

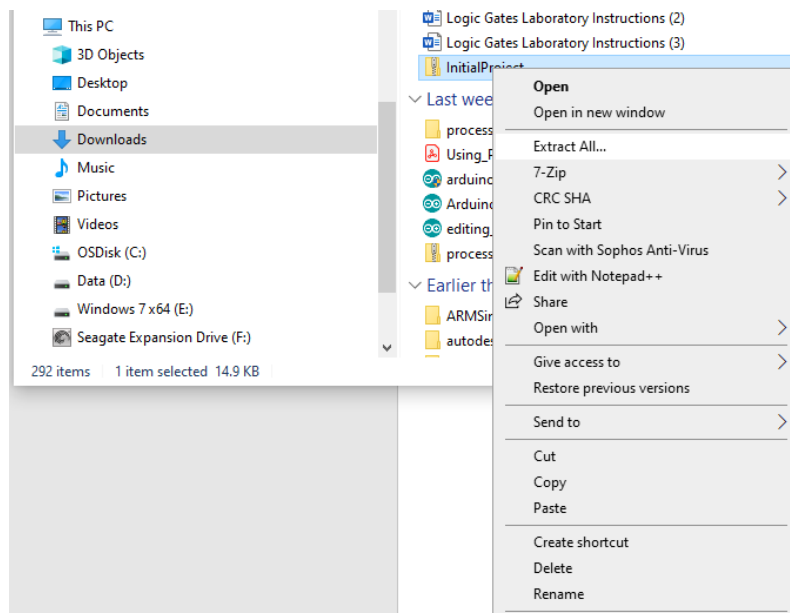
Functions and Variables

Introduction:

For this class you use a simple hardware development environment to introduce you to concepts associated with C Programming. This environment allows more access to hardware details. This week's reading and task is to allow you to understand how to use C functions and libraries.

Importing the Example Code:

Let's import some example code to get started. First, download the initial code , InitialProject.zip. On Windows, unzip the file by selecting the zip file, right clicking on the file name, selecting Extract All..., and chose where you want the unzipped directory to be on your computer. You can leave it in your Downloads directory if you like..

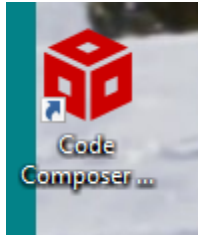


On MacOS, the file should unzip as soon as the download is complete.

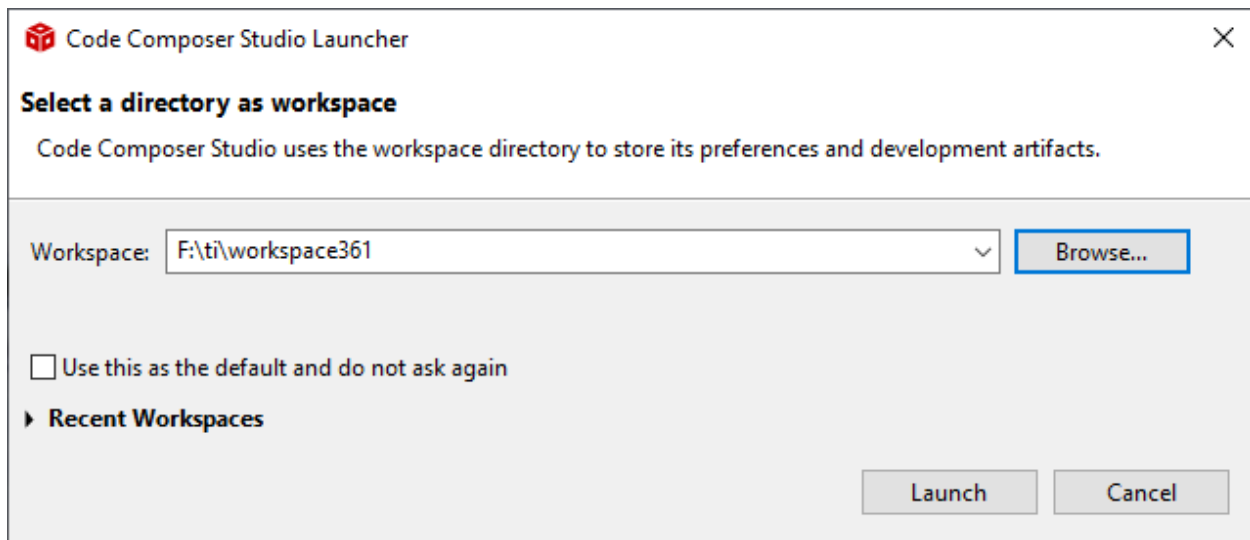
For both Windows and MacOS users, after the file is unzipped, import it into Code Composer.

