EECS 452 Lab 1—Introduction to the C5515 USB Stick

The purposes of this lab are to:

- Provide an introduction to the C5515 USB Stick, the board we will be using throughout this course.
- Provide an introduction to embedded systems and memory-mapped I/O devices.
- Provide a very basic experience with direct digital synthesis (DDS)

In addition, you will gain familiarity with the software development environment and the various forms of documentation and sample code that comes with this platform.

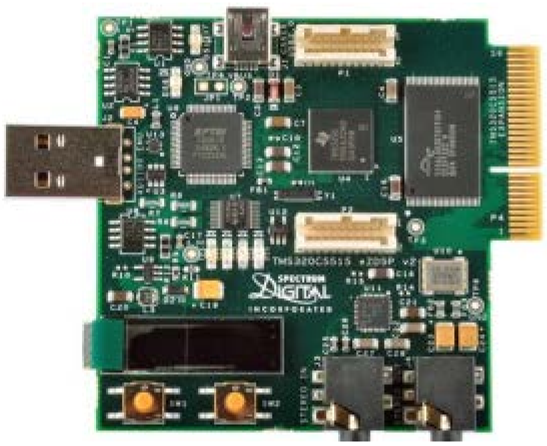


Figure 1: The C5515 eZDSP USB Stick

1. Background Material

There is quite a bit of material you'll want to understand before you start working on this lab. Some of that material was covered in previous courses, but some of it you'll be expected to pick up on your own.

Programming

All labs this semester will rely upon you having a fairly strong understanding of C programming. In EECS 280 and Engineering 101 you gained a fairly strong background in C++ programming. The differences between C and C++ are largely trivial. The two biggest, for purposes of this class, are that:

- There are no classes, nor are their function members in structures.
- Rather than using streams ("cout" and "cin") to do input and output we use "printf" and "scanf".

A brief tutorial on "C for C++ programmers" can be found on the 452 lab webpage under "General references"

Overview of Memory-mapped input/output

When using an embedded system, you generally need some way to move data into, and out of, the processor. How does the computer do that? In 280 the answer was simple; you just call a function/class (ofstream's *open* function to open a file was the most likely scheme). But that begs the question—how does the function do it? How do does the processor talk to the disk and other I/O devices?

The answer is fairly straightforward. Recall that your program and its data all live in memory. Each instruction, variable and array element are stored in a specific memory location. You can think of the computer's memory as a big array in which the program and its data reside. When you declare a variable, a memory location is reserved for that variable. Pointers are just the index into that big array that is memory. So when you do a *new* in C++, the pointer you get back from new is just the index into that big array where the operating system has given you space to work.

It turns out certain elements of that big memory array aren't really memory, they are instead dedicated I/O devices. What that means is that some memory locations are special and reads and writes to them might have side effects. A simple example would be to have memory location 0xDEADBEEF[1] mapped to an LED. It might be configured so that if the value in that memory location was non-zero the LED would turn on. One could have a different memory location where a switch's value could be found (say depending on the switches setting the memory location might be a 0 or a 1). Thus, your program could find the state of that switch simply by reading that memory location.

Significantly more I/O devices can be addressed with Memory Mapped Input/Output (MMIO). In fact all I/O devices; including disk drives, monitors and keyboards; communicate with the processor via MMIO. Wikipedia has a Memory Mapped I/O overview which you might find provides a helpful background.

Details of Memory-mapped input/output on the C5515

In most machines Memory Mapped I/O is simply a part of the overall memory map; but on the C5515 (and most TI DSPs) there is a separate memory space dedicated to the use of I/O devices. This is signaled by the keyword ***ioport***. This tells the processor that the address associate with this variable is mapped to the I/O address space, instead of the Data address space. Let's look at an example of a pointer to I/O memory:

```
ioport int * RandMemLoc;                //This declaration is "wrong" because it
RandMemLoc = (ioport int *) 0x12;       //lacks the volatile keyword!
```

The first line of this example declares and sets a variable named **RandMemLoc** which is pointer to an integer. It is then set to point to the memory location 0x12, in the ioport memory space. Here's an analogy which might help understand what the keyword **ioport** is doing here. If the address 0x12 is a page number then ioport is a title of a book.

There is also another keyword that must be used with MMIO, and that is the keyword ***volatile***. That keyword tells the system that it must load and store all the way to memory with each and every use[2]. Here is an example of how to set a variable to volatile. We'll do both the declaration and the assignment of value in one line this time.

```
volatile ioport int * RandMemLoc  = (ioport int *) 0x12;
```

[1] (Recall that "0x" indicates that the number that follows is written in hexadecimal, so 0xDEADBEEF would be 3735928559 base 10).

[2] Normally the *computer* will cache data and the *complier* will store commonly-used data in its register file. The volatile keyword announces the fact that the data could be read or written by something other than the program and therefore every read and write to the data needs to actually go all the way to memory (it can't stop in the register file or cache).

Handing in your work

In the pre-lab there will be a number of questions (labeled Q1, Q2, etc.). You are to create a single document (likely in Word) that answers each of the questions. It should be clear what question you're answering and the answers should be in order. The pre-lab is to be done individually, and *each student will turn in their own answers for the pre-lab.* Pre-labs are due at the beginning of the lab.

Make a copy of your pre-lab, you will need it for the in-lab part.

The lab and post-lab will also have a number of questions. The post-labs will tend to be longer questions that can be done outside of lab. Each team will turn in one set of answers for this part. The lab section also has a number of "sign-off" points where the GSI will need to sign-off that you've done a step correctly. The GSI may provide you with a blank sheet or you may need to create your own. You'll turn in this sign-off sheet along with the questions and any other requested information.

Post-labs are due at the beginning of the lab section on the following week.

2. Pre-Lab

The pre-lab questions in this class are to be done individually unless otherwise noted. Each student will turn in their own pre-lab.

