CHAPTER

2 Introduction to Embedded Software Development

CHAPTER OUTLINE

2.1 What are inside typical ARM® microcontrollers?

There are many different things inside a microcontroller. In many microcontrollers, the processor takes less than 10% of the silicon area, and the rest of the silicon die is occupied by other components such as:

- Program memory (e.g., flash memory)
- SRAM
- Peripherals

- Internal bus infrastructure
- Clock generator (including Phase Locked Loop), reset generator, and distribution network for these signals
- Voltage regulator and power control circuits
- Other analog components (e.g., ADC, DAC, voltage reference circuits)
- I/O pads
- Support circuits for manufacturing tests, etc.

While some of these components are directly visible to programmers, some others could be invisible to software developers (e.g., support circuit for manufacturing tests). Don't worry; to use a Cortex®-M microcontroller, we only need to have basic understanding of the processors (e.g., how to use the interrupt features), as well as the detailed programmer's model of the peripherals. Since the peripherals from different microcontroller vendors are different, you need to download and read the user manuals (or similar documents) from microcontroller vendors. This book is focused on the processors, although a number of examples on using peripherals are also covered.

Peripherals and control registers for system management are accessible from the memory map. To make it easier for software developers, most microcontroller vendors provide C header files and driver libraries for their microcontrollers. In most cases, these files are developed with the Cortex Microcontroller Software Interface Standard (CMSIS), which means it used a set of standardized headers for accessing processor features. We will cover more on this later in this chapter.

In most cases, the processor does all the work of controlling the peripherals and handles the system management. This book will cover a few examples of using a number of popular Cortex-M3/M4-based microcontrollers. In some microcontrollers there are also some smart peripherals that can do small amounts of processing without processor intervention. This depends on the vendor-specific peripherals on the microcontrollers and is beyond the scope of this book, but you can find the details in user manuals on the microcontroller vendor's website.

2.2 What you need to start

2.2.1 Development suites

With more than 10 different vendors selling C compiler suites for Cortex®-M microcontrollers, deciding which one to use can be a difficult choice. The development suites range from open-source free tools, to budget low-cost tools, to high-end commercial packages. The current available choices included various products from the following vendors:

- Keil™ Microcontroller Development Kit (MDK-ARM)
- ARM® DS-5™ (Development Studio 5)
- IAR Systems (Embedded Workbench for ARM Cortex-M)
- Red Suite from Code Red Technologies (acquired by NXP in 2013)
- Mentor Graphics Sourcery CodeBench (formerly CodeSourcery Sourcery g++)
- mbed.org

- Altium Tasking VX-toolset for ARM Cortex-M
- Rowley Associates (CrossWorks)
- Coocox
- Texas Instruments Code Composer Studio (CCS)
- Raisonance RIDE
- Atollic TrueStudio
- GNU Compiler Collection (GCC)
- ImageCraft ICCV8
- Cosmic Software C Cross Compiler for Cortex-M
- mikroElektronika mikroC
- Arduino

Some development boards also include a basic or evaluation edition of the development suites. In addition, there are development suites for other languages. For example:

- Oracle Java ME Embedded
- IS2T MicroEJ Java virtual machine
- mikroElektronika mikroBasic, mikroPascal

The illustrations in this book are mostly based on the Keil Microcontroller Development Kit (MDK-ARM) because of its popularity, but most of the example code can be used with the other development suites.

2.2.2 Development boards

There are already a large number of development kits for the Cortex®-M3/M4 microcontrollers from various microcontroller vendors and their distributors. Many of them are offered at an excellent price. For example, you can get a Cortex-M3 evaluation board for less than $12.

You can also get development kits from software tool vendors; for example, companies like Keil™ (an example is show in Figure 2.1), IAR Systems, and Code Red Technologies all have a number of development boards available.

A number of low-cost development boards are designed to work with particular development suites. For example, the “mbed.org” development boards, a low-cost solution for rapid software prototyping, are designed to work with the mbed development platform.

To start learning about ARM® Cortex-M microcontrollers, it is not always necessary to get an actual development board. Several development suites include an instruction set simulator, and the Keil MDK-ARM even supports device-level simulation for some of the popular Cortex-M microcontrollers. So you can learn Cortex-M programming just by using simulation.

2.2.3 Debug adaptor

In order to download your program code to the microcontroller, and to carry out debug operations like halting and single stepping, you might need a debug adaptor

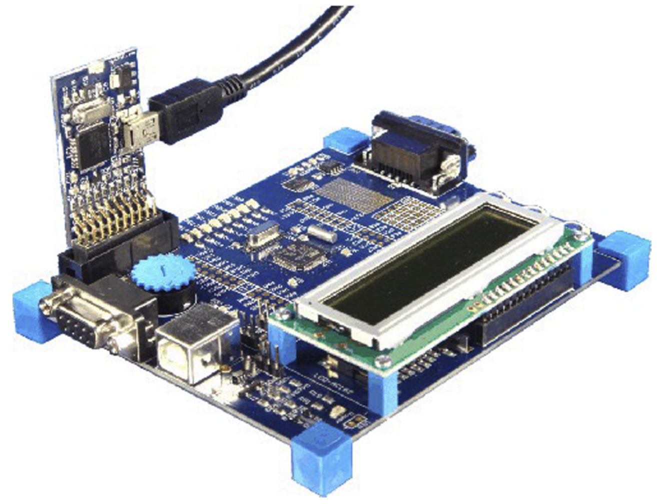

FIGURE 2.1

A Cortex-M3 development board from Keil (MCBSTM32)

to convert a USB connection from your PC to a debug communication protocol used by the microcontrollers. Most C compiler vendors have their own debug adaptor products. For example, Keil™ has the ULINK product family (Figure 2.2), and IAR provides the I-Jet product. Most development suites also support third-party debug adaptors. Note that different vendors might have different terminologies for these debug adaptors, for example, debug probe, USB-JTAG adaptor, JTAG/SW Emulator, JTAG In-Circuit Emulator (ICE), etc.

Some of the development kits already have a USB debug adaptor built-in on the board. This includes some of the low-cost evaluation boards from Texas Instruments,

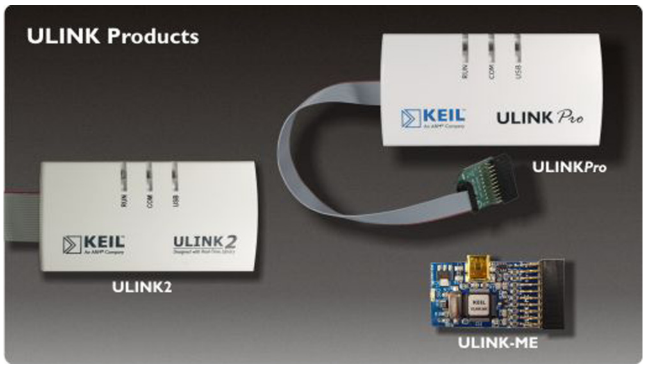

FIGURE 2.2

The Keil ULINK debug adaptor family

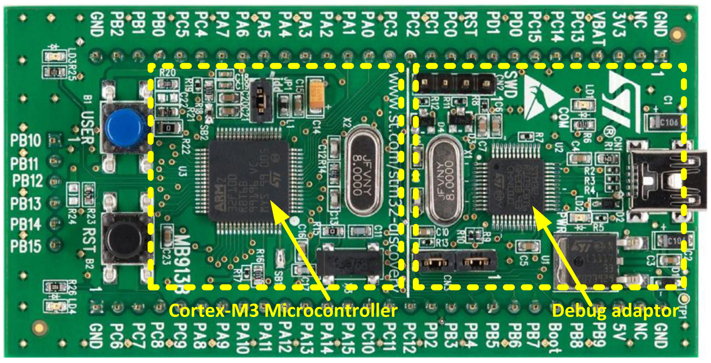

FIGURE 2.3

An example of development board with USB debug adaptor — STM32 Value Line Discovery

ST Microelectronics (e.g., STM32 Value Line Discovery; Figure 2.3), NXP, Energy-Micro, etc. Many of these onboard USB adaptors are also supported by mainstream commercial development suites. So you can start developing software for the Cortex®-M microcontrollers with a tiny budget.

In a number of evaluation/development boards, the built-in USB debug adaptor can also be used to connect to other development boards. You can also find "open-source" versions of such debug adaptors. The CMSIS-DAP from ARM and CoLink from Coocox are two examples.

While these low-cost debug adaptors work for most debug operations, there might be some features that are not well supported. There are a number of commercial USB debug adaptor products that offer a large number of useful features.

2.2.4 Software device driver

The term device driver here is quite different from its meaning in a PC environment. In order to help microcontroller software developers, microcontroller vendors usually provide header files and C codes that include:

- Definitions of peripheral registers
- Access functions for configuring and accessing the peripherals

By adding these files to your software projects, you can access various peripheral functions via function calls and access peripheral registers easily. If you want to, you can also create modified versions of the access functions based on the methods shown in the driver code and optimize them for your application.