Code Composer

Start Code Composer as you did last week by double clicking on the icon:

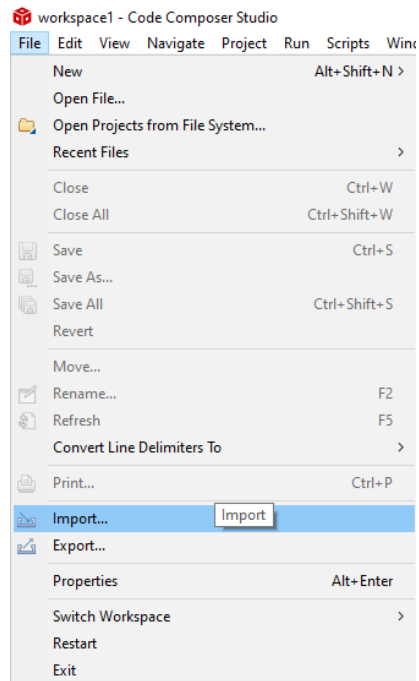


As the program starts you should be prompted for the workspace you want to use.

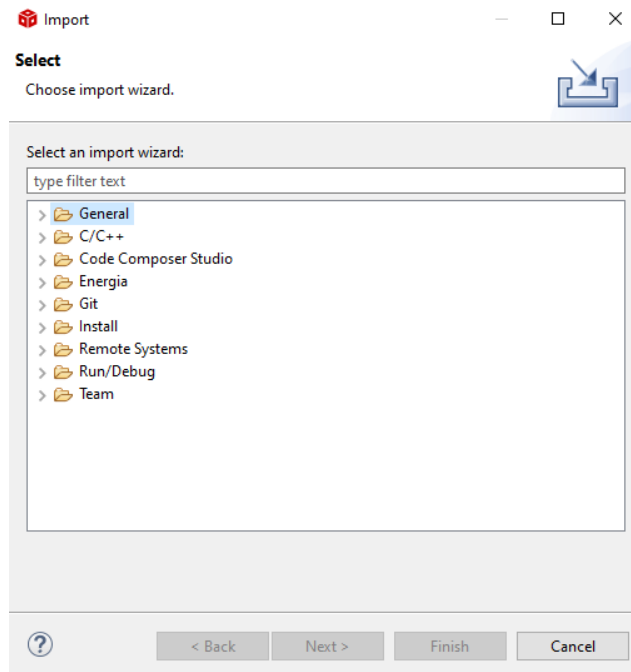


Remember this workspace is the directory where your project files will be placed. You can use several different workspaces if you want to work on very different projects. Click Launch when you have specified the workspace directory (you can just use the default.)

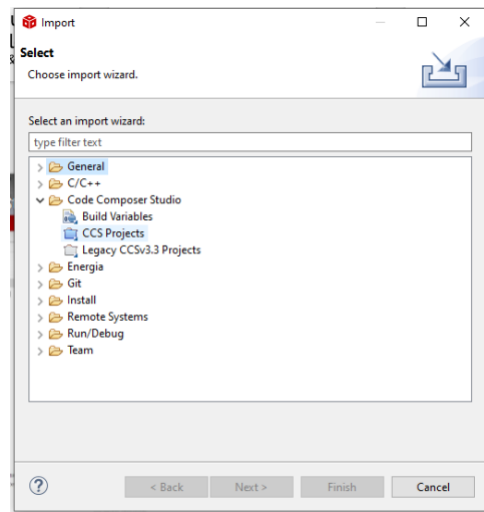
If you haven't already connect your MSP432P401R to the computer via a USB cable. Now let's import the project into Code Composer Studio. To do this select File->Import



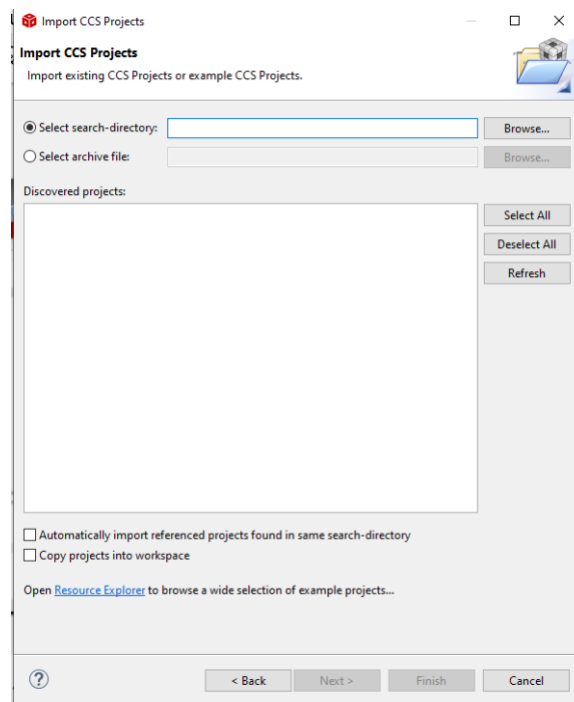
Then you should see this dialogue box:



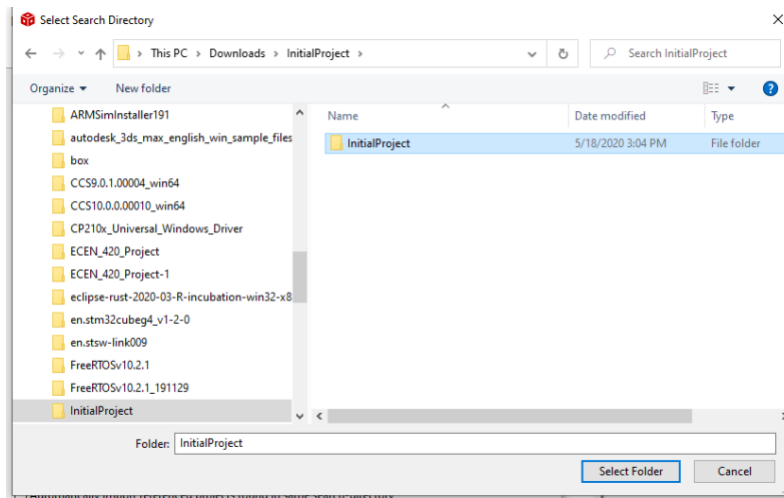
Now select Code Composer Studio, then CCS Projects:



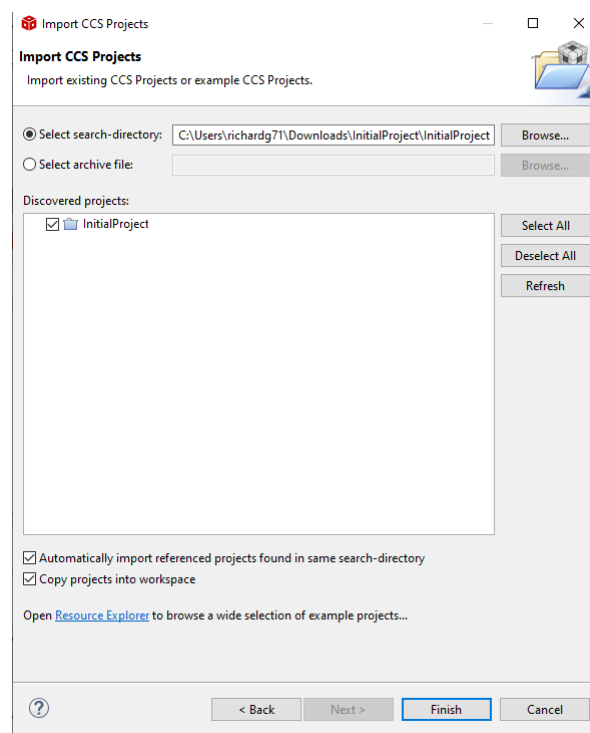
And you should see this dialogue box:



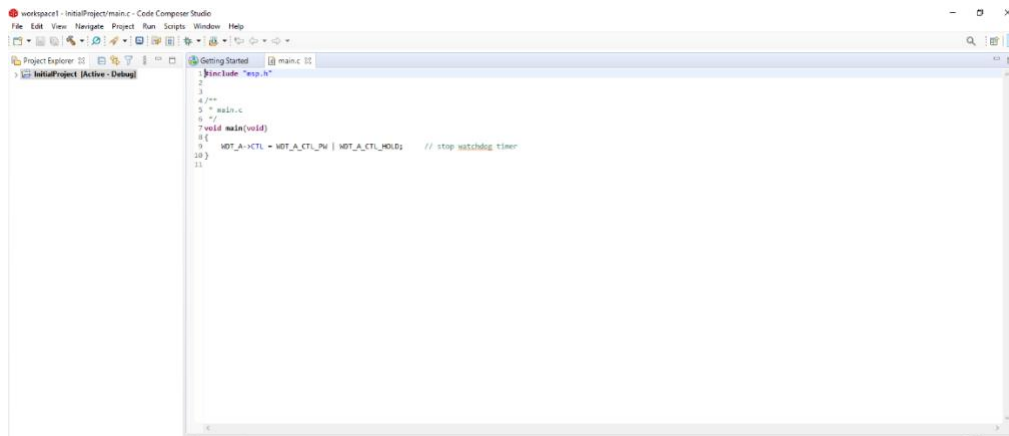
Browse to the directory where you unzip the archive. In my case it was my Downloads directory:



Click the Select Folder button. Then you should see this in the dialogue box:



Make sure the bottom two check boxes are checked, then click Finish. You should now see this project in the Project Explorer window:



You will also see the main.c file in the Editor Window. Let's look at some details. First, you will notice there is a main function. In C this is where execution of the code starts, not at the top of the file.

Also notice that there is a void before main function declaration. This tells the system that this particular function is set up to return nothing (void means nothing in C). C is a typed programming language, which means you'll always need to tell it the type of variables. Inside of the main function () you'll see a void, this simply means that in this case we are not going to pass the function anything.

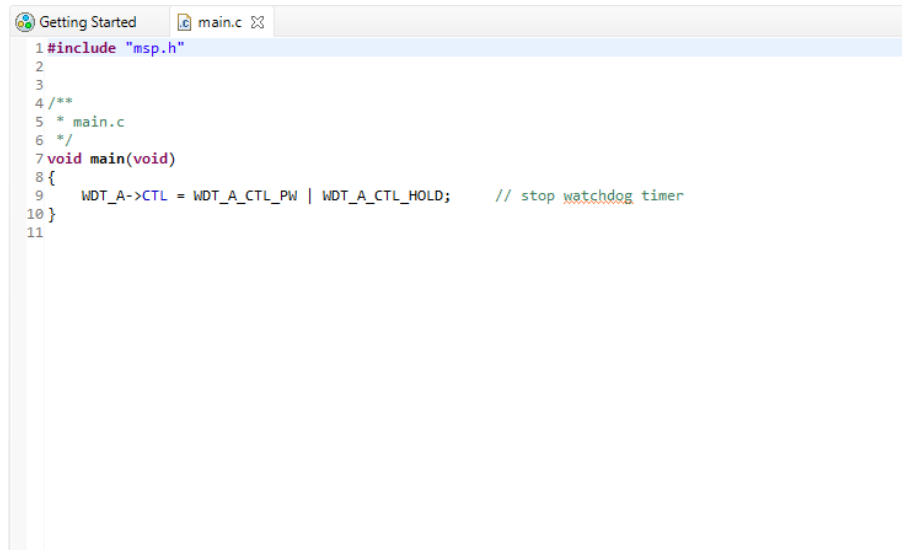
Also notice the curly braces around the statements in the function. In Python you use indentation to denote the scope of a function. C uses the { operator to indicate the beginning of the function's scope and the } operator to indicate the end of the function's scope.

As you have done and will do each week, make sure you are ready to start modifying the code in the project by compiling, running, and debugging it. If you have forgotten how to do this, go re-read the instructions in the reading for week02.

Now that you have a working, empty project, you can add the code for this week.

Simple Functions and Local Variables

Here is the code in main.c:



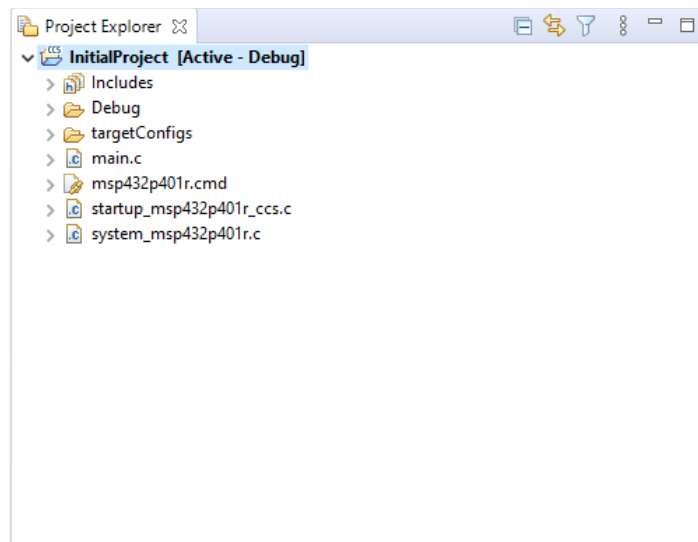
```
1 #include "msp.h"
2
3
4 /**
5  * main.c
6  */
7 void main(void)
8 {
9     WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer
10 }
11
```

As noted in the last exercise, the code will start execution with the main function. The all functions in C are using the same pattern. Going from left to right, first, the type of data the function returns when done is declared, then the function's name, then the parameters that will be passed into the function. In the case of the main function, the *void* keyword indicates that the function will not return any data, and has no parameters.

Notice line 1 in the code,

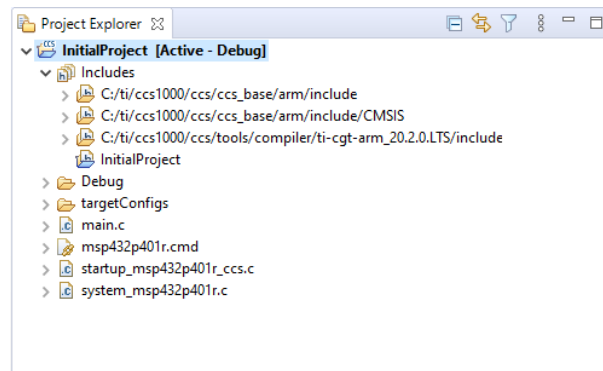
`#include "msp.h"`

This is a compiler directive to take the contents of the file "msp.h" and copy them into this file before compiling. This is a C version of python's *import* keyword and behavior. Where does the compiler find this file? If you look in the Project Explorer window (you may need to click on the arrow indicator just to the left of the project name to expand the view),

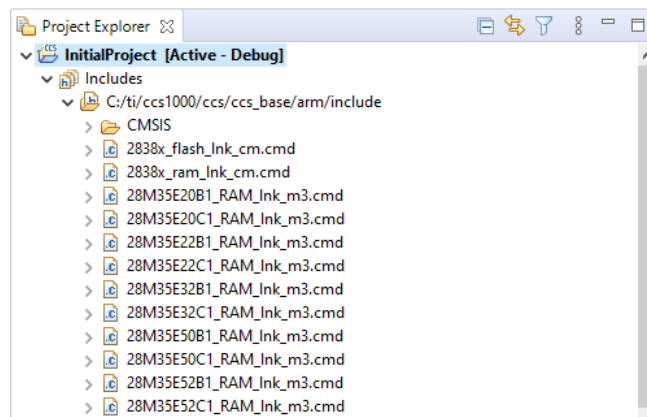


you will find all the directories and files associated with your project. The main.c file you've been modifying is listed. The msp432p401r.cmd, startup_msp432p401r_ccs.c, and system_msp432p401r.c files are configuration files used by the system to make configuration easier. You can safely ignore these for now.

