

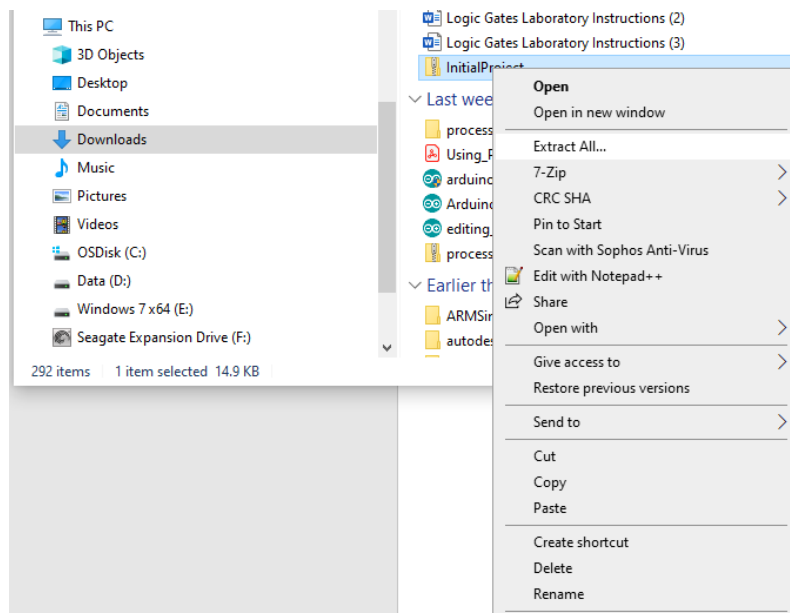
Introduction to Functions and Variables

Introduction:

For this class you will be using a simple hardware development environment to introduce you to concepts associated with C Programming. You will use this environment as it allows more access to the details of the hardware. The objective of this lab is to allow you to understand how to use functions and libraries in the C coding language.

Importing the Example Code:

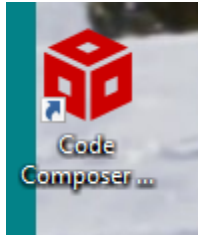
Let's import some example code to get started. To do this download the zip file InitialProject.zip from i-learn. Unzip the file by selecting the zip file and right clicking on the file name, then click Extract All... and chose where you want to extract the file. You can leave it in your Downloads directory if you like:.



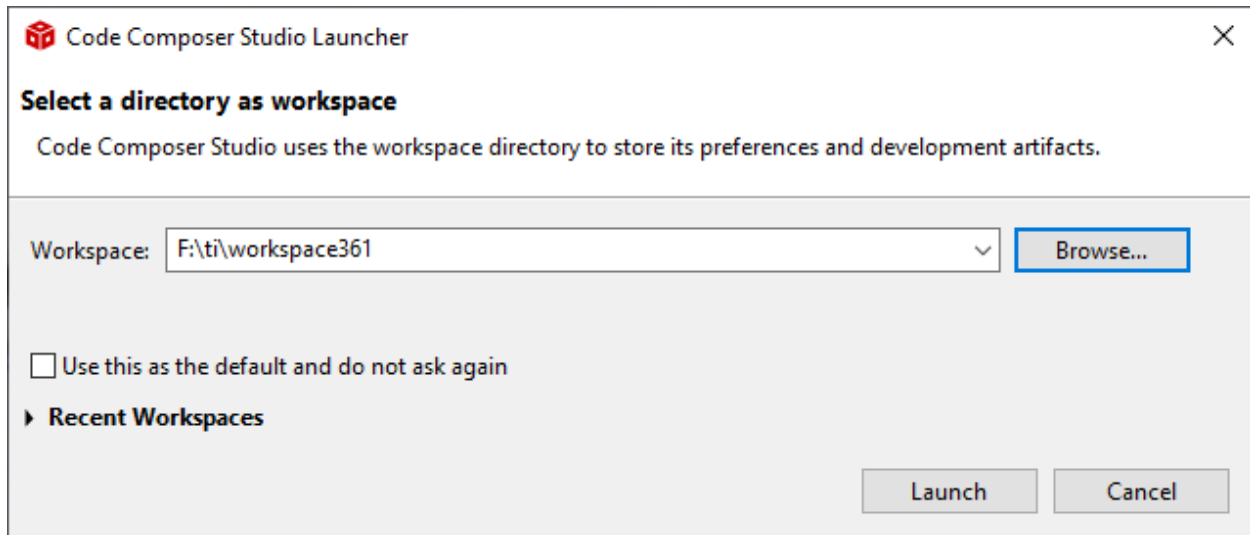
When you have the file unarchived, you can now import it into Code Composer.

Code Composer

It is already assumed that you have installed and are at least minimally familiar with the Code Composer IDE. So start Code Composer as you normally would by double clicking on the icon:

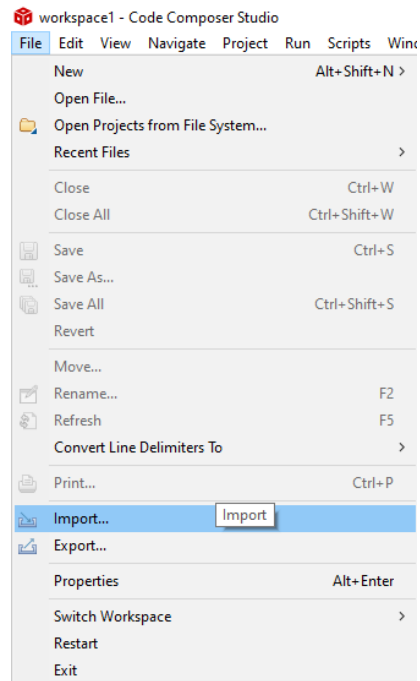


As the program starts you should be prompted for the workspace you want to use.

