Lab 7: Introduction to Microcontrollers

1 Microcontroller Advantages

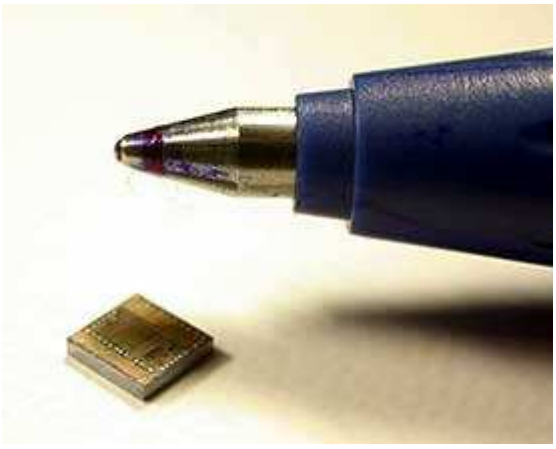
Microcontrollers are widely used in today's control systems for the following reasons:

- **Design and Simulation** – Because you are programming with software, detailed simulations may be performed in advance to assure correctness of code and system performance.
- **Flexibility** – Ability to reprogram using Flash, EEPROM or EPROM allows straightforward changes in the control law used.
- **High Integration** – Most microcontrollers are essentially single chip computers with on-chip processing, memory, and I/O. Some contain peripherals for serial communication and reading analog signals (with an analog-to-digital converter or ADC). This differentiates a microcontroller from a microprocessor. Microprocessors require that this functionality be provided by added components.
- **Cost** – Cost savings come from several locations. Development costs are greatly decreased because of the design/flexibility advantages mentioned previously. Because so many components are included on one IC, board area and component savings are often evident as well.
- **Easy to Use** – Just program and go! While in the past, programming has often involved tedious assembly code, today C compilers are available for most microcontrollers. Microcontrollers often only require a single 5V supply as well which makes them easier to power and use.

2 History

Microcontrollers were first considered at Intel in 1969 when a Japanese company approached Intel to build some integrated circuits for calculators. Marcian Huff used his previous experience on the PDP-8 to propose an alternate solution – a programmable IC. Frederico Faggin transformed this idea to reality and Intel bought the license from the Japanese company (BUSICOM) to create the 4004 4-bit microprocessor capable of 6000 operations per second. This was soon followed by the 8-bit 8008 in 1972. Intel's efforts were soon followed by Motorola with the 8-bit 6800 series and MOS Technology introduced the 6501 and 6502 for only $25 each. It was all downhill from there.

The microcontroller you will use in this lab can be purchased for $3.50 in quantities of 100, can run up to 16 MHz, contains 16 Kbytes of programmable Flash memory, 512 bytes of EEPROM, and 1 Kbyte of SRAM, as well as including many peripheral capabilities like an ADC, serial communication, PWM, and JTAG debugging capabilities. Sweet.

