Digital Electronics II: Microcontrollers

Purpose
In this experiment we introduce microcontrollers, powerful single-chip computers that can be programmed to perform almost any digital task.

Introduction
Let’s suppose you want to build something more complex than the digital circuits we studied in the previous experiment. For example, you might have a Wheatstone bridge and a null detector that you want to hook up to a computer so you can automatically balance the bridge and record the null point, maybe with an adjustable excitation voltage for the bridge. Or, say you want to build a tiny, battery-powered data transmitter to feed to a dolphin and find out how his body temperature varies while he’s swimming around in the ocean. Since it’s hard to transmit signals through a dolphin, the transmitter should store the temperature data until it is excreted by the dolphin and floats to the surface where your receiver can detect the signal.

These and many other applications require complex digital circuitry. Any digital system, including a programmable computer, can be built by combining the gates and flip-flops we discussed in the previous experiment. But it is hardly practical to build something like a computer by connecting together a bunch of TTL chips, since you could easily require thousands or even millions of gates. There are much better ways to get the job done. In fact, if you find yourself using more than a few discrete logic gates in a circuit you are probably making a mistake.

As a scientist, you should generally try to avoid building anything digital. The first example above could be handled by a PC with data acquisition cards and a power supply with a computer interface. For the potentiometer of the bridge you could use a digital potentiometer or a stepper motor driving an analog pot, both of which can be connected to a computer with commercial hardware. However, for the second example you would probably have to build everything from scratch.

There are three general approaches to building complex digital circuits: application specific integrated circuits (ASICs), programmable logic devices (PLDs) including the outrageously popular field programmable gate arrays (FPGAs are seriously impacting the market for complicated ASICs), and microcontrollers. ASIC is a general term for any complex integrated circuit that is devoted to a very specific task. Examples are chips designed for cell phones, TV sets, or controllers for liquid crystal displays. Generally, ASICs are designed to serve large commercial markets, and so you won’t find anything that will be much help for the two
applications mentioned above, or for most scientific applications. But if there is an ASIC that does what you need to do, or that can be adapted to your task, you should certainly use it. If you have a really large project with at least a million dollars to spend you could make a custom ASIC that does just what you want. ASIC manufacturers have many ready-to-go design elements (called standard cells), including complete computer processors, digital filters, counters and timers, memories and so forth, so you can generally avoid having to do much gate-level design work. In many cases it is possible to include analog circuitry as well, such as output drivers or ADCs. ASICs (and also programmable logic) are commonly used in high energy physics experiments. Check this link www.hep.anl.gov/elec_support/ for several examples.

Assuming there is no ASIC that does what you need and you can’t afford a custom ASIC, you might consider programmable logic devices like the FPGAs. These are chips containing many logic gates, and in most cases also flip-flops, which can be programmed (sometimes only once, sometimes many times) to interconnect the gates and flip-flops in a way that performs a custom function (see H&H 8.15, 8.27). A single PLD can replace a large pile of TTL chips, and the smaller PLDs containing a few thousand gates are inexpensive and relatively easy to program. The largest PLDs can contain 300,000 gates or more, and the programming effort can become a major project.

By far the most common approach to complex digital design is to use a microcontroller, a single chip programmable computer. Compared to programmable logic, microcontrollers process data more slowly but with much greater flexibility and power, and they are generally cheaper and more power efficient. They are available in incredible variety, from 8-pin versions costing less than a dollar to devices with processing power rivaling that of a PC. A given device may offer only digital input and outputs pins (I/O pins), or may include analog comparators or even ADCs. Modern versions usually contain their own program and data memory, and some have programmable non-volatile memory so they can remember what they were doing before the power went out. There are many specialized varieties of microcontroller, including DSP chips (digital signal processor) that can implement fast digital filters and Fourier transforms, and devices that include high-level digital interfaces, such as Ethernet or USB. To get an idea of the variety available, have a look at some of the manufacturer’s web sites (ATMEL at www.atmel.com, Microchip at www.microchip.com; Motorola at www.freescale.com; Rabbit Semi at www.digi.com/products/wireless-wired-embedded-solutions).

In this experiment, we will take a look at what microcontrollers can do by working with the Arduino Uno board. “Arduino”, as they say at their website (http://www.arduino.cc/) “… is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. It’s intended for artists, designers, hobbyists, and anyone interested in creating interactive objects or environments.” Arduino has been highly successful in bringing out simple microcontroller boards that can be programmed easily. These boards have a huge international following that has resulted in a wide range of books, publicly available programs, robotics kits, and the like. These
resources make the Arduino a common and easy choice when thinking about a microcontroller project because it’s very very likely that someone has already solved your problem and has posted the solution code for FREE on the Arduino website.

