Physics 335
Lab 5 – Introduction to Microcontrollers: The PIC16F88

Introduction

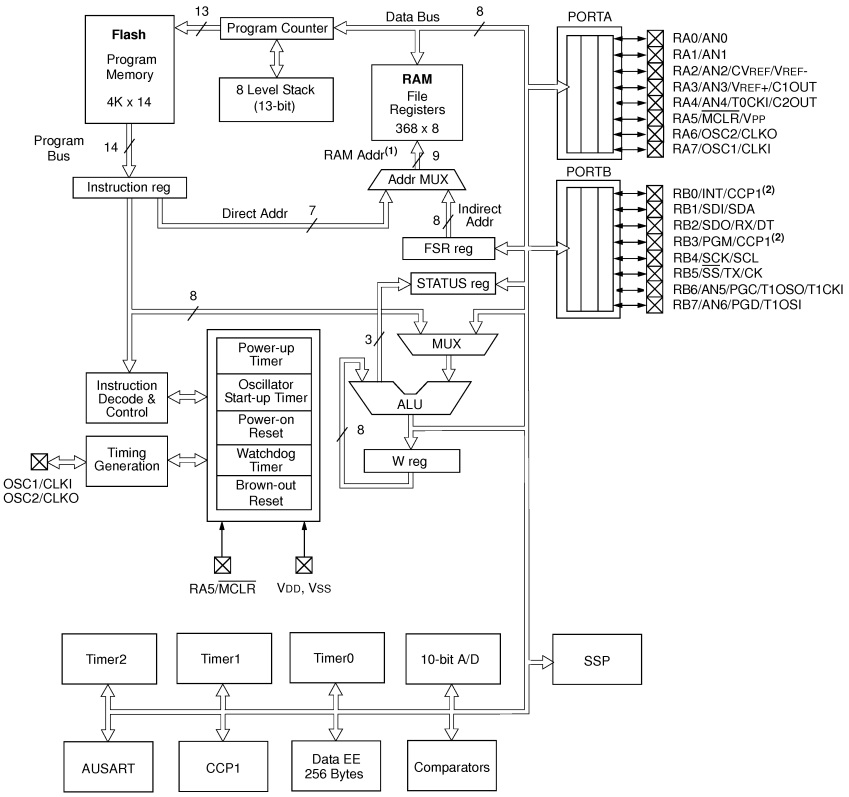
From now through the end of the quarter, we'll be working with microcontrollers. This will without a doubt be an incomplete introduction, by necessity. There are so many flavors of microcontrollers and so very many ways to use them and program them that we'll just scratch the surface. Still, we hope that it will give you enough of an introduction that you add microcontrollers as a tool in your electronics toolbox that you won't be afraid to use where it's appropriate. They should be considered any time your circuits needs some level of intelligence or decision making, but when you don't want to go through the trouble or expense of making a full-blown computer application to do the job.

We've deliberately chosen a very simple microcontroller for our labs. It's one of the simplest on the market, actually, which is both a benefit and curse, depending on what you're trying to do. But its simplicity will allow you to get your feet wet and (hopefully) get a few programs to work and learn about the underlying hardware.

The chip we've chosen to start with is made by MicroChip. We will be delving into the hardware of this chip, so have chosen to have you learn the assembly language of the microcontroller. This will force you to get intimate with the registers and other chip level functions on the chip. There are higher level programming languages that can be used with this chip—C and other variants—and you may choose to use them if you use these chips outside of class. But we'll use assembly lanugage and machine code so you can easily see the relation between your code and the actual bits in the chip's memory.

This chip is *simple*. But simple is relative. It has a nice, short, 200 or so page datasheet. Yes, that's short for a microcontroller. That's just for the chip. The programmer and development boards also have a couple extra hundred pages of documentation, provided on the course web site if you so choose to puruse them. You probably won't read the whole thing, and rare does it seem people do, but you'll need to read and be familiar with the parts you'll need to do your specific programming tasks. We'll try to steer you in the right direction in most cases.

By now you've had an introduction to the microcontroller in class. You'll recall it's a Harvard Architecture chip, meaning the program and register memories are separate. A block diagram of the chip is shown below.