In general demo boards are shipped with sample code that exercises the basic functionality of the board. The C5515 USB Stick is no exception. The problem with relying on such sample code is that A) sometimes (usually?) it's written poorly or in an unclear way and B) it can't cover everything. So you end up needing to fall back on the actual documentation which itself is sometimes less than clear.

For this pre-lab you are asked to write a function that "simply" allows you to turn on and off the four LEDs on our board. As is common in embedded systems, the problem is finding the right information in the documentation. In this pre-lab we'll walk you through finding the right information and writing the needed code from scratch. Below we have provided most of two functions which configure the LEDs and allow an individual LED to be turned on and off as well as a main which uses them. The function **My_LED_init()** configures the processor pins connected to the LEDs to be outputs. The function **toggle_LED(int index)** takes a number (0, 1, 2 or 3) and lights the corresponding LED. We've left a number of blanks for you to fill in.

The first step in locating the information we need to code the LED on/off function is to find how the LEDs are connected to the processor.
Our goal is to write code to turn on/off the LEDs. We know that the LEDs are connected to the MMIO. The first step is to find the documentation that tells us this mapping. Normal you would go to the manufacturer or distributor for this information, but you can find it on the EECS452 website. Look for the document named USBSTK5515 Technical Reference Revision A. You want to find which GPIO pins (General purpose I/O) are connected to the LEDs.

Q1. Which GPIO pin is connected to each of the four LEDs? (Red, Blue, Yellow and Green). Use the schematic (Apedix-12) not the table as there appears to be a typo in the table. In general, it is a good policy to trust the schematic above other references.

Once you find which GPIO pins are connected to the LEDs you have to find out how to use the GPIO pins. You can get this information in the TMS320C5515 General-Purpose Input/Output User's Guide. This user's guide ca be found on the EECS452 website (C5515_gpio_guide.pdf.) You will need to read sections 2.4, 3.1 and 3.3 at a minimum. You would be well-served to read all of chapters 2 and 3 to get a comprehensive understanding of the GPIO system on the C5515 processor.
Important note: *the LEDs are active low: to turn them on we need the GPIO pin to be 0.*

Q2. What are the names and memory locations of the registers you'll need to use to set the GPIO pins associated with the LEDs to be outputs?

Q3. For *each* of the four LEDs indicate which bit number of which register you'll need to write to in order to set that LED's GPIO pin to be an output.

Q4. Assuming the GPIO pins have been setup as outputs, indicate the register names, their addresses and which bits you'll need to write to what value in order to turn on the LEDs.

Now that we've figured out what values we need to write to which memory locations, we need to figure out how to write a single bit to a given memory location. Before answering the next question you may want to search a bit around the internet on the topic of "bitwise manipulation."

Q5. Answer the following questions. Assume all values are 16-bit unsigned where bit 15 is the most-significant bit and bit 0 the least significant.

a. What is the value of the number where bit 4 is a 1 and all other bits are zero? Provide your answer in both decimal and hex.
b. What is the value of the number (1<<4)? Provide your answer in both decimal and hex.
c. If x=0x2222, what is the hex value of x after the line **x|=(1<<4)**?
d. Write C code which sets bit 7 and 8 of the variable x.
e. Write C code which clears bit 1 of the variable x using the **&=** operator.
f. Write C code which toggles bit 13 of the variable x using the **^** operator.

Q6. Provide the code below with the blanks correctly filled in.

```
/*
 * main.c
 *      Author: GSI
 */

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

//Addresses of the MMIO for the GPIO out registers: 1,2
#define LED_OUT1 *((ioport volatile Uint16*)__blank a__ )
#define LED_OUT2 *((ioport volatile Uint16*)__blank b__ )
//Addresses of the MMIO for the GPIO direction registers: 1,2
#define LED_DIR1 *((ioport volatile Uint16*)__blank c__ )
#define LED_DIR2 *((ioport volatile Uint16*)__blank d__ )


//Toggles LED specified by index Range 0 to 3
void toggle_LED(int index)
{
        if(index == 3)  //Blue
                LED_OUT1 = LED_OUT1 ^ (Uint16)(1<<(__blank e__));
        else if(index == 2)  //Yellow(ish)
                LED_OUT1 = LED_OUT1 ^ (Uint16)(1<<(__blank f__));
        else if(index == 1)  //Red
                LED_OUT2 = LED_OUT2 ^ (Uint16)(1<<(__blank g__));
        else if(index == 0)  //Green
                LED_OUT2 = LED_OUT2 ^ (Uint16)(1<<(__blank h__));
}

//Makes the GPIO associated with the LEDs the correct direction; turns them off
void My_LED_init()
{
        LED_DIR1 |= __blank i__;
        LED_DIR2 |= __blank j__;

        LED_OUT1 |= __blank k__;  //Set LEDs 0, 1 to off
        LED_OUT2 |= __blank l__;  //Set LEDs 2, 3 to off
}

void main(void)
```

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

Now we'll change gears and get you started on some other parts of the lab: drawing on the LCD and generating discrete sine waves.

Q7. Consider the zigzag.c code found on the lab website (or CTOOLS.) It draws a zigzag pattern on the screen. *Draw*, freehand, what you expect this code to generate on the LCD. We are particularly concerned with how many zigzags you expect to see. You may find it helpful to read the document SSD1306 128 x 64 Dot Matrix OLED/PLED Segment/Common Driver with Controller (you can find it with a web search for "SSD1306.pdf"). Hint: ignore the OSD9616_send function for now, focus on the state of the **top** and **bottom** arrays. Page 25 of the document will also be helpful. Be aware that this document refers to a LCD with 8 pages but our LCD has only 2 pages.

