MODULE 10

MICROCONTROLLERS II

PREREQUISITES: MODULE 09: MICROCONTROLLERS I.

OUTLINE OF MODULE 10:

What you will learn about in this Module:

Additional information on using C compilers
Watch Dog Timer (WDT)
How to write, compile, and debug your own source code

What you will build in the lab:

You will modify the circuit you built in Module 09 to add a few components. Then you will use a microcontroller to carry out the logic functions that you studied in an earlier module (Module 07: Digital Circuits)
You will then program your device to function as a "reaction timer"

INTRODUCTION:

The versatility of microcontrollers is almost beyond belief. You can use them to perform almost any task that can be done using other electronic components. In this module, you will begin to learn more about programming the PIC16F84A microcontroller (from Module 09), as well as how to interface the microcontroller with switches from the outside world. You will then use the microcontroller to duplicate the function of the logic circuit that you built in Module 07, but this time the logic will be implemented in *software*, instead of hardware.

READINGS FROM HOROWITZ AND HILL (H&H): *ART OF ELECTRONICS*

11.13 (different types of microprocessors)
11.14 (Emulators & development systems)

ADDITIONAL READINGS & INTERNET RESEARCH:

You will be using the HELP files that are available with the PIC-C Compiler to learn several very useful commands.

GENERAL NOTES ON WRITING CODE FOR A MICROCONTROLLER:

Programming a microcontroller is actually quite easy if you have a compiler handy (so that you can write your source code in a common language, like "C" or "BASIC", instead of having to write in the much more cryptic machine codes or assembly languages). In general, when developing code for a microcontroller, you will go through several stages. This is recommended to avoid wasted time trying to fix code that was poorly designed in the first place. Here is an outline of how you should proceed when developing microcontroller code:

1- Define clearly what you want to accomplish. This could be a simple bulleted list of things that the system must do.

2- Write "**pseudo code**". This can take several forms, but it is essentially an outline of the code you are about to write. It can be a flow chart, a block diagram, or even an ordered list of actions, each with a specific line number (see example below). The more detail you include in this step, the easier it will be to write your source code in the next step.

3- Then, you write the **source code**. This is typically in a standard language like "C" or "BASIC"

4- You compile the source code into **HEX** (hexadecimal) **code**. In this series of modules this is done using the PIC-C compiler. Hexadecimal code is really just a series of numbers in base 16 (with digits 0 – 9, then A, B, C, D, E, F). So, the decimal number 15 can be represented as 1111 in binary (base 2) or as the digit "F" in hexadecimal. The programmer just uses hexadecimal code as a shorthand for binary (1's and 0's), that are burned onto the microcontroller.

5- The programmer for the microcontroller understands HEX code, so you just download the HEX code to the programmer (as in Module 09).

6- You burn the HEX code onto the microcontroller (as in Module 09).

An example of this process follows:

To begin with, assume you have three pushbutton switches attached to the microcontroller circuit that you built for Module 09, as shown a few pages below. You should go ahead and build this circuit, since you will be using it for the next part of this Module. All you need to do is add three push button switches and five resistors to the circuit you built for Module 09. Use a marker to label each switch on your PC Board (SW1, SW2, and SW3). Each 1 kΩ resistor will hold the input to the corresponding pin on the microcontroller at GROUND (0 volts, = logical FALSE), until you press the switch. When you press the switch, the corresponding pin on the microcontroller will be driven to +5V (logical TRUE). Since we are not using RA3 and RA4 in this module (pins 2 and 3 on the microcontroller), we will just pull those down to GROUND using a 1 kΩ resistor on each.

Example: you wish to run a simple program that will perform the following functions: if you press SW1 only, the LED connected to RB0 will turn ON; if you press only SW2 the LED connected to RB1 will turn ON, if you press both switches at the same time, all 8 LEDs will flash ON and OFF twice per second. No switches pressed = no LEDs are turned ON.