2.2.5 Examples

Don't forget to download some example code from the microcontroller vendor's website. Most of the microcontroller vendors put their device-driver codes and examples on their websites as free downloads. This can save you a lot of time in developing new applications.

2.2.6 Documentation and other resources

Aside from user manuals of the microcontrollers, often you can also find application notes, FAQs, and online discussion forums on microcontroller vendor websites. The user manuals are essential, as they provides the details of the peripherals' programmer models.

On the ARM® website, the documentation is placed in a section called Info Center (http://infocenter.arm.com). From there you can find the Cortex®-M3/M4 Devices Generic User Guides (references 2 and 3), which covers the programming model of the processors, as well as various application notes.

Finally, you can also find a number of useful application notes and online discussion forums from tool vendor websites.

2.2.7 Other equipment

Depending on the applications you are developing and the development board you are using, you might need additional hardware that interfaces to the development boards, such as external LCD display modules or communication interface adaptors. Also you might need some hardware development tools like a laboratory power supply, logic analyzer/oscilloscope, signal generator, etc.

2.3 Software development flow

The software development flow depends on the compiler suite you use. Assuming that you are using a compiler suite with Integrated Development Environment (IDE), the software development flow (as shown in Figure 2.4) usually involves:

Create project – you need to create a project that will specify the location of source files, compile target, memory configurations, compilation options, and so on. Many IDEs have a project creation wizard for this step.

Add files to project – you need to add the source code files required by the project. You might also need to specify the path of any included header files in the project options. Obviously you might also need to create new program source code files and write the program. Note that you should be able to reuse a number of files from the device-driver library to reduce the effort in writing new files. This includes startup code, header files, and some of the peripheral control functions.

Setup project options – In most cases, the project file created allows a number of project options such as compiler optimization options, memory map, and output file types. Based on the development board and debug adaptor you have, you might also need to setup options for debug and code download.

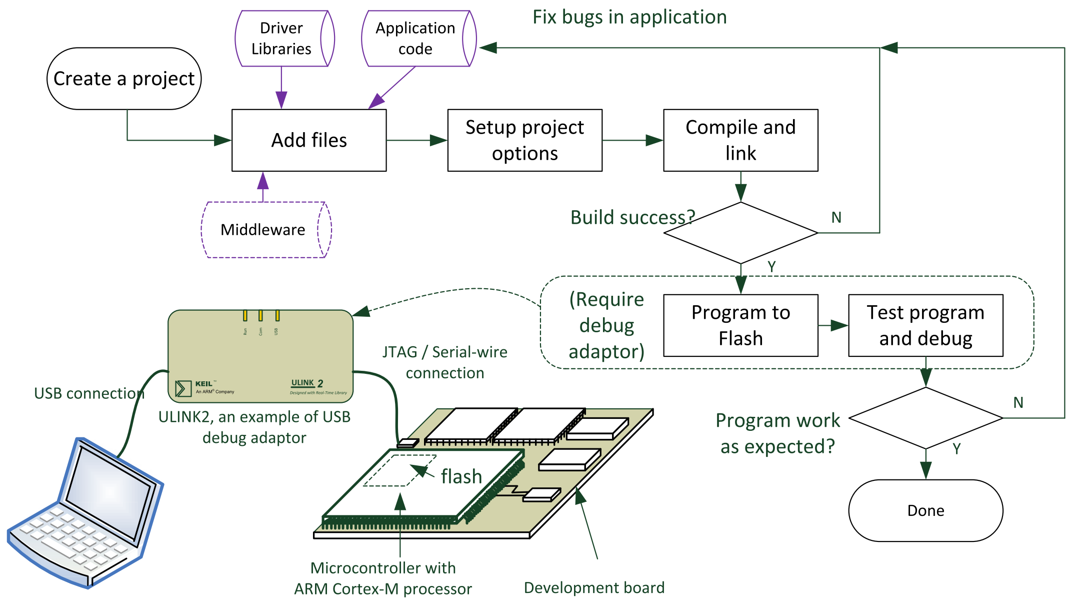

FIGURE 2.4

A simplified software development flow

Compile and link – In most cases, a project contains a number of files that are compiled separately. After the compilation process, each source file will have a corresponding object file. In order to generate the final combined executable image, a separate linking process is required. After the link stage, the IDE can also generate the program image in other file formats for the purpose of programming the image to the device.
Flash programming – Almost all of the Cortex®-M microcontrollers use flash memories for program storage. After a program image is created, we need to download the program to the flash memory of the microcontroller. To do this, you need a debug adaptor if the microcontroller board you use does not have one built in. The actual flash programming procedures can be quite complex, but these are usually fully handled by the IDE and you can carry out the whole programming process with a single mouse click. Note that if you want to, you can also download applications to SRAM and execute them from there.
Execute program and debug – After the compiled program is downloaded to the microcontroller, you can then run the program and see if it works. You can use the debug environment in the IDE to stop the processor (commonly referred as halt) and check the status of the system to ensure it is working properly. If it doesn't work correctly, you can use various debug features like single stepping to examine the program operations in detail. All these operations will require a debug adaptor (or the one built in to the development kit if available) to link up the IDE and the microcontroller being tested. If a software bug is found, then you can edit your program code, recompile the project, download the code to the microcontroller, and test it again.

If you are using open source toolchain, you might not have an IDE and might need to handle the compile and link process using scripts or makefile. Depending on the microcontroller product you are using, there can be third-party tools that can be used to download the compiled program image to the flash memory in the microcontroller.

During execution of the compiled program, you can check the program execution status and results by outputting information via various I/O mechanisms such as a UART interface or an LCD module. A number of examples in this book will show how some of these methods can be implemented. See Chapter 18 for some of the examples.

2.4 Compiling your applications

The procedure for compiling an embedded program depends on the development tools you use. Later in this book, a number of chapters cover the use of a couple of development tools to compile simple applications (Chapters 15 to 17). Here we will first have a look at some basic concepts of the compilation process.

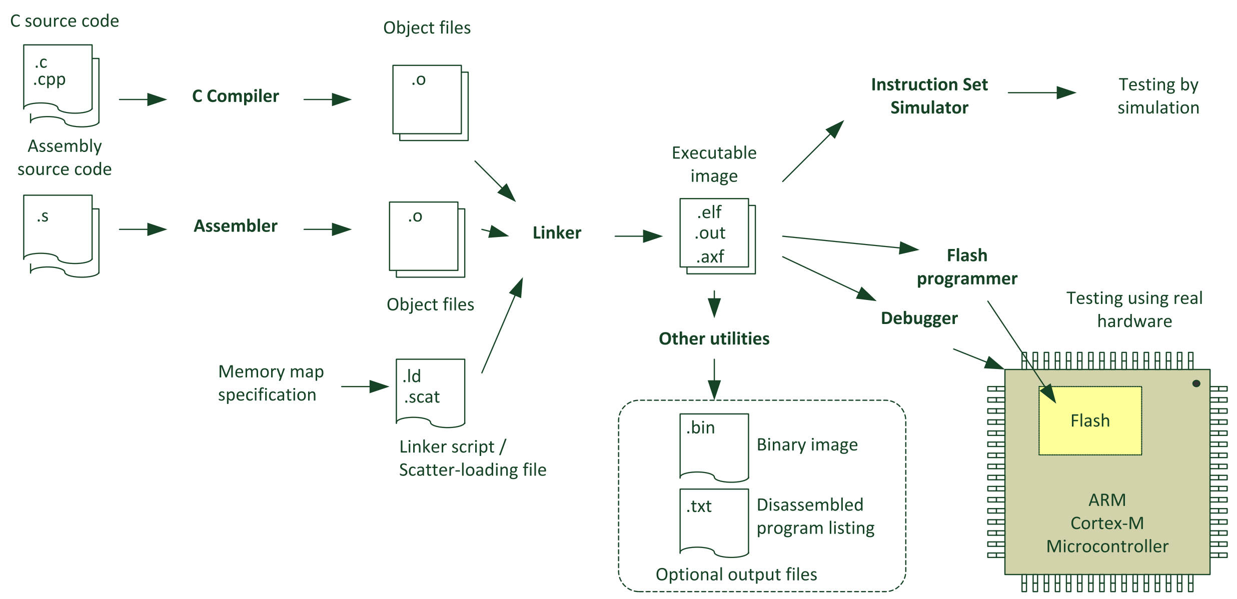

FIGURE 2.5

Common software compilation flow

First, we assume that you are developing your project using C programming language. This is the most commonly used programming language for microcontroller software development. Your project might also contain some assembly language files; for example, startup code that is supplied by microcontroller vendors. In most cases, the compilation process will be similar to the one shown in Figure 2.5.