Q8. Say you have a 50-entry sine table (that is the table[x] has the value for sin(2π*x/50)). Say you output each value for 1ms, then after you do the last value (table[49] in this case) you loop around and output table[0] again. Obviously the sine waves will be a bit choppy…

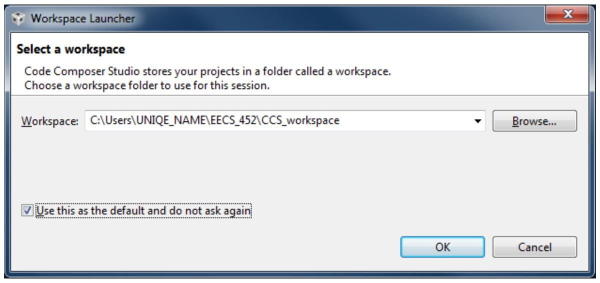
a. What is the frequency of the sine wave you'd generate?
b. What if you instead held each table entry for two samples in a row (2ms)? What would the frequency be?
c. What if you instead drove every other table entry for 1ms (so only table entries 0, 2, 4, etc.)? What would the frequency be?
d. Describe what you'd need to do to generate a 1Hz frequency sine wave.
e. Describe what you'd need to do to generate a 15Hz frequency sine wave.

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


Workspace Launcher

Select a workspace

Code Composer Studio stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

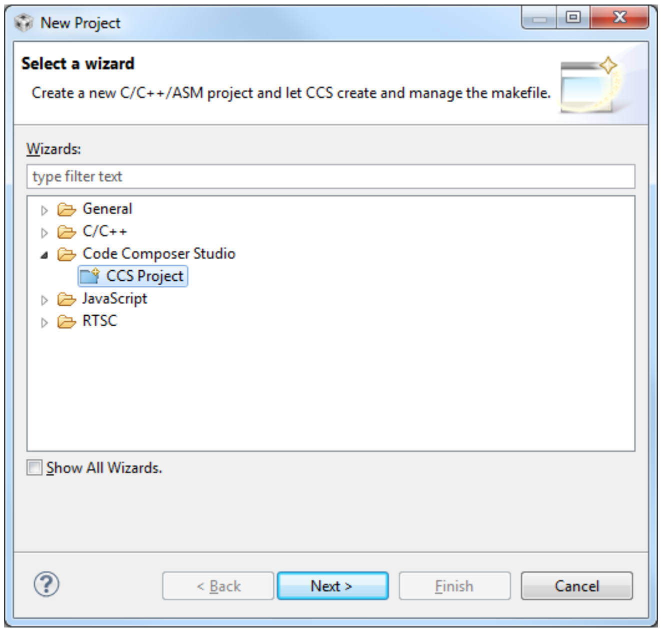
Workspace: C:\Users\UNIQE_NAME\EECS_452\CCS_workspace

[✓] Use this as the default and do not ask again

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\[uniqename]\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 an 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.'


New Project

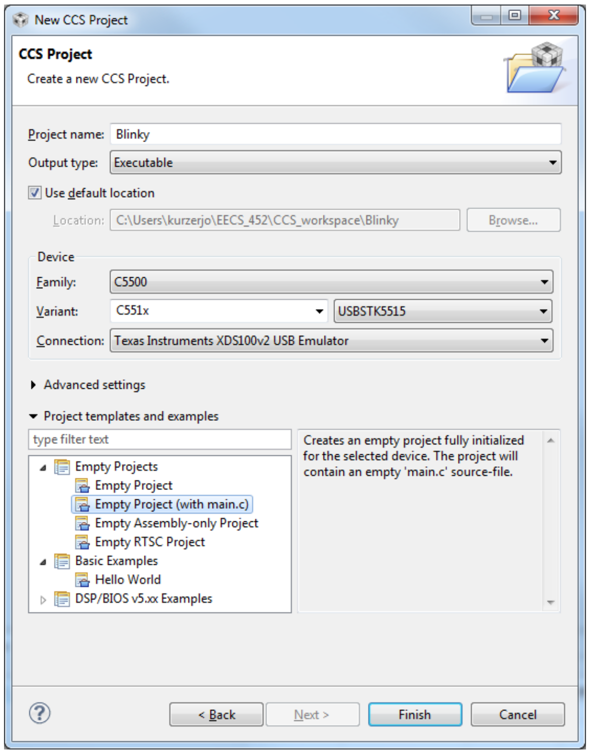
Select a wizard

Create a new C/C++/ASM project and let CCS create and manage the makefile.

Wizards:

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.' Finaly chose to create and 'Empty Project (with main.c) and click 'Finish.'



Project name: Blinky
Output type: Executable
☑ Use default location
Location: C:\Users\kurzerjo\EECS_452\CCS_workspace\Blinky
Device
Family: C5500
Variant: C551x USBSTK5515
Connection: Texas Instruments XDS100v2 USB Emulator

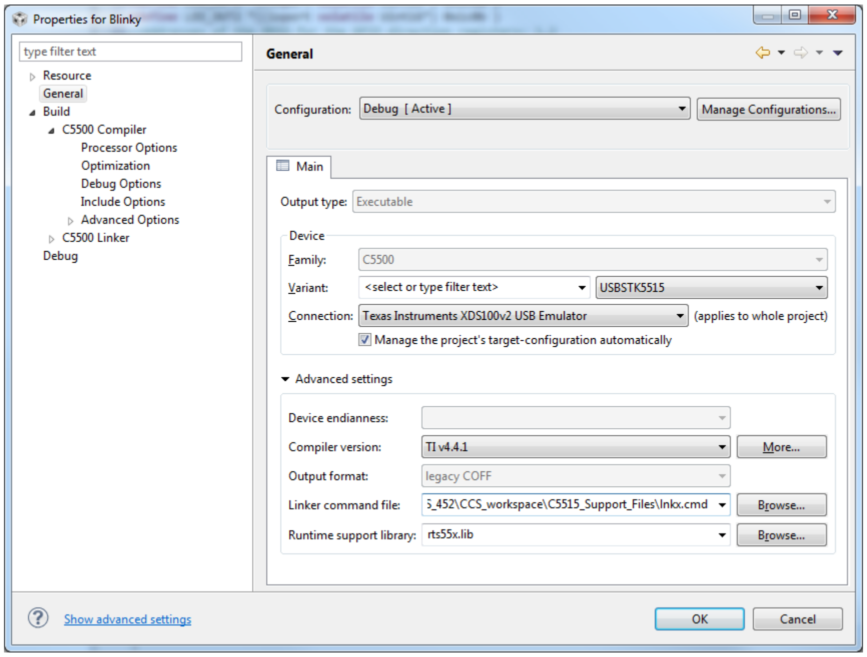
Creates an empty project fully initialized for the selected device. The project will contain an empty 'main.c' source-file.