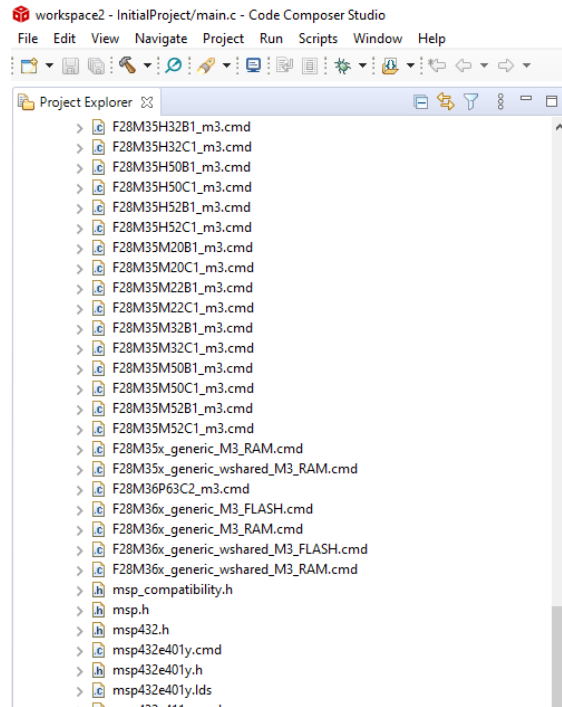
Expand the Includes sub-directory. Expand this directory and you'll see this:



This is where files are that you can use the pre-compiler command to include in your compiled application. They will only be included, however, if you use the pre-compiler command include them. Expand top directory




then scroll down a long way until you find the msp.h file.



Double click on this file and it opens in the editor window:

```
Getting Started | main.c | msp.h
16 *
17 * Neither the name of Texas Instruments Incorporated nor the names of
18 * its contributors may be used to endorse or promote products derived
19 * from this software without specific prior written permission.
20 *
21 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
22 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
23 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
24 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
25 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
26 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
27 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
28 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
29 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
30 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
31 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
32 *
33 * MSP432 Family Generic Include File
34 *
35 * File creation date: 12/06/17
36 *
37 *
38 *
39 #ifndef __MSP432_H__
40 #define __MSP432_H__
41 *
42 *
43 * MSP432 devices
44 *
45 #if defined (__MSP432P401R__)
46 #include "msp432p401r.h"
47 *
48 #elif defined (__MSP432P401M__)
49 #include "msp432p401m.h"
50 *
51 #elif defined (__MSP432P401Y__)
52 #include "msp432p401y.h"
53 *
54 #elif defined (__MSP432P401V__)
```

DO NOT MODIFY THIS FILE. It is a file that will include the appropriate include file required for your code to run correctly on the msp432p401r hardware, or other hardware if you were writing for it. Go back to the Project Explorer and open that file:



```
> msp432e401y.cmd
> msp432e401y.h
> msp432e401y.lds
> msp432e411y.cmd
> msp432e411y.h
> msp432e411y.lds
> msp432p4011.cmd
> msp432p4011.h
> msp432p4011.lds
> msp432p401m_classic.h
> msp432p401m.cmd
> msp432p401m.h
> msp432p401m.lds
> msp432p401r_classic.h
> msp432p401r.cmd
> msp432p401r.h
> msp432p401r.lds
> msp432p401v.cmd
> msp432p401v.h
> msp432p401v.lds
> msp432p401y.cmd
```

This is a very large file, and the system may complain a bit about it, but just click OK, and eventually you'll be able to view this file. This file contains all the hardware definitions for your MSP432P401R. Since this file is included in your application, you can use all of the definitions found here in your program.

You're going to add a function to main.c so switch back to editing it. Modify main.c to match this code example.

```

#include "msp.h"

void test();

/**
 * main.c
 */
void main(void)
{
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer

    test();

    return;
}

void test()
{
    int test1 = 0;
    test1 = 4;
    return;
}

```

Now let's look at what the code does.

First, the statement:

```
void test();
```

is called a prototype of the test() function. While the execution of the program will start at main(), the compiler program, the program that translates the program to machine code, starts at the top of the file and works through the file. The prototype statement tells the compiler that at some point in the file you will be creating a function called test() somewhere in the file, so you can use the function name anywhere in the file.

In the main function you'll notice this statement:

```
test();
```

When this command is reached the execution of the code will go to the test()_function and execute the commands there.