Most development suites contain the tools listed in Table 2.1.

Different development tools have different ways to specify the layout of the program and data memory in the microcontroller system. In ARM® toolchains, you can use a file type called scatter-loading file, or in the case of Keil™ MDK-ARM, the scatter-loading file can be generated automatically by the µVision development environment. For some other ARM toolchains, you can also use command line options to specify the locations of ROM and RAM.

In a GNU-based toolchain, the memory specification is handled by linker scripts. These scripts are typically included in the installation of commercial gcc toolchains. However, some gcc users might have to create these files themselves. A later chapter of this book contains examples for compiling programs using gcc, which covers more information on linker scripts.

When using the GNU gcc toolchain, it is common to compile the whole application in one go instead of separating the compilation and linking stages (Figure 2.6).

The gcc compilation automatically invokes the linker and assembler if needed. This arrangement ensures that the details of the required parameters and libraries are passed on to the linker correctly. Using the linker as a separate step can be error prone and therefore is not recommended by most gcc tool vendors.

Table 2.1 Various Tools You Can Find in a Development Suite

Tools	Descriptions
C compiler	To compile C program files into object files
Assembler	To assemble assembly code files into object files
Linker	A tool to join multiple object files together and define memory configuration
Flash programmer	A tool to program the compiled program image to the flash memory of the microcontroller
Debugger	A tool to control the operation of the microcontroller and to access internal operation information so that status of the system can be examined and the program operations can be checked
Simulator	A tool to allow the program execution to be simulated without real hardware
Other utilities	Various tools, for example, file converters to convert the compiled files into various formats

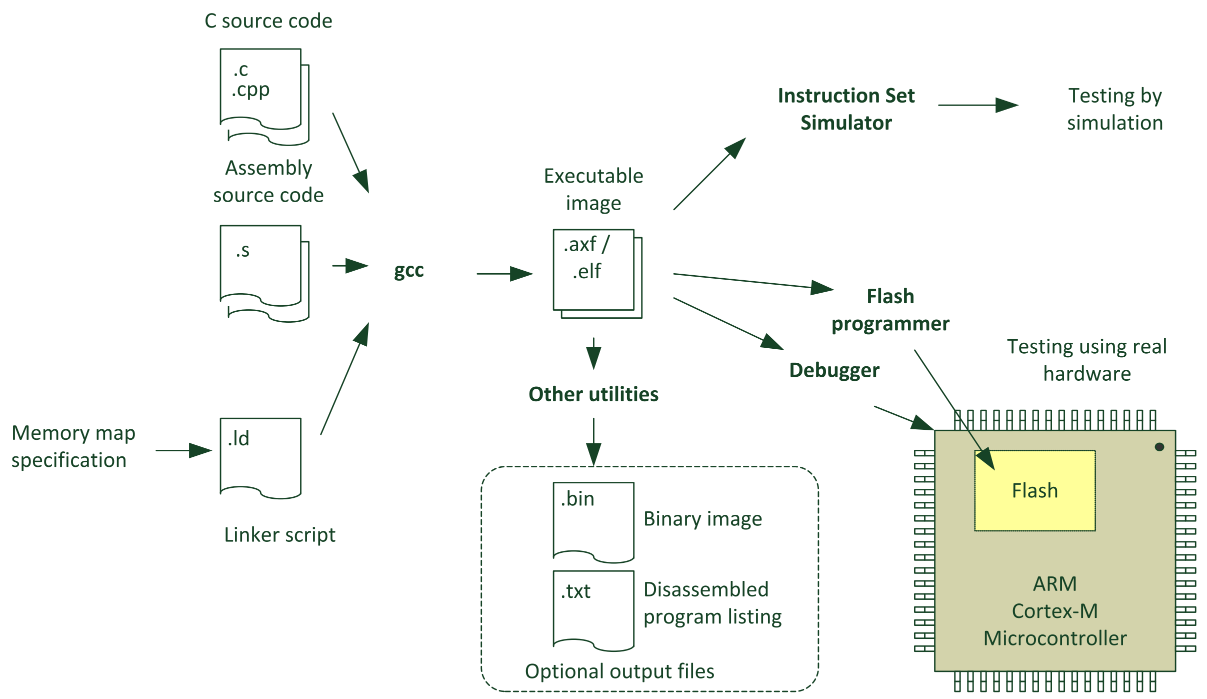

FIGURE 2.6

Common software compilation flow for GNU toolchain

2.5 Software flow

There are many ways to construct program flow for an application. Here we will cover some of the basic concepts.

2.5.1 Polling

For very simple applications, the processor can wait until there is data ready for processing, process it, and then wait again. This is very easy to setup and works fine for simple tasks. Figure 2.7 shows a simple polling program flow chart.

In most cases, a microcontroller will have to serve multiple interfaces and therefore be required to support multiple processes. The polling program flow method can be expanded to support multiple processes easily (Figure 2.8). This arrangement is sometimes called a "super-loop."

The polling method works well for simple applications, but it has several disadvantages. For example, when the application gets more complex, the polling loop design might get very difficult to maintain. Also, it is difficult to define priorities between different services using polling — you might end up with poor responsiveness, where a peripheral requesting service might need to wait a long time while the processor is handling less important tasks.

2.5.2 Interrupt driven

Another main disadvantage of the polling method is that it is not energy efficient. Lots of energy is wasted during the polling when service is not required. To solve this

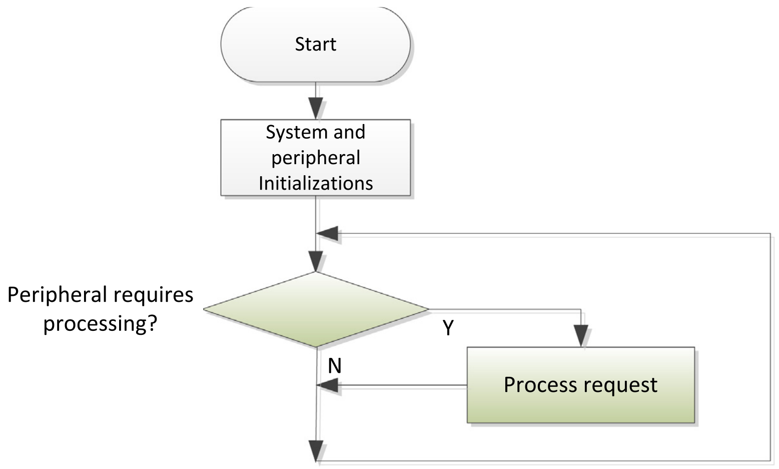
Start
System and peripheral Initializations
Peripheral requires processing?
Y
N
Process request

FIGURE 2.7

Polling method for simple application processing

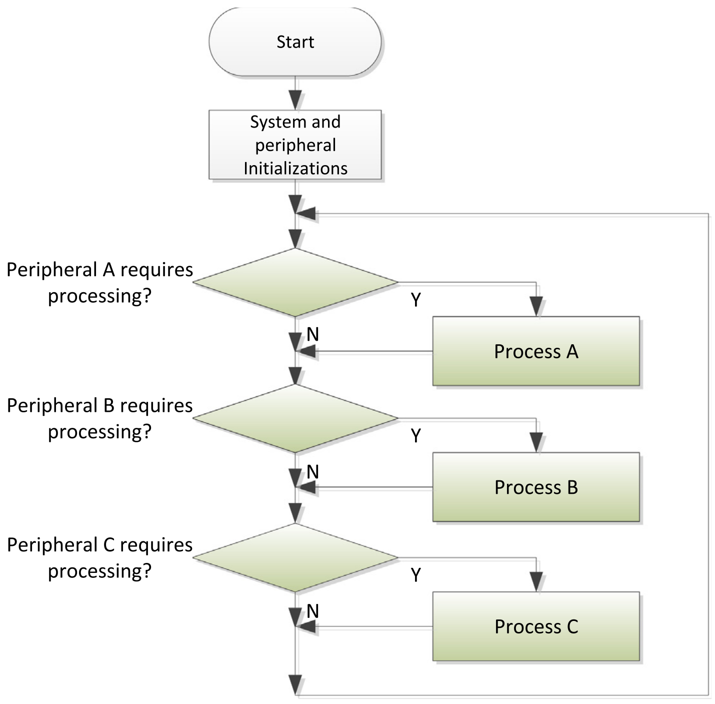
Start
System and peripheral Initializations
Peripheral A requires processing?
Y
N
Process A
Peripheral B requires processing?
Y
N
Process B
Peripheral C requires processing?
Y
N
Process C

FIGURE 2.8

Polling method for application with multiple devices that need processing

problem, almost all microcontrollers have some sort of sleep mode support to reduce power, in which the peripheral can wake up the processor when it requires a service (Figure 2.9). This is commonly known as an interrupt-driven application.