The heart of the Arduino Uno is the 28 pin ATME吕布 ATmega328P. It is a member of the ATMEL 8-bit AVR microcontroller family, with 32 kBytes of 8-bit wide program memory, 1 kByte of electronically erasable and programmable read-only memory (EEPROM), and 2 kBytes of 8 bit data memory. The data and program memories are completely separate, making this a ‘Harvard architecture’ processor. Program memory uses the latest ‘flash’ technology, so you can program it over and over again as you debug your hardware and code, and it is non-volatile, so the chip won’t forget its programming when you turn the power off. Programming can be done without taking the chip out of its socket on the Arduino Uno board via a serial digital USB interface. You download a program development environment from the Arduino site, plug the Arduino Uno board into a USB port on your laptop, and start uploading programs to the Arduino microcontroller.

Nearly every pin of the ATME吕布 ATmega328P microcontroller chip can be programmed in multiple ways for various uses, and up to 23 of the 28 pins can be used as bi-directional digital I/O. There is an internal, variable frequency clock and three separate timer/counter/scalers. For dealing with analog signals there is a comparator with a programmable voltage reference, 6 channels of pulse-width-modulated (PWM) output to simulate adjustable external analog voltages, and to top it all off, a 10-bit analog-to-digital converter with a 6-channel analog multiplexer, so you can digitize 6 separate analog signals. Amazingly, you can have all this for around $3 per chip, just a bit more than you pay for the single discrete 2N4416A JFET we used in Experiment 8.

Here’s the bad news: All this power, typical of modern microcontrollers, comes at the price of complexity. Have a look at the data sheet for this chip (on our web site or at www.atmel.com). It is 448 pages long, and many aspects of the chip are only summarized there. Despite this, we will show you that it’s not too hard to get to know the ATmega328P and the Arduino Uno, thanks to some excellent tutorial info available from Arduino and the programming software they provide. These resources, and much else that you can find on the web, are the main reason we have chosen the Arduino for this course.

A FEW WORDS ABOUT OBSOLETE IN ELECTRONICS

Although we tend to think of electronics as a rapidly developing area, almost all of the analog parts used in circuits today have been around for a long time. The 2N3904 and 2N4416 transistors date from the 1960s, and the LF356 op-amp dates from 1975. Since the second edition of Horowitz and Hill (1989) the main change in analog electronics has been that packages have become smaller. The 0.1-inch pin spacing of DIP packages (which is great for learning and prototyping) has been largely replaced by 0.05-inch pitch and smaller surface mount...
packages. (When mounted on a printed circuit board, surface mount packages don’t require you to drill holes through the board for the leads to go through. If you get stuck prototyping with these smaller packages, try not to go below 0.05-inch pitch, unless you want to work under a microscope). If a new edition of H&H were to come out today, Chapters 1-7 would hardly have to be changed, except for the various tables containing real part numbers, such as Table 4.1 showing op-amps. Many of those part numbers have changed as manufacturers come and go and improvements are made, but most of the kinds of parts listed are still available. In any case, tables like this are unnecessary today, since you can easily visit the manufacturer web sites to see what’s currently available.

In digital electronics, change has been more rapid, as anyone who buys PCs knows. Not only have packages become smaller, but much more functionality is now included within any single package. Maximum clock speeds have continuously increased. There is nothing obsolete about the concepts described in the digital chapters of H&H, but today no one would build a microprocessor system like the one shown in Fig. 11.10, even if you could get the parts to do it. All of the functionality shown in Fig. 11.10 and more is now available on a single chip.

If you start looking for parts on manufacturers’ web sites and then try to buy what you find, you will discover that not all of those parts are really available. Some are too old, some are too new, and some were never manufactured in any quantity, or are only manufactured when a big fish comes along looking for 100,000 units. How do you tell what is really available? The easiest way is to use the web sites of major distributors (Digi-Key at www.digikey.com, Newark at www.newark.com, Mouser at www.mouser.com). If the part you want is stocked in quantity at a big distributor, it is really available and will probably remain so for at least a few years.

Readings
1. FC does not have much relevant reading on this subject. The most important information you will need will be presented in lecture or on the Arduino website.

