Introduction to Microcontrollers - Beginnings

Mike Silva ● August 20, 2013

Welcome to this Introduction to Microcontroller Programming tutorial series. If you are looking to learn the basics of embedded programming for microcontrollers (and a bit of embedded hardware design as well), I hope these tutorials will help you along that journey. These are my first postings here, and I am writing this tutorial series because over the years I have seen countless newbies asking the same questions and tripping over the same stumbling blocks, and I thought I might be able to come up with something useful in answering those questions, and in avoiding those tripping points.

Quick Links

Target Audience

This tutorial series is intended for students, hobbyists, programmers and hardware designers who want to learn the basics of microcontroller programming, or who want to fill in some gaps in their knowledge of such programming. This tutorial will not teach you programming in general, although it will discuss programming techniques of particular interest for microcontrollers. This tutorial will also not teach you hardware design, although it will illustrate hardware issues commonly faced in employing microcontrollers. What it will do, I hope, is to help a newcomer understand what a microcontroller (µC) is, what capabilities it will typically have, and how to use those capabilities. It will go from the beginnings – how to wire up and program a µC to blink and LED (the microcontroller version of a “Hello World” program), on through the various features and peripherals typically found on a µC such as interrupts, timers/counters, UART, SPI, I2C, ADC, DAC, PWM, watchdog, and so forth. It will also examine common topics such as debouncing inputs, filtering ADC values, driving character LCDs, and other similar chores.

Caveat Lector

Every rule has ~~one~~ many exceptions. This applies to just about everything you will read in this tutorial. If you read "X" here, don't think it means "X and only X, in every possible situation, with no exceptions or qualifications, now and forever." Microcontroller designers have come up with many

different, interesting and sometimes just wierd ways of doing things. And as an microcontroller user and programmer you too can come up with many different and interesting ways of doing things too. In my experience, given an N-step program, there are probably at least N-squared ways of writing that program. The goal of this tutorial is to try and give you a solid foundation for µC programming, not to be a comprehensive encyclopedia of the field. For every example program, I will try and write it in a simple and understandable fashion and let you discover your own clever tricks further down the road.

Another possible source of confusion is in terminology. Different manufacturers quite often use different terminology for the same or similar features, registers and configuration/status options. In this tutorial sometimes I will adopt the terminology used by one of the µC families used in the tutorial, and other times I will use non-specific terminology. I will use whatever seems to be suitable for each situation.

What is Embedded Programming?

Embedded programming is the term for the computer programming that lives in and operates the great many computer-controlled devices that surround us in our homes, cars, workplaces and communities. To be clear, all microcontroller programming is embedded programming, but not all embedded programming is microcontroller programming. A little more will be said about this further along. Sometimes the terms will be used interchangeably, but the focus of this tutorial series is always on microcontrollers.

For every desktop or notebook or tablet computer you have, you may have a dozen or more (perhaps a great deal more) microcontrollers quietly doing their embedded duty, and with these devices many people don't even realize they involve a tiny computer running a program. But there is, and it is, and those programs had to be written, and that's why the world needs embedded programming. Microcontrollers add intelligence to countless devices and systems, enabling those devices and systems to operate better, faster, more safely, more efficiently, more conveniently, more usefully, and in many cases allowing the very existence of devices and systems that could not be built otherwise. Spend some time looking around you and trying to recognize where µCs are working, and you will begin to get a sense of how ubiquitous they have become since their invention some 40+ years ago.

On top of all that, many people, myself included, find µC programming a particularly fascinating and rewarding branch of the programming tree, and we just *like* to program embedded systems. In ways very different from most desktop or mainframe programming, embedded programs make stuff *do* stuff, and to an embedded programmer, stuff doing stuff is endlessly cool.

What is an Embedded System?

There's no perfect answer to that question, since every answer will have some exceptions. However, for our purposes let us declare that an embedded system is one that uses one or more microcomputers (that is, small to very, very small computers), running custom dedicated programs and connected to specialized hardware, to perform a dedicated set of functions. This can be contrasted with a general-purpose computer such as the familiar desktop or notebook, which are not designed to run only one dedicated program with one specialized set of hardware. It's not a perfect definition, but it's a start.

Some examples of embedded systems are:

• Alarm / security system

• Automobile cruise control

• Heating / air conditioning thermostat

• Microwave oven

• Anti-skid braking controller

• Traffic light controller

• Vending machine

• Gas pump

• Handheld Sudoku game

• Irrigation system controller

• Singing wall fish (or this gift season's equivalent)

• Multicopter

• Oscilloscope

• Mars Rover

For the most part I have listed example embedded applications on the less-complex end of the spectrum, since this is after all a beginning tutorial. By the end of this tutorial series you should have a good general idea how most of these applications would be programmed, and in rough terms what kinds of I/O, timing, interrupt and communications hardware and functionality they would require.

There are a few things worth noticing about the above list. While many embedded systems use fairly traditional user input-output devices (keypads, displays), many others do not. Also, many embedded systems interact directly with human beings, but others do not (and we're still waiting to see if the Mars Rover will interact directly with any Martians).

What is different about Embedded Programming?