In an interrupt-driven application, interrupts from different peripherals can be assigned with different interrupt priority levels. For example, important/critical peripherals can be assigned with a higher priority level so that if the interrupt arrives when the processor is servicing a lower priority interrupt, the execution of the lower priority interrupt service is suspended, allowing the higher priority interrupt service to start immediately. This arrangement allows much better responsiveness.

In some cases, the processing of data from peripheral services can be partitioned into two parts: the first part needs to be done quickly, and the second part can be carried out a little bit later. In such situations we can use a mixture of interrupt-driven and polling methods to construct the program. When a peripheral requires service, it triggers an interrupt request as in an interrupt-driven application. Once the first part of the interrupt service is carried out, it updates some software variables so that the second part of the service can be executed in the polling-based application code (Figure 2.10).

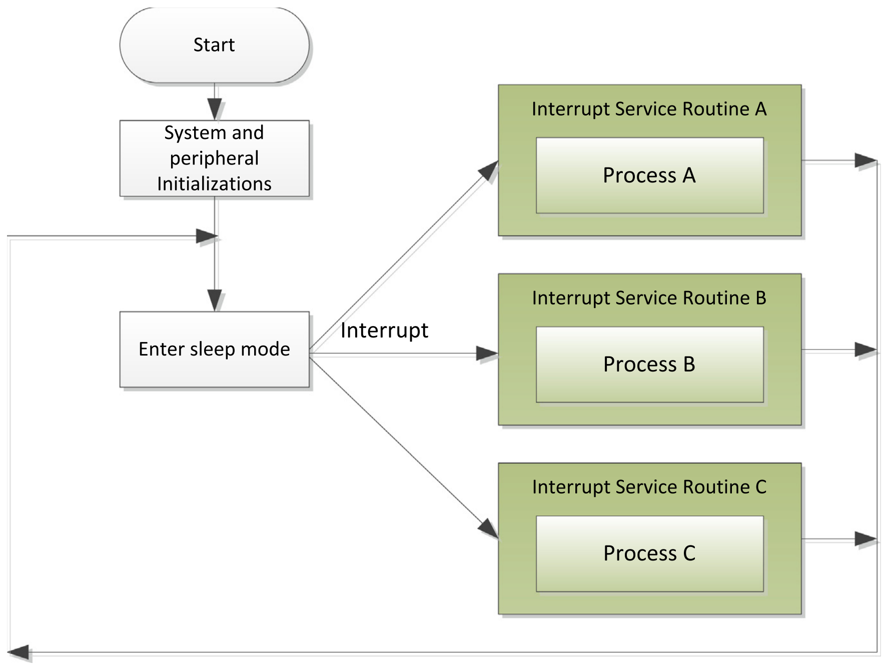
Start
System and peripheral Initializations
Enter sleep mode
Interrupt
Interrupt Service Routine A
Process A
Interrupt Service Routine B
Process B
Interrupt Service Routine C
Process C

FIGURE 2.9

Simple interrupt-driven application

Using this arrangement, we can reduce the duration of high-priority interrupt handlers so that lower priority interrupt services can get served quicker. At the same time, the processor can still enter sleep mode to save power when no servicing is needed.

2.5.3 Multi-tasking systems

When the applications get more complex, a polling and interrupt-driven program structure might not be able to handle the processing requirements. For example, some tasks that can take a long time to execute might need to be processed concurrently. This can be done by dividing the processor's time into a number of time slots and allocating the time slots to these tasks. While it is technically possible to create such an arrangement by manually partitioning the tasks and building a simple scheduler to handle this, it is often impractical to do this in real projects as it is time consuming and can make the program much harder to maintain and debug.

In these applications, a Real-Time Operating System (RTOS) can be used to handle the task scheduling (Figure 2.11). An RTOS allows multiple processes to be executed concurrently, by dividing the processor's time into time slots and allocating the time slots to the processes that require services. A timer is need to handle the timekeeping for the RTOS, and at the end of each time slot, the timer generates a timer interrupt, which triggers the task scheduler and decides if context switching should be carried

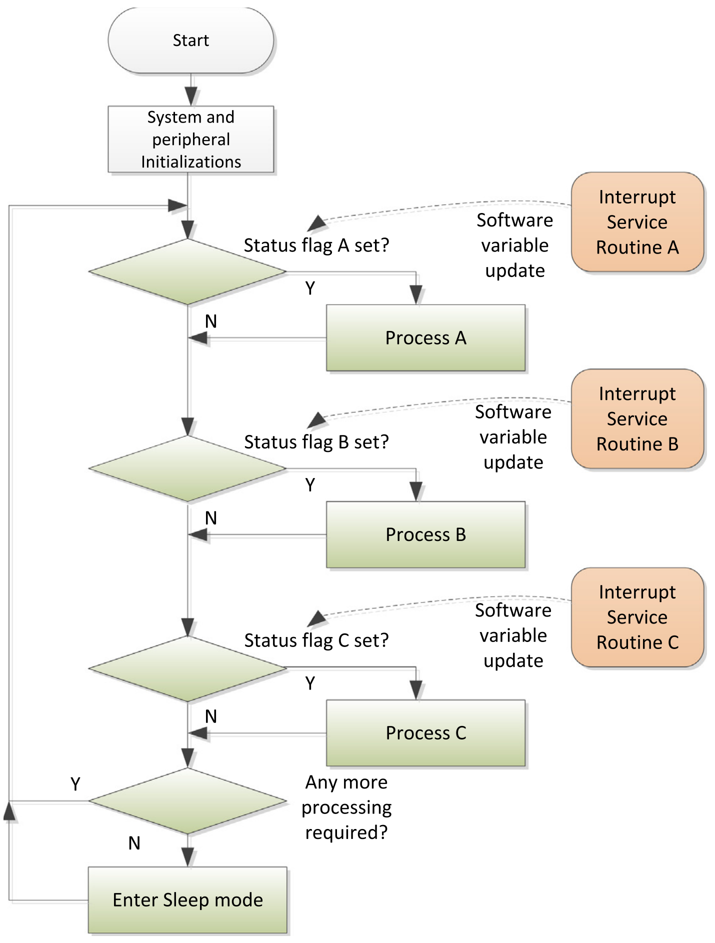

FIGURE 2.10

Application with both polling method and interrupt-driven arrangement

out. If yes, the current executing process is suspended and the processor executes another process.

Besides task scheduling, RTOSs also have many other features such as semaphores, message passing, etc. There are many RTOSs developed for the Cortex$^{®}$-M processors, and many of them are completely free of charge.

2.6 Data types in C programming

The C programming language supports a number of "standard" data types. However, the way a data item is represented in hardware depends on the processor architecture

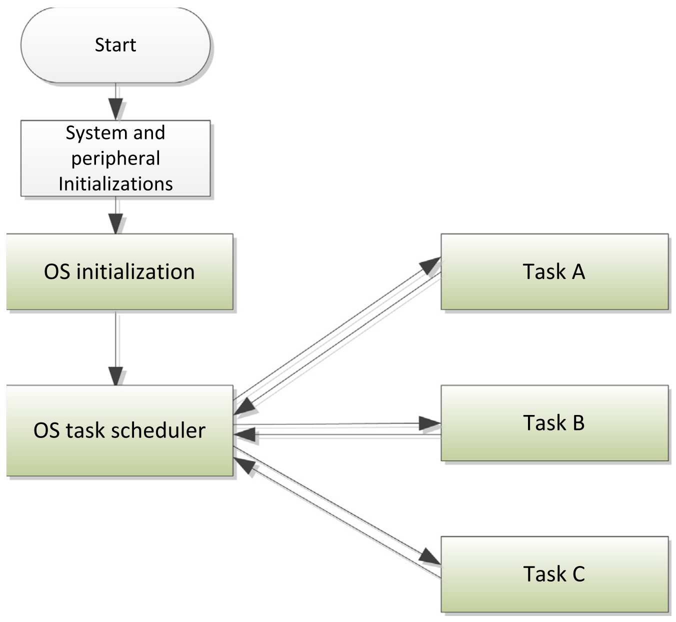

FIGURE 2.11

Using an RTOS to handle multiple tasks

as well as the C compiler. In different processor architectures, the size of certain data types can be different. For example, the integer is often 16-bit in 8-bit or 16-bit microcontrollers, and is always 32-bit in the ARM® architecture. Table 2.2 shows the common data types in ARM architecture, including all Cortex®-M processors. These data types are supported by all C compilers.

Because of differences of size in certain data types, it might be necessary to modify the source code when porting an application from an 8-bit or 16-bit microcontroller to an ARM Cortex-M microcontroller. More details on porting software from 8-bit and 16-bit architectures are covered in Chapter 24.

In ARM programming, we also refer to the size of a data item as BYTE, HALF WORD, WORD, and DOUBLE WORD, as shown in Table 2.3.

