

3. In-Lab

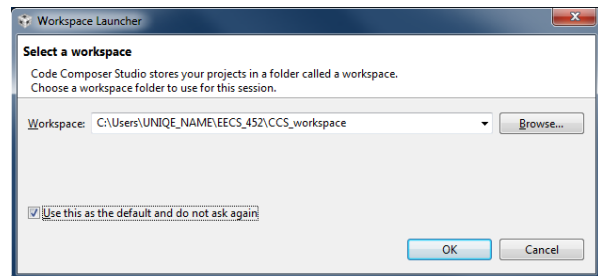
This lab has four parts. Also notice there is a “When things go wrong” section at the end.

- The first part is getting familiar with our tool (Code Composer Studio version 5) and learning how to set up a project from scratch using it. This part is a bit frustrating but you shouldn't need to do this very often (we'll generally just start a new project by copying a correctly set-up old one, but sometimes you'll need to do this from scratch).
- The second part will provide an introduction to using MMIO on the C5515. You'll insert the code you had in the pre-lab, fill in the blanks, and get the lights blinking.
- The third part will give you experience with more complex MMIO as you work with the LCD on the board.
- In the final part you will use the LCD to display sine waves you digitally create as well as signals from the function generator.

Part 1: Getting started with Code Composer Studio

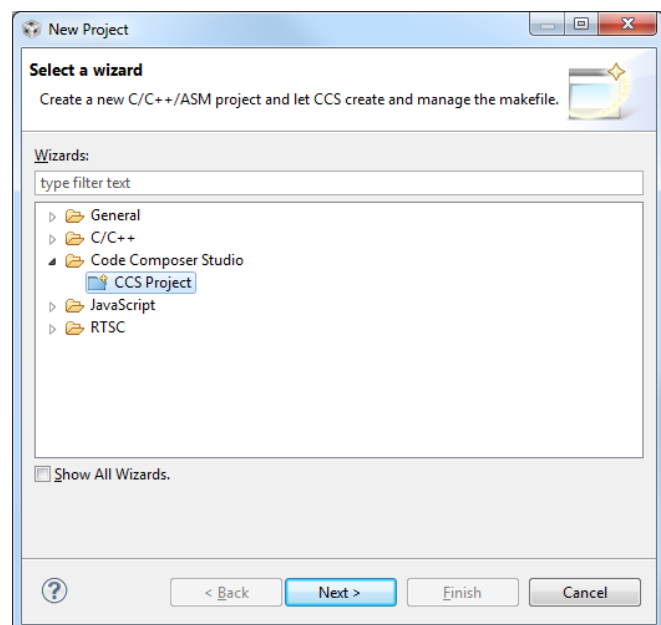


To start Code Composer Studio (CCS) version 5 double click on the grey Rubik's cube icon on your desktop. Because this is the first time you are opening CCS, you will be prompted to choose a workspace. It is important that you chose a path that CCS sees as “local” to the machine. CCS will not follow symbolic links at compile time. If on a CAEN computer, place your workspace in the C:\ drive. Put your workspace somewhere easy to remember. For example, C:\Users\[username]\EECS_452\CCS_workspace. Check the box so CCS doesn't bother you about it again. Click “OK.” CCS will create the directory for you if it doesn't already exist.



Next, add the support files you will need for the rest of the semester. On the course website download C5515_Support_Files.zip. Unzip this directory and place it in the workspace you just created.

Now click on File->New-> Project. You should get the image to the right. Under 'Code Composer Studio' select 'CCS Project' and click 'Next.'



In the next window, you should get something similar to the figure to the right. We will name this project **Blinky**. Double check the Location of your project is under your account. You should ALWAYS check your workspace before working on any project. You can do a Switch Workspace under “File” in CCS.