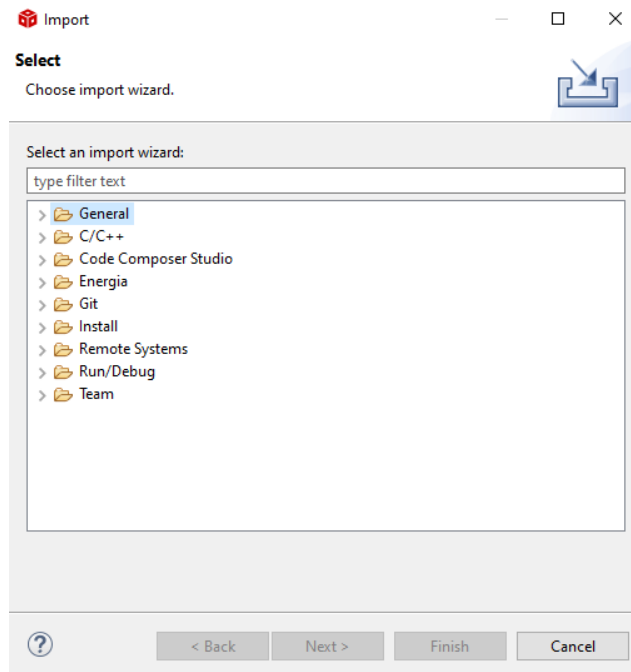


Remember this workspace is the directory where you project files will be placed. You can use several different workspaces if you want to work on very different projects. Click Launch when you have specified the work space directory (you can just use the default.)

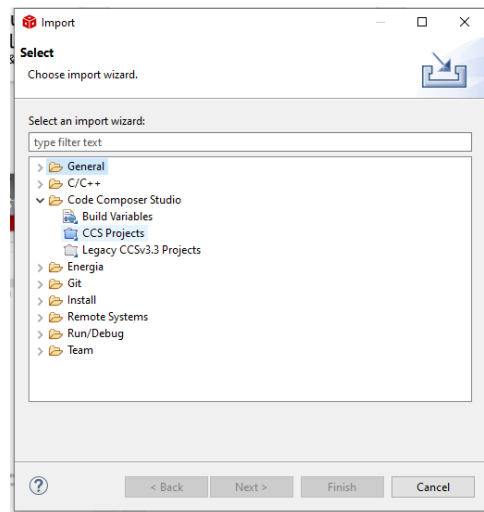
If you haven't already connect your MSP432P401R to the computer via a USB cable. Now let's import the project into Code Composer. To do this select File->Import



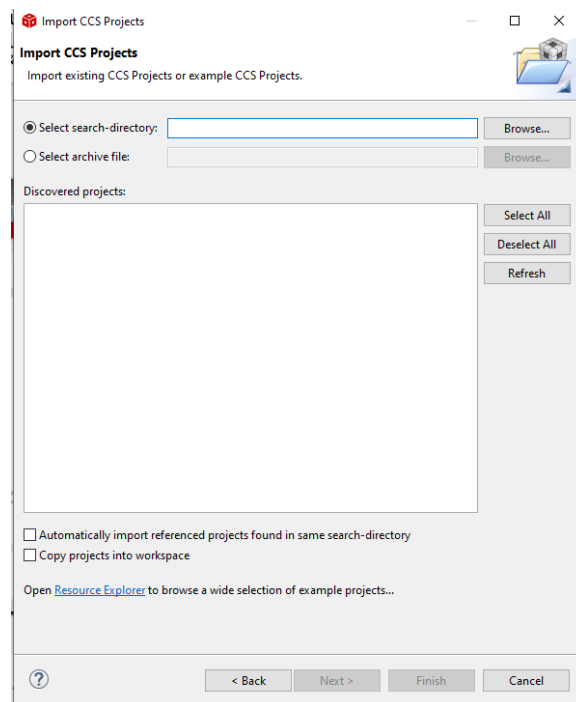
Then you should see this dialogue box:



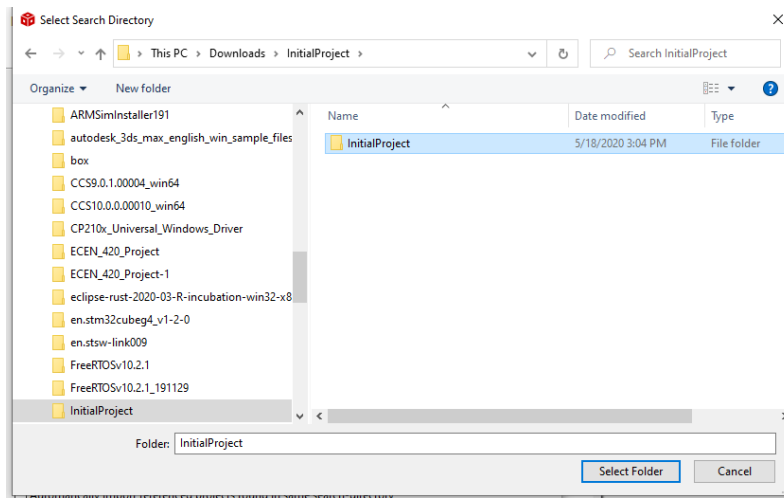
Now select Code Composer Studio, then CCS Projects:



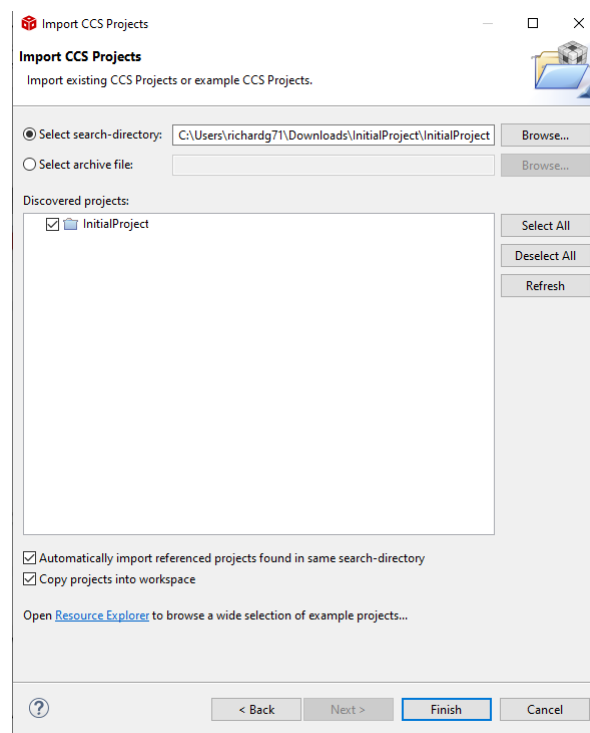
And you should see this dialogue box:



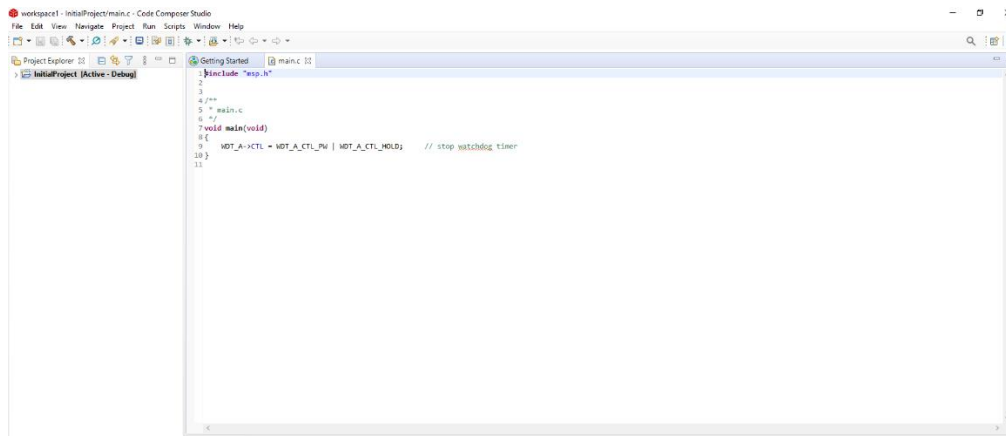
Browse to the directory where you unzip the archive. In my case it was my Downloads directory:



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. Now this will look somewhat different from a Python code set. So 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 statements in a function, in C you will use {}.

Before you start making changes, let's make sure the code compiles. Right click on the project, then click on Build Project. In the Console window you should see the following:



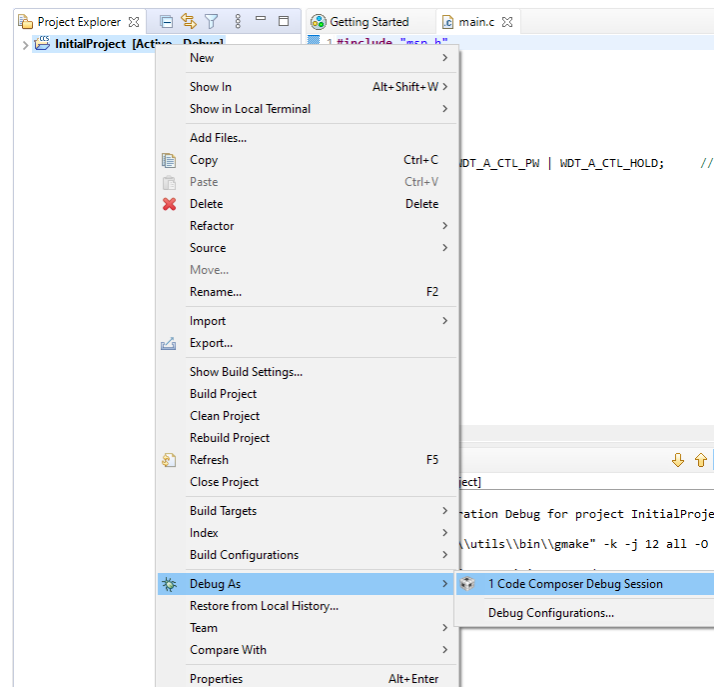
The first time you build a project you will often get suggested actions, like those shown above. They normally only occur the first time you build a new project.

The Build process is also unique to the C programming language, many languages, like Python or Java, are interpreted languages, means that there is a program running on the device that takes each command and interprets it

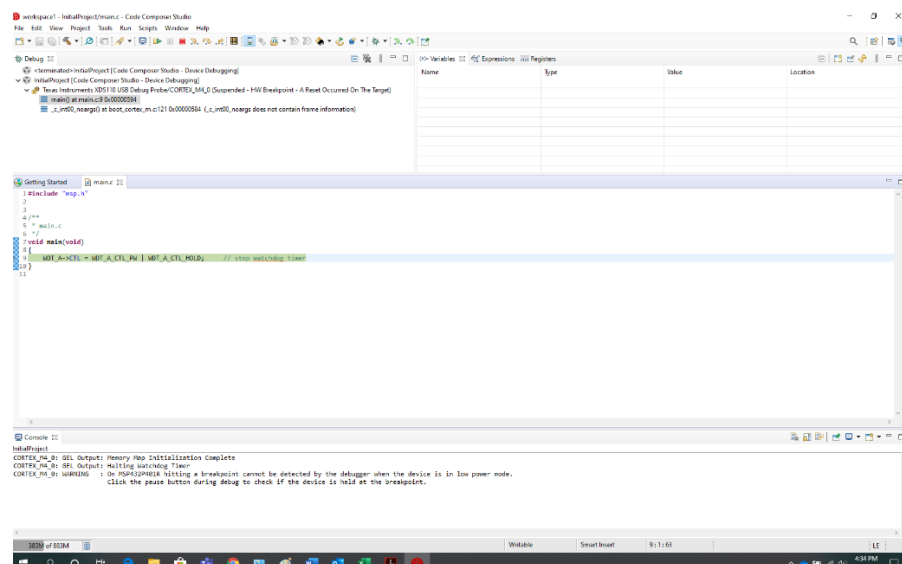
for the computer. Thus you need not only your code, but the interpreter program to be running, for your program to execute successfully.

