Embedded Programming for Computer Scientists

Peter M. Maurer
Dept. of Computer Science
Baylor University
Waco Texas, 76798-7356

Abstract – The past two decades have seen an explosion in embedded programming technology. Embedded controllers are everywhere, in practically every electronic device. These controllers, which are essentially computers, have become increasingly complex, with a corresponding increasing in the complexity of their programming. Embedded programming is a tremendous opportunity for computer science students. Here we give an overview of the three levels of embedded programming, bare machine, Real Time OS, and conventional OS. We also give suggestions for projects at each level. The aim is to allow the reader to construct his or her own course in embedded programming.

1 Introduction

Imagine yourself teaching Introduction to Computer Science. It's the first day and your classroom is filled with eager young students wanting to learn everything there is to know about computer programming. You give them their first "confidence builder" program, and they eagerly run off to the lab do the project (maybe!). However, the laboratory is not their first encounter with computers today. The alarm clock that woke them up for class has a computer in it, the microwave they used to warm their breakfast has a computer in it, the watch that told them when to leave for class has a computer in it, the cell phone that they (hopefully) silenced at the start of class has a computer in it, and if they drove to class, they interacted not with one computer, but with many. A car's electronic ignition has a computer, the automatic transmission has a computer, the anti-lock brakes have a computer, the intermittent windshield wipers have a computer, and on and on and on. If they drove a fairly new car, these computers did not work in isolation, but were consolidated into a communication network that controls the car as an integrated unit.

So the question is: who programs all of these computers? And who will program them in the future? Will it be your students? Will their undergraduate education prepare them to enter a world where everything they touch contains a computer? Will they understand even the basics of this type of programming? If yours is a typical undergraduate program, the answer is *probably not*.

There was a time when most embedded programming consisted of a few lines of code that could easily be hacked out by a non-expert. There are still embedded programs like this, but they are becoming the exception rather than the rule.

Processors designed for embedded programming are called "microcontrollers." A microcontroller is a computer in every sense of the word, except the name. As I am writing this, I'm looking at a sales blurb for a 32-bit microcontroller with more than half a megabyte of internal memory and a 32-megabyte external memory interface. (Cost: around $10.00 in quantity.) Any application that needs a device of this complexity also needs expert programming.

Of course, one can learn embedded programming "on the fly." One can learn anything "on the fly." It's usually better and faster to learn from a well-designed comprehensive course. Our purpose here is to discuss the design of such a course. There are, in fact, many textbooks devoted to embedded programming. Unfortunately, most of these focus on only one type of embedded programming. My aim here is to give a more broad-based view that will encompass most of the important issues. This will enable you to decide which topics are most important to you, so you can design your own course effectively.

The good news is that C has become the de-facto standard for embedded programming, and C++ is usually also available. Therefore the basic programming skills learned in an introductory programming course will also be used in embedded programming. (One can also use assembly language, but it's probably best to avoid this.)

There are three main levels of embedded programing. At Level 1, the programmer must write every line of code. There is no operating system and essentially no C library. The program is boot-strapped into the memory of a microcontroller, usually by burning it into non-volatile memory. Execution is initiated by a power-on interrupt. At Level 2, a miniature operating system (called a Real Time Operating System or RTOS) is used to assist with multi-tasking and other housekeeping chores. The RTOS, or portions of it, become part of the program and are boot-strapped into non-volatile memory along with the programmer's code. At Level 3, there is a real operating system, usually Linux. Special operations must be used to access the hardware features, but for the most part, you program the system as if it were an ordinary Linux system. The OS and the programmer's code are booted either from non-volatile memory or from a memory card.

It may be tempting to concentrate on Level 3, because the environment is familiar and extremely powerful. But this would give an inaccurate picture of the world of embedded programming. We recommend that all three levels be taught, with special emphasis on Level 1.

2 Before We Start

If you are not comfortable constructing simple hardware, or allowing your students to do so, it is best to cooperate with one or more colleagues from electrical or computer engineering. Even though we will focus on programming, the aim is to eventually use this programming to control special purpose hardware. It is best to have a number of pre-completed hardware projects available for students. These projects will have all the necessary hardware assembled and tested. There will be an empty socket that will eventually hold a microcontroller programmed with your students' code. At Level 1, five to seven projects will suffice. At Level 2, two or three complex projects will be needed. Simple projects at Level 2 can use the Level 1 hardware. At Level 3, one or two trivial projects should be used to familiarize students with the basic hardware, and then the sky's the limit.

