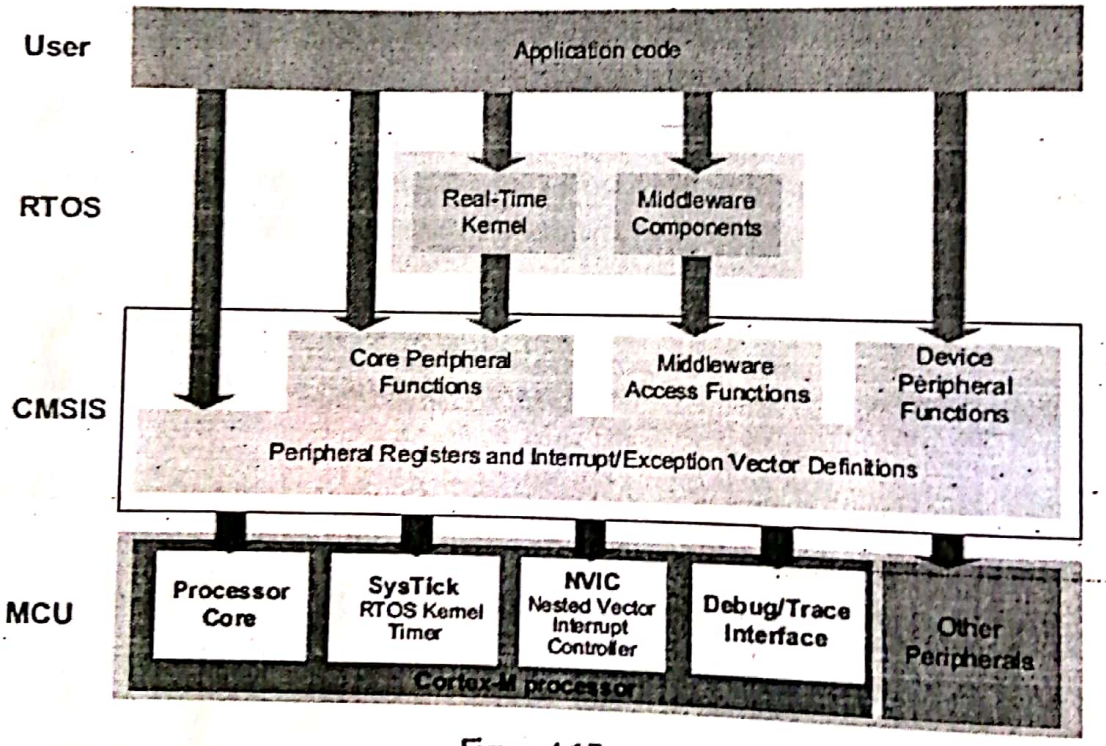


Figure 4.17:
CMSIS structure.

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).