Note the following:

- There are TWO busses - A data bus (8 bits wide) and a program bus (14 bits wide).
- The series of checked boxes on the right labeled PORTA and PORTB represent connections to the outside world.
- The is no direct connection between the program bus and the data bus. So getting a program on to the chip and into the program memory requires a few tricks. Fortunately the manufacturer has solved iths problem for us.
- There are a number of other "things" hanging on the data bus, a few of which we'll get into in future labs. These include A/D, counters and other goodies.

The set of opcodes for our chip, or instructions that it can do is rudimentary. Under 40 in all is all you'll need to understand. In practice, you will rarely use them all.

Don't be intimidated by something that looks this complicated at a first glance. After you spend some time with it, you will get comfortable with it, and you can start to use your imagination to make it solve problems in the lab. But enough babble—let's start using the chip.

5-1 Introduction to the Assembler

We'll be using a programming suite provided by MicroChip to develop our code. It has been pre installed on your lab computers. Log in and start it up by clicking on the MPLAB icon, MPLAB X IDE v2.05, on your desktop.

Your code will be organized in a project. Each project consists of a number of files of source code, that YOU write, along with a few extra files that assist the compiler to understand the chip and it's registers. We'll use just one to start; A file that defines the registers in the chip with easy to remember names. OK, next let's just dive right in and start a new project.

(a) Goto File:NewProject. Select MicrochipEmbedded:Standalone Project > Next

(b) Select Device:Family:MidRange 8-bit MCU:PIC16F88 (our chip) > Next

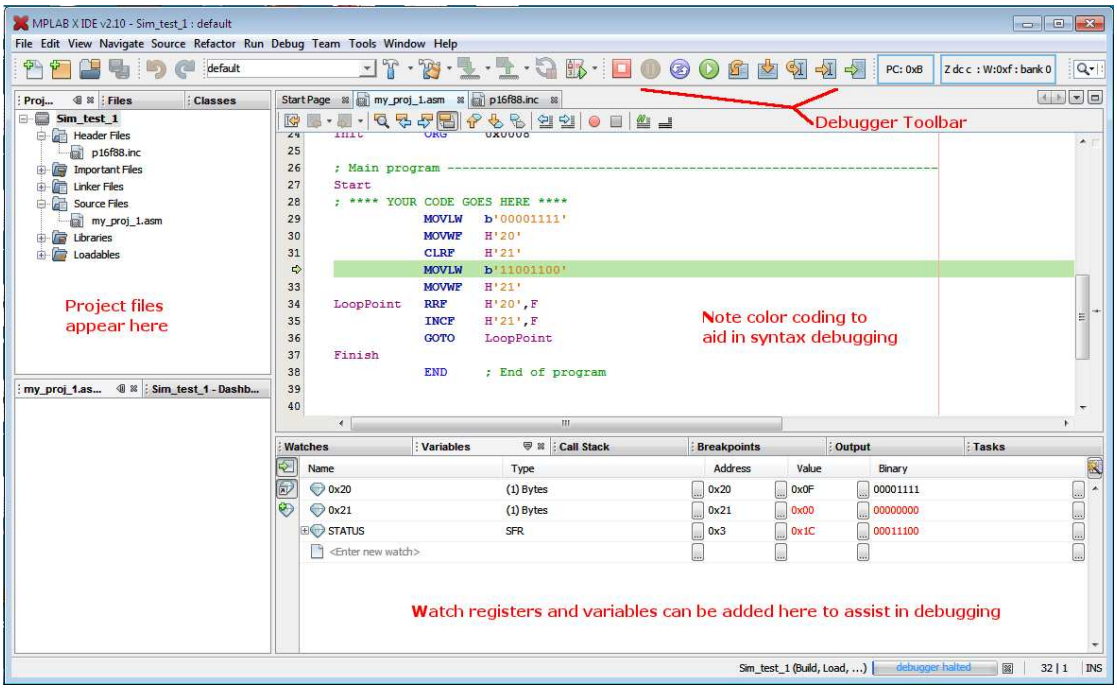
(c) HarwareTools:PICkit3 SN:(A serial number unique to your machine should be shown). Select it

(d) Select Compiler mpasm (highlight) > Next

(e) SelectProjectNameAndFolder: Give it simple name unique to you with no special charcters, spaces, etc. Maybe something like JohnProject1. Remember where the folder is that you store it in.