If your students are comfortable with designing and building simple hardware, then they should be allowed to do so. The microcontrollers discussed here are available in DIP packages that can be used with standard breadboards. Hardware-capable students should be encouraged to branch out to more complex projects of their own devising.

3 Level 1: Bare-Device Programming

At Level 1 there is a small microcontroller, some simple hardware and virtually nothing else. The procedure for writing programs is quite different from that used for a typical console-level program. There is no I/O interface. The objects *cin* and *cout* don't exist. There is some debugging help available, but usually you will not be able to debug the program while it is running on the microcontroller.

First, you must decide which microcontroller to use. PIC [1] is the most popular, but we have found the AVR [2] 8-bit microcontrollers easier to work with. These are not the only choices (there are hundreds), but they are probably the two best options for a starting point. Another popular choice is the ARM [3] architecture because it is the architecture found in smart phones and tablets. However, the ARM architecture requires a substantial investment in development equipment.

For simplicity, we will describe the tools needed for AVR development. In our examples we will concentrate on the Atmel ATTiny85 microcontroller (75 cents in quantity), and will use the free development system provided by the ATMEL corporation (Atmel Studio). There are similar tools available for the PIC and other platforms.

The first concern will be memory management. There are three types of microcontroller memory, flash RAM, which holds the program, static RAM which is used for variables and stacks, and Electrically Erasable Programmable Read-Only Memory (EEPROM), which holds constant data. Although the EEPROM (and sometimes the flash RAM) can be written by the program, one needs to exercise care when doing so. Both EEPROM and flash RAM can endure only a limited number of write cycles before wearing out (typically 10,000 to 100,000 writes) so these types of memory must not be used for program variables. The static RAM can be written an unlimited number of times, but there is only a limited quantity available. Deep subroutine nesting will cause problems. Recursive programs are probably not a good idea. Large arrays probably won't work. The ATTtiny85, which we are using for an example, has 8 Kilobytes of flash RAM, 512 bytes of EEPROM, and 512 bytes of static RAM. The ATTtiny85 is, of course, rather small, but even the larger 8-bit microcontrollers have strictly limited amounts of memory.

3.1 Getting Started

To help you understand what embedded programs do, we'll start with a simple example. Consider the circuit of Figure 1.

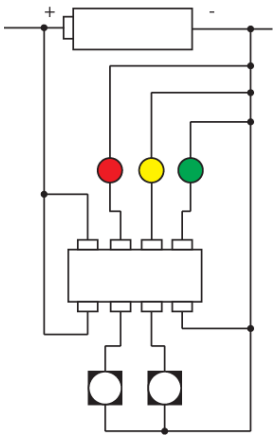

Figure 1. A Simple Circuit.

This circuit has an 8-pin, the TTiny85, three LEDs, and two buttons. Several programming assignments can be completed with this hardware. For a first assignment, we'll flash the LEDs on and off using the program of Figure 2.The TTiny85 has five general-purpose I/O pins, which are collectively called I/O PORTB. Each pin can be configured as an input or an output pin during program initialization. Programming is accomplished by moving values to memory mapped registers, which are defined as variables in the "avr/io.h" include file. These variables are DDRB, which is used to configure ports as input or output, PORTB, which is used to configure input pins and to set output pins to specific values, and PINB which is used to read the values of input pins. We won't use PINB in Figure 2. The Red, Yellow, and Green LEDs are attached to pins 2, 1, and 0, respectively. These pins are controlled by the three low-order bits of PORTB/DDRB/PINB. The left-hand button is attached to pin 3, and the right-hand button is attached to pin 4. These pins are controlled by bits 3 and 4 of PORTB/DDRB/PINB. The bits of each register are numbered from low order to high order starting with zero.

When setting the direction of a pin, a zero in DDRB indicates an input pin while a one indicates an output pin. For the input pins, it is necessary to use PORTB to set the internal pull-up resistors. This causes the default value of the pin to be 1. If we don't do this, the input pins will require extra external hardware to maintain their default values. A one-bit in PORTB activates the pull-up resistor, a zero lets the pin float.

```
#include <avr/io.h>
int main( ) {
    // initialization
    // Configure pins 0, 1, and 2 as output pins, 3, and 4 as input pins
    DDRB = 0x07;
    // activate the pull-up resistors of ports 3 and 4, setting the default input to 1
    PORTB = 0x18;
    // infinite loop
    for (;;) {
       PORTB = 0x1C; // turn on the RED LED
       PORTB = 0x18; // turn off the RED LED
       PORTB = 0x1A; // turn on the yellow LED
       PORTB = 0x18; // turn off the yellow LED
       PORTB = 0x19; // turn on the green LED
       PORTB = 0x18; // turn off the green LED } }
```

Figure 2. Rolling the Red/Yellow/Green LEDs.

