

Chapter 6: Introduction to Using the MPLAB X Simulator and Debugger

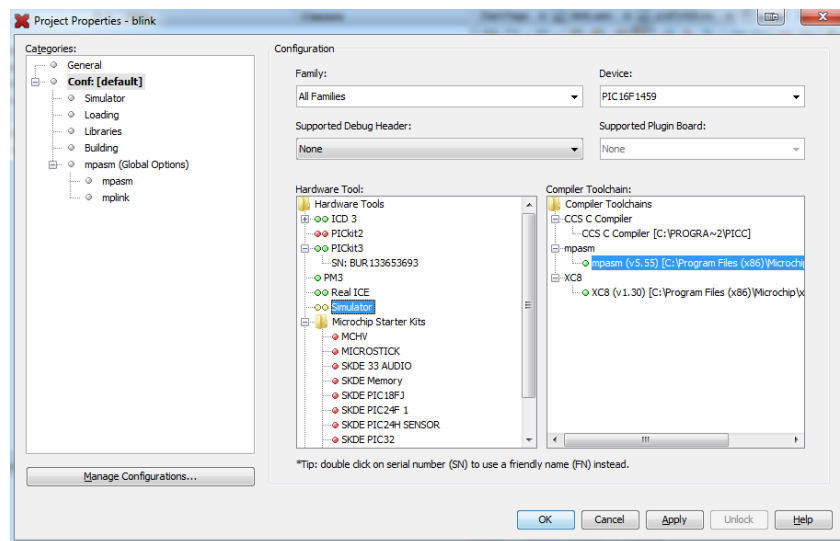
You will probably find that debugging embedded projects is more difficult than debugging PC software projects. The reason is that if the code does not run as expected, with embedded projects you often don't know whether the problem lies with your software design or with your hardware design. As a result, it is important to become comfortable using the debugging tools that are available in MPLAB X as soon as possible. MPLAB provides two tools that you can use that greatly facilitate this process when used in conjunction with the PICKit 3: a simulator and a hardware debugger. Both of these operate in a way that is very similar to the debuggers you might have previously used in PC software design IDE's like Microsoft Visual Studio, but they provide additional features that make them applicable to an embedded environment.

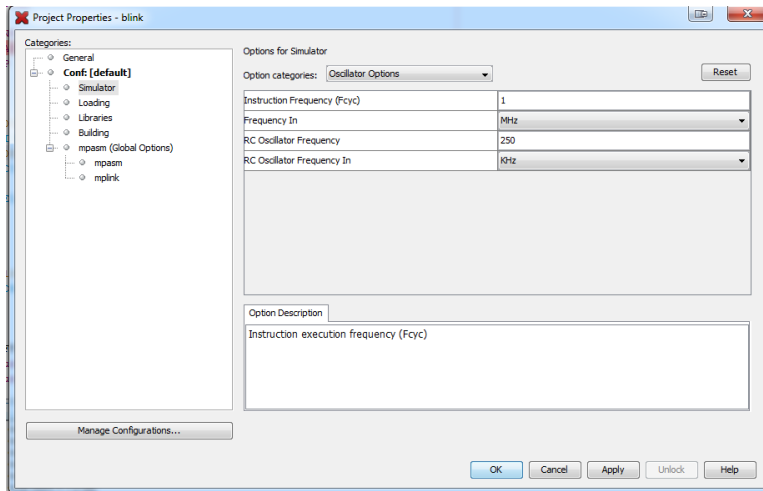
The simulator allows you to step through your code in software and observe how specific registers change. For example, suppose that you wrote the code presented in Chapter 5 and found that the LED never turned on. With the simulator you could step through the code while looking at the values of the PORTB and TRISC registers and confirm whether the pin was actually set as an output and whether its value in PORTB actually changed. If you saw both these things happening, it would cause you to suspect that the problem lay with the way you had set up the hardware.

The hardware debugger allows you to do something similar, but to actually do it in the hardware itself rather than a simulator. You can step through the program and see if the LED actually turns on when you get to the BLINK section of the code.