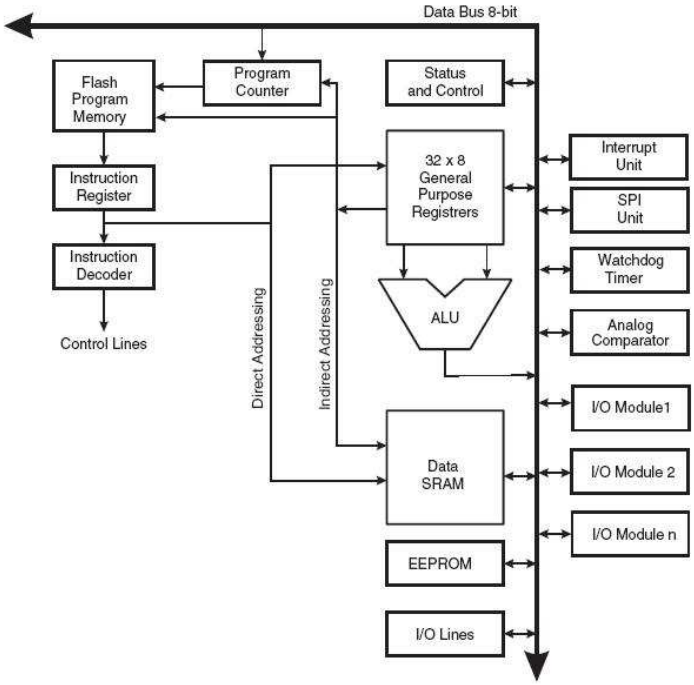


A single chip microcontroller + radio built by Jason Hill at UC Berkeley for use in sensor networks.

A single chip microcontroller + 900 MHz radio (known as "Spec") was designed by Jason Hill at UC Berkeley for a new field in engineering known as "sensor networks." Because microcontrollers and radios are so cheap and easy to use today, you could fill a building with thousands of them, all talking together over low power radios. By sensing their environment, these small sensors could control lighting, temperature, as well as numerous other environmental controls.

3 What's Inside a Microcontroller?

Our microcontroller, the Atmel ATmega16 integrates memory, clock, a central processing unit, input/output, timers, and an analog to digital converter.



ATmega16 Architecture.

3.1 Memory

Memory on a microcontroller can be used to store data and/or the program to be run. There are often several types of memory on a microcontroller:

- Random Access Memory (RAM)
- Read Only Memory (ROM)
- Programmable Read Only Memory (PROM)
    - Erasable Programmable Read Only Memory (EPROM)
    - Electronically Erasable Programmable Read Only Memory (EEPROM)
    - Flash Memory – a type of EEPROM

RAM can be either read or written, and this usually happens quite fast. Data stored on a microcontroller is often stored in RAM. However, the data stored in RAM is volatile which means that it is lost when power is turned off. ROM is non-volatile and therefore stored between power cycles, but may not be written to.

PROM is therefore a compromise between these two types of memory. PROM is non-volatile and also allows a user to program it at least once and possibly erase it. Some PROM may be erased by exposure to UV light, but more common today is EEPROM. EEPROM allows read and write access and is also non-volatile, but the sacrifice here is that data transfers take much longer than with RAM (order of milliseconds v. microseconds).

Flash memory is a type of EEPROM. Program memory (where the program is stored) on the ATmega16 is Flash memory. This is also the same as the memory used in digital cameras and cell phones. Data transfer using flash is much faster than EEPROM because it works in blocks of bytes instead of single bytes. This makes it perfect for program memory in our case.

3.2 Clock

The ATmega16 we use is run off an 8 MHz crystal oscillator. The rate of instruction execution is fixed and synchronized by this clock. However, this does not mean that each instruction takes 125 nsec. Different instructions require a different number of cycles.

3.3 CPU

This is brains of the microcontroller – the CPU executes instructions such as add, move, jump, multiply, etc. To do so, it must first fetch the instruction and any required data over its data bus.

3.4 Input/Output (I/O)

The ATmega16 offers 32 programmable I/O lines with 4 8-bit ports. By programming specific registers on the ATmega16, these lines may be set to input, output, or some

secondary function. If a pin is set as output, setting the corresponding bit in the output register to 1 will output Vdd on that pin and 0 will output ground. If the pin is set to input, it is possible to read either a 1 or 0 on that pin. These pins act just like memory locations so all that is required to output a value is setting a bit in a memory register. To read a pin, all you need to do is read a bit in a register.

More information on secondary functionality for I/O pins may be read in the ATmega16 datasheet.

3.5 Timers

Timers are internal clocks (2 8-bit timers and 1 16-bit timer is included in the ATmega16). Each timer can be scaled by some factor from the system clock (8 MHz as mentioned previously). These timers can then give us a sense of time and duration – information of great importance in digital control systems. In most cases, you'll just use a timer to count from 0 to 255 (for an 8-bit timer) or 0 to 65536 (for a 16-bit timer).

