Getting started with theATmega2560

Lab 1 & 2

Performance Check

_____ (45) Procedure 1-3

_____ (45) Procedure 4

_____ (90) Procedure 5

Introduction:

You used the Arduino Mega in your previous classes. The Arduino Mega has the ATmega2560 microcontroller, which we will use throughout this semester. This lab is to get you familiar with the development environment, learn how to use the tools, the simulator, and start building your own light and switch board. You will use this board throughout the rest of this semester, so do it right. If you have your light and switch board from previous course, you may use it.

There are detailed instructions in this lab. This lab can be used as reference for future labs. You are expected to comment your codes, follow proper C structures. Reference lecture notes on the required formats.

Although embedded software developers would like to think that they spend the bulk of their time designing software and coding it, that is not the case. Almost all actually spend more time getting the code to function correctly when developing software solutions in embedded (and indeed all) systems - a process that is called "**debugging**". Tools that aid in this process are vital. The compiling process can reveal simple syntactical coding errors but code that compiles yet does not function correctly can be difficult to debug. Indeed some bugs only surface later when the embedded system is being used in the designed application. How many patches and updates have you installed on devices you own? There are many methods to debug the code, one of which, you have learned from previous classes: use stdio.h functions such as printf() to output the values of variables to a PC screen via a terminal emulator. Another is to use PORTs to output data to a display device such as an LCD or LEDs. Although the practice of adding **printf()**

calls to log values is still amazingly common, it is quite problematic for embedded applications. There are several key issues:

1. Most implementations of **printf()** are quite large. The function needs to address a very wide range of formatting situations and that takes a lot of code. This does not matter in a desktop software context, but memory is rarely in abundance in an embedded system.
2. There is the question of where the output of **printf()** actually goes. Many embedded systems have nothing that looks like a "console" and directing output back to a host computer can be challenging.
3. Every time you want to change the variables you are looking at, a complete rebuild and download of the code is required.
4. Lastly, of course, formatting and sending the output (somewhere) takes time. If it is a real time application, this overhead can introduce problems.

Real debuggers:

So, for most systems, using **printf()** as a debug tool is not viable and the use of a "real" debugger makes more sense. Debuggers are available from a wide variety of suppliers of embedded software development tools and can take a number of different forms, largely characterized by how/where the code gets to execute. This ability to step through code and observe changes to system states and variables is **extremely** valuable.

Simulation:

It can be useful to start debugging before target hardware is available. One approach is to use some form of simulation. This term covers a broad spectrum of technologies. One example is of an IDE, such as the Atmel Studio, that includes a simulator and debugging tools that provide native execution of code on the host computer (e.g. PC) and instruction set simulation. Simulation permits code to be exercised without the need for the hardware and allows the contents of memory, state of PORTs, values of variables, ALU register content, peripheral registers etc. to be observed and their values checked as the code is single stepped or as an individual section of the code executed. This is a useful method for you as you work through the labs this semester.

In Circuit Emulator (ICE)

Many embedded CPU manufacturers include JTAG debugging facilities into the micro IC itself. In addition, hardware devices called ICEs (**I**n **C**ircuit **E**mulators) can be used to connect to the embedded system hardware from the IDE – usually through JTAG connectors – in order to run the code, single step through the code and observe the state of memory, registers and variables.

Generally, inclusion of a JTAG port has a minimal impact on system design. Debuggers that support a JTAG connection are widely available e.g. the Atmel AVR JTAGICE3.

This simple solution of attaching a debugger to a target system or an evaluation board using JTAG can give ready access to all the target data and addresses the five problems associated with **printf()** debugging thus:

1. There is no extra code on the target.
2. Data is displayed on the host computer without effort.
3. Code only needs to be rebuilt to fix bugs.
4. A JTAG connection is slightly intrusive in the time domain, but the impact is typically quite small.

Reference: "Who needs a debugger?" Colin Walls, Embedded.com, March 03, 2015

Objectives:

1. To become familiar with the Atmel Studio 6.2 / 7 Integrated Development Environment (IDE) by setting up and building a project (based on provided source code).
2. Configuring Studio 6 / Studio 7 for code simulation and debugging
3. Single stepping through code
4. Downloading the resulting .hex file to the ATmega2560 flash memory and examining its operation.
5. To gain practice at writing .c programs incorporating functions.
6. To describe the operation of the light and switch circuit and provide a schematics, write a program that use the Arduino Mega, to generate a Sweep display.

Procedures:

Procedure A: Getting started, write a simple multi-module program that blinks LED13

1. Launch Atmel Studio 7
2. Go to the menu path **File → New → Project→ GCC Executable Project** to create a project in Studio 7 called **Lab1_IO** in a folder called **ECET279_Lab1_ intro**.

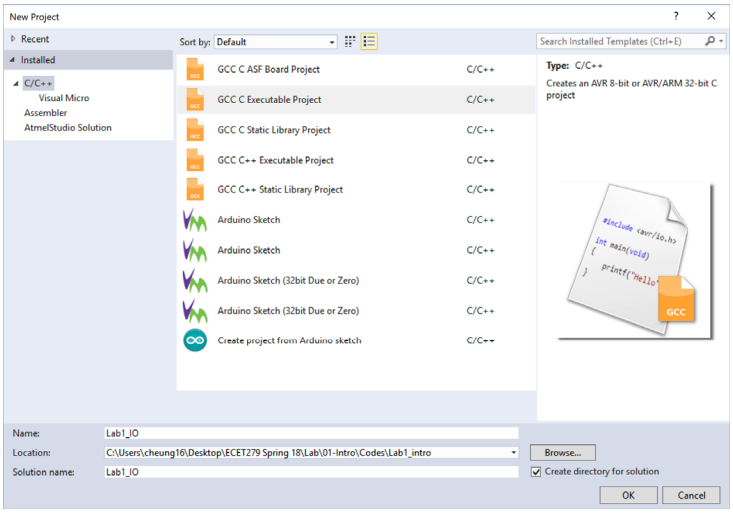
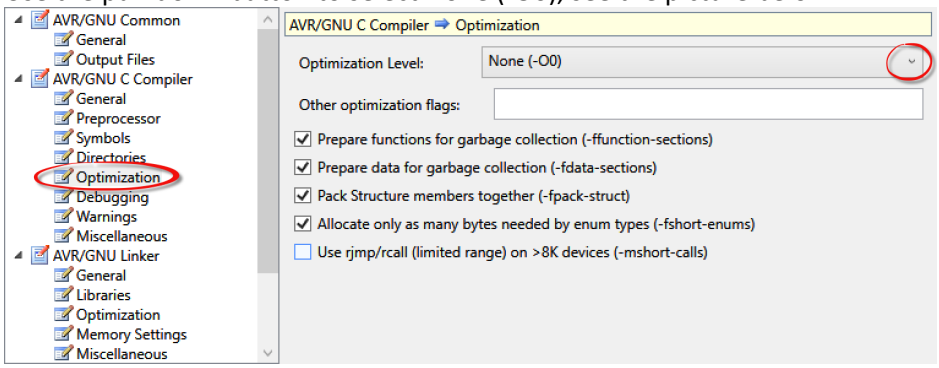


Figure 1: Atmel Studio New Project Open screenshot

3. Then select the device – Atmega 2560 in the next window. The project set-up is complete and shows the structure in the **Solution Explorer** window. The file <avr/io.h> is already included.
4. To enable simulations for later steps, select None for compiler's optimization.
   a. Select Project → Lab1_IO Properties → Tool Chain
   b. Select AVR/GNU C Compiler
   c. Use the pull-down button to select None (-O0), see the picture below
   d. 

   e. Select Save All to save this setting.

5. Select the main.c file, update the comment header with the information inside this picture:

```
/*
* FileName: Lab1_IO.c
*  version: 1
*
* Created: ____Time and Date_____________
* Author : Your Name
*
* Operations: Describe what this program do.
*      This is a multi-module program,
*          on-board LED is turned on every second
*
* Hardware Connection:
*  Atmega2560          Hardware
*  PORTx.x             LED13 Active high
*/
```

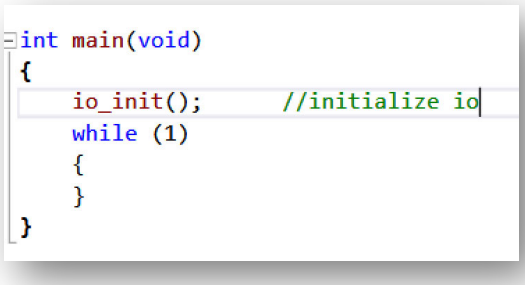
6. You need to find out where is LED13 connected to and put the information in the comment header. **Reference the attached Arduino Mega diagram at the end of this document.** This diagram is also available in Brightspace, inside the Additional Resources folder. **Find out the port and pin number for LED 13. This is the LED connected to pin 13 of the Arduino.**
7. Add a function called io_init, it contains port initialization for LED13.
   a. Add the following function, fill in the blanks of the port and pin number of LED13.

```
void io_init(void)
{
    DDR[ ] = [ ];       //LED 13 set as output
    PORT[ ] = [ ];      //turn off LED at initialization
}
```

   b. Place the io_init function's function prototype above the main function

```
// function prototypes
void io_init(void); //initialize io ports
```

8. We are now ready to call the function in the main function.
   a. Open the main.c file, call the io_init function inside the main function, above the superloop. Initialization codes should always be above the while(1) loop. Executed only once at the beginning of the code.



```
int main(void)
{
    io_init();      //initialize io
    while (1)
    {
    }
}
```

   b.
   c. Press F7 to build the solution. Clear any errors.
9. To blink an LED, add the _delay.h file, and define the F_CPU. Since this function is specific to this project only, they should be included in the main.c file. #define for F_CPU should be included as the first line of the code, right after the comment header.

```
#define F_CPU 16000000UL

#include <util/delay.h>
```

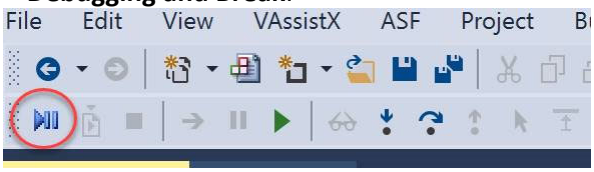
1. Write **one** line of code inside the while(1) loop to toggle LED13 on and off. Then add a time delay of 500ms after this line. Your code should only toggle the bit on and off, other bits in the same port should remain the same. We will use the simulator in the next procedure to verify your code. Be prepared to explain the operation of this line of code during check off.

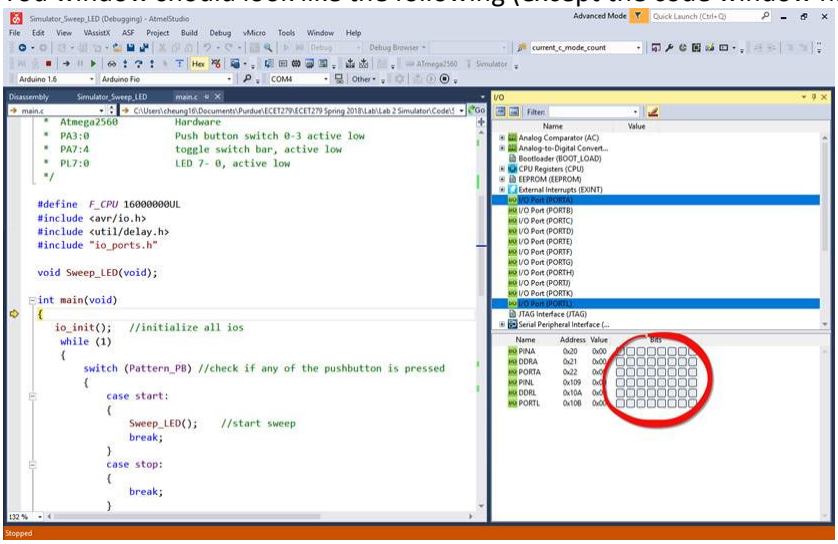
Procedure 2: Simulation implementation

In this procedure, we will run the code from previous procedure using the simulator, i.e. without the hardware.