The Simulator

To use the simulator, we need to specify that tool rather than the PICKit 3 as the one that will be used in this project. From the file menu, select Project Properties. Under Hardware Tools click on Simulator to change from the PICKit 3 to the simulator:

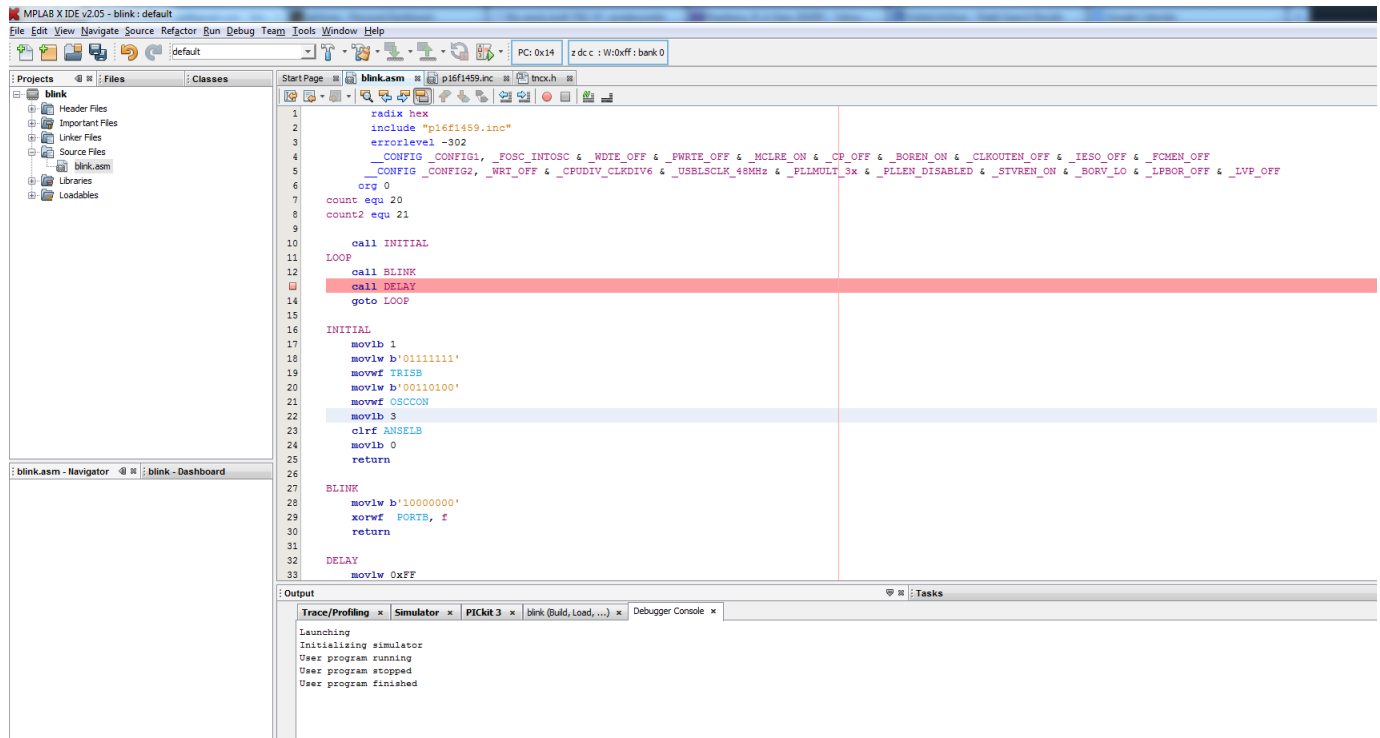




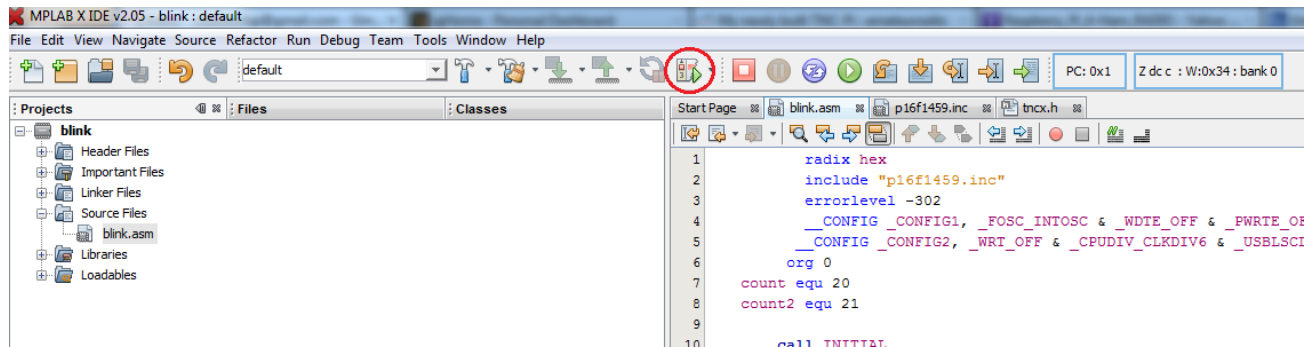
While we are in the Project Properties box, click on Simulator on the panel on the left (under Conf:). The box will change to show Simulator options. Notice that the Instruction Frequency is set to 1 MHz. Because the simulator is able to measure the time it takes to execute blocks of instructions, it needs to know how frequently instructions are executed. We have configured this PIC to run at 4 MHz and this PIC (and most others) execute 1 instruction ever 4 clock cycles. So if the oscillator frequency is 4 MHz, the instruction

frequency will be one fourth of that or 1 MHz. So, as it happens, the value is set correctly for the code that we are simulating. However, if we had run the oscillator at 8 MHz, for example, we'd have to change this value to 2 MHz.

Click OK to close the dialog box. We will now run the program in the simulator and make sure the state of the LED is changing as well as check to see how long the delay actually is. To do this, we will want the program to stop each time it loops through the LOOP code. We'll have it stop on the line of code calls the DELAY routine. To get it to stop we are going to set a "breakpoint" on that line of code. On the window with the code in it, click on the number of the line of code that calls DELAY. When you do this it will turn red:



Push the "Debug Main Project" button on the toolbar:



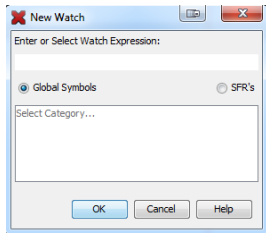
The simulator will now start running the program at the beginning and continue until it gets to this breakpoint. At that point it will stop, and the line of code with the breakpoint will turn green. In addition a new set of buttons will appear on the tool bar that will let you navigate through your code:



The first button stops the program and exits the debugging session. While the program is actually running (if there were no breakpoint it would run continuously) the second button would be enabled and you could press it to pause the debugger. This causes it to stop at whatever line of code it is currently viewing, but it does not end the debugging session. The Run button would start it going again. The Reset button stops execution of the code and prepares the program to start again from the top. The Step Over button causes the next instruction of the program to be run. If that instruction was a Call instruction it would continue executing instructions until it hit a return. That would then take it down to the next line of code in the main loop. The Step Into button is similar, but in this case if the next instruction was a Call instruction, it would jump the routine that is being called and set up to begin executing the first instruction that is there.

