# CMPE 12L Uno32/PIC32 MPLAB Compiling, Assembling, and Debugging

Prof. Matthew Guthaus

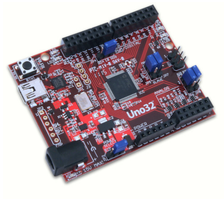
October 31, 2014

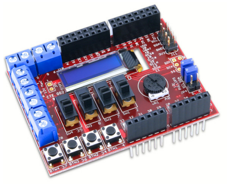
## 1 Requirements

This lab assumes that you have:

- MPLABX IDE v2.15
  `http://www.microchip.com/pagehandler/en-us/family/mplabx/`
- MPLAB x32 cross-compiler v1.33
  `http://www.microchip.com/pagehandler/en_us/devtools/mplabxc/`
- PICkit3
  `http://www.microchip.com/Developmenttools/ProductDetails.aspx?PartNO=PG164130`

- Chipkit Uno32
  `http://www.digilentinc.com/Products/Detail.cfm?Prod=CHIPKIT-UNO32`

  
- Chipkit Basic IO Shield
  `http://www.digilentinc.com/Products/Detail.cfm?&Prod=CHIPKIT-BASIC-IO-SHIELD`

- Two USB cables (one for PICkit3 and one for Uno32)
- **Header soldered to Uno32 debug port and cable to connect to PICkit3**

## 2 Introduction to MPLAB X IDE

The MPLAB X is an Integrated Development Environment (IDE) that is an interface to a full C/C++ compiler. This is software is available on Windows, Linux, and Macs. Our ISA is MIPS 4k, but it is implemented as a PIC32 microcontroller by Microchip Technologies and used in an Uno32 board by Digilent, Inc.

The general compilation of a high-level language such as C/C++ works as follows:

```
C/C++ --Compiler--> ASM --Assembler--> OBJECT --Linker--> ELF
```

Your high-level source files (with suffix .c, .C, .cpp, etc.) are converted by a compiler into assembly language files (.s, .asm, or .S). MPLAB requires .c and .s for C and assembly, respectively, in order to work properly. These ASM files are assembled into machine files which are called object files (.o, .a, .so or ELF). Multiple object files including your program and pre-made library files are typically "linked" into a final executable. The executable object file, usually in Executable and Linkable Format (ELF), is similar to binary files in previous lab assignments and can be run by the operating system. In windows, this is slightly different (and incompatible) and is known as an "exe" (executable) file. Your final file is also available as an "Intel HEX" file with the extension .hex that can be programmed into the PIC32 microcontroller flash memory. This is the closest thing to the .bin files that we saw in LC-3.

## 3 Creating a new project

For this part, you will use another simulator. Do not yet connect your board.

- Open MPLAB X IDE in the CMPE 12 folder.
- Create a new project in MPLAB X (Select File → New Project ) using the standard Microchip Embedded → Standalone Project option.
  - Select the PIC32MX320F128H device under the 32-bit MCUs (PIC32) family.
  - Select the simulator (6th option)
  - Select the XC32 compiler. If the XC32 compiler isn't an option then the XC32 compiler is most likely not installed. If it is your computer, install it now. If it is a lab machine, alert the TA/tutors and switch machines.
  - Choose a project name such as "Demo" or something similar.
  - Select a Project Location to be the parent directory for this project (such as your personal disk space or flash drive). The project directory will be created for you automatically by MPLAB X.

  - The Set as main project option will be enabled by default. The main project will be highlighted in bold in the projects listing and indicates which project is affected by the toolbar buttons and menu actions.

- You can now add files by right-clicking on the “Source Files” folder in the left panel, selecting “Add Existing Item...”, and specifying the demo.c that was provided to you. Later, you can add a a new file by selecting New → Assembly File, New → C Source File or New → C Header File.