(f) Click Finish

On the left side, you'll see what's in your project: a sort of file explorer of the files associated with your particular programming task. There will usually be at least a couple files to start off any programming task—A source code file, which for consistency should end with `.asm` (meaning "assembler source file"), and a file to assist in definitions specific to your particular chip



First a word about memory in the PIC. There are two types of memory: Program and Data memory. The program memory stores your program and it's not (typically) changed from the time you program the PIC. The Data memory is where you store all your variable and other transitory information that your program needs to interface with the world in the way you want it to. The data amemory is subdivided further into Special Function Registers (SFRs) and General Purpose Registers (GPRs). General purpose registers are just plain old memory locations. You can use them any time you need to store data and that's all they do. The Special Function Registers actually change the way the PIC executes its code. They're outlined in the datasheet and they are more or less named according to processor or peripheral functions they modify or are modified by.

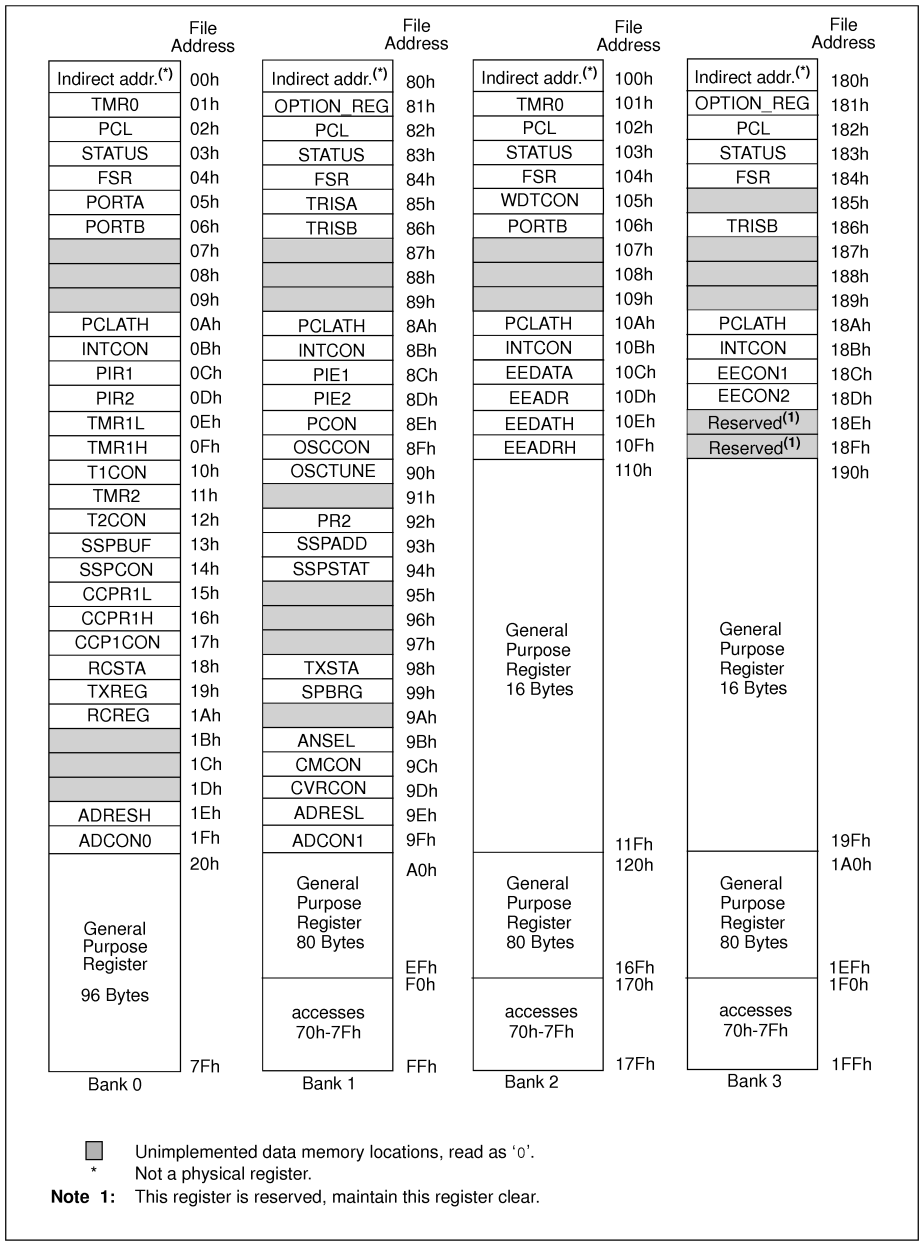
TAKE SPECIAL CARE THAT YOU ALWAYS KNOW WHICH REGISTERS YOU'RE WRITING TO OR READING FROM. IF YOU USE A SPECIAL FUNCTION REGISTER TO STORE PLAIN OLD DATA FROM YOUR PROGRAM, IT MAY CAUSE YOUR PROGRAM TO ACT IN A VERY PECULIAR WAY, AND IT CAN BE A BUGGER TO DEBUG.