These terms are very common in ARM documentation, including the instruction set descriptions as well as hardware descriptions.

2.7 Inputs, outputs, and peripherals accesses

Almost all microcontrollers have various Input/Output (I/O) interfaces and peripherals such as timers, Real-time Clock (RTC), and so on. For microcontroller products based on the ARM® Cortex®-M3, and M4 processors, as well as common

Table 2.2 Size and Range of Data Types in ARM Architecture Including 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	

Table 2.3 Data Size Definition in ARM Processor

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

interface peripherals such as GPIO, SPI, UART, I2C, you can also find many advanced interface peripherals like USB, CAN, Ethernet, and analogue interfaces like ADCs (Analog to Digital Converters) and DACs (Digital to Analog Converters). Most of these interface peripherals are vendor specific, so you need to read the user manuals provided by the microcontroller vendors to learn how to use them. In most cases you can also find programming examples on microcontroller vendor websites.

On these microcontrollers, the peripherals are memory-mapped, which means the registers are accessible from the system memory map. In order to access these peripherals registers in C programs, we can use pointers. We will see some examples of how this can be done in the following sections.

Typically, a peripheral requires an initialization process before it can be used. This might include some of the following steps:

- Programming the clock control circuitry to enable the clock signal connection to the peripheral, and clock signal connection to corresponding I/O pins if needed. Many modern microcontrollers allow fine tuning of clock signal distribution, such as enabling/disabling the clock connection to each individual peripheral for better energy saving. Typically the clocks to peripherals are turned off by default and you need to enable the clock before programming the peripheral. In some cases you might also need to enable the clock to the peripheral bus system.
- In some cases you might need to configure the operation mode of the I/O pins. Most microcontrollers have multiplexed I/O pins that can be used for multiple purposes. In order to use a peripheral, it might be necessary to configure its I/O pins to match the usage (e.g., input/output direction, function, etc.). In addition, you might also need to program additional configuration registers to define the expected electrical characteristics such as output type (voltage, pull up/down, open drain, etc.).
- Peripheral configuration. Most peripherals contain a number of programmable registers that need configuration before using the peripheral. In some cases, you can find the programming sequence a bit more complex than that of a 8-bit microcontroller, because the peripherals on 32-bit microcontrollers are often much more sophisticated than peripherals on 8-bit/16-bit systems. On the other hand, often the microcontroller vendors will have provided device-driver library code and you can use these driver functions to reduce the programming work required.
- Interrupt configuration. If a peripheral is to be used with interrupt operations, you will need to program the interrupt controller on the Cortex-M3/M4 processor (NVIC) to enable the interrupt and to configure the interrupt priority level.

All these initialization steps are carried out by programming peripheral registers in various peripheral blocks. As mentioned, peripheral registers are memory-mapped and therefore can be accessed using pointers. For example, you can define a General Purpose Input Output (GPIO) register set as a number of pointers as:

```
/* STM32F100RBT6B – GPIO A Port Configuration Register Low */
#define GPIOA_CRL (*((volatile unsigned long *) (0x40010800)))
/* STM32F100RBT6B – GPIO A Port Configuration Register High */
#define GPIOA_CRH (*((volatile unsigned long *) (0x40010804)))
/* STM32F100RBT6B – GPIO A Port Input Data Register */
#define GPIOA_IDR (*((volatile unsigned long *) (0x40010808)))
/* STM32F100RBT6B – GPIO A Port Output Data Register */
#define GPIOA_ODR (*((volatile unsigned long *) (0x4001080C)))
/* STM32F100RBT6B – GPIO A Port Bit Set/Reset Register */
#define GPIOA_BSRR(*((volatile unsigned long *) (0x40010810)))
/* STM32F100RBT6B – GPIO A Port Bit Reset Register */
#define GPIOA_BRR (*((volatile unsigned long *) (0x40010814)))
/* STM32F100RBT6B – GPIO A Port Configuration Lock Register */
#define GPIOA_LCKR (*((volatile unsigned long *) (0x40010818)))
```

Then we can use the definitions directly. For example:

```
void GPIOA_reset(void) /* Reset GPIO A */
{
  // Set all pins as analog input mode
  GPIOA_CRL = 0; // Bit 0 to 7, all set as analog input
  GPIOA_CRH = 0; // Bit 8 to 15, all set as analog input
  GPIOA_ODR = 0; // Default output value is 0
  return;
}
```

This method is fine for a small number of peripheral registers. However, as the number of peripheral registers increases, this coding style can be problematic because:

- For each register address definition, the program needs to store the 32-bit address constant, resulting in increased code size.
- When there are multiple instantiations of the same peripheral, for example, the STM32 microcontroller has five GPIO peripherals, and the same definition has to be repeated for each of the instantiations. This is not scalable and makes it hard for software maintenance.
- It is not easy to create a function that can be shared between multiple instantiations of the same peripheral. For example, with the above example definition we might have to create the same GPIO reset function for each of the GPIO ports, resulting in increased code size.

In order to solve these problems, the common practice is to define the peripheral registers as data structures. For example, in the device-driver software package from the microcontroller vendors, we can find:

```
typedef struct
{
  __IO uint32_t CRL;
  __IO uint32_t CRH;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t BRR;
  __IO uint32_t LCKR;
} GPIO_TypeDef;
```

Then each peripheral base address (GPIO A to GPIO G) is defined as pointers to the data structure:

```
#define PERIPH_BASE  ((uint32_t)0x40000000)
    /*!< Peripheral base address in the bit-band region */
...
#define APB2PERIPH_BASE  (PERIPH_BASE + 0x10000)
...
```

```
#define GPIOA_BASE  (APB2PERIPH_BASE + 0x0800)
#define GPIOB_BASE  (APB2PERIPH_BASE + 0x0C00)
#define GPIOC_BASE  (APB2PERIPH_BASE + 0x1000)
#define GPIOD_BASE  (APB2PERIPH_BASE + 0x1400)
#define GPIOE_BASE  (APB2PERIPH_BASE + 0x1800)
...
#define GPIOA   ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB   ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC   ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOD   ((GPIO_TypeDef *) GPIOD_BASE)
#define GPIOE   ((GPIO_TypeDef *) GPIOE_BASE)
...
```

In these code snippets, there are a number of new things we have not covered:

The "__IO" is defined in a standardized header file in CMSIS. It implies a volatile data item (e.g., a peripheral register), which can be read or written to by software. Aside from "__IO," a peripheral register can also be defined as "__I" (read only) and "__O" (write only).

```
#ifdef __cplusplus
 #define __I volatile       /*!< defines 'read only' permissions */
#else
 #define __I volatile const /*!< defines 'read only' permissions */
#endif
#define __O volatile  /*!< defines 'write only' permissions */
#define __IO volatile /*!< defines 'read / write' permissions */
```

The "uint32_t" (unsigned 32-bit integer) is a data type supported in C99. This ensures the data size is 32-bit, independent of the processor architecture, which can help the software to be more portable. To use this data type, the project needs to include the standard data type header (Note: if you are using a CMSIS-compliant device header file this is already done for you in the device header file):

```
#include <stdint.h>/* Include standard types */
  /* C99 standard data types:
     uint8_t  : unsigned 8-bit, int8_t   : signed 8-bit,
     uint16_t : unsigned 16-bit, int16_t : signed 16-bit,
     uint32_t : unsigned 32-bit, int32_t : signed 32-bit,
     uint64_t : unsigned 64-bit, int64_t : signed 64-bit
  */
```

When peripherals are declared using such a method, we can create functions that can be used for each instance of the peripheral easily. For example, the code to reset the GPIO port can be written as:

```
void GPIO_reset(GPIO_TypeDef* GPIOx)
{
  // Set all pins as analog input mode
  GPIOx->CRL = 0; // Bit 0 to 7, all set as analog input
  GPIOx->CRH = 0; // Bit 8 to 15, all set as analog input
  GPIOx->ODR = 0; // Default output value is 0
  return;
}
```

To use this function, we just need to pass the peripheral base pointer to the function:

```
GPIO_reset(GPIOA); /* Reset GPIO A */
GPIO_reset(GPIOB); /* Reset GPIO B */
...
```

This method for declaring peripheral registers is used by almost all of the Cortex-M microcontroller device-driver packages.

2.8 Microcontroller interfaces

The applications running in the microcontroller connect with external world using various peripheral interfaces. While usage of peripheral interfaces is not the main focus of this book, a few basic examples will be covered. In most cases, you can use device-driver library software packages from the microcontroller vendors to simplify the software development, and you can find examples and application notes on the Internet for such information.

Unlike programming for PCs, most embedded applications do not have a rich GUI. Some development boards might have an LCD screen, but many others just have a couple of LEDs and buttons. While the application itself might not require a user interface, often a simple text-based communication method is very useful for software development. For example, it can be handy to able to use printf to display a value captured by the Analog-to-Digital Converter (ADC) during program execution.