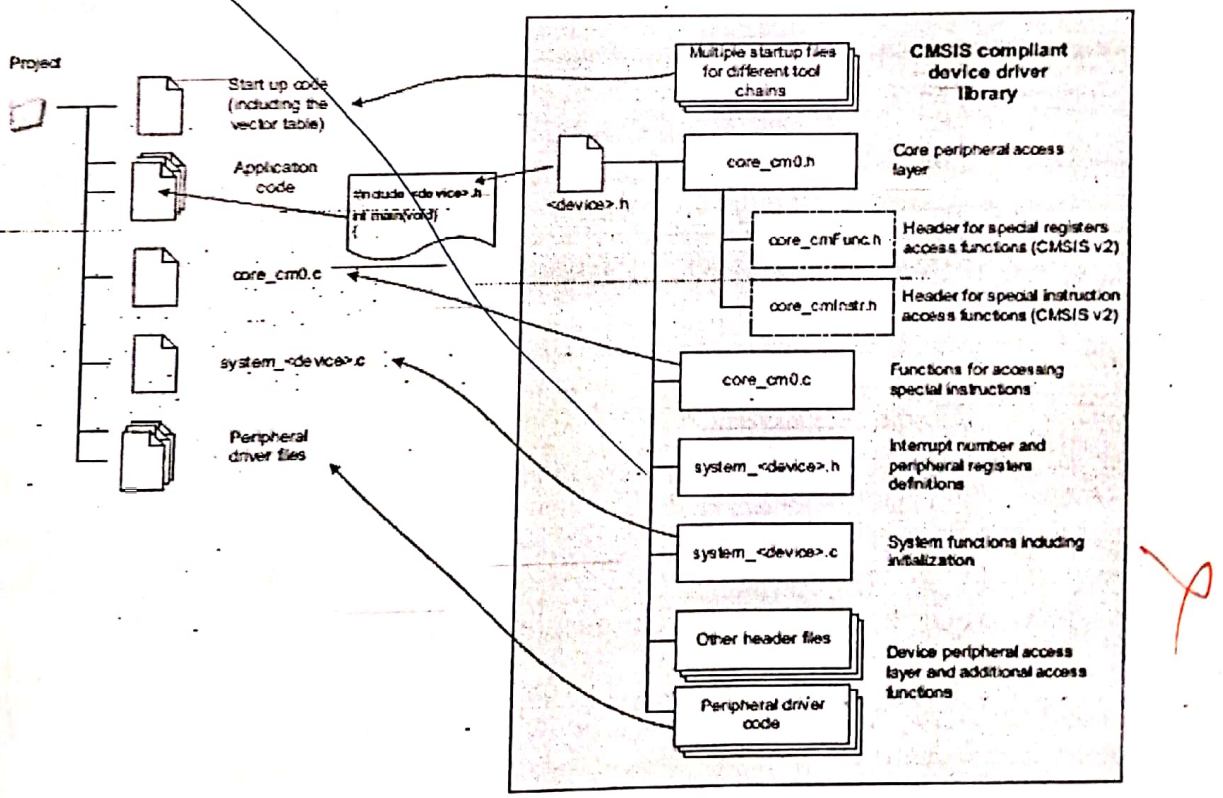


Figure 4.18:
Using CMSIS in a project.

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.

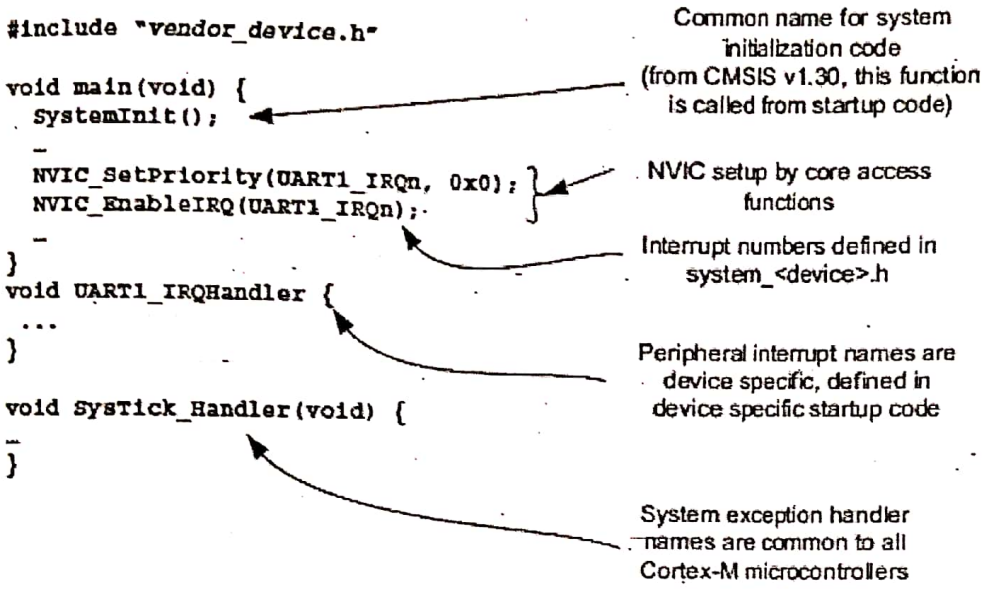


Figure 4.19:
CMSIS example.

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).

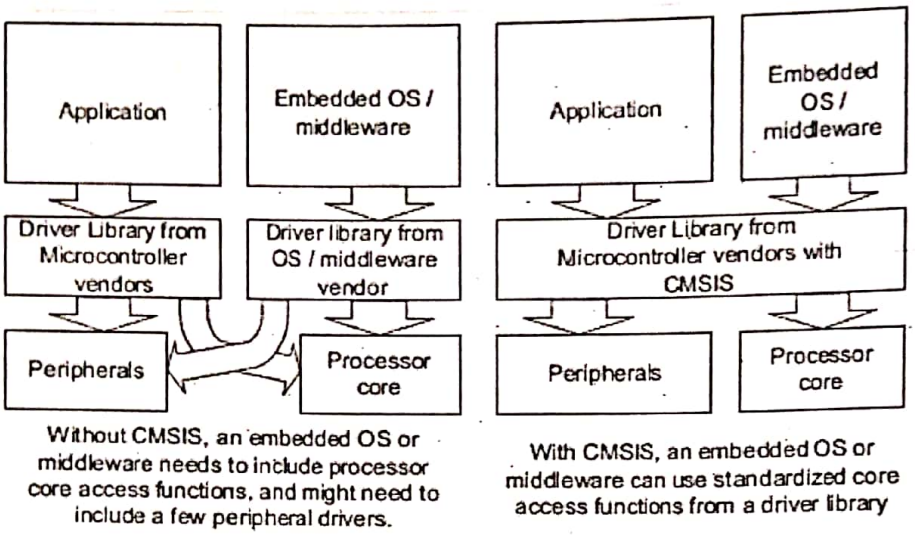


Figure 4.20:
CMSIS avoids overlapping of driver code.

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.