Embedded programs must work closely with the specialized components and custom circuitry that makes up the hardware. Unlike programming on top of a full-function operating system, where the hardware details are removed as much as possible from the programmer's notice and control, most embedded programming acts directly with and on the hardware. This includes not only the hardware of the CPU, but also the hardware which makes up all the peripherals (both on-chip and off-chip) of the system. Thus an embedded programmer must have a good knowledge of hardware, at least as it pertains to writing software that correctly interfaces with and manipulates that hardware. This knowledge will often extend to specifying key components of the hardware (microcontroller, memory devices, I/O devices, etc), and in smaller organizations will sometimes go as far as designing and laying out (as a printed circuit board) the hardware. An embedded programmer will also need to have a good understanding of debugging equipment such as multimeters, oscilloscopes, logic analysers and the like.

Another difference from general purpose computers is that most (but not all) embedded systems are quite limited as compared to the former. The microcomputers used in embedded systems may have program memory sizes of a few thousand to a few hundred thousand bytes rather than the

gigabytes in the desktop machine, and will typically have even less data (RAM) memory than program memory. Further, the CPU will often be smaller 8 and 16 bit devices as opposed to the 32 bit and larger devices found in a desktop (although small 32-bit microcontrollers are now under a dollar in moderate quantities, which is amazingly amazing). A smaller CPU word size means, among other things, that a program will require more instructions (and thus more clock cycles) than an equivalent program running on a CPU with a larger word size. And finally, the speed at which smaller microcontrollers run is much less than the speed at which a PC runs. Typical smaller microcontroller clock rates are between 1 and 200 MHz, not the GHz rates of PCs.

What are the differences between microcomputer, microprocessor and microcontroller?

A microprocessor is usually understood to be a single-chip central processing unit (CPU), with the CPU being the "brains" of a computer - the part of the computer that executes program instructions. A microcomputer is any computer built around a microprocessor, along with program and data memory, and I/O devices and other peripherals as needed. A microcontroller (often shortened to µC in this tutorial) is a single chip device which has built onto the chip not only a microprocessor but also on the same chip, nonvolatile program (ROM) and volatile data (RAM) memory, along with useful peripherals such as general-purpose I/O (GPIO), timers and serial communications channels. Thus it follows that all microcontrollers are microcomputers, but not all microcomputers use microcontrollers.

In smaller embedded systems it is most common to use microcontrollers rather than microprocessor-based designs since microcontrollers give the most compact design and the lowest hardware cost. Larger embedded systems, on the other hand, may use one or more microprocessors if a microcontroller of suitable speed and functionality cannot be found. This can extend to the use of industrial PCs and even more powerful hardware. It is also possible to include both microprocessors and microcontrollers in a complex embedded system. The only real rules are, use whatever device(s) fit the task, given the constraints on budget, availability, time, tools, etc.

It should also be pointed out that with most microcontrollers it is possible to add external memory and peripherals, should the on-board mix not take care of all the system needs. When it makes sense to add such external devices, as opposed to choosing a larger microcontroller with the needed resources on-board, is a choice that needs to be made on an individual design basis.

What is an N-bit CPU/microprocessor/microcontroller?

There is some discussion about what it means to call a device an N-bit processor, but it's fairly obvious in most cases. If the device can perform most of its data manipulation instructions on data words up to N bits in size, the device is an N-bit processor. By way of example, a device may have a full set of instructions that can operate on 8 bit data, along with a few instructions that operate on 16 bit data. That device should be considered an 8-bit design, even if the marketing department says otherwise and calls it a 16-bit chip.

By volume, 8-bit microcontrollers are the biggest segment of the embedded market. Many applications simply don’t need any more power, and never will. 16-bit devices are more powerful, but they are squeezed between the 8-bit devices on the low end and the 32-bit devices on the high end. 32-bit devices are at the high end of the embedded spectrum for all but the most complex or high-performance designs, but they are moving ever downward in price.

What microcontroller families are used in these tutorials?

To give a bit of an overview of the different flavors of microcontrollers available, this tutorial will be written around one 8-bit family (the Atmel AVR) and one 32-bit family (the ARM Cortex M3 architecture in the form of the STM32 family). These two families were chosen to give a fairly broad picture of the devices and approaches found in the world of microcontrollers. The first few software examples will be written in assembly language for each of these families, as well as in C. After that, examples will only be written in C.

What else is required for these tutorials?

While you could, I suppose, work through much of this tutorial using just a microcontroller simulator, I strongly recommend that you have either a microcontroller training/development board, or even just a bare µC chip, assorted components and a powered breadboard. In addition you will need a C compiler that targets your device, and optionally an assembler for your device. You should have no trouble finding a free assembler for your chip, and you should also be able to find a free C compiler, even if it is a reduced-functionality version of a commercial compiler. You will also need a method of downloading your programs into your µC. The details of this download process will depend intimately on the particular µC and board it is mounted on.