A number of methods can be used to handle such message display:

- Using a character LCD display module connected to the I/O pins of the microcontroller
- Using a simple UART to communicate with a terminal program running on a PC
- Set up a USB interface on the microcontroller as a virtual COM port to communicate with a terminal program running on a PC
- Use the Instrumentation Trace Macrocell (ITM), a standard debug feature on the Cortex$^{®}$-M3/M4, to communicate with the debugger software

In some cases, a character LCD might be part of the embedded product, so using this hardware to display information can be convenient. However, the size of the screen limits the amount of information that can be displayed at a time.

A UART is easy to use, and allows more information to be passed to the developer quickly. The Cortex-M3/M4 processor does not have a UART as standard, but most microcontroller vendors have included a UART peripheral in their microcontroller designs. However, most modern computers do not have a UART interface (COM port) anymore, so you might need to use a USB-to-UART adaptor cable to

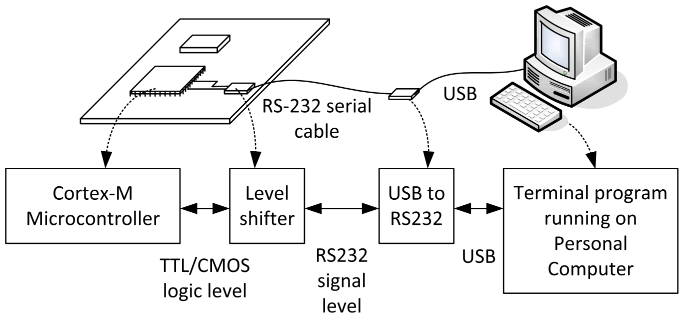

FIGURE 2.12

Using a UART to communicate with a PC via USB

handle this communication. In addition, you need to have a TTL-to-RS232 adaptor in your development setup to convert the signal's voltage (see Figure 2.12).

In some development boards (e.g., Texas Instruments Stellaris LaunchPad), the onboard debug adaptor has the feature of converting UART communications to USB.

If the microcontroller you use has a USB interface, you can use this to communicate with a PC using USB. For example, you can use a Virtual COM port solution for text-based communication with a terminal program running on a computer. It requires more effort in setting up the software but allows the microcontroller hardware to interface with the PC directly, avoiding the cost of the RS232 adaptors.

If you are using commercial debug adaptors like the Keil ULINK2, Segger J-LINK, or similar, you can use a feature called Instrumentation Trace Macrocell (ITM) to transfer messages to the debug host (the PC running the debugger) and display the messages in the development environment. This does not require any extra hardware and does not require much software overhead. It allows the peripheral interfaces to be free for other purposes. Examples of using the ITM are covered in Chapter 18.

The technique to redirect text messages from a "printf" (in C language) to specific hardware (e.g., UART, character LCD, etc.) is commonly referred as "retargeting." Retargeting can also be used to handle user inputs and system functions. The C code for retargeting is toolchain specific. Examples of retargeting for a couple of development tools will be covered in Chapter 18.

2.9 The Cortex® microcontroller software interface standard (CMSIS)

2.9.1 Introduction to CMSIS

Earlier in this chapter we mentioned CMSIS. CMSIS was developed by ARM® to allow microcontroller and software vendors to use a consistent software infrastructure to develop software solutions for Cortex®-M microcontrollers. As such, you can see that many software products for Cortex-M microcontrollers are CMSIS-compliant.

Currently the Cortex-M microcontroller market comprises:

- More than 15 microcontroller vendors shipping Cortex-M microcontroller products (see section 1.1.4 for the list of Cortex-M3 and Cortex-M4 microcontroller vendors), with some other silicon vendors providing Cortex-M based FPGA and ASICs
- More than 10 toolchain vendors
- More than 30 embedded operating systems
- Additional Cortex-M middleware software providers for codecs, communication protocol stacks, etc.

With such a large ecosystem, some form of standardization of the way the software infrastructure works becomes necessary to ensure software compatibility with various development tools and between different software solutions.

At the same time, embedded systems are also becoming more and more complex, and the amount of effort in developing and testing the software has increased substantially. In order to reduce development time as well as reducing the risk of having defects in products, software reuse is becoming more and more common. In addition, the complexity of the embedded systems has also increased the use of third-party software solutions. For example, an embedded software project might involve software components from many different sources:

- Software developed by in house developers
- Software reused from other projects
- Device-driver libraries from microcontroller vendors
- Embedded OSs
- Other third-party software products such as communication protocol stacks

In such scenarios, the interoperability of various software components becomes critical. For all these reasons, ARM worked with various microcontroller vendors, tools vendors, and software solution providers to develop CMSIS, a software framework covering most Cortex-M processors and Cortex-M microcontroller products.

The aims of CMSIS include:

- Enhanced software reusability — makes it easier to reuse software code in different Cortex-M projects, reducing time to market and verification efforts.
- Enhanced software compatibility — by having a consistent software infrastructure (e.g., API for processor core access functions, system initialization method, common style for defining peripherals), software from various sources can work together, reducing the risk in integration.
- Easy to learn — the CMSIS allows easy access to processor core features from the C language. In addition, once you learn to use one Cortex-M microcontroller product, starting to use another Cortex-M product is much easier because of the consistency in software setup.
- Toolchain independent — CMSIS-compliant device drivers can be used with various compilation tools, providing much greater freedom.

- Openness — the source code for CMSIS core files can be downloaded and accessed by everyone, and everyone can develop software products with CMSIS.

CMSIS is an evolving project. It started out as a way to establish consistency in device-driver libraries for the Cortex-M microcontrollers, and this has become CMSIS-Core. Since then additional CMSIS projects have started:

- CMSIS-Core (Cortex-M processor support) — a set of APIs for application or middleware developers to access the features on the Cortex-M processor regardless of the microcontroller devices or toolchain used. Currently the CMSIS processor support includes the Cortex-M0, Cortex-M0+, Cortex-M3, and Cortex-M4 processors and SecurCore products like SC000 and SC300. Users of the Cortex-M1 can use the Cortex-M0 version because they share the same architecture.
- CMSIS-DSP library — in 2010 the CMSIS DSP library was released, supporting many common DSP operations such as FFT and filters. The CMSIS-DSP is intended to allow software developers to create DSP applications on Cortex-M microcontrollers easily.
- CMSIS-SVD — the CMSIS System View Description is an XML-based file format to describe peripheral set in microcontroller products. Debug tool vendors can then use the CMSIS SVD files prepared by the microcontroller vendors to construct peripheral viewers quickly.
- CMSIS-RTOS — the CMSIS-RTOS is an API specification for embedded OS running on Cortex-M microcontrollers. This allows middleware and application code to be developed for multiple embedded OS platforms, and allows better reusability and portability.
- CMSIS-DAP — the CMSIS-DAP (Debug Access Port) is a reference design for a debug interface adaptor, which supports USB to JTAG/Serial protocol conversions. This allows low-cost debug adaptors to be developed which work for multiple development toolchains.

In this chapter we will first look at the processor support in CMSIS (CMSIS-Core). The CMSIS DSP library will be covered in Chapter 22. The CMSIS-SVD and CMSIS-DAP topics are beyond the scope of this book.

2.9.2 Areas of standardization in CMSIS-Core

From a software development point of view, the CMSIS-Core standardizes a number of areas:

Standardized definitions for the processor's peripherals — These include the registers in the Nested Vector Interrupt Controller (NVIC), a system tick timer in the processor (SysTick), an optional Memory Protection Unit (MPU), various programmable registers in the System Control Block (SCB), and some software programmable registers related to debug features. Note: Some of the registers in the Cortex®-M4 are not available in Cortex-M3, and similarly, some registers in Cortex-M3 and Cortex-M4 are not available in the Cortex-M0.

Standardized access functions to access processor's features — These include various functions for interrupt control using NVIC, and functions for accessing special registers in the processors. It is still possible to access the registers directly if needed, but for general programming using the access functions (or sometimes referred as Application Programming Interface, API, in some literature) can help software portability. More details of these functions are covered in Appendix E.

Standardized functions for accessing special instructions easily — The Cortex-M processors support a number of instructions for special purposes (e.g., Wait-For-Interrupt, WFI, for entering sleep mode). These instructions cannot be generated using generic IEC/ISO C[1] language. Instead, CMSIS implements a set of functions to allow these instructions to be accessed within C program code. Without these functions, the users would have to rely on toolchain specific solutions such as intrinsic functions or inline assembly to inject special instructions into the application, which make the software less reusable and might require certain in-depth knowledge of the toolchain in order to handle them correctly. CMSIS provides a standardized API for these features so that they can be easily used by application developers.