Step 1: Define what you want to accomplish (this is already stated clearly in the above paragraph)

Step 2: Pseudo code

Set PORT B to all OUTPUTS (this will be used to control each LED)

Set PORT A to all INPUTS (this will be used to detect each switch)

Setup watch dog timer (WDT) (the internal timer prevents hang-ups)

Initially set output_B = 0b00000000 (explicitly set each bit of port B to zero)
(This will turn off all LEDs)

Loop through the following cycle infinitely:

INPUT data from PORT A (this will check all switch states simultaneously)

x = Input A

if x = 0b00000000 then turn OFF all LEDs (no switches are pressed)

if x = 0b00000001 then turn ON RB0 (only SW1 is pressed)

if x = 0b00000010 then turn ON RB1 (only SW2 is pressed)

if x = 0b00000011 then flash all LEDs on and off (both switches are pressed)

restart WDT (restart the watch dog timer)

return to beginning of infinite loop

There are several ways to write pseudo code, this is just one example. Your choice of approaches will depend upon the nature of the program: if the program always just marches straight through a sequence of actions, then a bulleted or numbered list is perfect, if your program will simply branch straight through a series of logical decisions, then a logic tree might be the best way to go, if your program will repeatedly cycle through a lot of conditional branches and complicated decisions that could send the program execution in many different directions on each cycle, depending on changing inputs, then you are better off using a flowchart or a block diagram with arrows showing what to do for each possible input at each point that a decision is made.

Step 3: Source Code (you write this using the C compiler)

Using the C compiler, you could write source code that would follow exactly what you had outlined in the pseudo code above. I have written this source code for you. It is on the following page in a program called “mod10a.c”. You can also find this code on the course web page (just click on the link), so you can easily burn it onto a microcontroller to test your circuit if you like.

```
//         mod10a.c      RGD 1/4/03
//  This is a simple test program for module 10 (ME499)

#include <16f84a.H>
// this tells the compiler that we will use a 4 MHz crystal
#fuses HS, WDT, NOPROTECT
//  This sets up the internal device fuses: HS crystal, WDT ON, Code Protect OFF
#use delay (clock=4000000)
#byte PORTA = 5
#byte PORTB = 6

unsigned int x;  // this defines an 8-bit unsigned integer variable, x

////////////////////////////////////////
// SUBROUTINES
void init_ports(void) {
   SET_TRIS_A(0b11111111);  // PORTA = all outputs
   SET_TRIS_B(0b00000000);  // PORTB = all inputs
//  set Watch Dog Timer prescaler to 1152 ms:
   SETUP_COUNTERS(RTCC_INTERNAL, WDT_1152MS);
// Default output values:
   PORTB = 0b00000000;   //  turn off all LEDs
// Default variable values
   x = 0b00000000;
}

////////////////////////////////////////
// PIC16F84a microcontroller goes here at RESET

void main() {
      init_ports();  // Initialize ports
      restart_wdt(); // Reset the WDT

 CYCLE:  // Run continuously
   x = input_a();    // read (input) all of the bits from port A into the variable x
      //NOTE: Port A only has five I/O pins, the remaining 3 bits are always = 0

   if(x == 0b00000000) OUTPUT_B(0b00000000);
      // No buttons are pressed, so turn off all LEDs
      // note also the use of == as opposed to a single =
      // two == signs means "are the two values equal?"
      // a single = sign means "set the left variable to equal the value on the right"

   if(x == 0b00000001) OUTPUT_B(0b00000001);
      // Only SW1 was pressed, so turn ON the LED connected to RB0

   if(x == 0b00000010) OUTPUT_B(0b00000010);
      // Only SW2 was pressed, so turn on the LED connected to RB1

   while(x == 0b00000011) {  // if both SW1 and SW2 are being pressed...
     OUTPUT_B(0b11111111);   // turn ON all LEDs
     delay_ms(250);          // wait for 1/4 of a second
     OUTPUT_B(0b00000000);   // turn OFF all LEDs
     delay_ms(250);          // wait for 1/4 of a second
     // NOTE: the above 4 lines will cause all of the LEDs to flash
     // for as long as both SW1 and SW2 remain pressed
     restart_wdt();
     x = input_a();    // check to be sure both SW1 and SW2 are still pressed
   }  // this is the end of the WHILE loop

 goto CYCLE;  // go back and infinitely loop to the label CYCLE
} // main…this is the end of the program
```

Step 4: Compiled Code (this is in HEXADECIMAL, generated by the C compiler)

