6-1-2012

Modernizing the Microcontroller Laboratory with Low-Cost and Open-Source Tools

Kenneth J. Hass
Bucknell University, kjh016@bucknell.edu

Juliana Su

AC 2012-4195: MODERNIZING THE MICROCONTROLLER LABORATORY WITH LOW-COST AND OPEN-SOURCE TOOLS

Prof. K. Joseph Hass, Bucknell University

K. Joseph Hass was a Distinguished Member of the technical staff at Sandia National Laboratories, where he worked in embedded signal processing and radiation-tolerant microelectronics, before beginning his career in academia. He joined the Microelectronics Research Center at the University of New Mexico and continued his work on radiation-tolerant microelectronics, adding an emphasis on unique signal processing architectures, reconfigurable computing elements, and ultra-low-power CMOS electronics. The research group at UNM moved to the University of Idaho, where Hass studied memory circuits based on magnetic tunnel junctions, earned his Ph.D., and began teaching in the Electrical and Computer Engineering Department. In 2009, Hass accepted a teaching position as an Assistant Professor at Bucknell University, where he teaches courses in digital design and embedded computing.

Juliana Su, University of Virginia

Juliana Su received a B.S. degree in computer science and engineering and an M.S. degree in electrical engineering from Bucknell University in 2009 and 2011, respectively. She is currently pursuing a Ph.D. degree in computer engineering at the University of Virginia. Her research interests include body sensor networks, embedded systems, reconfigurable computing, and field-programmable gate arrays.

Modernizing the Microcontroller Laboratory with Low-Cost and Open-Source Tools

Instructors in the area of embedded systems face an ongoing struggle to incorporate current design and development techniques into their laboratory exercises. In addition to the difficulty of keeping pace with technological advances in the field, a significant investment is often made in the design tools and development boards with the expectation that these costs will be amortized over five years or more. Fortunately, a number of microcontroller manufacturers have adopted the IEEE 1149.1 Standard Test Access Port, more commonly known as a JTAG interface, to facilitate programming and debugging their processors. Software development tools have also begun to converge into a collection of open-source point tools, such as a compiler and assembler, that are managed by an open-source integrated development environment. As a result, instructors can easily provide a sophisticated development environment for embedded systems using tools and techniques very similar to those used in industry, supporting a variety of microcontrollers for less than the cost of a typical textbook.

We have used such a development environment in a microcontroller systems design course for second-year students in Electrical Engineering and Computer Engineering, using ARM Cortex-M3 microcontrollers. Our students are able to program the microcontrollers in both C and assembly language, or a combination of the two. We use the GNU debugger, gdb, with a commercial high-speed USB-to-JTAG interface and a low-cost development board. Students performed all of the essential development tasks, writing their programs, compiling them, and debugging their code, from within the Eclipse integrated development environment.

We found an important pedagogical benefit accrued from using this tightly integrated development environment, in that students were able to learn and practice more sophisticated debugging techniques. They are generally accustomed to programming on a conventional computer where the interaction between the software and hardware is just an abstract notion. When programming embedded systems, this interaction often leads to programming errors and the students' previously learned debugging techniques are of little help. The instructor may suggest that the student insert instructions to light an LED at certain points in the program, or add print statements to send messages to a terminal. Unfortunately, this is a tedious method that forces the student to modify and recompile their program repeatedly and may introduce its own errors. In some cases, the student may be able to use a software simulation of the microcontroller, but these simulations have a limited ability to simulate hardware beyond the microcontroller itself. In our microcontroller laboratory, students use graphical tools to add breakpoints or single step through their code, either by C program line or assembly instruction. The contents of all processor registers, C variables, and specified memory addresses are automatically updated and can be displayed in a number of useful formats. As a result, students clearly see the connection between the C program, the compiled assembly version of the same program, and their interaction with the processor and other hardware resources.

Introduction

In 2010, we began to completely redefine our introductory course in microcontroller system design, in anticipation of offering the revised course in the spring of 2011. In prior years the course had focused primarily on assembly language programming of the Motorola 68HC11.

Since this processor is essentially obsolete and the tools used in our laboratory sections were significantly out-of-date, this was an appropriate time to wipe the slate clean and reconsider all aspects of the course.

We identified several unique pedagogical goals for this particular course. First, it is important that students use programming languages that are consistent with current industrial practice. Surveys taken over the last decade have shown that about 80% of embedded projects will use C and about 60% of these projects will rely on C as the primary programming language.[1] Assembly language programming is still used in about 60% of the projects but is rarely the primary language. While C++ is also a popular choice, twice as many projects use C as the primary language and assembly language is more likely to be used than C++.

Clearly, engineers designing embedded systems need to be familiar with both C and assembly language programming, at least for the foreseeable future. Perhaps more importantly, these engineers must understand the linkage between C code and assembly. They must have some understanding of how a compiler will convert their C programs to assembly language so that they may write more efficient programs and effectively debug those programs. For example, students need to learn how variable declarations in C (such as *static*, *volatile*, or *const*) affect how that variable is stored in memory, whether it is initialized at run time, and how it will be treated by compiler optimizations. Since many embedded systems are created using both C and assembly language, students must be able to craft functions in assembly language that can be invoked, with parameter passing, from C and then return their results to the calling C program using the established compiler conventions.