Standardized function names for system exception handlers — A number of system exception types are presented in the architecture for the Cortex-M processors. By giving the corresponding system exception handlers standardized names, it makes it much easier to develop software solutions that can be applied to multiple Cortex-M products. This is especially important for embedded OS developers, as the embedded OS requires the use of several types of system exception.

Standardized functions for system initialization — Most modern feature-rich microcontroller products require some configuration of clock circuitry and power management registers before the application starts. In CMSIS-compliant device-driver libraries, these configuration steps are placed in a function called "SystemInit()." Obviously, the actual implementation of this function is device specific and might need adaption for various project requirements. However, having a standardized function name, a standardized way that this function is used and a standardized location where this function can be found makes it much easier for a designer to pick up and start using a new Cortex-M microcontroller device.

Standardized software variables for clock speed information — This might not be obvious, but often our application code does need to know what clock frequency the system is running at. For example, such information might be needed for setting up the baud rate divider in a UART, or to initialize the SysTick timer for an embedded OS. A software variable called "SystemCoreClock" (for CMSIS 1.3 or newer versions, or "SystemFreq" in older versions of CMSIS) is defined in the CMSIS-Core.

In addition, the CMSIS-Core also provides:

A common platform for device-driver libraries — Each device-driver library has the same look and feel, making it easier for beginners to learn how to use the devices.

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

This also makes it easier for software developers to develop software for multiple Cortex-M microcontroller products.

2.9.3 Organization of CMSIS-Core

The CMSIS files are integrated into device-driver library packages from microcontroller vendors. Some of the files in the device-driver library are prepared by ARM® and are common to various microcontroller vendors. Other files are vendor/device specific. In a general sense, we can define the CMSIS into multiple layers:

- Core Peripheral Access Layer – Name definitions, address definitions, and helper functions to access core registers and core peripherals. This is processor specific and is provided by ARM.
- Device Peripheral Access Layer – Name definitions, address definitions of peripheral registers, as well as system implementations including interrupt assignments, exception vector definitions, etc. This is device specific (note: multiple devices from the same vendor might use the same file set).
- Access Functions for Peripherals – The driver code for peripheral accesses. This is vendor specific and is optional. You can choose to develop your application using the peripheral driver code provided by the microcontroller vendor, or you can program the peripherals directly if you prefer.

There is also a proposed additional layer for peripheral accesses:

Middleware Access Layer – This layer does not exist in current version of CMSIS. The idea is to develop a set of APIs for interfacing common peripherals such as UART, SPI, and Ethernet. If this layer exists, developers of middleware can develop their applications based on this layer to allow software to be ported between devices easily.

The roles of the various layers are summarized in Figure 2.13.

Note that in some cases, the device-driver libraries might contain additional vendor-specific functions for the NVIC implemented by the microcontroller vendor. The aim of CMSIS is to provide a common starting point, and the microcontroller vendors can add additional functions if they prefer. But software using these functions will need porting if the software design is to be reused on another microcontroller product.

2.9.4 How do I use CMSIS-Core?

The CMSIS files are included in the device-driver packages provided by the microcontroller vendors. So when you are using CMSIS-compliant device-driver libraries provided by the microcontroller vendors, you are already using CMSIS.

Typically, you need to do the following.

- Add source files to project. This includes:
  - Device-specific, toolchain-specific startup code, in the form of assembly or C
  - Device-specific device initialization code (e.g., *system_<device>.c*)
  - Additional vendor-specific source files for peripheral access functions. This is optional.

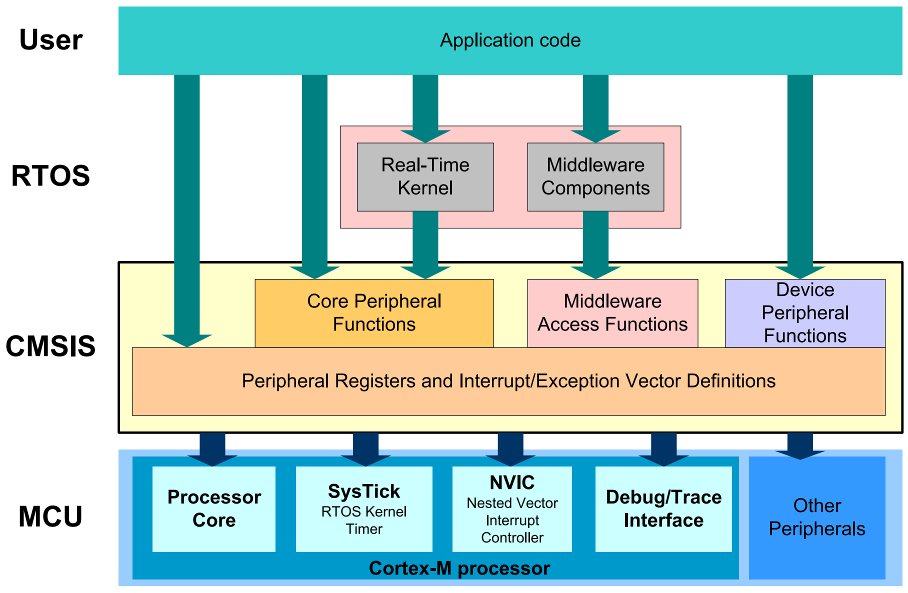
User
Application code
RTOS
Real-Time Kernel
Middleware Components
CMSIS
Core Peripheral Functions
Middleware Access Functions
Device Peripheral Functions
Peripheral Registers and Interrupt/Exception Vector Definitions
MCU
Processor Core
SysTick RTOS Kernel Timer
NVIC Nested Vector Interrupt Controller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

FIGURE 2.13

CMSIS-Core structure

- For CMSIS 2.00 or older versions of CMSIS-Core libraries, you might also need to add a processor-specific C program file (e.g., core_cm3.c) to the project for some of the core register access functions. This is not required from CMSIS-Core 2.10.

- Add header files into search path of the project. This includes:
  - A device-specific header file for peripheral registers definitions and interrupt assignment definitions. (e.g., *<device>.h*)
  - A device-specific header file for functions in device initialization code (e.g., *system_<device>.h)*
  - A number of processor-specific header files (e.g., *core_cm3.h*, *core_cm4.h*; they are generic for all microcontroller vendors)
  - Optionally additional vendor-specific header files for peripheral access functions
  - In some cases the development suites might also have some of the generic CMSIS support files pre-installed.

Figure 2.14 shows a typical project setup using a CMSIS device-driver package. Inside the device-driver package obtained from the microcontroller vendor, you will find the various files you need, including the CMSIS generic files. The names of some of these files depend on the actual microcontroller device name chosen by the microcontroller vendor (indicated as <device> in the diagram).

When the device-specific header file is included in the application code, it automatically includes additional header files, therefore you need to set up the project search path for the header files in order to compile the project correctly.

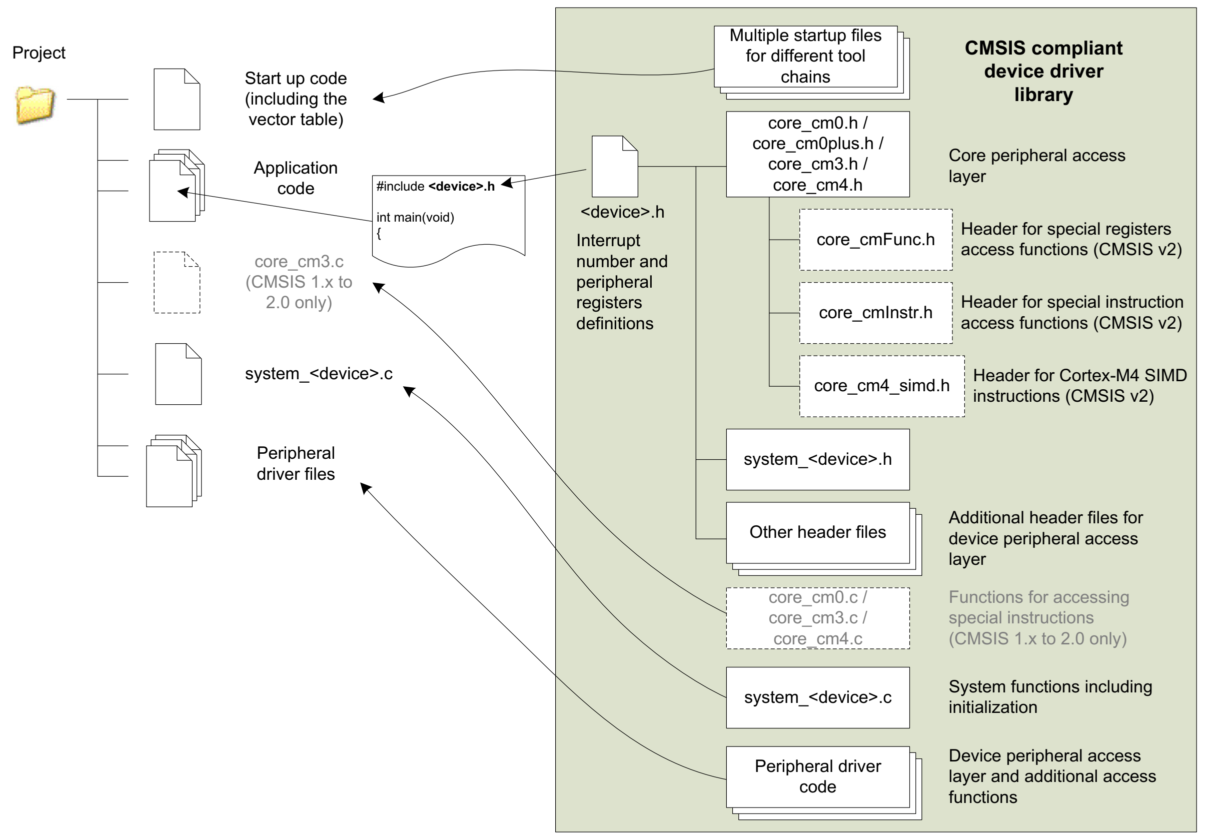
Project
Start up code (including the vector table)
Application code
#include <device>.h
int main(void)
{
core_cm3.c (CMSIS 1.x to 2.0 only)
system_<device>.c
Peripheral driver files
CMSIS compliant device driver library
Multiple startup files for different tool chains
<device>.h
Interrupt number and peripheral registers definitions
core_cm0.h / core_cm0plus.h / core_cm3.h / core_cm4.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_cm4_simd.h
Header for Cortex-M4 SIMD instructions (CMSIS v2)
system_<device>.h
Other header files
Additional header files for device peripheral access layer
core_cm0.c / core_cm3.c / core_cm4.c
Functions for accessing special instructions (CMSIS 1.x to 2.0 only)
system_<device>.c
System functions including initialization
Peripheral driver code
Device peripheral access layer and additional access functions

