

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), 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. These are chips containing many logic gates, and in most cases also flip-flops, that can be programmed (usually once) 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 (Microchip at www.microchip.com; Motorola at www.freescale.com; Rabbit Semi at www.rabbitsemiconductor.com).

In this experiment, we will take a look at the 14-pin Microchip PIC16F676. It is a member of the Microchip PIC x14 processor family, with 1024 words of 14-bit wide program memory and 64 bytes 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 your circuit via a serial digital interface that Microchip calls ICSP (In-Circuit Serial Programming). Besides program and data memory, there are 128 bytes of non-volatile data memory that can be programmed by the processor to remember the state of your system when power is off. Nearly every pin of the chip can be programmed in multiple ways for various uses, and up to 12 of the 14 pins can be used as bi-directional digital I/O. There is an internal variable frequency clock and two separate timer/counter/scalers. For dealing with analog signals there is a comparator with a programmable voltage reference, and to top it all off, a 10-bit analog-to-digital converter with an 8-channel analog multiplexer, so you can digitize 8 separate analog signals. Amazingly, you can have all this for just over one dollar per chip, about the same cost as the 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.microchip.com). It is 129 pages long, and many aspects of the chip are only summarized there. To get the full story you have to read the PICmicro Mid-Range MCU Family Reference Manual, which runs to 688 pages! Despite this, we will show you that it's not too hard to get to know the PIC, thanks to some excellent tutorial info available from Microchip, and the excellent (and free) programming software they provide. These resources, and much else that you can find on the web, are the main reason we have chosen the PIC for this course.

A FEW WORDS ABOUT OBSELESENCE 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 gotten 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 on op-amps. Many of those part numbers have changed as manufactures 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 a single package, and 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, if you could even 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 manufacturer's 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. Read H&H Section 8.03. It is essential to understand hexadecimal representation and 2's complement arithmetic. We won't be using BCD or Gray code.
2. Everything else you will need this week can be found on our web site or at the Microchip web site. First, read the Microchip tutorial "PICmicro x14 Architecture", and then have a look at the data sheet for the 16F676, pages 1-8, and the instruction set pages 71-78. This may be all you need to read. If you have questions about the instruction set, look at the Microchip tutorial "PICmicro x14 Instruction Set".
3. (optional) PICkit1 Users Guide and other PICkit1 documentation at the Microchip web site. The PICkit1 is the hardware we will be using to program the 16F676. You can download the software for free if you like.
4. (optional) MPLAB IDE Quick Start guide and related docs at the Microchip web site. MPLAB is the programming software we will be using. The MPLAB software is also free.
5. (optional, for hard-core deviants only) PICmicro Mid-Range MCU Family Reference Manual

Theory

BINARY NUMBERS

There are several ways to represent binary numbers. Consider the 8-bit number 10110111. If you add up the place value of each set bit, you will find 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=B7h, 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.

PIC16F676 PROGRAM AND DATA MEMORY

The most important thing to know about the 16F676 is the memory organization, described on pages 7 and 8 of the data sheet. Words in the program memory are 14 bits wide and there are 1024 of them, extending from locations 0000h to 03FFh. After a power-on reset program execution starts at location 0000h.

The data memory is 8 bits (or one byte) wide and is divided into two 'banks', with bank 0 extending from location 00h to location 5Fh, and bank 1 from location 80h to DFh. This banking system reduces the number of bits needed to specify the location of data in the 14-bit machine-code instructions, so that all instructions fit into a single word. The first 32 bytes of each bank are used by 'special function registers' which control all sorts of things like I/O ports and which bank you are currently addressing. You can store program variables in the 64 locations in bank 0 at 20h (the 33rd location) through 5Fh. You can also address the 64 locations in bank 1 at A0h through DFh, but these are 'mapped' to the corresponding locations in bank 0, so they are not independent storage locations. The 128 bytes of data EEPROM (programmable, non-volatile memory) are accessed one location at a time using three of the special function registers, but we won't be using them for this experiment.

One other crucial feature of all the PIC processors is the 8-bit W register or 'working' register. It does not have an address like all other data registers, but many of the instructions interact with it.

Generally, data has to be moved into the W register before it can be moved elsewhere or operated on.