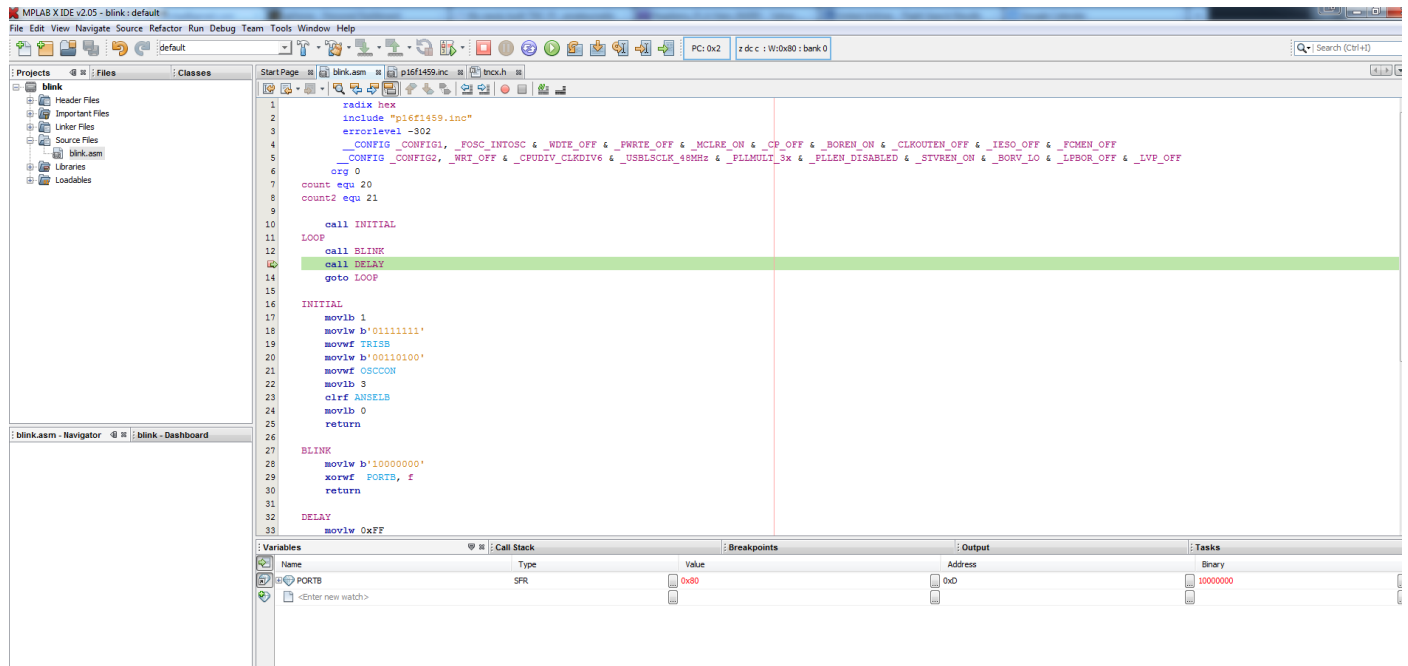
At this point it is important to clarify the difference between the cursor and the program counter. The cursor is the point where you click your mouse. It is the instruction that has the flashing cursor on it. The program counter is the instruction that is just about to be executed. These can be the same line of code, but they don't need to be. If you use the mouse to click on another line of code, the cursor will move there but the program counter will stay where it is. This is a useful feature. Suppose you want to execute the next two lines of code. One way to do this would be to establish a second breakpoint two lines lower. Another would be to place your cursor two lines down and push the Run to Cursor button on the toolbar. You could also right-click your mouse two lines down and select "Run to Cursor" from the popup menu. It is possible to have the program counter jump to where your cursor is using the "Set PC at Cursor" button, but note that this just moves the program counter, it doesn't execute the instructions in between the two (if you want to do that, use the "Run to Cursor" button instead. Finally, you can bring the cursor back to where the program counter is by pressing the "Move Cursor to PC" button.

Now let's establish a "watch window" that will allow us to keep track of the values of certain registers while the program is running. To do this, select "New Watch" from the Debug menu:

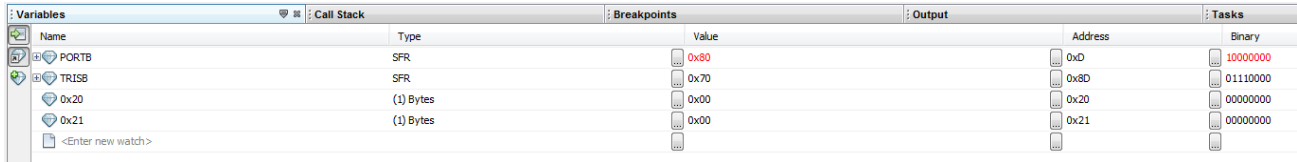


If you push the SFR's radio button a list of all of the Special Function Registers will appear below. You can either do that and select PORTB from the list, or you can just type PORTB in the box at the top. You can also enter the names of variables that you have declared like "count" if you like. Push OK and the popup dialog will be dismissed. A new panel will be added at the bottom of the screen that will present the value of PORTB.

