ARM® Cortex® M4 Cookbook

Embedded microcontrollers are at the core of many everyday electronic devices. The so-called Internet of Things drives the market for such technology, so much so that embedded cores now represent 90% of all processors sold. The ARM® Cortex® M4 is one of the most powerful microcontrollers on the market. The book begins with an introduction to the ARM® Cortex® family and we cover the installation of the ARM® uVision Integrated Development Environment and topics such as target devices, evaluation boards, code configuration, and GPIO. You will then learn about core programming topics. You will find out about advanced aspects such as data conversion, multimedia support, real-time signal processing, and real-time embedded systems. By the end of the book, you will be able to successfully create robust and scalable ARM® Cortex® based applications.

What this book will do for you...

- Use ARM's uVision MDK to configure the microcontroller runtime environment (RTE), create projects and compile, download, and run simple programs on an evaluation board
- Use and extend device family packs to configure I/O peripherals
- Develop multimedia applications using the touchscreen and audio codec beep generator
- Write multi-threaded programs using ARM's real-time operating system (RTOS)
- Write critical sections of code in assembly language and integrate these with functions written in C
- Fix problems using ARM's debugging tool to set breakpoints and examine variables
- Port uVision projects to other open source development environments

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions to solve problems efficiently
- Clear explanations of what you did
- Solutions that can be applied to solve real-world problems

Over 50 hands-on recipes that will help you develop amazing real-time applications using GPIO, RS232, ADC, DAC, timers, audio codecs, graphics LCD, and a touch screen

Dr. Mark Fisher


“Community Experience Distilled”

Free Sample
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'A Practical Introduction to ARM® CORTEX®'
- A synopsis of the book’s content
- More information on ARM® Cortex® M4 Cookbook
About the Author

Dr. Mark Fisher is a chartered engineer, MIET. He started his career as an electronics apprentice with the UK Ministry of Defence. This was before he studied Electrical and Electronic Engineering at Aston University, Birmingham. After his graduation, he joined Ferranti Computer Systems, Manchester. However, he returned to academia to study Microprocessor Engineering and Digital Electronics at Manchester University (UMIST), and he then remained as a research assistant within the Department of Computation to gain a PhD in Applied Machine Learning. Currently, he is a senior lecturer at the School of Computing Sciences, University of East Anglia, and the course director of the Computer Systems Engineering Degree programme. Many of the recipes in this book were originally developed in the context of a taught module that Mark leads, which is popular among undergraduate and master's students in the school.

Mark currently researches in the fields of medical imaging and computer vision, and he is a co-author of over a hundred journal and conference papers in this area.
This book begins with an introduction to the ARM Cortex family and covers its basic concepts. We cover the installation of the ARM uVision Integrated Development Environment and topics, such as target devices, evaluation boards, code configuration, and GPIO. You will learn about the core programming topics that deal with structures, functions, pointers, and debugging in this book. You will also learn about various advanced aspects, such as data conversion, multimedia support, real-time signal processing, and real-time embedded systems. You will also get accustomed with creating game applications, programming I/O, and configuring GPIO and UART ports. By the end of this book, you will be able to successfully create robust and scalable ARM Cortex-based applications.

What this book covers

Chapter 1, A Practical Introduction to ARM® Cortex®, shows you how to compile, download, and run simple programs on an evaluation board.

Chapter 2, C Language Programming, introduces you to writing programs in C, a high-level language that was developed in the 1970s and is popular among embedded-system developers.

Chapter 3, Programming I/O, investigates some of the functions that configure I/O devices, and you will gain an understanding of what is involved in writing I/O interfaces for other targets.

Chapter 4, Assembly Language Programming, explains how to write functions in assembly language. Assembly language is a low-level programming language that is specific to particular computer architecture. Therefore, unlike programs written high-level languages, programs written in assembly language cannot be easily ported to other hardware architectures.

Chapter 5, Data Conversion, introduces approaches to data conversion, namely analog to digital conversion and vice versa. This chapter also covers the principal features used by microcontrollers for data conversion.
Chapter 6, Multimedia Support, discusses support for various multimedia peripherals, which are discrete components connected to the microcontroller by a bus. Support for an LCD touchscreen, audio codec, and camera peripherals is a very attractive feature of the STM32F4xxx microcontroller, and selecting an evaluation board that includes these peripherals, although more expensive, will be covered in this chapter.

Chapter 7, Real-Time Signal Processing, introduces you to Digital Signal Processing (DSP) and reviews the ARM Cortex M4 instruction set support for DSP applications. This chapter will walk through a DMA application using the codec, followed by designing a low-pass filter.