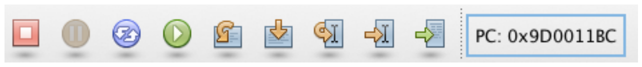
## 4 Building and Debugging

You can build the project by clicking the hammer which will result in a file under

$$YOURLAB\backslash dist\backslash default\backslash production\backslash YOURLAB.X.production.hex$$

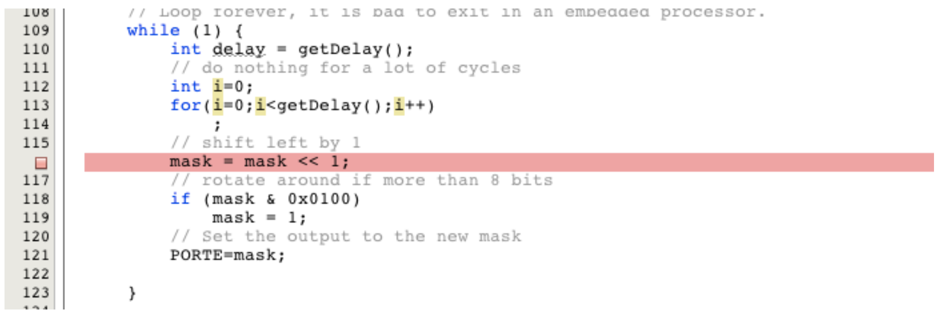
This file is in the Intel HEX file format (`http://en.wikipedia.org/wiki/Intel_HEX`). You should see black BUILD SUCCESSFUL text at the end of a successful build. A failed build will show a red BUILD FAILED message.

You can run it by clicking the “Debug Main Project” button ( ). This builds the project if it hasn’t done so and runs the debugger. The debugger will open a new toolbar that looks like:



with the controls, in order: Stop, Pause, Reset, Continue, Step Into, Step Over, Run to Cursor, Set PC at Cursor, and Focus Cursor at PC. The current program counter (PC) is shown on the right.

You can click on the line numbers to add a “break” symbol as shown here:



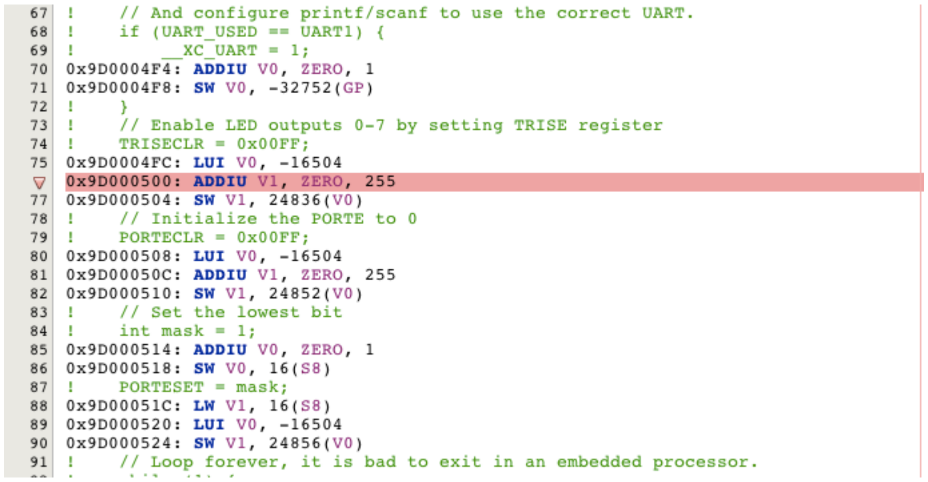
```
    // Loop forever, it is bad to exit in an embedded processor.
    while (1) {
        int delay = getDelay();
        // do nothing for a lot of cycles
        int i=0;
        for(i=0;i<getDelay();i++)
            ;
        // shift left by 1
        mask = mask << 1;
        // rotate around if more than 8 bits
        if (mask & 0x0100)
            mask = 1;
        // Set the output to the new mask
        PORTE=mask;

    }
```

If you run a program, it will automatically stop at that line before executing the line.

Put a breakpoint on the line that shifts the mask (“mask = mask << 1;”) and press the “Debug Main Project” button. The program should halt on this line and highlight it in green. If you click on the Variables tab, you can see the address and the value of the mask. Press the Continue button a few times and see the value change.