You will find the watch window displays the value of the variable in binary and hex and the variable's address in memory. You can add other entries as well (say, the decimal value of the variable) by right clicking on the table headings and picking the entry you want added. Notice the line under the PORTB line let's you add more variables to this watch window by double clicking on it. Add TRISB as well. You should be to add variables like count that you have declared using the equ notation and you can, but as of this writing there is a bug in MPLAB X that causes those variables to not be displayed while a debugging session is going on. Terminate the debugging session and those variables will be visible. Start a new debugging session and they will disappear. You can get around this bug by entering the address (0x20 for count, 0x21 for count2) as the name of the variable instead of their actual names. Perhaps by the time you read this, this bug will be fixed.



With all of the variables added it would look like this:



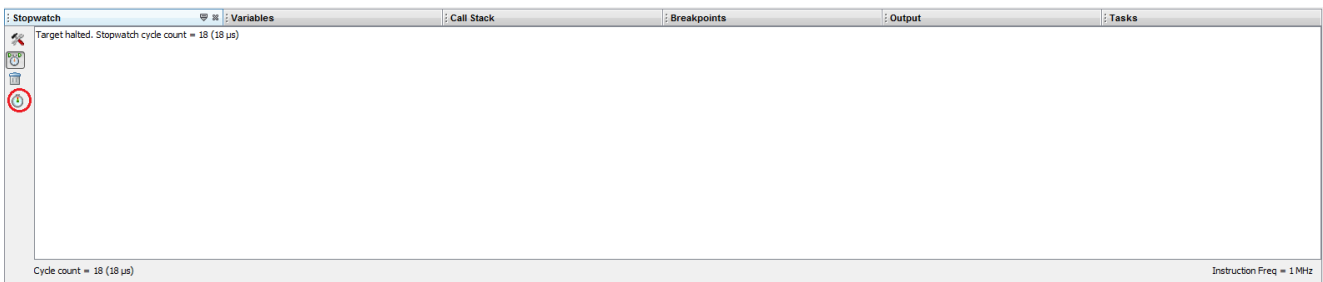
Name	Type	Value	Address	Binary
PORTB	SFR	0x80	0xD	10000000
TRISB	SFR	0x70	0xB	01110000
0x20	(1) Bytes	0x00	0x20	00000000
0x21	(1) Bytes	0x00	0x21	00000000
<Enter new watch>				

Looking first at TRISB, you'll notice that the left-most bit is a zero, indicating that the pin is an output. Despite the fact that we set all the other pins to inputs, you'll also notice that pins RB0 through RB3 seem to be outputs as well. This is because these pins don't actually exist on this particular model of PIC; it has only pins RB4-RB7.

Pin RB7 of PORTB is a 1. It started as a 0, but because we set the breakpoint just after the BLINK call it has been changed to a 1. Because it was changed in the previous line of code that was executed, its value is in red. Any value that changed since the time the execution was last stopped will be indicated in red to help you see where the changes are.

This time press the "Step Into" button instead of the run button and watch the values of count and count2 change as you step through the code in the delay routine. It will give you a very clear view of how that routine works.

