Embedded Systems Fundamentals with ARM Cortex-M based Microcontrollers: A Practical Approach
Alexander G. Dean, ARM Education Media, 2017, ISBN 978-1-911531-03-6

Preface

Introduction

It is an exciting time to develop embedded systems! Modern microcontrollers (MCUs) offer remarkable performance at a very low cost. The Internet provides an abundance of source code and documentation. The combination of inexpensive hardware platforms (e.g., Arduino, Raspberry Pi, and Beaglebone) with the right software (to abstract away details and guide users) has helped lower the barriers to embedded system development, allowing experimentation without requiring encyclopedic knowledge.

Unfortunately, these supports become shackles when trying to scale up to a larger, more complex system with tighter constraints. Industrial designers of embedded systems draw from a large toolbox of technical methods in order to meet requirements such as speed, responsiveness, cost, weight, reliability, or energy use.

Many of the hardware tools are built into the MCU: a central processing unit (CPU) to execute software, an efficient interrupt system enabling quick responses to events, fast memory to hold the program and data, and specialized hardware peripheral circuits to reduce the need for a high-speed CPU. Hardware peripherals can often signal and control each other, eliminating the need to involve software on the CPU. MCUs offer a range of low-power modes so the designer can trade off performance and power consumption as needed.

Other tools are provided by the software, which is typically written in C or C++ and compiled to run in the processor's native machine language. This avoids the run-time delays and memory overhead of interpretation or scripting. Multitasking software is scheduled on the CPU using interrupts and a scheduler, which may be cooperative (e.g., state machines) or preemptive (e.g., a real-time kernel).

To summarize, successful embedded system designs use peripherals and well-structured software on the CPU with light-weight context switching to provide responsive concurrency. This textbook aims to explain how to develop microcontroller-based embedded systems using these industry-standard methods and practice these with the most widely used processor architecture in embedded computing today: the ARM Architecture.

Challenges of Embedded Systems Education

There are several interesting challenges to learning (or teaching!) embedded systems in a college or university. First, the field builds on concepts from several areas: computer engineering (CPE), electrical engineering (EE), and computer science (CS). Some students will be able to study joint degrees, or even double or triple major, but most will not. Second, these concepts and their solutions must target embedded system design spaces, which are quite different from the mainstream general-purpose or high-performance computing design spaces covered by most courses. Third, there are so many areas to cover that it is easy to concentrate on the familiar, which crowds out the unfamiliar. Presenting the areas with just enough detail (but not too much) can be difficult.

Challenge 1: Spanning Electrical and Computer Engineering and Computer Science

Successful embedded system designers need a variety of skills from CPE, EE, and CS, but not too much, and the right version given the context. These skills are typically split across ECE (electrical and computer engineering) and CS departments. To make things worse, CPE and CS courses are constantly pulled toward higher performance computers and higher levels of abstraction (to manage the increased application complexity enabled by the increased performance). This widens the gap with the embedded system design space.

The following areas in CPE and EE are the most critical for students in the field of embedded systems:

- Computer organization and assembly language programming are fundamental to an understanding of the CPU, memory, peripherals, and interrupt system.
- Digital design is necessary to understand not only how the CPU works, but more importantly to understand how peripheral circuits work. These digital circuits (e.g., GPIO, timers, DMA) provide cheap concurrency because they operate independently of the CPU. A good design will offload computationally expensive software tasks to allow a relatively slow MCU to provide precise timing and predictable performance at a low cost and with little power consumption.
- Basic analog circuit design and analysis skills are needed for adding external circuits such as LEDs, switches, and sensors. Knowing how to use an oscilloscope or logic analyzer to examine and understand the timing of events within a system is essential for effective debugging.

The following areas in CS are the most critical for students in the field:

- C language programming is necessary because it is the dominant language for programming embedded systems.
- Compilers and assembly language programming provide an understanding of how a CPU really does the work specified by the source code. Knowing how the program is compiled and structured helps with avoiding errors (e.g., preemption), debugging, designing efficient systems, and improving performance.
- Operating systems' task scheduler concepts enable students to understand how to share a single CPU among the multiple concurrent activities of the embedded system. These topics include multitasking, preemption, and prioritization. Students need to understand how to design multitasking systems using intertask communication and synchronization to avoid common bugs such as data race hazards.