2. For additional information you can read H&H Section 8.03. We won’t be using hexadecimal representation, 2’s complement, BCD, or Gray code (though binary arithmetic is important and useful to understand). However, hexadecimal, etc. are immediately important if you want to program many of the other microcontrollers on the market. Many of them are programmed in ‘machine code’, which uses hex coding. The Arduino environment uses a variant of C, and translates to AT machine code for you.

3. Everything else you will need this week can be found on our web site or at the Arduino web site. First, go online to the Arduino site at http://www.arduino.cc/ and click on the “Getting started” tab. Read the introduction, sections on downloading and installation, and read about the Arduino development environment and the available libraries

4. The ATmega328P datasheet from our website and read at least the first 7 pages about the capabilities of the microcontroller (OPTIONAL: To pg. 38 for more information).
Theory

BINARY NUMBERS

Microcontrollers are binary digital computers. Their entire operation is based on the use and manipulation of binary numbers. For example, the program memory is simply a sequential set of memory locations that store a list of particular binary numbers. Therefore, it pays to understand a bit (so to speak!) about the binary number system.

There are several ways to represent binary numbers. Recall that a single ‘bit’ can take on one of two values, typically listed as 0 or 1. Next, consider the 8-bit number 10110111. Each ‘place’ represents from right to left, increasing powers of 2, rather the same as how decimal numbers encode increasing powers of 10. For the number 10110111, if you add up the place value of each set bit, you will end up with the decimal integer $1+2+4+16+32+128=183$. In computer code, it is most common to represent an 8-bit binary number by two hexadecimal digits: $10110111=B7_{\text{h}}$, where the little h is H&H’s notation for hexadecimal (Section 8.03). Other notations you will see for this same number are: $D'183'$, $B'10110111'$, $H'B7'$, and $0xB7$ (the last one seems to be peculiar to Microchip). The range of positive 8 bit numbers is 0 to 255.

The same bit pattern 10110111 has another meaning if it is considered to be a signed integer in 2’s complement representation, where the range of integers we can represent with 8 bits goes from -127 to +128. To change the sign of a number in this system you complement and then add one. If you do this to our number you get 01001001 (decimal +73), so it represents -73 in the 2’s complement system. Notice if you add the 2’s complement representations of -73 and +73 you get all zeros plus a bit carried to the 9th place.

MICROCONTROLLERS AND BINARY NUMBERS

The microcontroller is a self-contained digital computer that can be programmed to do a huge number of useful jobs. As you likely already know, a program is a list of instructions stored in the microcontroller’s program memory. The microcontroller is designed to start at the beginning of the program memory and to sequentially retrieve the specific instructions, and execute the operations that are associated with each instruction. A system clock regulates the timing of instruction retrieval and execution. Each specific instruction will require a documented number of clock cycles for complete execution. The clock runs, the program executes, and the process marches forward. Some programs will repeat execution. Some programs simply stop at the end.

Each microcontroller on the market is designed with a specific set of instructions. The complete list of 131 instructions available for programming the ATmega328P is listed in Section 31 Instruction Set Summary, starting on page 427 of the datasheet. Each instruction is coded into the program memory as a specific binary code. With 131 possible instructions, the codes easily
fit in an 8-bit memory location (because 8 binary bits is enough to code for 256 distinct numbers). These instructions allow you to write very general programs. While each instruction is associated with a specific 8-bit number, we also associate the instruction with simple three-letter or four-letter names that help us remember what they do. Instructions will include Arithmetic and Logic instructions e.g., ADD for adding binary numbers, Branch Instructions e.g., JMP, that allow the program to make choices in behavior, Bit and Bit Test instructions such as BSET or BCLR, that allow the program to set or test for different logical conditions, and Data Storage and Transfer instructions e.g., MOV for moving a number from one memory location to another, allow the program to retrieve and modify its memory.

For the genuinely hard-core folks (folks with nothing but time to kill…), it is absolutely possible to program a microcontroller by directly writing a list of binary numbers into the program memory, and then telling the microcontroller to start execution. Essentially, NOBODY programs this way, because the list of binary numbers is nearly impossible to read. Even if you write the program yourself, in a couple of days you will not be able to just look at it and understand what it does.