In addition, many interrupts can be triggered off of timers. An interrupt is a piece of code triggered by a particular event. That event might be a timer overflowing, or reaching a particular value.

3.6 Analog-to-Digital Converter (ADC)

In most cases, the real world gives us analog signals. Reading light levels from a photoresistor as in Labs 5 and 6 will give us an analog voltage relating to the current light falling on the photoresistor. For the microcontroller to deal with this information, it must be converted to a digital format. An analog-to-digital converter (ADC) does exactly that.

The ATmega16 provides an 8-channel 10-bit ADC. The number of channels is the number of pins supporting the ADC functionality (this is one of the secondary functions mentioned earlier in I/O). The number of bits tells us the resolution with which we can read the analog data. Given a 5V supply, the least significant bit gives us

$$\frac{5V}{2^{10}} = 4.9mV$$

This means that we can see the difference between 4.9 mV and 9.8 mV.

4 Introducing CodeVision, the ATmega16, and the STK500 Development Board

Now that you have the basics of what a microcontroller is and is capable of, let's put it to good use by learning how to use it.

We'll be using the CodeVision C compiler, so any code you write will be done here. The STK500 is a development board designed by Atmel as an evaluation board for its

microcontrollers. A development board generally includes LEDs to check the I/O functionality of pins, as well as headers to connect anything else you'd like. We'll be connecting a separate board designed to control a motor later, but it will connect to the port headers on the STK500. For more information on what the STK500 contains, take a look at the User's Guide. The ATmega16 plugs into the STK500 and therefore all of the functionality that the STK500 offers is connected to the ATmega16.

4.1 Hardware Setup

First, everything must be plugged in. For now we'll just be using the STK500, so set everything else aside. You'll need a 12V DC supply to plug into the power port and a serial cable from the PC to plug into the RS232 CTRL port. Make sure you note which COM port you're attached to on the PC.

You can also look at the STK500 to confirm that you have an ATmega16 microcontroller plugged into SCKT3100A3 and an 8 MHz crystal in the Crystal port.

Finally, the first 3 programs will require use of the LEDs. Connect the PORTB header to the LED header so that PORTB of the ATmega16 will be connected to the LEDs.

4.2 CodeVision

Next, open the CodeVision AVR C Compiler. This is the development environment and C compiler we will be using to program the ATmega16. We'll start off by creating a new project – HelloWorld. The microcontroller equivalent of "Hello World" is to blink an LED.

Go to *New -> Project* and DO NOT use the CodeWizardAVR. The CodeWizard will provide skeleton code for a lot of functionality and while convenient for large projects, it will just be more complicated than necessary for us.

Name your project something descriptive like *helloworld.prj* (it is recommended that you save this information to your home directory).

It should pop up a box asking for your helloworld project configuration. Let's first make sure it knows which microcontroller we're using.

Click on the *C Compiler* tab and the *Code Generation* sub-tab. Under "Chip", select ATmega16 and make sure the "Clock" is set to 8 MHz. Everything else should be fine as default. Say "OK" to continue.

Next we'll set the COM port. Go to *Settings -> Programmer* and select the COM port you're using on the PC to talk to the STK500. Make sure the "AVR Chip Programmer Type" is set to Atmel STK500/AVRISP as well. Say "OK."

Hardware and software should be ready to go now!

4.3 Hello World

Now that everything is set up, we can write a program to start blinking some LEDs. First, let's add a file to the project. This will include the "main" function which indicates the beginning of every program. Go to *New -> Source* to create a new C file and save this as *helloworld.c*.

Next, this file must be added to the project. Go to *Project -> Configure* and under the *Files* tab, click "Add." Find the *helloworld.c* that you just saved and say "OK."

Finally, we can start writing code. It's good to start off every program with a commented section that gives a description of the program. In our case, we're going to have an LED blink on and off.

Since we're going to be using register names defined specifically for the ATmega16, you'll also want to include the header file that defines the names for each of these registers.

```
#include <mega16.h>
```

This file is included on the website if you'd like to see what's in it.