A second pedagogical goal is to have the students experience current design techniques and tools. Embedded system tools have evolved dramatically, and sophisticated tools are often a necessity when designing with modern microcontrollers; gone are the days when students should spend their time deducing the correct hexadecimal representation of an instruction given the operation and the addressing modes of the operands. Programming is commonly done from within an integrated development environment (IDE) that combines a smart text editor with automated techniques for compiling, linking, and debugging. While experienced programmers may be quite comfortable writing makefiles and linker scripts, the tight coupling between the source code editor and the debugging tools can greatly improve the efficiency of finding and eliminating program errors.

The third pedagogical goal for our course is that students should be able to easily transfer what they have learned from the microcontroller systems course into other courses, research activities, and their own personal projects. We hope that students become excited about the creative potential of embedded systems and continue to design with microcontrollers after the course ends. In particular, students should be able to easily incorporate microcontrollers into their fourth-year capstone design or undergraduate research projects. To that end, we want the embedded design environment to be *portable* in every way:

- The IDE should be physically portable, in the sense that it could be used in any campus laboratory or on a student's personal computer. This implies that the IDE must be free of licensing restrictions.
- The IDE should be portable across target embedded systems. Ideally, the same IDE could be used to develop applications on a wide variety of microcontrollers, so that student projects could scale upward into research or capstone requirements.

- Similarly, the IDE should be portable from small projects to large projects. The tool set should be easily scalable to more ambitious projects than we would encounter in this course.
- The IDE should be portable across host operating systems. Our campus laboratories use both Windows and Linux platforms, and student laptops often run Apple's OS X. Tools that can be used on any of these platforms will be more accessible to students.

Our final goal is to teach microcontroller system design using a modern microcontroller architecture. While there is certainly a strong demand for 8-bit and 16-bit microcontrollers, the strongest growth in embedded systems is occurring for 32-bit devices.[2,3] The complexity and cost of microcontroller devices is driven more by the program memory and peripherals integrated onto the same chip and less by the width of the processor registers and arithmetic-logic unit, so that complete 32-bit microcontrollers are now available for less than $1 in large quantities. Therefore, the need to teach students how to perform multibyte arithmetic or deal with a segmented memory space has greatly diminished, allowing us to incorporate higher-level content in introductory courses.

Of course, the cost of providing a new suite of software tools and laboratory hardware is always a limitation. Our laboratory sections typically have no more than nine student stations so, as a minimum, we would require ten sets of software and hardware tools that could be simultaneously in use. Allowing for capstone projects and research activities could double that number. While our capital budget might allow for a significant investment in the laboratories that would be amortized over several years, the desire for true portability drives us toward a different perspective. Our goal is to allow any student to acquire his or her own complete suite of development tools, essentially identical to what is being used in our laboratories, for about the same price as a typical engineering textbook. Ideally, the cost will be reduced to the point that we can reasonably require every student to purchase their own development tools. This is not intended as a cost-saving measure for the university but rather to encourage the proliferation of microcontroller projects beyond the classroom.

Developing a new infrastructure for the microcontroller laboratory required simultaneously selecting three interdependent components, as shown in Figure 1: the IDE, the debug adapter, and the microcontroller development board. As discussed above, the IDE is a collection of software tools that the student uses to write program code, compile it, and generate a final form for the program that is suitable for execution on the target microcontroller. The IDE runs on a desktop host computer, typically under Microsoft Windows or Linux in our laboratories. The debug adapter is a small hardware interface between the IDE and the target microcontroller. It receives commands from the IDE, typically via a USB connection, to program the microcontroller and control the execution of the firmware. Status and debugging information is also returned to the IDE through the debug adapter. At the end of the chain is the microcontroller development board, which of course contains the target microcontroller but also provides physical connections to the debug adapter and to any student-built circuitry.

Integrated Development Environment

After considering commercial development tools available from both the microcontroller vendors themselves and third-party software vendors, we realized that these tools had fairly restrictive licensing conditions. Purchasing sufficient licenses for all of our laboratories, even with an

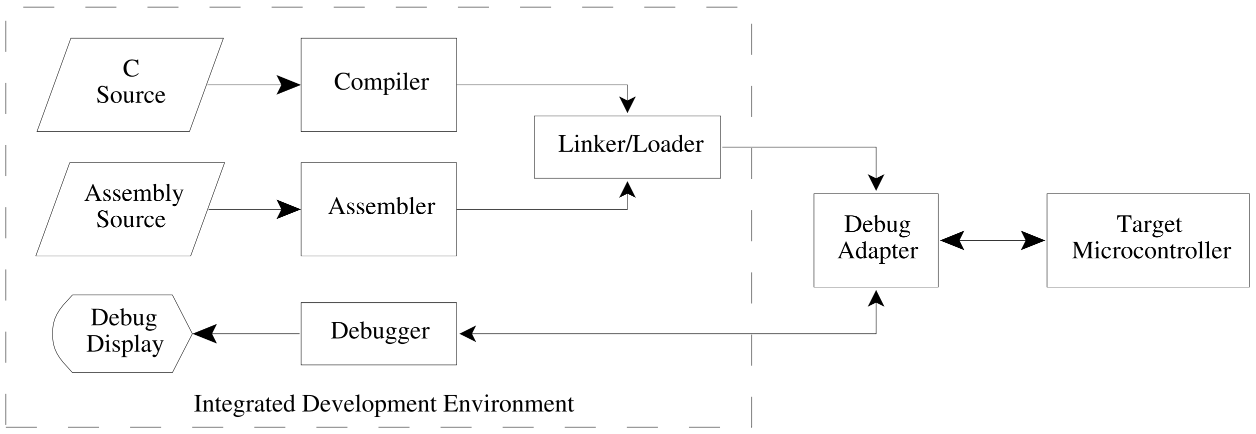


Figure 1: Typical microcontroller system development environment

educational discount, would represent a considerable investment. A less-expensive alternative would be the free "evaluation" versions of a particular vendor's tools, but these are typically limited in the size of the program that can be compiled or the software license expires after some months of use. Furthermore, in many cases the commercial tools were not available for a Linux or OS X host computer and could produce firmware for only a small set of target microcontrollers.