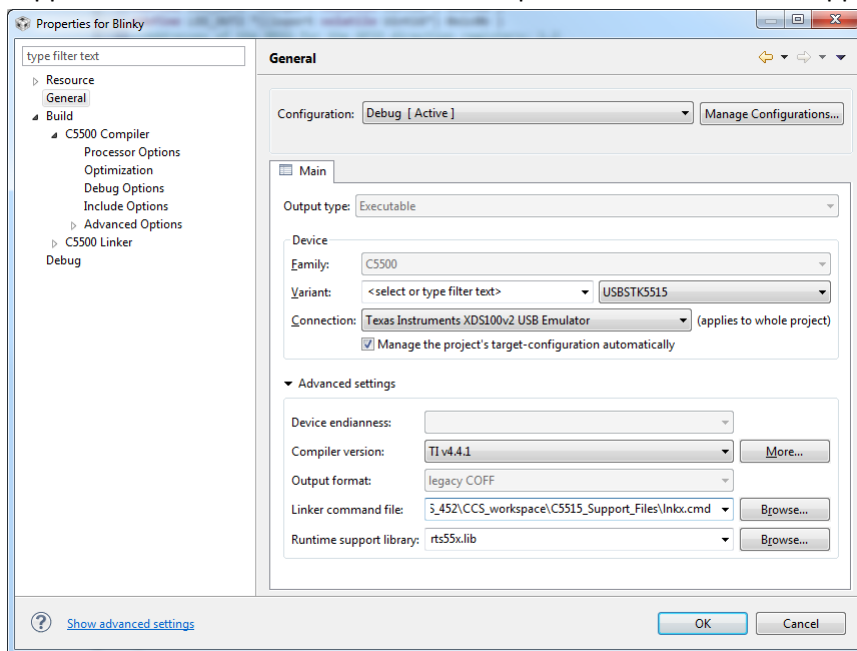
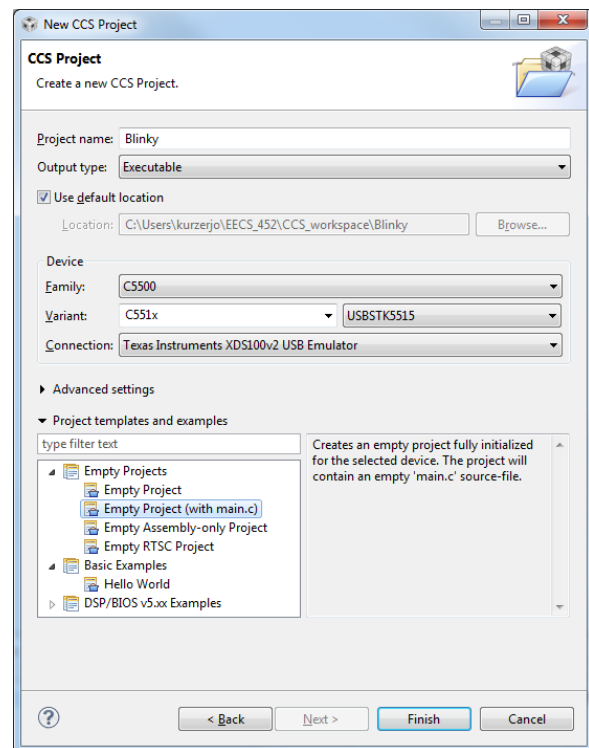
In the device section, set the C5500 family, and the USBSTK5515 variant. For ‘Connection’, chose the ‘Texas Instruments XDS100v2 USB Emulator.’ Finally chose to create an ‘Empty Project (with main.c)’ and click ‘Finish.’

You should see that your workspace is populated with a project named Blinky in the ‘Project Explorer’ window. You may need to close the ‘TI Resource Explorer’ to see this.

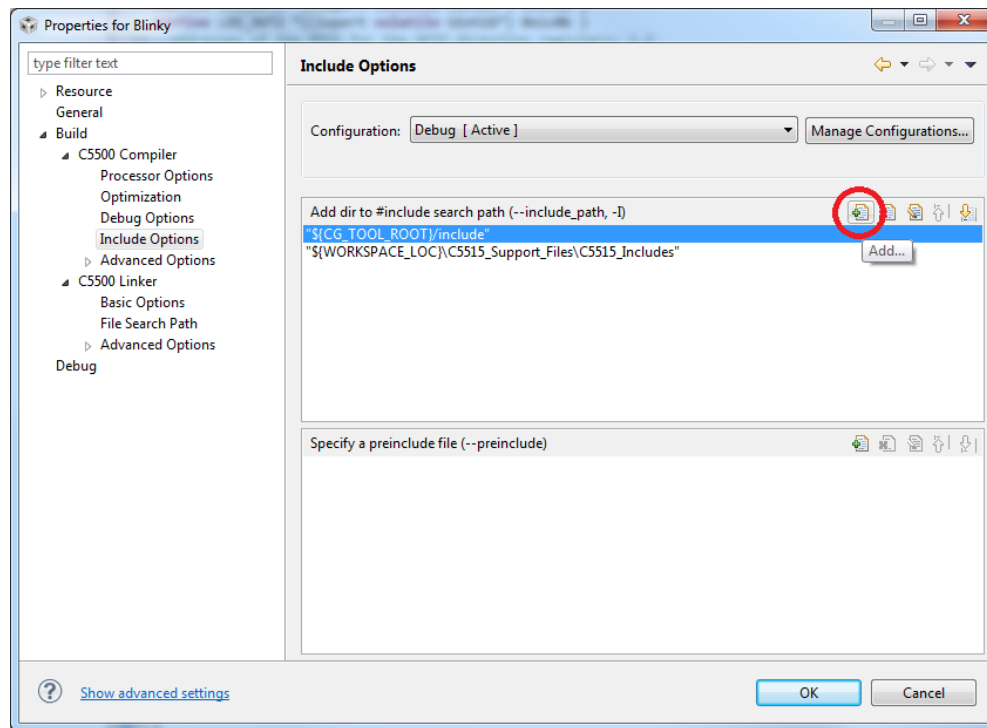
Double click on main.c under Blinky, and you will see an empty main function that the project wizard has created for you. Copy the code from the pre-lab (complete with blanks) into main.c. Replace the blanks as per your answers to the pre-lab. The pre-lab code can be found on the course website inside the Lab1_Files directory named main.c. Save this file with “File->Save” or “CTRL-S” on the keyboard.