Now let's start with our "main" function.

```
void main(void) {

    // Add code to blink LEDs here

}
```

This function will execute when the microcontroller is reset so this is where we'd like to add the code to make the LED blink. For now, we'll do this with a very simple while loop and move on to more complicated methods later.

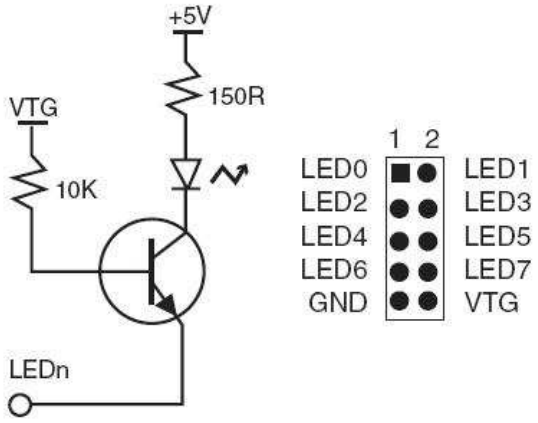
```
while (1) {
    // Toggle LED

    delay = 100000;
    while (delay > 0)
        delay--;
}
```

Using a 1 as the argument for the main while loop means that this program will run forever which is what we'd like a microcontroller to do. The second while loop just provides some delay so that you can actually see the LED blink. If this was missing, the

LED would blink too quickly for you to see. Be sure to declare your delay variable (as a long because it's so big!) somewhere above.

Finally, we need the code to actually toggle the LED. You'll probably want to create a variable to keep track of the LED status (on or off), or you could read this value before changing it each time. If you check the User's Guide for the STK500, you'll notice that the LEDs are connected to PORTB and the circuit driving them looks like the following.



LED Circuit on STK500.

This means that to turn off the LED, you need to output a 1 (or Vdd). To turn it on, you need to output a 0. To set all of the bits of PORTB high, you can use the command

PORTB = 0xFF;

PORTB is 8 bits wide, so to access a specific bit, use this C compiler uses the syntax:

PORTB.bit# = 1;

Make sure in your code that all of the LEDs are turned off to start.

You can read the value that a port or bit is set to by just treating the register name as a variable.

ledStatus = PORTB.0;

All of the I/O ports on the ATmega16 default to being inputs. For this reason, we also need to set PORTB to be all outputs. This information is controlled by the port's direction control register (DDRB). This information can be found in the ATmega16 datasheet. To set a pin to output, write a 1 to the DDR*x* register. For now, let's set all of PORTB (all of the LEDs) to output pins.

DDRB = 0xFF;

You should now have all of the pieces necessary to write a program to toggle the LED on bit 0 continuously.

Once you have written this program, it is necessary to compile it and download it to the microcontroller. To compile, go to *Project -> Make*. A window should pop up to give a summary of the compilation – make sure you have no errors or warnings before continuing.

To download your program to the microcontroller, make sure the serial cable is connected, the board is plugged in and turned on (there's a power switch next to the plug), and select *Tools -> Chip Programmer*. Make sure the correct chip is selected and click "Program All." When it asks you if you'd like to load a file to EEPROM, select "NO."

4.4 Timing Interrupts

Now we'll move on to slightly more complicated topics. A very important structure in writing software for microcontrollers is the interrupt. Interrupts are linked to hardware events (a timer overflowing, ADC returning data, the signal on an input pin changing, receiving serial data, etc) and jump the program to a piece of code designed to handle such an event. It is generally a good practice to keep this bit of code small so that you don't miss other interrupts happening in the system. If you are currently servicing one interrupt, it is possible to miss or delay other interrupts.

For now, we'll just look at using a timer interrupt to toggle the LED at a fixed interval of once per second (at 1 Hz). It is recommended that you create a new project for this section.

Interrupts require some setup to tell the microcontroller that you want the interrupt to occur in the first place. All of the registers and bits that need to be set up are described in the ATmega16 datasheet, but are reviewed here.