Slowing the flashing down to the point where it is visible will be the student's first challenge. With the code of Figure 2, the flashing won't be visible. All three LEDs will glow dimly. To slow things down, we need to add a delay between turning the LEDs on and off. The simplest way to do this is to add a programmed delay as shown in Figure 3. (Remember that this is an 8-bit microprocessor, hence the *unsigned char* variables.) This delay might be enough, but it probably won't be. One can add nested loops to increase the delay to any desired value. Once the lights start blinking at the desired rate, we could try something fancier, like make the hardware work like a traffic light: *long red, short yellow, long green.*

The next challenge is to use the input pins to detect a button-push. With our sample hardware it will be necessary to debounce the button. A button does not go from off to on instantly. It goes from off, to rapidly changing between on and off, to on. When detecting a change in the button, the microcontroller program must wait a few milliseconds, and then confirm the status of the button. A change can be detected by reading a register and testing a bit. Debouncing can also be done with external hardware, which simplifies the

programming. We will use the left-hand button is attached to pin 3. Figure 4 shows how to read the state of the button.

```
#include <avr/io.h>
int main( ) {
   unsigned char i,j,k;
      // initialization
      // Configure pins 0, 1, and 2 as output pins, 3, and 4 as input
      pins
      DDRB = 0x07;
      // activate the pull-up resistors of ports 3 and 4, setting the
      default input to 1
      PORTB = 0x18;
      // infinite loop
      for (;;) {
         PORTB = 0x1C; // turn on the RED LED
         for (i=255 ; i>0 ; i--);
         PORTB = 0x18; // turn off the RED LED
         for (i=255 ; i>0 ; i--);
         PORTB = 0x1A; // turn on the yellow LED
         for (i=255 ; i>0 ; i--);
         PORTB = 0x18; // turn off the yellow LED
         for (i=255 ; i>0 ; i--);
         PORTB = 0x19; // turn on the green LED
         for (i=255 ; i>0 ; i--);
         PORTB = 0x18; // turn off the green LED
         for (i=255 ; i>0 ; i--); } }
```

Figure 3. Visibly Rolling the Red/Yellow/Green LEDs.

```
#include <avr/io.h>
int main( ) {
   unsigned char i,j,k;
   // initialization
   // Configure pins 0, 1, and 2 as output pins, 3, and 4 as input pins
   DDRB = 0x07;
   // activate the pull-up resistors of ports 3 and 4, setting the default
   input to 1
   // Also, turn on the RED LED
   PORTB = 0x1C;
   // infinite loop
   for (;;) {
      // zero means the button is down
      if ((PINB&0x80) == 0 {
         // wait a bit
         for (i=255 ; i>0 ; i--);
         // is the button still down?
         if ((PINB&0x80) == 0) {
            // code to turn off the current LED
            // and turn on the next one is placed here
            while ((PINB&0x80) == 0) {
               // wait for the button to start back up  }
                  for (i=255 ; i>0 ; i--); } } } }
```

Figure 4. Responding to a Button.

Completing these simple assignments will teach the student some important basic points. In embedded programming, "input" means reading the status of an input pin, "output" means setting the value of an output pin, and significant amount of code is required to configure the microcontroller for different operations.

As the preceding examples show, an embedded program consists of a set of initializations followed by an infinite loop. Delays of some sort are almost always necessary. Programmed delays are conceptually simple, but a better method is to place the microcontroller in a wait state, using an interrupt from an internal timer is then used to terminate the delay. This method consumes less power, because much of the microcontroller's circuitry will enter a sleep state during the wait.

3.2 The programming process.

Microcontroller programs must be developed on a standard platform such as MS Windows and cross-compiled it to produce microcontroller object code. The object code is then downloaded into a microcontroller for execution. As mentioned above, we use the free software from Atmel Corporation for code development and cross-compilation. The object files are in Intel's ".hex" format, which can be downloaded using a free software package called "avrdude" [4]. Figure 5 shows a picture of some sample hardware similar to that of Figure 1, but without the buttons. The black rectangle in the center is an 8-bit socket. The black rectangle below it is the microcontroller, which has been removed from its socket for programming.