Now we have the code, but we still have to tell the compiler and linker where to look for source files and libraries that we need. Right click on the project (“Blinky”) and select “Properties”. The window below will appear. Look under the “General” settings.

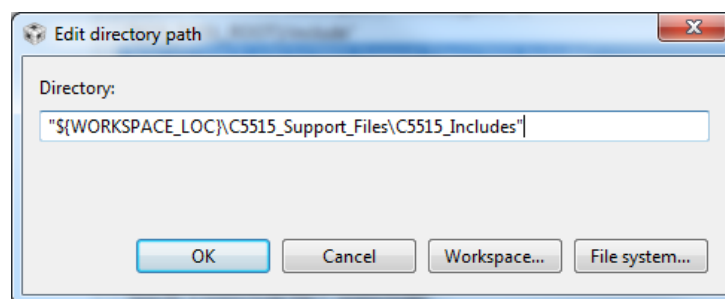
Under “Linker Command File” click on “Browse...” and navigate to your workspace. Inside the C5515 support files select “lnkx.cmd” and click “Open.” Under “Runtime Support Library” select “rts55x.lib.”



Next go to “Build” → “C5500 Compiler” → “Include Options” and you should see the following window:



Select the “Add” icon in the upper right of the window, (circled red in the above image). CCS will ask you for a directory. In the field enter “\$(WORKSPACE_LOC)\C5515_Support_Files\C5515_Includes” **INCLUDING QUOTES**.



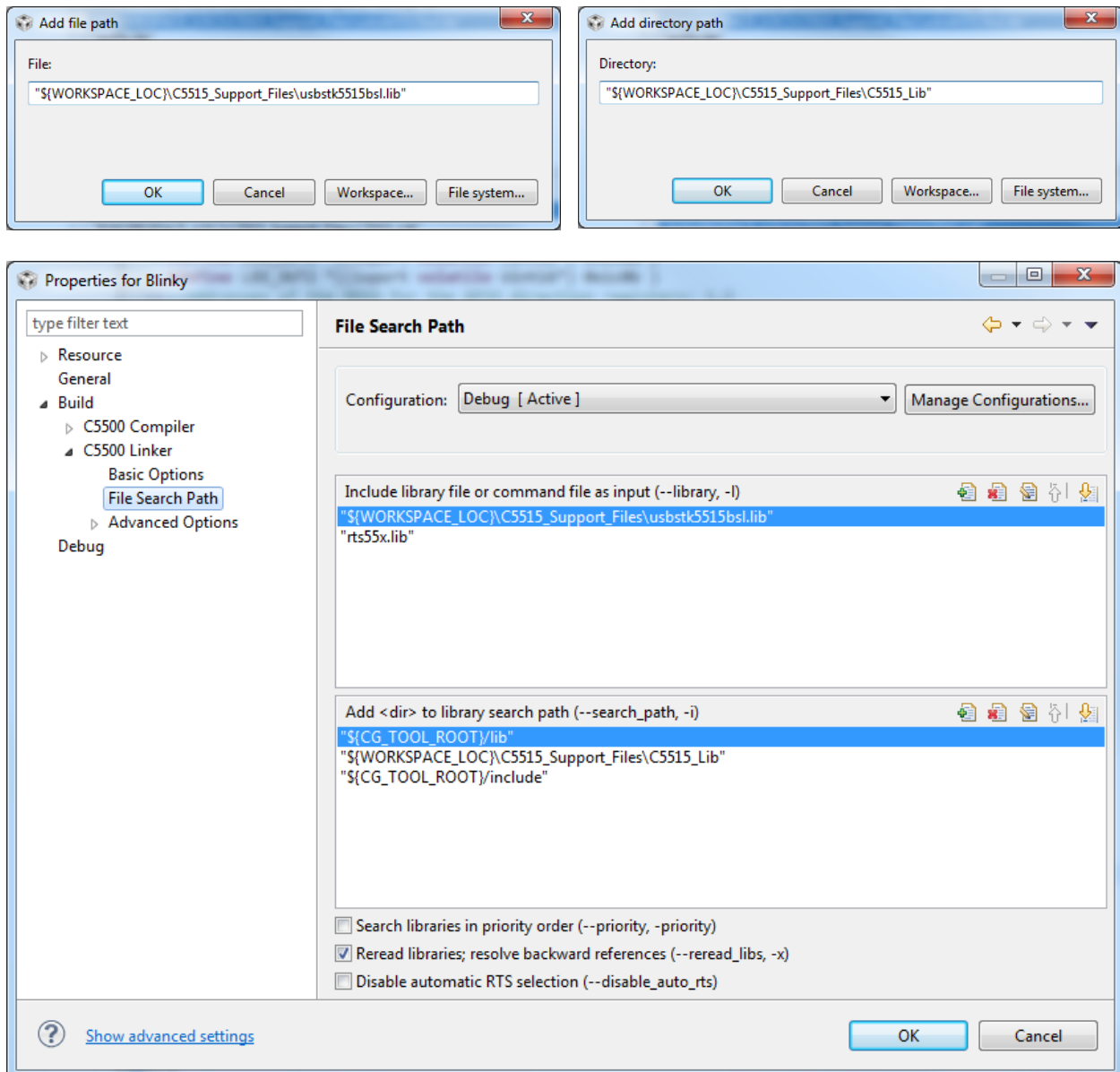
Note: \$(WORKSPACE_LOC) is a path variable that is specific to your CCS setup. You can see this and other defined path variables in the properties window. “Resource” → “Linked Resources” in the “Path Variables” tab.”