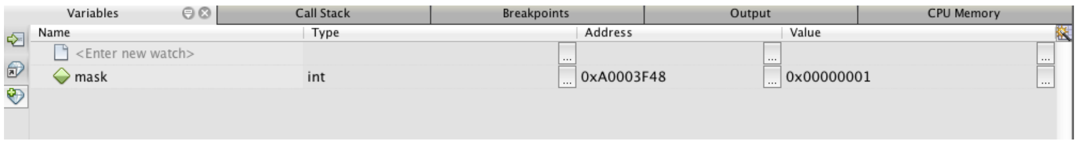
If you are in a current debug session, you can also view the disassembled output of your program by selecting Window → Debugging → Disassembly as shown here:



```
!    // And configure printf/scanf to use the correct UART.
!    if (UART_USED == UART1) {
!        __XC_UART = 1;
0x9D0004F4: ADDIU V0, ZERO, 1
0x9D0004F8: SW V0, -32752(GP)
!    }
!    // Enable LED outputs 0-7 by setting TRISE register
!    TRISECLR = 0x00FF;
0x9D0004FC: LUI V0, -16504
0x9D000500: ADDIU V1, ZERO, 255
0x9D000504: SW V1, 24836(V0)
!    // Initialize the PORTE to 0
!    PORTECLR = 0x00FF;
0x9D000508: LUI V0, -16504
0x9D00050C: ADDIU V1, ZERO, 255
0x9D000510: SW V1, 24852(V0)
!    // Set the lowest bit
!    int mask = 1;
0x9D000514: ADDIU V0, ZERO, 1
0x9D000518: SW V0, 16(S8)
!    PORTESET = mask;
0x9D00051C: LW V1, 16(S8)
0x9D000520: LUI V0, -16504
0x9D000524: SW V1, 24856(V0)
!    // Loop forever, it is bad to exit in an embedded processor.
```

You can add break points just like the C program by clicking on the line number of an assembly instruction. In the disassembled output, you can see original C program in the comments and the assembly instructions that it translates to in the output.

Once the program is running, you will see a number of tabs on the bottom. Click on the tab labelled "Variables" and you can expand some by clicking on the triangle. If the program used more variables (unlike ours), you will see something like this:



| Name | Type | Address | Value |
|---|---|---|---|
| <Enter new watch> | | | |
| mask | int | 0xA0003F48 | 0x00000001 |

Since you previously set a breakpoint, it will show all the variables currently in the scope of the current line. In this case, it is just the mask integer. For this, it shows the address of the memory location and the current value. If you click on the "CPU Memory" tab, it will show you all the special CPU registers including the general purpose registers and the contents that they hold. Similarly, if you click on the "Call Stack" tab, you can see the state of the function calls from main down to the inner most call. The "Breakpoint" tab just lists all the breakpoints in our program.

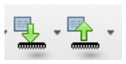
One more useful feature of the simulator is the Windows → PIC Memory Views. This can let you look at the instruction (called execution) memory, data memory, peripheral memory (I/O registers), etc. It is much like the LC3 simulator, but the whole PIC32!

It is VERY slow when you have a lot of instructions to simulate, because the PIC32 is quite complicated and your program will often be many millions of executed instructions. Remember, it executes 80M instructions per second but the simulator needs to execute thousands of instructions to simulate a single instruction along with the datapath, memory, and peripherals! In most cases, you will want to run on the hardware directly instead.

## 5 Running on the Uno32

Now that you can simulate code, let's run on the PIC32 processor that is on the Uno32 board! Your TA should have instructed you how to set this up at the beginning of lab. Connect the board to your computer with the USB cable. Also, connect the PICkit3 USB to the computer and make sure the header end of the PICkit3 is also plugged into the angled header on the Uno32 board. The first USB just supplies power to the Uno32 while the PICkit3 is going to allow us to load programs and debug them. If all is correct, the red power LED on the Uno32 should be lit. On the PICkit3, the green power LED should be lit, the blue active LED should be lit and the green status should be lit. If not, try flipping the cable between the PICkit3 and the Uno32 board – it is probably backwards.

If you already have the project created, right-click on the project folder on the left-hand side, select Properties, and select the PICkit3 under Hardware Tools. This will switch MPLAB from using the simulator to the real hardware. You can go back and select the simulator later if you want to debug without your board. You will still see the same "Debug Main Project" button as before, but you will now see two new toolbar options that look like:



The left one lets you buid your project and upload it to the Uno32's flash memory. (The second one will load the Uno32's flash memory into the IDE – we won't use it in class.)