First, let's be sure you have a good, clean, and unadulterated copy of the file that defines the registers of the chip. Download the file called `p16f88.inc` from the course website. Save it to your project directory. You need to tell the compiler that you want to use the file as a part of the project. Right click on "Header Files" – Then "Add existing item". Find your file and select it (Leave "store path as relative").

You should see the file populate the tree under header files. Double click it to open it.

The purpose of this file is nothing magical. Mainly it is a list of "equates" (so called) as indicated by the assembler directive `EQU`. These simply create a list of easy (or easier) to remember names that are associated with various numbers (all in hex). For example, you may see `STATUS EQU H'0003'`. What this does is allow you to write `STATUS` instead of `H'0003'` when you want to refer to the address of the status byte. A listing of the organization of the registers in the 16F88 chip (taken from the data sheet) is shown below.

FIGURE 2-3: PIC16F88 REGISTER FILE MAP



One of the most important special function registers, (or places in data memory) is the `STATUS` register. This gives some basic information about how a particular instruction executed and has a number of bits that can be set or cleared based on the results of an instruction. When you use an equate you can tell your program to tell it to load or modify `STATUS` instead of "memory location 3." You can actually refer to it either way. However, remember: *if you want to refer to any of these registers by name it's critical that you include this file as a part of your project. otherwise the assembler doesn't know what you mean when you tell it to read the* `STATUS` *register, for example.*

5-2 Compiling and Using the Debugger

Ok, let's start by looking at a simple program to get a feeling for how this all comes together. This program won't do anything useful; we're just trying to get you to become a bit more familiar with the tools you'll need to actually do the useful stuff.

(A) Assemble the Program

Download the file named `PIC16F88_template1.asm` from the course website and save it to your project directory. Right click on "Source Files" and add this to the tree. Then save it as another file with a simple name (without blanks or special characters) and ending with an `.asm` extension.

This template includes many directives for the assembler that need not concern you at this moment. Later you may dig into the structure of the file.

We'll start by looking at a few basic instructions, what they do, and how we verify that the program is doing what we think it's doing in order that we can make the processor do what we expect it to do. This latter step of debugging is sometimes challenging.

Carefully enter the following code into your assembler editor right after the comment `**** YOUR CODE GOES HERE ****`. [Note: We recommend that you *type* the code in. Some people choose to cut and paste, but if you do be aware that sometimes the formatting of the text can get a little strange. We've also noticed that the single quote characters don't copy over correctly. Use the single quote by your enter key (if it changes to another such as left quote/accent grave, you'll need to change it back).]

```
; Main program ----------------------------------------------------
Start
; **** YOUR CODE GOES HERE ****

            MOVLW   b'00001111'
            MOVWF   H'20'
            CLRF    H'21'
            MOVLW   b'11001100'
            MOVWF   H'21'
LoopPoint   RRF     H'20',F
            INCF    H'21',F
            GOTO    LoopPoint
Finish
            END     ; End of program
```

Save all files in the project.

Now, what does the above mean? Don't worry about about the stuff at the top of your file just yet. They are various directives for the assembler that converts the language to machine code (the actual contents of the program memory). We'll come back to it later. Let's start with the code at the label for `Main Program`.

We use several instructions in this simple program. A LOT of instructions are devoted to to moving information from one place to another in the processor, so we'll start with some of those.

Our most important register is the working register (W). Pretty much all the data we use will move through this register. In fact, there's no way to move data *directly* from one memory location to another without using the W register in the middle. So in our first instruction (`MOVLW`) we execute "move a 'literal' value" (a number written right into our program memory/code) into the W register. The W register is probably the most important register in the entire processor: it is used in almost every instruction, either as a source or a destination for the result of an operation. (As a register, however, it is not directly addressable, like say, the STATUS register.

You'll need to reference the data sheet throughout the remainder of your time working with the PIC controllers. Let's look up the `MOVLW` instruction.