7	6	5	4	3	2	1	0
FOC0	**WGM00**	**COM01**	**COM00**	**WGM01**	**CS02**	**CS01**	**CS00**

TCCR0 register to control Timer0.

In this register, the bits we are most interested in are the CS*xx* bits (bits 0-2). More detail on the other bits in this register can be found in the datasheet. These control a pre-scaler for the timer. While our system clock is set at 8 MHz, it's possible to run the timer slower than that by setting these bits according to the table below.

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer is stopped)
0	0	1	CLK/1 (No prescaling)
0	1	0	CLK/8
0	1	1	CLK/64
1	0	0	CLK/256
1	0	1	CLK/1024

Therefore, if the Clock Select bits are set to 011, the timer will increment every 1/(8 MHz/64) seconds or 1/125kHz or 8 usec.

TCCR0 = 0x03;

This timer will overflow every 256*8usec = 2.048 msec. The 256 comes from the fact that Timer0 is an 8-bit timer – Timer1 is a 16-bit timer and could give you a longer interval before overflowing if that was important to you.

The next register of importance when using Timer0 is the register that actually contains the current Timer value. This register may be read or written to.

TCNT0

TCNT0 register.

Finally, we are interested in the TIMSK register, which controls turning on and off timer interrupts.

7	6	5	4	3	2	1	0
OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0

TIMSK register for timer interrupts.

The shaded bits are used for other timers, and the unshaded bits correspond to Timer0. The OCIE*xx* bits correspond to "Output Compare" interrupts. In addition to the overflow interrupts already mentioned, the timer can be set to interrupt when it reaches a particular value stored in a separate register (OCR*xx*). More information on output compare functionality can be found in the datasheet. The TOIE*x* bits correspond to overflow interrupts. If you'd like the timer to interrupt when it overflows, set this bit to a 1.

TIMSK.0 = 1;

However, setting this bit alone is not enough to make the interrupt happen when the timer overflows. There is another bit that controls all system interrupts. Because this bit is so widely used, it has its own instruction associated with it.

#asm("sei")

To clear the bit and turn off all system interrupts, use

#asm("cli")

The #asm macro tells the CodeVision compiler that this command is in assembly code.

Finally, you must specify the code that you want the program to jump to when you interrupt. This is defined by the interrupt keyword in the CodeVision compiler:

interrupt [*INTERRUPT_NAME*] void *function_name*(void) {
// Interrupt contents go here
}

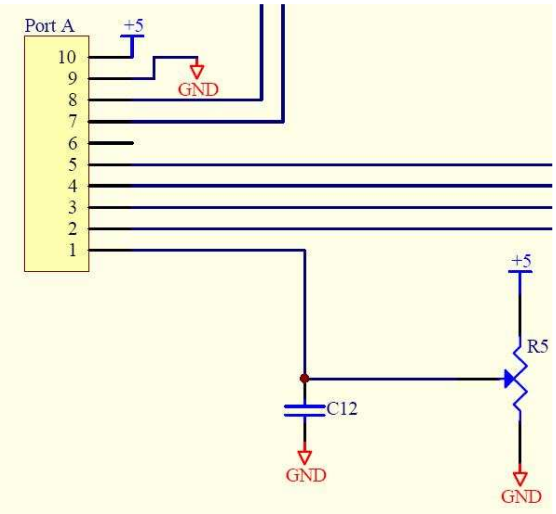
The interrupt names that correspond to specific hardware interrupts can be found in the Mega16.h file linked on the website. For example, the Timer0 Overflow interrupt is TIM0_OVF. You may use whatever descriptive function name you'd like.

Now that you've gotten a glimpse at how interrupts work, write a program to blink an LED at 1 Hz.

4.5 Reading Analog Inputs

A capability of the ATmega16 that makes it very useful in sensing applications is the Analog to Digital Converter (ADC). The ADC channels provided on the ATmega8 are located on the PORTA pins, which indicates that they are connected to the PORTA header on the STK500 board. Make sure the PORTA header is connected to the corresponding header on the Motor Control Board. In this section, we're going to read an analog input (the potentiometer on the Motor Control Board) and display its output using the LEDs. It is recommended that you create another new project.