PROGRAMMING AND CODE TEMPLATE

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 PICs is the availability of template code (at the Microchip site, in books about the PIC, and all over the internet). The template for our experiment is the file Alarm16F676.asm 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. 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, which is bit 0 of I/O port C. Power and ground are pins 1 and 14, while pins 13, 12, and 4 are used for the in-circuit serial programming. The template code will turn both LEDs on and send 100 cycles of a 1 kHz square wave to the buzzer, then turn the LEDs off and do nothing for 100 ms, and then repeat these steps forever.

This first thing you will notice about the template is that it is not a list of 14-bit binary numbers, even though that is what we need to program the 16F676. Instead, it is written in assembly language, a set of mnemonics like **movlw D'200**, an instruction which corresponds to H'30C8', or B'11000011001000'. (This instruction moves the decimal number 200=C8h into the W register. Read it as 'move literal into W, decimal 200'. A literal is a constant value specified in the instruction itself.) We will use a program called an assembler to translate our assembly language template (file extension .asm) into the binary file (file extension .hex) needed to program the PIC. (Actually the assembler produces an ASCII hexadecimal file, which is then translated to binary by the programmer).

You will also notice that most of the text in the template is not processor instructions at all. Everything that follows a semicolon on a line is a comment that helps to explain what is going on. When you write code it should include as many comments as our template does. We will not worry about all of the details in the template that configure the processor for our application, but do notice the **#define** statements. These define symbols that represent binary numbers, so that we can refer to an I/O pin with an LED connected to it as LED1, instead of by a binary code.

This particular symbol is defined in terms of another symbol **PORTC**, which stands for the address 07h of the PORTC special function register (see Fig. 2-2 on page 8 of the data sheet). Many standard symbols are defined in the ‘include’ file p16f676.inc, so you don’t have to know the address of all the special function registers, you can instead refer to them by name. The statements between **cblock** and **endc** define symbols that stand for addresses of variables in the data memory, so these can be referred to by name also.

The first three instructions are needed to jump past the starting location for handling hardware interrupts, which we won’t be using. Next is the initialization routine, which calibrates the internal oscillator, configures the I/O ports, and turns off the analog comparator. We do not need to worry about these details.

The main program starts with three **bcf** instructions, which stands for ‘bit clear file register’. A file register can be either a data memory location or a special function register, and this instruction can clear (set to zero) any bit in any file register. The symbols **BUZZ**, **LED1**, and **LED2** have been defined so that these instructions turn off the two LEDs and the buzzer output. We enter the main program loop at **Mainloop**, and turn on both LEDs with the **bsf** (bit set file register) instruction. The next instruction we have already discussed, it moves D’200’ into the W register, so that the following instruction **movwf counter** can initialize the variable **counter** to D’200’. Read **movwf** as ‘move contents of W register to file register’. The next two instructions load the W register with a predefined bit pattern, and then use it to toggle the I/O port bit that goes to the buzzer. If you look at the toggle mask bit pattern you will see that it picks out bit 0. The **xorwf PORTC,f** instruction is read as ‘exclusive or W with PORTC, and put the result in the PORTC’. If the destination f in this instruction were replaced by w, it would put the result in the W register instead.

The next instruction **call timer** calls a time delay subroutine that is defined later in the code. The subroutine just executes a loop for 0.5 ms, and then returns. We have set up the processor to run with a 4 MHz internal clock, and it takes four clock cycles per instruction, so a 0.5 ms delay takes 500 processor instructions. After the delay comes a **decfsz counter,f** instruction which decrements the **counter** variable and then skips the next instruction if it reaches zero. If it has not yet reached zero the processor executes a **goto** back to the location **Buzzloop1**, and the loop repeats, thus sending a 1 kHz square wave to the buzzer. After 200 cycles of the loop (100 cycles of the square wave) the **counter** variable reaches zero and the **goto** instruction is skipped.

At this point you can probably see what the rest of the code does. First the LEDs are turned off,

and then another loop is entered as before, but this time the buzzer I/O pin is not toggled, so the buzzer will be silent. After 100 ms, the program returns to `Mainloop`, and the cycle continues forever.

We have now explained all of the programming you will need for this experiment. Besides **call** and **goto**, we only use five instructions: **movlw**, **movwf**, **bsf**, **bcf**, and **decfsz**. These are also the only instructions used in the subroutine, except for **return**, which just returns from the subroutine call. Look up our five instructions on the data sheet and see if you can make sense of what it says.

Pre-lab Problem

Read through this entire write-up, and the required readings listed above. Then, write the assembly language code you will need to make the ‘super buzzer’ described below. You can write the code on paper by hand, use a text editor, or download the MPLAB IDE, create an assembly file (.asm) and try to compile it into a .hex file. Whatever you do, make sure you bring a paper copy of your assembly code to the lab with you.