Below is the Instruction Set from the Pic16F88 Datasheet. Under "Literal and Control Operations" find `MOVLW`. You'll find a brief description of the function ("Move literal to W") and its numerical opcode. Also in the datasheet is a detailed summary of each instruction that gives the syntax, the allowable values for the operands, which STATUS bits if any are affected, and a description of what the intruction does.

TABLE 16-2: PIC16F87/88 INSTRUCTION SET

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS						
ADDWF	**f, d**	Add W and f	1	00 0111 dfff ffff	C,DC,Z	1,2
ANDWF	**f, d**	AND W with f	1	00 0101 dfff ffff	Z	1,2
CLRF	**f**	Clear f	1	00 0001 lfff ffff	Z	2
CLRW	**-**	Clear W	1	00 0001 0xxx xxxx	Z	
COMF	**f, d**	Complement f	1	00 1001 dfff ffff	Z	1,2
DECF	**f, d**	Decrement f	1	00 0011 dfff ffff	Z	1,2
DECFSZ	**f, d**	Decrement f, Skip if 0	1(2)	00 1011 dfff ffff		1,2,3
INCF	**f, d**	Increment f	1	00 1010 dfff ffff	Z	1,2
INCFSZ	**f, d**	Increment f, Skip if 0	1(2)	00 1111 dfff ffff		1,2,3
IORWF	**f, d**	Inclusive OR W with f	1	00 0100 dfff ffff	Z	1,2
MOVF	**f, d**	Move f	1	00 1000 dfff ffff	Z	1,2
MOVWF	**f**	Move W to f	1	00 0000 lfff ffff		
NOP	**-**	No Operation	1	00 0000 0xx0 0000		
RLF	**f, d**	Rotate Left f through Carry	1	00 1101 dfff ffff	C	1,2
RRF	**f, d**	Rotate Right f through Carry	1	00 1100 dfff ffff	C	1,2
SUBWF	**f, d**	Subtract W from f	1	00 0010 dfff ffff	C,DC,Z	1,2
SWAPF	**f, d**	Swap nibbles in f	1	00 1110 dfff ffff		1,2
XORWF	**f, d**	Exclusive OR W with f	1	00 0110 dfff ffff	Z	1,2
BIT-ORIENTED FILE REGISTER OPERATIONS						
BCF	**f, b**	Bit Clear f	1	01 00bb bfff ffff		1,2
BSF	**f, b**	Bit Set f	1	01 01bb bfff ffff		1,2
BTFSC	**f, b**	Bit Test f, Skip if Clear	1 (2)	01 10bb bfff ffff		3
BTFSS	**f, b**	Bit Test f, Skip if Set	1 (2)	01 11bb bfff ffff		3
LITERAL AND CONTROL OPERATIONS						
ADDLW	**k**	Add literal and W	1	11 111x kkkk kkkk	C,DC,Z	
ANDLW	**k**	AND literal with W	1	11 1001 kkkk kkkk	Z	
CALL	**k**	Call subroutine	2	10 0kkk kkkk kkkk		
CLRWDT	**-**	Clear Watchdog Timer	1	00 0000 0110 0100	$\overline{TO}$,$\overline{PD}$	
GOTO	**k**	Go to address	2	10 1kkk kkkk kkkk		
IORLW	**k**	Inclusive OR literal with W	1	11 1000 kkkk kkkk	Z	
MOVLW	**k**	Move literal to W	1	11 00xx kkkk kkkk		
RETFIE	**-**	Return from interrupt	2	00 0000 0000 1001		
RETLW	**k**	Return with literal in W	2	11 01xx kkkk kkkk		
RETURN	**-**	Return from Subroutine	2	00 0000 0000 1000		
SLEEP	**-**	Go into Standby mode	1	00 0000 0110 0011	$\overline{TO}$,$\overline{PD}$	
SUBLW	**k**	Subtract W from literal	1	11 110x kkkk kkkk	C,DC,Z	
XORLW	**k**	Exclusive OR literal with W	1	11 1010 kkkk kkkk	Z	

Note 1: When an I/O register is modified as a function of itself (e.g., `MOVF PORTB, 1`), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.

2: If this instruction is executed on the TMR0 register (and, where applicable, d = 1), the prescaler will be cleared if assigned to the Timer0 module.

3: If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a `NOP`.