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.

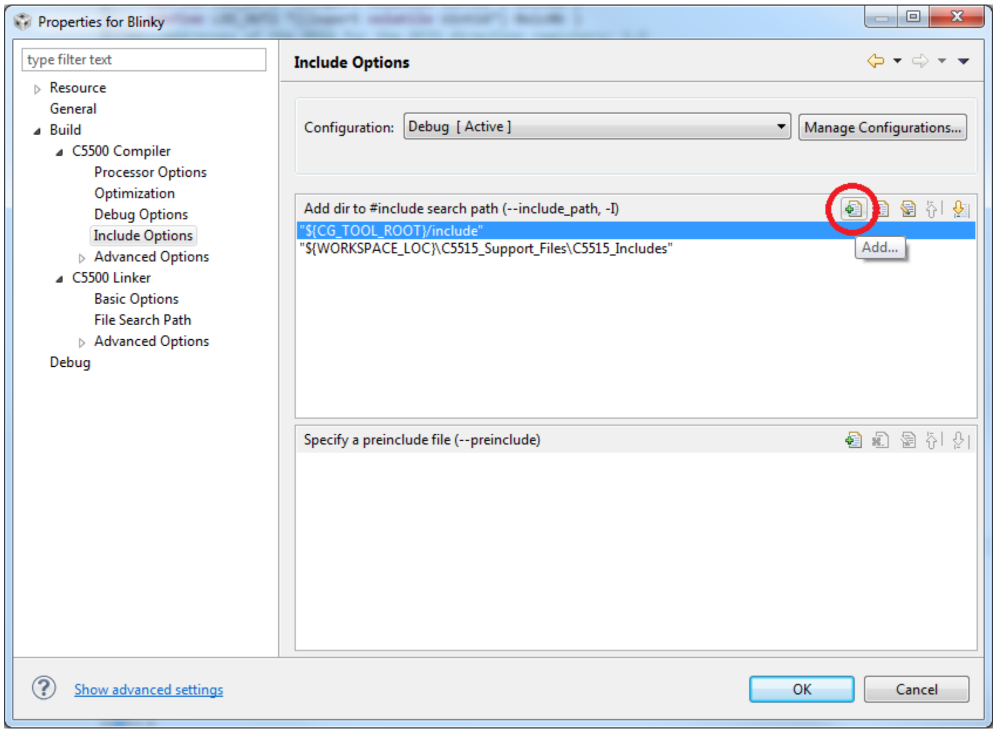
Now we have the code, but we still have to tell the compiler and linker where to look for source files and libraries that we need. Right click on the project ("Blinky") and select "Properties". The window below will appear. Look under the "General" settings.

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**."



Configuration: Debug [Active]
Output type: Executable
Device
Family: C5500
Variant: <select or type filter text> USBSTK5515
Connection: Texas Instruments XDS100v2 USB Emulator (applies to whole project)
☑ Manage the project's target-configuration automatically
Device endianness:
Compiler version: TI v4.4.1
Output format: legacy COFF
Linker command file: 5_452\CCS_workspace\C5515_Support_Files\lnkx.cmd
Runtime support library: rts55x.lib

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


Properties for Blinky

Include Options

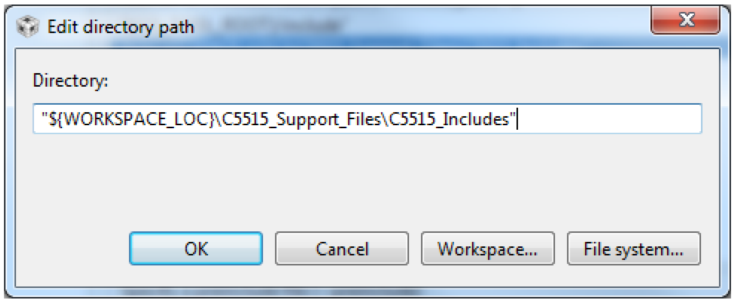
Configuration: Debug [Active]

Add dir to #include search path (--include_path, -I)

"${CG_TOOL_ROOT}/include"
"${WORKSPACE_LOC}\C5515_Support_Files\C5515_Includes"

Specify a preinclude file (--preinclude)

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**.


Edit directory path

Directory: "${WORKSPACE_LOC}\C5515_Support_Files\C5515_Includes"

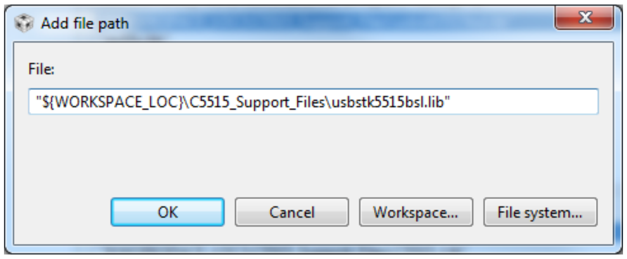
Note: ${WORKSPACE_LOC} is a path variable that is specific to your CCS setup. You can see this and other defined path variables in the properties window. “Resource” → “Linked Resources” in the “Path Variables tab.”