Challenge 2: Targeting the ES Design Space

For each area mentioned, the practical solutions depend on the design space. The design spaces for most embedded systems are quite different from those of general purpose and high-performance computing, because of different drivers and constraints. For example:

- Computer Organization: Embedded processors typically do not need the raw speed sought in general-purpose or high-performance computing systems. As a result, they don't require high clock speeds and the deep processor pipelines and multilayer memory systems to support them.
- Operating Systems: OS courses generally target a resource-rich Linux system that features a preemptive scheduler, ample compute and memory resources, a virtual memory system with hardware support, and user and supervisor modes. This type of system does not offer the precise timing control needed for many embedded systems and is often too complex, power-hungry, and expensive. Students must be able to apply the concepts of task scheduling, synchronization, and communication to a system built on interrupts, peripherals, and a simple scheduler (whether a preemptive real-time kernel or a cooperative scheduler).
- Programming: Embedded systems use compiled languages instead of scripted or interpreted languages for reasons of predictability, efficiency, and compactness. Because of this history there is a large installed base of C/C++ development infrastructure. However, many programming curricula target Java (or even a scripted language). This abstracts away low-level and implementation issues that can make or break an embedded system.

Challenge 3: Providing Sufficient (but Not Excessive) Coverage

With all of these areas to cover, it is easy to emphasize the familiar, crowding out the unfamiliar. Furthermore, the hands-on nature of embedded systems courses often slows down the progress as the student or instructor tries to get a code example working to demonstrate an important concept.

This book tries to present the areas with just enough detail (but not too much) and with practical solutions for the design space. This book does not try to present an exhaustive, complete education of all possible ways to do something. Instead, it presents the most practical options given the constraints.

Notes to the Instructor

Why Use This Book?

In this textbook, I have sought to present the most important topics for embedded systems using a coherent, compelling, hands-on format.

First, the textbook uses a hands-on approach to get students excited and motivated. Each chapter has illustrated, working examples based on a real MCU evaluation board. These activities start early, with Chapter 2 showing how to read switches and light LEDs using GPIO and C code.

Second, the textbook introduces concepts of concurrency and responsiveness early. Chapter 3 uses a running example of scanning LEDs according to switch positions to introduce concepts important for creating modular, responsive, and efficient systems. By stepping through and evaluating these improvements, the student is given a solid foundation on which to investigate real-time kernels (in a later course). Concurrency and responsiveness are introduced using the following sequence:

1. Starting with a simple program with software to poll switches, flash LEDs, and delay using busy-waiting
2. Restructuring the software into tasks
3. Scheduling the tasks cooperatively
4. Improving the responsiveness of cooperatively scheduled tasks by using state machines to break up long operations
5. Using interrupts and event-driven software to replace polling of switches
6. Replacing busy-waiting delay loops by using a timer peripheral
7. Prioritizing tasks
8. Scheduling tasks preemptively

Third, the textbook covers how to improve performance by using peripheral hardware in place of software. An analog waveform generator is used as a running example. It is introduced as an application of the digital-to-analog converter, with timing fully dependent on software execution speed. It reappears in the timer chapter, with a timer-driven periodic ISR updating the DAC to improve timing stability. The final appearance is in the DMA chapter, in which the DMA controller under timer control automatically copies data from a memory buffer to the DAC.

Fourth, the textbook covers C code as implemented in assembly language by the compiler. The main goals are to help students understand why their code is slow or large, how to make it faster or smaller, to understand preemption risks for shared data, and to help debug programs by working at both the source and object code levels. This textbook does not expect students to program in assembly language, although they may do so in a later course, given this foundation.

Course Material Linkage

This textbook is designed to be used for a one- or two-semester course introducing students to embedded systems. It complements the Efficient Embedded Systems Design and Programming Education Kit from the ARM University Program. If you are an instructor, you can receive a donation of this Education Kit by emailing university@arm.com. The Education Kit includes lecture materials and licenses to ARM's Keil MDK-ARM professional software. Students need prerequisite knowledge in C programming, digital design, and basic circuit theory.

Target Platform

This textbook targets the ARM Cortex-M0+ processor, which executes the instructions of the program. The processor is a component within the microcontroller, which adds circuits to clock the processor, memory to hold the program and data, and peripheral devices that simplify programs and improve their performance. This processor is available in microcontrollers from a wide range of manufacturers.