As far as test equipment goes, digital multimeters are really cheap, and there's no excuse not to have one. Places like Harbor Freight sometimes have them on sale for a few dollars. The other piece of equipment that any embedded engineer must have is a decent oscilloscope. Don't panic, a scope is not required for these tutorials. However, if you can get ahold of one, you will learn more and save yourself a fair amount of time in the bargain. USB scopes give good bang for the buck, as do some import scopes (or, of course, a working used scope). At the end of last year I treated myself to a beautiful Agilent scope with a huge (to me) screen, and every time I use it I'm glad I spent the money.

Regarding the microcontrollers used in these tutorials, here are the details of the hardware and I will be using for each of the processor families:

AVR

•Hardware: Atmel STK-500 board with ATmega8515 installed

•Tools: Atmel Studio 6 (free)

ARM Cortex M3

•Hardware: STM32VLDiscovery Board, mounted on a custom docking board

•Tools: Rowley Crossworks ($150 for personal license - suggest IAR Embedded Workbench Kickstart Edition for free toolset)

Which programming language?

This is a good time to talk a bit about the various programming languages that one can use to write embedded software. The two languages I will use in this tutorial are C and assembly language. The first thing I want to point out is that these are *not* the only two languages available to embedded programmers, and that in many cases other languages may be a better choice. That being said, both C and assembly language are useful not only for learning about µC programming, but also for actually doing productive µC programming. They are also ubiquitous in that no matter what microcontroller you choose, it will almost certainly have available both an assembler (for processing

assembly language source code) and a C compiler (for processing C source code). The same is definitely not the case for other languages. But I would encourage you to consider other languages if you are so inclined and, big IF, if they are available for your device family.

On the subject of assembly language, even if you don't plan on using assembly language in your embedded programming, I would strongly suggest that you become at least somewhat familiar with the concepts, and with the instruction set of your µC. The reason for this is that, even if you don't end up writing any assembly language (I hardly ever do any more), you will find yourself at some point needing to examine the output of your compiler and/or your compiler-supplied startup files written or output in assembly language.

Also note that the term "assembly language" will often be shortened, in this tutorial and elsewhere, to "asm" or "ASM."

How does an embedded program run?

Before talking much more about embedded programming, this is a good place to give a brief overview of how an embedded program starts up and runs. Assuming that you have generated a program file and have loaded it into the µC program memory (all steps that we will talk about in more detail later), the good stuff happens when you either turn on the device or you push the RESET button. When the µC comes out of reset from either action it will always go to a particular memory location, as defined by the manufacturer, to begin executing whatever code is found there, or pointed to there. Sometimes this memory location contains code directly; e.g. upon coming out of reset, program execution begins at program address 0. Other times the fixed memory location is a vector, a location that holds the actual address of the beginning of the program; e.g. upon coming out of reset, the controller will load its program counter with the value found at program address 0xFFFE and thus start executing code at the address found in locations 0xFFFE and 0xFFFF (assuming a 16-bit program address, stored in 2 bytes). In the first instance you will have to make sure that your program has loaded at the specified startup address, while in the second instance you will load your program wherever the program memory has been placed in the controller address space, and you will have to make sure that you then load that startup address into the reset address vector. Note that the choice of startup method is not up to you, but will be built in to the design of the µC you have chosen. AVR uses the first method, and Cortex M3 uses the second.

When an embedded program starts to run, there is usually a fair amount of initialization and housekeeping that must be done before the meat of the program begins. Most of this initialization is something that the average desktop programmer never sees, since it is handled by the computer boot code and operating system. But in an embedded system, it is just as likely as not that there is no operating system, and all boot code and other startup code must be explicitly provided. Some very critical hardware may need to be initialized first e.g. hardware that controls memory access times and address maps, as well as system clock hardware. Then some software initialization may need to happen, such as setting up a stack pointer and perhaps copying data from nonvolatile memory to volatile memory where it can be accessed and perhaps modified. After that will usually come another round of hardware initialization, setting up any peripheral devices that the system requires, and setting initial output states. Finally, yet another round of software initialization may occur.

This initialization is usually broken up into two sections, with the first hardware and software initialization steps often being done in what is known as the startup code, and the later hardware and software steps being done in the user program. This delineation is more distinct in a C program, where the startup code is invisible to the C program, being the code that happens before main() is run, and ending in a jump or call to main() where the visible C program begins. In an assembler program all the initialization steps may be equally visible in the user code, although even then the

first steps may reside in a separate startup source file.

A Note on the Example Programs

Each tutorial section will include a number of short example programs. The examples will start with the simplest concepts and add some concepts in each succeeding program. Along the way, some comments will be trimmed to try and help keep the visual clutter down and keep the focus on the newer concepts being presented. As an example, comments to the effect that "this bit/port/address needs to be adjusted for your particular hardware" will eventually disappear, because by then you should know that e.g. if I am discussing an LED output on PORTA bit 0 and on your hardware you are using an LED on PORTB bit 7 then you'll make that change accordingly. Or when I mention in the first programs that after a "ret" instruction that you'd better have set up the stack first, after a while that comment and others like it will disappear.

What next?

Before we can proceed to our first microcontroller program, our LED blinky "Hello World," there are more details we need to cover concerning the design and operation of microcontrollers. That will be the subject of the next tutorial in this series.

Next post by Mike Silva:
Introduction to Microcontrollers - Further Beginnings