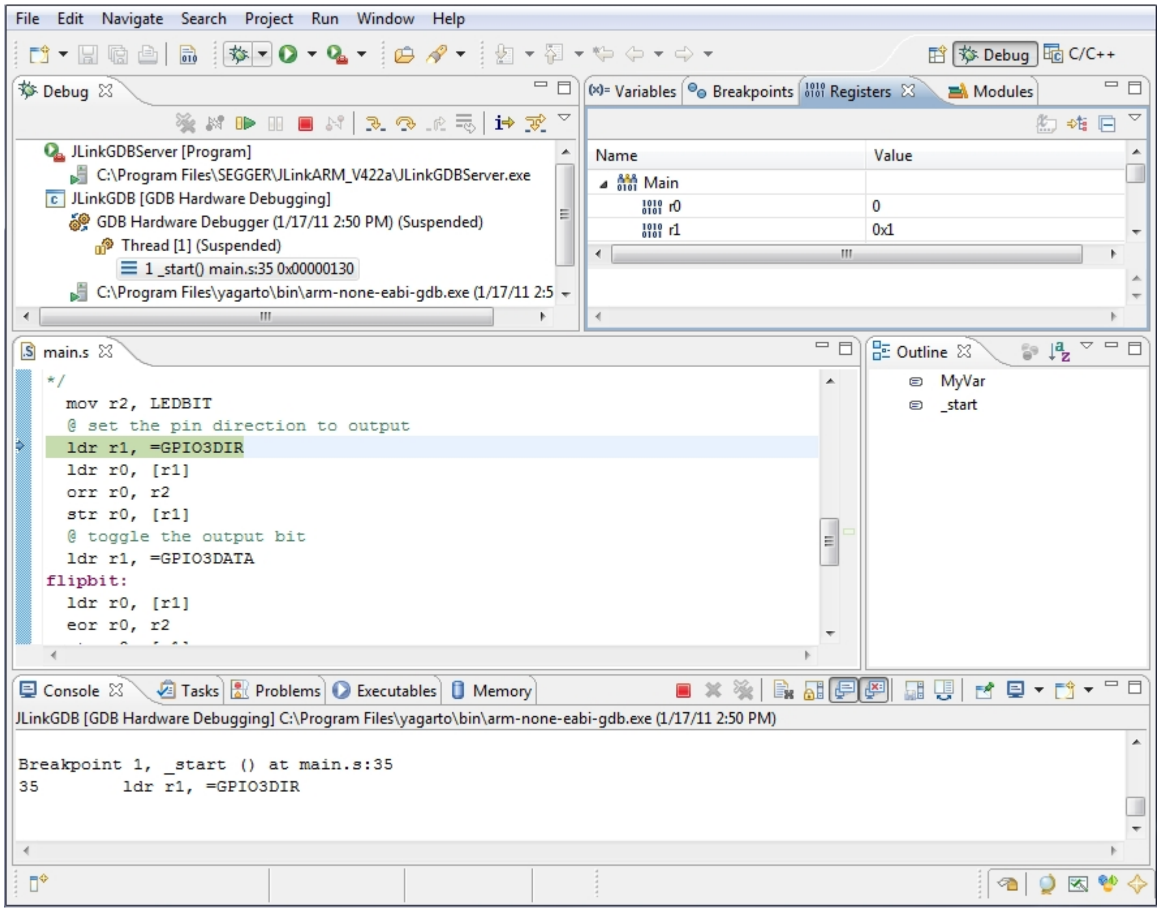
With the goals of portability and low-cost in mind, we decided to develop our microcontroller laboratory around the Eclipse IDE.[4] While originally developed by IBM for Java programming, Eclipse was released as an open-source project in 2001 and has been extended to support a wide variety of programming languages. Eclipse is itself written in Java, which means that it can be used on any operating system that can run the Java Virtual Machine, including those most commonly encountered in our engineering laboratories. Eclipse is well supported and actively developed. We found that, in many cases, the commercial IDEs offered for sale consisted of Eclipse with proprietary "plugins" developed by the software vendor.

We certainly recognize that commercial software development tools can enhance the productivity of the embedded system developer. Tool vendors add value to the development environment in several ways, such as providing a user interface that is efficiently tailored to the target processor, automating the tedious tasks of configuring hardware peripherals, and supplying well-written subroutines for common applications. However, for pedagogical purposes it is important for the student to have some understanding of the fundamental behavior and usage of the microcontroller, just as we require students to learn circuit analysis before they use SPICE or Karnaugh maps before they employ logic synthesis. We believe that the deeper understanding gained in this manner outweighs the initial drudgery in the learning process.