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.***

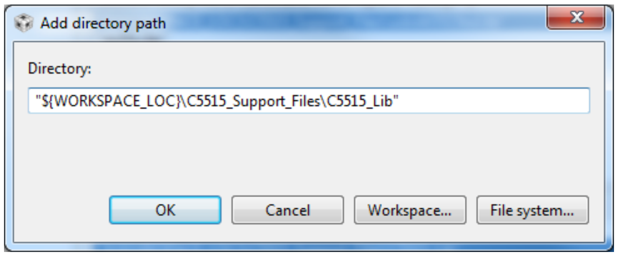
Now that we have told the compiler where to find our header files, we still need to tell it where to find the code to resolve the symbols declared in those header files. To do this, go to “Build” → “C5500 Linker” → “File Search Path.”

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.)

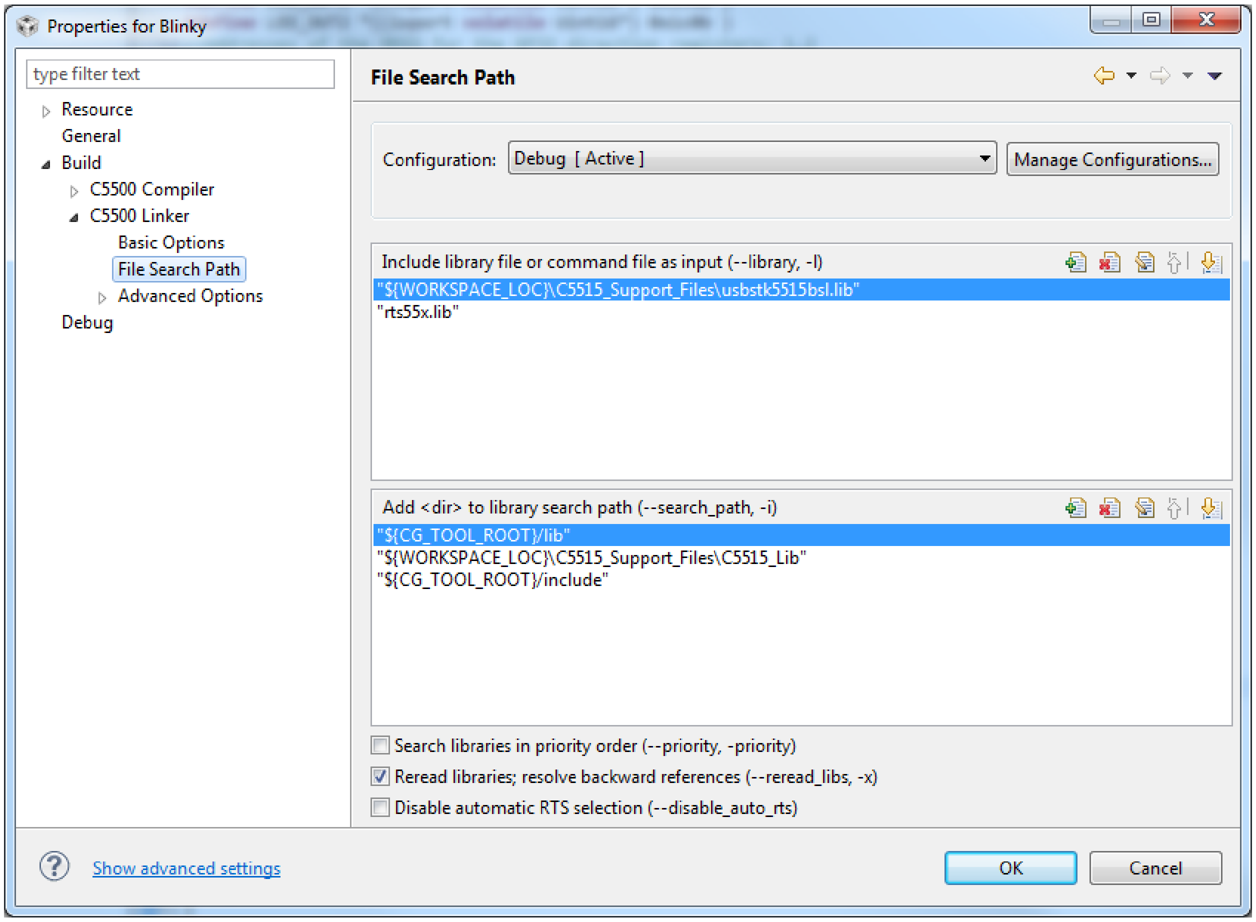
In the “Add <dir> to library search path” section, clickthe “add” button as before. In the “Add directory path” window that pops up, enter "${WORKSPACE_LOC}\C5515_Support_Files\C5515_Lib" (again, include quotes.)



File: "${WORKSPACE_LOC}\C5515_Support_Files\usbstk5515bsl.lib"



Directory: "${WORKSPACE_LOC}\C5515_Support_Files\C5515_Lib"



File Search Path

Configuration: Debug [Active]

Include library file or command file as input (--library, -l)

"${WORKSPACE_LOC}\C5515_Support_Files\usbstk5515bsl.lib"
"rts55x.lib"