Once your source code is written, you press the "compile" button (see Module 09) and the compiler will generate several files based on your source code. The most important one for you to think about is the *.hex file, which you will import into the chip programmer and burn onto the microcontroller, just as you did in Module 09. The resulting *.hex code for this program will be called mod10a.hex, and it looks like this:

```
:10 0000 0000 308A 0033 2800 00FF 3065 0000 3066 00B1
:10 0010 000E 308C 008C 1D14 2807 3081 0181 3084 0043
:10 0020 0000 08C0 390F 3880 0064 0081 3084 0000 0867
:10 0030 00C0 390C 0480 0086 018E 0137 280F 3084 00FF
:10 0040 0000 0803 1932 2801 308D 008C 018C 0B26 2802
:10 0050 008D 0B25 284A 308C 008C 0B2C 2800 0000 00CA
:10 0060 0080 0B23 2800 3484 011F 3083 0504 2864 009A
:10 0070 00FF 3083 1685 0083 1205 088E 008E 0803 1D4D
:10 0080 0046 2800 3083 1686 0083 1286 010E 0B4E 2808
:10 0090 0000 3083 1686 0001 3083 1286 000E 0802 3C71
:10 00A0 0003 1D58 2800 3083 1686 0002 3083 1286 0014
:10 00B0 000E 0803 3C03 1D75 2800 3083 1686 00FF 30B0
:10 00C0 0083 1286 00FA 308F 001E 2000 3083 1686 00CF
:10 00D0 0083 1286 01FA 308F 001E 2064 00FF 3083 16E1
:0E 00E0 0085 0083 1205 088E 0058 2838 2863 001A
:00 0000 01FF
;PIC16F84A
```

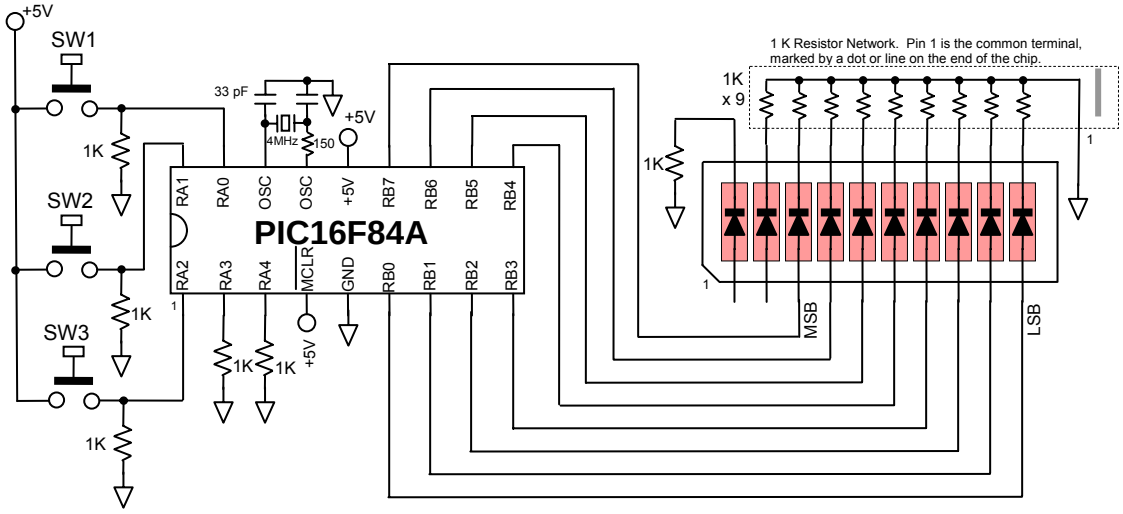
The above *.hex code should make absolutely no sense to you, but this is what you will see when you import your program into MPLAB to burn the program onto the microcontroller. You will see it appear in one of the windows when you import the *.hex code.

At this point you just burn the code onto the microcontroller, then plug the microcontroller into the circuit you built and test to see how well it functions.

To make changes, you change the source code, recompile to generate a new *.hex file, then load the new *.hex file onto the microcontroller.

MODULE 10

Build the following circuit (just add a few components to your circuit from Module 09):



10-segment red LED Bar Display
Black dot or chamfered corner on side denotes pin 1. This side has all of the diode anodes. NOTE: you should use a 20-pin IC Socket so that you can flip the Bar Display around if you put it in backwards!