Now, below the main function, the test() function is defined:

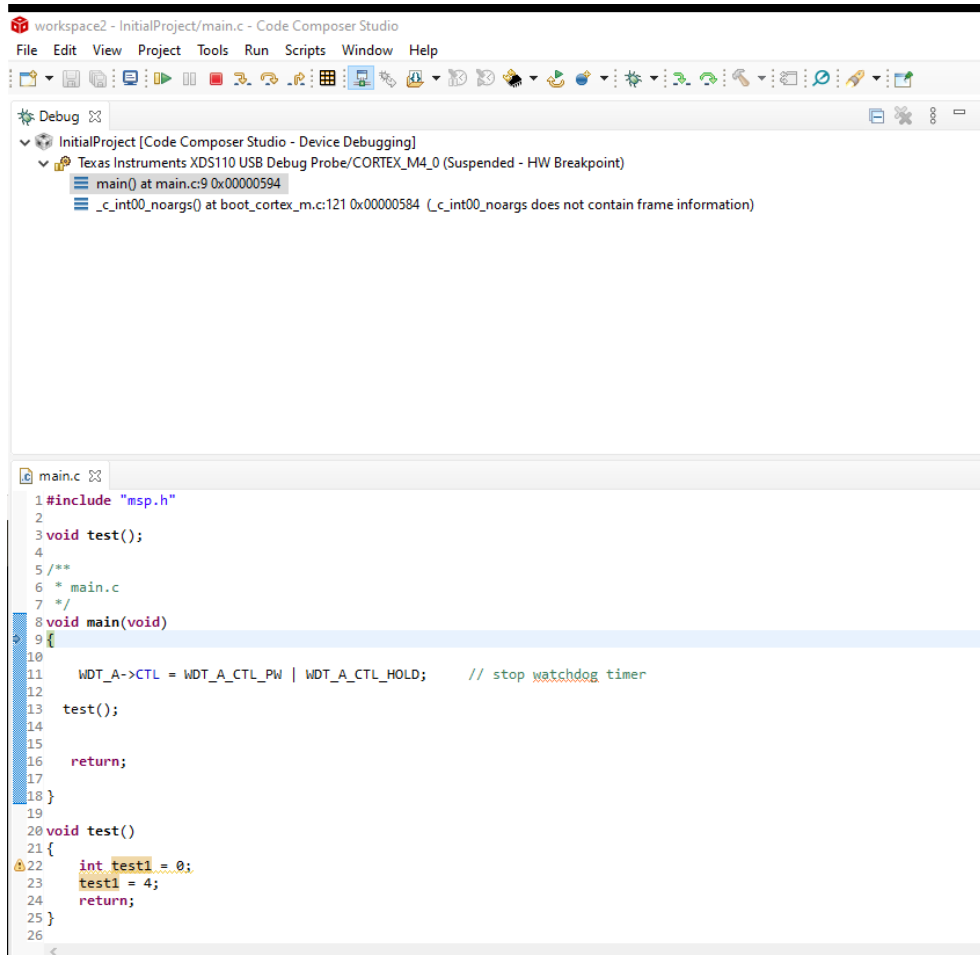
```

void test()
{
    int test1 = 0;
    test1 = 4;
    return;
}

```

The void before the test() function name means that the function will return no data. The empty parenthesis () means that no data will be passed to the function. Build the program and then select Debug As... -> Code Composer Debug Session.

You should see this:



```
workspace2 - InitialProject/main.c - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help

Debug
InitialProject [Code Composer Studio - Device Debugging]
  Texas Instruments XDS110 USB Debug Probe/CORTEX_M4_0 (Suspended - HW Breakpoint)
    main() at main.c:9 0x00000594
      _c_int00_noargs() at boot_cortex_m.c:121 0x00000584 (_c_int00_noargs does not contain frame information)

main.c
1 #include "msp.h"
2
3 void test();
4
5 /**
6  * main.c
7  */
8 void main(void)
9 {
10
11     WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer
12
13     test();
14
15     return;
16 }
17
18
19
20 void test()
21 {
22     int test1 = 0;
23     test1 = 4;
24     return;
25 }
26
```

Not select the Run->Step Over (F6) selection twice. Then select the Run->Step Into (F5) selection. You should now step into test() function.


```

    int test1 = 0;
    test1 = 3;

    test(test1);

    return;
}

void test(int test2)
{
    test2 = 4;
    return;
}

```

Compile, debug, and run the program. Then use step-wise debugging and step into *test()*_function. Notice that the value of *test2* is 3. This is a copy of the variable value *test1* used in the *main()* function.

(x)= Variables Expressions Registers			
Name	Type	Value	Location
(x)= test2	int	3	0x2000FFF0

It is very important to understand that the storage location *test1* and *test2* are at two different locations. Even if these two variables had the same name, they still would not share the same address. Now step through the function and you'll notice *variable1*'s value changes to 4:

(x)= Variables Expressions Registers			
Name	Type	Value	Location
(x)= test2	int	4	0x2000FFF0

Keep stepping until you return to *main()*. You'll notice that the value of *test1* is still 3.


```
    return test2;  
}
```

Notice that you have changed the `test()` function declaration by declaring the return type to be `int`.

```
int test(int test1);
```

This tells the compiler that you will be returning an integer from your function. Now in the calling statement you can use the assignment operator, `=`, to assign the value that returned from the function to the variable named `test1`. The memory assigned to contain `test1`'s value will be updated.