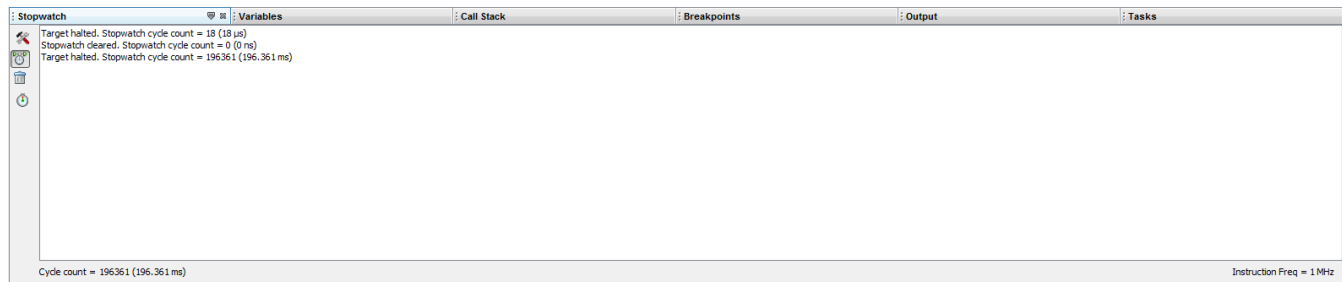
Now let's measure exactly how long the delay is between the running of the BLINK routine. We have a breakpoint established when the DELAY routine is called so we need to be able to measure how long after we push "run" at this point it takes until the code hits the breakpoint again. To turn on the stopwatch, select Window/Debugging/Stopwatch from the menu. This will bring up a Stopwatch panel at the bottom of the screen. Start a debugging session and the program will run until you hit the breakpoint. At this point you'll have to select the stopwatch panel again:



Stopwatch	Variables	Call Stack	Breakpoints	Output	Tasks
Target halted. Stopwatch cycle count = 18 (18 μs)					
Cycle count = 18 (18 μs)					
Instruction Freq = 1 MHz					

You'll note that the stopwatch has determined that 18 instruction cycles have gone by and it has taken a total of 18 us. We don't want to count those 18 instruction cycles as part of the time between blinks so we have to zero the stopwatch. Do that by pressing the clock icon on the left of the screen.

Now press the Run button again on the debugging tool bar and you'll see how long the period between the LED being turned on and being turned off is:



It takes about 196 ms (~196000 us). So that means to go from off to on and back to off again it will take twice this long or about 4 tenths of a second. This is how I knew that the LED was blinking about 2 1/2 times per second.

There is a lot more that the simulator can do for you, but this will get you going enough to do some useful work debugging programs.

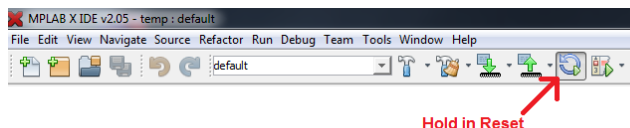
The Hardware Debugger

So far everything we have done in this chapter involves only the simulator. We have not reprogrammed the hardware on your breadboard or run any of the code in your microcontroller. Just as it is possible to step through the code in the simulator, with many PIC chips it is also possible to step through the code in the actual hardware itself. To do this, go back to the Project Properties dialog box and switch back from the simulator to the PICKit 3.

Now almost everything we've done in this chapter can also be done in hardware. There are some limitations though:

1. You can have only 2 breakpoints at one time if you want to use features like Run to Cursor, etc. If you are creating breakpoints only to use the Run button, you can have 3 breakpoints, but no more.
2. The stopwatch is not available in the hardware debugger.
3. When you start a debugging session in the hardware debugger in addition to assembling the code (if necessary) the PICKit 3 will also have to reprogram the chip. When it does, the code will start running.

One note: You can't switch between running the program and debugging the program in the hardware debugger while the program is actually running in hardware. If you are running the program, push the "Hold in Reset" button before you switch to the debugger:



Similarly, make sure that when you are using the debugger you select “Finish Debugger Session from the Debug menu before you run the code..

Aside from those caveats, everything works pretty much the same as it does in the simulator. You can step through code, establish breakpoints, watch register values, etc. This is an extremely powerful set of tools to have in a \$45 programmer. There was a time when this capability cost much more.

TRY THIS: Use 6 LEDs to represent the red, yellow and green lights of a traffic intersection. Let one direction be north/south and the other be east/west. Start with the red light of north/south on and the green light of east/west on. After 3 seconds, have the green eastwest light go off and the yellow east/west light come on. After 1 second, have the yellow east/west light go off and the red east/west light come on. Also at that point have the red north/south light go off and the green north/south light go on. Then repeat the sequence allowing the traffic to go the other direction. Note this should work just like a real traffic intersection, the lane that is moving will be warned with a yellow before it is required to stop and the lane that is not moving will not start moving until the other direction is stopped. Use the simulator to get the timing right on the light changes.