Debugging your microcontroller circuit:

The first thing you have to accept is that programming a microcontroller introduces a whole new set of problems: you have to worry about problems with the hardware as well as the software. This is usually not the case when programming a regular computer, normally you just need to worry about the code you are writing. When your microcontroller circuit fails to operate properly, you should ask the following questions to narrow the search for the source of the problem:

1- Is it a problem with the *hardware* or with the *software*?
(it could be wired correctly (hardware is OK), but programmed incorrectly, or vice versa, or even worse, BOTH could be wrong!)

2- If the hardware is correct, is it a problem with the logical flow of the software, or is it a syntax error?

To help you with this, I suggest that whenever you are working with microcontrollers, that you always take small steps, and check everything as you go, each step along the way. Here is what I mean:

First, you decide what you want the microcontroller to do (write pseudo code).
Then you design the circuit (hardware).
Then you build the circuit.
Then you write a *very short program* that allows you to check the hardware.

This will seem like a waste of time, until the day that you spend countless hours debugging the software for your microcontroller, only to discover, after much wasted

effort and anguish, that you had the microcontroller wired up incorrectly. Therefore, I suggest that you always check your hardware first by programming and running a very simple program, like this one:

```
CHECK:
    delay_ms(250);
    OUTPUT_BIT(PIN_A1, 1);
    delay_ms(250);
    OUTPUT_BIT(PIN_A1, 0);
    restart_wdt();
goto CHECK;
```

Of course, you need to add the header lines of the code, as you can see from Module 09, but the code above is all that you will need to test the correct function of the basic hardware of your circuit: does your IC have power correctly connected; is the oscillator correctly connected, is the chip correctly initializing

How do you use this simple program? Just program this onto the microcontroller in the "main" section of the code and run the device. You can either use an oscilloscope, a voltmeter, or even an LED (as in Module 09) to check to see of pin A1 is turning ON and OFF twice each second. If it is not, you have a hardware problem you need to fix. Most often, you have not correctly wired the oscillator crystal, or you have not applied power and ground to the correct pins. If it works, then you can confidently go ahead and write your real program code, knowing that the hardware is probably OK. If you want to be very careful, after you have done this very simple first test, you could then write a slightly longer bit of code, just to check all of your hardware. For example, if you have 4 input switches going to the microcontroller, and 8 output LEDs (sort of like the circuit in Module 09) you could write a simple piece of code to check them all.

How does the simple program work? Here is a line-by-line explanation:

CHECK: This is called a LABEL. It is a point in the code that you can go to.

delay_ms(250); This is a function that is built into the C compiler. It allows you to tell the microcontroller to sit there and do absolutely nothing for exactly 250 milliseconds. You can change the value in the parenthesis: if it is a variable, it can be from 0 to 255, if you put it in as a constant (as in this example), it can range from 0 to 65535

OUTPUT_BIT(PIN_A1, 1); This will output a logical "TRUE" to the register bit A1, which is pin #18 on the PIC16F84A microcontroller. TRUE means that the voltage on the pin will be set to the power supply voltage for the microcontroller, usually +5V. Note that some microcontrollers (such as this one) can operate very well over a range of voltages, for

example, the Microchip microcontrollers that we use in this course can operate from about 2.8 to 6.0 volts. This depends on the exact chip you are using.

delay_ms(250); Just another 250 ms delay.

OUTPUT_BIT(PIN_A1, 0); Now note that we are setting bit A1 to a logical value of FALSE. This will result in pin #18 being set to a voltage of 0 V.

restart_wdt(); This is short for "restart the watch-dog timer". The watchdog timer is an internal counter inside the microcontroller chip. You set it up at the beginning of your code (see the source code for Module 09). This timer runs independently from the microcontroller. When it counts all the way to the end (you set how long this should take), it will automatically reset the microcontroller. This is a very good thing: it prevents the microcontroller getting stuck somewhere. This is a particular problem with microcontrollers in rugged applications since an electrical shock or other transient blast (such as a partial power loss) from the environment can sometimes cause the microcontroller to lose it's place and get hung up without performing properly. The watch-dog timer (WDT) will only allow this to happen for a fixed, brief period of time (usually less than a few seconds), then it will reset the device to the beginning of the program. For this reason, it is important to write your code such that upon *reset*, the microcontroller will always get everything set up properly to commence operation without any help from a human. You will therefore need to carefully consider the startup conditions that are necessary for safe and robust operation of whatever system you are designing, and then develop the code accordingly.

goto CHECK; This line just sends you back to the label CHECK, so the program runs in an infinite loop.