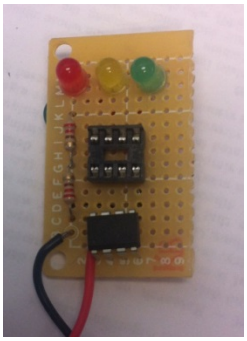

Figure 5. Sample Hardware.

Figure 6 shows an inexpensive programmer called the "Pocket AVR Programmer" [5]. It is about 1x1.5 inches in size and costs around $15. The gold socket at the top is a mini-USB connecter which is used to connect to a desktop PC or laptop. The cable at the bottom goes to a breadboard which is wired to communicate with several different microcontrollers. Figure 7 shows the breadboard with the microcontroller in place for programming. The connector to the programmer is at the far left.

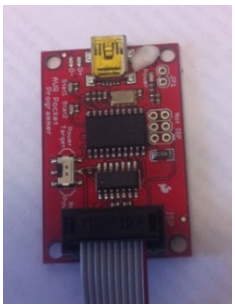

Figure 6. The Pocket AVR Programmer. [5]

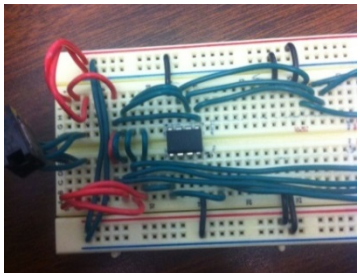

Figure 7. The Microcontroller in Place for Programming.

Once the microcontroller is programmed, it is placed back in its socket and the red and black wires at the lower left of Figure 5 are attached to a power supply. If one is very lucky, something will happen. However, in most cases, the program will have to be debugged before it starts to work properly. Many microcontrollers have in-place debugging features that can be invoked by a sophisticated development system, but with hardware shown in Figure 6, it is necessary to debug the program "offline." The AVR software has an offline simulator that can be used to verify program correctness. It is best to simulate the program before downloading it for the first time. Once the program is working correctly in the simulator, it can be downloaded into the real hardware for further testing.

3.3 Advanced Programming

Additional assignments can explore the input and output features of the microcontroller as well as some of its internal features. The logical next step is to replace the programmed delays of Figures 3 and 4 with internal-timer based delays. This will result in an interrupt-driven program. This style of programming is substantially different from the examples given in Figures 2-4, but is necessary to access the more advanced features of the microcontroller.

Some of the additional features which should be explored are pin-change interrupts, analog input using pin comparators or analog-to-digital conversion. Analog-to-digital input is often used to determine the position of a dial attached to a potentiometer, but it could be used in connection with other features to record analog signals. Another important feature is analog output using Pulse Width Modulation (PWM). PWM can be used to vary the intensity of an LED, or control the speed of an electric motor. Some advanced microcontrollers provide digital-to-analog output, although this is rare.

Some microcontrollers provide Universal Serial Asynchronous Receiver Transmitters (USARTs) that can communicate with the serial ports of a conventional computer. This port can be read and written by a terminal program, allowing a user to communicate directly with the program on the microcontroller.

An increasingly important feature of embedded program is networking between two or more microcontrollers using either the Serial Peripheral Interface (SPI) protocol or the Two Wire Interface (TWI) protocol. These protocols can be explicitly programmed, but most microcontrollers provide built-in hardware which simplifies the programming process.

Another important consideration is power management. Certain features permit unused portions of the microcontroller to be powered down, and others permit the microcontroller to enter various levels of sleep modes when waiting for events to occur. These modes of operation are extremely important for battery-powered circuits, so it will be beneficial for students to learn about them.

4 Level 2: Real-Time Operating Systems

At level 2 we add an operating system, the main benefit of which is multi-tasking. Everything we learned at level 1 is still valid. We do things in much the same way as at level 1, but now we can control multiple independent devices simultaneously with only moderate difficulty. There are many microcontroller-based Real Time Operating Systems (RTOS) available, some of which are free. One of the most popular is FreeRTOS [6], which is available for a number of different platforms, including the AVR ATMega series. Unfortunately, the larger RTOS's are too large for the smaller microcontrollers such as the ATTiny85. An even smaller RTOS, FemtoOS [7], is available for these smaller microcontrollers.

The OS itself is the only new tool needed at this level. Cross-compiling, simulating, and downloading, are done using precisely the same tools used in Level 1.

The first step in creating an RTOS-based program is to copy the OS files into your project. The OS itself becomes the project with your code consisting of a set of subroutines that are called by the OS at various times. The main routine will be provided by the OS, but there will be a primary subroutine for each task. Each task will consist of an initialization section followed by an infinite loop. Tasks are pre-emptible via interrupts, and will be scheduled according to a system of priorities which you can designate using configuration files. Operating system calls are available for task synchronization, timed waits, and inter-task communication.