Chapter 8, Real-Time Embedded Systems, shows you how to write a multithreaded program using flags for communication and ensuring mutual exclusion when accessing shared resources.

Chapter 9, Embedded Toolchain, teaches you how to install the GNU ARM Eclipse toolchain for the Windows Operating System and to build and run a simple Blinky program on the MCBSTM32F400 evaluation board. This chapter will also show you how to use the STM32CubeMX Framework (API) and how to port projects to GNU ARM Eclipse.
In this chapter, we will cover the following topics:

- Installing uVision5
- Linking an evaluation board
- Running an example program
- Writing a simple program
- Understanding the simple use of GPIO
- Estimating microcontroller performance

**Introduction**

This chapter will show you how to compile, download, and run simple programs on an evaluation board. A software tool called a Microcontroller Development Kit (MDK), including an Integrated Development Environment (IDE), is the simplest way of achieving this. Keil (a company owned by ARM) markets an extensive range of software tools to support embedded system development. Amongst these, the MDK-ARM development kit represents an integrated software development environment, supporting devices based on the Cortex-M (and associated) cores (see http://www.keil.com/arm/ldk.asp).
Installing uVision5

A free evaluation version of the IDE known as the MDK-ARM Lite edition, running (albeit with limited functionality) under the Windows operating system, is available for download. The main limitation of the environment is that programs that generate more than 32 KB of code cannot be compiled and linked (see http://www.keil.com/demo/limits.asp). However, since most programs written by novices tend to be quite small, this limitation is not a serious problem. For those who expect their executable image to exceed 32 KB, other open source compiler and IDE options are considered in Chapter 9, Embedded Toolchain.

uVision5, the latest version of the IDE is distributed as two components. An MDK core contains all the development tools, and software packs, together with Cortex Microcontroller Software Interface standard (CMSIS) and middleware libraries, which add support for target devices.

Installation involves downloading and running an executable (.exe) file. Users can download and install the latest version after first registering their contact details at http://www2.keil.com/mdk5/install/.

How to do it...

1. Download the latest version of the software by following the instructions provided by Keil. Device-specific libraries are not included in installations from version 5 onwards, so at the end of the installation, we must configure the IDE using the Pack Installer to choose the resources (that is target devices, boards, and examples) that we need.
2. Select the **Boards** tab, choose the **MCBSTM32F400** Keil evaluation board featuring the STM32F407IGx STMicroelectronics part, as this is the target for all the practical examples described in this cookbook.

3. With the **Packs** tab, in addition to the default installation options: CMSIS and Keil ARM Processional Middleware for ARM Cortex-M-based devices, board support for **MCBSTM32F400** is also needed. Select the latest version **Keil::32F4xx_DFP (2.6.0)**.

4. Select the **Examples** tab, and copy the board-specific example programs to a convenient local folder. Note: the example programs illustrate many useful features of the evaluation board, and are an invaluable resource.

5. Once we have downloaded and installed MDK-ARM uVision5, the IDE can be invoked from the Windows Taskbar. If we wish to update the installation, the pack installer can be invoked by selecting the pack installer icon on uVision5 toolbar.
6. We demonstrate the basic features of uVision in this chapter, but later on, we'll probably need to access the uVision user guide via the Help menu (also available at http://www2.keil.com/mdk5) to learn about the more advanced features of the IDE. A useful guide to getting started with uVision5 can be found at https://armkeil.blob.core.windows.net/product/mdk5-getting-started.pdf. An overview of uVision5 is available at http://www2.keil.com/mdk5, and this includes some video clips that describe the design philosophy, and explain how to use the Pack Installer and create a new project.

**How it works...**

Computer programming involves specifying a sequence of binary codes that are interpreted by the machine as instructions that together enable it to undertake some task. The instruction sets of early computers were small and easily memorized by programmers, so programs were written directly in machine code, and each instruction code word was set up on switches and written to memory. Finally, once all the instructions had been entered, the program was executed. With the development of more powerful machines and larger instruction sets, this approach became unworkable. This motivated the need to program in higher level (human understandable) languages that are translated into machine code by a special program called a compiler. Modern day programmers rarely need to interpret individual binary codes; instead, they use a text editor to enter a sequence of high-level language statements, a compiler to convert them into machine code, a linker to allow programs to reuse previously written (library) code, and a loader to write the binary codes to memory. The steps comprising edit, compile, link, load can be undertaken by running each program (editor, compiler, linker, loader) separately. However, nowadays they are usually packaged together within a wrapper called an IDE. Some IDEs are language-specific and some are customizable, allowing developers to create bespoke programming environments for any target language and/or machine.