MODULE 10

Laboratory Projects:

Before doing the projects, you should read through the remainder of this module so that you know more about the commands and functions available in the C compiler.

1- Program the microcontroller to perform exactly the same logic function that you implemented using logic gates in Module 07. This time you will use software to do the job instead of discrete logic gates. You can actually do any or all of the logic projects from Module 07, but you should at least do the one that you already built for Module 07.

2- Now you get to experience how much more flexible a microcontroller is than just logic gates. Here is your next project: write a program that carries out the following functions.
   - If only SW1 is pressed, make the led bar graph count upwards (0 -> 255)
   - If only SW2 is pressed, make the LED bar graph count downward (255 -> 0)
   - If only SW3 is pressed, reset to zero (no LEDs are ON)
   - If all three switches are depressed simultaneously, make all 8 segments of the bar graph flash on and off three times per second.
   - If no switches are pressed, the display should hold its current value.
   - You should also add a short delay when detecting switches being pressed so that the device correctly detects switch closure. Sometimes the switch contacts “bounce”, sending a brief stream of spikes into a circuit. You can deal with this in software by detecting the switch closure for a full 1/10 of a second before you take any action. This also allows you to correctly detect simultaneous switch closures, even if the human fingers pressing the switches are not perfectly timed simultaneously.

3- Now for something a bit more complex. You can use the circuit you have already built to make a “reaction timer”. The basic Idea is to have the micro-controller give you a signal, then measure the amount of time it takes for you to respond. Try to write some simple C code to perform the following functions:
   - Turn OFF all of the LEDs.
   - Wait until the person presses and holds SW1.
   - Turn all LEDs ON for about 5-10 seconds (this is the signal for the person to BE READY TO MOVE WHEN THE LEDs GO OFF).
   - Turn all LEDs OFF and begin counting. Count up +1 every millisecond until the person presses SW2 or SW3 (you decide which one). Once the person presses SW2 or SW3, you can display the counted value on the LEDs as a binary number that represents the reaction time in milliseconds. You would need to read binary (it is easy), but it would tell you your exact reaction time to the nearest millisecond.
   - If you really want to make this interesting, you could try to add a pseudo-random number generator to vary the delay times before each reaction time test. The best way to do this might be to simply read the value of the WDT register at the moment of the last response from the person, and use this value times some constant to set the delay for the next reaction time to a value from 2-10 seconds. This is kind of challenging, but you can do it if you want to.

General troubleshooting tips for microcontroller software:
Here is a brief list of the most common mistakes you will encounter when programming a microcontroller:

Missing semicolon at the end of a line.

Variable overflow (your data type is not large enough to handle the values)

Calculations using different data types as variables (example: int8 / float)
(solution: always do calculations with data of the same type)

Device reset: failure to restart the watch-dog timer
(solution: make sure to frequently call out “restart_wdt();”, especially inside loops)

Missing a bracket: ( ), { }

Case sensitivity: The compiler is supposed to be case insensitive by default, but do not rely on this.

MichiganEngineering

MODULE 10

SUMMARY OF PIC-C VARIABLES, STATEMENTS, AND EXPRESSIONS:

In general you can find a lot more information about the commands available in the PIC-C compiler by using the HELP file that comes along with the compiler. The compiler is available on the computer in the lab, and is also available for a few hundred dollars from ccs@ccsinfo.com

Variables (Data Definition):

int1	defines a 1 bit integer (0 or 1)
int8	defines an 8 bit integer (range = -128 to 127)
int16	defines a 16 bit integer (range = -32,768 to 32,767)
int32	defines a 32 bit integer (-2,147,483,648 to 2,147,483,647)
char	defines an 8 bit character
float	defines a 32 bit floating point number
short	same as int1
int	same as int8
long	same as int16

You can modify the integer variables to be unsigned or signed. For example:

unsigned int x;
the variable "x" is defined as an *unsigned* 8 bit integer (range = 0 to 255)

signed int y;
the variable "y" is defined as a *signed* 8 bit integer (range = -128 to 127)

signed long z;
"z" is defined as a *signed* 16 bit integer (range = -32,768 to 32,767)

Although the compiler default is that variables are unsigned, I always explicitly state "signed" or "unsigned" to make the distinction clear. You should refer to the HELP file in the compiler software for examples of how to use these data type definitions, or look at the sample code from Module 09.