It will generally be necessary to modify the OS source files to some degree, although these modifications will normally be confined to the OS configuration files. Because of this it is usually a good idea to start with a sample project that is as close as possible to the project you wish to complete. You can then modify the sample project to do what you want. This saves the trouble of creating configuration files from scratch.

Figure 8 shows a two-task FemtoOS project using the ATTMega168 microcontroller. Two pins of the microcontroller are wired to LEDs, one red and one green. These two pins are controlled by the two least significant bits of the devLedPORT and devSwitchPIN registers. The variables devLedPort, and devSwitchPIN, are mapped to microcontroller registers by the OS include file. The button is wired to a pin controlled by the third bit of the devSwitchPIN variable, which is also mapped to a device register.

The program blinks the two LEDs alternatively, switching from one the other when the button is pushed. There are two tasks, one to manage the blinking, and another to watch the button. A global variable is used for inter-task communication. The programming of device registers is essentially the same as at level 1. However, delays are handled through the OS, not by programmed loops.

The two functions "appLoop_LEDtask1" and "appLoop_DoButton" are the main routines for the LED task and the Button task respectively. The "TimeToSwitch" variable is used for task communication. The first loop in the LED task turns on the red LED, delays 200 milliseconds, then turns off the red LED and delays another 200 milliseconds. This procedure is repeated until the "TimeToSwitch" variable is set to true by the Button task. When "TimeToSwitch" becomes true, the first loop is broken, "TimeToSwitch" is set back to false, and the second loop is entered. The second loop repeatedly turns on the green LED, delays 200 milliseconds, turns off the green LED, and delays another 200 milliseconds. As with the first loop, if the button task sets "TimeToSwitch" to true, the loop is broken, "TimeToSwitch" is set to false, and the task returns to the first loop.

The Button task waits for a button to be pressed. When a button press is recognized, the "TimeToSwitch" variable will be set to true. The LED task has the responsibility for setting "TimeToSwitch" back to false. The Button task is a bit more complicated than one might expect, because it must debounce the button. The Button task waits for the button to be pressed. This will be detected as a change in bit three of the devSwitchPIN register. If such a change is detected, the task delays 100 milliseconds, confirms that the bit has indeed changed, and sets "TimeToSwitch" to true. Then the task waits for the button to be released. This will cause bit three of devSwitchPIN to return to zero. Waiting is done by repeatedly delaying 100 milliseconds and retesting the devSwitchPIN bit. Finally, the task waits for "TimeToSwitch" to be set to false by repeatedly delaying for 100 milliseconds. After "TimeToSwitch"

becomes false, the Button task resumes waiting for the next button press.

The code of Figure 8 is compiled and simulated using the same tools as were used in Level 1. The initial include statement inserts the required OS definitions into the users program. The FemtoOS source-code files must be made part of the project, which allows the OS code to be compiled into the user's program. Once the code has been simulated, the object file is downloaded to the microcontroller and run just as in Level 1.

```
#include "femtoos_code.h"
static Tbool TimeToSwitch = false;

void appLoop_LEDtask1(void) {
   for (;;) {
      for (;;) {
         devLedPORT |= 1;
         taskDelayFromNow(200U);
         devLedPORT &= 0xFE;
         taskDelayFromNow(200U);
         if (TimeToSwitch) break;   }
      TimeToSwitch = false;
      for (;;) {
         devLedPORT |= 2;
         taskDelayFromNow(200U);
         devLedPORT &= 0xFD;
         taskDelayFromNow(200U);
         if (TimeToSwitch) break;   }
      TimeToSwitch = false; } }

void appLoop_DoButton(void) {
   for (;;) {
      if (devSwitchPIN & 4) {
         taskDelayFromNow(100);
         if (devSwitchPIN & 4) {
            TimeToSwitch = true;
            while (devSwitchPIN & 4) {
               taskDelayFromNow(100);  }
            while (TimeToSwitch) {
               taskDelayFromNow(100);  } } } }}
```

Figure 8. Switch from One Blinking LED to Another.

The programming exercises for Level 2 should be more complex than those for Level 1. To facilitate this we recommend that a larger (in terms of pin count) microcontroller be used, primarily because it is easier to control multiple devices if more pins are available. We suggest the ATTiny861, or the ATMega168.