The pack installer framework allows MDK-ARM uVision5 to be customized and extended to target a large number of devices and evaluation boards using ARM cores. But while, IDEs represent the most popular and efficient route to programming, uVision represents just one of a number of IDEs that are widely available. Other manufacturers and open source communities offer alternatives, some of which we investigate later in the book.

**Linking an evaluation board**

This book focuses on the Keil STM32F400 evaluation board that features a STM32F407IGHx STMicroelectronics part to illustrate practical work. A wide range of other evaluation boards are available, and many of these are supported by the uVision5 IDE (that is, using the pack installer to download appropriate software components).
How to do it...

1. Once we have installed uVision, linking the evaluation board is simply a matter of connecting the two USB cables shown in the following image to your PC. The small daughter board shown in the image is Keil's ULINK-ME debug adaptor (http://www.keil.com/ulinkme/) that provides the data connection. The Windows plug-and-play feature will automatically find and install the driver (downloaded with uVision5).

2. The second USB cable provides power. Evaluation boards can usually be powered by a laptop or PC host connected via the USB port, but some laptop PSUs may be unable to supply sufficient current, and a USB hub might be required. Alternatively, an external supply can be connected via a separate power plug.

The first time the ULINK device is used, its firmware needs to be configured. The configuration depends on the MDK version, and if we wish to use different versions of the MDK (that is, perhaps because we have legacy code developed using uVision4) then the ULINK configuration may need to be erased. http://www.keil.com/support/docs/3632.htm provides some further information and a download utility for this purpose.
How it works...

A USB-Link adaptor is needed to enable the executable code produced by the IDE to be uploaded to the evaluation board. The adaptor supports a Joint Test Action Group (JTAG) interface on the evaluation board, and offers a number of debugging possibilities (depending on the type of adaptor used). There are several debug adaptor connection options. Firstly, the Keil ULINK-ME debug adaptor (http://www.keil.com/ulinkme/), packaged together with the board as a starter kit, connects to the 20-pin JTAG connector and supports serial wire programming and on-chip debugging. Keil's ULINK-2 adaptor (http://www.keil.com/ulink2/) represents a more robust solution with similar functionality, and ULINK-Pro (http://www.keil.com/ulinkpro/) offers extended debug facilities employing high-speed streaming trace technology.

There's more...

The MCBSTM32F400 (http://www.keil.com/mcbstm32f400/) evaluation board shown in the preceding image features the STMicroelectronics STM32F407IGHx microcontroller part. The board specification includes the following:

- STM32F407IG Microcontroller
- On-chip and external memory
- 2.4 inch QVGA TFT LCD and touchscreen
- USB 2.0 Ports
- CAN interface
- Serial/UART Port
- Micro SD Card Interface
- 5-position Joystick
- 3-axis accelerometer
- 3-axis Gyroscope
- ADC Potentiometer input
- Audio Codec with Speaker and Microphone
- Digital Microphone
- Digital VGA Camera
- Push Buttons and LEDs directly connected to I/O ports
- Debug Interface

MCU manufacturers like Texas Instruments (TI), STMicroelectronics, Freescale, Atmel, Analog Devices, Silicon Labs, MikroElektronika, NXP, and Nordic Semiconductor all market evaluation boards featuring the Cortex-M4. Some of these offer cheaper, entry-level board options costing just a few dollars with functionality that can be enhanced by adding additional modules.
An insight into the range of microcontroller devices supported by MDK-ARM can be gained by scrolling through the list of packs listed by the Pack Installer. Keil markets a range of Cortex-M evaluation boards designed by themselves and other manufacturers (http://www.keil.com/boards/cortexm.asp) that feature a number of microcontrollers. Keil's range of boards features NXP, STMicroelectronics, and Freescale microcontrollers. The MCBSTM32 (Cortex-M3) and MCBSTM32F400 (Cortex-M4) evaluation boards offer one of the more expensive evaluation routes, but they are populated with a comprehensive set of I/O peripherals, including a QVGA TFT LCD touchscreen. STM (http://www.st.com) markets a similar evaluation board called the STM3241G-EVAL, offering almost identical features to Keil's but employing a slightly different PCB layout and using the STM32F417IG part.

Netduino (http://netduino.com/) offers a series of open source evaluation boards based on the STM32F405RG microcontroller featuring a Cortex-M4 core with open source software development support. Netduino is supported by an enthusiastic community of developers—a selection of projects which demonstrate the potential of the device are available.