Running your project in debugging mode is very similar to the way you used the simulator in the previous parts. Clicking the "Debug Main Project" button will NOT run the simulator – it actually runs the program on the PIC32. However, it can still set breakpoints, query register values and perform similar operations to the simulator using the hardware debug port on the processor. The PICkit3 is a special hardware "debugger" that allows this to be done by setting a special interrupt when the PC or data in a memory location matches a certain value. This is really handy because it is MUCH, MUCH faster than the simulator.

The Program Device button actually puts your code in flash memory (just like a USB memory stick) that is located on the processor chip so that it will run even with the PICkit3 disconnected. If you see the error: "Target device was not found. You must connect to a target device to use PICkit 3." then you might have to swap one end of the ribbon cable because it is connected the wrong way. Also press the reset button on the PICkit3 or replug in its USB to reset it. Otherwise, you should see: "Programming... Programming/Verify complete" in the PICkit 3 tab.

# 6 PIC32 Assembly

## 6.1 Reference Material

There is a lot of material on PIC32 and we can't require you to read it all. Part of the course is learning to find the information that you need in the given documents like a real engineer will do. This involves "sifting through" reference materials. The table of contents is your friend!

There should be coverage from the lectures as you complete this lab, but in the meantime, you can also look at this web page to see detailed information on the MIPS assembly architecture and assembly language:

`http://en.wikibooks.org/wiki/MIPS_Assembly`.

There is also an extra resource at:

`http://www.johnloomis.org/microchip/pic32/resources.html`

There is a ton of information on PIC32 at the manufacturer's website here:

`http://www.microchip.com/pic32`

Specifically, if you click "Resources" and "Training" there are extra videos (not required, but very interesting). The 4 minute video on the execution pipeline and the 18 minute architecture overview are both good, but are not necessary for this assignment. If you click "Data Sheets" on the left, the PIC32MX3/4 Data Sheet has the complete information on the hardware. We provide some of these documents in the Resources/References section on eCommons.

## 6.2 C Program Structure

There is a lot of initialization that must go on for the Uno32 to be ready run your code. This involves setting up the configuring the clock frequency, configuring the peripheral clock frequency, setting up the stacks, setting up the interrupt vector tables, and most importantly setting up the serial port for debugging. Most of this is done with some X32 compiler libraries. The demo.c program provided with this tutorial calls the setup routines for you in the C language.

If you write an assembly program, it is best to call it as a function from the C program at the end. The function should be declared as external using

```
extern void myfunction();
```

for example, then you can call it at the end of main like any normal C function:

```
myfunction();
```

Remember, the extern declaration tells the compiler or assembler that the symbol is not yet defined but will be resolved when another object is linked to the current one. By default, MPLAB X IDE creates one object per file. In this case, you will have an object from the .c file and one from your .s file that are linked together into one executable.

## 6.3 Assembly Program Struture

Your assembly program should follow this template:

```
.ent myfunction
.text
.global myfunction
myfunction:
  /* Your assembly code goes here */
.data
  /* Any data goes here */
.end myfunction
```

There are a few things of importance in this program structure:

- .global myfunction
  This puts the symbol "myfunction" into the symbol table of the object so that external code (specifically main.cpp) can call the function.

- .text
  This specifies the “instruction” part of your program.
- .set noreorder
  This (optionally) tells the assembler not to reorder instructions in a “smart” way to maximize performance.
- .ent myfunction
  This specifies where the entry to myfunction starts.
- myfunction:
  This is the label of the address for our function. Note that labels here require a colon after them.
- .end myfunction
  This specifies the end of your file.
- .data
  This specifies the “data” part of your program.

## 7 Debugging over Serial

Once we add this code, assemble our program, and upload it to the PIC32, we can start a terminal program to communicate between the host and the board. This will send every “printf” or “puts” output to the serial port which you can display.