Be sure to always choose a data type that can handle the numbers you intend to use. Otherwise, the variable may simply overflow and begin from the lowest value, never reaching the value you had intended. For example, if you are planning to count an event, and you will allow a value as high as 10,000, then you must count the event using at least a 16 bit integer data type. In general, be sure to do all of your calculations on variables of the same type, otherwise you can lose data or the behavior of the microcontroller can be erratic. Also, be sure that any intermediate calculations do not cause a data overflow.

Variable names can be up to 32 characters long, must start with a non-numeric character, and can contain numeric digits and the underscore "_" character.

MODULE 10

CONSTANTS:

When you assign a value to a variable, you need to be sure the compiler knows what number system you are using. The most common number systems you will encounter when using a microcontroller are: binary, decimal, and hexadecimal (HEX).

Binary: all 1's and 0's (base 2)
Decimal: the number system we use every day (base 10)
Hexadecimal: base 16 counting system, useful for efficiently representing code.

You should brush up on your number base systems if you have forgotten about them. Here is an example of the three common number systems, when counting to the decimal value 16, then on ahead to the value 255:

Binary	Decimal	Hex
00000000	0	00
00000001	1	01
00000010	2	02
00000011	3	03
00000100	4	04
00000101	5	05
00000110	6	06
00000111	7	07
00001000	8	08
00001001	9	09
00001010	10	0A
00001011	11	0B
00001100	12	0C
00001101	13	0D
00001110	14	0E
00001111	15	0F
00010000	16	10
.	.	.
.	.	.
.	.	.
11111111	255	FF

To define a number correctly, you should use the following rules when programming:

123 this would be viewed as a decimal number by default (no characters before the numbers to define the number system, so decimal is used by default).
0x123 this is a HEX number because you added a "zero-x" in front of the constant.
0b11001101 this is binary, because you added a "zero-b" ahead of the constant.

MODULE 10

STATEMENTS and FUNCTIONS:

You will want to familiarize yourself with the following Statements and Functions, using the HELP file. First just search for and glance at these, then look these up as you need them. The HELP file gives examples with correct syntax for each. There are many more Statements and Functions available, but these are the most commonly used ones:

if (expression); **else** statement;
while(expression) statement;
do statement **while**(expression);
for(expression 1; expression 2; expression3) statement;
switch(expression) {**case**…}
goto label
label: statement
break;

The built-in functions include:
OUTPUT_LOW();
OUTPUT_HIGH();
OUTPUT_FLOAT();
OUTPUT_BIT();
INPUT();
OUTPUT_X();
INPUT_X();
delay_us();
delay_ms();
delay_cycles();

setup_wdt();
restart_wdt();
setup_timer();
set_timer();
get_timer();
setup_counters();

bit_clear();
bit_set();
bit_test();

transcendental functions: **sin(); cos(); tan(); asin(); acos(); atan();**
other functions: **abs(); log(); log10(); pow(); sqrt();**

Also, have a look at the mathematical and logical OPERATORS in the HELP file.
// anything following the double slash mark is a REMARK, and is ignored
/* anything between a slash-asterisk and asterisk-slash is also a REMARK */

MODULE 10

SELF QUIZ

1: When you first assemble your microcontroller circuit it does not work properly. How would you go about checking the circuit, listing the first things you would check first, then progressing down through the list in the order that you would troubleshoot. Draw this logical process of debugging a microcontroller circuit as a logic diagram in the space provided below, with branching to show how each test result would flow into the next logical step.

PLEASE ANSWER THE ABOVE QUESTIONS AND E-MAIL TO THE INSTRUCTOR
"I have neither given nor received aid on this examination, nor have I concealed any violation of the Honor Code"

X_____

MODULE 10

FEEDBACK

Was this Module useful and informative?

_____

Is there a topic that should get more or better coverage?

_____

In what way can this Module be improved:

Content: _____

Depth of Coverage: _____

Style: _____

Any additional comments that will help us to improve this course:

_____

_____

_____

If you prefer, you may e-mail comments directly to Bob Dennis: yoda@umich.edu