The Eclipse IDE provides an editor that recognizes the syntax of assembly and C language files, uses color highlighting for syntactic elements, and provides a simple means for indenting code. Eclipse also integrates project management capabilities for determining dependencies between source code files, locating libraries, and maintaining up-to-date executable files. However, Eclipse does not inherently include compilers, linkers, or debuggers for for C or assembly. These tools must be obtained separately, and Eclipse must be told how to invoke them. Fortunately, Eclipse is aware of common formats for generated files, such as object files and log files, and can generally work with such files with little assistance. It is not surprising that when asked to name their favorite software tools, the most popular response to a survey in 2011 was Eclipse.[3]

For our work, we decided to use open-source “back-end” tools from the Free Software Foundation. These tools are popular with industrial embedded systems developers,[3] are well supported by the user community, and are being actively developed. We rely heavily on the GNU Compiler Collection,[5] which includes *gcc*, the C/C++ compiler, *as*, the assembler, and *ld*, the linker/loader. Debugging is performed using the GNU Project Debugger, *gdb*.[6] These tools are easily integrated with Eclipse by adding the C/C++ Development Tooling package.[7]

A typical debugging session in Eclipse is shown in Figure 2. In this example, an assembly



```
JLinkGDBServer [Program]
    C:\Program Files\SEGGER\JLinkARM_V422a\JLinkGDBServer.exe
JLinkGDB [GDB Hardware Debugging]
    GDB Hardware Debugger (1/17/11 2:50 PM) (Suspended)
        Thread [1] (Suspended)
            1 _start() main.s:35 0x00000130
    C:\Program Files\yagarto\bin\arm-none-eabi-gdb.exe (1/17/11 2:5
```

Name	Value
Main	
r0	0
r1	0x1

```
*/
  mov r2, LEDBIT
  @ set the pin direction to output
  ldr r1, =GPIO3DIR
  ldr r0, [r1]
  orr r0, r2
  str r0, [r1]
  @ toggle the output bit
  ldr r1, =GPIO3DATA
flipbit:
  ldr r0, [r1]
  eor r0, r2
```

```
MyVar
_start
```

JLinkGDB [GDB Hardware Debugging] C:\Program Files\yagarto\bin\arm-none-eabi-gdb.exe (1/17/11 2:50 PM)

```
Breakpoint 1, _start () at main.s:35
35          ldr r1, =GPIO3DIR
```

Figure 2: Screen capture from Eclipse debugging session

language program is executing on a microcontroller. A large pane on the left side comprises the editor view of the source file, with syntax coloring for assembly language code. The highlighted line is the next instruction that will be executed by the debugger. When debugging a C program the IDE shows both the C and assembly language versions of the program and the debugger can apply breakpoints or single-stepping to either representation. The upper right pane shows the current values of the microcontroller registers, and this view is automatically updated whenever the debugger stops program execution. When debugging a C program, a similar view shows the values of the C variables that exist in the current scope. The tabs in the wide bottom pane can be used to examine the contents of specified memory addresses or view console messages from the various back-end tools.

Software development for small embedded systems is complicated by the fact that the software is being developed on a host computer, which is running an operating system such as Linux or Windows, while the final form of the program must be executable on the (much different) target platform. For our purposes, the target platform is typically a small "bare metal" microcontroller board that does not in itself have the resources to support an operating system or the software development tools. Compilers that work in this way are known as *cross compilers*, and it is possible to create a cross compiler for any of the many possible combinations of host computer and target platform that are supported by gcc. Fortunately, this task has already been performed for most of the commonly encountered cases. The current version of gcc supports a wide variety of target processors, including most ARM processors, the Atmel AVR, and the Motorola 68HC11 or 68HC12.

Target Microcontroller Selection

Selecting the microcontroller to be used as the design vehicle in our introductory course was the first decision to be made. We recognized that several microcontrollers are already well established in higher education, particularly the Motorola/Freescale 68HC11 and 9CS12 processors[8–10] and the Intel 8051 architecture.[11] These processors are well-supported for educational purposes, with a variety of textbooks and prepared laboratory exercises. However, in both cases, the processor architecture is somewhat dated and we felt that neither choice was representative of current trends in embedded system design.

The Atmel AVR processors, particularly in the form of the Arduino platform, also deserved careful consideration. The Arduino ecosystem includes a very user-friendly design environment and a rich selection of "shields" that can be easily plugged on to the top of a processor board and extend the hardware capabilities. For hobbyists and experimenters who are more interested in accomplishing some task than understanding the details of embedded systems, the Arduino is an attractive option. Unfortunately, it is exactly these details of embedded system design that we desire to teach in our microcontroller course so the Arduino is not particularly attractive to us. Furthermore, our experience with Arduino projects suggests that this platform may not have the flexibility or computational throughput to support the range of student projects that we encounter.

We noted over the past few years that there has been a dramatic increase in the number of semiconductor manufacturers offering microcontrollers available with an ARM architecture, including Freescale and Atmel. The dominance of 32-bit processors in current embedded design projects was noted earlier, and closer examination shows that 54% of survey respondents in 2011 indicated that they were considering an ARM processor for their next project (up from 45% in 2010).[3] In comparison, 32% said they would consider *any* processor from Freescale, 21% would consider a processor from Atmel, and 19% would consider a processor from Intel. While ARM processors do not yet enjoy the same popularity in the classroom, there are successful educational precedents.[12]

Furthermore, we found that cross-compiler versions of gcc that would produce executable code for ARM microcontrollers are readily available. Michael Fischer maintains the open-source YAGARTO project, which includes the gcc back-end tools and detailed installation guidelines for a Windows host computer.[13] ARM also maintains and provides its own open-source gcc toolchain for their microcontrollers, with pre-built binary versions as well as documentation and source code.[14] Mentor Graphics makes available a free version of its CodeSourcery development tools, including gcc and gdb for Linux or Windows host computers.[15] All of these options comprise ready-to-run development tools that need only be installed on the host computer, but it is certainly possible to create one's own version of the tools from the source code available from the Free Software Foundation.