Note: Additional information on the mid-range instruction set is available in the *PICmicro® Mid-Range MCU Family Reference Manual* (DS33023).

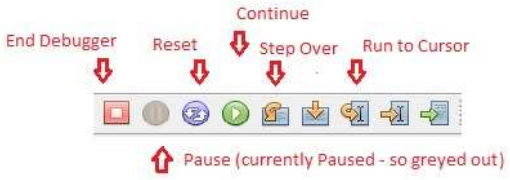
The next instruction is `MOVWF`, which moves the new value of W into a memory location, in this case memory location 20h. Find the instruction and study it's description until you understand it. If you look at the data memory map in the datsheet (or earlier in this lab writeup), you'll find it's the first of the "General Purpose Registers" in "Bank 0" that can be used to store program data.

(B) Compile and Debug

Next, it is time to compile this program and watch how it executes. But first a little housekeeping: we'll need to tell the programmer/debugger to power the chip from the PICkit instead of from an external power supply. Right-click on your project title. On the drop-down menu select **Set Configuration** > **Customize**. Under `Conf:` highlight `PICkit3`. A dialog box will open. Then under **Option Categories** select **Power**, and check the box next to **Power target circuit from PICkit3** and finally select a **Voltage Level** of **5.0**. Click OK to close.

Compile the program as folows. Under **Debug**, select **Debug Main Project**. You may see a warning about powering the chip that may cause damage that sounds rather ominous, but just make sure the PICkit is plugged into the 2nd slot of the PicDem board and all will be well—accept the warning.

Allow the device to program. In the "output" window ou may see various messages about "firmware updates" and the like; give it a little time. Once you get a message of "running," pause the program by clicking the orange "pause button" on the toolbar at the top (see below). Then hit the blue circle reset button. This will hold the processor in a reset state for the moment. Now let's watch how this all works, in slow motion.



Assume we're trying to debug a pesky problem in our program and we can't figure out what we did wrong. We might want to look at what's going on inside the process and watch the memory locations as the are operated on by the different instructions. How do we do that?

First note you'll see your first line of code (`GOTO Init`) highlighted. Click on the **Variables** window below to enter things you want to watch. (if you accidentally closed it, it can be found and reopened under the "window:debugging" drop down). Double click on `<Enter New watch>` and enter the following watch points, relavent to our program.

```
0x20
0x21
STATUS
```

Now let's step through the program. See the buttons above for the **Step Over** button. This button will let you single step through the program but considers any function calls as a

single function. Press it a few times and watch how it walks you slowly through the program. The watch window highlights any of your watches that change value in red. Special function registers often even can be opened up further to watch individual bits by name. For example, watch what happens at `STATUS<2>` as the `CLRF` function is executed. Not surprisingly, the 0x21 register is set to zero, and the zero (Z) flag (Bit 2) of the `STATUS` register is set.

Also notice the two blue-bordered boxes next to the tool bar that contain cryptic messages similar to `PC: 0xE` and `z dc C : W:0x0 : bank 0`. The first of these shows the "program counter". You will notice that it advances at each step (usually by 1). It shows the current location in program memory being used by the program. The second box shows the `STATUS` bits for carry (C), digit carry (DC), and zero (Z). When these are in upper case, the bit is set; in lower case, the bit is clear. After this is the contents of the W register (in hex). Finally the current bank of data memory being accessed is shown.

Look up the remaining instructions used in this program and verify that you can understand what they're doing by reading about them in the documentation and watching what they do as you step through the program.

5-3 Interfacing Ports

OK, fine, and good: we can step through a program, but the chip is not doing anything except cycling through its own internal program. How do I make it something useful? Enter the "Interfacing Ports."

All of the pins on the PIC are multipurpose, except for the power pins V_{DD} (pin 14, 5.0 V) and V_{SS} (pin 5, ground). The pins can be used in a number of software configurable ways, but we'll start with the easiest one—using them as digital inputs or outputs. When they are used in this way, they are referred to as "Port A" and "Port B."

Certain special function registers are very important for controlling their function:

`PORTA` and `PORTB` The Special Function Registers located at 0x05 and 0x06. Eight pins are associated with each digital port. They can be used for either input or output and each pin can be set individually. If set for output, it is especially important that there be no "line contention" (other voltage sources connected to the same wire as the active-out pin). When using a PORT as an input, one *reads* the associated register; when using it as an output, one *writes* to the associated register.

