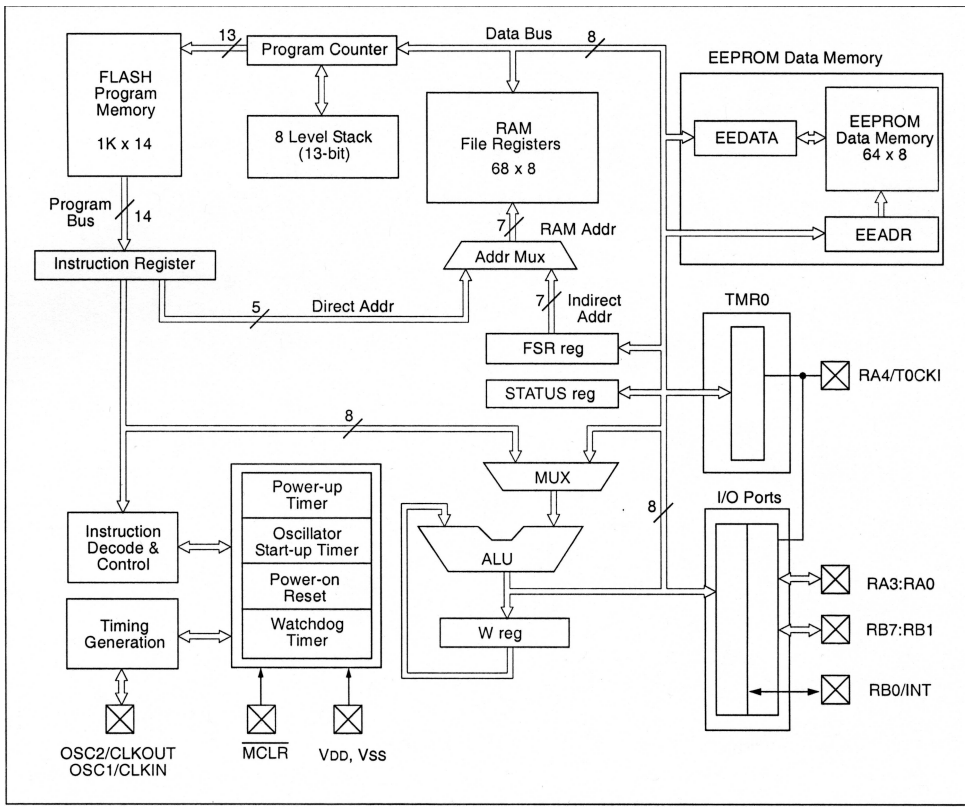
Physics 335 – Intro to MicroControllers and the PIC Microcontroller

May 4, 2009

1 The Pic Microcontroller Family

Here's a diagram of the Pic 16F84A, taken from Microchip's data sheet.



Note that things are slightly more complicated than the simple diagram above would indicate. But the essentials are the same. You should roughly consider the Flash Program Memory to correspond to the Program Memory in a simplified Harvard Processor and the RAM File Registers and EE Data Memory to correspond to the data memory. To jog your memory on Harvard Architecture computers and memory, review your class lecture notes.

As we indicated, in this course, we will be having you focus on the hardware aspects of digital microcontrollers. We want you to get a feed for what is going on "under the hood" – i.e. We want you to really understand how the microcontroller works, not just have it be a magic box that

does some fancy things when you tell it to. As a result, we'll use low level languages for all our programming for the remainder of the quarter.

Materials for the microcontroller labs will be distributed by your TA. You should upload any source files to your UW account or a flash drive. While we have no intentions of erasing any, in the event of a computer failure, we reserve the right to restore any of these machines to a "pristine state" without notice.

So lets get started.