We decided to focus our microcontroller course on the latest version of the ARM 32-bit architecture, which is used in the *Cortex* family of processors. The Cortex-A processor line is used in high-performance multimedia and communications appliances, and can be found in virtually all smart cellphones and tablet computers. The Cortex-R series is intended for applications requiring a moderate level of real-time performance. At the lowest performance levels are the Cortex-M processors, although these devices may include hardware floating-point and operate at clock frequencies up to 100 MHz.

Debugging Interface

Since the debugging process is a critical component of the microcontroller course, we placed a high priority on the availability of debugging tools that represent the current practice in embedded systems design. Our portability goal provided an additional incentive to employ debugging tools that could be used with a variety of microcontrollers. Thus, for our purposes, an important aspect of the ARM Cortex architecture definition is that the debugging interface is standardized and well documented, so that the same debugging tools can be used on any semiconductor manufacturers implementation of a Cortex-M3 microcontroller.

In their specification for the Cortex-M3 processor, ARM provides definitions for two standardized debugging interfaces. The first uses the standard JTAG (IEEE Std 1149) interface definition and requires a minimum of four unidirectional signal lines. JTAG has been the traditional debugging interface for ARM processors but is now considered to be a legacy interface, and new processor designs are encouraged to implement the ARM Serial Wire Debug (SWD) interface instead. SWD uses only two signal pins, a clock and a bidirectional data line, which is an important consideration for microcontrollers that have few pins. Furthermore, the SWD interface supports a data rate more than twice that achievable with JTAG. As a practical matter, the currently available Cortex-M3 microcontrollers may provide either JTAG, SWD, or both.

Having committed to free, open-source, software development tools, the largest expense for the microcontroller teaching laboratory becomes the *debug adapter*, the physical and electrical interface between the host computer and the target development board. A variety of low-cost adapters are available for the legacy JTAG port, and simple interfaces can be built from a handful of common components that connect to the host computer's parallel printer port. On the other hand, there are relatively few commercial debug adapters for ARM's SWD interface and a more complex USB connection to the host computer is required to support the higher data bandwidth requirements.

Fortunately, we located a commercial debug adapter, the J-Link from SEGGER Microcontrollers, that supports both the JTAG and SWD interfaces and can be purchased with a substantial educational discount. SEGGER also supplies a *gdb server*, software that communicates between the GNU debugger (gdb) and their J-Link adapter.[16] When the power requirement is modest, as is the case for our laboratory activities, the J-Link is capable of providing power for the microcontroller through the debug connector. Incorporating the J-Link and gdb server into the Eclipse environment is straightforward, and is discussed in the YAGARTO documentation.[13] As of this writing, SEGGER's software for the J-Link is well-supported for host computers running Microsoft Windows but a beta version for use with Linux is also available.

For those who prefer to use only open-source software, the Open On-Chip Debugger (OpenOCD) project, created by Dominic Rath, aims to provide software support for a variety of hardware debug adapters.[17] While OpenOCD does not currently support the SWD interface this is a planned enhancement.

Development Board

The final puzzle piece in our microcontroller laboratory development system was the selection of a Cortex-M3 development board. Since our course is intended for both electrical engineering and computer engineering students, a large component of the course content involves the electrical interfaces between the microcontroller itself and common peripheral devices. Therefore, our preference is for a very simple development board, including only the minimum necessary capabilities for programming the processor. Ideally, we would supply the students with the microcontroller devices and a generic prototyping board and they would assemble all of the required components themselves. The lack of a Cortex-M microcontroller available in a common dual in-line (DIP) package makes that approach impossible now, but we are optimistic that such devices will be available in the near future.[18]

Some of the Cortex-M3 microcontroller manufacturers themselves offer simple, low-cost development boards. Examples of such products include the LPCXpresso boards from NXP Semiconductor[19] and the STM32 discovery from ST Microelectronics.[20] These boards have a built-in debug adapter so they can be connected directly to a host computer so they provide a very easy and low-cost introduction to that particular microcontroller. However, the software tools provided with the board are often restricted to a particular tool vendor's "evaluation" tools, which may be limited to developing only small programs or may require the purchase of a new license after several months. To avoid these limitations, intrepid engineers have often been able to physically separate the target microcontroller from the debug adapter and use a generic adapter with open-source development tools.

When selecting a development board to use in our course, we searched for a low-cost board that included the Cortex-M3 microcontroller, its system clock crystal, a standard connector for the debug adapter, and a standard connector for supplying power to the microcontroller. Since we anticipated connecting the microcontroller to student-built circuits on a prototyping board we also wanted a relatively easy means for making those wired connections between the development board. We ultimately selected the Olimex[21] LPC-P1343 prototyping board, as shown in Figure 3. This board includes the NXP LPC1343 microcontroller, a 20-pin connector for the debug adapter, and a mini-USB connector. Eight LEDs and two pushbutton switches are provided and are connected to the microcontroller's general purpose I/O port pins. A small prototyping area on the right side of the board includes an array of holes with a standard 2.54 mm spacing, and most of the microcontroller pins are directly accessible from the two columns of holes at the left edge of the prototyping area. We soldered a receptacle connector onto the board so that students could insert wires in the receptacle and connect the microcontroller pins to a breadboard.