The C programming language is a compiled language. This means that a special program called a compiler takes your program and turns it into machine code. When you want to run your program you don't need any other program, your program is in machine code in a run-able state.

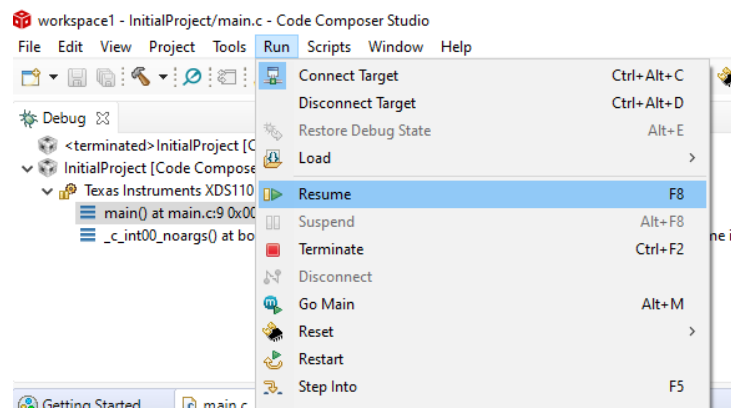
You can actually now run your program by right clicking on the project and hitting the Debug As ->Code Composer Debug Session;



The program will now be uploaded to the MSP432P401R through the USB cable. The view will also switch to the Debug View, and you should see this:



The program is on the MSP432P401R hardware, but you now need to press Resume to actually run the code:



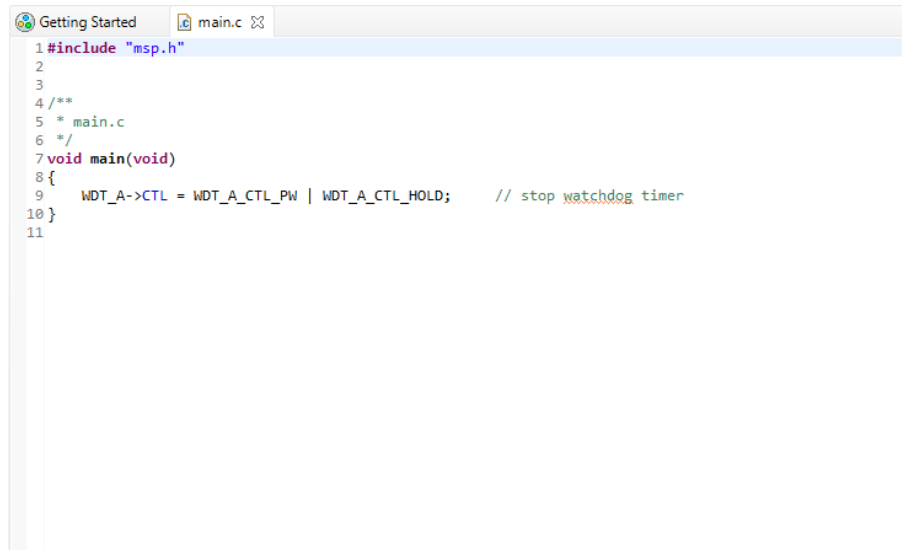
In this case the program will run, and then reach the end of the main function, and abort:



Now that you have a framework, you can add code.

Simple Functions and Local Variables

Here is the code in main.c:



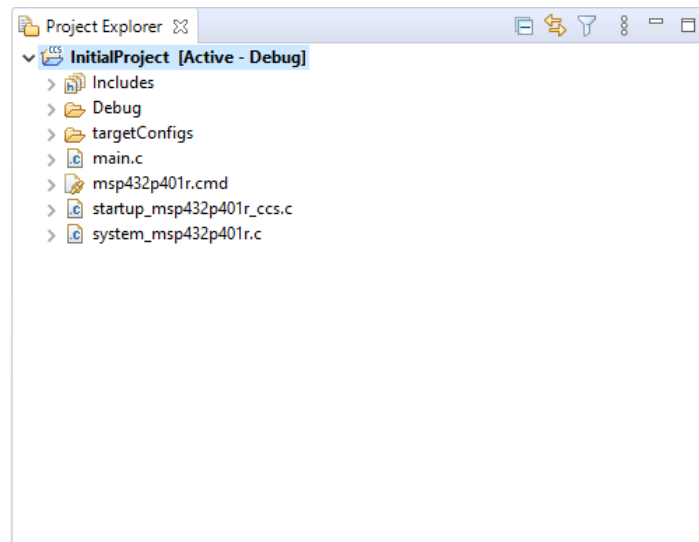
```
1 #include "msp.h"
2
3
4 /**
5  * main.c
6  */
7 void main(void)
8 {
9     WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer
10 }
11
```

As noted in the last exercise, the code will start execution with the main function. The main function is defined using a return type, then the name, then the data that will be passed into the function. In this case the void tell us that the function will not return any data, and it will also not need any data passed to it.