FIGURE 2.14

Using CMSIS-Core in a project

In some cases, the Integrated Development Environment (IDE) automatically sets up the startup code for you when you create a new project. Otherwise you just need to add the startup code from the device-driver library to the project manually. Startup code is required for the starting sequence of the processor, and it also includes the exception vector table definition that is required for interrupt handling.

2.9.5 Benefits of CMSIS-Core

So what does CMSIS mean to users?

The main advantage is much better software portability and reusability:

- A project for a Cortex®-M microcontroller device can be migrated to another device from the same vendor with a different Cortex-M processor very easily. Often microcontroller vendors provide devices with Cortex-M0/M0+/M3/M4 with the same peripheral and same pin out, and the change required is just replacing a couple of CMSIS files in the project.
- CMSIS-Core made it easier for a Cortex-M microcontroller project to be migrated to another device from a different vendor. Obviously, peripheral setup and access code will need to be modified, but processor core access functions are based on the same CMSIS source code and do not require changes.
  CMSIS allows software to be much more future proof because embedded software developed today can be reused on other Cortex-M products in the future.

The CMSIS-Core also allows faster time to market because:

- It is easier to reuse software code from previous projects.
- Since all CMSIS-compliant device drivers have a similar structure, learning to use a new Cortex-M microcontroller is much easier.
- The CMSIS code has been tested by many silicon vendors and software developers around the world. It is compliant with Motor Industry Software Reliability Association (MISRA). Therefore it reduces the validation effort required, as there is no need to develop and test your own processor feature access functions.
- Starting from CMSIS 2.0, a DSP library is included that provides tested, optimized DSP functions. The DSP library code is available as a free download and can be used by software developers free of charge.

There are also a number of other advantages:

- CMSIS is supported by multiple compiler toolchain vendors.
- CMSIS has a small memory footprint (less than 1KB for all core access functions and a few bytes of RAM for several variables).
- CMSIS files contain Doxygen tags (http://www.doxygen.org) to enable easy automatic generation of documentation.

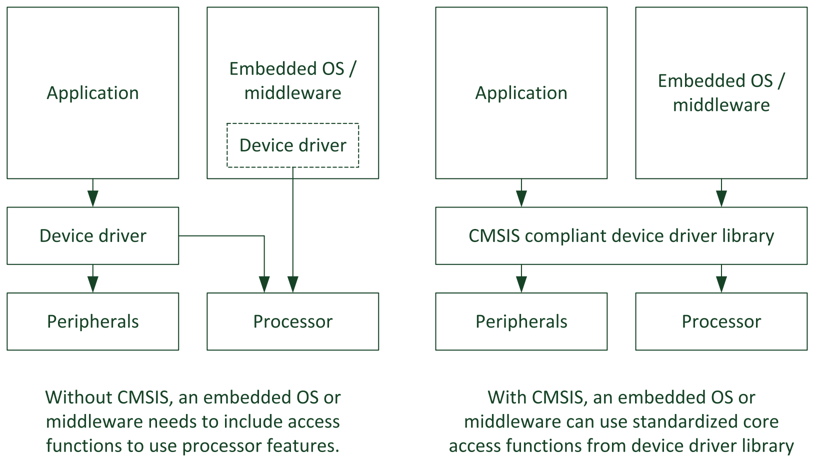
Application
Embedded OS / middleware
Device driver
Device driver
Peripherals
Processor
Without CMSIS, an embedded OS or middleware needs to include access functions to use processor features.
Application
Embedded OS / middleware
CMSIS compliant device driver library
Peripherals
Processor
With CMSIS, an embedded OS or middleware can use standardized core access functions from device driver library

FIGURE 2.15

CMSIS-Core avoids the need for middleware or OS to carry their own driver code

For developers of embedded OS and middleware, the advantage of CMSIS is significant:

- By using processor core access functions from CMSIS, embedded OS, and middleware can work with device-driver libraries from various microcontroller vendors, including future products that are yet to be released.
- Since CMSIS is designed to work with various toolchains, many software products can be designed to be toolchain independent.
- Without CMSIS, middleware might need to include a small set of driver functions for accessing processor peripherals such as the interrupt controller. Such an arrangement increases the program size, and might cause compatibility issues with other software products (Figure 2.15).

2.9.6 Various versions of CMSIS

The CMSIS project is evolving. Over the last few years several versions of CMSIS have been released, bringing wider processor support and improvements. Apart from coding improvement, there have also been a number of other changes:

Version	Main Changes
1.0	Nov 2008 Initial release. Support Cortex®-M3 processor only.
1.10	Feb 2009 Support for Cortex-M0 added.
1.20	May 2009 Add support for TASKING compiler. Add more functions to manage priority settings in NVIC.

Version	Main Changes
1.30	Oct 2009 The system initialization function *SystemInit()* is called in startup code instead of beginning of *main()*. SystemFrequency variable renamed to SystemCoreClock to reflect the processor clock definition. Additional functions "void SystemCoreClockUpdate(void)" added. Add support for data receive for debug communication. (Previous versions use ITM for data output in debug communication.) Add bit definition for processor's peripheral registers. Directory structure changed.
2.0	Nov 2010 Support for Cortex-M4 added. Included a CMSIS DSP library (CMSIS-DSP) for Cortex-M4 and Cortex-M3. New header files *core_cm4_simd.h*, *core_cmFunc.h* and *core_cmInst.h* introduced, with a number of core access functions are moved to these files and become inlined. Add CMSIS System View Description
2.10	July 2011 CMSIS-DSP library for Cortex-M0 added. Added big endian support for DSP library. Directory structure simplified. Processor specific C program files (e.g., *core_cm3.c*, *core_cm4.c*) are no longer required and are removed. Reworded CMSIS-DSP library example. Documentation update.
3.0	October 2011 Added support for GNU Tools for ARM Embedded Processors. Added function __ROR. Added Register Mapping for TPIU, DWT. Added support for SC000 and SC300 processors. Corrected ITM_SendChar function. Corrected the functions __STREXB, __STREXH, __STREXW for the GNU GCC compiler section. Documentation restructured.
3.01	March 2012 Added support for Cortex-M0+ processor. Integration of CMSIS DSP Library version 1.1.0.

In normal cases, embedded applications can work with different versions of the CMSIS source files without problems. Most microcontroller vendors keep their device-driver library up to date with the most recent versions of CMSIS, but there is always the chance that the device-driver library package from microcontroller vendors could be a couple of releases behind the latest CMSIS version. This is not usually a problem, as the functionalities of the driver functions remain unchanged.

In a few cases, application code might need to be updated to allow it to be used with a newer version of the CMSIS driver package (e.g., when the “SystemFrequency” variable is used, which is replaced by “SystemCoreClock” from CMSIS 1.3).

You can download the latest version of the CMSIS source package from http://www.arm.com/cmsis.