In addition to the standard ARM debugging port, the LPC1343 microcontroller incorporates the circuitry and firmware for a USB interface. A jumper on the Olimex board allows it to appear as a USB mass storage class device (i.e. a flash drive) when connected to a PC. When accessed in this way, the LPC1343 appears to contain a single file that comprises the program memory of the microcontroller. The microcontroller can be reprogrammed simply by deleting this file and dragging a new program file to the device. While there is no debugging capability, we have found this to be a convenient method for restoring the Olimex boards to their original state between laboratory sessions. The board may also be powered through the USB connector, whether behaving as a USB device or simply running the student's microcontroller firmware.

As a final programming method, the LPC1343 contains in-system programming (ISP) firmware in ROM. When in the ISP mode, the microcontroller responds to commands given over a serial line connected to its UART pins. ISP commands allow the user to erase, reprogram, read, and execute the microcontroller program memory.

For ease of handling in the laboratory we have mounted the Olimex boards onto one side of a standard breadboard, as shown in Figure 4. The J-Link is connected to the host computer via the USB cable on the far left, and it is connected to the debug connector on the Olimex board with a ribbon cable. Note that a receptacle has been added to the prototyping area on the Olimex board, and wires have been added so that breadboard power is supplied from the Olimex board, which in turn is powered by the J-Link adapter.

Course Materials

A common problem in all engineering courses is providing instructional materials that reflect the current state-of-the-art. A variety of textbooks are available that present the Freescale 9S12, Intel 8051, or Microchip PIC microcontrollers. While not written as textbooks, there are also quite a few introductory and tutorial books for the Arduino platform. However, to our knowledge there are no textbooks for university courses that use the Cortex-M architecture as a model microcontroller. For the first offering of the updated microcontroller course we required students to purchase Yiu's book on the Cortex-M3, which was written as a general introductory and reference text.[22] As such, it assumes some prior understanding of digital electronics and microcontroller systems.

The course instructor provided supplemental material to ensure that students had the necessary

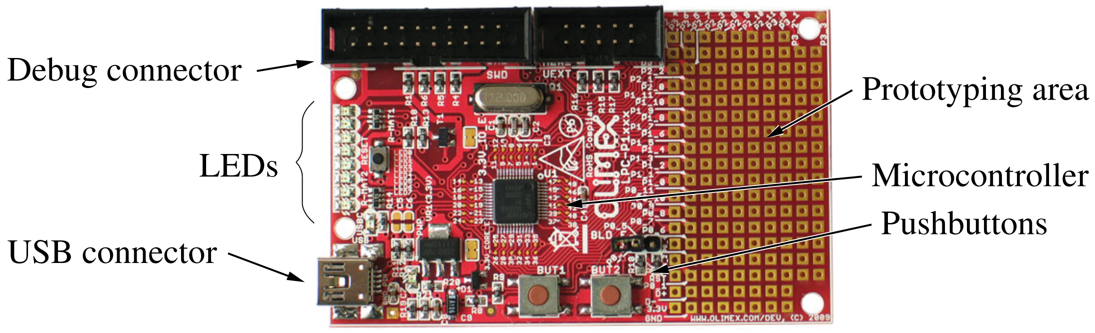


Figure 3: Olimex LPC-P1343 prototyping board (photo from Olimex[21])

background for the course, recognizing that there was a considerable range in experience across the computer engineering and electrical engineering students. The students tend to have experience programming in Java from their computer science coursework, but very few have programmed in C or C++, so a large part of the course content comprises learning to program in a new language.

While Yiu's book thoroughly presents the Cortex-M3 architecture and instruction set, each microcontroller vendor is free to add their own peripheral I/O functions around the standard ARM processor core. There is no guarantee that a given Cortex-M3 microcontroller will incorporate a UART, for example, and the UART in an NXP microcontroller may look much different from the UART in a Cortex-M3 microcontroller manufactured by Texas Instruments or ST Microelectronics. Therefore, students must also study the user's manual and data sheet for the particular microcontroller being used. We provide these documents for the students in our course, but at a total of almost 400 pages they find them to be quite daunting and they are often frustrated when an internet search does not quickly return example code to meet their needs.

Laboratory Exercises

Our microcontroller class has the typical schedule for our engineering courses, with three one-hour lectures and one three-hour laboratory session each week. Students work in pairs under the guidance of the course instructor and an upper-class student teaching assistant. The laboratory exercises for the course emphasize both the unique programming aspects of embedded computing as well as the important electrical considerations, again reflecting the fact that our students are a mix of computer and electrical engineers.

In the first few laboratory sessions, the students are provided a program skeleton that they must complete to perform simple I/O functions, such as reading the state of a pushbutton switch and turning an LED on or off. These exercises are intended to familiarize the students with the mechanics of using the IDE, debug adapter, and microcontroller board. By executing one