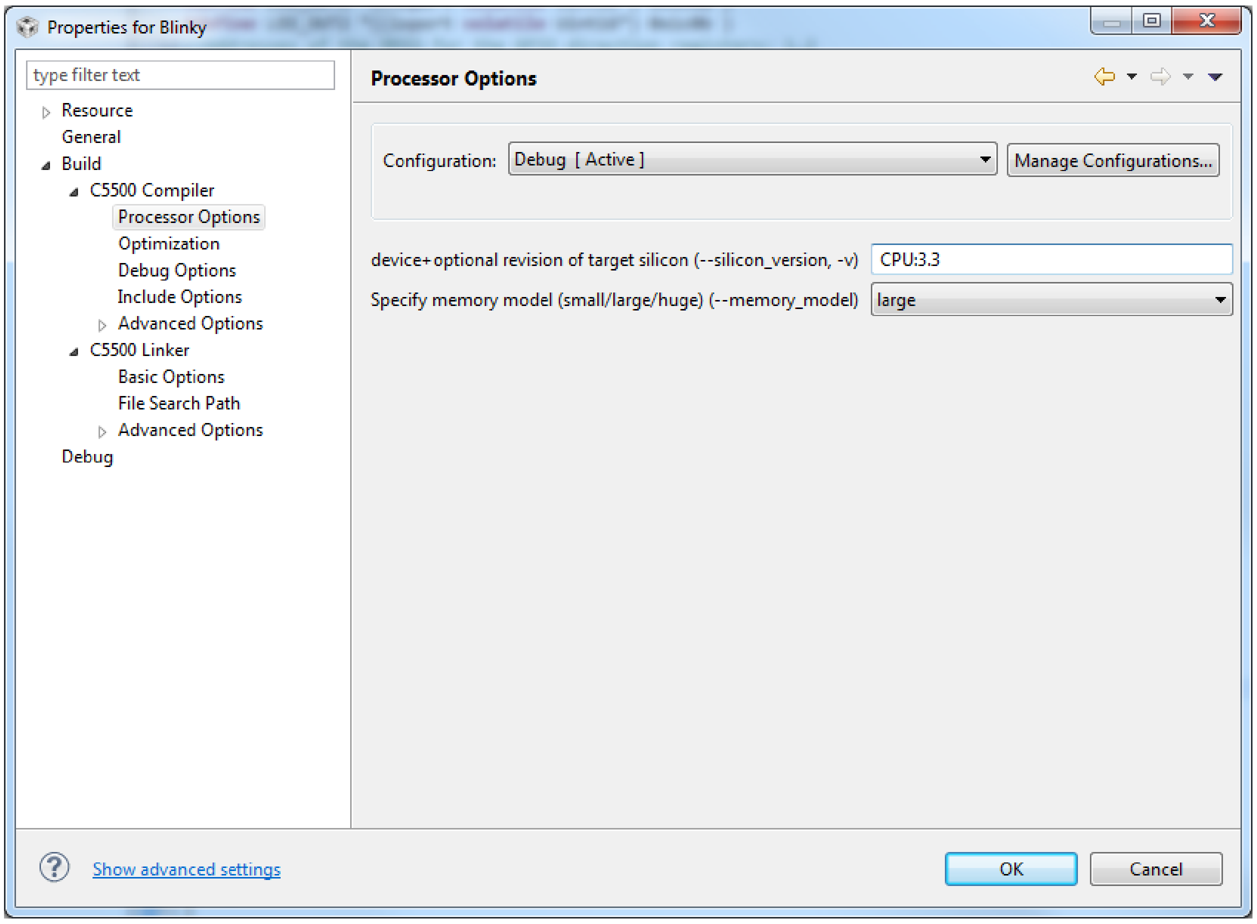
Add <dir> to library search path (--search_path, -i)

"${CG_TOOL_ROOT}/lib"
"${WORKSPACE_LOC}\C5515_Support_Files\C5515_Lib"
"${CG_TOOL_ROOT}/include"

☐ Search libraries in priority order (--priority, -priority)
☑ Reread libraries; resolve backward references (--reread_libs, -x)
☐ Disable automatic RTS selection (--disable_auto_rts)

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.


Properties for Blinky

Processor Options

Configuration: Debug [Active]

device+optional revision of target silicon (--silicon_version, -v): CPU:3.3

Specify memory model (small/large/huge) (--memory_model): large

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.*

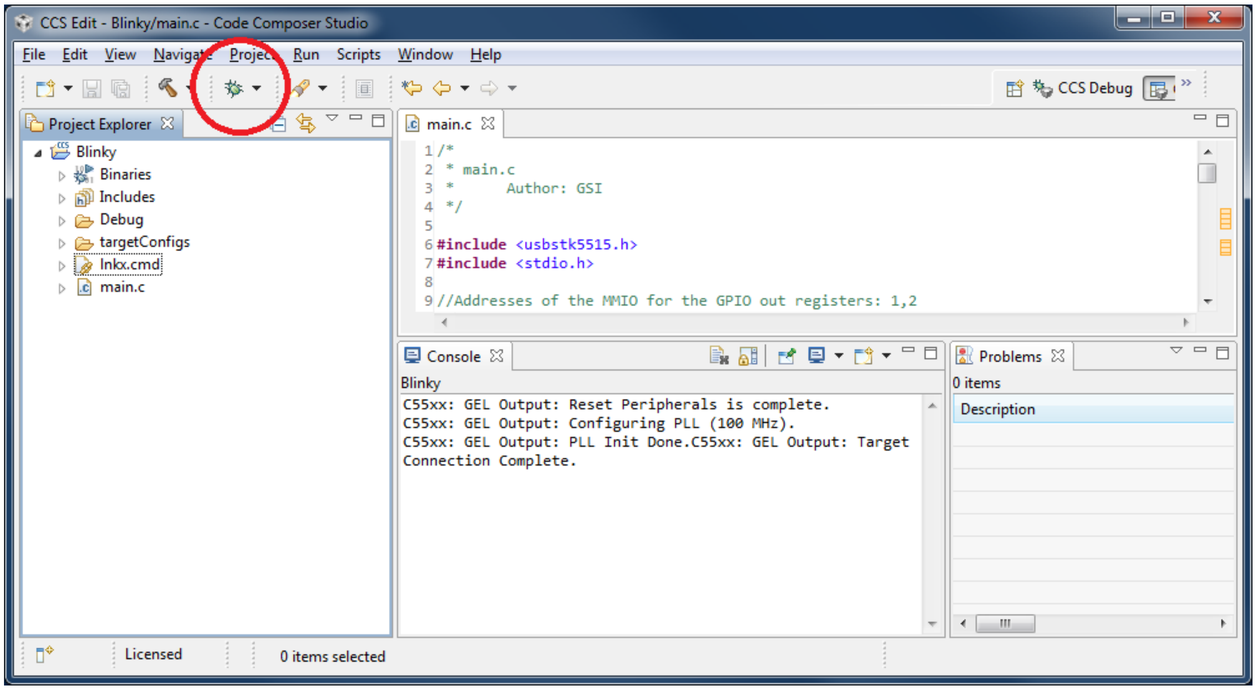
We've now got things set-up, but still need to build the project. This can be done by right clicking the project and selecting the "Build Project" option or by going to the Project tab and selecting "Build