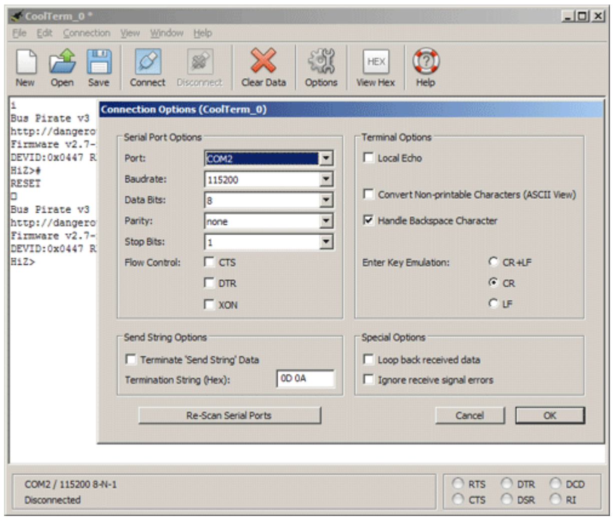
Every platform has many different terminal programs. In lab, we have CoolTerm installed. I include some others for your convenience, but TA/tutor expertice will be very limited.

### 7.1 CoolTerm

CoolTerm is available on Windows, Macintosh and PCs. It is also installed on all the lab machines.

Determining your serial port number This is done by opening the start menu and searching for ”devices”, which will yield the ”Devices and Printers” menu within the Control Panel. Open this. Under the Unspecified section you should see ”FT232R USB UART” and double-click on it. Switching to the ”Hardware” tab you will see a line like ”USB Serial Port (COM42)”, this will tell you what COM port the Uno32 is connected over. Remember this number and all the Properties window and the Devices and Printers window.

Once you determine the port, launch CoolTerm. Select the “Options” button and you will see something like this:


CoolTerm_0 *
File Edit Connection View Window Help
New Open Save Connect Disconnect Clear Data Options View Hex Help
Re-Scan Serial Ports
Cancel
OK

```
1
Bus Pirate v3
http://dangero
Firmware v2.7-
DEVID:0x0447 R
HiZ>#
RESET
□
Bus Pirate v3
http://dangero
Firmware v2.7-
DEVID:0x0447 R
HiZ>
```

Connection Options (CoolTerm_0)

Serial Port Options

Port: COM2
Baudrate: 115200
Data Bits: 8
Parity: none
Stop Bits: 1
Flow Control: ☐ CTS ☐ DTR ☐ XON

Terminal Options

☐ Local Echo
☐ Convert Non-printable Characters (ASCII View)
☑ Handle Backspace Character
Enter Key Emulation: CR+LF [✓ CR] LF

Send String Options

☐ Terminate 'Send String' Data
Termination String (Hex): 0D 0A

Special Options

☐ Loop back received data
☐ Ignore receive signal errors

COM2 / 115200 8-N-1
Disconnected

RTS DTR DCD CTS DSR RI

Select the previous COM port and set the baud to 115200. Everything else should remain the same. If you now press OK and select Connect you should see "Hello, World!" over and over from the demo program. Remember to Disconnect after you are done. You can only connect one terminal at a time.

## 7.2 screen

On a Mac OSX or Linux machine, install the "screen" package. You can connect to the terminal port from the command line with:

```
screen /dev/tty.usbserial-A1009SCN 115200
```

except that our specific USB serial device name may change depending on the port. This was on my Mac and it will usually be /dev/tty.usb something. On a Linux machine, the names are typically /dev/ttyUSB something instead. Find the port by trying these:

```
ls /dev/ttyUSB*
ls /dev/tty.usb*
```

To disconnect in screen, type "Ctrl-a k" and answer Y to kill. Note, if there is more than one copy of screen running (i.e. you don't kill the last one), it will give an error when you try to connect such as "Resource busy". You can re-attach to the terminal by typing "screen -R" and then properly kill it.

## 7.3 Printing output to stdout in assembly

The previous serial output used a C function "printf", but how can we do that from our assembly program? Simple. Call the function! For example:

```
.text
la $a0, myStr
jal puts
```

```
la $a0, myStr2
li $a1, 5
jal printf
nop

.data
myStr: .asciiz "Hello, world!\n"
myStr2: .asciiz "Hello, world! %d"
```

This uses either one or two arguments. In the first case, it simply prints the string until a null character. The .asciiz directive is identical to .stringz in LC-3. The second example uses a special format specifier in the print string which will substitute the second argument into the string as ASCII. This is useful for display data or address values!

Remember, the .text and .data directives will tell the linker where in memory to put the instructions and data, respectively. It is better to do this than to intersperse data with our program like we did in LC-3.

Another note: The above jal instruction will clobber the ra register, so you must save it if you want to ever return from your assembly code. Similarly, all caller save registers must be pushed on the stack.