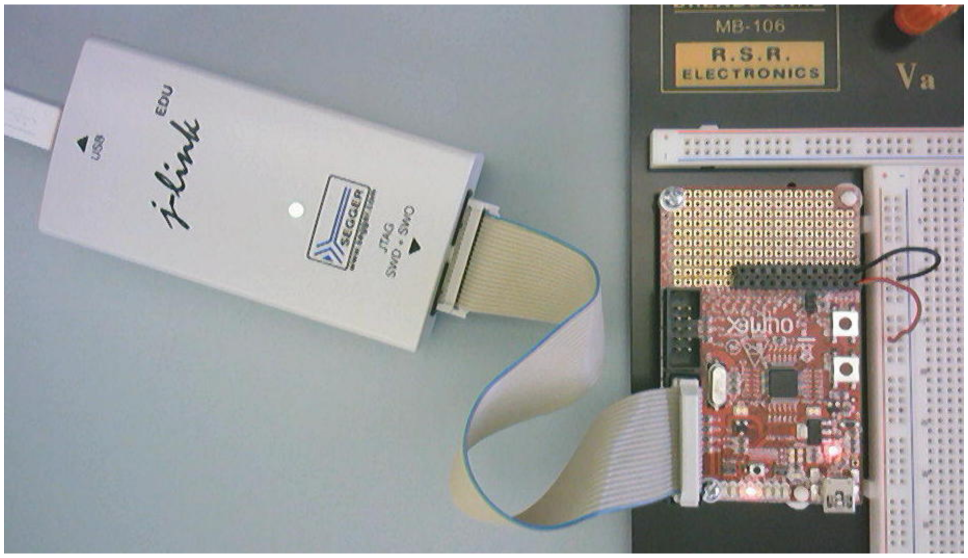


Figure 4: Basic laboratory configuration of J-Link and Olimex board

assembly instruction at a time, they observe the interaction between their code and the hardware.

More sophisticated programming examples explore the design and behavior of interrupt service routines, such as using timer interrupts to generate a pulse-width modulation (PWM) control for an LED. In one session, they write an assembly-language program that recursively computes the Fibonacci sequence. This exercise illustrates a number of difficult concepts, including the use of the stack for both data storage and function return linking. By calculating successively larger Fibonacci numbers, the students can directly observe the increase in the stack size and program run time.

In another laboratory session, the students do nothing but observe the d.c. characteristics of the microcontroller I/O pins and compare their observations to the data sheet specification. They use a simple program that reads the state of an input pin and then turns an LED on or off based on the observed logic value. By connecting a variable voltage source to the input pin, students can determine the input switching thresholds (V_{IH} and V_{IL}). They make these measurements twice, for a normal input pin and for an input pin with hysteresis. They usually notice that the LED is dim when the input voltage is near the switching threshold and there is no hysteresis, and we explain that electrical noise is causing the input to behave erratically. When hysteresis is used, the input switches cleanly between logic levels and the LED is either fully illuminated or completely dark. Replacing the LED with a fixed resistor allows the students to make measurements of the output voltage (V_{OH} and V_{OL}) and calculate the corresponding output current values (I_{OH} and I_{OL}).

Some laboratory exercises combine electrical and programming concepts. For example, we provide the students with a simple 16-button multiplexed keypad and ask them to write a program that displays the hexadecimal value of the pressed key on four LEDs. The notion of multiplexing is discussed in a prior lecture, and the students understand that they must program the keypad row wires to be output pins while the column wires are inputs, look for a pressed key, reverse the roles of the row and column wires, and look for a pressed key again. The students are free to use whatever programming technique they chose to convert the row and column information into the value of the key, and some will create a lookup table while others will create a mathematical algorithm or devise a large nested conditional structure. To receive full credit for their work, the students must implement “rollover” detection so that pressing two keys simultaneously does not result in the display of an incorrect value.

Assessment of the student’s performance in the laboratory sessions takes several forms. In many of the exercises the students are required to directly observe and measure some aspect of the microcontroller’s behavior, such as toggling an I/O port pin before and after executing a function and observing the function’s execution time on an oscilloscope. By initializing all RAM locations with a known value the students can clearly see how the stack grows when a function is invoked with different parameters, and they are able to correlate this observation to the number of registers they chose to save on the stack in their functions. During the laboratory session the students are required to record observations such as these, and to provide brief interpretations of the data, on laboratory worksheets.

A second in-class assessment technique is that students are often required to demonstrate their work for the lab instructor. Of course, the students must demonstrate that any software or hardware they created during the lab will function properly and meet the specifications given to them, but they are also expected to show that they can employ common debugging techniques. We expect that all students will be able to execute their code by single-stepping, both by C

Assessment Statement	Mean Response
The laboratory section was a valuable part of this course.	4.75
I would recommend this course to other students interested in this subject.	4.35
I can write and use programs in assembly language.	4.21
I can write and use programs in C for a microcontroller.	4.32
I am able to debug a microcontroller program.	4.30
I can interface a microcontroller to simple devices such as LEDs and switches.	4.46

Table 1: Assessment results

statements and by assembly instructions, and use breakpoints. Students must be able to use the IDE to observe the run-time behavior of processor registers, C variables, and RAM locations. The laboratory worksheets include checklists of such skills that must be demonstrated, and the instructor will initial the worksheet after the successful demonstration. This also provides a focused opportunity for the instructor to discuss and illustrate the importance of good debugging techniques.

A final laboratory assessment is conducted outside of the laboratory session. When a laboratory exercise requires that students write their own code they must provide that code to the instructor by email. The students are given guidelines for writing embedded software in assembly language and C, and their submitted code is graded for compliance with those guidelines. The guidelines have been drawn from a sampling of industry "best practices" and include recommendations for basic attributes such as effective comments and self-documenting identifiers.[23,24] While we do not have rigid requirements for code style, we do expect that students will use consistent capitalization for identifiers and uniform indentation.

Reflections and Conclusion

The revised microcontroller course was well-received in its first offering. Selected assessment results are shown in Table 1, where students gave a numerical response from 1 (disagree strongly) to 5 (agree strongly). Students were able to see an engineering design path from the simple ARM microcontroller used in the course to the much more powerful Cortex-A processors used in their smart phones and touchpad computers. Allowing the students to add circuitry of their own around the microcontroller gave them insight into the electrical engineering aspects of embedded system design.

