

Contents

Atmel Lab Startup Instructions.....	4
Logistical Challenges and Solutions.....	4
Project Management.....	4
Part 1 – Creating a Project.....	5
Part 2 – Update Project Settings	7
Part 3 – Connecting and Running Software	9
Atmel Studio Debugger Tutorial.....	11
Debugger Overview	11
Debugger Tutorial Exercises	12
Reference Material.....	17
Submitting Atmel Projects.....	17
Adding an ASF Module	18
Digilent Analog Discovery 2.....	19
Lab A1 – Flashlight.....	20
Objective	20
Part 1 – Add Breadboard LED	20
Part 2 - External Push Button (SW1) Circuit	23
Part 3 – System Analysis.....	26
Part 4 – Can you turn this into a flashlight?	27
Part 5 – Design and Implementation.....	28
Implement check_gpio_input_state	30
Implement main.c	31
Final Screenshots.....	32
Submitting the Lab	33
Lab A2 – Stopwatch State Machine.....	34
Part 1: State Machine Specification and Design.....	34
Part 2: Implementing the State Machine	35
Logistics:	35
State Machine Code Concepts	35
State Machine Code Implementation	36
Atmel LCD Usage	36

Part 3: Displaying Stopwatch Time Values on LCD.....	37
Part 4: Displaying Clock Time Values on LCD	37
Part 5: Implementing Time.....	38
Part 6: Final Integration.....	39
Submitting the Lab	39
Lab A3 – Event-Driven System.....	40
System Specification.....	40
Additional Requirements.....	40
Part 1 - Design	41
Part 2 – Unit Testing.....	42
Step 1 – System Time	42
Step 2 – LCD Backlight	42
Step 3 – Set up the GPIO interrupt for the light sensor	43
Step 4 – Change the LCD backlight based on the light sensor	43
Step 5 - Set up the EIC interrupt for SW0.....	44
Step 6 - Change the wireless icon based on the SW0 interrupt.....	44
Step 7 - Set up the Timer Counter to interrupt at a certain frequency.....	44
Step 8 - Turn the breadboard LED on/off using the timer counter	45
Step 9 - Measure the breadboard LED pin frequency using Digilent	45
Part 3 – System Integration.....	46
Step 1 – Integrate System Time and Light Sensor Capability.....	46
Step 2 – Integrate EIC Interrupt Capability.....	46
Step 3 – Integrate Timer Counter Capability.....	46
Step 4 – Configure Interrupt Priorities	47
Step 5 – Implement and utilize mdelay.....	47
Step 6 - Verify that the entire system works properly and meets the specification.....	47
Submitting the Lab	48
Lab A4 - Digital Thermometer	49
System Specification.....	49
Additional Requirements.....	49
Part 1 – Design.....	50
Part 2 – Implementation	51
Print to Putty	51

Submitting the Lab	54
Lab A5 – FreeRTOS	56

Atmel Lab Startup Instructions

Logistical Challenges and Solutions

For Embedded labs, students are allowed to use the lab computers and/or their personal laptops. Most students will use lab computers during class, though you are free to bring your laptop to class. Lab computers have the necessary limitation that they do not allow students to permanently store files on the C: drive. So as most of you know, if you do work on the C: drive and log off, that work will likely be lost, even if you log back into the same computer. This is what we want to avoid...lost work. We have these fundamental challenges:

1. We need the files to be local on the computer (or at least close to local) to get reasonable performance.
2. We need to save our files in a place where we won't log off and lose them!
3. We need to be able to work on our labs outside of our normal classroom setting.

To solve these challenges, we will use Box Drive to store your files on Box. If done properly, Atmel Studio should run at a reasonable speed. This will store your files in the cloud on Box and are therefore accessible anywhere that you have a network connection.

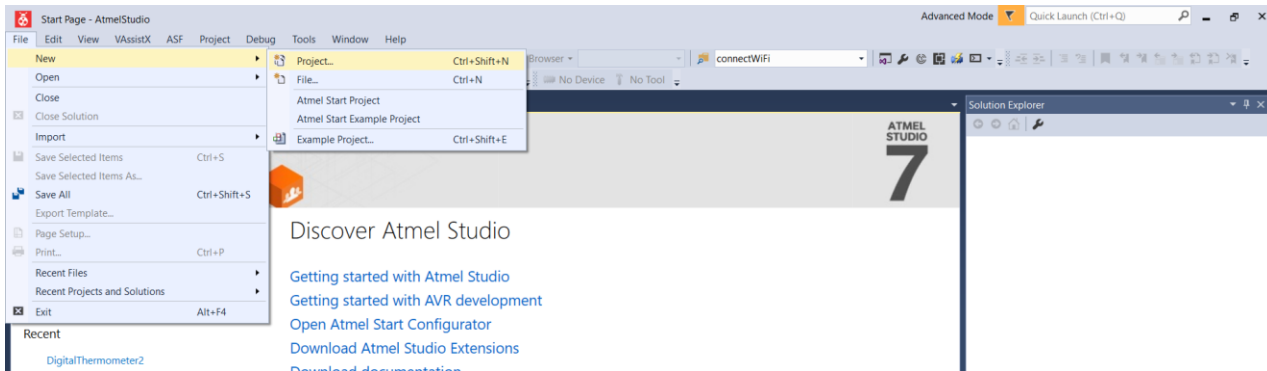
Project Management

The labs in this course will build upon each other, but we are not building up to one final project like we do in Computer Organization (for those who have taken it). Because of this, we want to leverage the work that we have done so that we don't have to recreate the wheel for each lab. However, we want to save a copy of each lab, as each lab is somewhat unique. To do this, please follow the following methodology for all Atmel labs.

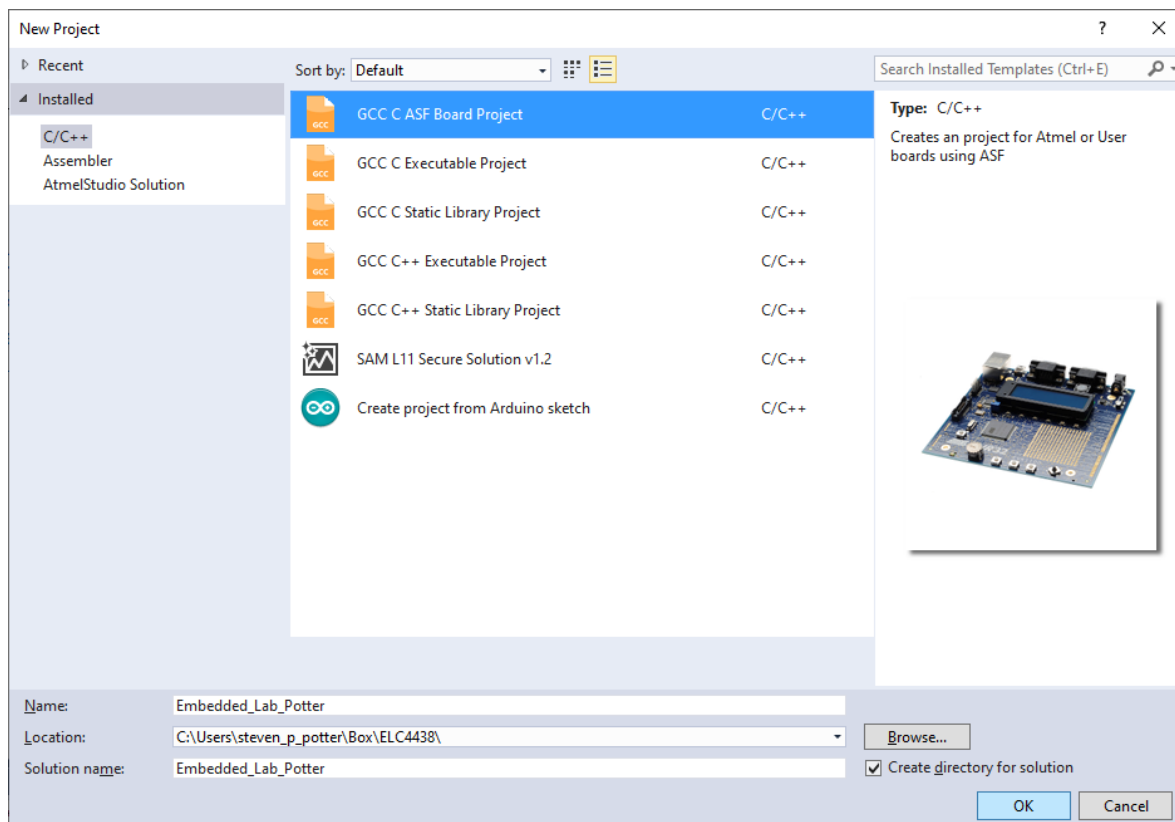
1. Using Windows File Explorer (the file icon on the bottom taskbar that you probably use on a regular basis), create a directory called:
 - a. C:\Users\your_user_name\Box\ELC4438
2. Right-click on ELC4438 and select 'Make Available Offline'.
 - a. This will cause all files in the ELC4438 to be downloaded to the local computer in the background. They are still stored on the cloud, but it will also silently create a local copy on the hard drive, greatly increasing compile speeds.
3. Navigate into ELC4438 and create a directory called LabA1.
4. When you create your project, save your project to this directory and work on this project until you are finished with LabA1.
5. When you have finished LabA1 and are ready to start LabA2, do NOT create a new project for LabA2. Instead, create a directory called:
 - a. C:\Users\your_user_name\Box\ELC4438\LabA2
6. Copy your entire project directory from the LabA1 directory and paste it into the LabA2 directory. Then open the project in the LabA2 directory work on this project until it is complete. This leaves us a clean copy of LabA1 yet allows us to leverage our work from LabA1 for LabA2. Please make sure to follow my project naming convention when you create your project....the project name is generic and does not specify which lab assignment it is.
7. For each Atmel lab, keep using this method. It will be very important to have each lab available to you during application exams.

Part 1 – Creating a Project

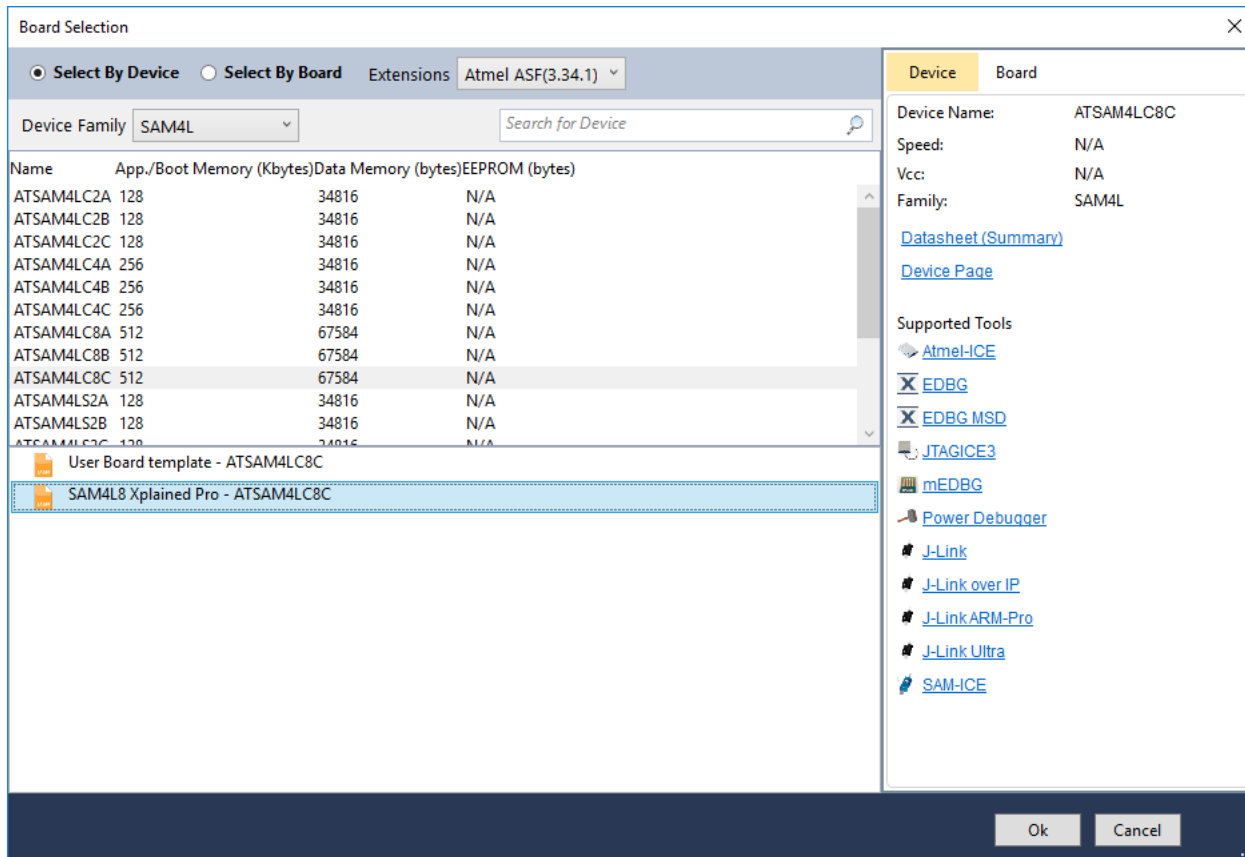
1. Open Microchip Studio
2. Open a new project using File -> New -> Project



3. Click GCC ASF Board Project
4. The Name should be 'Embedded_Lab_Lastname' (where you use your lastname, not the word Lastname).
The project name should be generic, not specifying Lab1. Please trust me and follow this naming convention.
5. The Location should be C:\Users\your_user_name\Box\ELC4438\LabA1.



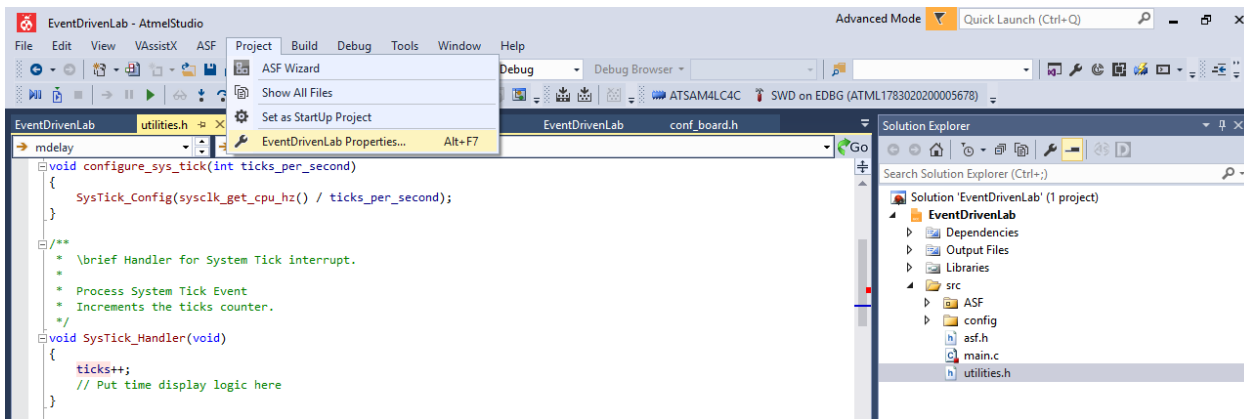
6. After it creates the project (can be several minutes), use the Board Selection dialog to choose the Atmel SAM4L XPlained Pro board
 - a. Under Device Family, select 'SAM4L'
 - b. Then select 'ATSAM4LC8C'
 - c. Then select 'SAM4L8 XPlained Pro – ATSAM4LC8C'
 - d. Then click OK



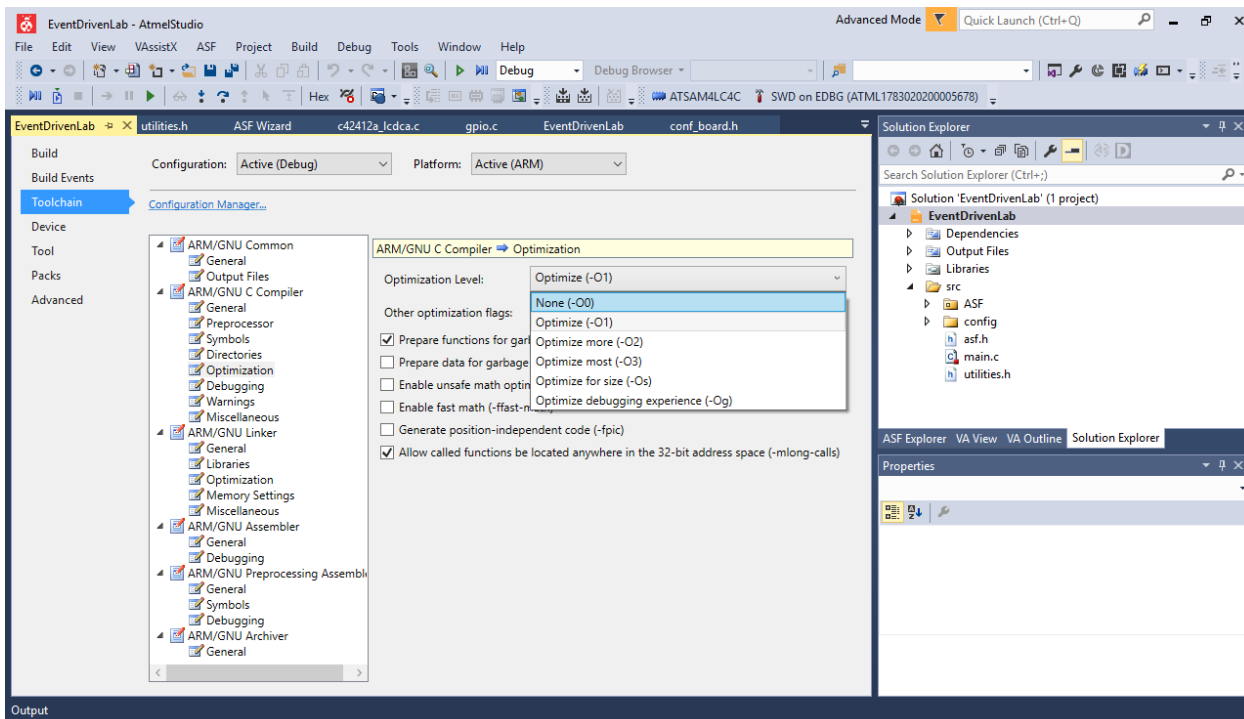
Part 2 – Update Project Settings

Several project settings must be updated for the project to work as expected. The intent is that you create one project and keep using a copy of this project for all of the labs. However, sometimes students need to create a new project. **Please remember that if you do create a new project, you must do these steps on the new project. It will cause you problems and waste your time if you do not.**

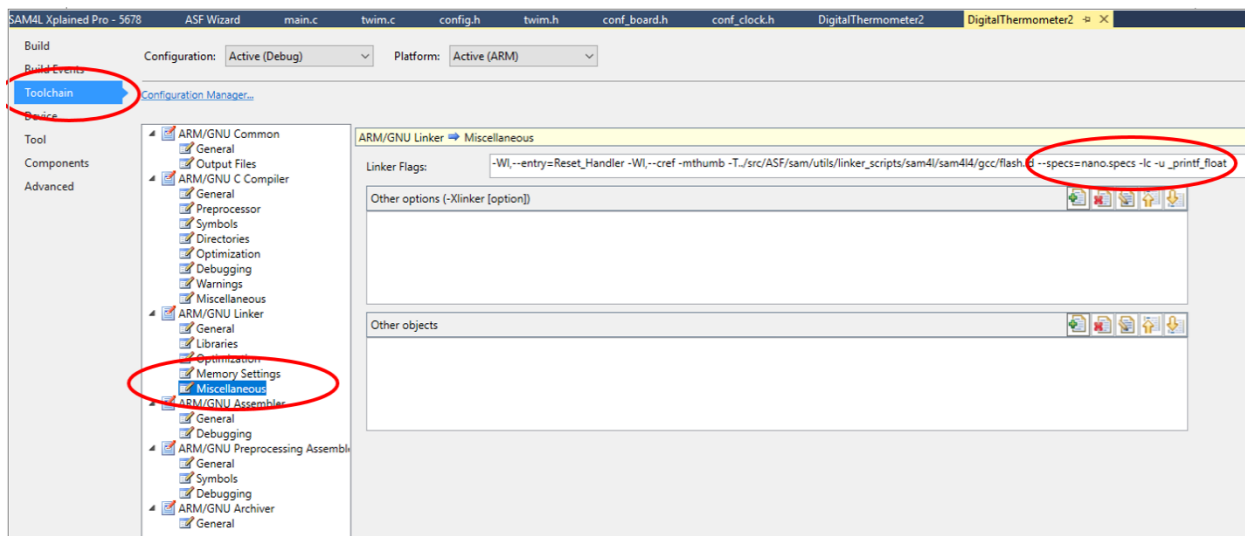
1. Next, we need to turn off Compiler Optimization. This will allow our code to execute exactly as we write it as well as allow the debugger to work properly. Please follow these steps:
 - a. Go to the properties menu for your project as shown below



- b. Select Toolchain on the left side, then select Optimization under the ARM/GNU C Compiler section, then change the Optimization Level to None (-O0). Then click 'Save'.

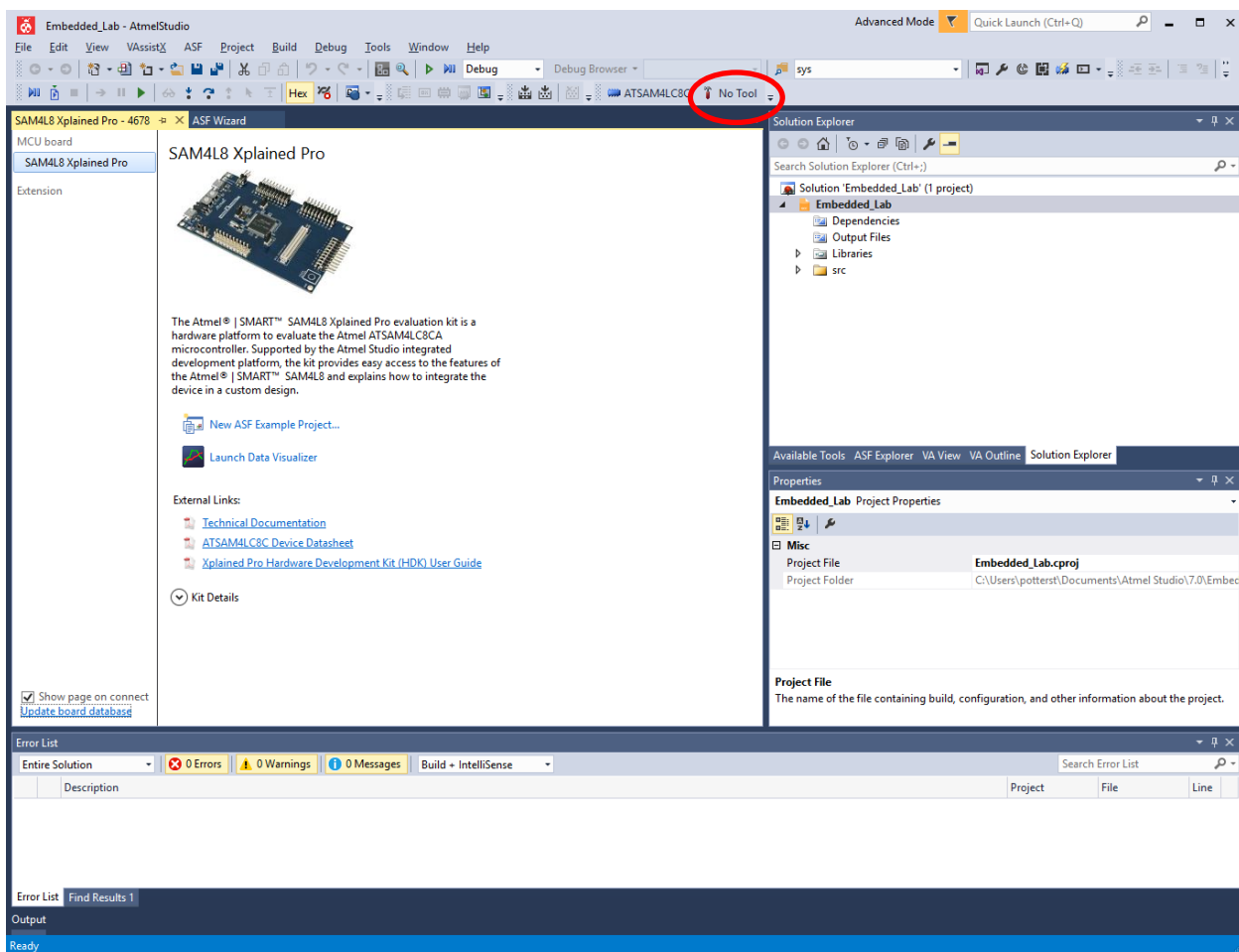


2. Atmel Studio has a bug when printing out floats, which is something we will do later in the semester. Let's go ahead and apply the fix right now since we will keep the same project going for each lab.
- To fix this, you will need to add the following line to the ARM/GNU Linker -> Miscellaneous line in the Toolchain properties (see pic below). Note that you might need to type this, as sometimes Microsoft adds strange characters when you copy and paste from a Microsoft product. Also, note that the character where it says -lc is a lower case L, not an I. Finally, note that there should be a space between the current end of the line and the new text that you are adding. After adding click 'Save'.
 - `--specs=nano.specs -lc -u _printf_float`

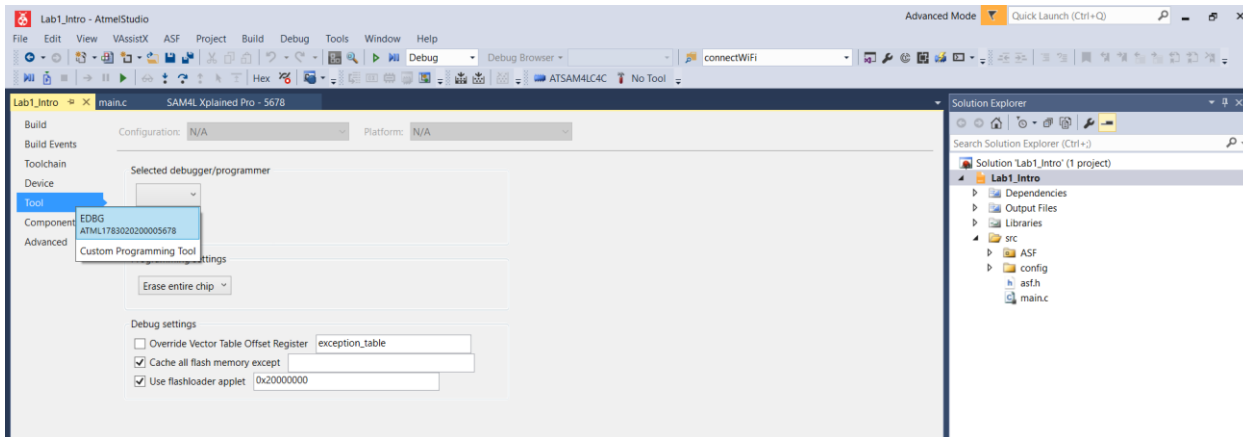


Part 3 – Connecting and Running Software

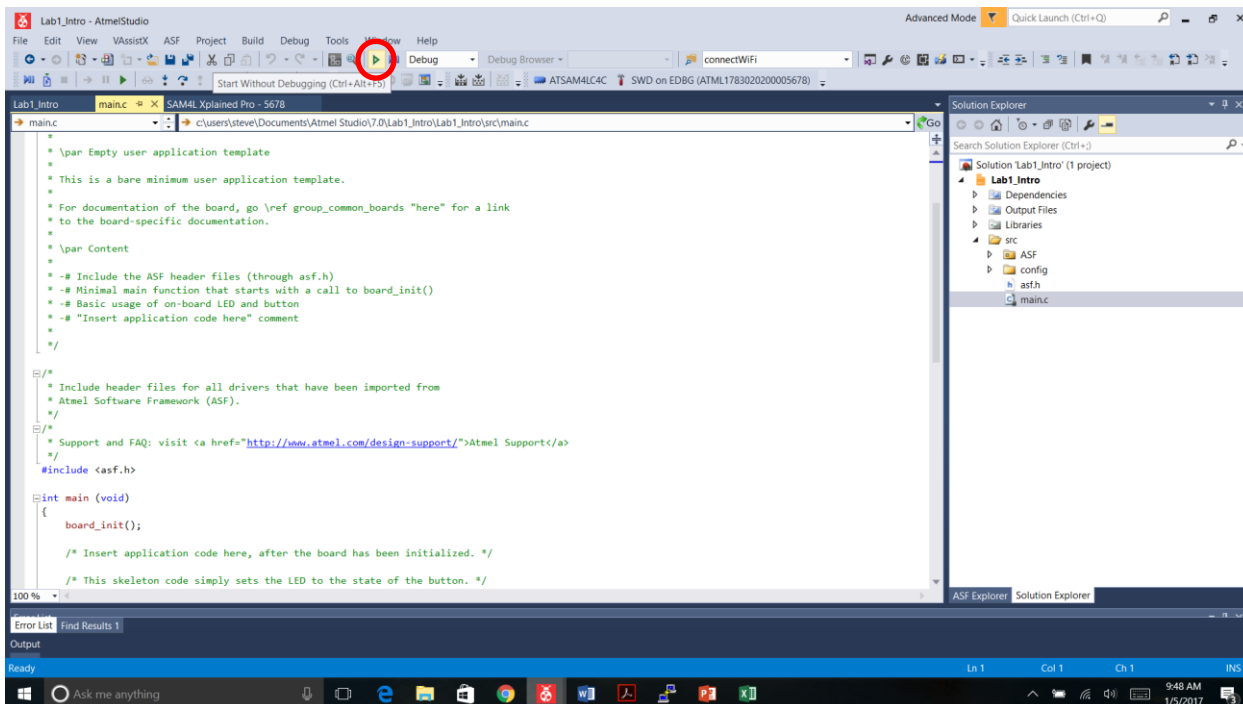
1. Go to the Solution Explorer (on the right hand side):
 - a. Expand 'src'
 - b. Double-click on main.c to open it for editing
 - c. Examine this code to determine what it does
2. Connect your SAM4L8 Xplained Pro by doing the following:
 - a. Plug the USB cord into your PC
 - b. Plug the Micro USB end into the 'DEBUG USB' port on the board (note that there is also a Micro USB port labeled SAM4L8 USB, which will not interact with Atmel Studio)
 - c. When connected properly, you should see the following screen open on Atmel Studio (it might be in a different tab on Atmel Studio):



- Once connected, click on 'No Tool' (circled in red above, some versions of Atmel Studio say 'None on' instead...same thing), and select the EDBG debugger/programmer and save your project using the Save Icon.



- Run the program without debugging by clicking on the unfilled green triangle (see figure below). This will build the software and program the microcontroller with the code in main.c. Therefore you do not need to do a separate compile step each time you want to run.



- On the SAM4L8 XPlained Pro board, press the button labeled SW0. LED0 should light up while SW0 is pressed.

Atmel Studio Debugger Tutorial

Debugger Overview

The debugger is an extremely useful tool in all software projects, but especially in embedded systems. Often in embedded systems, you do not even have the ability to print messages to a console, so you need another way to determine what is causing unexpected behavior in your program. Thankfully for us, there is an embedded debugger chip (EDBG) on the Atmel board that can be used in conjunction with Atmel Studio to run a debugger to step through the code.

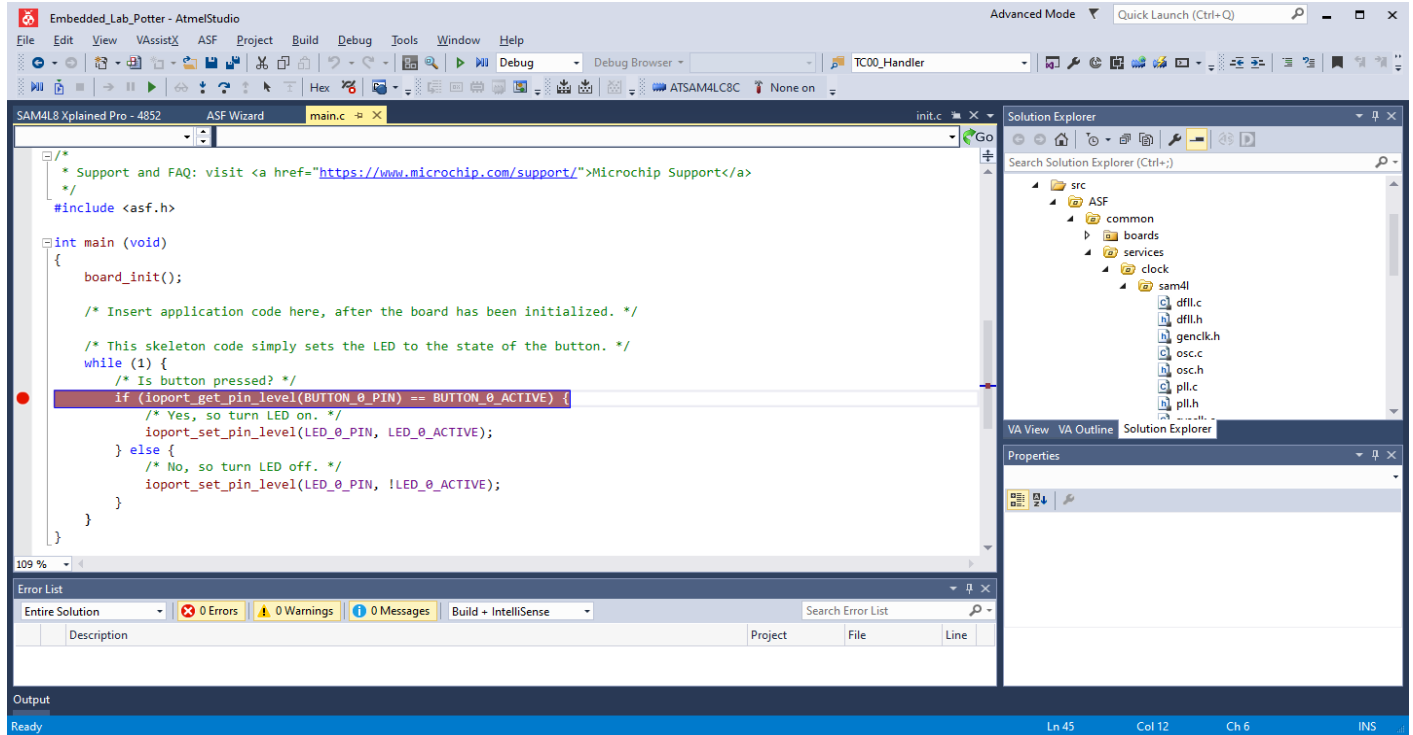
1. Often, you want to set a breakpoint in the code and run to that breakpoint and pause. The easiest way to set a breakpoint is to go to the code and click on the gray vertical bar on the far left side of the code. This will insert a breakpoint at that line. Below is a description of each of the Debugger icons. You can see the names of each icon by hovering the mouse over the icon.
 - Start Debugging – this is the filled green arrow, and this is what you use to start the debugger. Once you have pressed this button and the debugger is started, this button will change to 'Continue'.
 - Continue – this is the filled green arrow, and it is used to continue debugging after the debugger has paused due to hitting a breakpoint or using the 'Break All' button.
 - Break All – this is the 'pause' button that will pause the debugger at whatever line of code it is executing. This is extremely helpful when the code is not responding at all like you think it should. Sometimes you will pause the code and find it in 'Dummy Handler', meaning there was an exception. Sometimes you will find that it is stuck in an infinite loop, etc.
 - Stop Debugging – this will stop the debugger and end the debugging session.
 - Step – these buttons will step to the next line of code. Please note that when a line of code is highlighted, it has NOT been executed yet. It will be executed when you step or continue.
 - Step Into – if the highlighted line of code is a function call, this button will step into the function and allow you to step through each line of code in the function. When you reach the end of the function, it will return to the line after the function call. If the highlighted line of code is not a function, it will just step to the next line of code.
 - Step Over - if the highlighted line of code is a function call, this button will step over the function and move to the next line of code. The function is still executed, but you just don't step through it. This is especially handy for ASF functions, printf, etc where you did not write the code. If the highlighted line of code is not a function, it will just step to the next line of code.
 - Step Out – if you are debugging inside of a function, you can press this button to step out of the function and go back to the function where the code was called from.
2. When the debugger is paused (whether at a breakpoint or just paused from using the 'Break All' button), the bottom-left pane will show variable values. There are 4 tabs for variables:
 - a. Autos – these are variables chosen by the debugger that may or may not be helpful.
 - b. Locals – these are the local variables in the function where you are paused.
 - c. Watch 1 and Watch 2 – these are variables that you choose that you specifically want to watch. You can go to any variable and right-click -> Add Watch. This includes going to the code, hovering over a variable, and right-clicking on the variable in the code.
3. In the bottom-right pane, there are several tabs. The one that I look at is the Call Stack tab. This shows where you currently are in the code as well as how you got there (where were functions called from). Often, you know this information already. However, if your code is not responding to you and you have no idea

what is happening, you can run the debugger, use the 'Break All' button to pause, then look at the Call Stack. This often helps you figure out where things went wrong.