You will also find that the schematic diagram of the evaluation board, at http://www.keil.com/mcbstm32f400/mcbstm32f400-schematics.pdf, is also useful for resolving ambiguities in the libraries. If you use MDK-ARM, then once a new project has been created and the target microcontroller identified, most of the relevant documentation can be accessed via the Books tab within the project window.
Running an example program

Manufacturers usually make a small number of example programs available that provide a tutorial introduction and demonstrate the potential of their evaluation boards. A simple program that flashes (that is, blinks) a Light-emitting diode (LED) on the board is usually provided. ANSI C is by far the most popular language amongst embedded system programmers, but other high level languages such as C++ and C# may also be supported. A brief introduction to the C programming language is provided in Chapter 2, C Language Programming.

The Examples tab in the pack installer for the STM32F4 series MCUs provides a link to a C program called CMSIS-RTOS Blinky (MCBSTM32F400) that flashes an LED connected to a GPIO port. The program is integrated within an MDK-ARM Project. Integrated development environments such as MDK-ARM usually manage software development tasks as projects, as in addition to the program source code itself, there are other target-specific details that are needed when the code is compiled. A project provides a good container for such things. We review the steps required to create a project from scratch in the next section.

How to do it...

1. Invoke uVision5. Open the Pack Installer, and copy the example program to a new folder (name the folder CMSIS-RTOS_Blinky).
2. Connect the evaluation board as described in the previous section. In addition to the ULINK cable, remember to connect a USB cable to supply power to your evaluation board.
3. Invoke uVision5 from the taskbar, select Project → Open Project; navigate to the folder named CMSIS-RTOS_Blinky, and open the file named blinky.uvprojx.
4. Build the project by selecting Project → Rebuild all target files, and then download the executable code to the board using Flash → Download. Take a moment to locate the Build, Rebuild, and Download shortcut icons on the toolbar as these save time.
5. Finally, press the RESET button on the evaluation board, and confirm that Blinky is running. You may notice that the Blinky example program does a little more than just flash one LED.
6. Once you have confirmed that your evaluation board is working, close the project (Project → Close Project), and quit uVision5.

How it works...

The program uses some advanced concepts such as CMSIS-RTOS (discussed in Chapter 8, Real-Time Embedded Systems.) to produce a visually interesting flashing LED pattern. We will not attempt to explain the code here, but the next section will develop a much simpler Blinky project called hello_blinky.uvprojx.
Writing a simple program

This section explains how to write, build, and execute a simple program. We also describe the various files that, together, make up a uVision project.

How to do it...

1. Use Windows Explorer to create a new (empty) folder called helloBlinky_c1v0. Invoke uVision5, and create a new project (Project → New uVision Project...). Navigate to the folder, and create a project file called hello_blinky.uvprojx. When prompted, choose the STM32F407IGHx device. Click OK.
2. In Manage → Run Time Environment, choose the MCB32F400 board support using the drop-down list, and tick the LED API (since our application will flash an LED). Expand the Device option list, and tick Startup and Classic.

![Screenshot of ARM Cortex settings](image)

3. Notice that the Validation Output pane display warns us that, to drive LEDs, we also need CMSIS core, GPIO driver, and system start-up components. Press the Resolve button to automatically include any libraries needed by the board features selected, then click OK. The project window in uVision5 should show that the files have been successfully loaded. The names of the folders can be changed using a right-click menu, and fields can be expanded to show individual components, thereby allowing the file components to be edited. Note: Some library files are read-only.
4. Right-click **Source Group 1**, and select **Add New Item to Group 'Source Group 1'**; then select a C File (.c) template. Name the file `hello_Blinky.c`, and enter the following program:

```c
#include "stm32f4xx_hal.h"
#include "Board_LED.h"

int main (void) {
    const unsigned int num = 0;
    unsigned int i;

    LED_Initialize();    /* LED Initialization */

    for (; ;) {            /* Loop forever */
```

/*
 * Recipe: helloBlinky_c1v0
 * Name: hello_blinky.c
 * Purpose: Very Simple MCBSTM32F400 LED Flasher
 */

* Modification History
* 16.01.14 Created
* 27.11.15 Updated
* (uVision5 v5.17STM32F4xx_DFP2.6.0)
*
* Dr Mark Fisher, CMP, UEA, Norwich, UK
*/