`TRISA` and `TRISB` These registers located at 0x85 and 0x86 control the "tri-state" buffer of the ports, and determine whether the port is an input or an output. In other words, for a pin to act as a digital output, one must enable the tri-state buffer by setting the appropriate TRIS digit and then write to the appropriate PORT register digit. Setting a value of 1 to a TRIS bit makes the corresponding pin an input; clearing the bit (setting to 0) makes it an output. So setting TRISB to 0x0F makes pins 6, 7, 8, and 9 (RB0 through RB3) into digital inputs and pins 10, 11, 12, and 13 (RB4 through RB7) into digital outputs.

`ANSEL` This register determines whether the pins associated with Port A are digital inputs or analog inputs. The microcontroller has an onboard A/D converter; when it is used the associated pins must be turned on for this function. By clearing `ANSEL` one set the inputs to be of digital type.

This is also a good time to talk a little more about the `STATUS` register. As you have found, The `STATUS` register is automatically updated when certain instructions are executed. Here's the layout of the `STATUS` register, from the PIC 16F88 Datasheet

REGISTER 2-1: STATUS: ARITHMETIC STATUS REGISTER (ADDRESS 03h, 83h, 103h, 183h)

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit 7							bit 0

bit 7 **IRP:** Register Bank Select bit (used for indirect addressing)
1 = Bank 2, 3 (100h-1FFh)
0 = Bank 0, 1 (00h-FFh)

bit 6-5 **RP<1:0>:** Register Bank Select bits (used for direct addressing)
11 = Bank 3 (180h-1FFh)
10 = Bank 2 (100h-17Fh)
01 = Bank 1 (80h-FFh)
00 = Bank 0 (00h-7Fh)
Each bank is 128 bytes.

bit 4 **$\overline{TO}$:** Time-out bit
1 = After power-up, CLRWDT instruction or SLEEP instruction
0 = A WDT time-out occurred

bit 3 **$\overline{PD}$:** Power-Down bit
1 = After power-up or by the CLRWDT instruction
0 = By execution of the SLEEP instruction

bit 2 **Z:** Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1 **DC:** Digit carry/$\overline{\text{borrow}}$ bit (ADDWF, ADDLW, SUBLW and SUBWF instructions)(1)
1 = A carry-out from the 4th low-order bit of the result occurred
0 = No carry-out from the 4th low-order bit of the result

bit 0 **C:** Carry/$\overline{\text{borrow}}$ bit (ADDWF, ADDLW, SUBLW and SUBWF instructions)(1,2)
1 = A carry-out from the Most Significant bit of the result occurred
0 = No carry-out from the Most Significant bit of the result occurred

Note 1: For borrow, the polarity is reversed. A subtraction is executed by adding the two's complement of the second operand.
2: For rotate (RRF, RLF) instructions, this bit is loaded with either the high or low-order bit of the source register.

You've already met a couple of the bits - For example the Zero bit (Z) or `STATUS<2>`. But another set of *really* important bits in the STATUS register are the "bank select" bits, RP0 and RP1, also designated `STATUS<5>` and `STATUS<6>`.

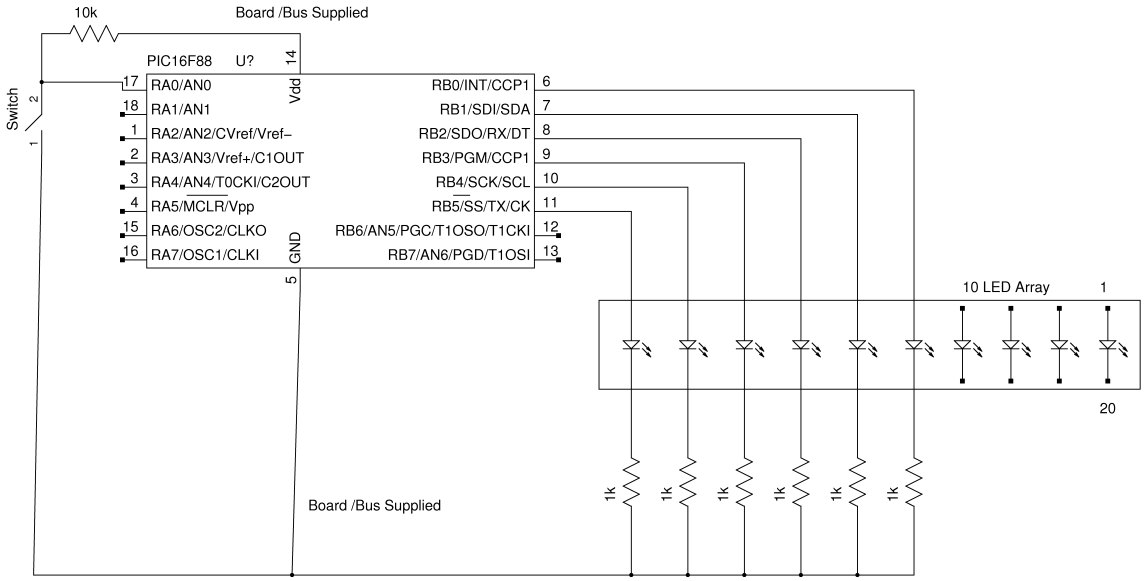
Look again at the data memory map. We'll remind you that they are a combination of General Purpose and Special Purpose Registers. You'll note that this memory is divided into 4 *banks*. The structure of the PIC's opcode allows you to set the lower 7 bits of the address. This allows you to address 128 bytes of data memory space. To access more data memory space, the upper two bits are set by setting or clearing the bank select bits which reside in the STATUS register.