The potentiometer on the Motor Control Board is setup as follows. This can also be seen on the Motor Control Board schematic on the website.



Potentiometer Setup from Motor Control Board Schematic.

PORTA, pin 1 on the STK500 is connected to the ATmega16 as PORTA, pin 0. From the circuit, it should be clear that by turning the potentiometer, the voltage will swing from 0 to +5 V. Because the ADC on the ATmega16 is 10 bits, we should be able to read values from 0 to 1023.

As always, there is some setup of registers necessary to use the ADC. The ADMUX register determines which channel of the ADC to look at (0 in the case of the potentiometer).

7	6	5	4	3	2	1	0
REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0

ADMUX register which determines which ADC channel to sample.

There are 4 mux bits for 8 channels because they may also be used differentially (see datasheet for more details). We are only interested in single-ended input and will therefore only require the use of bits 0-2. These should be set to the desired channel. To read the input on channel 3,

ADMUX = 0x03;

The next register required is the ADC Control and Status Register (ADCSRA).

7	6	5	4	3	2	1	0
ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

ADCSRA ADC Control and Status Register.

The ADEN bit enables the ADC for use and must be written to 1. Writing a 1 to ADSC starts a conversion. However, because this conversion takes some time, we will use another interrupt to indicate to us that the conversion has finished. Setting ADIE to 1 enables this interrupt (ADC_INT). The ADPS*x* bits define the prescaler used for the ADC clock similar to the prescaler bits for the timers above. You can check the datasheet to determine an appropriate value for these bits.

Now that you have enabled the ADC and its interrupt, and started a conversion, you need to be able to read the results of that conversion. The ADC value sampled is stored in the ADCH and ADCL registers (2 registers are needed because the value is 10 bits long). When reading the ADC, the CodeVision compiler also allows the use of ADCW to read in the full 16-bit ADC value.

15	14	13	12	11	10	9	8
						ADC9	ADC8
7	6	5	4	3	2	1	0
ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0

ADCH (bits 8-15) and ADCL (bits 0-7) store the ADC value from the most recent conversion.

You should now have all of the information required to use the ADC on the ATmega16. Write a new program to read the input from the potentiometer on the Motor Control Board and display its value on the LEDs. You can do this one of 2 ways – display the 8

most significant bits on the LEDs, or use the LEDs as a bar meter (turning on increasing numbers of LEDs for an increasing value from the potentiometer).

4.6 Using the DAC on the Motor Control Board

There are a couple other components you will need to use on the Motor Control Board to write the servo controller for the next lab. One of these components that will be extremely useful for debugging and is used to control the motor is a Digital to Analog Converter (DAC). The DAC included is an Analog Devices AD7247, which contains two 12-bit DACs (datasheet is on the website). This chip takes a parallel input as seen on the Motor Control Board schematic, uses the CSA and CSB pins (active low) to determine which DAC the inputs are connected to. The chip is currently configured to provide output between –5V and 5V. All of this information can also be found in the AD7247 datasheet and Motor Control Board schematic.

Because the DAC inputs are connected to ports B and C on the ATmega16, make sure that you designate these ports as outputs and physically connect them from the STK500 to the Motor Control Board. In general you will want to keep the CSA and CSB pins high and pulse them low when you want to write new data to the DAC. Since you are writing bits to PORTC as well as controlling the DAC select pins from PORTC, make sure you don't change one when sending data to the other.

Write a program to output the analog data you were reading from the potentiometer previously on the DAC and display this voltage on an oscilloscope or multimeter. Use DAC B for now and set the DAC input so that your analog input is on the most significant bits of the DAC and you can see the entire range of voltages.

You should now be sufficiently fluent in microcontroller basics to design your servo controller for the next lab. Congratulations!