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 instructor[3].



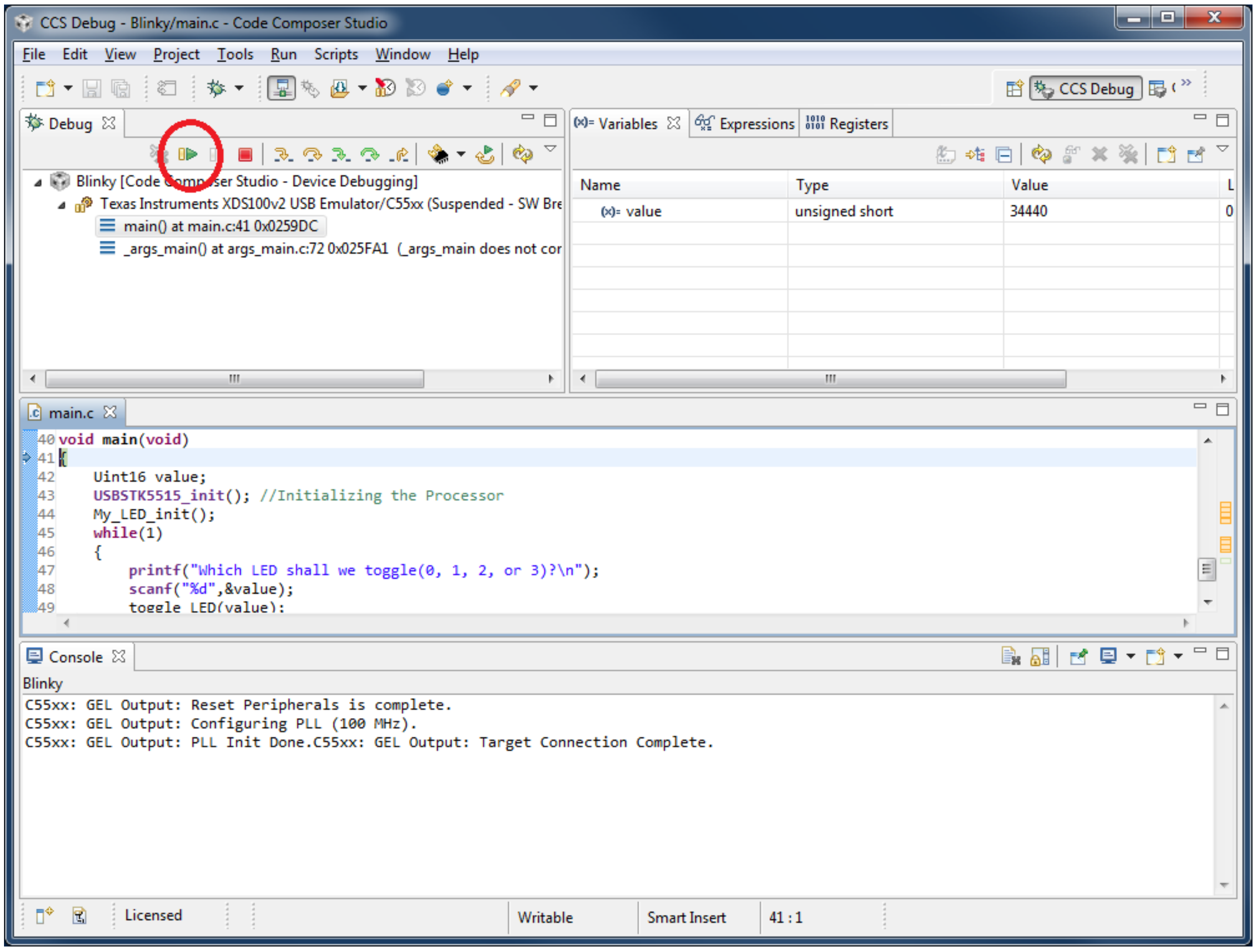
```
/*
 * main.c
 *      Author: GSI
 */

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

//Addresses of the MMIO for the GPIO out registers: 1,2
```

Blinky

```
C55xx: GEL Output: Reset Peripherals is complete.
C55xx: GEL Output: Configuring PLL (100 MHz).
C55xx: GEL Output: PLL Init Done.C55xx: GEL Output: Target
Connection Complete.
```

[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.