We need to use the RP0 an RP1 bits to address the SFRs `TRISA`, `TRISB`, and `ANSEL` because they reside in memory locations above 0x7F.

The following code sets up `PORTB<0..5>` as an output and `PORTA<0>` as an input. A switch allows us to interact with the processor. When we switch the switch, we change from one set of distictive output bits on Port B to another

Note we do not use Port B bits 6 and 7 at this time because these pins are in use by the debugger. They are available for use, but not if we compile the program in debugging mode as they're used to control the debug execution state and stepping.

Wire up the following circuit. The LEDs are all on one little "stick." Look for a thick white DIP-socket chip with little orange rectangles all in a row.



Now, open up a new project, load in the template, and add the following code in the `Main Program` section:

```
; Main Program --------------------------------------------------------------
Start
            BSF STATUS, RP0     ; Go to bank 1
            MOVLW b'00000000'
            MOVWF TRISB         ; Set Port B as Outputs
            MOVWF ANSEL         ; ANSEL = 0x00 --> inputs are digital
            MOVLW b'11111111'
            MOVWF TRISA         ; set Port A as input
            BCF STATUS, RP0     ; Go to back to bank 0

MainLoop    BTFSS PORTA, 0      ; Look at PORTA<0>
            GOTO PB1Pushed      ; branch if lo, skip to next if hi
            MOVLW b'00111101'
            MOVWF PORTB         ; Output a distinctive LED pattern
            GOTO MainLoop       ; loop back

PB1Pushed   MOVLW b'11000010'
            MOVWF PORTB         ; Outut a contrasting pattern
            GOTO MainLoop       ; And go back to the Main Loop

Finish
            end                 ; end of program
```

Compile and run it under Debug Mode as you did before, although there is no need to single-step through the code unless you have a problem. You will know it works when you see the LED pattern change when you flip the switch.

5-4 Your Own Code

Finally, you will use your own creative imagination to write a program that extends this port output functionality. The goal is to light up an array of LEDs *sequentially*. Heres a general outline:

(a) Initialize PortB and TRISB appropriately.

(b) Turn on one LED.

(c) Pause an appropriate period of time.

(d) Change the value on PORTB to turn off the first LED and turn on the next one over.

(e) Repeat endlessly returning to the first LED at the end of when you run out of bits on the port.

Use the template and examples above to save time. A few things to think about:

- Be sure your pause is long enough, but not too long.
- Do not assume that read and write functions can be accomplished equivalently on output ports.
- You might find the rotate instructions `RRF` and `RLF` useful.

You may use the following delay code, or feel free to write your own. You should try to understand it in any case.

```
; Entering delay loop

Delay       MOVLW    H'FF'
            MOVWF    H'30

Loop        MOVLW    H'FF'
            MOVWF    H'31'
SubLoop     DECFSZ   H'31'
            GOTO     SubLoop
            DECFSZ   H'30'
            GOTO     Loop

; Leaving delay loop
```

If you complete this, add functionality to have the switch control the direction that the light array rotates.

Turn in source code and demonstrate a functional controller by the end of class. Your TA will check off that your code functions as advertised.