It is worth noting that you can also have the compiler search your projects for files that you have made. To do this click on the add path icon again. But this time you’d select “Workspace...” to get the window seen below. **Because we aren’t using our own header file at this time, you will skip this step, but you will to do it in the future if you have any of your own header files.**

Now that we have told the compiler where to find our header files, we still need to tell it where to find the code to resolve the symbols declared in those header files. To do this, go to “Build” → “C5500 Linker” → “File Search Path.”

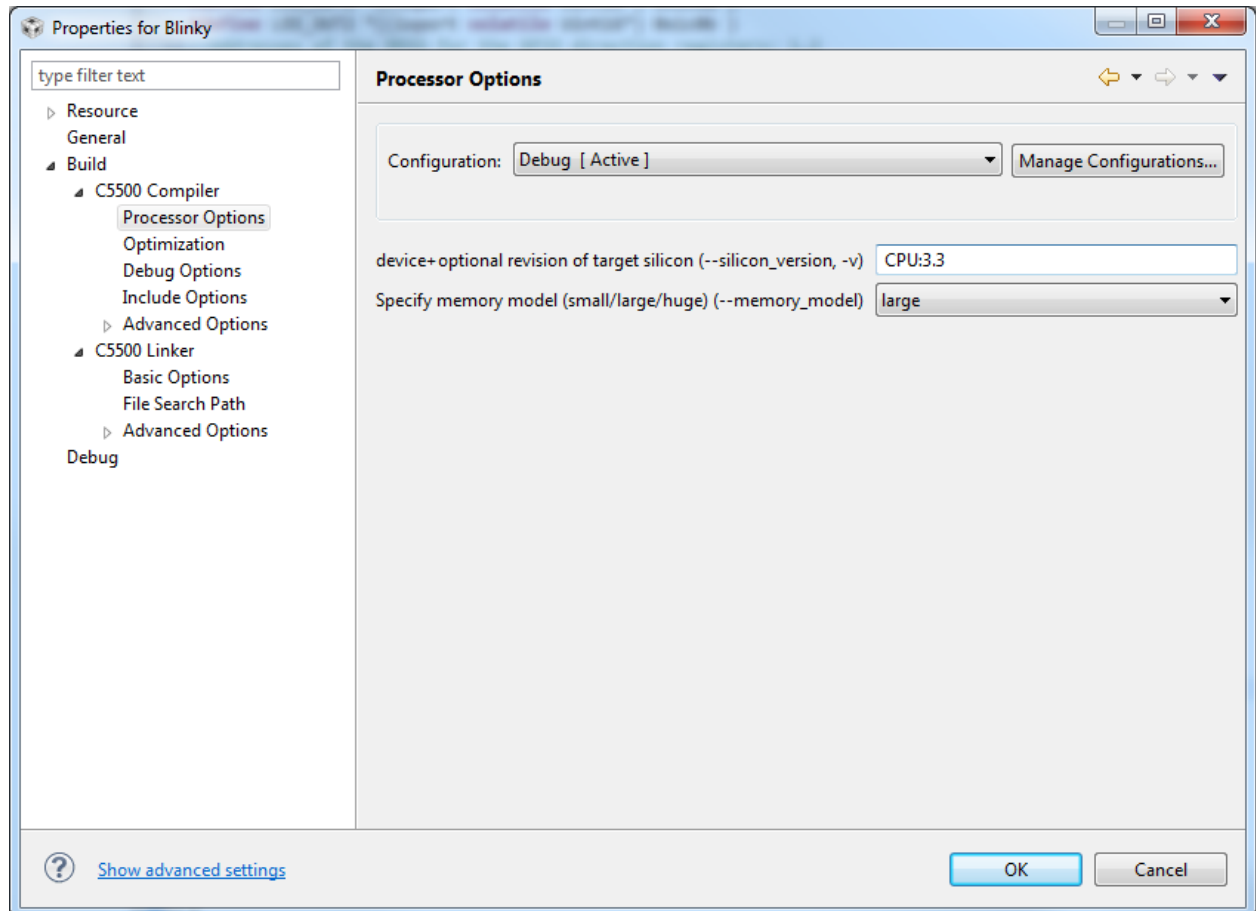
In the “Include library file or command” section, click the “add” button as before. In the “Add file path” window that pops up, enter “\${WORKSPACE_LOC}\C5515_Support_Files\usbstk5515bsl.lib” (again, include quotes.)

In the “Add <dir> to library search path” section, click the “add” button as before. In the “Add directory path” window that pops up, enter “\${WORKSPACE_LOC}\C5515_Support_Files\C5515_Lib” (again, include quotes.)



We still need to tell the compiler which silicon version and memory model we will be running. To do this, select the “Build” → “C5500 Compiler” → “Processor Options”.

Replace the 5515 in the revision of target silicon with “CPU:3.3” (which tells the compiler which revision of the processor we are using). Make sure the memory model is specified to large. Once you’ve done that, click OK in the bottom right hand corner of the window and you will return to your project window.



Part 2: MMIO and the LEDs

We finally have CCS setup correctly. What you need to do now is build the project. It is likely that you’ve got errors of various sorts (syntax, logical, other) in your code. The rest of these directions assume your code is working correctly. If it’s not, you are going to have to figure out what’s wrong. It could be that you did something wrong in part 1, it could be you’ve introduced a syntax error or it could simply be that your pre-lab answers are wrong. This is going to take a fair bit of time. *Some debugging suggestions are found below, so take the time to read ahead a bit before you do anything.*

We’ve now got things set-up, but still need to build the project. This can be done by right clicking the project and selecting the “Build Project” option or by going to the Project tab and selecting “Build

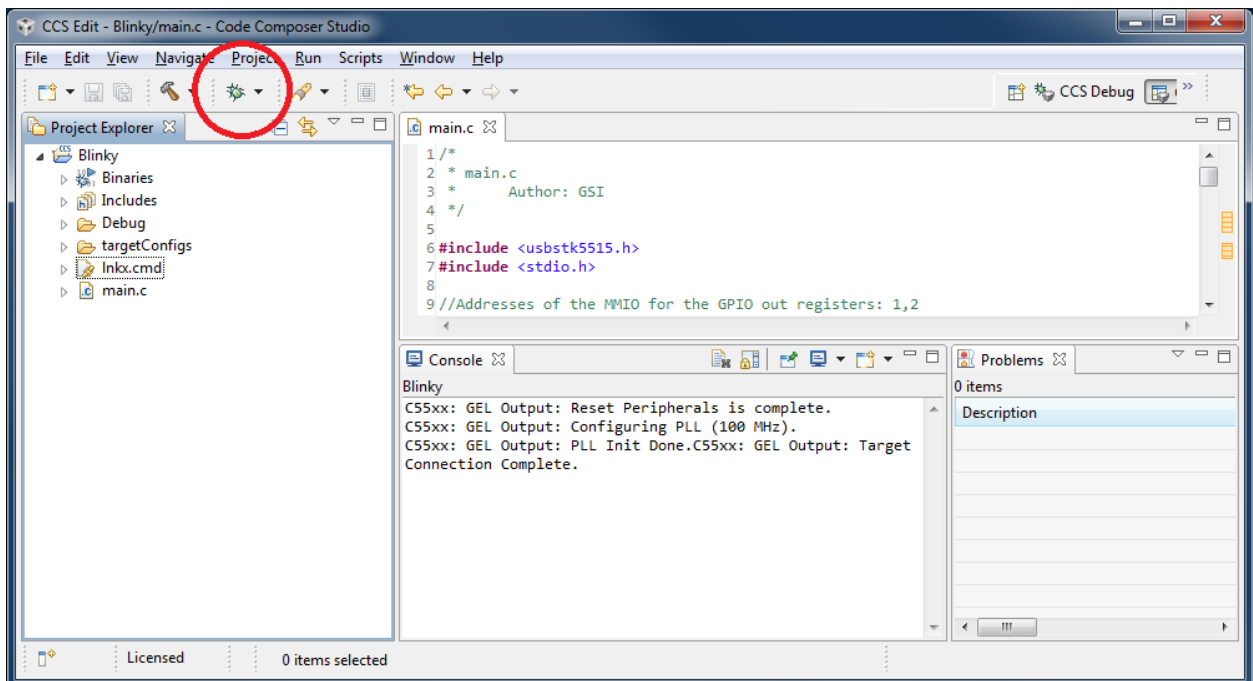
Project” from there. Building readies the project to be loaded on to board by generating the necessary binaries that the C5515 will run. You’ll need to fix any errors that pop up here.

If you get stuck at this step consider commenting out all but the following few lines. This should compile and run just fine (though it doesn’t do anything other than prompt you for a number). If it doesn’t, you’ve likely done something wrong in Part 1. Also, notice the section at the end of the In-lab called “When things go wrong”.

```
#include <usbstk5515.h>
#include <stdio.h>

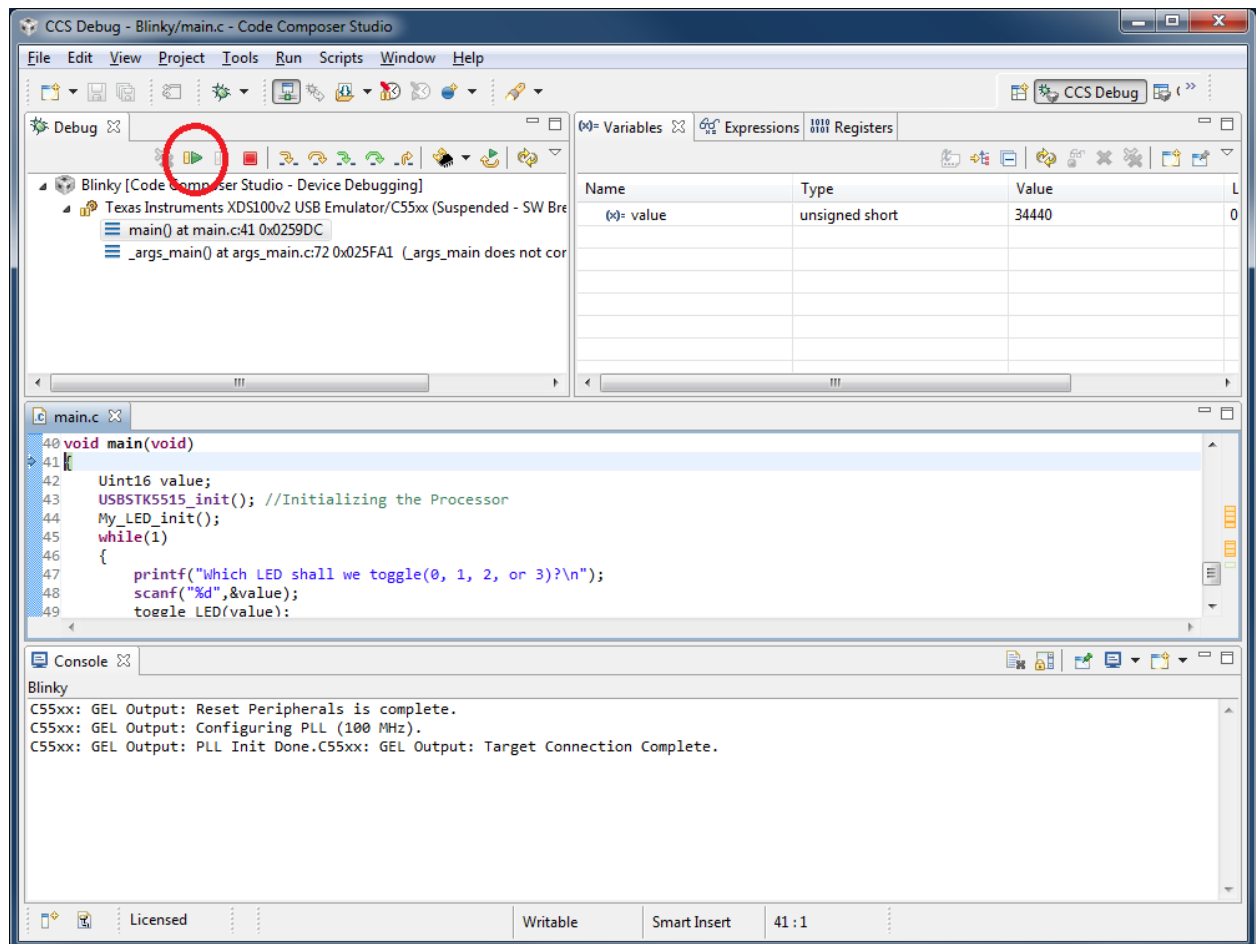
void main(void)
{
    Uint16 value;
    USBSTK5515_init(); //Initializing the Processor
    while(1)
    {
        printf("Which LED shall we toggle(0, 1, 2, or 3)?\n");
        scanf("%d",&value);
    }
}
```

Now we need to launch the program and load it on to the C5515. First be sure the eZDSP stick is connected to your computer with a USB cable. Then select the “Debug” icon which we circled in red in the following figure. If you get an error message at this point, ask for help from a lab instructor³.



³ It may be the case you need to get a first debug configuration set up. This largely involves finding the “.ccxml” file appropriate to your device. Right click on the project folder and chose Open Target Configuration. From there you are looking for the Texas Instruments XDS100v2 USB Emulator as the connection type and the USBSTK5515 as the device.

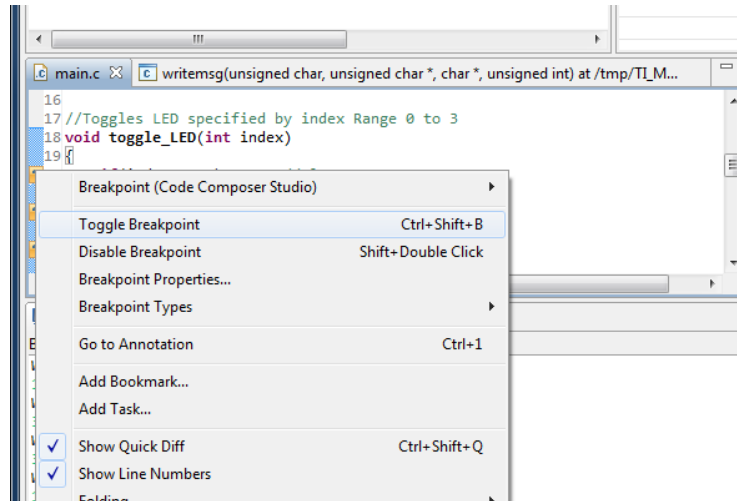
If all goes well you should be changed to the “CCS Debug” perspective as shown below.



This debug perspective allows you to easily run the program you just loaded. To run the program, click the green arrow toward the top-left side of the screen (circled in red in the figure above). The console should prompt you to enter which LED to toggle. If you don't see your console, make it visible “View” → “Console”. Click in the “console” and enter 0, 1, 2 or 3 with the keyboard. The LEDs should toggle when you send the command through the console.

Of course, when things go wrong simply running the program doesn't provide enough feedback for you to understand *what* is going wrong. This is what breakpoints are for. Breakpoints pause a running program so that you can use the debug perspective to see all the values of variables and the values stored in particular places in memory.

To set a breakpoint you will want to pause the program. Click the “Suspend” icon next to the green arrow icon (double yellow bars) to pause the program. Now let’s set a breakpoint for the first line of the function `toggle_LED` so we can see the default values of the local variables and memory. To do this go to the window displaying your code and right click on the line you wish to have the breakpoint. Make sure you right click on the actual line number and not inside the code. Select “Toggle Breakpoint”. Do this again for the last line of the function.



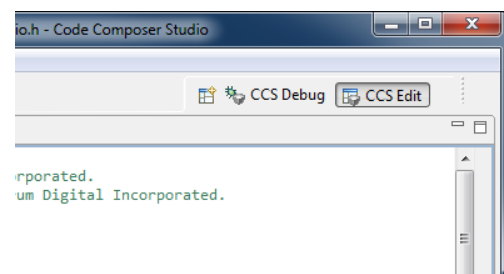
To view the values stored in memory click on the tab labeled “Memory Browser” (if no such tab exists you can bring it up via “View→ Memory Browser” from the menu). In the dropdown select “IO”. In the field to the right of that dropdown, type in the address of LED_OUT1 (don’t forget the 0x before the value because this tells the program the address is in hex instead of decimal.)

Now click on the green arrow again to run the program. It won’t stop until your function is called so toggle LED 3 from the console window and wait for CCS to pause the program (it should not take long.) Set the “IO” memory location to 0x1c0a. Then press the green arrow again. You should see that all of these values have changed. If you find yourself juggling windows, you may find it helpful to “detach” the memory browser and/or console window so you don’t need to swap back and forth between them (right click on the tab and select “Detached”.) Or you can drag the various tabs around the CCS main window to rearrange things however you like.

- Q1.** What values do you see at memory locations 0x1c0a as you toggle LED 3 on and off? Does toggling LED1 on and off cause that memory location to change? Why or why not?
- Q2.** What about memory location 0x1c0b? What values do you see there as you toggle LED 3 and off? Does that value change? Why or why not?

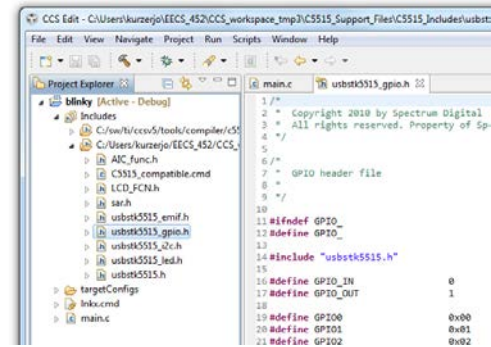
Digital Spectrum (the company that made the board) has provided several functions for us to use so we don’t always have to manually manipulate the MMIO.

To find these functions first terminate our run by clicking the red square next to the suspend button. This should automatically switch you back to the “CCS Edit” perspective. You can verify which perspective you are in by looking at the icon labeled “CCS Edit”⁴ in the upper right of the screen.



⁴ This will sometimes be hidden, it’s in the tab on the upper-right, you may need to click on a “>>” icon.

Now in the Blinky project folder you see a folder labeled “Includes.” Expand this section to show all the folders that hold your header files. Expand the folder “C5515_Includes”. You will see all the header files in that folder. Now double click on the file named `usbtk5515_gpio.h`. This will open it up in the code viewing window. Notice the functions that are prototyped at the end of this header file. Search for the source code for these functions (either on the Internet or, ideally, on your computer) and figure out what they do.



Q3. Rewrite the code for the “if(index == 3)” case of `toggle_LED` using these functions rather than the way we did it. What would you say an advantage of using the pointers was rather than this library code? What is an advantage of using the library code rather than the pointers?

G1. Have your GSI test your working code.

Part 3: More MMIO using the LCD.

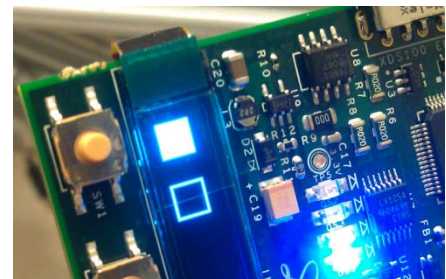
There are plenty of peripherals on the board in addition to the LEDs. In this section we will look at how to communicate to the small liquid crystal display (LCD). We will be using functions that were given to us to communicate to the LCD. There is a document which covers the LCD in detail that can be found with the rest of the lab documentation. It is the **SSD1306 128 x 64 Dot Matrix OLED/PLED Segment/Common Driver with Controller** (never mind that our LCD is 96 x 16, that’s the controller they used).

Copy your blinky project in the project explorer. Rename the copy “Square_Maker”. Replace the `main.c` code with `zigzag.c` (found on the Ctools). The supplied code will produce a zig-zag line on the LCD. You will need to add one new file to the project: `LCD_FCN.c`, which is also available on the course website. Get this code up and running.

G2. Demonstrate the working zigzag code to your GSI.

- Q4.** The C5515 has a built-in LCD controller. The folks who designed the C5515 eZDSP Stick chose not to use that built-in controller and instead used an external I2C controller.
- Explain, in a few sentences, what I2C is.
 - Look at the **OSD9616_send** function. What does it do?
 - In `main.c` what do you think the purpose of the “top” and “bottom” arrays are? Try to clear (turn off) the entire display. Try to make the entire display turn on.
 - In the prelab you were asked to draw the zigzag figure by hand. Does your answer match what was displayed? What’s different if anything?
 - The display is only 16x96 even though the code might lead you to believe it is 16x128. Which parts of top and bottom aren’t being displayed?
 - After the `printf` statement there are a bunch of **OSD9616_send** function calls. What do you suspect their purpose is?

G3. Modify this sample code and make two squares appear anywhere on LCD. Both squares are to be 16x16. One should be with filled in and the other just an outline.



They should not be touching. Something like the image to the right. Demonstrate this working code to your GSI.

Part 4: Generating and working with basic waveforms

Now that you have a basic understanding of how the LCD works, let's work with real signals and try to display something a bit more interesting. For this portion of the lab you will import a pre-made project. This project takes keyboard input and uses that to decide what to display on the LCD. You can display either a 1 KHz sine wave, or you can display an external input. If you push both buttons, you'll be again prompted to choose what to display.

In the Lab1 files from the course website, there is an archived project named "Audio_Project.zip". To import this project go to "File->Import...". Expand the "Code Composer Studio" selection and select "Existing CCS Eclipse Projects" and click "Next." Choose the "Select an archive file" option and "Browse" to the Audio_Project.zip file. Click Finish. Now build and run Audio_project.

You should get a prompt asking you what you want to display. Choose "0" and you'll get a simple sin wave. Your other options are to see the input from the left or right stereo input. The code grabs its data once you make a selection and then displays it. **It does *not* continually update, the data is just sampled once and thrown in a buffer.** Next, use the function generator to generate a 1.5 KHz sign wave, centered at ground and with a Vpp of 1.4V. Display this output on our oscilloscope. (You will want the function generator's output load set to Hi-Z by pressing "Utility" → "Output Setup" → "High Z" → "DONE".)

Connect the function generator to the "stereo in" of the board using either the left (white) or right (red) channel. Ask your GSI for help if needed.

Q5. At what voltage levels does our converter start saturating? (Do not drive more than 4V peak-to-peak to the board please; it can likely handle it, but...). Don't worry about being overly precise (within 0.2V will be fine).

Now modify the provided code so that if "0" is selected, you are again prompted to choose between displaying the following frequencies: 500Hz, 750Hz, 1kHz, 2kHz, and 3.5kHz. Your code must use the same sine table. This means either repeating or skipping entries in the sine table when creating the array to be displayed. Spend a few minutes thinking about how to efficiently implement this code without using division or mod. While efficiency isn't really important here as this operation isn't done continuously, it's good practice. Bit shifting is the key here.

Q4. Implement these changes in main.c under "if (mode == 0)" code. Show your results and code to the GSI. Your code should be reasonably efficient.

WHEN THINGS GO WRONG:

Here is a list of fairly common issues you might encounter.

- If there is a compilation error regarding unresolved symbols it may mean that your include files or paths are wrong.

- If an error occurs during linking then your file search path in the Linker options doesn't include a folder with that symbol in it.
- If there is an error about the support library not supporting model 3, check that you set the memory model and silicon revision correctly.
- If your program fails while running check your console. If you see a warning or error message saying one or more sections fall into non-writable memory, then go into your Linker Options->Runtime Environment and switch it from ROM to RAM. You may still get the warning but it probably won't impede your program's performance.
- If you get a linker error stating the maximum space for local variables is exceeded, then move your declarations of your arrays outside the function. This will make them global variables and cause the compiler to store them in a different place.
- If you get a runtime error, restart code composer.
- If you can't get DEBUG working, right click on the project folder and chose "Open Target Configuration". From there you are looking for the XDS100v2 USB Emulator as the connection type and the USBSTK5515 as the device.