Our curriculum had traditionally required students to take the microcontroller course in their second year, while the first course specifically dedicated to digital systems design was not taken until their third year. Electrical engineering students would have taken one computer science course, and computer engineering students would have taken two computer science courses, before the microcontroller course. In this course structure, the students are able to contextualize the microcontroller as a much smaller extension of their desktop computer, which enables them to transfer their programming experiences to a new platform. However, the students were generally unable to place the microcontroller in the context of digital systems, where it can be seen as simply a programmable finite state machine. Students also lacked an intuitive grasp of synchronous digital systems so they often struggled with the simple notion of storing data in a register. As a result of these difficulties we have reviewed our course sequence and reversed the order of the digital design course and the microcontroller course, beginning with the class of 2015.

The students were most critical of the reading material used in the course. They expected to have a typical textbook that would incorporate most of the critical background material and walk them, step by step, through the learning process. When required to learn from a reference text (like Yiu) or from the manufacturer's data sheet and user manual they became frustrated, and they seem to have difficulty gleaning the relevant information from these sources. Since this is often how practicing engineers are forced to learn new technology we feel that we should help students become more accustomed to the situation rather than removing all of their difficulties, and we will be more proactive in addressing this issue at the beginning of the course.

By the careful use of open-source development tools, a low-cost debug adapter, and a very simple development board we have succeeded in creating a new laboratory environment for our microcontroller system design course. The students are using design tools that are consistent with current industry practices and are studying a modern, 32-bit processor architecture. The total cost of each laboratory station, including the development board and debug adapter, is significantly less than a typical new engineering textbook. As this course continues to evolve we will develop additional learning materials and laboratory exercises to support our students, and we look forward to seeing the fruit of these efforts in future capstone design projects.

References

[1] M. Barr, "Real men program in C," *Embedded Systems Design*, pp. 9–11, July/August 2009.

[2] C. Holland. (2011, Dec. 12) MCUs: High-end devices flourish. [Online]. Available: http://www.edn.com/article/520296-MCUs_High_end_devices_flourish.php

[3] UBM / EE Times Group, "2011 Embedded Market Study," 2011.

[4] The Eclipse Foundation. (2011) Eclipse. [Online]. Available: http://www.eclipse.org

[5] Free Software Foundation. (2011, Dec. 6) GCC, the GNU compiler collection. [Online]. Available: http://www.gnu.org/software/gcc/

[6] ——. (2011, Dec. 6) GDB: the GNU project debugger. [Online]. Available: http://www.gnu.org/software/gdb/

[7] The Eclipse Foundation. (2011) Eclipse CDT. [Online]. Available: http://www.eclipse.org/cdt/

[8] R. Alba-Flores, "Laboratory enhancements for improving embedded systems education," in *American Society for Engineering Education Annual Conference & Exposition*, 2007.

[9] T. Morton, "New developments for courses in embedded microcontrollers," in *American Society for Engineering Education Annual Conference & Exposition*, 2007.

[10] C. Choi, "A microcontroller applications course and Freescale's microcontroller student learning kit," in *American Society for Engineering Education Annual Conference & Exposition*, 2008.

[11] D. Wilcox, S. Wilson, and G. Wostenkuhler, "Embedded design in a sophomore course," in *American Society for Engineering Education Annual Conference & Exposition*, 2008.

[12] A. Clements, "ARMs for the poor: Selecting a processor for teaching computer architecture," in *ASEE/IEEE Frontiers in Education Conference*, 2010.

[13] M. Fischer. (2011, Nov. 20) YAGARTO - yet another GNU ARM toolchain. [Online]. Available: http://www.yagarto.de/

[14] J. Ye. (2011, Dec. 22) GNU tools for ARM embedded processors. [Online]. Available: https://launchpad.net/gcc-arm-embedded

[15] Mentor Graphics. (2011, Dec. 29) Sourcery codebench overview. [Online]. Available: http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/overview

[16] (2012) SEGGER microcontroller. [Online]. Available: http://shop-us.segger.com/

[17] (2012) Open On-Chip Debugger. [Online]. Available: http://openocd.sourceforge.net/

[18] NXP Semiconductors. (2011, Oct. 25) Cortex-M0 microcontrollers in high-volume TSSOP and SO packages target 8/16-bit applications. [Online]. Available: http://www.nxp.com/news/press-releases/2011/10/nxp-cortex-m0-microcontrollers-in-high-volume-tssop-and-so-packages-target-8-16-bit-applications.html

[19] ——. (2012) LPCXpresso. [Online]. Available: http://ics.nxp.com/lpcxpresso/

[20] STMicroelectronics. (2010, Sep. 14) STM32 discovery kit. [Online]. Available: http://www.st.com/internet/com/press_release/p3065.jsp

[21] Olimex, LTD. (2009, Dec. 14) Development boards and tools. [Online]. Available: http://www.olimex.com/dev/index.html

[22] J. Yiu, *The definitive guide to the ARM Cortex-M3*, 2nd ed. Elsevier, 2010.

[23] M. Barr, *Embedded C Coding Standard*. Netrino, LLC, 2009.

[24] "Joint Strike Fighter air vehicle C++ coding standards for the system development and demonstration program," Lockheed Martin Corporation, Tech. Rep. 2RDU00001 Rev C, 2005.