5 Level 3: A Complete System

Level 3 is quite similar to ordinary Linux programming. There are many devices available, but we recommend the Raspberry PI [8] which is available for less than $50. It is a complete Linux system that can be used as a desktop computer, if you wish. (As a desk top, it is rather slow.) You can use it both as a development environment and as part of an embedded system. There are commercial kits available for the Raspberry PI, that cover everything from robotics to home theater. There are even educational laboratories aimed at the K through 12 audience. There is online documentation (available from many sources) that covers programming, OS modifications, and sample projects.

It is best to ignore all the commercial products and focus on programming the general purpose I/O pins. This is done in the same manner as in Level 1, namely by moving values to or reading values from memory mapped registers. Figure 9 shows a picture of the Raspberry PI. The General Purpose I/O (GPIO) pins are at the lower right. The device is booted from an SD card, which is the rectangle at the far right. The device pictured is wired to operate as a desktop computer, with the video port at the bottom, power at the upper right, internet connector at the upper left, and USB hub at the center left. Mouse and keyboard are attached to the USB hub.

General purpose I/O is managed through a Broadcom BCM2835 ARM peripherals chip [9], whose GPIO registers are mapped to the Raspberry PI's memory space starting at 0x2002000. Figure 10 shows the general outline of an LED flasher task. Unlike the ATTiny and ATMega microcontrollers, the Raspberry Pi has two separate registers, one for setting pins to 1 and another for setting pins to 0. (Such variations between different products are common.)

The LED must be connected between pins 7 and 8, with a resistor to limit the voltage. Figure 11 shows the pinout of the GPIO pins. In Figure 9, Pin 1 is the rightmost pin of the top row. The pins marked as GPIO can be used for any purpose and can be configured as input or output. The other pins (other than power and ground pins) can also be configured for general purpose I/O if their specialized functions are not required.

Figure 9. The Raspberry PI.

Configure *Pin-7* as an output, using *Function-Select* register.
for (;;) {
Move 1 to *Pin-7 GPIO-Set* register
Pause
Move 1 to *Pin-7 GPIO-Clear* register
Pause }

Figure 10. A Raspberry PI LED Flasher Program.

3.3V	1	2	5V
I2C1 SDA	3	4	5V
I2C1 SCL	5	6	GROUND
GPIO4	7	8	UART TXD
GROUND		10	UART RXD
GPIO 17	11	12	GPIO 18
GPIO 27	13	14	GROUND
GPIO 22	15	16	GPIO 23
3.3V	17	18	GPIO 24
SP10 MOSI	19	20	GROUND
SP10 MISO	21	22	GPIO 25
SP10 SCLK	23	24	SP10 CE0 N
GROUND	25	26	SP10 CE1 N

Figure 11. Raspberry PI GPIO Pinout [6].

The possibilities for the Raspberry PI are nearly limitless. The SP10 pins (See Figure 11) can be used to communicate with Level 1 or 2 microcontrollers using the SPI protocol. The I2C1 pins do the

same using the TWI protocol. This gives one the power to construct complex systems with Level 1 or 2 microcontrollers performing the low-level tasks and Linux-based programs providing the over-all control.

The UART pins can be used to communicate with a serial port on another computer, or with a microcontroller USART interface. There are several other connectors in addition to the pins described in Figure 11. There is a camera connector near end of the Ethernet connector. There is an LED screen connector above the memory card slot. There is a JTAG interface and an additional GPIO header. (These last two have no pins attached and require expert handling.)

The power of this $30 device is astounding. I will leave the exploitation of this power to your imagination.

6 Conclusion

As mentioned above, the past two decades have seen an explosion in embedded programming. Microcontrollers have proliferated into almost every electronic device. In addition, controllers have become increasingly more complex and increasingly less expensive. The explosion in embedded programming can only be expected to continue. As systems become more complex, there will be an increasing need for expert programmers to program them. This is an important opportunity for computer science students, one that cannot be ignored. A one-semester course in embedded programming will prepare students for this exciting and rewarding opportunity.

7 References

1. PIC Microcontrollers http://www.microchip.com
2. AVR Microcontrollers http://www.atmel.com
3. ARM Cortex Microcontrollers http://www.ti.com/lsds/ti/arm/overview.page
4. AvrDude http://www.nongnu.org/avrdude
5. Pocket AVR Programmer http://www.sparkfun.com
6. FreeRTOS http://www.freertos.org
7. FEMTOOS http://www.femtoos.org
8. Raspberry PI http://www.raspberrypi.org
9. BCM2835 http://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf