ECE2049 – Embedded Computing in Engineering Design

Lab 0 – Introduction & Programming Puzzles

In this lab, you will be introduced to the Code Composer Studio (CCS) development environment that we will be using to program and debug our custom MSP430F5529 Launchpad based lab board.
Before you begin, you should install CCS on your computer, see the course website for instructions on how to do this.
This lab is tutorial in nature and does not have a pre-lab. You are expected to sign-off each section on the lab and to answer all the questions highlighted in yellow in your lab report.

Part 1: Getting Started

1. Before you get started, you should watch the Lab 0 Introductory video, which is available on the course website with the lab materials, or on the Echo360 page on Canvas.

2. Plug the USB cable into the board and the computer. If you do not have a USB cable packaged with your board for some reason, please ask the course staff for a replacement.

3. Once the board is connected, start Code Composer Studio. On Windows, you can find it by selecting: **Start** > **All programs** > **Texas Instruments** > **Code Composer Studio**.

4. When CCS starts up, it will ask you to select a workspace as shown in Figure 1. All projects you create will be stored in this path.

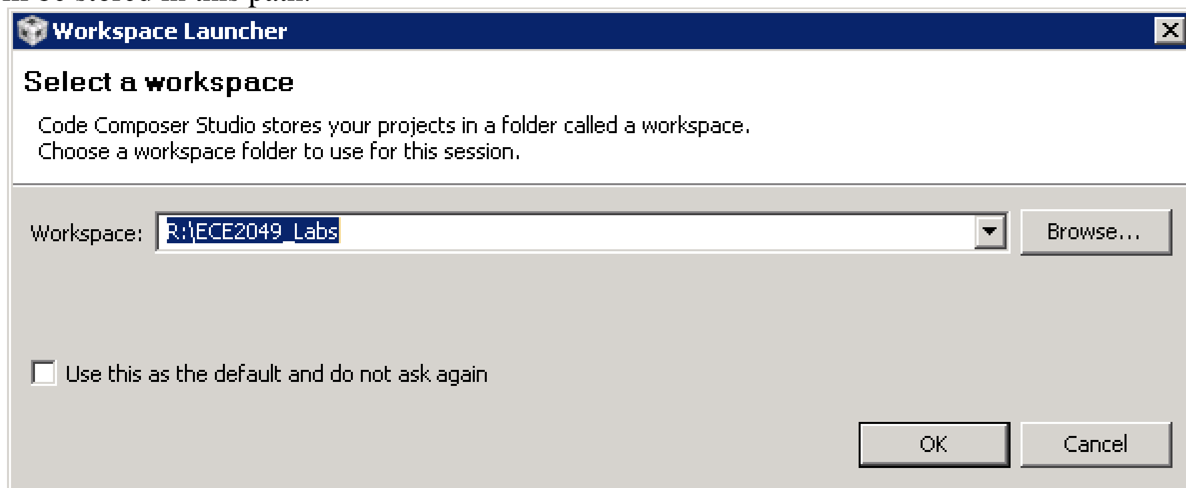


Figure 1: Selecting a Workspace

5. When CCS opens, it may prompt you to select a license file. <u>If this happens, select the **Code Size Limited** license and click **Finish**.</u>

6. When CCS starts for the first time, it may show a window labeled "TI Resource Explorer" that hides all of CCS's features. To remove this, click the "X" on its tab to see the full view of CCS.

7. CCS groups windows into different "perspectives" for different tasks. At startup, CCS will be in the **Edit Perspective**, which is designed for editing and compiling your code. In the Edit perspective, CCS should look like Figure 2.

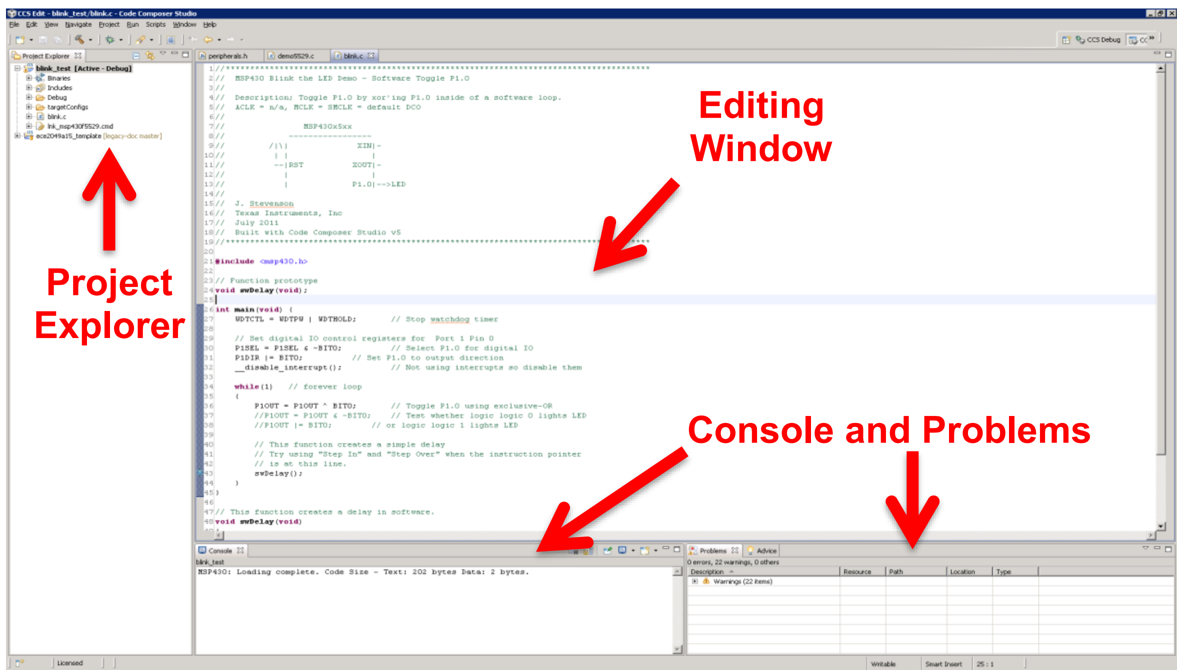


Figure 2: CCS Edit Perspective

In Edit perspective, CCS has the following panes:

- **Project Explorer**: This pane shows all of the projects that exist in the current workspace. A **project** is a set of code and configuration files that make up a single embedded application. (Your workspace likely doesn't contain any projects yet.)
- **Editing Window**: This pane shows all of your open source files in different tabs.
- **Console and Problems Panes**: These two panes show the output produced by the compiler when building your code. The full build output is shown in the Console window; errors and warnings are highlighted in the Problems window.

Using these panes, you can perform most of the actions you need to find files in projects, edit code, and compiler it for the board. More useful features of these panes will be introduced later in this lab.

Part 2: Building your first project – Don't blink!

We will start by building a simple project from a C file. This is a good way to make sure your CCS environment can program a board and demonstrate how the development process works. It will be over quickly, so don't blink!

8. Download the file **blink.c** from the course website. This file contains the source code for a simple MSP430 program, which just blinks an LED.

9. Back in CCS, go to the **Projects** menu at the top of the window and select **New CCS Project**. You should see a window that looks like Figure 3. Enter a project name and set the other fields as shown in the figure. Note that it is critical that you select the model of processor to match the one we use, the MSP430F5529, which tells the compiler how to compile and link the code for our chip. Select **Empty Project** and then click **Finish**.

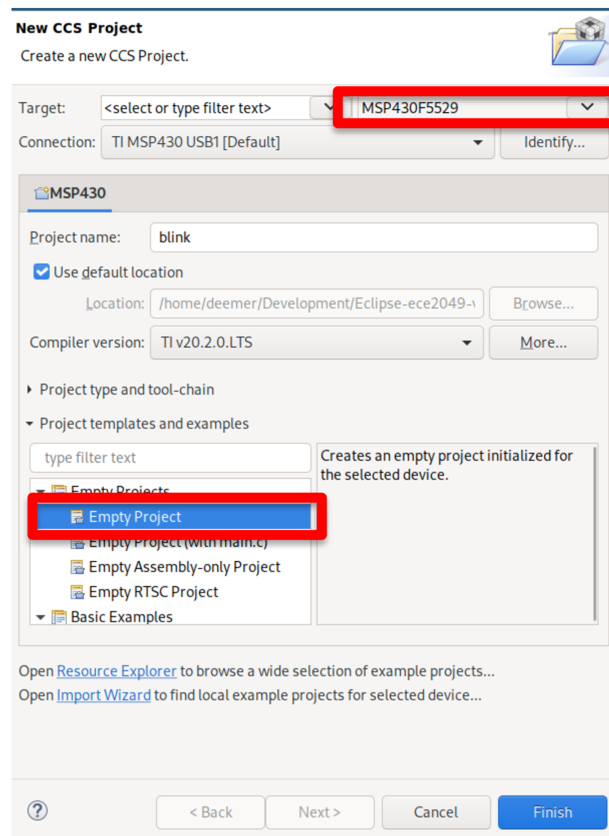


Figure 3: New CCS Project Dialog

Clicking Finish should return you to Edit perspective, where you should see an entry for the project you just created in the Projects Explorer.

10. Click on the project and expand it—you should see a few files that configure the project to use our board. **If a skeleton .c file was created in your project, you should delete it.** Note that clicking on your project sets is to the *Active Project*, as indicated next to the project's name. This is important for some operations we will perform later in the lab.

11. Now we need to add our .c file to the project. Right-click on the name of your project to open the context menu shown in Figure 4. This is a very useful menu—it contains functionality for performing operations specific to your project. To add blink.c, select **Add Files** from the menu and browse to the file you downloaded. When prompted, select "Copy Files" to copy the new file into your project. You should now see blink.c listed in the tree of files for your project. If the file does not open in your editor window, double-click on the file in the Project Explorer and it should open.

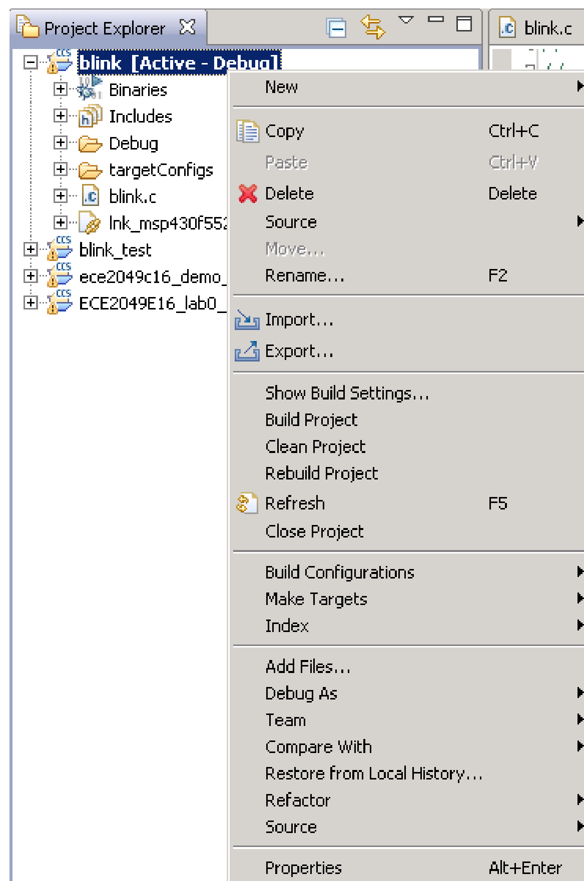
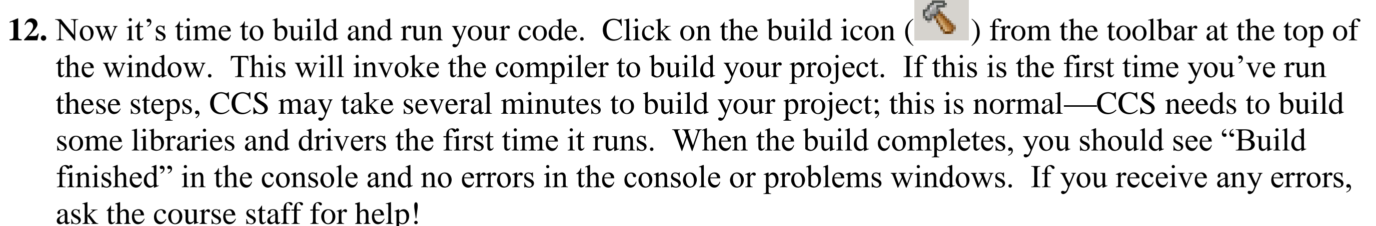
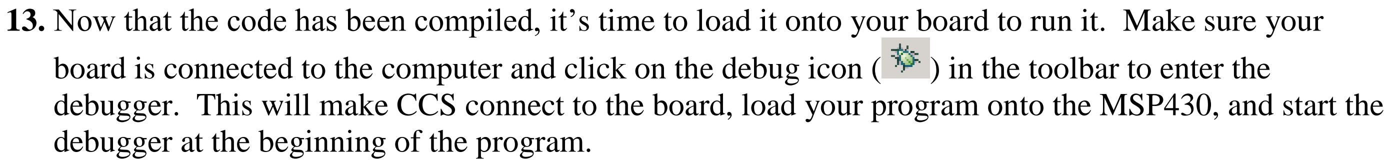


Figure 4: Project configuration menu



12. Now it's time to build and run your code. Click on the build icon ( ) from the toolbar at the top of the window. This will invoke the compiler to build your project. If this is the first time you've run these steps, CCS may take several minutes to build your project; this is normal—CCS needs to build some libraries and drivers the first time it runs. When the build completes, you should see "Build finished" in the console and no errors in the console or problems windows. If you receive any errors, ask the course staff for help!

13. Now that the code has been compiled, it's time to load it onto your board to run it. Make sure your board is connected to the computer and click on the debug icon () in the toolbar to enter the debugger. This will make CCS connect to the board, load your program onto the MSP430, and start the debugger at the beginning of the program.
If this is the first time starting the debugger, this may also take a while. During the process, if you get a pop-up window about with MSP430 Ultra Low Power (ULP) warnings, you can dismiss it.

Also, if you see a message saying a firmware update is required, press "Upgrade" wait for a moment while CCS updates the debug firmware on the MSP430 board.
This process will take about 60 seconds, **do not unplug the board during this step**.

If the process is successful, CCS should switch to *Debug Perspective*, which looks like Figure 5. If you encountered any errors in the process, ask the course staff for help.

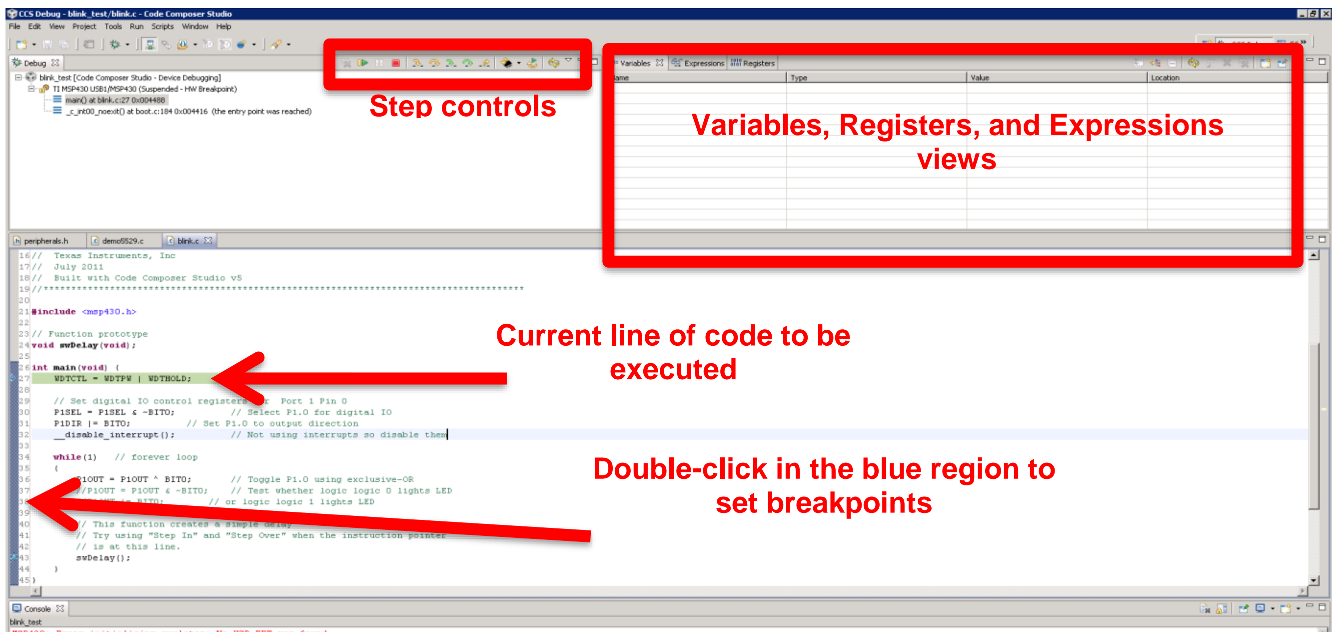
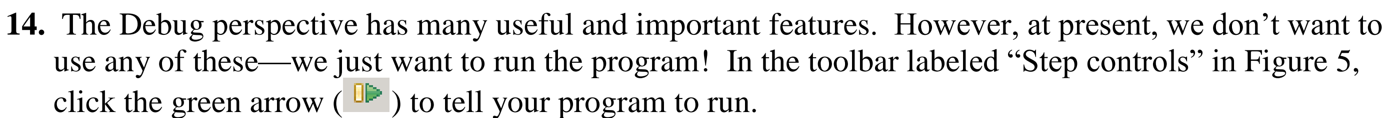


Figure 5: CCS Debug Perspective

The first line of your program should be highlighted in green. The highlighted line indicates that the position of the *program counter*, a pointer to the next line of the program that will execute. In this state, the debugger has paused the execution of your program *before* running this line and is waiting for you to let it continue.

14. The Debug perspective has many useful and important features. However, at present, we don't want to use any of these—we just want to run the program! In the toolbar labeled "Step controls" in Figure 5, click the green arrow () to tell your program to run.

The red LED on the small board should be blinking! Blinking lights are cool. Optionally, ~~take a video to~~ show your mom what you do at school.

Using the Debugger

Before continuing, let's examine some of the step controls more closely, shown in Figure 6.

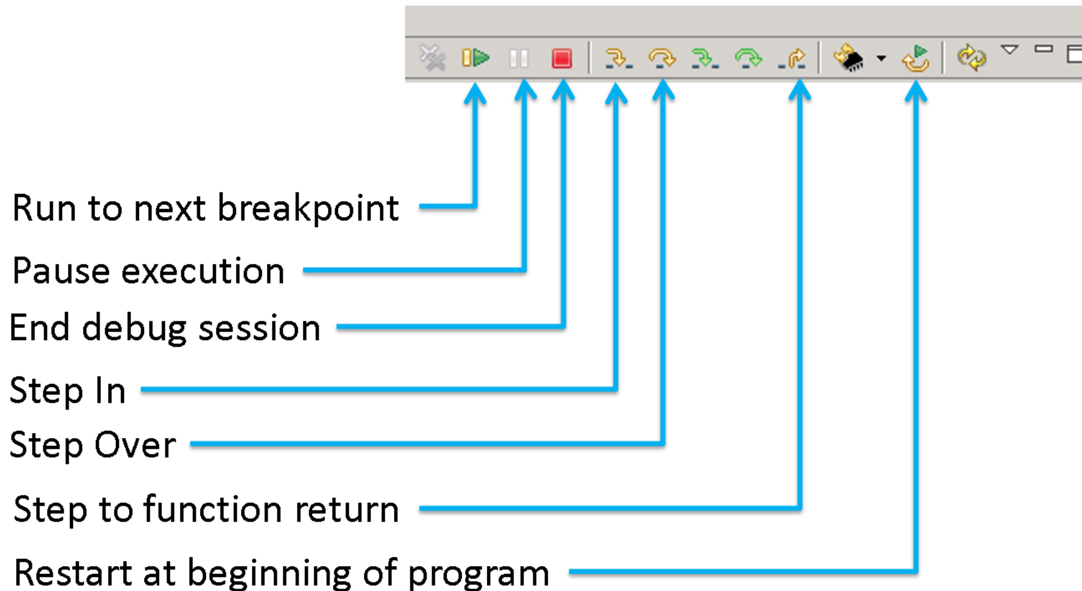


Figure 6: Debugger controls

The debugger controls allow you to control how your program executes. For the remainder of this portion of the lab, you will try out the step controls to see how they work. The most commonly-used step controls include:

- **Run to next breakpoint** (also called "Go"): This control runs your program until it encounters a *breakpoint*, or a marker that indicates that the debugger should stop running the program so you can examine it.
- **Pause execution:** Stops the program at its current position. You can use this button to see what your program is doing at a certain time.
- **Terminate/End debug session**: Stops the debugger and detaches the debugger connection from the MSP430. The MSP430 will continue running your program, but the debugger can no longer control it. Use this button when you are done debugging and want to go back to Edit perspective.
- **Step In/Over/Return**: Runs a single line of your program and halts execution again. You can use these buttons to run your program step-by-step to see how it executes. We will use these buttons later in the lab.
- **Restart**: This control resets the program counter to the beginning of the program, allowing you to start it again as if you had just started debugging. Note that this button does **not** reload any changes you have made to your program since you started debugging, it just restarts the current program from the beginning of `main()`.

15. Now that you are familiar with the step controls, hit the **Pause** button and **Restart** your program at the beginning. Use the **Step Over** button to step through the setup portion of the code, then step through the while loop a few times. You should see the LED change state for each iteration of the loop.

16. Often, it is tedious to manually step through your all of code to a point you want to observe. As an alternative, you can set a **breakpoint**, which denotes a point in the code where the debugger should pause the program. To set a breakpoint on a specific line of code, double-click the blue margin next to the line numbers—you should see a blue circle appear where you clicked, as in Figure 7.
To test this, set a breakpoint on the call to the `swDelay` function. Then, click the **Run** button a few times—each time, your program will run for a moment and pause when the line is reached again in the next iteration of the loop. Like in the last example, you should see the LED change state with each iteration.

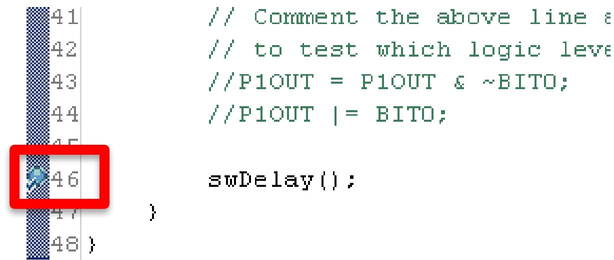


Figure 7: An example breakpoint

We can examine how the LED changes state in more detail. As we will discuss later in class, the LED is connected to a *Digital I/O* port, a hardware peripheral that can set a specific pin on the microcontroller to a logic 0 or logic 1. On the Launchpad, the red LED is connected to Digital I/O port 1, pin 0 (P1.0), which itself is wired to a physical pin on the chip and then connected to the LED. The LED will turn on or off depending on whether a logic 1 (3.3V) or logic 0 (0V) is present on that pin.

In the code, the output value of P1.0 is controlled by bit 0 of the register P1OUT; setting this bit to 0 will output a logic 0 on the pin, while setting it to a 1 will output a logic 1. By changing the value of this single bit in P1OUT, blink.c can turn the LED on and off.

17. We can observe the output register P1OUT changing state in the debugger. In the top-right corner of the Debug perspective (see Figure 5), we can view the state of variables and registers in our program. In blink.c, there are no variables, so the variables pane is pretty boring. Instead, switch to the **Registers** tab and find the register P1OUT listed under the header "Port1_2", as shown in X. You can see the value of the entire register next to P1OUT, or expand it to view the values of each bit (eg. P1OUT5 shows the value of bit 5 of P1OUT).
If your breakpoint from the previous step is still set, click the **Go** button again to let the program run through another iteration of the loop. When the program stops, you should see the value of P1OUT change and the state of the LED change accordingly.

(x)= Variables | Expressions | Registers

Name	Value	Description
Port_1_2		
P1IN	0xBE	Port 1 Input [Memory Mapped]
P1OUT	0x01	Port 1 Output [Memory Mapped]
P1OUT7	0	P1OUT7
P1OUT6	0	P1OUT6
P1OUT5	0	P1OUT5
P1OUT4	0	P1OUT4
P1OUT3	0	P1OUT3
P1OUT2	0	P1OUT2
P1OUT1	0	P1OUT1
P1OUT0	1	P1OUT0
P1DIR	0x00	Port 1 Direction [Memory Mapped]

Figure 8: Viewing register P1OUT
(Here, P1.0 has a value of 1)

18. We will now examine how the code actually changes the value of the LED. In blink.c, P1.0 is toggled by XOR'ing (^) the current value of the output register P1OUT with the constant BIT0, as shown in the line below:

```
P1OUT = P1OUT ^ BIT0;
```

The name BIT0 is a constant defined in `msp430.h`. To see its value, hold down the **Ctrl** key and click on the name BIT0 in blink.c in CCS. This should open up the relevant portion of msp430.h to show you that BIT0 is defined as 0x0001 (you can say no to any pop-up messages about scalability features). Congratulations, you just used CCS to find the definition of a variable—this is one of its most useful features!

Back to the point: we now know that this line of code toggles the LED by XOR'ing the current value of the output register with 0x0001.

How does this toggle the state of the LED? Explain how this works in your report.

(Another way to ask this question: How does this operation change the value of P1OUT?)

(continued on the next page)

Part 3: Importing and Building the Demo Project

We are now ready to build the demo project, which is a much larger project that contains libraries for interacting with all of the peripherals on our development board. Rather than making you set up this project yourself, the course website provides an archive of the project that you just need to import into CCS. In the remainder of the lab, you will import this project and play around with the debugger to answer some questions about it.

Importing the Demo Project

19. Download the **Lab 0 template** from the course website (ece2049e20_demo_template.zip). Back in CCS, if you still have a debug session running, return to Edit perspective by clicking the **Stop** button in the step controls. Then, in the **Project** menu, select **Import Existing CCS Eclipse Project**, which will open the dialog shown in Figure 9.

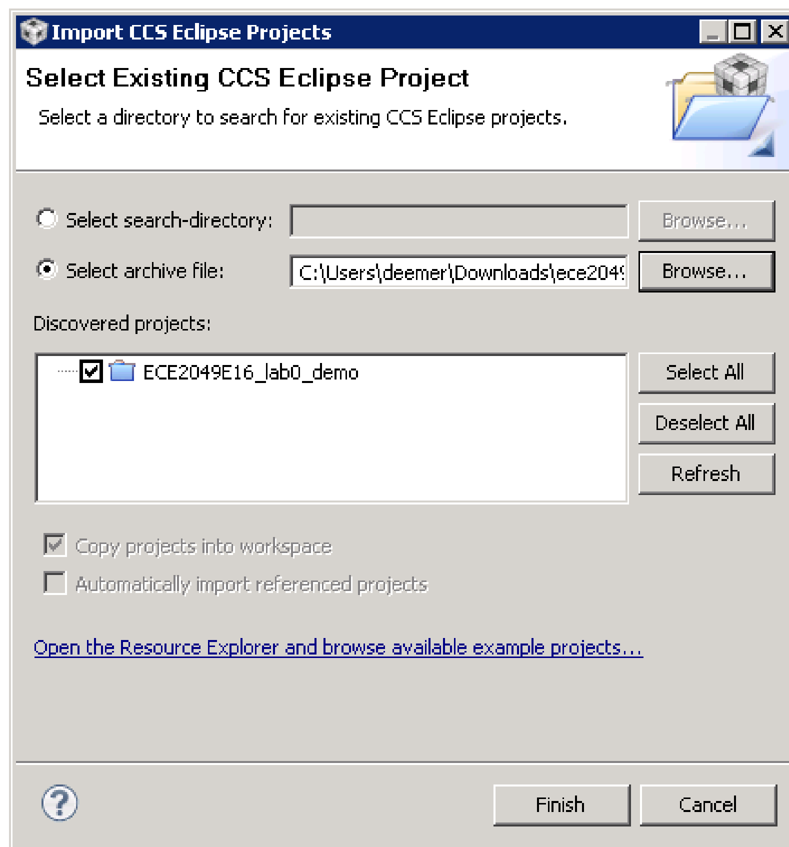


Figure 9: Importing the demo project

20. As shown in Figure 9, choose "Select archive file" and browse to the zip file for the demo project you just downloaded. CCS should find the project in the archive file and display it in the pane. Click **Finish** to import the project.
    If the import process fails at the last second with an error about the project metadata, repeat the process and it should succeed this time. If you have issues importing the project, please ask the course staff for help.

21. Once the demo project has been imported, it should appear in your Projects Explorer. Click on the demo project to make it the active project, and then build it. You should see some warnings about unused variables—these are safe to ignore for now. Enter the debugger and run the project. You should see some text display on the LCD, and you can play with the buttons to make things happen.

Note: If your LCD module is the older 96x96 display (part number 430BOOST-SHARP96), you will need to modify the demo template slightly before continuing. See the section Using the Sharp 96x96 Display for details.

Exploring the Demo Project

22. When you are done playing with the demo project, stop the debugger and examine main.c in the editor window.
You will notice that this project is quite a bit more complex than blink.c—this is because it includes libraries required to use the LCD screen, and some library functions that utilize the buttons and LEDs. Since all of our labs will use the LCD in some way, we will start our labs by importing a variant of this project. To implement the labs, you will use the graphics library functions to use the LCD and expand on some of the other library functions to add new capabilities.

23. Starting with example code and modifying it to perform desired tasks is a typical strategy in embedded systems. Experiment with the demo project to answer the following questions. You should include the answer questions in the discussion section of your report:

1. Play with the LCD write commands (`Graphics_StringDrawCentered`) to move the text to different positions on the screen. **Explain how to control the position of the text on the screen in your report.**

2. Use the Ctrl+Click method described in step 17 of the lab to find the definition of any function in `main()`. This is a very useful feature of CCS—you will use it a lot in your labs!

3. Step through the code until you get to the assignment for variables `a_flt`, `X`, and `test`. **Answer the questions asked in the code in your report by using the debugger to find the new value for test and explain why this result is correct.**
   Note that you can change the way CCS displays a variable in the variables window by right-clicking on it and selecting "Number Format", as shown in Figure 10.

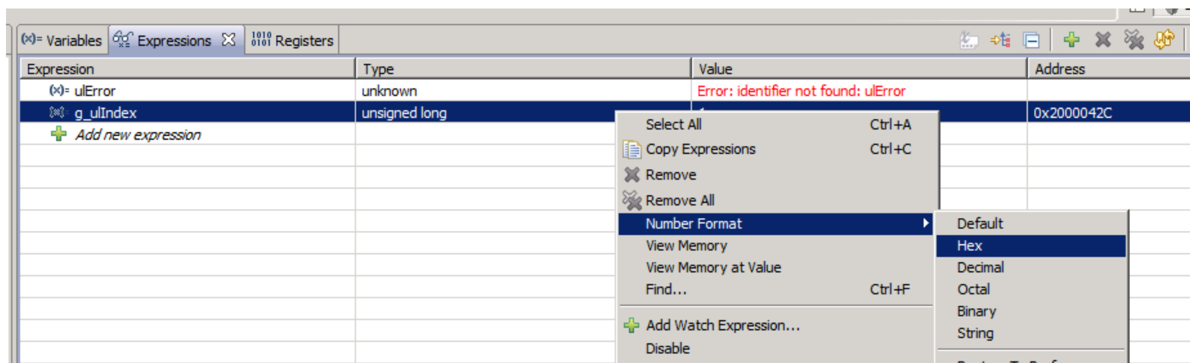


Figure 10: Changing number format in the variables window

(continued on the next page)

For reference, the questions asked in the code regarding these variables are as follows:

```
// What value stored in myGrade (i.e. what's the ASCII code for "A")?
// What is the new value of test? Explain?
```

4. **How does what you press on the launchpad buttons change the value of `button_state`? (In other words, what does `readLaunchpadButtons()` return?)**

5. The `while(1)` loop contains the line:
   `buffer[1] = button_state + '0'`
   What does the "+ '0'" do? Why is it necessary?

Once you have modified the demo project and *thought about* these questions, you can go onto the next section. **You don't need to have answers to all the questions yet**—the next section will be helpful for figuring them out. If you have any questions about the lab, feel free to ask the course staff before you write your report.

24. Continue on to the next section: programming puzzles!

Part 3: Programming Puzzles

Next, you will work with a series of "puzzles," or tiny functions you will write help build your embedded programming skills. These puzzles are designed based on the kind of operations you are likely to perform when developing for embedded systems, and deal with some of the number representations we are discussing in class.

Open the file `puzzles.h`, located in your demo project. You will see a series of empty functions, or *stubs*, with descriptions. For each puzzle, you will complete the function based on the description in this file and based on the table at the end of this section.
Find the function `test_all_puzzles` in puzzle_tests.c, which is called from `main()`. This function contains many commented functions starting with "`test_`". Each test function has a few test cases for a single puzzle. When you write a solution for a puzzle, you can have the board run these tests to check your work. If a test fails, the debugger will pause after the test is run, allowing you to use the debugger to gather more information.

For some puzzles, there are **restrictions** on the operators you can use. This is because it is important to learn how to perform certain operations efficiently (ie, without using loops or conditionals). The table on the next pages lists each puzzle and their restrictions. More information about each puzzle can be found in puzzles.c.

TL;DR: How this works

1. Look at the table of puzzles on the next page
2. Pick a puzzle, **write** your implementation in puzzles.c. Note the restrictions for each puzzle.
3. **Uncomment** the respective tests for the puzzle in `test_all_puzzles()`
4. **Build and run your program.** If a test fails, the debugger will halt to show you which test did not pass.

   If all the tests pass, the LEDs on the board will flash, and your program will continue running normally. **Note that this is the default state, because no tests will run until you uncomment them!**
5. **When you are done, pick <u>three</u> puzzles you found interesting or challenging and describe how you implemented them in your report. For each puzzle you describe, describe how an example input would be evaluated.**

Table of Puzzles

Puzzles in each section are assigned a "Group." Puzzles in the same section and same group require similar concepts: once you have completed one puzzle in a given group, the rest may be more straightforward. Some puzzles are marked "Bonus," which count for extra credit.

1. Warmup

These puzzles are designed to familiarize you with the puzzle framework.

Puzzle	Brief Description (see puzzles.c for more info)	Group
`int linear(x)`	Return `2*x+4`	--
`int gt20(int a, int b)`	Return 1 if both a and b are greater than 20	--

2. Logical Operators

The following puzzles deal with logical operators and conditionals, as well as ASCII characters.
Restrictions: For puzzles in this section, you are limited to using standard C operators, conditionals (if/else) statements, assignments, and casting—**loops are not permitted**.

Puzzle	Description (see puzzles.c for more info)	Group
`int myMax(int a, int b)`	Return the larger of two integers	A
`int clamp(int x, int lo, int hi)`	"Clamp" x within upper and lower bounds lo and hi	A
`int isLetter(char x)`	Return true if x is an ASCII letter (A-Z, or a-z)	B
`int letterShift(char x, int s)`	Shift letter x by s letters in the alphabet	B

3. Manipulating Bits and Bytes

These puzzles are designed to give you practice with bitwise operators and learn some fundamental operations we will use throughout the course.

Restrictions: for puzzles in this section, you are limited to using only the following operators:
`! ~ | & ^ + << >>`
You are also limited to writing *straight-line code*—**that is, no loops or conditionals (if statements)**.
Use of casts, as well as the assignment statement (=), are permitted.

Puzzle	Brief Description (see puzzles.c for more info)	Group
`int setBit(x, b)`	Return x with bit b set to 1	A
`int clearBit(x, b)`	Return x with bit b set to 0 ("clearing" the bit)	A
`int divpwr2(int x, int n)`	Return $x/(2^n)$	A
`char getByte(long x, n)`	Return byte n from x (eg, if n is 0, return bits 0-7, if n is 1, return bits 15-8, etc.)	B
`long swapBytes(long x)`	Swap byte order of x	B
`int bitCount(char x)`	Count the number of 1's in x	Bonus

4. Two's Complement

The following puzzles deal with two's complement representations.
Restrictions: Same as previous section.

Puzzle	Description (see puzzles.c for more info)	Group
`int negate(int x)`	Return –x, without using -	A
`int isPositive(int x)`	Return true if x >= 0, without using >=	A
`int bang(int x)`	Compute !x without using !	Bonus

5. Loops and Arrays

The following puzzles deal with logical operators and conditionals, as well as ASCII characters.

Restrictions: For these functions, you can use loops and arrays. Essentially, you have no restrictions—except that you may not call any library functions.

Puzzle	Description (see puzzles.c for more info)	Group
`char average(char *x, char n)`	Return the average of an array of n elements	A
`int over21(char *vals, char num)`	Return 1 if all values in array are >= 21	A
`int strLength(char *s)`	Compute the length of a null-terminated string	B
`void caesar(char *s)`	Shift all letters in a string by 3	B

6. Bonus Problems: Floating point and more

Want even more puzzles? **These puzzles count for extra credit.**
Restrictions: See the description of each puzzle.

Puzzle	Description (see puzzles.c for more info)	Group
`int ilog2(int x)`	Compute floor(log2(x))	Bonus
`float float_neg(f)`	Compute –f, without using floating-point operations	Bonus
`float i2f(x)`	Return x encoded in single-precision floating-point format	Bonus

If you have questions on the mechanics of any puzzle, or on how to implement it, please consult the course staff: we are happy to help you! <u>This part of the lab is fairly new, we are working to gauge the appropriate difficulty level for this assignment—please do not feel discouraged if you get stuck.</u>

Finally, you are welcome to collaborate with your peers on this assignment, but the solutions your lab team writes down must be your own. There are also many online resources on how to do similar puzzles—while are you welcome to read online, *do not copy from these sources, as the solutions you develop must be your own work*. Be prepared to answer questions about how you solved a puzzle!

When you are done, pick three puzzles that you found challenging and write a description how they work in your report. For each puzzle you discuss, describe how an example input would be evaluated.

Writing your Report

Since this lab was mainly a tutorial, the report does not need to be substantial—however, we are asking you to write one as practice for future reports. Your lab should be written in a professional style. It should be an electronically-prepared technical document like what you would submit to a fellow engineer or your boss.

Only one report is required per lab team. The report should include:

- **Introduction** (1-2 paragraphs max): Succinctly state the objectives of the lab and give an overview of what you accomplished.
- **Discussion and results**: Discuss what you did in each part of the lab and how you solved any problems. **For this lab, you need only include two things:**
    1. Your answers to any questions asked in this document
    2. A discussion of how you implemented three of the puzzles, with an example for how each should be evaluated
- **Summary and Conclusion** (1-2 paragraphs max): Summarize what you accomplished in the lab and what you learned. This should be a “bookend” to the introduction.
- **Appendices**: You should not need any in this lab. **DO NOT PASTE YOUR CODE INTO THE END OF THE LAB REPORT!** Instead, you will submit your code as a separate archive file.

Lab reports are important. In industry, the FIRST view of YOUR work by *anybody* other than your immediate supervisor will see will probably be in WRITING!

Learning to be an effective communicator of technical information is probably THE MOST IMPORTANT job skill you can have.

Submitting your Work

When you are done with your report, you will submit it and your code on Canvas for grading. In order to receive a grade, you must submit **both** your code and your report online.

Signoffs/Demo: In addition to your code and report submission, we may require an additional demonstration of your work to replace the in-person lab signoff. Details on how this will work will be available soon!

To submit your code for grading, you will need to create a zip file of your CCS project so that the course staff can build it. You can also use this method to create a complete backup copy of your project (perhaps to send to your partner or save for later). To do this:

1. Right click on your project and select "**Rename...**"
2. If you are submitting your project, enter a name in the following format: **ece2049e20_lab0_username**, where “username” is your WPI username.
3. Click **OK** and wait for CCS to rename your project.
4. Right click on your project again and select "**Export...**” then select “General” and "Archive file" from the list and click **Next**.

5. In the next window, you should see the project you want to export selected in the left pane and all of the files in your project selected in the right pane. Select all. You should not need to change which files are selected.
6. Click the "**Browse**" button, find a location to save the archive and type in a file name using the EXACT SAME NAME used in Step (2).
7. Click "Finish". CCS should now create a zip file in the directory you specified.
8. Go to the Assignments page on the class Canvas website. Click on the assignment for Lab 0 and attach the archive file of your project that you just created and your report. When you are ready, hit the Submit button. Only one code and report submission is required per team.

Using the Sharp 96x96 Display

Our lab template supports using one of two LCD displays, depending on the hardware you have available. If you ordered your components for E term 2020 or later, you likely have the Sharp 128x128 LCD display (part number BOOSTXL-SHARP128). The lab template is configured to use this display by default—if you have it, you do not need to do anything.

If you are using the older 96x96 LCD display module (part number 430BOOST-SHARP96), you will need to make one change to the lab template before you can use your display:

1. Open the file peripherals.h. Near the top of the file, find the section entitled “Display module selection.”

2. This section contains two compiler directives to select one of the two displays. To disable the 128x128 display and enable the 96x96 display, comment out the directive `DISPLAY_SHARP128x128` and uncomment `DISPLAY_SHARP96x96`. The result should look like this:

```
// ****** DISPLAY MODULE SELECTION ******
// Uncomment ONE of these directives to select the display
// you are using:
#define DISPLAY_SHARP96x96
//#define DISPLAY_SHARP128x128
```

3. Safe the file and rebuild the project. If you haven’t built the program as part of the lab yet, just continue following the instructions. When you load your code onto the board, you should see text correctly displayed on the screen. If you instead see a bunch of horizontal lines, or no text at all, check that your `peripherals.h` matches the example above, and then feel free to ask the course staff for help.