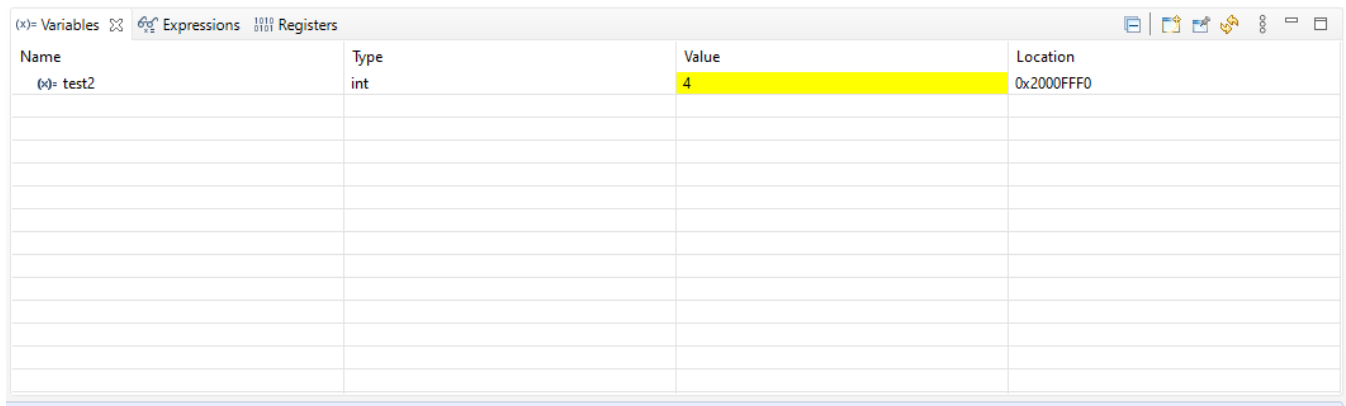
```
test1 = test(test1);
```

The final step is to change the function itself and add to the return statement the value you wish to return, in this case it is whatever is stored in the location named `test2`.

```
return test2;
```

Save `main.c` then compile, run, and debug this code.

Now step through the code. When you get to the return statement of the function you should see the `test2` value in the debug window:



The screenshot shows a debugger window with tabs for Variables, Expressions, and Registers. The Variables tab is active, displaying a table with columns for Name, Type, Value, and Location. The variable `test2` is listed with a type of `int`, a value of `4`, and a location of `0x2000FFF0`. The value `4` is highlighted in yellow.

Name	Type	Value	Location
<code>test2</code>	<code>int</code>	4	0x2000FFF0

Now step until you reach the end of the main function. Notice how the value of the memory location named `test1` has changed to 4 based on the return from the function.

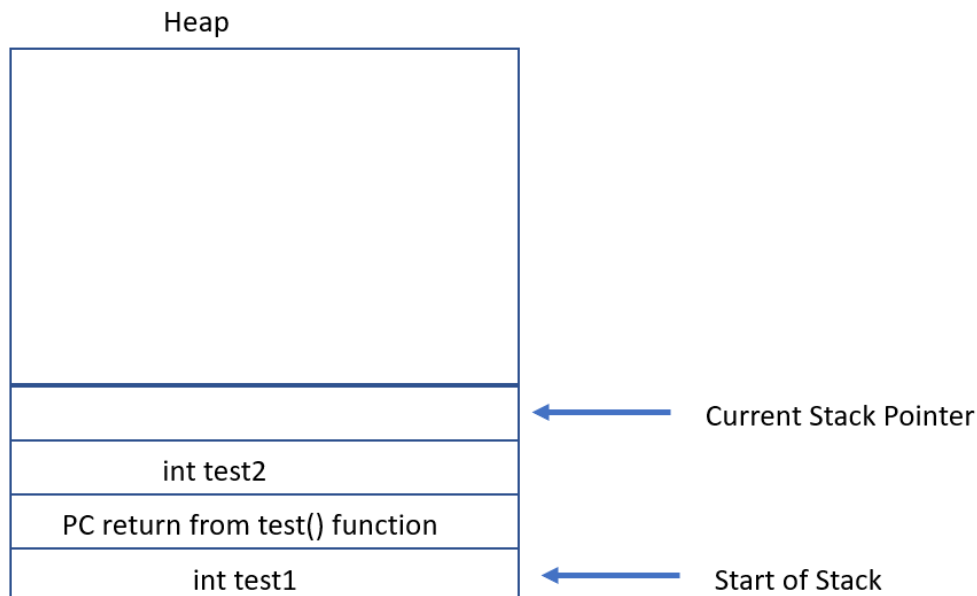

```

5 /**
6  * main.c
7  */
8 void main(void)
9 {
10
11     WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD; // stop watchdog timer
12
13     int test1 = 0;
14     test1 = 3;
15
16     test1 = test(test1);
17
18
19     return;
20 }
21
22
23 int test(int test2)
24 {
25     test2 = 4;
26     return test2;
27 }
28
29

```

Name	Type	Value	Location
test1	int	4	0x2000FFF8

Now a couple of details for using local variables. Normally a computer system has two types of memory. One is the program memory space, this memory holds the actual instructions that your CPU will execute. The other is data memory space, this memory holds values that you will use in your program. The data memory space is called the Heap, and starts as one contiguous space of memory. The compiler uses a data context called a stack to organize this space. The stack holds several pieces of information. One of the items it holds is the history of the functions called up until the current command. This is called the call stack. It also holds all of the local variables. For this example at the end of the test_function (line 19) it would look like this:



You can see the local variables. The PC return is the Program Counter address that the program needs to return to the proper point in program execution when the `test_function` is complete.

