ME 461 Laboratory #0
Soldering and Introduction to the Hardware and Software

Goals:

1. Learn to solder and practice by soldering the base set of components to the breakout board.
2. Become familiar with the MSP430 hardware and the Code Composer Studio v6.0 Integrated Development Environment.
3. Build and run your first microcontroller program.
4. Become familiar with the available debugging and communication tools.

Exercise 1:

In this exercise, you are going to solder components to your breakout board. The TA will distribute the printed circuit boards, which will have some components already soldered to them. You will also receive an MSP430 Launchpad for use debugging and communicating using a serial port. The TA will give a short soldering demo and show you where to find wire and other components in the lab. The list of components you are to solder is given below. Use the demo board in lab as your main reference.

- 0.1µF, 0.01µF, and 10µF capacitors and 6.8µH inductor for power line filter
- 8 large LEDs and 8 220Ω resistors arranged in a circle
- 4 pushbutton switches for intersection pressure sensors
- 1 LED and 1 220Ω resistor by the 4 pin debug connector
- 3 LEDs by the FTDI chip

Wire the LEDs to the port 1 pins in a logical order. Wire one side of each switch to port pins P2.4-P2.7. Wire the opposite side of two of the four switches to DGND and the other two switches to DV_{CC}.

You will notice that your breakout board has two sets of GND and V_{CC}. The DGND and AGND are digital and analog ground, respectively, and the DV_{CC} and AV_{CC} are digital and analog power, respectively. Use the digital power set for digital applications and the analog power set for analog applications. Ask your TA if you are unsure as to which category a given application belongs. Separating power supplies helps to increase the accuracy of analog measurements and minimize digital noise effects. When you are done soldering, **show your TA** your breakout board.

Exercise 2:

The ME461 laboratory is based on the MSP430F2272 low-power 16-bit microcontroller. The microcontroller (MCU) has several integrated peripherals, a 16-MHz RISC core, onboard RAM and Flash memory, and 32 general-purpose input/output (GPIO) pins. The IDE (integrated development

environment) we will use for writing, compiling, and downloading programs is Code Composer Studio (CCS) v6.0.

To create new projects, you will first need to create a new workspace on your computer's C:\ drive that you will use throughout the semester. Start CCS and when prompted to select a workspace, type "C:\<your netID>\workspace\". When CCS loads, close the welcome screen by clicking the X next to the "Welcome" tab.

To create a project for the MSP430F2272 in CCSv6.0, there is a project creator at N:\msp430\me461\Fall16\ProjectCreator. When you use the project creator, name your project Lab0, or something similar (no spaces or special characters), and place it in the directory you just created above (C:\<your netID>), but NOT inside the "workspace" directory. Next, go back into CCS and choose *Project → Import Existing CCS/CCE Eclipse Project*. With "Select search-directory" selected, browse to the new project folder in your workspace and click Ok. Your project should appear checked in the "Discovered projects" section. Click "Finish" and your project will be added to your workspace (it should appear bolded in the "C/C++ Projects" file tree on the left side of CCS as "Lab0 [Active – Debug]," or whatever you named your project.

The CCS interface is based on "perspectives", which are customizable layouts for developing and debugging programs. The two perspectives we will use this semester are the "C/C++" and "Debug" perspectives. Their names are self-explanatory: the C/C++ perspective is used for developing programs in C/C++, and the Debug perspective is for debugging those programs. Switch between the two perspectives by using the buttons in the top-right corner of the window.

On the left side of the CCS window you will find a project explorer that lists the files in your project. Most of the work you will do can be done in the single "user_LabX.c" file, but you are free (and indeed encouraged) to create additional header and source files as you see fit by choosing *File → New →* (*Header file* or *Source file*) while in the C/C++ perspective. Now, let's build and download your program. Choose *Project → Build All*. Note that the Console window at the bottom of the screen is filled with cryptic output. You may need to refer to the console for descriptions of errors, but you will more likely find the *Problems* window more helpful to that end.