#include "stm32f4xx_hal.h"
#include "Board_LED.h"
```
LED_On (num);        /* Turn specified LED on */
for (i = 0; i < 10000000; i++)
    /* empty statement */ ;       /* Wait */
LED_Off (num);     /* Turn specified LED off */
for (i = 0; i < 10000000; i++)
    /* empty statement */ ;       /* Wait */
} /* end for */

5. The RTE manager of uVision5 will have configured the device options with values from the device database, but the debug options should be reviewed by selecting Project → Options for Target 'MCBSTMF400'... to ensure that they specify the ULINK2/ME Cortex Debugger.
6. Build the project by selecting **Project → Rebuild all Target Files.** Again, there is a toolbar icon that provides a helpful shortcut.

7. Write the executable code to the microcontroller’s flash memory using **Flash → Download.** Press the **RESET** button on the evaluation board to run the program.

**How it works…**

Those familiar with uVision4 will notice that the most obvious feature for of this program is that a call to `SystemInit()` is missing, as this code is executed before `main()` is called. The function called `main()` is the entry point for our program, and each project should declare only one file that defines a main function. Conventionally, this might be called `main.c`, or adopt a file name that is shared by the project such as `helloBlinky.c`.

Most of the file `helloBlinky.c` comprises comments, which are highlighted in green. Comments do not produce any executable code, but they are essential for understanding the program. You may be tempted to omit comments, but you will appreciate their value if, at some later date, you need to reuse code written by others, or even yourself.

The source code file begins with a large comment statement that extends over several lines and contains information about the program. Then there are C pre-processor directives; we discuss these in **Chapter 2, C Language Programming.** The program comprises a main function that declares two variables named `i` and `num`. There follows a function call to `LED_Initialize()` (written by developers) that sets up the GPIO peripheral which drives the LEDs. The program contains three so called `for` loops. The outer loop, is known as a superloop and never terminates. These statements within this loop are executed again and again, forever (well for as long as power is supplied to the evaluation board). The statements within the loop turn the specified LED ON and OFF by calling yet another function written by Keil developers. The other two `for` loops, nested within the superloop, simply waste time by incrementing the loop variable `i`. Implementing a delay in this way represents a very naïve approach, and we'll explore much more efficient techniques later. If you have not programmed in C before, then although you'll probably appreciate that this program is very compact, you may find it confusing. Don't worry, we'll revisit this program again when we introduce the C programming language in **Chapter 2, C Language Programming.**
There's more...

The structure of the uVision MDK projects has evolved considerably over the past few years and uVision5 represents a significant revision in this respect. Developers of uVision5 have attempted to make microcontroller software development much simpler by providing library functions that can be used to control peripherals such as LEDs, accelerometers, touchscreen, and so on. Many application developers migrating from uVision4 find this burdensome, and favor more classic approaches that do not rely on intrinsic interface functions. Application programmers who wish to use their own middleware functions are advised to download the ARMs MDK legacy support pack (http://www2.keil.com/mdk5/legacy). The source files that, together with the project options, define the helloBlinky project are summarized in the following table:

<table>
<thead>
<tr>
<th>File Type</th>
<th>File extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C File</td>
<td>.c</td>
<td>Source code written in ANSI C.</td>
</tr>
<tr>
<td>Header File</td>
<td>.h</td>
<td>File containing additional information to be included in the source code</td>
</tr>
<tr>
<td>Assembly Language File</td>
<td>.s</td>
<td>Source code written in ARMs Thumb2 assembly language (Cortex-M cores)</td>
</tr>
<tr>
<td>Text File</td>
<td>.txt</td>
<td>Text file, usually containing description of the project or instructions for running the code.</td>
</tr>
</tbody>
</table>

A configuration wizard is provided to customize some files (for example, startup_stm32F40xx.s). However, we will deal with these more advanced aspects in subsequent chapters. Further, library and header file components, declared within the source files themselves, are also listed in the project window, and can be opened in the editor window. The file types you will encounter are described briefly in the following table, but will be discussed in more detail in Chapter 2, C Language Programming.

<table>
<thead>
<tr>
<th>File Type</th>
<th>File extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C File</td>
<td>.c</td>
<td>Source code written in ANSI C.</td>
</tr>
<tr>
<td>Header File</td>
<td>.h</td>
<td>File containing additional information to be included in the source code</td>
</tr>
<tr>
<td>Assembly Language File</td>
<td>.s</td>
<td>Source code written in ARMs Thumb2 assembly language (Cortex-M cores)</td>
</tr>
<tr>
<td>Text File</td>
<td>.txt</td>
<td>Text file, usually containing description of the project or instructions for running the code.</td>
</tr>
</tbody>
</table>
The project options are functionally grouped together. They are accessed through the tabs within the **Project Options** menu, and summarized in the following table. Further details are available in the uVision User Guide.

<table>
<thead>
<tr>
<th>Tab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>Select the microcontroller device from the database</td>
</tr>
<tr>
<td>Target</td>
<td>Specify hardware parameters</td>
</tr>
<tr>
<td>Output</td>
<td>Define output files of the tool chain</td>
</tr>
<tr>
<td>Listing</td>
<td>Specify all listing files generated by the tool chain</td>
</tr>
<tr>
<td>User</td>
<td>Specify user programs executed before compilation / build</td>
</tr>
<tr>
<td>C/C++</td>
<td>Set C / C++ compiler-specific tool options</td>
</tr>
<tr>
<td>Asm</td>
<td>Set assembler-specific tool options such as macro processing</td>
</tr>
<tr>
<td>Linker</td>
<td>Set linker-related options, and define physical memory parameters.</td>
</tr>
<tr>
<td>Debug</td>
<td>Specify settings for the uVision debugger</td>
</tr>
<tr>
<td>Utilities</td>
<td>Configure utilities for flash programming</td>
</tr>
</tbody>
</table>

The options allow the developer to control quite small details of the build—for example, you might find it more convenient to execute code as soon as it is downloaded to the target by configuring the flash programming settings using the utilities tab as shown in the following image:

The STM32F400IGHx microcontroller implements 1MB On-chip Flash memory. **RAM for Algorithm** defines the address space used by the programming algorithm for the device.
Understanding the simple use of GPIO

Making an LED blink involves connecting it to a signal that alternately switches ON and OFF. **General purpose input/output (GPIO)** is the name of a microcontroller peripheral that provides functionality to source many signals at once (that is, in parallel). GPIO peripherals are designed to be very flexible, so configuring them can be rather confusing but using the RTE manager makes this process much simpler. We will modify our helloBlinky_c1v0 recipe to simultaneously make all the LEDs blink rather than just one. Each LED on the evaluation board is connected to a pin on the microcontroller, so to illuminate an LED the microcontroller needs to provide a voltage and current similar to that of a torch battery. To source this current, the corresponding GPIO port bit connected to the pin must be configured as an output that is switched ON and OFF by statements in our program that write to the port output data register.

**How to do it...**

To configure the GPIO follow the steps outlined:

1. Make a copy of the helloBlinky_c1v0 folder from the previous recipe (and its contents) and rename this copy as helloBlinky_c1v1. Open the folder and open the helloBlinky project (double-click on the file). Then edit the main function defined in the helloBlinky.c file search for the following statement:
   ```c
   LED_On (num);
   ```
2. Replace this statement with the following one:
   ```c
   LED_SetOut (On_Code);
   ```
3. Also, search for the following statement:
   ```c
   LED_Off (num);
   ```
4. Replace this statement with the following one:
   ```c
   LED_SetOut (Off_Code);
   ```
5. The variables, On_code and Off_Code, are declared, as follows:
   ```c
   const unsigned intOff_Code = 0x0000;
   const unsigned intOn_Code = 0x00FF;
   ```
6. A complete listing of the main function is as follows:

```c
/*--------------------------------------------------
* Recipe: helloBlinky_c1v1
* Name: helloBlinky.c
* Purpose: Simultaneous MCBSTM32F400 LED Flasher
*--------------------------------------------------
* Modification History
* 16.01.14 Created
* 03.12.15 Updated
* (uVision5v5.17+STM32F4xx_DFP2.6.0)
* 
* Dr Mark Fisher, CMP, UEA, Norwich, UK
*--------------------------------------------------*/
#include "stm32F4xx_hal.h"
#include "Board_LED.h"

int main (void) {
    const unsigned intOff_Code = 0x0000;
    const unsigned intOn_Code = 0x00FF;
    unsigned inti;

    LED_Initialize();                      /* LED Init */

    for (;;) {                         /* Loop forever */
        LED_SetOut (On_Code);            /* Turn LEDs on */
        for (i = 0; i< 1000000; i++)       /* Wait */
            /* empty statement */ ;
        LED_SetOut (Off_Code);          /* Turn LEDs off */
        for (i = 0; i< 1000000; i++)       /* Wait */
            /* empty statement */ ;
    } /* end for */
}
```

7. Build, download, and run the application in exactly the same way as we did in the previous version.
How it works...

The GPIO interface is a particularly important feature in microcontrollers because it is designed to be easily integrated within user systems to drive light emitting diodes, read the state of switches, or connect to other peripheral interface circuits. Early I/O ports were prewired to provide either output or input interfaces, but soon they evolved into general purpose interfaces that could be programmed to provide either output or input connections. Later devices included more programmable features. As GPIO is so important for microcontroller applications, designers are keen to specify as many I/O pins as possible on their devices. However, increasing the device pin-out adds cost because the device becomes physically larger to accommodate the pins. This motivates manufacturers to develop devices that have pins that are configured by software. As you can imagine, configuring such a device is quite a challenge, so we're lucky that Keil's developers have provided library functions that make this task more manageable. As GPIO represents the interface between hardware and software, the evaluation board's schematic ([http://www.keil.com/mcbstm32f400/mcbstm32f400-schematics.pdf](http://www.keil.com/mcbstm32f400/mcbstm32f400-schematics.pdf)) is essential to understanding the I/O.

The STM microcontroller used by the evaluation board provides eight GPIO ports, named A-I. Port pins PG6,7,8; PH2,3,6,7; PI10 are connected to LEDs. Those who have never encountered an LED may imagine it as a filament lamp, but an LED is a semiconductor device and behaves slightly differently. However, sticking with our initial lamp analogy (for the time being), we'll first consider a battery-operated torch comprising a battery, switch, and lamp. These components are connected by a copper wire that is often hidden within the body of the torch. We'll assume that the torch uses two AA batteries providing a voltage of about 3 Volts. We can depict the circuit as a diagram with symbols representing each of the components, as shown in the following diagram:

When we close the switch, the battery voltage (denoted V) is applied directly to the lamp, a current flows (denoted I), heating the lamp filament, and this in turn, gives out light.
The electrical resistance (denoted R) of the filament determines the amount of current that flows according to Ohm's Law that is as follows:

\[ I = \frac{V}{R} \]

Lamp filaments used in torches usually have a resistance of about 10 Ohms (10 Ω), so the amount of current flowing is about 0.3 A or 300 mA.

Imagine that a fault develops, which produces a short across the lamp. The current flowing is now only limited by the resistance of the copper wire and the internal resistance of the battery; these are both very small (a fraction of an Ohm). A high current will circulate which might, if the battery stored enough energy, cause the copper wire to heat up and melt the plastic case of the torch. However, AA batteries are unable to store sufficient energy for this to be a serious problem and in most cases the battery will discharge within a few seconds.

In modern torches, the lamp is replaced by an LED, which is a semiconductor device (its electrical properties lie between those of conductors, such as copper, and insulators, such as glass). An LED is a two terminal device with special properties. One of the terminals is known as the anode and the other as the cathode. If we replace the lamp in our torch with an LED, then current will only flow and the LED will illuminate when the anode is connected to the positive-battery terminal and the cathode to the negative-battery terminal, as depicted in the following diagram:

![Diagram of LED connection](image)

If we connect the device the other way round as depicted in the right side of the preceding diagram, then no current will flow; so, make sure that the batteries in your LED torch are fitted the right way round! When the anode is connected to the positive-battery terminal, the diode resistance is very low and the diode is said to be forward biased. When the cathode is connected to the positive-battery terminal the diode exhibits an extremely high resistance (negligible current flow) and the diode is said to be reversed biased. When forward biased, the LED exhibits an extremely low resistance, so an additional resistor must be placed in the circuit to limit the current flowing.
GPIO can also be used to read the state of switches that are connected to microcontroller pins. For this operation, each port bit must be configured as an input. When configured for input (that is, output is disabled), each bit of the parallel port's input data register is connected to a pin on the integrated circuit (on which the embedded processor is fabricated). Let's assume that we wish to connect a simple push-button switch to an input bit such that when the switch is operated, a voltage is applied to the port (pin), otherwise, no voltage is applied. The circuit a) shown as follows will achieve this. A complementary circuit that produces a voltage when the switch is open, and no voltage when the switch is operated (closed) is shown in b):