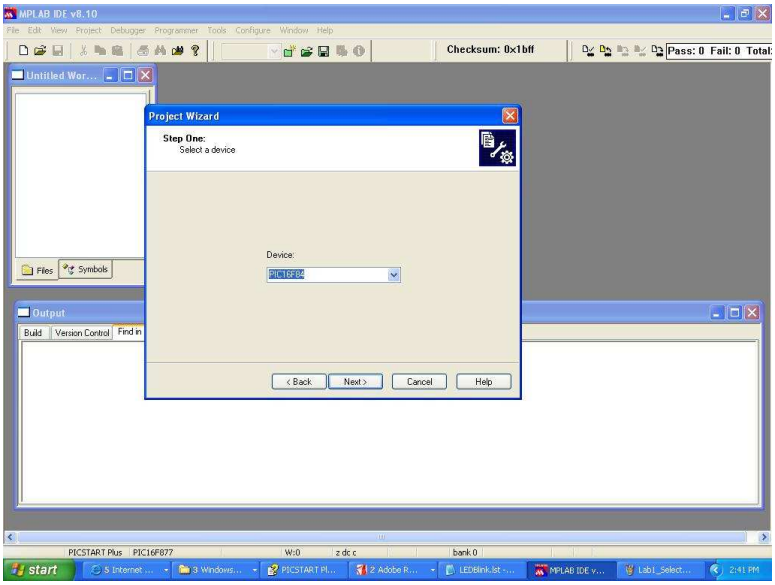
2 The MPLAB IDE

As you are learning in lecture, all intelligent digital controllers understand only ones and zeros, and can work with them in bewildering speed. Sadly for us, it's often difficult for people to remember long strings of ones and zeros. So, we've invented software translators that turn the language we speak (words) into the language computers speak (ones and zeros) – This software tool is called an assembler.

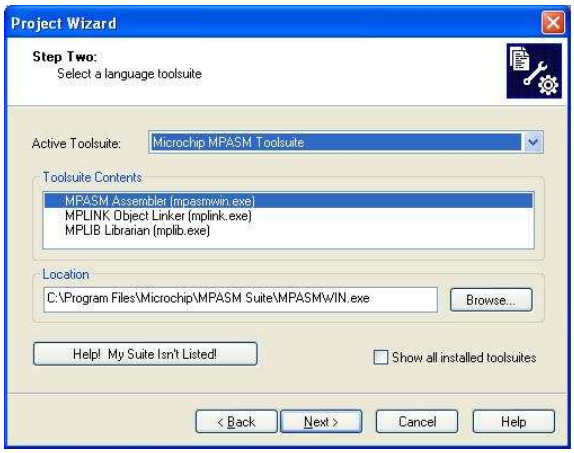
Before beginning, create a folder in the My Documents folder for your work (Use your last name or group name). Also, download the p16f84a.inc file from the course website and save it to that folder.

On your desktop or in the start menu, double click on the icon for the MPLAB IDE (Integrated Development Environment) and open it.

All of your work in the Microchip IDE will be organized into groups of files called "Projects" Start a new project by selecting (Project) -¿ (Project Wizard). You should now see something like:



Select a Device (the one we will develop for today is the 16F84A). Select it and click next. Select the language toolsuite... For us for now, Select the MPASM Toolsuite and MPASM Assembler.



Select next and create a New Project: You can call if "First Project", "FastToggle" or whatever you like... When the project is created, be sure that both the Project and Output options are checked under the VIEW dropdown.

If you like, you can add pre-existing files to your project by right clicking on an appropriate folder – Browse to find the include file you downloaded form the course website. Select it and add it to the project.

As we stated, your 16F84 microcontroller is basically a tiny computer. We will give it commands to execute in assembly language. These are translated into the ones and zeros of the machine code, which the processor itself actually understands. The assembler is a useful program to turn the Assembly Language code, which we can (fairly) easily understand, into the machine code, which the processor understands.

You'll see that your project is divided into several types of files. You added an assembly language include file by using the project wizard. We'll now add an assembly file with some very simple instructions to toggle one of the pins on the pic on and off. Keep in mind, the pic can execute these instructions very quickly... About a million a second at the rate we will be clocking it.

The following contains an instruction discussed in class lecture. Review your notes on how to use the TRIS instruction. Note that this instruction always write to the second register bank (i.e. will write to 86H below). This instruction may not be included in coming versions of the assembler... A pity, because it's handy for what we need to do at this moment because it saves us a few confusing steps.