Something that you may not have noticed is line 1,

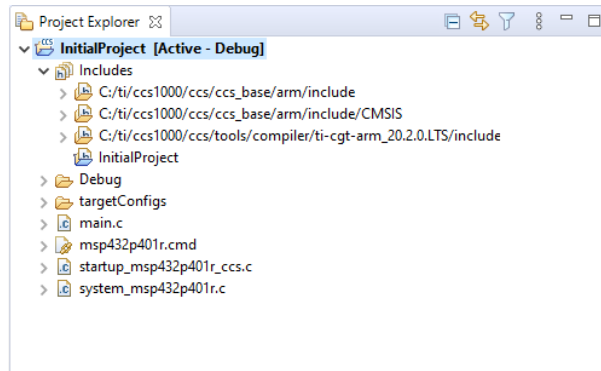
`#include "msp.h"`

This is a compiler directive to take the contents of the file “msp.h” and copy them into this file before compiling. Where does the compiler find this file. Well, let’s look at 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):

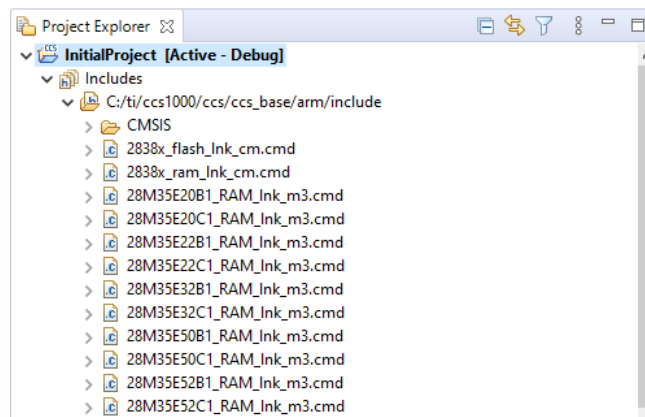


These are all the directories and files associated with your project. You see main.c, you’ve been modifying it. The msp432p401r.cmd, startup_msp432p401r_ccs.c, and system_msp432p401r.c files are configuration files used by the system to make configuration easier.

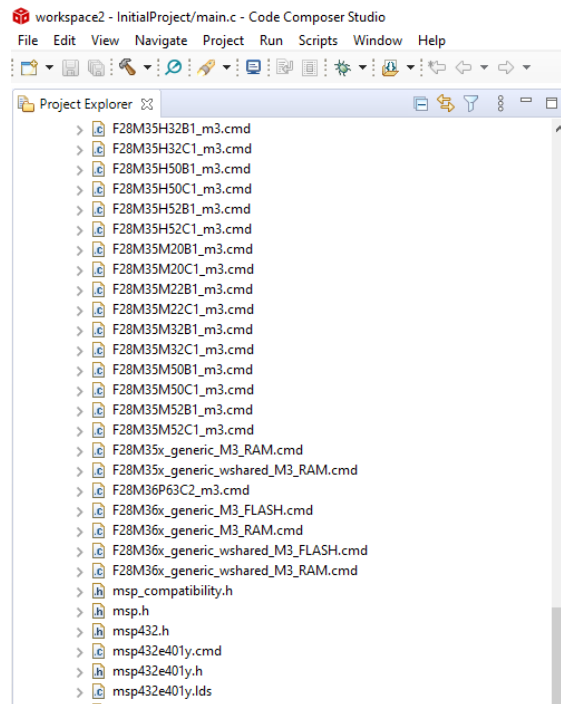
But let’s look at the top directory, the Includes directory. Expand this directory and you’ll see this:



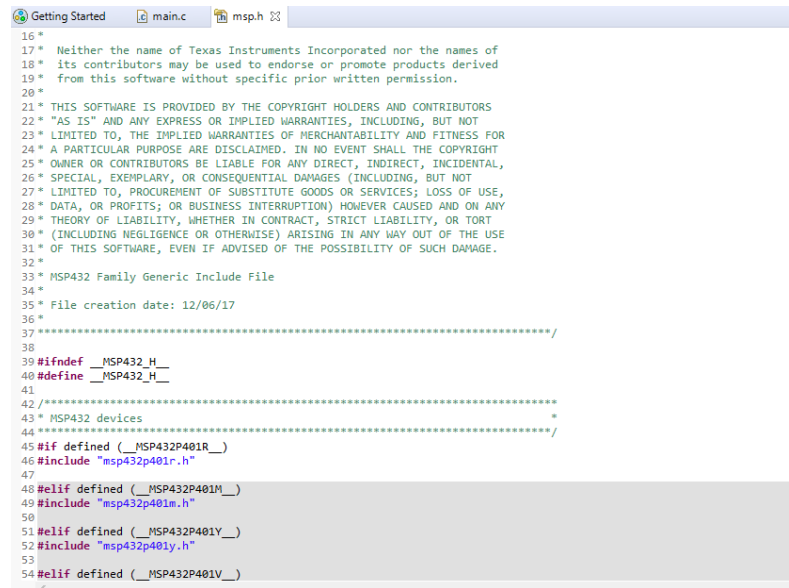
There are more directories here, include files that are available to your project. They will only be included, however, if you request them. Now let's expand the list a bit further, expand the top directory:



And now go down quite a ways and you'll see your msp.h file:



Double click on this file and you can see the file in the editor window:



```
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__)
```

This is a fairly generic file that will, depending on the hardware configuration, include the appropriate include file, in your case the msp432p401r.h file. Now let's go back to the Project Explorer and open that file:



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. When you include this file you have access to all of these definitions and can use them in your program.

Now go back to main.c.