The target platform is the FRDM-KL25Z development board from NXP Semiconductor, with a list price of under $20. It uses the NXP KL25Z128VLK4 microcontroller from the Kinetis L ultra-low-power family. This device features a Cortex-M0+ processor capable of running at up to 48 MHz, and contains 128 kB of flash ROM, 16 kB of RAM, and a wide range of peripherals. The development board adds a USB debug interface (OpenSDA), power supplies, and input and output devices. A three-axis accelerometer is used to detect acceleration. Because it also senses the force of gravity, it can be used to determine the inclination (tilt) of the board. A touch-pad slider can measure the position of a fingertip using a capacitive sensor. A three-in-one output device is included: three high-brightness LEDs (red, green, and blue). These LEDs can be lit with varying levels of brightness to produce a full range of colors.

The material in this textbook can be used with other Cortex-M0+ platforms. Four of the first five chapters are essentially independent of the MCU's peripherals and apply to all Cortex-M0+ processors. The remaining chapters and the Appendix are closely integrated with the peripherals by necessity. NXP's other Kinetis MCUs use many of the same peripherals as the KL25Z, making it easier to use those MCUs and their associated FRDM evaluation boards. Targeting an MCU family from a different vendor will require porting the peripheral examples.

Software Development Environment

Software examples in this textbook are written in C and compiled to run without an operating system. ARM's Keil MDK-ARM integrated development environment is used throughout the textbook. The free version of MDK-ARM supports all of the code examples in this textbook and associated course materials. Note for instructors: If the object code size limitation of the free version (currently 32 KB) is a constraint, please request a license donation from ARM for the full professional version of MDK-ARM.

Organization

The textbook is organized as follows:

Chapter 1 introduces students to the concepts of MCU-based embedded systems, and how they differ from general-purpose computers. It then introduces the ARM Cortex-M0+ CPU, the Kinetis KL25Z MCU, and the FRDM-KL25Z MCU development board.

Chapter 2 presents the general purpose I/O peripheral to provide an early, hands-on experience with reading switches and lighting LEDs using C code. It also introduces the CMSIS hardware abstraction layer, which simplifies software access to peripherals.

Chapter 3 introduces multitasking on the CPU, with the goals of improving responsiveness and software modularity while reducing CPU overhead. The interplay of interrupts, peripherals, and schedulers (both cooperative and preemptive) is examined.

Chapter 4 presents the ARM Cortex-M0+ processor core, including organization, registers, memory, and instruction set. It then discusses interrupts and exceptions, including CPU response and hardware configuration. Designing software for a system with interrupts is discussed, including program design (and partitioning work), interrupt configuration, writing handlers in C, and sharing data safely given preemption.

Chapter 5 first gives an overview of toolchain, which translates a program from C source code to executable object code. It then shows side by side the source code and the object code the toolchain has generated to implement it. Topics covered include functions, arguments, return values, activation records, exception handlers, control flow constructs for loops and selection, memory allocation and use, and accessing data in memory.

Chapter 6 presents analog interfacing, starting with theory and ending with practical implementations. Quantization and sampling are presented as a foundation for both digital-to-analog conversion and analog-to-digital conversion. The DAC, ADC, and analog comparator peripherals are presented and used.

Chapter 7 presents timer peripherals and their use for generating a periodic interrupt or a pulse-width modulated signal, or for measuring elapsed time or a signal's frequency. Watchdog timers, used to detect and reset an out-of-control program, are also discussed. The SysTick, PIT, TPM, and COP timers are examined.

Chapter 8 discusses serial communication, starting with the fundamentals of data serialization, framing, error detection, media access control, and addressing. Software queues are introduced to show how to buffer data between communication ISRs and other parts of the program. Three protocols and their supporting peripherals are investigated next: SPI, asynchronous serial (UART), and I^2C. UART communication is demonstrated using the FRDM-KL25Z's debug MCU as a serial port bridge over USB to the PC. I^2C communication is demonstrated using the FRDM-KL25Z's built-in 3-axis accelerometer with I^2C interface.

Chapter 9 introduces the direct memory access peripheral and its ability to transfer data autonomously, offloading work from the CPU and offering dramatically improved performance. Examples include using DMA for bulk data copying, and for DAC-based analog waveform generation with precise timing.

An Appendix covers how to measure the power and energy use on the FRDM-KL25Z board, including disconnecting the debug MCU to reduce power. Methods to measure energy consumption using an ultracapacitor are highlighted.