Let's explore what happens when you generate a compiler error. Add a line immediately before the first **`#include`** statement in the program and type "`Hello!`". Choose *Project → Build All*. Note that the *Problems* tab now contains a description of the compiler errors and the lines at which they occurred. Double-click the first item in the list of errors and the statement that *the compiler believes* is the cause of the error is highlighted. Remove the line you added and build the project again.

Connect your MSP430 Launchpad to an open USB port on the computer via Mini USB cable and wait for any drivers to install. Connect the Launchpad to your breakout board using the provided cable. **Pay attention to the paint on the plug and header (colors should all match on the plugs and connectors)**, since the connections will not work in reverse. The power indicator LED next to the plug should light. By making this connection, you are providing power and the debug connection to your breakout board. The "EMULATION" half of the Launchpad functions as a debugger. Using a second USB cable, connect to

the USB socket on your breakout board. You will again need to wait for Windows device drivers to install on the PC. This is a virtual RS-232 serial port that you will use to communicate information to the PC. Now choose *Run* → *Debug*. The program has now been downloaded to the MCU's flash memory and is ready to run.

Open the COMx serial port on the PC by opening Tera Term (shortcut on the desktop). In Tera Term, click "Serial" and choose "COMx:USB Serial Port [COMx]" for the port in the window that pops up. Click Ok and you are connected. Now, in CCS, choose Run → *Resume* or press F8 (you can also click the green arrow on the debug toolbar). You should see the LED connected to P1.0 blinking and the text "Hello" followed by a counter printing to the Tera Term window. **Show your TA** the LED and terminal output. Congratulations, you just successfully programmed your first microcontroller!

Exercise 3:

Now, you are going to modify the default program slightly, but first you need to save the work you've done so far. End your debug session by choosing *Run* → *Terminate* or by clicking the red square icon on the debug toolbar. Use Windows Explorer to find your project directory. Copy and paste the "user_Lab0.c" file and name the copy "user_Lab0ex3.c". Back in CCS this new file should have been automatically added to your project but if not, right-click on your project name and select "Add Files". Find "user_Lab0ex3.c" that you just created and click "Link to Files". Then a dialog will appear and make sure to check "Create Link Locations Relative to" and select "PROJECT_LOC." In your project you will now have two C files to choose from. Right click on the original C file "user_Lab0.c" and select "Resource Configurations -> Exclude File(s) from Build" and check both Release and Debug. This method will allow you to build on your programs without losing your work and will provide an easy way to trace your progress throughout the semester. **Follow this methodology from this point forward when completing lab exercises.** Of course, commenting lines in your program is still a perfectly acceptable way to quickly remove statements from the build process, but it is not recommended for large changes.

This time, build and download the program by clicking on the button that looks like a beetle (De-"Bug") in the center of the top toolbar. Run the program for a few moments. Now, place a breakpoint at line 74 of the program by double-clicking in the gray margin to the left of the line numbers in the editor window. Notice that execution has halted at the line where you placed the breakpoint. Highlight the reference to the `timecnt` variable in the argument list of `UART_printf`, right-click and choose *Add Watch Expression*. Notice that the value of the `timecnt` variable is displayed in the *Expressions* window. Right-click the expression and choose *Change Value*. Set the value to zero. Now, remove the breakpoint by double-clicking on it, and show the terminal window. Resume execution by clicking the *Resume* button (which resembles the "Play" button on a video player) in the *Debug* window. You should see in the terminal window that the count has restarted from zero. Press the *Halt* button (which resembles a "Pause" button) in the *Debug* window and notice that you may view the value of `timecnt` in the *Expressions* window again. You may also hover the mouse over a variable in the editor window while execution is halted to view its value. Though the details of its contents are outside the scope of this lab assignment, view the *Registers* (choose *View* → *Registers*) and navigate through some of the contents.

The *Registers* window is a special "Watch" window for viewing the contents of the registers on the F2272. You will learn more about registers in lecture and future labs.