You're going to add a function to your file. Edit your file to make it look like this:

```

#include "msp.h"

void test_function();

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

void test_function()
{
    int variable1 = 0;
    variable1 = 4;
    return;
}

```

Now let's look at what these statements do. First, the statement:

```
void test_function();
```

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_function() 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_function();
```

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:

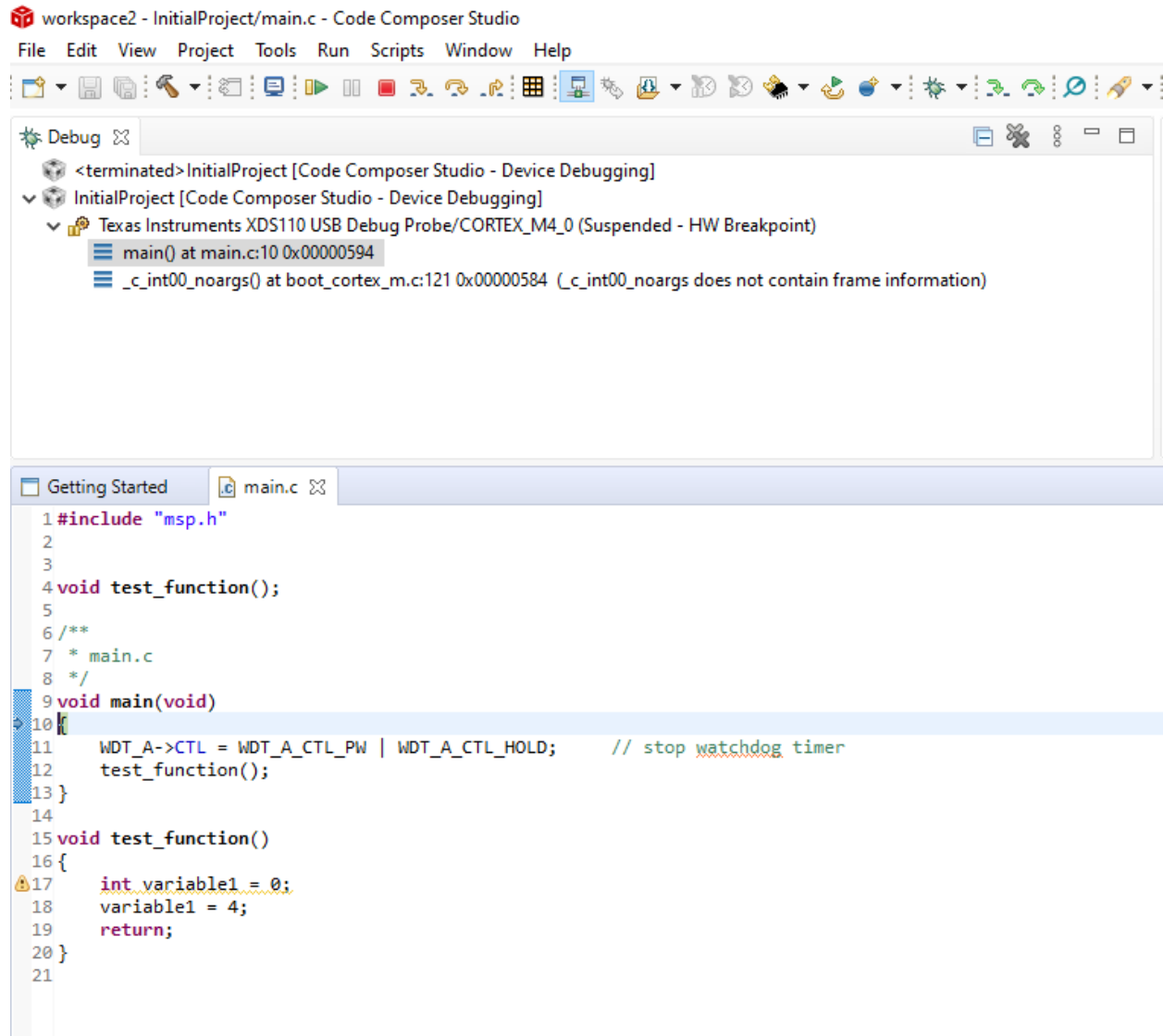
```