New Apparatus and Methods

PICKit1 PROGRAMMER

The PICKit1 programmer 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 contains an empty ‘evaluation socket’ where you can plug in an 8-pin or 14-pin flash memory PIC that you would like to program. It also contains a PIC16C745 which is used to implement the USB interface. There is extensive documentation for the PICKit1 at the Microchip site, including the USB interface code that runs on the 16C745.

If you wanted to build a circuit for a PIC running in the evaluation socket, you would have to solder components onto the PICKit1. We don’t want to do this because the PICKit1 costs \$36 and we want to be able to share and re-use them. Instead, we will connect a 14-pin DIP jumper from the PICKit1 evaluation socket to our prototyping board, and build the circuit on the proto board in our usual way. The only wires from the PICKit1 we will use are power, ground, and the three lines needed for in-circuit serial programming. Once you have your circuit programmed and running, you can disconnect the PICKit1 programmer. You will then need to provide power to your circuit with an external +5 V supply. Be careful not to exceed +5.5 V even for an instant,

or you will destroy the PIC. Also, be careful you don't try to use an external 5 V supply at the same time that you have power from the PICKit1 connected to your circuit, as this can destroy the voltage regulator on the PICKit1 board. Your code will begin to execute as soon as power is applied to your circuit.

PIEZO BUZZER

We will use a piezoelectric buzzer to get a little sound out of our PIC circuit. 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 PIC through a transistor driver to make it louder.

Experiment

THE CIRCUIT

A schematic of the circuit we will build is shown in Figure 1. Be sure to put your LEDs in with the correct polarity. Pins 1,4,12,13,14 on the 16F676 are jumpered to the Evaluation Socket on the PICKit1. (1 and 14 are +5V power and ground, 13 and 12 are serial data and a clock for programming, and 4 is +13V for programming.) Notice that once it is programmed, the PIC requires no external parts to run, except for a bypass capacitor for the power supply, and whatever you want to hook up to the I/O pins. This is a huge simplification compared to earlier microcontrollers. (Do not use a bypass capacitor larger than 0.1 uF when programming or the PICKit1 will think your chip is not connected. We are not sure why this happens, but it probably has to do with the rate that the supply voltage rises when the PICKit1 applies power to your chip.)

RUN THE TEMPLATE CODE

Once your circuit is assembled, use a 14-pin DIP jumper to connect it to the Evaluation Socket on the PICKit1. Be careful that the jumper plugs have pin 1 in the correct place so that pin 1 on the evaluation socket is connected to pin 1 on your 16F676, and similarly for pins 4, 12, 13, 14. Then plug the PICKit1 into a USB port on a PC and try to program the 16F676 with our template code (Alarm16F676.asm) as follows:

1. Launch the MPLAB IDE, choose Open from the File menu, navigate to the Alarm16F676.asm template file, and open it. The assembly language code should show up in an

editor window within the main MPLAB window.

2. Chose Select Device under the Configure menu and select the PIC16F676.
3. Click on the editor window and then choose Quickbuild Alarm16F676.asm under the Project menu. This should assemble the template code and generated a .hex file. An output window should open with some messages; the last line should say BUILD SUCCEEDED. There will also be warnings about being in the correct memory bank.
4. Choose Disassembly Listing under the View menu, and look at the listing that appears. In the left two columns is the hexadecimal code generated by the assembler. If you ever want to see exactly what the assembler is doing, you can look here.
5. Now minimize the main MPLAB window, and launch the PICkit software. The green LED on the PICkit1 board should now be on, and you should be able to turn the yellow LED on by clicking the Device Power check box. The 2.5 kHz Osc check box should be off.
6. Choose Import HEX under the File menu, navigate to the Alarm16F676.hex file that you created above, and open it. The Program Memory window should fill up with hex numbers. Click the Write Device button, and your device should be programmed and start to run. You can turn the power to your PIC chip on and off using the Device Power check box.

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 .hex file into the PICkit software before you write it to your chip. You can use the PICkit software to read the program code in your device (Read Device button), and it can check that the programmed data is the same as the most recently imported .hex file (the Verify button). Make a print out of your final code or e-mail it to yourself so you can include it in your lab report.

Disconnect the PICkit programmer from your circuit, connect a +5 V power supply, and verify that it will operate stand-alone.