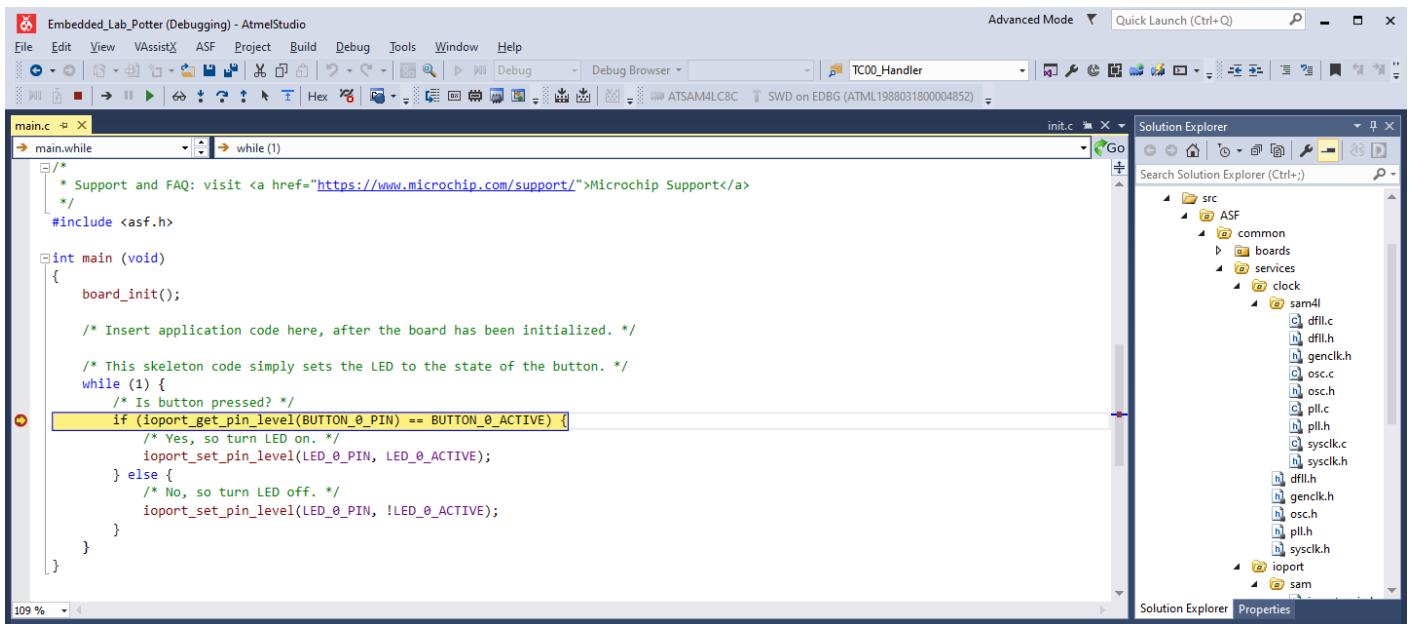
Debugger Tutorial Exercises

Now we will go through some hands-on exercises to show us how to use the debugger. Please open your project that you created above and follow through these steps:

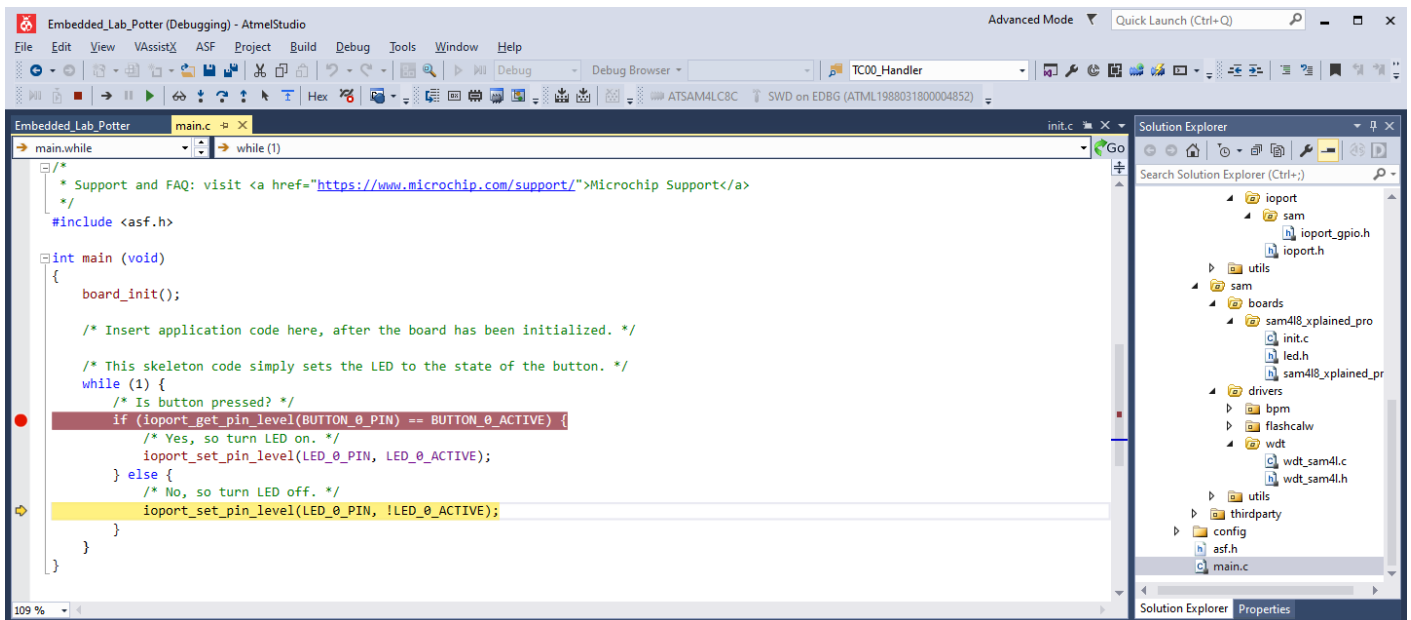
1. Set a breakpoint on Line 45 as shown below. Please note that the line number can be found in the lower right corner of Atmel Studio.



2. Click the 'Start Debugging' icon (filled green arrow). After compiling, the code will run until it reaches the breakpoint, and it will stop at the breakpoint. Please note that the line with the breakpoint has NOT been executed yet. It is about to be executed next.

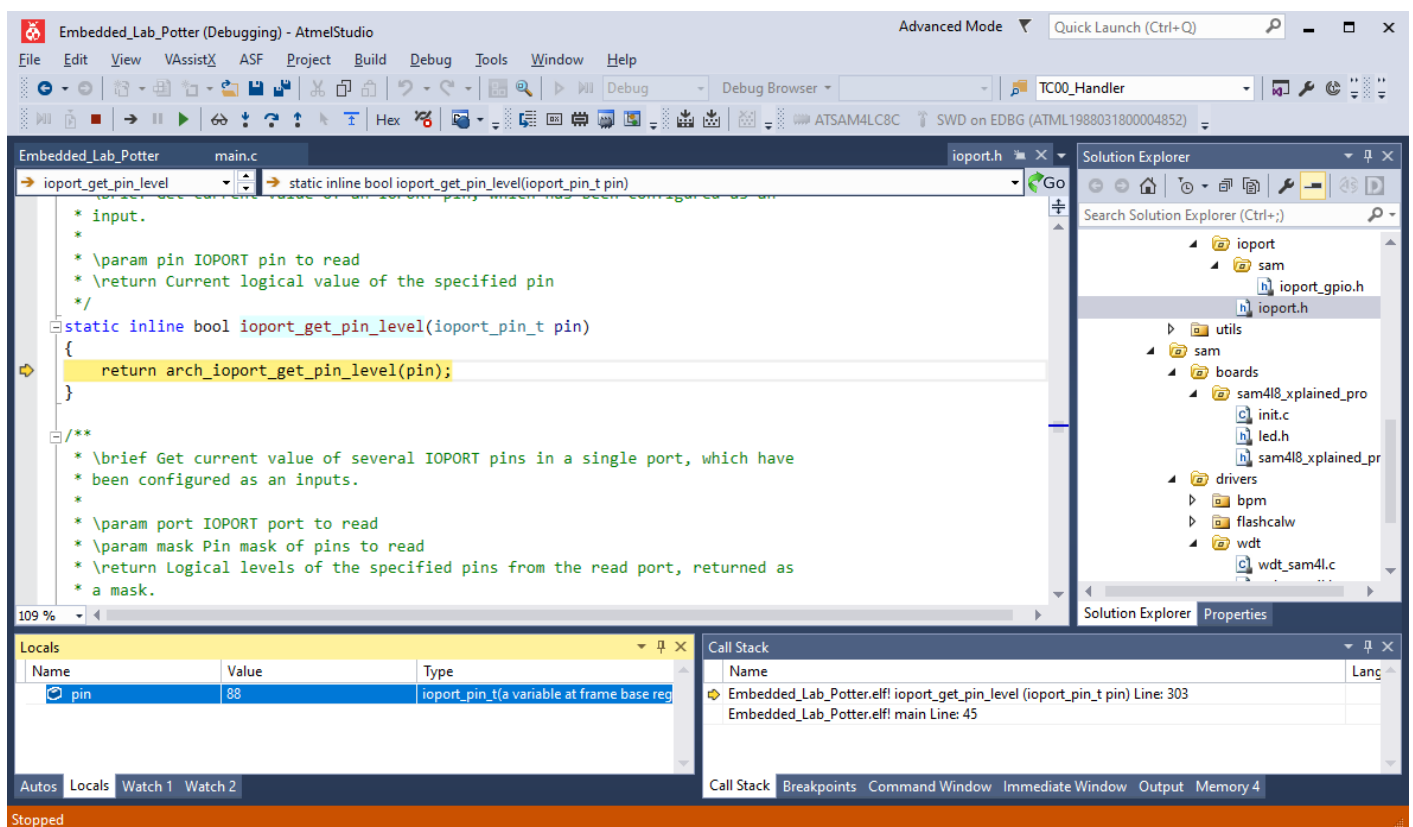


- Now press the 'Step Over' button. Since the SW0 button is not currently pressed, the 'if' statement is not true and it will go into the 'else' statement, as shown below.



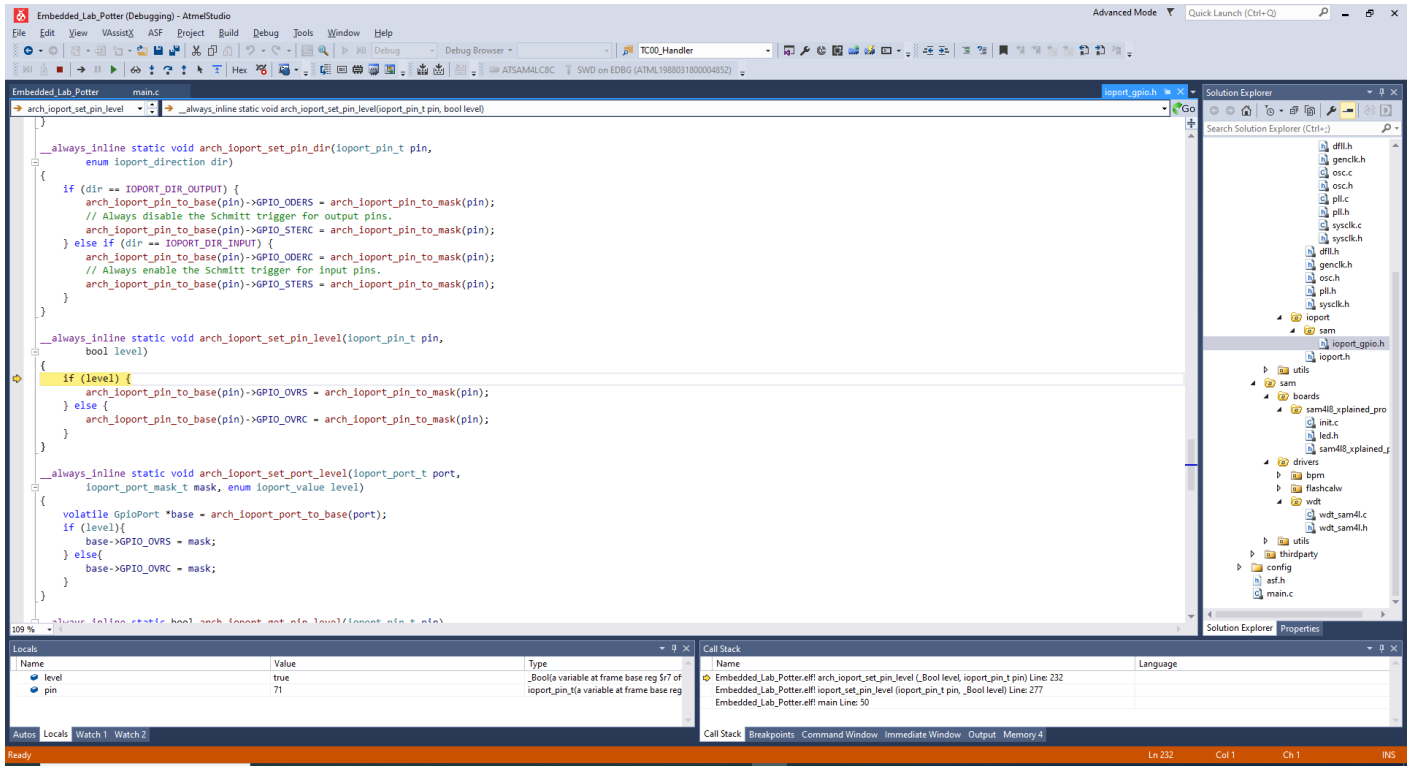
- Now press the 'Continue' button (filled green arrow). The debugger will finish the loop, go back to the top of the loop, and stop at the breakpoint again. After it stops at the breakpoint, hold down SW0 and press 'Step Over'. You will now see it go into the 'if' statement. Now press 'Step Over' again. The LED should now turn on because the ioport_set_pin_level function has now executed. Do this a few more times where sometimes you press the button and sometimes you don't press the button. Note that the only time the button has an impact on operation is when the 'if' statement executes. The rest of the time, it does not matter whether you press or release the button.

5. Now get back to the breakpoint and press 'Step Into'. This takes you into the `ioport_get_pin_level` function code. In the screenshot below, you can see that:
 - a. The `ioport.h` file has been opened and that the `ioport_get_pin_level` function is about to call 'return `arch_ioport_get_pin_level(pin);`'.
 - b. At the bottom left of the screen, you can select 'Locals' to see local variables and their values. This screenshot shows the 'pin' has a value of 88 and is of type 'ioport_pin_t'.
 - c. At the bottom left of the screen, you can select 'Call Stack' and see that the top of the Call Stack (currently line of code) is Line 303 of the currently open file (`ioport.h`). More importantly, you can see that this line of code was called from `main.c` Line 45. If you double-click on a line in the Call Stack, it will open that file and take you to that line of code. The Call Stack can be extremely useful when we start having many nested functions and are trying to figure out how we got to the currently running line of code.



6. At this point, you realize that you do not want to see the internal details of the `ioport_get_pin_level` function. Press 'Step Out' to get out of that function and go up a level in the code.
7. At this point, you should be back to the breakpoint again. Note that the `ioport_get_pin_level` function did not run yet...we stepped out of it rather than stepping through the whole function (or stepping over it, which executes the function without going line-by-line).
8. Now remove the breakpoint and press 'Continue'. The program will run without stopping, and you should be able to press SW0 and see LED0 illuminate. While running, press the 'Break All' button. This will stop the program immediately at whatever line of code is currently executing. I have a screenshot below of where mine stopped, but yours will likely stop in a different place. You can trace back how I got here by looking at the Call Stack. Why is this important? There will be times that your code is not working and you will have no

idea why it is not working or where it is currently executing. This method will show you where your code is executing, and it often surprises us, especially if there has been a fault and it is in a fault handler or something like that.



9. At this point, you can keep messing around with the debugger, or you can press 'Stop Debugging' to exit the debugger. Please note that you have to stop the debugger before compiling and executing code changes.
10. The debugger is great for a lot of purposes, but you cannot debug unless your code compiles. When you have a compile error, it will ask you if you want to continue and run a previous version of the code...say No. Then you will see something like the screenshot below. The first error is simply telling me that compilation failed. The second error is telling me that the I forgot a semicolon on Line 51 of main.c. To go fix it, I double-click on the second error and it takes me directly to that line of code.
 - a. Side note: If you get a strange error that is pointing to the end of the file, it usually means that you forgot a curly brace or have an extra curly brace somewhere. Please note that 'somewhere' includes any file that is included. This compiler inserts the contents of the include files at the top of the code and then compiles it all as a single file. So a missing curly brace in a header file can show up as an error in main.c.

Embedded_Lab_Potter - AtmelStudio

Advanced Mode Quick Launch (Ctrl+Q)

File Edit View VAssistX ASF Project Build Debug Tools Window Help

Debug Debug Browser TC00_Handler

ATSAM4LC8C None on

SAM4L8 Xplained Pro - 4852 ASF Wizard Embedded_Lab_Potter main.c x ioport_gpio.h x

ioport_set_pin_level static inline void ioport_set_pin_level(ioport_pin_t pin, bool level){...}

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

    /* Insert application code here, after the board has been initialized. */

    /* This skeleton code simply sets the LED to the state of the button. */
    while (1) {
        /* Is button pressed? */
        if (ioport_get_pin_level(BUTTON_0_PIN) == BUTTON_0_ACTIVE) {
            /* Yes, so turn LED on. */
            ioport_set_pin_level(LED_0_PIN, LED_0_ACTIVE);
        } else {
            /* No, so turn LED off. */
            ioport_set_pin_level(LED_0_PIN, !LED_0_ACTIVE)
        }
    }
}
```

109 %

Error List

Entire Solution 2 Errors 0 Warnings 0 Messages Build + IntelliSense Search Error List

Description	Project	File	Line
recipe for target 'src/main.o' failed	Embedded_Lab_P otter	Makefile	276
expected ';' before '}' token	Embedded_Lab_P otter	main.c	51

Output

Build failed

Solution Explorer

Search Solution Explorer (Ctrl+;)

- ioport
 - sam
 - ioport_gpio.h
 - ioport.h
- utils
- sam
 - boards
 - sam4l8_xplained_pro
 - init.c
 - led.h
 - sam4l8_xplained_pro.h
- drivers
 - bpm
 - flashcalw

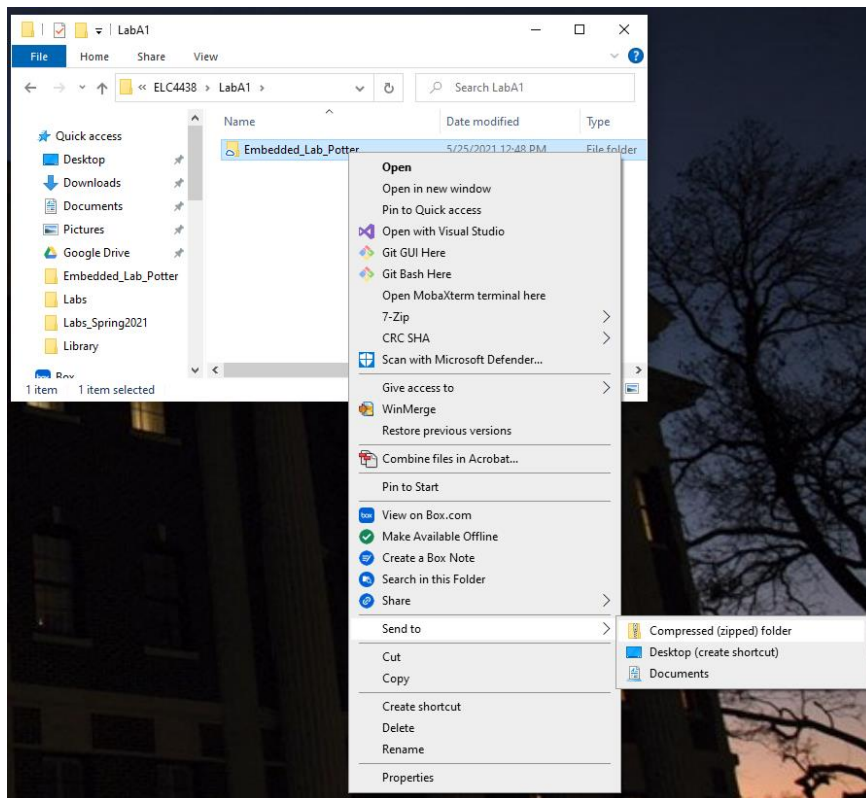
VA View VA Outline Solution Explorer

Properties

Reference Material

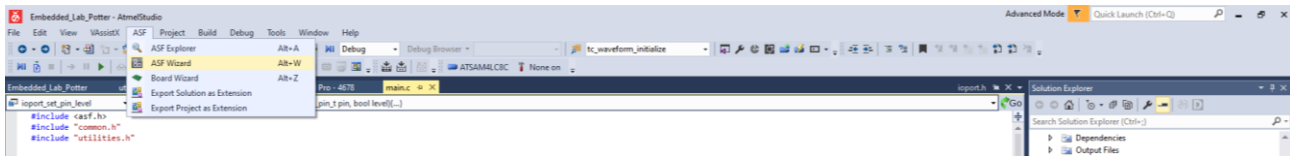
Submitting Atmel Projects

1. Before submitting your project, please open it in Atmel Studio and select 'Build -> Clean Solution'.
 - a. This will remove all of the binary files but leave your code. The binary files are much larger than the code files. This will make the project a fraction of the size that it would be otherwise.
2. Zip up the entire directory that contains your project by navigating to C:\users\your_user_name\Box\ELC4438\LabA1 and zipping up the entire Embedded_Lab_Lastname directory so that I can unzip, execute, and test. See screenshot below.
3. If there are screenshots or other files to submit, please submit those separately from the Atmel project zip file.

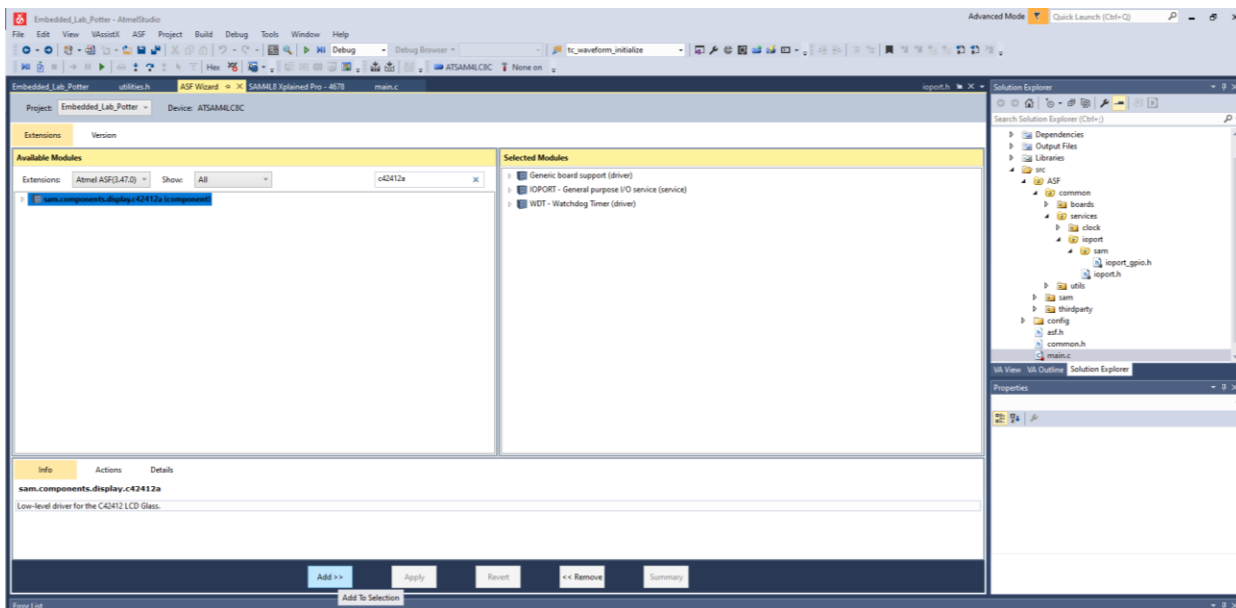


Adding an ASF Module

1. In Atmel Studio, verify that you are in 'Advanced Mode', as displayed near the top-right corner of the screen. Then Select 'ASF Wizard' from the 'ASF' drop-down menu. It might take up to a minute for it to open.



2. In the 'Available Modules' pane on the left, search for 'c42412a' to find the correct module to add. Note that you should have Atmel Studio maximized on your screen so that you can see the search box. Once you select the module, click 'Add'. This will put the module in the 'Selected Modules' pane on the right side, but it does not add the module. To add the module, you must click 'Apply'. Once you click 'Apply', you can use the module in your code because it has been added to the ASF.



[Digilent Analog Discovery 2](#)

The [Digilent Analog Discovery 2](#) is a small device that is packed with functionality. You access this functionality by connecting wires to your circuits and using the [Waveforms](#) program on the computer. During this semester, we will be using:

- Oscilloscope
- Wave Generator
- Power Supply
- Voltmeter
- Logic Analyzer
- Protocol Analyzer

Lab A1 – Flashlight

Objective

The end goal of this lab is to create a flashlight with 2 push buttons and 2 LEDs. The button on the Atmel board will control the LED on the Atmel board. The button on the breadboard will control the LED on the breadboard. We will take a methodical approach. The process is as important as the final product in this lab. Please follow the steps closely, utilizing all of the methods, file structure, and macros that I specify in the lab manual.

Part 1 – Add Breadboard LED

When you created your project, there was sample code that turns the LED on while the button is pressed and turns the LED off while the button is not pressed. In this section, we will add a second LED to the hardware and software. Follow the steps below.

1. Replace the original code with the code below and click the unfilled green arrow to compile and run this new code. This code will work the same as the original code, except it adds a second output pin, EXT1_PIN_6. EXT1_PIN_6 is configured as an output pin and initialized to LED_0_INACTIVE before the while(1) loop. In the while(1) loop, it is set to the same logical value that LED_0_PIN is set to.

```
#include <asf.h>
// #include "common.h"
// #include "utilities.h"

#define ONE_BUTTON_TWO_LED
// #define TWO_BUTTON_TWO_LED
// #define DOUBLE_FLASHLIGHT

int main (void)
{
    board_init();
#ifdef ONE_BUTTON_TWO_LED
    // configure BREADBOARD_LED_PIN
    ioport_set_pin_dir(EXT1_PIN_6, IOPORT_DIR_OUTPUT);
    ioport_set_pin_level(EXT1_PIN_6, LED_0_INACTIVE);

    while (1)
    {
        // Is button pressed?
        if (ioport_get_pin_level(BUTTON_0_PIN) == BUTTON_0_ACTIVE)
        {
            // Yes, so turn LED on
            ioport_set_pin_level(LED_0_PIN, LED_0_ACTIVE);

            // Yes, so set EXT1_PIN_6 to same value as LED
            ioport_set_pin_level(EXT1_PIN_6, LED_0_ACTIVE);
        }
        else
        {
            // No, so turn LED off
            ioport_set_pin_level(LED_0_PIN, !LED_0_ACTIVE);

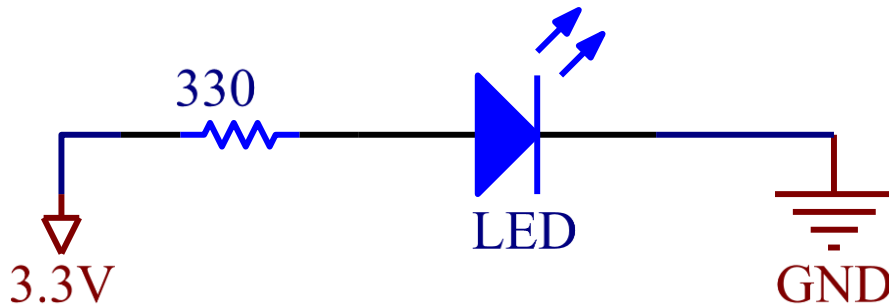
            // Yes, so set EXT1_PIN_6 to same value as LED
        }
    }
}
```

```

        ioport_set_pin_level(EXT1_PIN_6, !LED_0_ACTIVE);
    }
}
#endif

#ifdef TWO_BUTTON_TWO_LED
#endif
#ifdef DOUBLE_FLASHLIGHT
#endif
}

```



2. Build this circuit using the provided components. Note that the anode(+) of the LED has a longer lead than the cathode(-). The 3.3V source should come from the Digilent device.
 - a. Connect the 30 wire connector and use the V+ as 3.3V power and one of the 4 ground wires as GND.
 - b. In Waveforms, use the 'Supplies' option (rather than 'Scope' option you used before).
 - i. Change the V+ voltage to 3.3V
 - ii. Click the Master Enable to turn the power supply on.
3. Use a multimeter to measure the voltage across the LED. You should get a voltage around 2V.
4. Use a multimeter to measure the voltage across the 330 ohm resistor. You should get a voltage around 1.3V
5. Use a multimeter to measure the current in the circuit.
 - c. In order to do this, you must break the circuit and put the multimeter in series with the circuit. If you have questions on how to do this, please ask! A lot of people don't remember this from circuits, so don't be embarrassed to ask me or a neighbor.
 - d. You must have the black multimeter lead in the COM port
 - e. You must have the red multimeter lead in a port designated for measuring current
 - f. You must have the dial turned to DC Amps. Different multimeters might have different settings. We expect to measure about 4-5mA.
 - g. If you are using a school-owned Fluke Multimeter, you must press the orange button to change the reading from AC to DC.
6. NOTE: A green LED has a voltage drop of roughly 2V, so the current should be around 3-5mA
 - h. Different LEDs have different voltage drops, so these results do not apply for all LEDs
 - i. 330 ohms is the minimum resistance value that we should use to ensure that we do not pull too much current. While you can see the illumination, it is not extremely bright.
7. If your voltages and currents match the expected values (approximately), then we can connect this LED to your Atmel board. To do this:
 - j. Completely disconnect the Digilent device from your breadboard.

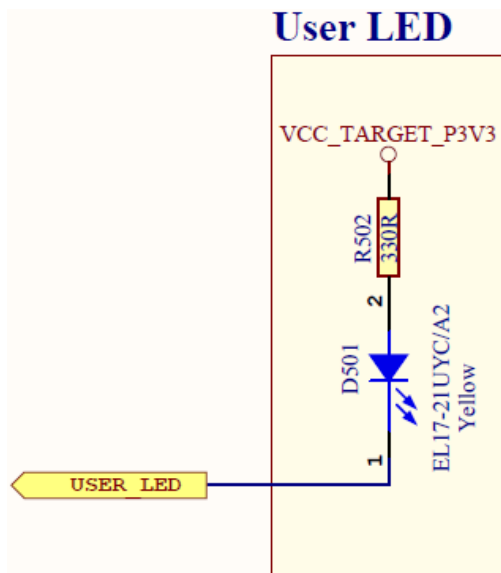
- k. Connect your circuit GND to a GND pin on the Atmel board. I recommend using the GND pin on the power header. The power header is a 4-pin header that is near the USB port. The pins on this header are:

Table 3-6. Xplained Pro Power Header

Pin number	Pin name	Description
1	VEXT_P5V0	External 5V input
2	GND	Ground
3	VCC_P5V0	Unregulated 5V (output, derived from one of the input sources)
4	VCC_P3V3	Regulated 3.3V (output, used as main power supply for the kit)

- l. Now we will replace the 3.3V source from the schematic with pin EXT1_PIN_6. This will allow us to control the state of the LED with software.
8. Run your existing software on the board. You should notice that this LED illuminates only when the LED on the Atmel board is off, and it turns off when the LED on the Atmel board turns on. This is expected. See explanation below.

In the Library section of Canvas, open the SAM4L8 Xplained Pro Schematic. Go to page 4 and examine the User LED schematic (shown below). Compare that schematic with the schematic that we built on our breadboard. The yellow USER LED label represents the output pin that is connected to the circuit. Notice how they connect the LED circuit to VCC and an output pin, whereas we connected to an output pin and GND. In their circuit, if the output pin is 3.3V, then no current flows because there is no potential difference. If the output pin is 0V, then current flows from the 3.3V source, through the resistor and LED and to the output pin.



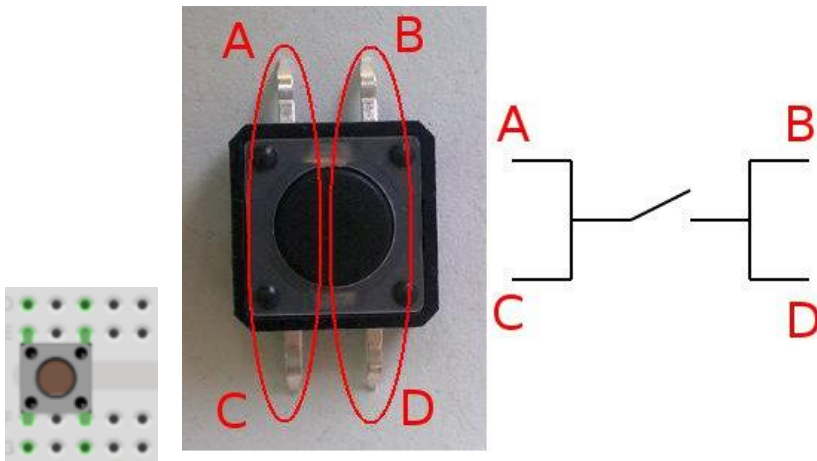
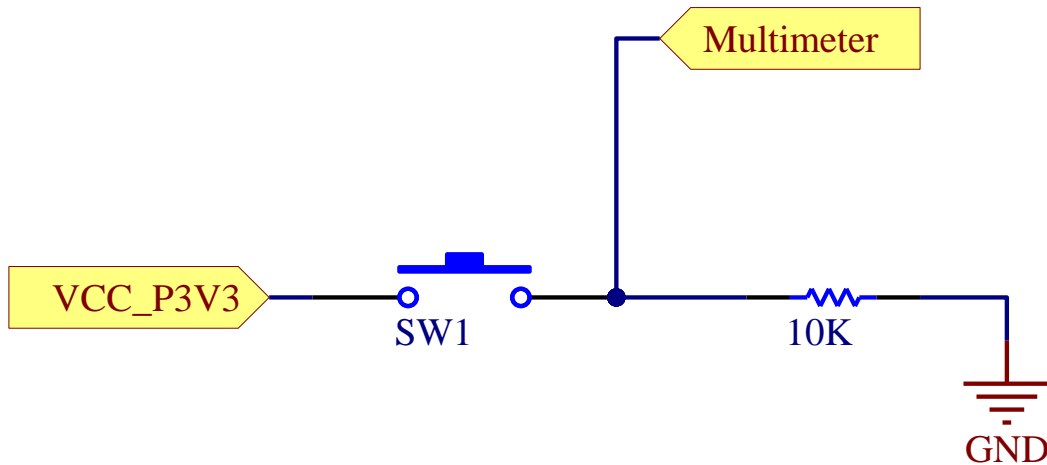
9. Create a new file called common.h by doing the following:
 - a. In the Solution Explorer window on the right, right-click on 'src', then select 'Add' -> 'New Item'
 - b. Click on 'Include File' and name the file (at the bottom) common.h, then click 'Add'
10. Using the same method, create another file called utilities.h

11. Open common.h and remove all contents. Go to Canvas and find the provided common.h file in the 'Labs' section. Open the file, copy the contents of the file, then paste them into your common.h.
 - a. Repeat this step for utilities.h.
12. Make sure to include common.h and utilities.h in main.c.
13. Examine the macros and enumerations contained in common.h. You are required to use these in your software. For now:
 - c. In main.c, replace all instances of EXT1_PIN_6 with BREADBOARD_LED_PIN
 - d. In main.c, when calling ioport_set_pin_level on BREADBOARD_LED_PIN, use the macros BREADBOARD_LED_OFF and BREADBOARD_LED_ON rather than LED_0_INACTIVE and LED_0_ACTIVE
14. Run your code with these updates. Note that, if done correctly, the breadboard LED should now mirror the Atmel board LED
 - e. We should understand this from a circuit perspective at this point, but did I have to pull out a Digilent and/or multimeter to figure this out? No!
 - i. In main.c, right-click on LED_0_ACTIVE and select 'Go to Implementation'. You will see that the macro LED_0_ACTIVE just references another macro, LED0_ACTIVE. So keep going with the 'Go to Implementation' until you get to the bottom level where there is a real value. You will discover the LED_0_ACTIVE evaluates to 'false', which evaluates to 0.
 1. Remember that a macro is just another name for a value. The compiler will replace LED_0_ACTIVE with a 0 before it compiles the code. So it would be functionally equivalent to use 0 instead of LED_0_ACTIVE, but it would be far less readable.
 - f. In the Library section of Canvas, open the Atmel SAM4L8 XPlained Pro User Guide. Go to the Table of Contents and find the section about the LED. Note the sentence that says "The LED can be activated by driving the connected I/O line to GND".

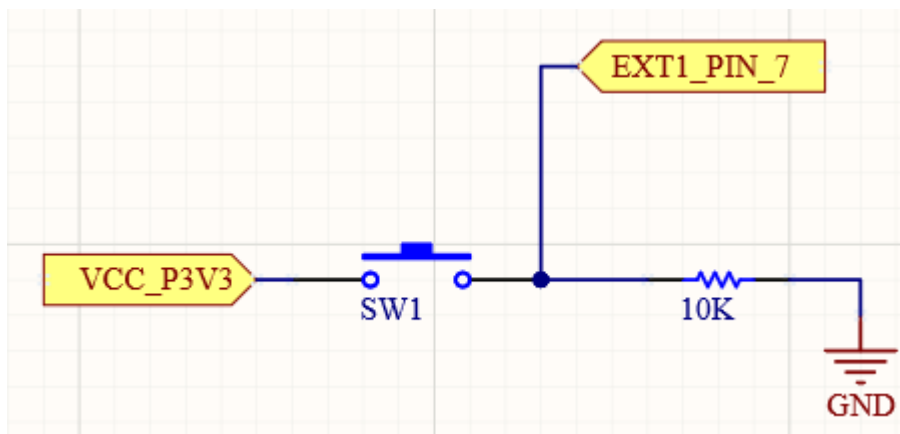
Part 2 - External Push Button (SW1) Circuit

Now we will add a second push button by building a push button circuit on the breadboard and connecting this circuit to EXT1_PIN_7. This circuit will be substantially different from the push button circuit that is implemented on the Atmel board (SW0).

1. Before connecting to the Atmel board, you should build the circuit below and test the voltage and current with a multimeter.
 - a. The push button should be across the bridge on the breadboard. The pinout and diagram below show how that pins A and C (across the bridge from each other) are always connected. B and D are always connected too. To use the push button effectively, you should connect one side of the circuit to A or C and the other side to B or D so that the two sides of the circuit are only connected when the button is pressed.
 - b. Your measured values should be approximately:
 - i. Button not pressed – 0V and 0A
 - ii. Button pressed – 3.3V and 0.33mA (will show up as either 0 mA or 1 mA on multimeter)

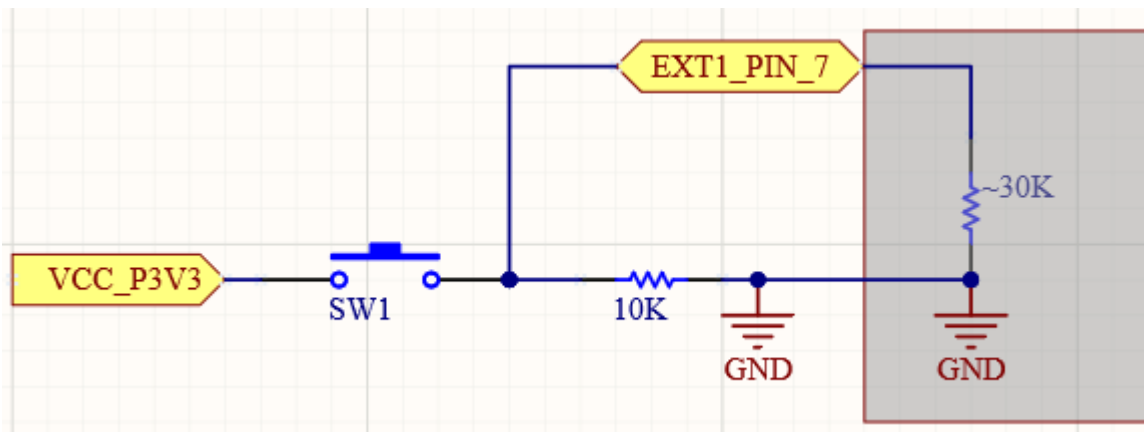


2. Once you have verified that your circuit works correctly, you should connect EXT1_PIN_7 to the circuit as shown below:



If you try to analyze this circuit as it is shown above, it is somewhat incomplete. EXT1_PIN_7 is not just a floating tag on a diagram. It is an input pin into the microcontroller, and it must be configured as either pullup or pulldown in order to know how it will truly behave. We will configure EXT1_PIN_7 as pulldown, which will make the complete

circuit like the one shown below. Note that the components in the gray box are inside of the microcontroller. When we configure the button as pulldown, it connects the input pin to a resistor (can vary, usually around 30k) that connects to GND.

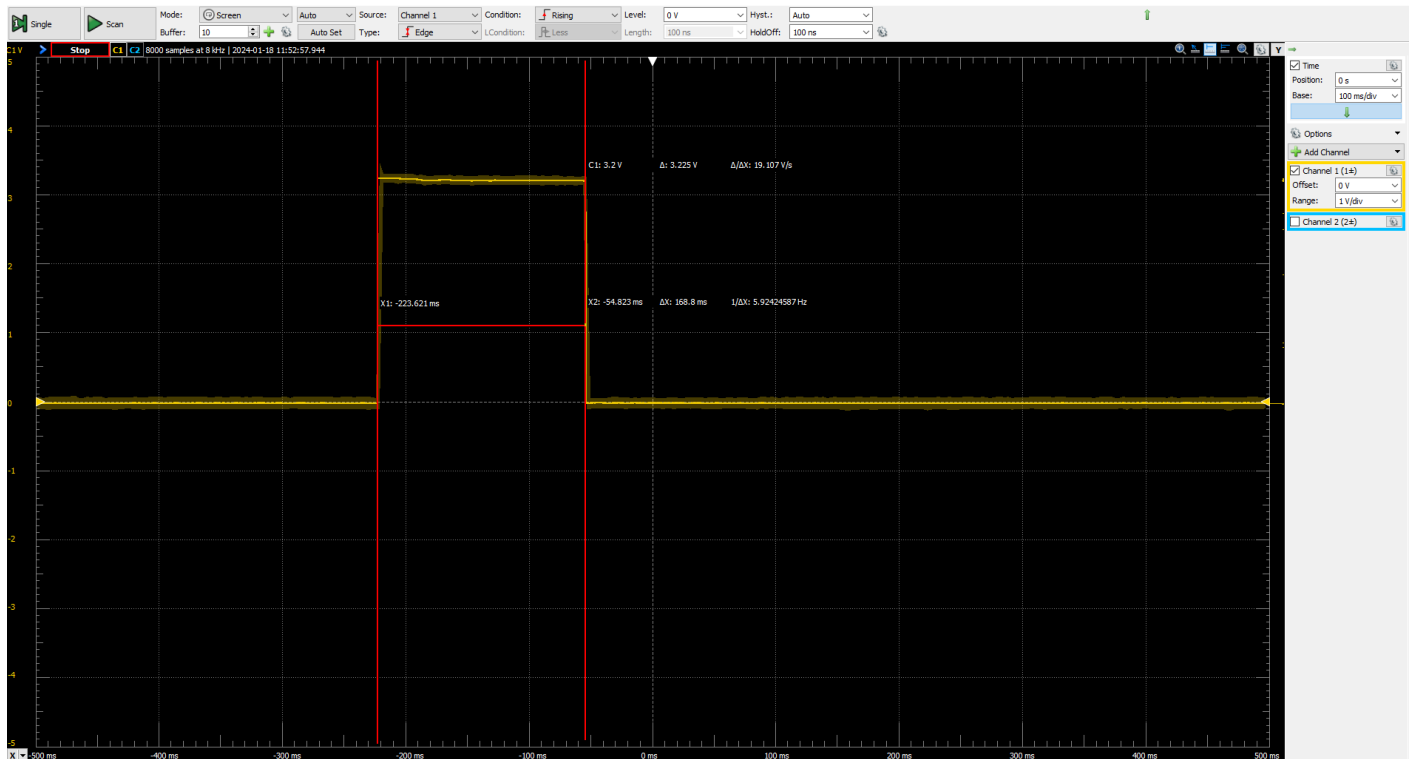


Now we can properly analyze this circuit. GND is common. I just showed both to make a point about parts internal to the microcontroller. This circuit should produce a 0 when the button is not pressed and a 1 when the button is pressed.

3. To use the breadboard button, you must configure the GPIO pin correctly. This should be done in your code, before the while loop.
 - a. Direction -> Input
 - b. Mode -> Pulldown
 - c. To see examples of how to this, find the `board_init()` function call, right-click, Go to Implementation, and use the `BUTTON_0_PIN` configuration as a guide
4. Now modify your code to use the breadboard circuit push button to control the breadboard LED. To do this:
 - a. Comment out the `ONE_BUTTON_TWO_LED` macro and uncomment `TWO_BUTTON_TWO_LED`
 - b. Start filling in the `TWO_BUTTON_TWO_LED` section of code by copying and pasting the code from the `ONE_BUTTON_TWO_LED` section.
 - c. Add another if/else that checks the breadboard button to see if it is pressed.
 - i. If so, turn the breadboard LED on
 - ii. If not, turn the breadboard LED off
 - iii. Make sure to remove breadboard LED control from the if/else that checks Button 0.
 - d. Use the Atmel Debugger to verify that breadboard button presses are being detected:
 - a. Add a breakpoint at the line of code that checks the breadboard button pin
 - b. When it reaches the breakpoint:
 - i. Hold the breadboard button and click 'Step Over' in Atmel Studio. The debugger should go into the if/else block that indicates the button is pressed.
 - ii. Press the filled green arrow to 'Continue'
 - c. When it reaches the breakpoint again:
 - i. Do not hold the breadboard button down and click 'Step Over' in Atmel Studio. The debugger should go into the if/else block that indicates the button is not pressed.

Part 3 – System Analysis

1. Now we will use the Digilent Analog Discovery 2 to capture waveforms to characterize the signals in this system. Please follow the steps below:
 - a. Plug in Digilent Analog Discovery 2 to your computer via USB.
 - b. Open the Digilent -> WaveForms software on your computer.
 - c. Select 'Scope' on the left-hand side and use the software to view the output from the scope. It will take a little bit of time and practice to get used to this scope, as it does with any new piece of equipment.
 - d. I like to start out by setting the Mode to 'Screen', which just shows the signal over time without any specific triggers, etc. To capture a fast-moving signal and get more precision, you would need to use 'Repeated' mode and use triggers. But for now, 'Screen' mode is fine.



- e. Connect a ground wire from the Digilent GND (any black wire) to the Atmel GND.
- f. Connect the 1+ wire (orange wire in the top left of connector) to the same node that EXT1_PIN_7 connects to in the breadboard button circuit.
- g. Capture a waveform that shows the beginning and end of a button press, as shown above.
- h. Use the cursors to measure the ΔX of a single button press, as shown above. To get into cursor mode, double-click on the waveform and red cursors will appear. You can use these cursors to measure the duration of the button press (time from beginning to end of the press).
- i. Use the Snipping Tool to take a screenshot of the Waveforms window that shows the ΔX measurement, create a Word document, and paste this image into your Word document. Make sure to add relevant titles to each screenshot, as we will be adding more screenshots and info as we go.

Part 4 – Can you turn this into a flashlight?

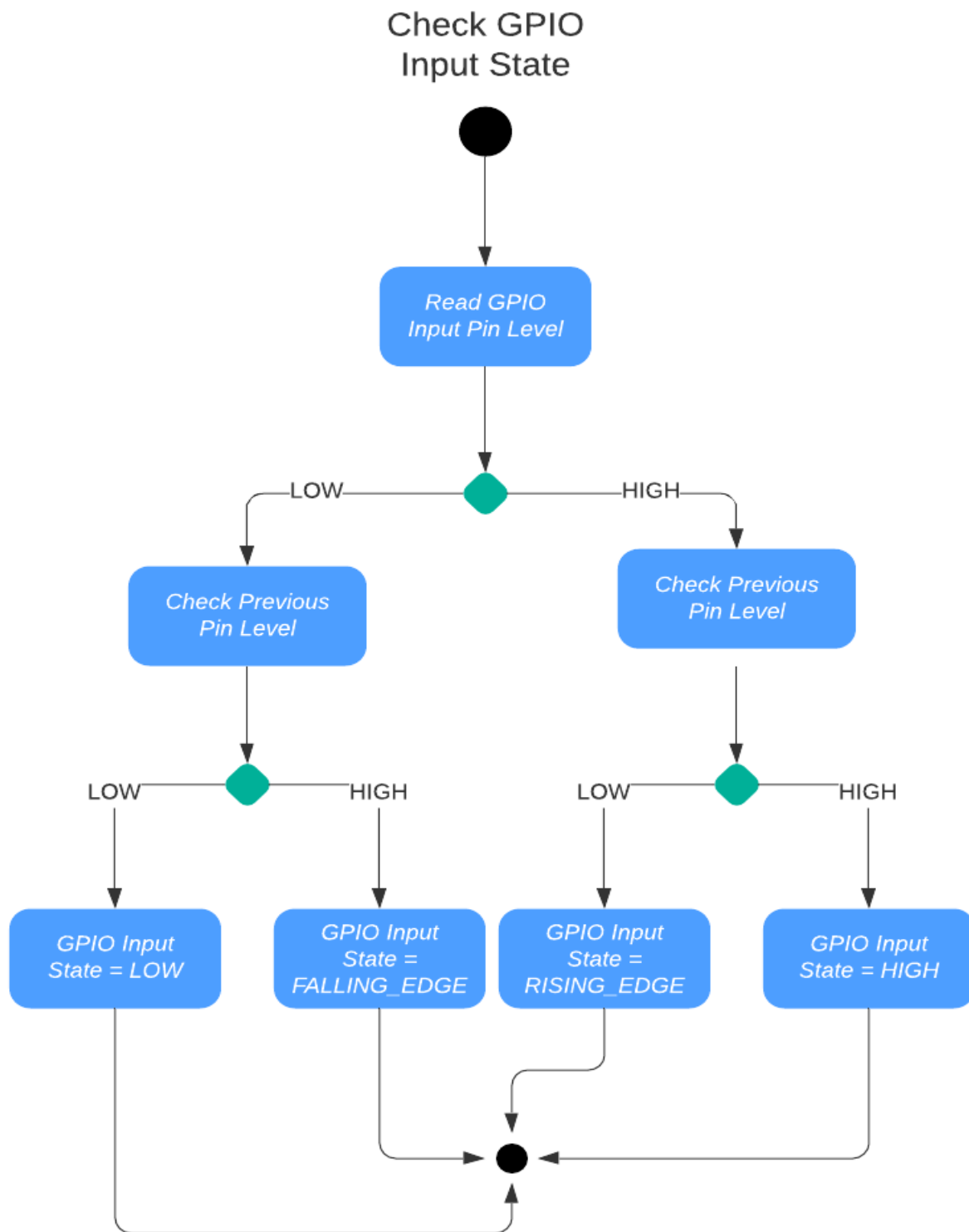
1. Update the logic in main.c to work like a standard flashlight:
 - a. When SW0 is pressed, turn LED0 on and keep it on (flashlight on), even once the button is released.
 - b. When SW0 is pressed again, turn LED0 off and keep it off (flashlight off), even once the button is released
 - c. When the breadboard button is pressed, turn the breadboard LED on and keep it on (flashlight on), even once the button is released.
 - d. When breadboard button is pressed again, turn breadboard LED off and keep it off (flashlight off), even once the button is released
 - e. The change from on -> off or off -> on should occur when the button is pressed, not when the button is released.
 - f. You should check at least 10 times to see if it works 100% of the time.
 - g. One restriction is that you may not add a second loop. In other words, the original while(1) is the only loop you are allowed to use.
 - h. The flashlight should initially be off.
 - i. Every line of code must have at least one line of comments explaining what the code is doing.
2. Most students will get to a point where it works sometimes but not others. When you reach this point, connect the Digilent as you did in part 2 where Channel 1 shows a button press. Channel 2 should show what the LED is doing by connecting the 2+ wire from the Digilent to the same node as EXT1_PIN_6.
 - a. Channel 2 will likely surprise you and should lead you toward understanding why it only works intermittently.
 - b. Once you have an understanding of why it is working intermittently, move to the next part of the lab.
3. You do not need to get this section fully operational before class next time, just give it a good try.

Part 5 – Design and Implementation

Now that you've had the opportunity to struggle a bit, let's put some thought into the design on this simple system. If you were able to get the flashlight working in the previous part, you still need to update it to use the methods in the rest of the manual.

To know when to turn the flashlight on or off, we need to poll the button pin and look for a transition from low to high or high to low. Essentially, we are looking for a rising edge or falling edge of the button signal. We only want to update the LED when we detect the correct edge on the button pin. Which edge we look for is dependent on the circuit.

- We observed with the Digilent that the breadboard button pin reads 0V when the button is not pressed and 3.3V when it is pressed. So a new breadboard button press is detected when the voltage goes from 0V to 3.3V, which is classified as a rising edge.
- We have also learned that the SW0 button the Atmel board reads 3.3V when not pressed and 0V when pressed. So a new button press is detected when the voltage drops from 3.3V to 0V, which is classified as a falling edge.
- So how do we detect an edge? By implementing the following UML Activity Diagram:



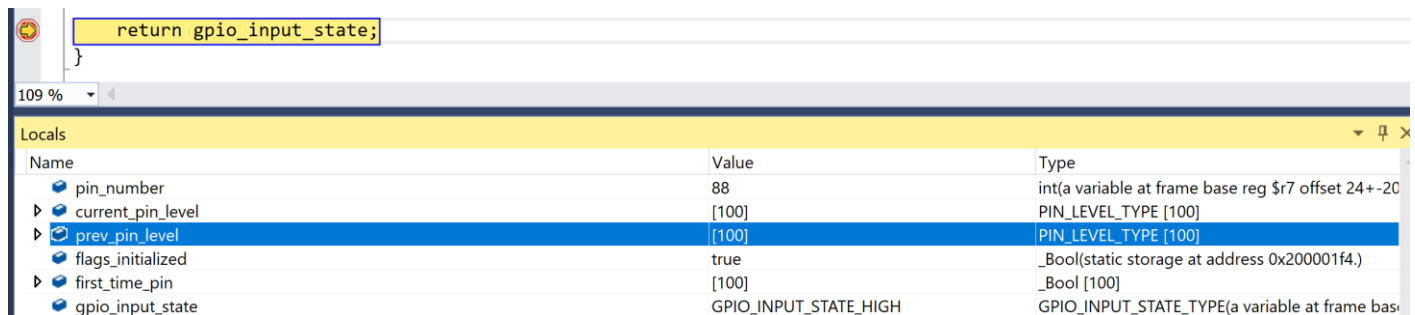
Implement check_gpio_input_state

The utilities.h file that I provided in Canvas has some starter code to help you implement this function. Please follow these instructions to implement the check_gpio_input_state function.

1. Open common.h and carefully read the comments and code for PIN_LEVEL_TYPE and GPIO_INPUT_STATE_TYPE. Make sure you understand the difference between the two.
2. Open utilities.h and carefully read the comments and provided code. If you understand what is provided, then it should not be difficult to finish the function.
3. Your job is to fill in the TODO section with logic to determine gpio_input_state. The logic is provided for you in the Check GPIO Input State UML Activity Diagram above.

To verify that this function works properly, we will use the Atmel Studio Debugger.

1. Comment out any code you have in your while(1) loop in main.c
2. Call check_gpio_input_state(BUTTON_0_PIN) in the while(1) loop in main.c. This should be the only active code in the while(1) loop.
3. Put a breakpoint in utilities.h at the following line:
 - a. return gpio_input_state;
 - b. You can add a breakpoint by clicking in the far-left margin on the line where you want the breakpoint.
4. Run the Debugger by clicking the filled green arrow. When it reaches the breakpoint, it will stop and breakpoint line will become yellow. When it stops, evaluate the 'Locals' in the window at the bottom. Since current_pin_level and prev_pin_level are arrays, you will need to expand those arrays and look at element 88 for each of them. Pin 88 for both of those arrays should have a value of PIN_LEVEL_HIGH.

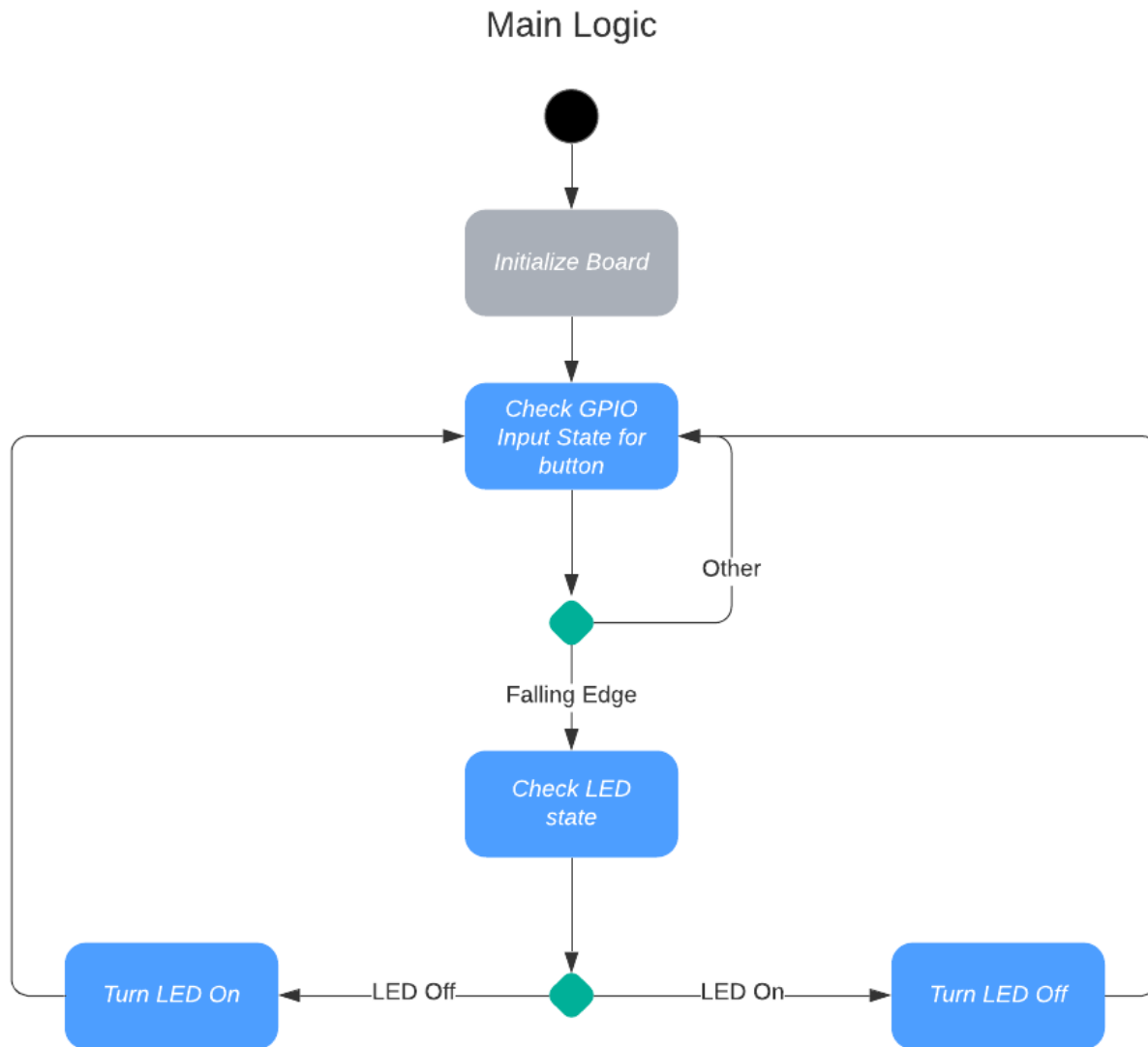


5. Now hold the push button down and click the filled green arrow again (this 'Continues' the program until it hits another breakpoint). Once it stops again, check the local variables again to see if they updated properly. Now prev_pin_level should be PIN_LEVEL_HIGH and current_pin_level should be PIN_LEVEL_LOW. gpio_input_state should be GPIO_INPUT_STATE_FALLING_EDGE.
6. Now hold the button down again and click 'Continue'. This time, the gpio_input_state should be GPIO_INPUT_STATE_LOW.
7. Now release the button and click 'Continue'. gpio_input_state should be GPIO_INPUT_STATE_RISING_EDGE.
8. Continue one more time to verify that the state goes back to GPIO_INPUT_STATE_HIGH.
9. If your function does not work properly the first time (which is to be expected), use the debugger to step through your code with the following buttons. You can hover over the buttons to see what each does.



Implement main.c

Now that we have a working `check_gpio_input_state` function, we can implement the following UML Activity Diagram for the main function. The diagram below shows the logic for the SW0 button/LED only. You will need to replicate the basic logic for the breadboard button/LED in your code.



Start by turning this UML Activity Diagram into a comment skeleton to get the basic logic. Remember that you will need similar logic for the breadboard button/LED also, so add that to the comment skeleton as well. The comment skeleton should look something like:

```

int main (void)
{
    // initialize board

    while (1)
    {
        // check GPIO Input State for button

        // if button state is FALLING EDGE
        {
            // if LED state is LED Off
            {
                // turn LED On
                // update LED state
            }
            // if LED state is LED On
            else
            {
                // turn LED Off
                // update LED state
            }
        }

        // if the button state is not FALLING EDGE, do nothing
    }
}

```

Once you have a comment skeleton like this, then fill in the details to make a working flashlight. The comment skeleton helps you to get started, but do not think that you just need one line of code for every comment. It is meant to get the general logic correct but it is not intended to be 100% complete.

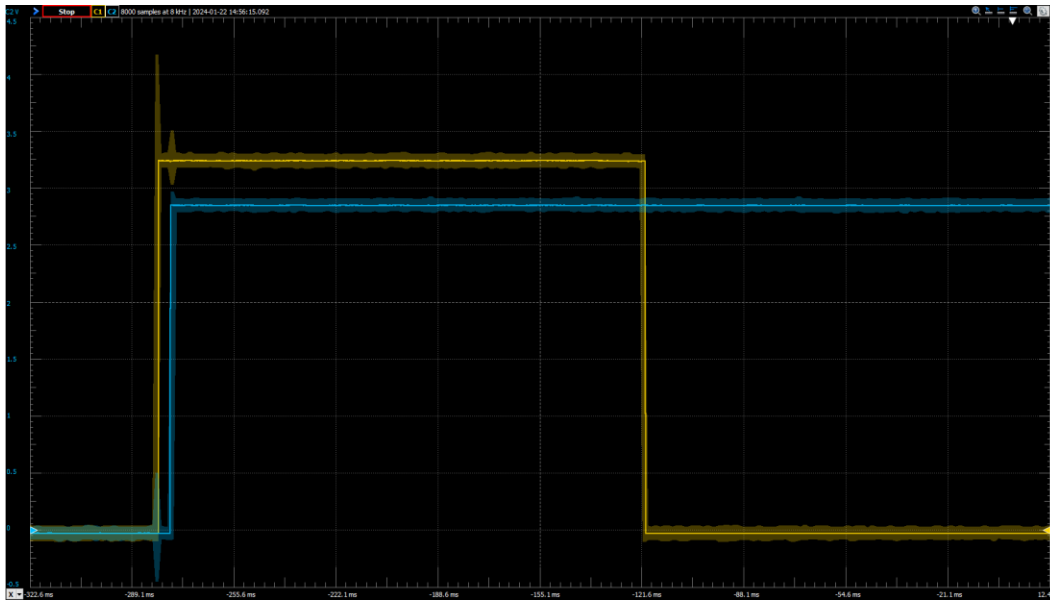
***IMPORTANT NOTE:** It is required that you use the enumerated data types that are defined in common.h. There should not be any instances of using a constant number like 0 or 1 in your code. The data types you are required to use are:

1. PIN_LEVEL_TYPE
2. GPIO_INPUT_STATE_TYPE
3. LED_STATE_TYPE

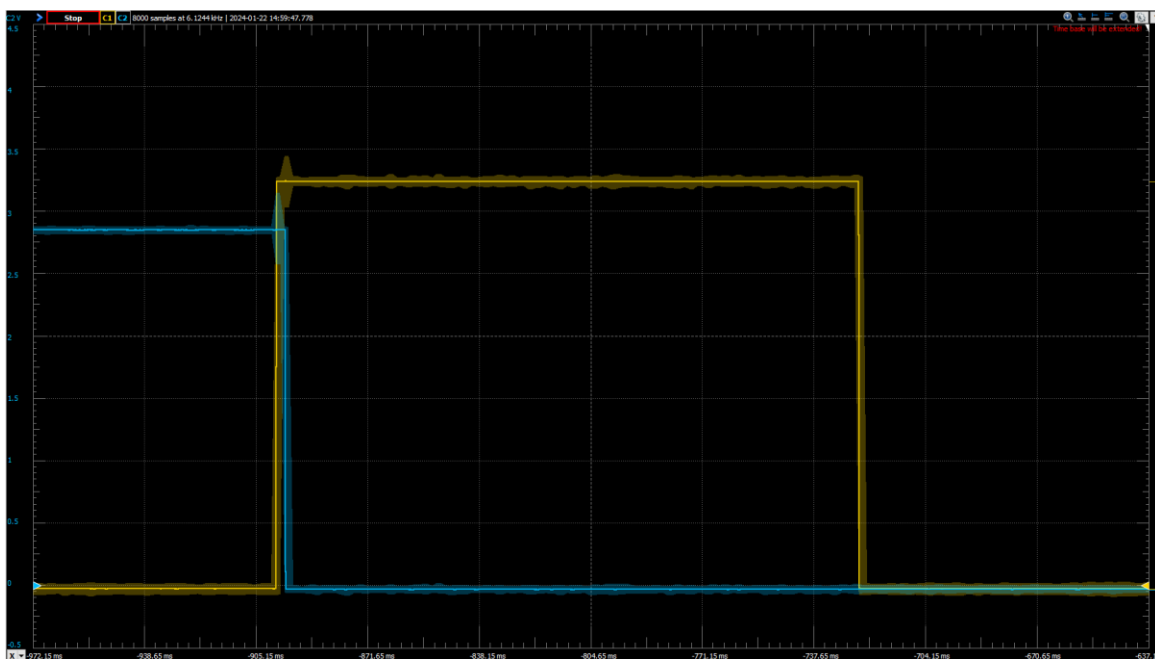
Final Screenshots

Now use the Digilent to capture two 2-channel screenshots that show the breadboard button on Channel 1 and the breadboard LED on Channel 2. These screenshots should capture the transition of the button and LED when turning the LED on and off. Please add these screenshots to the Word document. Here are some example screenshots that show what I am looking for:

1. LED Turning On



2. LED Turning Off



Submitting the Lab

1. Zip up the Atmel project by following the instructions for 'Submitting an Atmel Project' section in the appendix.
2. Submit the zip file and the Word document as 2 separate files on Canvas.

Lab A2 – Stopwatch State Machine

Part 1: State Machine Specification and Design

This lab must be designed and implemented using a state machine and meet the following specifications:

System Specification

User needs a stopwatch that operates as follows:

1. The stopwatch should start by displaying 0 on the LCD.
2. If SW0 is pressed while 0 is displayed on the LCD:
 - a. The stopwatch time should begin incrementing and should be displayed in seconds with 2 decimal places on the LCD. Example:
 - i. 23.57
 - b. When SW0 is pressed again, the stopwatch should pause and continue displaying the previously displayed stopwatch time. During this paused state:
 - i. If the SW0 button is pressed, the stopwatch should continue incrementing from where it left off.
 - ii. If the breadboard push button (SW1) is pressed, the stopwatch should reset to 0 and be ready to start again.
 - iii. NOTE: The external push button (SW1) should be ignored while the stopwatch is running.
3. If SW1 is pressed while 0 is displayed on the LCD:
 - a. The current clock time should be displayed on the LCD. We do not have an actual time source where we can get accurate time, but we can start the clock at a given time and have it accurately update from that point. In order to accelerate the process of testing, we will use MM:SS rather than HH:MM. Your starting time should be 09:30, where 09 represents minutes and 30 represents seconds. From that starting time, the clock should update accurately.
 - b. When SW1 is pressed again, the stopwatch should go back to its original state and display 0.
 - c. NOTE: SW0 should be ignored while the clock time is displayed.

IMPORTANT NOTE: If you meet all of these specifications but do not use the state machine methodology discussed in lecture and described below, the highest grade you can get is a 30/100.

State Machine Design

Based on the System Specifications, you must design a state machine for this system. Please submit the PDF of the State Diagram to the design assignment in Canvas. The three states for this system should be defined as:

```
typedef enum{
    IDLE = 0,
    RUNNING = 1,
    PAUSED = 2,
    CLOCK = 3
} STOPWATCH_STATE_MACHINE_TYPE;
```

There are only two types of events that could trigger a state change:

- SW0 Button Press
- SW1 Button Press

Using this information and the specification above, create a State Diagram for this system. For each state, you need to include:

- Name of the state in the top part of the box
- Actions that should occur in that state in the bottom part of the box
 - o This could be something like “Display incrementing stopwatch time on LCD”
 - o Please use the terms “stopwatch time” and “clock time” to differentiate between the two.

For each event that causes a state transition, you need to include two pieces of information on the line:

- A general description of the event. In this case, either ‘SW0 Press’ or ‘SW1 Press’.
- An enumeration for that event. In this case, we will be using the enumerations from GPIO_INPUT_STATE_TYPE (defined in previous lab and below). This will help us to identify what physical events we should be looking for.

```
typedef enum{
    GPIO_INPUT_STATE_LOW = 0,
    GPIO_INPUT_STATE_RISING_EDGE = 1,
    GPIO_INPUT_STATE_HIGH = 2,
    GPIO_INPUT_STATE_FALLING_EDGE = 3
} GPIO_INPUT_STATE_TYPE;
```

Part 2: Implementing the State Machine

Logistics:

In your ELC4438 directory, create a new directory called Lab A2. Copy and paste the entire project from Lab A1 to Lab A2. Use this as your starting point.

State Machine Code Concepts

A state machine is an extremely effective way of implementing some embedded systems such as this stopwatch, and it is required that you use a state machine for your solution. For it to be effective, it needs to be executed in a consistent and orderly fashion. Your main function will start with all necessary initialization, then it will enter a while(1). As discussed in lecture, you should do the following inside the while loop. Please see the lecture for an example. Also note that you are required to follow this method of implementation.

1. Check for any events that might initiate a state change.
2. Enter a ‘switch’ statement where each case is a system state. Make sure to have a ‘break;’ at the end of each case.
3. The state machine will enter the ‘case’ for the current system state (regardless of what events have been detected).
 - a. If no state changing events have been detected, then the software should perform whatever actions need to occur while in that state.
 - b. If a state changing event has been detected, then the software should perform any necessary actions, but then it should update the system state variable to the new state.
 - i. Sometimes it is necessary to take other actions such as saving a timestamp or clearing the value of a variable when this state changing event is detected.

4. If the state changed during that pass through the switch statement, then the system state variable will have been updated so that it will go to the new state the next time through the loop.

State Machine Code Implementation

Using this method of implementing a State Machine, you should implement the Stopwatch State Machine Design that you created for this lab. To do this:

1. Add the `STOPWATCH_STATE_MACHINE_TYPE` (specified in Part 1) to `main.c`. This is specific to this stopwatch lab, therefore it should remain in `main.c`.
2. Declare variables and perform any initialization steps in `main.c` before the while loop.
3. Implement the state machine per your State Diagram, with the following exception:
 - a. Do not try to display the time value on the LCD. There are two problems with trying to display the time at this point:
 - i. You don't yet have a time value
 - ii. If you divide your focus between the state machine and time aspect of the lab, it will be much more difficult to accomplish.
 - b. Instead of displaying time, you should display the state number on the LCD so that you can verify that you can traverse through the states correctly.
 - i. IDLE – display 0
 - ii. RUNNING – display 1
 - iii. PAUSED – display 2
 - iv. CLOCK – display 3
 - c. The Debugger is a very good way to debug any state machine issues that you have, so I highly encourage you to use it during this part of the lab.
 - d. There is one problem though....you don't know how to use the LCD yet. Instructions are below.

Atmel LCD Usage

The LCD screen (C42412A) is provided with your Atmel board. Make sure that it is firmly and evenly seated on the Atmel board. To get the LCD working properly, you will need to do the following:

1. Check the 'VLCD A' and 'VLCD BC' jumpers are 'open'. An 'open' jumper implies that the jumper is not electrically connecting the 2 pins. A 'closed' jumper implies that the jumper is electrically connecting the 2 pins. Some microcontroller pins go to multiple destinations, and these jumpers are used to re-route the destination of those pins. A 'closed' jumper would re-route some of the traces that go to the LCD, therefore making the LCD not work properly. Since we want to keep the jumpers but we do not want them to be 'closed', you can just connect the jumper to 1 pin and leave it hanging off the side so that it does not connect to the second pin.
2. Add the Atmel Software Framework (ASF) module for the C42412A LCD. See detailed instructions in the appendix of this document. Please note that the ASF module name does not begin with C42412A, so it is a bit hard to find in the list unless you use the 'Search for modules' box in the 'Available Modules' pane.
3. Call `c42412a_init()` in the initialization section of the code. Please note that order of initialization functions does matter. `board_init()` should always be called first. As we add capabilities, you will see that we always need to be careful of the order of initialization.
4. For details on the c42412a ASF module, visit the appropriate links that are provided for you in 'ASF_Documentation_Links.docx'

Part 3: Displaying Stopwatch Time Values on LCD

We do not have actual time values yet, but we can develop the ability to display stopwatch time values. To do this, we can choose sample values to display. To do this:

1. Create a function in main.c with the following definition:
 - a. `void display_stopwatch_time(uint32_t ms_value)`
 - b. This function should display the time in stopwatch format (as discussed in the System Specification) on the LCD. The input parameter is the `ms_value`, which is simply the number of ms that you want to display on the LCD.
 - c. Please note that the C42412A ASF Module does not have a function to display a floating point number. But you can display an integer and place a decimal point icon in the appropriate place. Macros that define each icon are defined in the C42412A ASF Module code in `c42412a_segmap.h`.
2. To test your `display_stopwatch_time` function:
 - a. Immediately before the `while(1)` loop, add the following line of code:

```
uint32_t test_value = 5432;  
display_stopwatch_time(test_value);  
return 0;
```

3. With a `test_value` of 5432, you should see the value 5.43 on the LCD screen.
4. Repeat the test with the following values (expected lcd display shown to the right of ->)
 - a. 321 -> .32
 - b. 76543 -> 76.54
 - c. 123456 -> 123.45
 - i. Note that you do not need to round up.
5. When testing is complete, make sure to remove the temporary code.

Part 4: Displaying Clock Time Values on LCD

We do not have actual time values yet, but we can develop the ability to display clock time values. To do this, we can choose sample values to display. To do this:

1. Create a function in main.c with the following definition:
 - a. `void display_clock_time(uint32_t ms_value)`
 - b. This function should display the time in clock format (as discussed in the System Specification) on the LCD. The input parameter is the `ms_value`, which is the number of ms that have elapsed since the start of the program. You should use `ms_value` to add onto the starting clock value to accurately display the time on the LCD.
2. To test your `display_clock_time` function:
 - a. Immediately before the `while(1)` loop, add the following line of code:

```
uint32_t test_value = 0;  
display_stopwatch_time(test_value);  
return 0;
```

3. With a `test_value` of 0, you should see the value 09:30 on the LCD screen.
 - a. Note that the leading 0 is necessary. You must always have 4 digits displayed for the clock value.
4. Repeat the test with the following values (expected lcd display shown to the right of ->)

- a. 5123 -> 09:35
 - b. 34321 -> 10:04
 - c. 123456 -> 11:33
5. When testing is complete, make sure to remove the temporary code.

Part 5: Implementing Time

As discussed in lecture, we use the SysTick Timer to keep up with the time value for our systems. To do this:

6. Add the System Clock Control ASF Module using the ASF Wizard.
7. To ensure accurate time, we need to configure the clock to use a crystal oscillator as its source and use the DFLL to set the clock frequency. To do this, you need to modify `conf_clock.h` to look like:

```
#ifndef CONF_CLOCK_H_INCLUDED
#define CONF_CLOCK_H_INCLUDED

#define CONFIG_SYSCLK_SOURCE      SYSCLK_SRC_DFLL
/* 0: disable PicoCache, 1: enable PicoCache */
#define CONFIG_HCACHE_ENABLE      1
/* Fbus = Fsys / (2 ^ BUS_div) */
#define CONFIG_SYSCLK_CPU_DIV      0
#define CONFIG_SYSCLK_PBA_DIV      0
#define CONFIG_SYSCLK_PBB_DIV      0
#define CONFIG_SYSCLK_PBC_DIV      0
#define CONFIG_SYSCLK_PBD_DIV      0
#define CONFIG_USBCLK_SOURCE      USBCLK_SRC_PLL0
/* Fusb = Fsys / USB_div */
#define CONFIG_USBCLK_DIV          1
#define CONFIG_PLL0_SOURCE         PLL_SRC_OSC0
/* Fpll0 = (Fclk * PLL_mul) / PLL_div */
#define CONFIG_PLL0_MUL             (48000000UL / BOARD_OSC0_HZ)
#define CONFIG_PLL0_DIV             1
#define CONFIG_DFLL0_SOURCE         GENCLK_SRC_OSC32K
/* Fdfl1 = (Fclk * DFLL_mul) / DFLL_div */
#define CONFIG_DFLL0_FREQ           48000000UL
#define CONFIG_DFLL0_MUL             (CONFIG_DFLL0_FREQ / BOARD_OSC32_HZ)
#define CONFIG_DFLL0_DIV            1

#endif /* CONF_CLOCK_H_INCLUDED */
```

8. To start the system clock, call the following function immediately after `board_init`:
 - a. `sysclk_init();`
9. Now you need to configure the SysTick interrupt to occur every 1 ms by calling:
 - a. `SysTick_Config(uint32_t val_for_1ms);`
 - b. The input parameter specifies how many times you want the SysTick timer to increment before it interrupts.
 - c. The value that you pass into `SysTick_Config` cannot be a constant number. It must be calculated using the result of calling:

- i. `sysclk_get_cpu_hz()`
 - ii. This function returns the actual CPU speed, not the theoretical speed
 - 1. While we tell the frequency to be 48MHz in `conf_clock.h`, it is not that precise, therefore we need to use the internally measured clock speed, which is returned by `sysclk_get_cpu_hz()`
- 10. Finally, add code in `utilities.h` to handle the SysTick interrupt. This code should create a volatile global variable called 'ticks' (or another name of your choice) and initialize it to 0. Also add the `SysTick_Handler` function as discussed in lecture. The 'ticks' variable should be incremented every time the SysTick interrupt occurs. If the SysTick interrupt is correctly configured to increment every 1ms, then this 'ticks' variable represents the number of ms since the program began.
- 11. We made 'ticks' global so that it could be accessed from other functions.
 - a. IMPORTANT NOTE: Though it can be accessed in other functions, it should never be modified in other functions.
- 12. To test your `display_stopwatch_time` function with a real incrementing time value, let's add some temporary code to the top of the `while(1)` loop in `main`:
 - a. `display_stopwatch_time(ticks);`
 - i. Please comment out or remove the function calls that write the state numbers (0, 1, 2, 3) to the LCD screen, as these will impact your time values on the LCD.
 - b. This will display the time since the program began. It should match up to the real amount of time since the program began. You can use a phone stopwatch to check it. Load your code onto your board, then simultaneously start your phone stopwatch and press the reset button the Atmel board. The time values should match very closely.

Part 6: Final Integration

At this point, we have:

1. A State Machine that traverses the states correctly
2. The ability to display a stopwatch time value and clock time value on the LCD
3. A clock that keeps up with time since the program began

Now, let's remove the temporary call to `display_stopwatch_time` that we just added in the previous section to verify that our time value works correctly. It is time to use our `display_stopwatch_time` and `display_clock_time` functions to display the correct time values according to the system specifications. Now you should call `display_stopwatch_time` and `display_clock_time` from within your state machine in the appropriate places. Please do not think that this is as simple as just calling `display_time(ticks);` The value passed into `display_time` will require some calculations to account for when the system initially goes into `RUNNING`, when the system is in `PAUSE`, etc.

IMPORTANT NOTE: You may not change the value of the 'ticks' variable anywhere except for inside of `SysTick_Handler`. This is our system time, so we want to view it but not change it in our main application code.

Submitting the Lab

1. Zip up the Atmel project by following the instructions for 'Submitting an Atmel Project' section in the appendix.
2. Submit the zip file on Canvas.

Lab A3 – Event-Driven System

System Specification

The Event-Driven Lab system requirements are:

1. When the system starts up, it should turn on LED0, then turn it off after 3 seconds. 3 seconds later, it should turn the LED back on and it should stay on. This should be done in main.c during initialization. During this time, nothing else should be seen (no LCD output, no LCD backlight).
2. The rest of the requirements refer to the steady state operation, which is the time after the LED is turned back on.
 - a. The system time (number of seconds since the program started) should be displayed on the LCD screen and should be continuously updating. The LCD should show the number of seconds with 2 decimal places of precision. For example, after 13472 milliseconds, the LCD would show:
 - i. 13.47
 - ii. The highest number that you need to display accurately is 999.99.
 - b. When the light sensor does not detect light, it must turn on the LCD backlight.
 - c. When the light sensor detects light, it must turn off the LCD backlight.
 - d. While the SW0 button is pressed down, the LCD must show a full wireless signal.
 - e. While the SW0 button is not pressed down, the LCD must show a single dot wireless signal.
 - f. The system should always be generating a 2kHz square wave. This will be viewed by connecting the square wave pin to the breadboard LED, which will be illuminated at half brightness.

Additional Requirements

1. There must not be any code in the main while(1) loop. All code (except for initialization code) must live in an interrupt handler. All interrupt handlers should be in utilities header files.
2. The system time update to the LCD should be done in the SysTick Handler.
3. The SW0 button must use the EIC Interrupt Handler. This requires:
 - a. EIC ASF Module
4. The light sensor must use the GPIO Interrupt Handler. This requires:
 - a. GPIO ASF Module
5. The square wave should be generated using a Timer Counter. This requires:
 - a. Timer Counter ASF module
 - b. NOTE: There are two TC – Timer Counter modules. Please choose the one that has an “Info” field that starts with “Driver for the TC...”. The info field is in a box below the module list in the ASF Wizard.
6. In Lab A1, I told you that you were not allowed to have any additional loops other than main while(1) loop. The reason is that loops can potentially stall the system, breaking the polling paradigm. In the case of an event driven system, additional loops may be used (with caution) since we are not polling.
7. The IO1 Xplained Pro board must be connected to EXT2.

Part 1 - Design

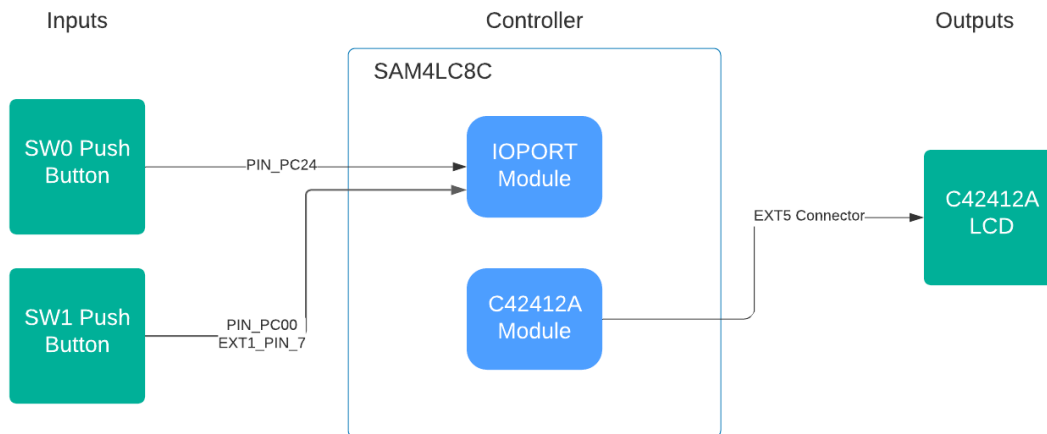
In this lab, we will create a detailed architecture diagram of the system. This will require you to look through the documentation to figure out the details of the hardware used in the system. The architecture diagram should help us to identify:

1. The inputs and outputs
2. The pin numbers of each input/output
3. The ASF modules needed to interact with these inputs and outputs
 - a. The 'Additional Requirements' section of this document discusses each module that needs to be added.

After completing the architecture diagram:

1. Submit a PDF of the Architecture Diagram to the 'Lab A3 Design' assignment on Canvas.
2. Identify any ASF modules that are not currently in your project and add them to your project via the ASF Wizard.

Below is an example of an architecture diagram from Lab A2.



Part 2 – Unit Testing

This lab has more inputs and outputs than previous labs, which can make it more overwhelming to approach. Often, students look at the system and try to implement the entire system at once, resulting in confusion and lack of success. Therefore, I'm going to walk you through the process of creating a solution for this system. Please read the entire lab manual, then begin completing this section.

We will start by creating new functionality and then testing this new functionality individually. This is often referred to as unit testing. In each step, I will provide the code for your main function. Note that I use the minimum possible code in main for these tests. The intent is that we test each new functionality without any other functionality (from this lab or another) interfering with the results. Please use the 'Main Function' code provided.

Step 1 – System Time

This part should be very straightforward. You already have the system time from the SysTick_Handler function from the previous lab. We are simply outputting the "ticks" value that is incremented in the SysTick_Handler.

1. To display system time, we need to call the display_time function that we created in the previous lab.
2. Since we cannot have any code in the main while(1) loop, the display_time function must be called from the SysTick_Handler.
3. Verify that the time is displayed correctly on the LCD.

Main function:

```
board_init();  
sysclk_init();  
c42412a_init();  
SysTick_Config(sysclk_get_cpu_hz() / 1000);
```

Step 2 – LCD Backlight

This part will simply build and test the capability of turning the LCD backlight on and off.

1. Create an LCD_BACKLIGHT_PIN macro in common.h.
 - a. LCD_BACKLIGHT_PIN should be set to the pin you identified in the architecture diagram
 - b. There is a macro for every microcontroller pin that looks something like:
 - i. PIN_PC07
 - c. An example (not the correct pin number) of the macro would be:
 - i. #define LCD_BACKLIGHT_PIN PIN_PC07
2. Create an enumerated data type called LCD_BACKLIGHT_TYPE in common.h. The enumerated values in this type are LCD_BACKLIGHT_OFF and LCD_BACKLIGHT_ON.
3. Create a new file called utilities_lcd.h
4. Create a function called configure_lcd_backlight in utilities_lcd.h to configure the LCD backlight pin as an output.
5. Create a function called set_lcd_backlight in utilities_lcd.h to turn the backlight on or off, depending on the argument passed to the function. The definition for this function should be:
 - a. void set_lcd_backlight(LCD_BACKLIGHT_TYPE level)

6. Configure the LCD backlight and turn the backlight on in main.
7. Verify that the LCD backlight comes on.
8. Change your main code to turn the backlight off after configuring the pin.
9. Verify that the LCD backlight turns off.

Main function:

```
board_init();  
sysclk_init();  
configure_lcd_backlight();  
set_lcd_backlight(LCD_BACKLIGHT_ON); // change from ON to OFF for last step
```

Step 3 – Set up the GPIO interrupt for the light sensor

This part will configure the GPIO interrupt and verify that the interrupt handler is called when a change to the light sensor pin occurs.

1. Create a LIGHT_SENSOR_PIN macro in common.h
 - a. LIGHT_SENSOR_PIN should be set to the pin you identified in the architecture diagram
2. Create a new file called utilities_light_sensor.h
3. Within this file, add two functions:
 - a. configure_light_sensor
 - b. light_sensor_handler
4. The configure_light_sensor function should do the following:
 - a. Configure the light sensor pin
 - i. Configure as an input
 - ii. Based on the circuit diagram of the light sensor in the IO1 Xplained Pro schematic, should you configure the mode as pullup, pulldown, or leave as high impedance (default)?
 1. If you choose high impedance, you should not call ioport_set_pin_mode
 - b. Configure the GPIO Interrupt for the light sensor using GPIO ASF module functions. See GPIO Module API documentation (link is in ASF_Documentation_Links.docx in Library section of Canvas) for information on the GPIO interrupt ASF functions. To configure the interrupt:
 - i. Enable the pin interrupt
 - ii. Set the pin callback to point to light_sensor_handler
5. Call configure_light_sensor from main.
6. Use the debugger to verify that the interrupt handler is called when there is a transition from light to dark or from dark to light.
 - a. Shine a flashlight on the sensor for light and cover the sensor for dark.

Main function:

```
board_init();  
sysclk_init();  
configure_light_sensor();
```

Step 4 – Change the LCD backlight based on the light sensor

This part will integrate the light sensor and the LCD backlight.

1. Add code to the light sensor handler to turn the LCD backlight on or off when the light sensor interrupt occurs.
2. Verify that the LCD backlight turns on when you cover the sensor and turns off when you shine a flashlight on the sensor.

Main function:

```
board_init();  
sysclk_init();  
c42412a_init();  
configure_lcd_backlight();  
configure_light_sensor();
```

Step 5 - Set up the EIC interrupt for SW0

This part will configure the EIC interrupt and verify that the EIC interrupt handler is called when the button is pressed.

1. Create a new file called utilities_eic.h
2. Use the EIC Quick Start Guide (link is in ASF_Documentation_Links.docx) to create two functions:
 - a. eic_setup
 - b. eic_callback
3. 'Go to Implementation' on the board_init function and find the macro needed that will enable the EIC pin to be configured properly. Add this macro to conf_board.h.
4. Call eic_setup from main.
5. Use the debugger to verify that the interrupt handler is called when the button is pressed.

Main function:

```
board_init();  
sysclk_init();  
eic_setup();
```

Step 6 - Change the wireless icon based on the SW0 interrupt

This part will display wireless signal while the button is being pressed and display a single dot wireless icon while the button is not pressed.

1. Add code to eic_callback to display a full wireless icon while the button is pressed.
2. Add code to eic_callback to display a single dot wireless icon while the button is not being pressed.
3. Verify operation by viewing the wireless icon on the LCD display while the button is being pressed and while the button is not being pressed.

Main function:

```
board_init();  
sysclk_init();  
c42412a_init();  
eic_setup();
```

Step 7 - Set up the Timer Counter to interrupt at a certain frequency

This part will configure the timer counter interrupt and verify that the interrupt handler is called.

1. Create a new file called utilities_tc.h

2. Copy the contents of the `utilities_tc.h` file on Canvas under Files -> Labs -> Lab A3 and replace the contents of your file.
3. Fill in the two TODO parts of the `configure_tc` function to create a timer counter that can be used for generating a 2kHz square wave.
4. Call `configure_tc` from main during your initialization section
5. Use the debugger to verify that `TC00_Handler` is being called. It is very difficult to verify the timing with the debugger, so we will verify the timing in the next part.

Main function:

```
board_init();  
sysclk_init();  
configure_tc();
```

Step 8 - Turn the breadboard LED on/off using the timer counter

This part will use the `TC00_Handler` to create a square wave.

1. Complete the TODO section in `TC00_Handler`. This code should turn the breadboard LED pin either on or off each time the interrupt handler is called. You will need to use a static variable to keep track of whether the LED is currently on or off.
2. Connect your breadboard LED circuit to your Atmel board as you have in previous labs.
3. Verify that the LED is dimly illuminated. If it is not illuminated, you can use the debugger and/or your multimeter, and/or the Digilent to determine why.

Main function:

```
board_init();  
sysclk_init();  
ioport_set_pin_dir(BREADBOARD_LED_PIN, IOPORT_DIR_OUTPUT);  
configure_tc();
```

Step 9 - Measure the breadboard LED pin frequency using Digilent

This part will use the Digilent to verify the frequency of the square wave.

1. Use the oscilloscope feature of the Digilent to capture the square wave.
2. Verify that it is a 2kHz square wave.
3. Please capture the square wave and use the cursors to show that the frequency is 2kHz.
4. Paste the capture with the measurement into a Word document and submit it with this lab.

Part 3 – System Integration

Now that we have each of these capabilities working individually, we need to start integrating them together and work toward making a system that meets the steady state requirements.

Step 1 – Integrate System Time and Light Sensor Capability

Now we will integrate system time and the light sensor. Use the main function below. We expect to see the time incrementing and the LCD backlight turn on/off depending on the light level read by the light sensor.

Main function:

```
board_init();
sysclk_init();
c42412a_init();
SysTick_Config(sysclk_get_cpu_hz() / 1000);
configure_lcd_backlight();
configure_light_sensor();
```

Step 2 – Integrate EIC Interrupt Capability

Now integrate the EIC interrupt with the system from the previous step. Verify that the time increments, the LCD backlight responds to the light sensor, and the LCD shows the wireless icon appropriately while SW0 is pressed. Do not be surprised if it does not work perfectly. Right now, we just want to see that each capability generally works.

Main function:

```
board_init();
sysclk_init();
c42412a_init();
SysTick_Config(sysclk_get_cpu_hz() / 1000);
configure_lcd_backlight();
configure_light_sensor();
eic_setup();
```

Step 3 – Integrate Timer Counter Capability

Now integrate the Timer Counter with the system from the previous step. Verify that the time increments, the LCD backlight responds to the light sensor, and the LCD shows the wireless icon appropriately while SW0 is pressed. Also verify that the breadboard LED illuminates at half brightness.

Main function:

```
board_init();
sysclk_init();
c42412a_init();
SysTick_Config(sysclk_get_cpu_hz() / 1000);
configure_lcd_backlight();
configure_light_sensor();
eic_setup();
ioport_set_pin_dir(BREADBOARD_LED_PIN, IOPORT_DIR_OUTPUT);
ioport_set_pin_level(BREADBOARD_LED_PIN, BREADBOARD_LED_ON);
configure_tc();
```

Step 4 – Configure Interrupt Priorities

At this point, your system should generally work, but it will likely have some hiccups due to interrupt priority issues. It is time to configure the priorities. You should configure the priorities after the other initialization code in the main function but before the while(1) loop.

1. Use the NVIC_SetPriority function to specifically tell the NVIC what priority value you want for each interrupt. Do not trust the initialization functions to set the priority, even if it had a priority parameter.
2. Immediately before the while(1) loop, call the NVIC_GetPriority function for each interrupt and store these priority levels in variables.
3. Use the debugger to verify that the NVIC is reporting the priority levels according to your desired priority. The priority levels do make a difference in the operation, so we want to be sure that our priorities are configured the way we think they are.
4. Now verify that your system works correctly according to the steady-state specification (requirement 2 of the system specification).

Step 5 – Implement and utilize mdelay

Now we just need to add the mdelay function and turn LED0 on during the delay.

1. Create a new function in utilities.h
 - a. static void mdelay(uint32_t delay_ms)
2. This function should receive the desired delay expressed in a number of milliseconds. To do this, you should stay in a while loop until the desired number of milliseconds has elapsed.
3. In the initialization section of main, turn on LED0, call mdelay(5000), then turn off LED0. LED0 should stay on for 5 seconds then turn off.

Main Function:

```
board_init();
sysclk_init();
c42412a_init();
SysTick_Config(sysclk_get_cpu_hz() / 1000);
ioport_set_pin_level(LED_0_PIN, LED0_ACTIVE);
mdelay(5000);
ioport_set_pin_level(LED_0_PIN, LED0_INACTIVE);
```

Step 6 - Verify that the entire system works properly and meets the specification

Now it is time for final assembly. We have the working pieces, now we just need to put them all together in the right order.

1. Verify that for the first 5 seconds of operation, LED0 is on and there is nothing on the LCD, including the backlight.
2. After the 5 seconds has elapsed, LED0 should turn off and the steady-state operation should begin. Despite there being no code in the while(1) loop in main, the system should respond to all inputs appropriately.
3. Make sure that the LCD backlight is correct when the system starts steady-state operation. This will require a bit of extra code at the end of the initialization section to check the light sensor pin and set the backlight appropriately. To test:

- a. Cover the light sensor and press the reset button, wait the 5 seconds, and verify that the LCD backlight comes on after the 5 seconds.
 - b. Shine a flashlight on the light sensor and press the reset button, wait the 5 seconds, and verify that the LCD backlight does not come on after the 5 seconds.
4. You do not need to test or account for the case where the button is pressed when the system is started up. We will assume that the button press event can only occur during steady-state operation.

Submitting the Lab

1. Zip up the Atmel project by following the instructions for 'Submitting an Atmel Project' section in the appendix.
2. Submit the zip file on Canvas. Also submit the Word doc with your Digilent capture of the 2kHz square wave.

Lab A4 - Digital Thermometer

System Specification

User needs a digital thermometer that does the following:

1. Monitors temperature and displays it on the LCD
 - a. Temperature updates every 200 ms
 - b. Resolution (smallest distinguishable change in output) of temperature data should be 0.0625 degrees C
 - c. Temperature is reported to 3 decimal place on the character LCD (for example, 98.375)
 - d. Units (Celsius or Fahrenheit) should also be displayed on the character LCD. Output should look like one of the examples below:
 - i. Temp: 21.875 C
 - ii. Temp: 77.258 F
 - e. The thermometer does not need to be able to measure negative temperatures or temperatures above 99.999.
2. Temperature can be toggled between Fahrenheit and Celsius with SW0 button.
3. The temperature value should also be printed to Putty in the following format:
 - a. 23.1875 C
 - i. Units (C or F) should match the units being displayed on the LCD
 - ii. Atmel Studio has a bug with printing floats that we fixed when we created the project. If you have started a new project since then, please make sure to apply the fix from the Atmel Startup Instructions near the beginning of the manual.

Additional Requirements

1. The temperature sensor is on the IO1 XPlained Pro board. This board should be connected to EXT2 for this lab. To interface with the temperature sensor using TWIM, you will need to add the following ASF Module:
 - a. TWI – Two-Wire Master Interface (driver)
2. You must use the Virtual Com Port to print to Putty. The stdio ASF module is needed to assist with printing to the Virtual Com Port. Please see section later in this document for details on how to use it.
3. You must also submit screenshots of the output of the Protocol Analyzer on the Digilent Analog Discovery 2 device.
 - a. The screenshot should show the I2C messages from approximately the first 5-10 seconds of program execution. All that I really want to see are the messages that are sent at the beginning for configuration, then a few samples of temperature reading messages.
4. You must create a file called `utilities_temperature_sensor.h` that contains all functions relating to the temperature sensor. This file should include the following functions:
 - a. `void initialize_temperature_sensor(void)`
 - b. `float read_temp_sensor(TEMPERATURE_UNIT_TYPE tempUnits)`
 - i. `TEMPERATURE_UNIT_TYPE` should be created and should have the following enumerations:
 1. `CELSIUS`

2. FAHRENHEIT

- c. An individual function for each TWIM message you will be sending.
5. When creating structs to use with the TWIM module, you should make sure to initialize ALL of the members of the struct. There are additional struct members that the Quick Start Guide does not show. Please make sure to look at the struct definitions for the full list of members. Any members not shown in the Quick Start Guide can be set to 0.
6. In main.c, create a function called display_temp. This function is implemented in main, as it is specific to our LCD rather than the temperature sensor itself.
 - a. void display_temp(float temp, TEMPERATURE_UNIT_TYPE units)
7. Your main function should consist of:
 - a. Checking for a button press and changing units if it is pressed.
 - b. Calling the read_temp_sensor function.
 - c. Printing the temperature to the console.
 - d. Calling the display_temp to display the temp
 - e. Please note that the temperature sensor should only be read once every 200ms, but the button will need to be read more often so that we do not miss button presses.

Part 1 – Design

In this lab, we will be creating 2 design documents before beginning implementation:

1. ICD
 - a. The ICD should show every message that will need to be used in this lab.
 - b. See Lecture slides for an example and explanation. Here are the guidelines for each message shown in the ICD:
 - i. The address byte should always have every bit identified and shown on the diagram.
 - ii. The data byte(s) should have every bit identified if the master is sending the data. If the master is receiving data, you cannot fill in the bits because you do not know what it will send you.
 1. However, if the message is a Read Temperature message, you should indicate the equation used to convert the data bytes into a valid temperature value.
 - iii. Every byte on the diagram should be labeled either:
 1. Microcontroller Sends
 2. Temp Sensor Sends
 - c. I recommended doing this in Word, but you can try it in Lucid Chart if you prefer. I have not tried it in Lucid Chart.
2. Architecture Diagram
 - a. Make sure to include Putty as an output. Putty should connect to the 'Stdio Module' in the controller. Our architecture diagram does not need to show the EDBG, USB, etc. The line between the Stdio module and Putty should specify:
 - i. The USART module used, such as USART0, USART1, etc.
 - ii. The TX and RX pins that go from the microcontroller to the EDBG.
 - b. The temperature sensor should be connected to the TWIM module. On the line to the temperature sensor, you should identify the following:

- i. Which TWIM hardware module the temperature sensor is connected to. There are 4 TWIM modules in the microcontroller (0-3), and the temperature sensor only connects to one of them.
- ii. The pin numbers for SCL and SDA
- iii. The chip address of the temperature sensor

Part 2 – Implementation

Please implement this in an incremental way. I am not going to dictate the steps you must take, but please don't try to write all of the code and then test it. However, if I were doing this lab, here are the milestones I would be trying to accomplish:

1. Use the 'Print to Putty' section to initialize and test the Virtual Com Port.
2. Write the `initialize_temperature_sensor` function and the `read_temperature` function.
3. Call these functions from main and check the TWIM signals with the Digilent scope.
 - a. Note that the debugger might break the timing of the TWIM signals, so it is not going to be a very effective tool for us in this lab.
4. Check to see whether the signals match your Read Temperature message in your ICD.
5. Check to see whether the temperature value returned by the temperature sensor appears to be valid.
6. Print the two bytes of the temperature to Putty and see if they look valid.
7. Calculate the floating point temperature value that was returned by the temperature sensor and print it to Putty.
8. Now that you have a valid temperature, the hardest part is done. Now begin implementing the steps necessary to meet the requirements of the lab.

Print to Putty

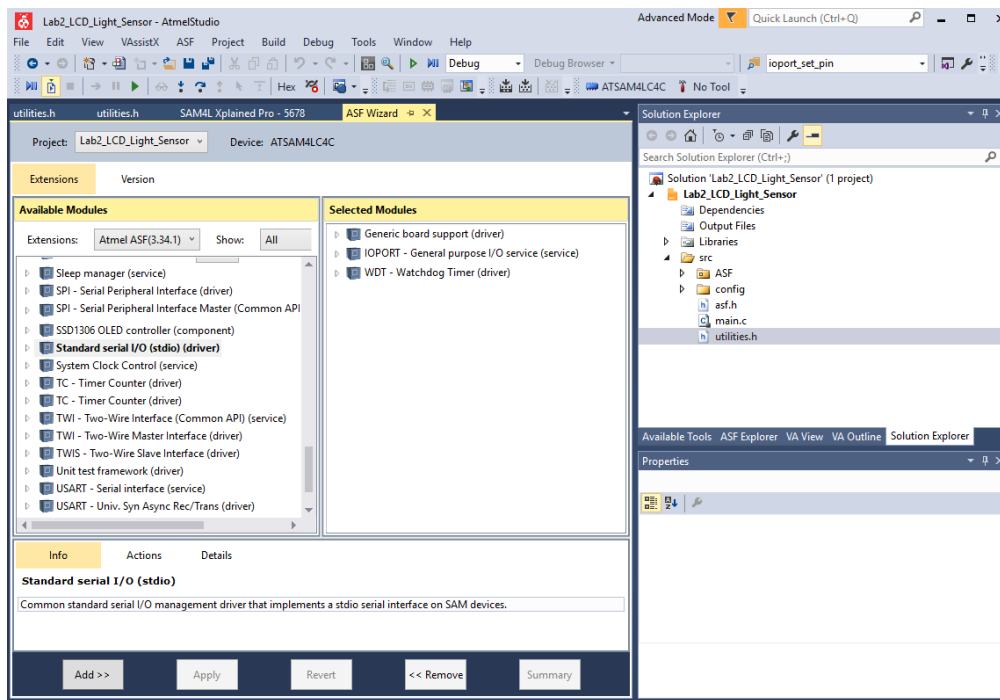
Please do not do this section until after you have watched Lecture 7, which discusses the Virtual Com Port and UART communication protocol.

1. Add a 'configure_console' function to `utilities.h`. This function will set up a specific serial port to send data to the EDBG Virtual Com Port, which will forward the serial data to your computer over the USB cable. You can read or write serial data using Putty. The `configure_console` function should look like this:

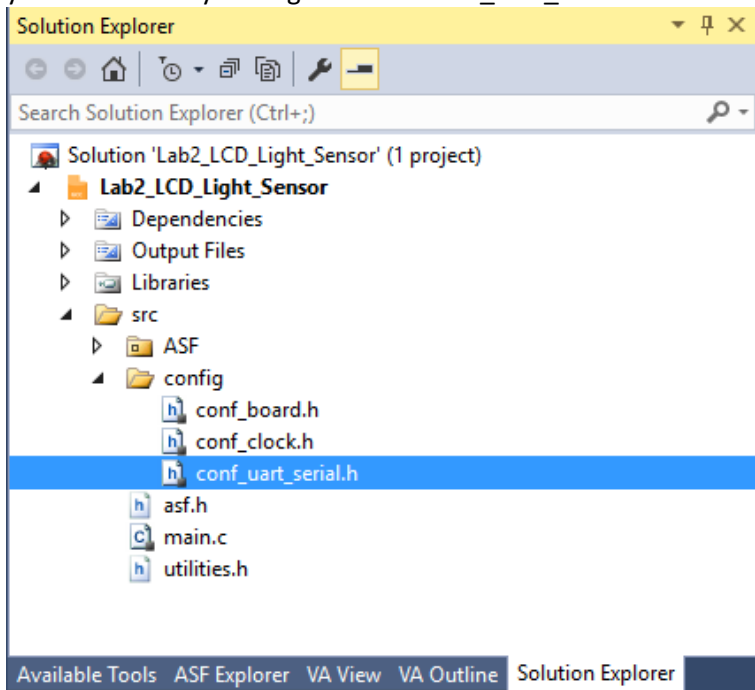
```
static void configure_console(void)
{
    const usart_serial_options_t uart_serial_options =
    {
        .baudrate = CONF_UART_BAUDRATE,
        .charlength = CONF_UART_CHAR_LENGTH,
        .paritytype = CONF_UART_PARITY,
        .stopbits = CONF_UART_STOP_BITS,
    };

    /* Configure console. */
    stdio_serial_init(CONF_UART, &uart_serial_options);
}
```

2. When you add this function, you will see a lot of unknown variables. This is because we do not have the correct ASF module loaded yet. To get the correct ASF module, go to the ASF Wizard and add the Standard Serial I/O driver.



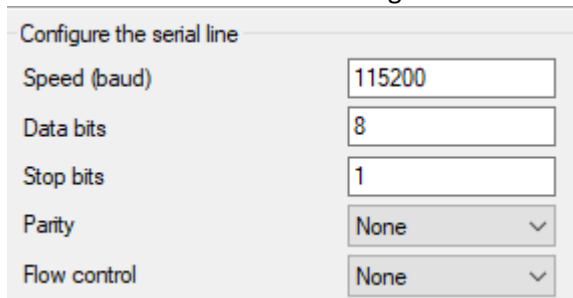
3. Now that you have added the ASF module, you will still see a lot of unknown variables, but we can fix that. When you added the ASF module, a new file was created, `src/config/conf_uart_serial.h`. You need to include that file in your `utilities.h` by adding `#include "conf_uart_serial.h"` at the top.



4. After all of this, you will still see unknown variables. Evaluate `conf_uart_serial.h`. You will see that it includes a number of macros that are commented out. You must define the macros in `conf_uart_serial.h` so that they can

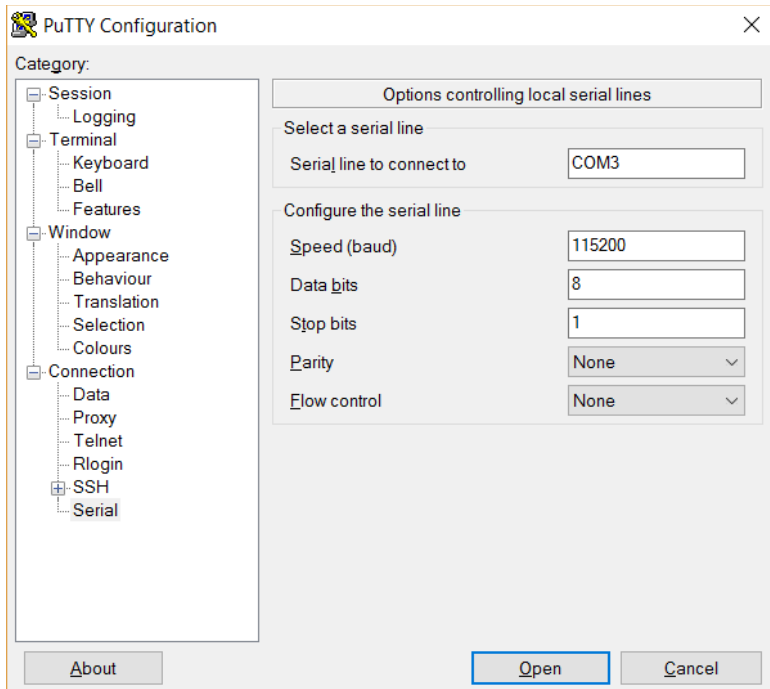
be used in your `configure_console` function. Refer to the `configure_console` function to determine which macros need to be defined.

- a. There are two sources of info that will provide these settings:
 - i. `CONF_UART` can be found with a combination of the SAM4L8 XPlained Pro User Guide and the SAM4L8 XPlained Pro Schematic. Start with the User Guide, reviewing the Virtual COM Port section. The Virtual COM Port allows us to send text to our desktop computer and view it with Putty.
 - ii. The rest of the terms are settings that must match between your program and Putty. Here is a picture of the Putty configuration that will be used to grade the labs....so match your settings to these settings.

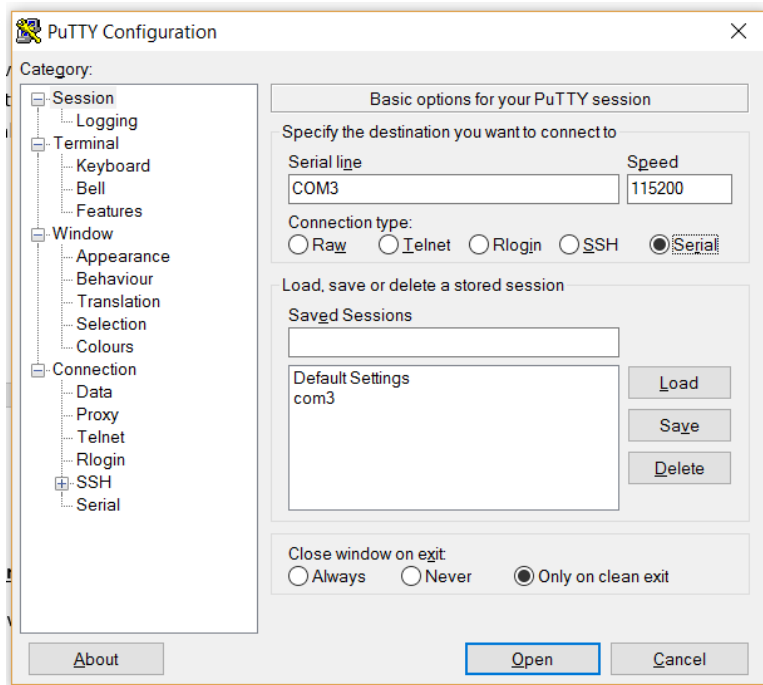


Configure the serial line	
Speed (baud)	115200
Data bits	8
Stop bits	1
Parity	None
Flow control	None

5. At this point, we have a `configure_console` function that should compile. Now, you need to call add 2 function calls in the following order (order matters) right after `board_init()` in `main()`:
 - a. `sysclk_init();` // The serial console relies on the system clock to generate a clock signal
 - b. `configure_console();`
6. Note that in the previous step that we are calling `board_init` (as we will in every lab). Evaluate `board_init` and see if there is another macro that needs to be defined to properly initialize the Virtual Com Port. If so, define this macro in `conf_board.h`. `conf_board.h` is used to define macros that are needed to enable certain modules in the code.
7. Also note that we added a call to `sysclk_init()`. You will find that just about every function that you call will require some configuration in a config file, and the system clock is no different. Please ensure that your clock settings in `conf_clock.h` are the same as the settings specified in Lab A2.
8. Now, after all these steps, you should be able to print to the Virtual Com Port. To verify that your system works, add the following line in `main` after the call to `configure_console`:
 - a. `printf("Initialization Complete\r\n");`
 - b. Note that you cannot call `printf` before you call `configure_console`. If you do, your program will hang at that point and never progress
9. Use Putty to establish a serial session with the Virtual Com Port. Use the settings below, except your serial port will likely not be COM3. Use Windows Device Manager or another means to find out what COM port you are connected to.



Once the settings are correct, select Session, then Serial, then Open.



Submitting the Lab

1. Zip up the Atmel project by following the instructions for 'Submitting an Atmel Project' section in the appendix.

2. Submit the zip file and the Digilent screenshot file as 2 separate files on Canvas.

Lab A5 – FreeRTOS