Instead, most microcontrollers are programmed using a more readable language, referred to as Assembly Language. Assembly language programs use the nice names (see above) for instructions and are much easier to read. Each microcontroller has its own special assembly language instructions, with its own specific choice of names for each instruction. While instruction names are similar from one microcontroller to the next, they are never-the-less unique to each different manufacturer. Therefore, once you have selected the microcontroller that you’d like to use, you typically then need to learn the assembly language for that machine. For those of you who are interested, you can find the instruction set for the PIC16F676 microcontroller from Microchip, another popular brand and chip, on the Useful Documents page of the website.

Microcontrollers have modest-sized program memories. It can be important to write efficient code. Therefore, many people who use microcontrollers take the time to learn the assembly language and worry about nitty-gritty details of careful coding.

PROGRAMMING MICROCONTROLLERS WITH HIGHER LEVEL LANGUAGES

However, coding in assembly language almost guarantees that your code will need to be rewritten if you move to a different microcontroller, or that code you’ve borrowed from other sources will need a careful rewrite. Further, assembly language, while more readable than a list of binary numbers, is not as transparent as higher-level languages like C or BASIC. Therefore, a number of the microcontroller manufacturers have produced programming environments that allow you to program using high-level languages. The environment can then translate your code
to assembly, and then translate assembly to machine code, and then uploads your translated program to the microcontroller. The Arduino environment supports a variation of the C programming language, with a few special commands provided to allow simpler interaction with the microcontroller. Further, the Arduino user group encourages the sharing of code, so you can find a huge list of programs to use as starting points for your own projects.

**PROGRAMMING AND CODE TEMPLATES**

Try not to start with a blank page when you write code for a microcontroller. Instead, get a template, a working program that can run on your hardware and that does at least some of what you need to do. Then you can slowly modify the template to make it do what you want, always checking that you are not introducing bugs or killing it entirely. A big reason to use a popular series of devices like the Arduinos is the availability of template code (at the Arduino site, in books about the Arduino, and all over the internet). The template for our experiment is the file AlarmArduino_template.ino on our web site. *Bring a paper copy of the template to the lab.*

The hardware is specified at the top of the template (see schematic below, Fig. 10.1.) We will connect red LEDs to pins 9 and 8, which are bits 1 and 2 of I/O port C. We will also connect a piezoelectric buzzer to pin 10. The template code will turn on one LED, turn the other off, buzz the buzzer at 440Hz for 0.75 seconds, and then switch off the on-LED, turn on the off-LED, and buzz at 330Hz. Sounds like the Gestapo is coming!
Pre-Lab Problem

1) Download the Arduino software environment onto your laptop or one of the laptops in the laboratory. You’ll find it at the Arduino website, [http://www.arduino.cc](http://www.arduino.cc) under the Download tab.

2) Download the AlarmArduino_template.ino code from our website. Read through the code, this entire write-up, and the required readings listed above. Leave any notes in your lab notebook that you think are helpful in remembering how to use the template and the environment.

3) Write the code you will need to make the ‘Super (annoying) Alarm’ described below. You can write the code on paper by hand, use a text editor, or use the Arduino environment to make the ‘sketch’. Whatever you do, make sure you bring a paper copy of your assembly code to the lab with you, in your lab notebook.

4) On the Arduino website, click on the Learning tab. Find one or another of the example programs that looks interesting to you. Read about it. Describe what it does and what extra items you’d need to test it.
New Apparatus and Methods

Arduino Uno board and software environment

The Arduino Uno is a small circuit board with a USB interface that can be plugged into a USB port on any PC. The board gets its power from the PC’s USB, and it has a 5V regulator that you can use to power your circuit (they don’t say how much current you can draw from this power source, but you should probably keep it below 100 mA). The board has a 28-pin DIP socket that contains the ATmega328P microcontroller chip. The socket allows you to pop out the chip after you’ve programmed it, and install it into stand-alone circuit boards if desired. The board also has a separate power connector that you can use after you've programmed the microcontroller, so you can disconnect the Uno and run it on separate power sources. Finally, the board carries the clock for the microcontroller, and a variety of plugs that allow you to connect the microcontroller chip’s pins to the outside world.

We will connect our Uno boards to external LEDs and a piezo-buzzer using these pins.

The Arduino software environment allows you to write code (referred to as ‘sketches’ in the Arduino community), upload the code to the Arduino board, and read back information from the board.