To eliminate the need for an additional resistor, the GPIO port input circuit includes one that can be configured by software as pull-up, pull-down, or disconnected. Obviously, when the port is configured as an output, both resistors are disconnected.

There's more...

Section 7 of STMicroelectronics Reference manual RM0090 (www.st.com) for microcontrollers featuring the Cortex-M4 provides comprehensive programming details for the GPIO port. As well as producing logic signals (for example, making LEDs blink) and reading logic levels (for example, from switches), GPIO ports also provide an I/O path for other peripheral functions, such as Times and Digital-to-Analogue converters. We'll take a closer look at GPIO later on in this cookbook when we write programs that include more functionality.

Estimating microcontroller performance

The millions of instructions that can be executed per second (MIPS) is one measure of processor performance. This figure depends on the processor architecture, the clock speed, the memory performance, and so on. The microcontroller can be clocked from one of three oscillator sources. A high speed external (HSE) clock is derived from a 25 MHz crystal oscillator connected between two pins of the microcontroller. A high speed internal (HSI) clock is sourced from an internal 16 MHz resistor-capacitor (RC) controlled oscillator, and a Phase Locked Loop (PLL) can be configured to provide multiples of either HSE or HSI.

A peripheral called reset and clock control (RCC) allows the clock source to be selected and configured using a circuit known as a clock tree. The RCC peripheral also sources clocks for other microcontroller peripherals, and these also need to be configured. Following a hard reset, the RCC configuration is determined by the RCC register default values given in the RM0090 Reference Manual (www.st.com). Selecting Startup from the Device submenu of the RTE manager copies an assembly language file named startup_stm32f407xx.s (the .s file extension is conventionally used to identify assembly language files) to our project. This file holds the exception table. The reset exception generated by a hard reset (that is, activating the reset button on the evaluation board) causes the microcontroller's program counter to be loaded with the address of the reset handler (identified by symbol Reset_Handler), and this in turn calls a function named SystemInit() defined in the file, system_stm32f4xx.c. This function configures the RCC to use the 16 MHz HSI clock before calling the function main().
How to do it...

1. Run `helloBlinky`, and measure the frequency of the ‘blinks’. We should see about 4 blinks/second or 4 Hz. It may be easier to count the blinks in a 10-second period.

2. When we examine the program code shown earlier, we see that the program spends most of its time executing the two nested `for` loops. The statements inside these loops are executed thousands of times. Some readers may have spotted that there are no statements called inside the loop; but even so, the loop counter must be updated on each iteration. This operation requires an addition (ADD) instruction followed by a compare (`CMP`) instruction to be executed.

3. We need to do some elementary math to work out how long it will take to execute these two instructions. Checking Table 3.1 of the *ARM Cortex-M4 Processor Technical Reference Manual*, we see that these each take 1 cycle to execute. Since `SystemInit()` configures the RCC to use the HSI (16 MHz) clock, the time needed to switch the LED ON/OFF once will be 

\[
2 \times (1000000) \times \frac{1}{16 \times 10^6} \times 2 \text{ (instructions)} = 250 \text{ ms}
\]

(that is, about 4 times per second).

There's more...

To understand how the processor achieves this level of performance, we need to look at the processor architecture. The processor implements the ARMv7-M architecture profile described at [http://infocenter.arm.com](http://infocenter.arm.com). ARMv7-M is a 32-bit architecture and the internal registers and data path are all 32-bit wide. ARMv7-M supports the Thumb Instruction Set Architecture (ISA) with Thumb-2 technology that includes both 16 and 32-bit instructions. ARM processors were originally inspired by *Reduced Instruction Set Computing (RISC)* architectures developed in the 1980s. RISC architecture attempted to improve on the performance of traditional computer architectures of the era that employed the so-called *Complex Instruction Set Computing (CISC)* architectures, by defining an ISA that supported a small number of instructions, each of which could be executed in one processor clock cycle, and so achieve a performance advantage. In the three decades since RISC was proposed, the size and complexity of RISC ISA’s has increased, but the goal is still to minimize the number of clock cycles needed to execute each instruction. With this in mind, ARM Cortex-M3 and M4 processors have a three-stage instruction pipeline and Harvard bus architecture. Computers that use Harvard architecture have separate memories and busses for instructions and data rather than the shared memory systems used by von Neumann architectures, and the higher memory bandwidth this affords can achieve better performance.
The Cortex-M4 processor also provides signal processing support including a **Single Instruction Multiple Data (SIMD)** array processor and a fast **Multiply Accumulator (MAC)**. Together with an optional **Floating Point Unit (FPU)**, these features allow the Cortex-M4 to achieve much higher performance in **Digital Signal Processing (DSP)** applications than the earlier Cortex-M3.

**See also**

Besides manufacturers' data sheets, there are a few books that address the Cortex-M4. Joseph Yiu's books (http://store.elsevier.com/Newnes/IMP_73/) on the Cortex-M3 and M4 processors are aimed at programmers, embedded product designers, and System-on-Chip (SoC) engineers. Books for undergraduate courses include a series of books by Jonathan Valvano (http://users.ece.utexas.edu/~valvano) and a text written by Daniel Lewis (http://catalogue.pearsoned.co.uk). Trevor Martin has also written an excellent guide to STM32 microcontrollers. This document is one of a number of insider guides that can be downloaded from http://www.hitex.com.
Where to buy this book

You can buy ARM® Cortex® M4 Cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.