Global Variables

While the use of global variables is considered bad, programmers writing C to run on controllers and other small pieces of hardware do use them. Most often it is used to hold addresses of key hardware pieces that might be needed throughout the program. A global variable is created whenever you create a variable outside of a function, like this:

```
#include "msp.h"

int test(int test2);

int test3 = 0;

/**
 * main.c
 */
void main(void)
{
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer

    int test1 = 0;
    test1 = 3;
    test3 = 2;

    test1 = test(test1);

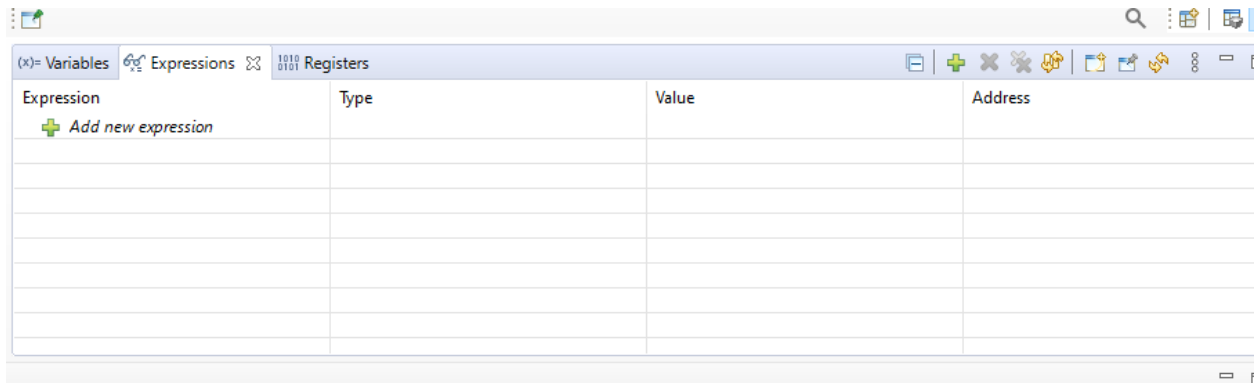
    return;
}

int test(int test2)
{
    test2 = 4;
    test3 = 3;
    return test2;
}
```

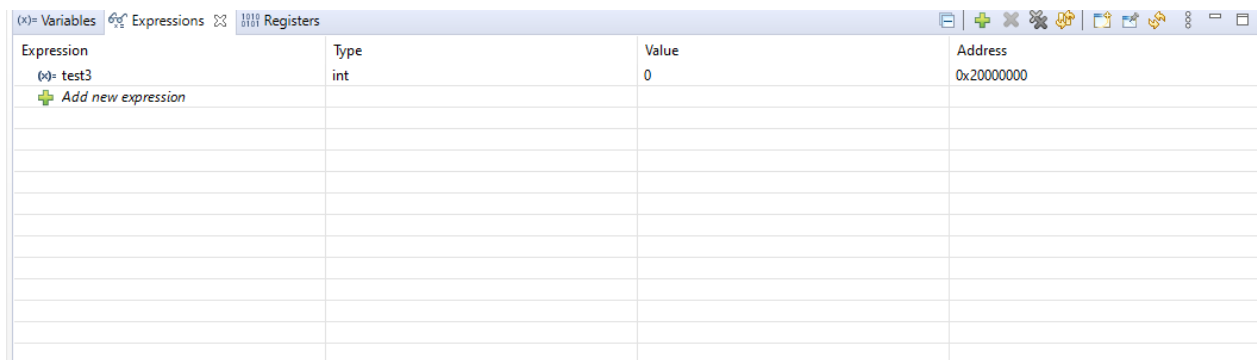
All global variables you create exist the whole time your program is running and can be accessed from anywhere in your code. You might be tempted to use global variables all the time instead of local ones, that way you wouldn't have to pass any parameters or return any values. If you make this choice, your application will have problems. Since your variables will be alive all the time, instead of only being placed on the stack when they are needed, your app will waste a lot of memory. It will also make your code buggy, crash, and hard to debug. Use global variables ONLY when you have to.

You are going to want to use stepwise debugging to see what happens to `test3` and when it changes. Unfortunately, it won't show up in your variables watch list like local variables, so you'll need to add it by hand as an expression. I'll step you through how to do this.

In the watch window, select the Expressions tab.



Now click the + sign before the Add new expression. Enter the `test3` name. You should now be able to see the value of the `test3` global variable:



After completing this, compile, run, and debug the global variable version of the program and step through it. Watch what happens to the value of the global variable `test3`.