PIEZO BUZZER

We will use a piezoelectric buzzer to get a little sound out of our Arduino. The data sheet is on our web site (CUI part no. CEP-1110). It is just a 25 nF capacitor with a piezoelectric dielectric made of ceramic PZT (lead zirconate titanate), and a flexible brass electrode. When a voltage is applied the PZT shrinks causing the brass electrode to bend. If you send it a sine wave or a square wave, you can make a sound like a smoke alarm, which is the main commercial application for these things. It will be pretty quiet in our application because we will drive it with just 5 V, and at a frequency that is below its natural resonance. If you want you could connect it to the Arduino through a transistor driver to make it louder.

Experiment

THE CIRCUIT

A schematic of the circuit we will build is shown in Figure 10.1. Be sure to put your LEDs in with the correct polarity. Pins of the ATmega328P microcontroller are run out to the edges of the Arduino Uno board and are labeled at the connector pins. The template code uses the DIGITAL pins 8, 9, and 10 on the upper edge of the board. Get one of the 14 pin patch cables, plug half of the cable into the pins 8 through GND on the Arduino board. Then connect the other
end of the cable to your prototyping board and hook up the two LEDs and the piezo buzzer. Notice that once it is programmed, the Arduino requires no external parts to run, except power (assuming you disconnect it from the USB), and whatever you want to hook up to the I/O pins.

SETUP THE ARDUINO ENVIRONMENT AND THE DEVICE DRIVERS

In the prelab, you were supposed to have downloaded the Arduino zip file from their website. Now that you are in the lab and have an Arduino Uno board, you can go back to their website at http://www.arduino.cc and click on the ‘Getting Started’ tab. Follow the instructions to unzip the downloaded files, install the Arduino device drivers, start the Arduino programming environment, and run the Blink sketch as a test that your board is working. Many of the Arduino boards are shipped with the Blink program already installed, so try changing the blink rate to verify that your board is working properly.

RUN THE TEMPLATE CODE

Now that the Arduino is connected to the computer and the board and environment are working, it’s time to program it with our template code (AlarmArduino_template.ino) as follows:

1. If the Arduino environment is not already running, launch it, plug the board into the USB, look under the Tools>Board menu and verify that the environment is expecting an Uno board. Look under the Tools>Serial Port menu and select the correct COM port for your Arduino device (usually a COM port number greater than 3). Choose Open from the File menu, navigate to the AlarmArduino_template file, and open it. You can download the template file from our website. The environment is a bit strange in that it expects all sketch files to be located in a folder of the exact same name as the file… Go Figure. The template code should show up in an editor window.

2. There is a row of control buttons located just below the menus in the environment window. The left-most button is a ‘check’ sign and allows you to check that the code is free of arrows. Click on that button and verify that the code is in good shape.

3. Click on the ‘Upload’ button (the next-to-left-most button) to upload the code to the board. After a short delay, you should see the ‘TX’ and ‘RX’ (transmit and receive) LEDs flicker. This behavior indicates that the board is communicating with the laptop. If the upload is successful, you should see ‘Done uploading’ in the dialog box at the bottom edge of the environment, along with some information about the size of your program.

4. After a brief delay, the template code should begin execution, resulting in blinking of the two LEDs and a pleasant beeping of the buzzer. Success!
MODIFY THE TEMPLATE

Modify the template program to make a more annoying Super Alarm that does this:
1. Turn on one of the LEDs and turn off the other one.
2. Drive the buzzer at 1 kHz for 100 ms.
3. Turn off the LED that was on and turn on the one that was off.
4. Drive the buzzer at 500 Hz for 100 ms.
5. Return to step 1 and repeat forever.

As you debug your code, be sure to import the latest version of your .ino file into the Arduino environment and save it regularly. Make a print out of your final code or e-mail it to yourself so you can include it in your lab report.

DEMONSTRATE or MODIFY AND DEMONSTRATE SOME OTHER TEMPLATE

Find another program that you can either copy or download and make the Arduino do something fun or interesting. In the prelab Problem 4, you picked a program from the Learn tab on the Arduino site. That program could be a good choice.

Very quickly, you can see that there is a very wide range of things you can do with these microcontrollers. There are piggy-back boards you can buy that let the Arduino talk to the internet (remote control of anything by the internet, etc.), boards that can communicate with the GPS system (robots that know where they are and can report back to you their location), and so on.

Think about projects you would like to build and get to it!
DIP jumper to Arduino Uno. Use pins 8-14 inserted into Arduino pins 8-GND

This end of ribbon in the Arduino

This end of ribbon in protoboard.

Plug in protoboard

Figure 10.1. Alarm hardware using Arduino Uno board and its ATmega329P microcontroller