3 Hello Pic World - A first program

On the source folder, right click and add a new source .asm file. Enter the following:

title "First Project - Some practice with MPLAB IDE" ;

```
; loopledfast
; L. Rosenberg, Phys335, 25Apr09
        processor 16F84A
        INCLUDE "p16f84A.inc"
        __CONFIG _CP_OFF & _WDT_OFF & _XT_OSC & _PWRTE_ON
        org H'00'

        movlw    H'00'        ; load w with zeros
```

```
        tris    H'06'            ; move w to portb, make output
loop    movlw    H'FF'         ; load w with all ones
        movwf    H'06'         ; move w to port b output
        movlw    H'00'         ; load w with all zeros
        movwf    H'06'         ; move w to port b output
        goto loop             ; loop again

        end
```

(We'll worry about most of this is a few moments...) For now, let's start working with it...

Since the microprocessor understands instructions that are listed in it's datasheet and nothing else and since each microprocessor may have a different instruction set, it's important to get the datasheet that applies to your microprocessor. (Microprocessor families generally have similar instructions, though, so you won't have to start from scratch every time, don't' worry). These instructions are, if you have spent any time programming in object oriented languages, C++, Visual Basic, etc., very basic. However, basic instructions such as these are the basis of what each processor understands and we will focus on them this quarter. This course is a hardware focused course, and while we recognize that there are higher level languages we can use, we won't focus on them this quarter, to be sure you have a very good understanding of what's happening "under the hood" of your microprocessor. We don't want you to think of it as a magic black box, but rather a very understandable piece of hardware. In addition, we've chosen a simple microprocessor for you to start with to keep things manageable and understandable for you. The instruction set for the microprocessor you are using today is quite small, only 35 single word instructions, which is incredibly simple for a microcontroller. To keep things simple for your first project, we consider this to be a good thing, though it will of course limit you to simpler projects.

This is deliberately a very simple program – just about the simplest we could imagine, to show you how this all comes together. White space in the program is mostly cosmetic, though you'll note we keep labels to the left and indent instructions slightly. Conditional instructions often are given an additional indent. Note capitalization for variable names, if they are used (we have chosen not to use them, yet, and address registers directly.)

This program uses only a couple instructions. A couple move instructions, to initialize the registers, a tris instruction that allows us to configure an output port, a loop variable... And of course some processor directives telling about the processor we're using and how we want it configured. We use two registers that are special for this program. These two registers control an eight bit input/output port that we'll use for this first output to the real world.

- **PORTB** – ***Address 06h – An eight bit Register that is connected internally to the outside world. This register can be written to to output a digital value (0 or 1) to eight pins on the microcontroller, or, if configured as an input port, read from to read a digital value from the outside world.***
- **TRISB** – ***Address 86hControls the DIRECTION of the data flow on PORTB. Set to 0 for output, 1 to input. The target register of our TRIS instruction above.***

Each bit in TRISB individually controls the direction of each bit on PORTB. Set TRISB = 0x00 for all output, 0xFF for all input, or 0x0F for 4 bits of output and four of input. Similar registers exist for the other input/output port available to us, PORTA.

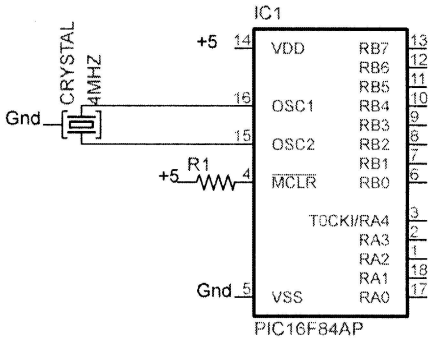
There are several more registers that control operation and execution of this PIC microcontroller, and these are outlined in Figure 2-2 of the PIC16F84A datasheet. We'll come back and discuss some of these further in future labs.

Save the file under file:save as (choose a name). Then: Build all.

Now we have to put our simple program to work.

Put the processor in the programmer. Be sure pin one is all the way "up" in the socket. Use the lever to GENTLY lock it into place. Now Go Under: Programmer: Select Programmer: PicStartPlus – Then Programmer: Enable Programmer – And Finally: Programmer: Program. *DO NOT REMOVE OR OTHERWISE DISTURB THE CHIP WHILE THE PROGRAMMING IS TAKING PLACE!* You should, in the end, get some message to the effect that it was completed successfully. Depending on the software version you have, you may get a message asking where you want to locate compiled code in memory... For our purposes it won't matter at this point. Accept the default (relocatable).