CCS Debug - Blinky/main.c - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
CCS Debug
Debug
Variables
Expressions
Registers
main.c
Console
Licensed
Writable
Smart Insert
41 : 1

Blinky [Code Composer Studio - Device Debugging]
Texas Instruments XDS100v2 USB Emulator/C55xx (Suspended - SW Bre
main() at main.c:41 0x0259DC
_args_main() at args_main.c:72 0x025FA1 (_args_main does not cor

Name	Type	Value
value	unsigned short	34440

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

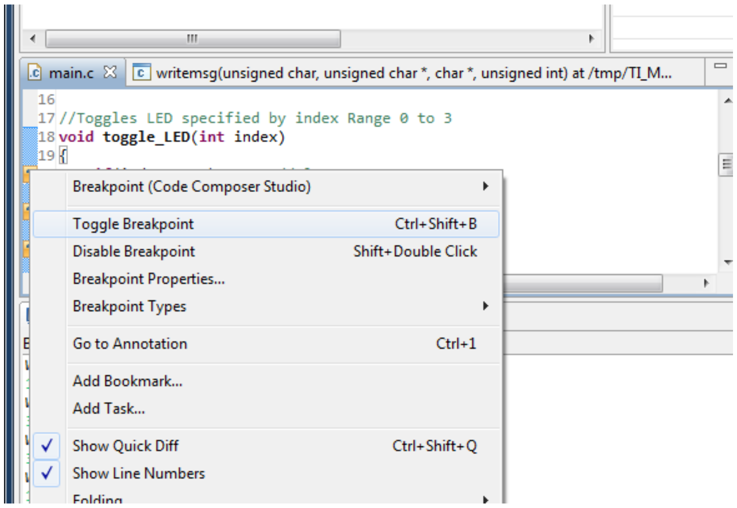
Blinky

```
C55xx: GEL Output: Reset Peripherals is complete.
C55xx: GEL Output: Configuring PLL (100 MHz).
C55xx: GEL Output: PLL Init Done.C55xx: GEL Output: Target Connection Complete.
```

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 to 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 you 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.


main.c
writemsg(unsigned char, unsigned char *, char *, unsigned int) at /tmp/TI_M...
Breakpoint (Code Composer Studio)
Toggle Breakpoint Ctrl+Shift+B
Disable Breakpoint Shift+Double Click
Breakpoint Properties...
Breakpoint Types
Go to Annotation Ctrl+1
Add Bookmark...
Add Task...
Show Quick Diff Ctrl+Shift+Q
Show Line Numbers
Folding

```
16
17 //Toggles LED specified by index Range 0 to 3
18 void toggle_LED(int index)
19 {
```

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.)

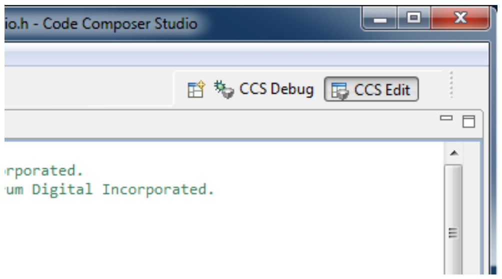
Now click on the green arrow again to run the program. It won't stop until you function is called so toggle LED 3 from the console window and wait for CCS to pause the program (it should not take long.) Set the "IO" memory location to 0x1c0a. Then press the green arrow again. You should see that all 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.

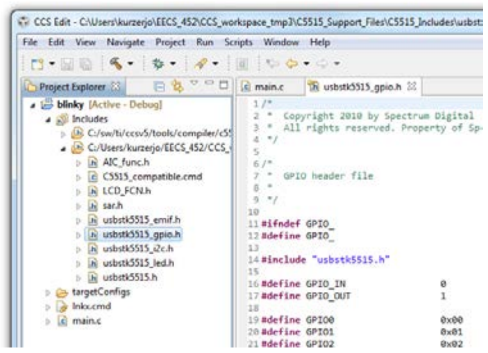
To find these functions first terminate our run by clicking the red square next to the suspend button. This should automatically switch you back to the "CCS Edit" perspective. You can verify which perspective you are in by looking at the icon labeled "CCS Edit"[4] in the upper right of the screen.



```
rporated.
um Digital Incorporated.
```

[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 usbstk5515_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.



```
/*
 *  Copyright 2010 by Spectrum Digital
 *  All rights reserved. Property of Sp-
 */

/*
 *  GPIO header file
 *
 */

#ifndef GPIO_
#define GPIO_

#include "usbstk5515.h"

#define GPIO_IN                  0
#define GPIO_OUT                 1

#define GPIO0                    0x00
#define GPIO1                    0x01
#define GPIO2                    0x02
```

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).

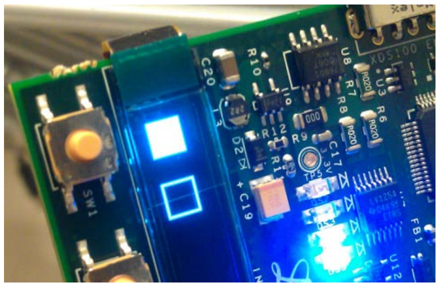
Copy your blinky project in the project explorer. Rename the copy "Square_Maker". Replace the main.c code with zigzag.c (found on the Ctools). The supplied code will produce a zig-zag line on the LCD. You will need to add one new file to the project: LCD_FCN.c, which is also available on the course website.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

a. Explain, in a few sentences, what I2C is.
b. Look at the **OSD9616_send** function. What does it do?
c. 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.
d. 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?
e. 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?
f. 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.

G4. 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 you 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.

4. Post-Lab

Q1. Go to the C5515_Lib folder and examine the function in usbstk5515_led.c which initializes the ULEDs. Copy that function and describe what is happening line-by-line for that function. You will need to look at the #defines in usbstk5515.h and usbstk5515_led.h file to truly understand what's going on there.

Q2. Go into the AIC_func.c file and provide a line-by-line description of the **AIC_write2** function.

Each group should hand-in the following material, neatly stapled:

- Your sign-off sheet. It should be on the front and include each partner's name and unique name.
- A typed set of answers to the questions from the in-lab and post-lab. If figures are required, neat, hand-drawn, figures are acceptable.
- The portion of the code you modified to generate the different frequencies in part 4. Don't provide the whole file please; just the section you changed.