1. In Studio click on Project and select Properties. Select the Simulator as the Tool and save the setting by clicking the floppy disc symbol in the toolbar.
2. Also under **Properties** select **Toolchain** then **Optimization** under the AVR/GNU C Compiler tab. Change the Optimization Level from Optimize (-O1) to None (-O0). Again save the setting.
3. Under **Tool**, use the pull-down menu to select **Simulator** as the debugger/programmer, then select Save All
4. When simulating code we CANNOT use the function _delay_ms(); (it will take too long) to generate any time delay because of the need to turn off the optimization, **so comment out ALL the calls to _delay_ms().**

5. Now build the project by pressing F7.
6. Start the debugger by clicking on **Debug** to open the debug window. Now select **Start Debugging and Break**.



7. Select **Windows** in the debug window and open an **I/O** window and a **Watch** window. Move the windows to positions so that you can still see the C code listing.
8. In the I/O window you will need to open PORTB for outputs to observe the state of the LEDs on PORTB. If you like to observe status of other ports, hold the Ctrl key while selecting the second port.
9. Make sure you have eight bits for each port instead of six bits.
10. You window should look like the following (except the code window has different code):
11. 

12. notice the 8 bits in the IO ports
13. Observe the position of the **yellow** arrow in the window displaying the C file text. This arrow indicates the NEXT instruction to be executed and corresponds to the program memory address in the processors program counter register.
14. Press **F11** to single step through the code. Step the code until it reaches the end of the io_init function. Notice the changes in PINn, DDRn, and PORTn. When the bit equals 1, the corresponding square turns red and its value (to the left of the square) changes accordingly.
15. Press F11 until you reach the main code, watch LED13 (PORTB.7) turns on and off as you step through the code. At the same time other bits in PORTB will not change.
16. What is the address of PINB, DDRB, and PORTB? _____
17. Be prepared to show the simulator operations during check off.

Procedure 3: Download code using Atmel Studio through USB

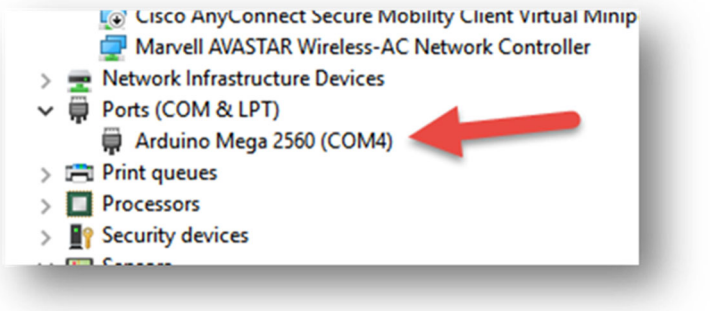
Once you verify the operation of LED13, you are ready to download the code to the hardware. Don't forget to remove the comment from the delay code. Follow these steps to download your code using Atmel Studio. You need to have AVRdude already installed. If not, download the

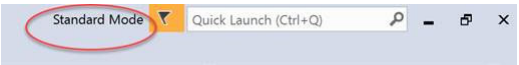
Arduino IDE desktop version (NOT THE APP version) to your computer before continue to the next step.

1. Find out which COM port the Mega is connected to**:**

    a. Start Device Manager

    b. Expand Ports (COM & LPT), in this case, COM4



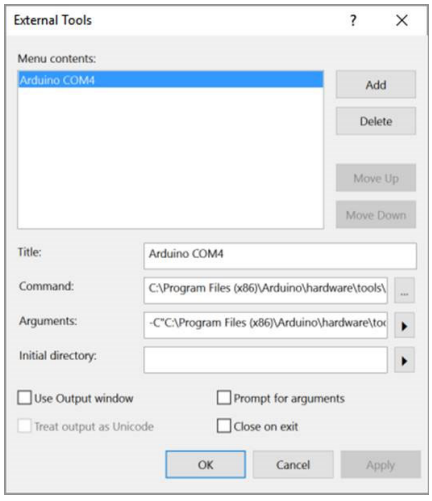
2. 

Set up Advanced Mode: On the top right hand corner, make sure you the User Interface Profile is in Advanced mode.

3. Set up Atmel Studio 7

    a. Tools → External Tools

    b. Fill in the Title, similar to the following window:



c. Please note: avrdude may not be installed at the same location, find out where and substitute where applicable. Use the windows search function to find this file “avrdude.exe”. Then copy its file location and append avrdude.exe to the command line. You may also download Arduino

IDE Desktop version (NOT THE APP), which includes avrdude. Below, an example of the command line.

C:\Program Files (x86)\Arduino\hardware\tools\avr\bin\avrdude.exe

. Please note: find out where the conf file is located and substitute where applicable. Substitute the COM4 with your COM number.

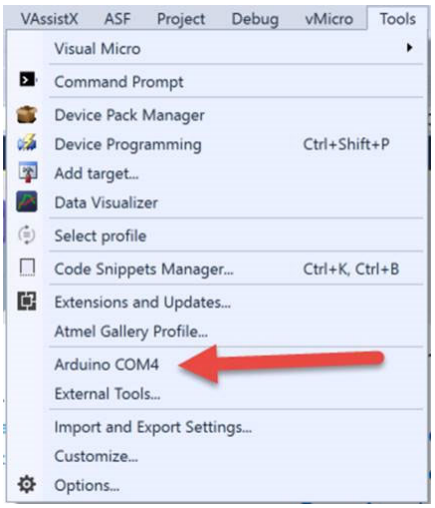
i. **The following text is in Brightspace, under Resources, AVRdude argument**.

ii. Copy and Paste the text to the Arguments location of the external tool, then change the location and COM number. **DO NOT COPY FROM THIS DOCUMENT, use the text format from the Brightspace text document.** The text below is to show you the content only.

-C"C:\Program Files (x86)\arduino\hardware\tools\avr\etc\avrdude.conf" -patmega2560 -cwiring -P\\.\COM4 -b115200 -D -Uflash:w:"$(ProjectDir)Debug\$(TargetName).hex":i

e. **Uncheck Close on exit**, then Click OK

4. Under the Tools menu, the Arduino COM4 tool appeared.

5. 

6. Download the hex file to the Arduino.
7. Show your instructor you are able to run the simulator, know where LED 13 is located, and download the code through Atmel Studio. There is no partial credit for each check off.

Instructor Check off
1. LED13 PORT and PIN location 2. How to toggle LED on and off with one line of code 3. Show Operations of simulator 4. Download code using Microchip Studio External Tool

Procedure 4:

You have completed the basic exercise of making a program that flashes a LED every 500ms, then download it through Atmel Studio to the microcontroller itself. The following procedures are simple code that you can use later to test your hardware.

1. You will need 8 LED outputs. Determine the output port you'd like to use.
2. Design a circuit which turns the LED on when the corresponding output bit is set to 1. Draw the schematics in the space below. If you are uncertain, consult with your instructor before wiring the circuit in your breadboard. If you already have your light and switch board, then you do not need to design the circuit. You need to determine when the output pin is set to 1, is the LED turns on or off. Draw the schematic of one of the LEDs in your light and switch board here, be prepared to explain the circuit operations, use proper symbol for each component in the circuit. You will need to look for datasheet for some of these components:

3. Modify the io_init function to include these LED outputs. Each LED should be set to off initially.
4. In the comment header to indicate the hardware connection information.
5. Write a function LED_Sweep inside the main file. Make sure to place the function prototype above the main function, the function definition should be placed after the main function. If you don't know where these locations are, ask your instructor or lab assistants.
    a. In the LED_Sweep function (You may not hard code each step in this function. Use one of the loop statements.)
    b. A LED is turned on, in sequence from right to left. Once it is turned on, it stayed on until all the LEDs are turned on. Then it turned off one at a time. This is similar to the volume bar.
    c. Use the simulator to verify your code's operation. Watch the video in Brightspace, under Labs → Lab 1&2 Bit operations Simulator and Debugger → Lab 1 Videos. The first video shows this sequence.

6. In the main function:
    a. Turn off all LEDs at power up
    b. Call the LED_Sweep function
    c. Repeat the sequence when the right most LED is turned on.
7. Build the project and fix all the errors or warnings.
8. Set up break points in your code to observe the LED turning on one at a time. Move the mouse to the line you'd like to stop the code, right click, and select Breakpoint and then add Breakpoint. Or you can move the mouse to the left gray area of the line, then left click. A red circle indicating the breakpoint.
9. Build the circuit and download your code, verify its operation with the hardware.

Instructor Check off

1. Hardware circuit operates properly
2. Demonstrate use of break point and steps in simulator
3. Schematic of a LED circuit, explain its operation

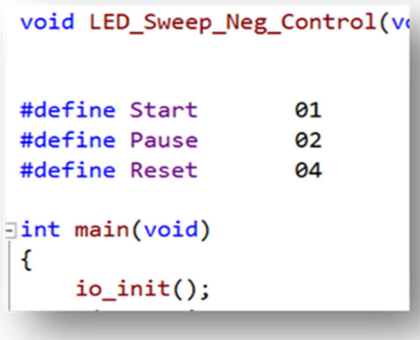
Procedure #5: Input pushbuttons

1. We will monitor three switches. Make a uint8_t variable called input_sw. This variable will store the input switch status. Declaring this variable should be placed above while(1) loop.
2. Design three input circuits. If you have the board from your previous class, you may use it. If you don't have a circuit, design one and then build it. If you are not sure, check with your instructor before building the circuit. Review notes from your previous classes for the circuit. Draw its schematic here. Decide when the pushbutton is pressed, what is PINx.y receives? 1 or 0?

3. Inside the while(1) loop, assign input_sw variable to PINx (x is the input port that you have selected from previous step). Input_sw should only equals to input pins that you've assigned, the rest of the pins should be assigned zero. Assume that the input switches are connected as positive logic. i.e. when the switch is pressed, controller receives a 1. This is accomplished by using bit masking with the use of << operators. For example:

   three input switches are located in PORTA2:0
   Input_sw = PINA & ((1<<PA0) | (1<<PA1) | (1<<PA2));
   This statement is equal to Input_sw = PINA & 0x07, but it is easier for readers to know which bit is being used. Therefore, it increases readability of the code.

5. After reading input_sw variable contains the switch values, we can use it to start the LED_Sweep sequence.
6. Your code should be easy to maintain and easy to read. Let's use #define to identify the various switch functions. This will increase readability and maintainability of the code. E.g. if the first switch located at PINA.0 is used to start the LED_Sweep function, a #define statement is placed above the main function as the following:

7. 
```
void LED_Sweep_Neg_Control(v

#define Start       01
#define Pause       02
#define Reset       04

int main(void)
{
    io_init();
```

8. Place a conditional statement that runs the LED_Sweep function when the first Sw is pressed. Since we have the #define above the main function, we can use the name of the switches. E.g. if (input_sw & Start), then call the sweep function.
9. Modify the LED_Sweep function to include the following operation:
    a. When the Pause switch is pressed and held down, the LED sweep sequence stops at where it was.
    b. When the Pause switch is released after it was held down, the LED sweep continues
    c. When the reset switch is pressed, all LED turns off. LED sweep do not continue until the Start switch is pressed.
    d. The LED_Sweep only once after Start pushbutton is pressed.
10. Use the Simulator to verify its operation. If the simulator seems to be stuck in a loop, press CTRL+F5 to break from the loop. Watch video online on how to simulator this operation.
11. Verified the operations with the simulator. Watch the video in Brightspace under Labs → Lab 1&2 Bit Operations Simulator and debugger → Lab 1 Videos → select the second video.
12. Depends on your circuit's hardware wiring. If it is negative logic, we need to change the code to expect negative logic inputs. This is very simple, place a ~ in front of the every PINx in your code. E.g. from previous code: Input_sw = PINA & ((1<<PA0) | (1<<PA1) | (1<<PA2)), we will change it to this: Input_sw = **~PINA** & ((1<<PA0) | (1<<PA1) | (1<<PA2)). This will make the code to expect negative logic inputs. The rest of the code will remain the same. Please note: it is ok if the sweep sequence only run through one time, then the start switch need to be pressed again to run the sequence again. Watch video on its operations.
13. Download the error and warning free code to the controller.
14. Verify the operation before proceeding to check off.

Instructor Check off
1. Hardware circuit operates properly 2. Demonstrate use of break point and steps in simulator 3. Schematic of a input circuit, explain its operation