Finally, we need a basic circuit for the processor to run. Use the following, which has only power, ground, a crystal to use for a clock signal for the processor, and reset pins tied to appropriate voltages. Most processors have a way of resetting the processor, in our case, this signal is MCLR*, on pin 4. If this pin goes low, the processor resets all registers and starts execution at the start of the program again. We of course don't want this to happen accidentally, so we tie this pin HI so it's not prone to pick up any noise.



Look at the output from pin RB0 on your scope probe (pin 6). What do you see? Does it make sense? All 16F84A instructions execute at a rate of 1MHz with the exception of branching instructions which take twice as long. What frequency do you expect? Is your output 50% positive duty cycle? Why or why not? Can you modify your code and reprogram so that you get an output with a 10% positive duty cycle (of any frequency), using only the nop instruction in addition to those instructions used? See the data sheet for the Pic 16F84A for all instructions. Remember that a 10% duty cycle wave means that the output is positive 10% of the time.

Next lets do a few modifications to our first program than interacts with the real world. The first will be nice and simple – Flash an LED on and off at a regular interval so that we can see it blinking.

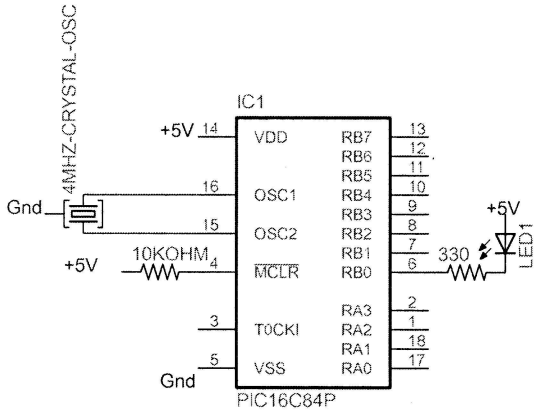
4 LED Blink

The create a new project, as before. This time the project should slow the pic down so that it's working on a more "human time scale"

You may find the following routines useful. Or feel free to write your own. We are using literals for the registers for the moment. Using assembler defined variables, as we'll do next week, this can be done in a more elegant and pretty way, code wise.

```
pause    movlw    H'FF'        ; move 255 to w
         movwf    H'0E'        ; move w to location 0E
loadN    movlw    H'FF'        ; move 255 to w
         movwf   H'0D'        ; move w to location 0D
decN     decfsz   H'0D', 1        ; decrement location 0D
         goto decN            ; if '0D' ne 0 then loop
         decfsz   H'0E', 1        ; else decrement '0E'
         goto loadN           ; if '0E' ne 0 then loop
         return               ; exit subroutine
```

Your objective this time: Toggle the output port bit again, this time, slower so you can see it blinking clearly (on/off for a good chunk of a second). Use the following circuit modification. Modify the original code from above. Have your TA verify that your circuit functions correctly and initial that you have a working circuit before proceeding.



The remaining pins can remain unconnected for now. We will turn the LED on and off by changing the value output at pin PortB.0. Will the LED be lit when the value is High or Low?

5 Extension Lab Exercise

For the remainder of the class, you will write a program that extends this blinker functionality to the remaining pins on PortB and lights up an array of LED's sequentially.

Here's a general outline:

- Initialize PortB and TRISB appropriately

- Turn on one LED
- Pause an appropriate period of time
- Phange the value on PORTB to turn off the first LED and turn on the next over.
- Pepeat endlessly returning to the first LED at the end of when you run out of bits on the port.

You can feel free to use the configuration statements from the LED blink routine to save you a little time.

A couple gotcha's and useful points:

- 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 useful. Initialize PortB and TRISB appropriately
- Be sure you use a resistor in series with EACH LED. Keep in mind the direction the diode conducts and whether HI or LO will make it light.
- All components can source or sink currents only up to a point. Using 1k resistors will probably allow you not to think too much about this, though.

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

That's it!