Exercise 4:

Throughout the course you will use two methods for exchanging data with the PC. The first involves the function `UART_printf`. The syntax for calling `UART_printf` is exactly the same as the standard `printf` function, except that the output is printed to the serial (UART/RS-232) port, rather than the console as in a PC-based application. If you are not familiar with `printf`, refer to any of the C reference materials available in lab or online. Experiment with the `UART_printf` function by adding more expressions to the argument list and expanding the format string accordingly. Try it for several different datatypes and view the results in the terminal window. **Show your TA.**

The `UART_printf` function is handy for quickly displaying variable values, but what if we want to send data from the PC to the microcontroller? Or, what if we want to display more than a few numbers (say, 10) and record their time histories? This can all be done by using the second method for communicating with the PC: the serial I/O application. The application is designed for bidirectional real-time data exchange between the PC and your microcontroller and is available on the course website or on N:\me461\serialIO\. Start the application and take note of the layout. In the left pane are 10 text boxes where the user can enter numbers to be sent to the microcontroller. In the right pane are 10 display boxes for data sent to the PC from the MCU.

On the microcontroller, data is sent to the PC by calling the `UART_send` function and received by calling `my_scanf`. Both the `UART_send` and `my_scanf` functions are called with a variable number of arguments. The `UART_send` function requires as its first argument the number of values to be sent. The arguments following the first are the expressions themselves. **The expressions to be sent must evaluate to floating-point numbers. If an integral type is to be sent, it must be typecast to a `float` before sending.** Also, the number of values to send is limited to 10.

The `my_scanf` function syntax is similar to the standard `sscanf` function, except for the format string. The `my_scanf` function requires as its first argument a pointer to an array where the data is stored. This argument should always be `rxbuff`. The arguments following the first are addresses of **floating-point** variables in which to store the data. **Once again, if integral data is desired, it must be cast from a floating-point type.** The number of arguments following the array pointer should match the number of checked boxes in the serial application. The `my_scanf` function should always be called in the while loop, and always from inside the `if(newmsg)` block.

These functions and the I/O application are designed to send full-precision floating-point data back and forth as opposed to ASCII-formatted text strings. For more details regarding the protocol, see the code for the functions and the Windows application or ask your TA. The two methods for serial communication (the terminal window and the serial I/O application) are mutually exclusive. That is, **you cannot use both the `UART_send`/`my_scanf` functions and the `UART_printf` function in the same program as they both use the same serial port**.

Now you are going to experiment with data exchange using the serial I/O application. In user_Lab0ex3.c, comment out the line in your program that calls UART_printf and uncomment the line that calls UART_send. Also, add a global **float** variable called temp and paste the following code inside the if(newmsg) block.

```
my_scanf(rxbuff,&temp);
timecnt = (unsigned long)temp;
```

Disconnect from the COMx port in the Tera Term terminal window and connect to the COMx port (it will be the same port number that you used in Tera Term) in the serial I/O application at a baud rate of 9600. Build and run your program. You should see the counter value displayed in the first box in the right pane of the serial I/O application. Now, put a check mark in the first checkbox in the left pane, write a value of zero in the text box, and click *Send*. The counter value in the display box should restart at zero. **Show your TA.**

You can modify the labels above the boxes to help you remember which variables you are viewing or sending. You can also store and recall the labels and the left-pane values by choosing *File* → *Save* and *File* → *Load*. You may practice using the *Save* and *Load* features if you wish. Using this application you can observe and effect changes on program variables in real-time. The recommended rate for displaying data using UART_send is ~10 Hz, but this is not a limit or a rule. You will learn more about how to change the rate at which events occur in Lab 2.

One final point: as you should already be aware, we are using a fixed-point MCU; that is, it has no floating-point unit (FPU). Do not be led astray by the fact that UART_send and my_scanf are designed for floating-point data; this is done to maintain the highest level of compatibility. You must exercise *extreme* caution when using floating-point types on a fixed-point processor due to the extraordinary amount of code space and processing time required to work with these non-native types.

Correspondence:
Dan Block: d-block@illinois.edu
Steve Keres: Stephen_L_Keres@whirlpool.com