void test_function()
{
    int variable1 = 0;
    variable1 = 4;
    return;
}

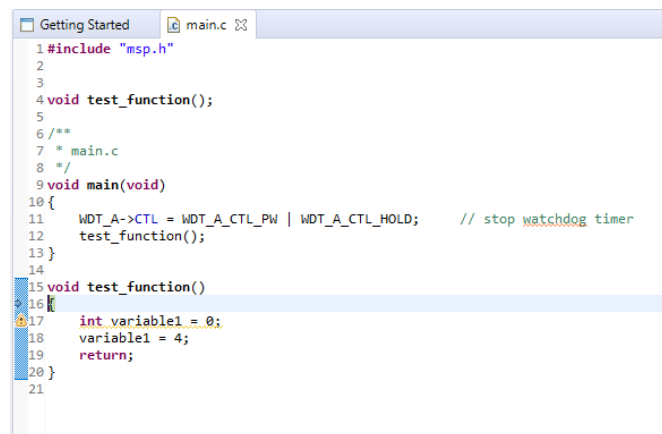
```

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. So build the program and the select Debug As... -> Code Composer Debug Session.

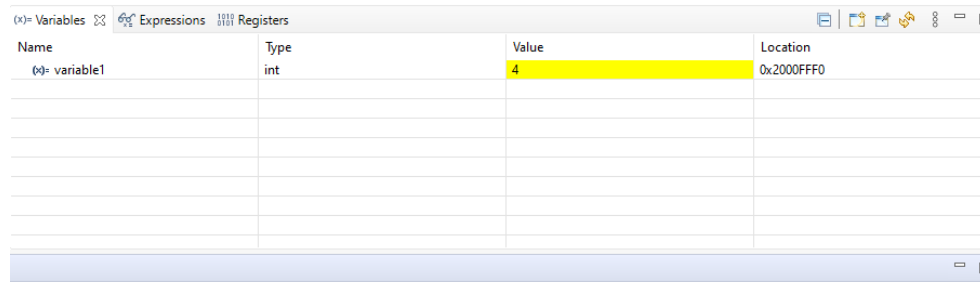
You should see this:



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



Select Run->Step Over 3 times. You should now be at the end of the function. You can see that the variable1 has changed value.



The screenshot shows the 'Variables' window in an IDE. It has tabs for 'Variables', 'Expressions', and 'Registers'. The 'Variables' tab is active, showing a table with columns: Name, Type, Value, and Location. The first row shows '(*) variable1' with Type 'int', Value '4' (highlighted in yellow), and Location '0x2000FFF0'. There are several empty rows below it.

Name	Type	Value	Location
(*) variable1	int	4	0x2000FFF0

Select Run->Step Over again and you will leave the function and return to the main() function.

Now let's talk about passing values to and return values from the function. First, let's pass a variable to the function. Change main to look like this:

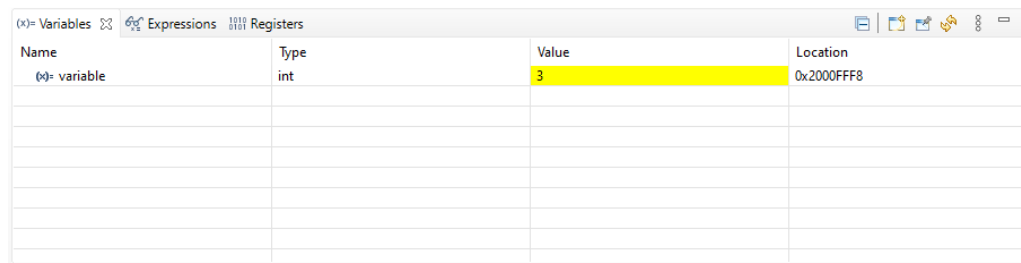
```
#include "msp.h"

void test_function(int variable1);

/**
 * main.c
 */
void main(void)
{
    int variable = 0;
    variable = 3;
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD; // stop watchdog timer
    test_function(variable);
}

void test_function(int variable1)
{
    variable1 = 4;
    return;
}
```

Compile and Debug the program. Go ahead and step the program to the test_function() call. Notice that the value of variable is 3. Also note the address of the variable:



The screenshot shows the 'Variables' window in an IDE. It has tabs for 'Variables', 'Expressions', and 'Registers'. The 'Variables' tab is active, showing a table with columns: Name, Type, Value, and Location. The first row shows '(*) variable' with Type 'int', Value '3' (highlighted in yellow), and Location '0x2000FFF8'. There are several empty rows below it.

Name	Type	Value	Location
(*) variable	int	3	0x2000FFF8

Now Step Into the function. Notice that variable1 is created and given a value of 3. It is very important to understand that the storage location variable and variable1 are two different locations. Note the address of variable1:

(x) Variables Expressions Registers			
Name	Type	Value	Location
(x) variable1	int	3	0x2000FFF0

Even if these two variables had the same name, they still would not share the same address. Now walk through the function and you'll notice variable1's value will change to 4:

(x) Variables Expressions Registers			
Name	Type	Value	Location
(x) variable1	int	4	0x2000FFF0

Now keep stepping until you return to main(). You'll notice that the value of variable is still 3.

(x) Variables Expressions Registers			
Name	Type	Value	Location
(x) variable	int	3	0x2000FFF8

It is important to note that even if these variables had the same name they wouldn't have the same address. Each time a function is called the variables in the argument list inside of the parenthesis are given their own memory addresses and the values are initialized using the values passed to them by the calling statement.

There is a way, in C, for the variables of calling function and the called function to share the same address, using pointers. We'll get to that in the later section on pointers.

The last think we need to cover with respect to variables and functions is how to return a value. This is done by declaring the type of return variable when declaring the function, and then using the return statement in the function itself to return the value. So now create the following code in main.c:

```
#include "msp.h"

int test_function(int variable1);

/**
 * main.c
 */
void main(void)
{
    int variable = 0;
    variable = 3;
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD; // stop watchdog timer
    variable = test_function(variable);
}
```

```
int test_function(int variable1)
{
    variable1 = 4;
    return variable1;
}
```

Notice that you have changed the test_function declaration by adding the int before the function name:

```
int test_function(int variable1);
```

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 take the value that is going to be returned from the function and place it in the memory location named variable.

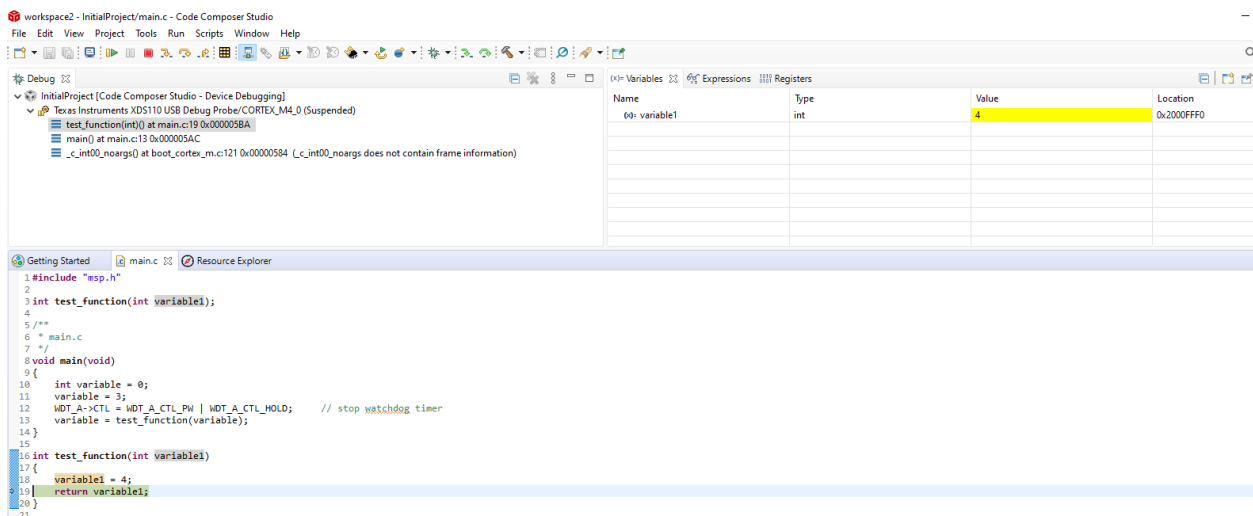
```
variable = test_function(variable);
```

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 variable1.

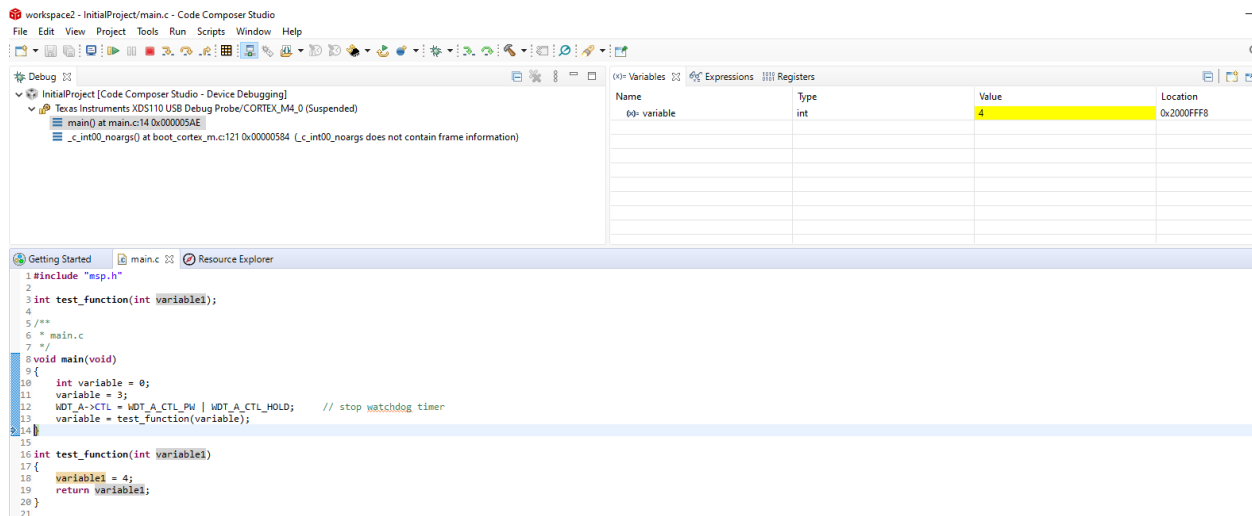
```
return variable1;
```

Go ahead and save, build, and debug this code.

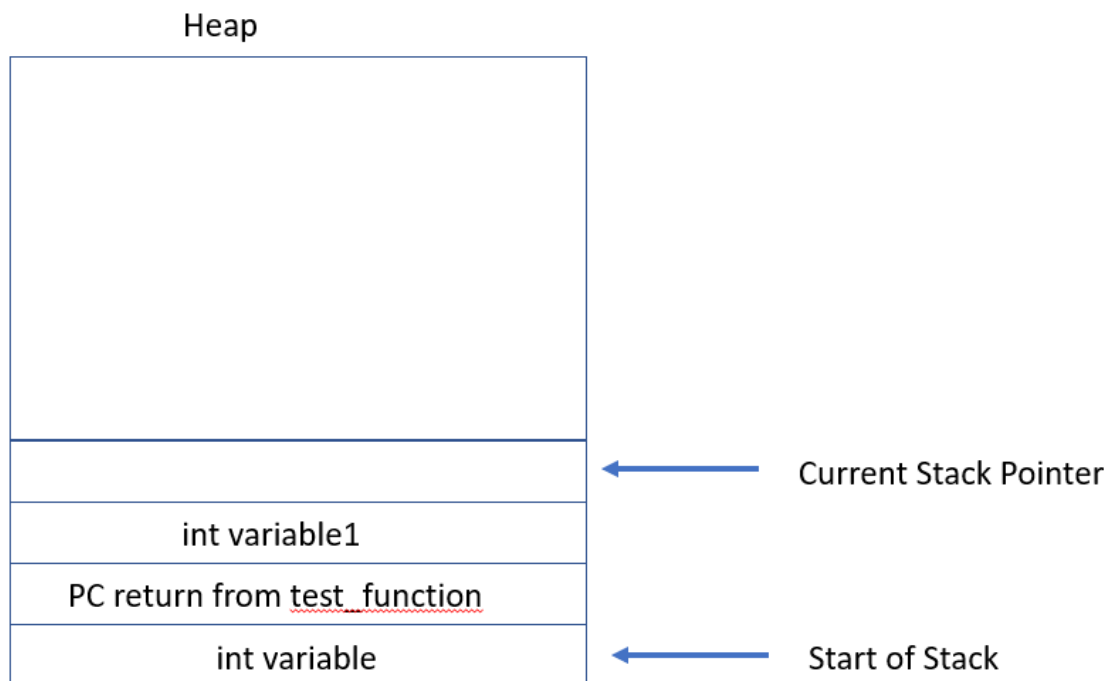
Now step through the code. When you get to the return statement of the function you should see the variable1 value in the debug window:



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



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 Variable

While the use of global variables is generally discouraged, it is often used in C programming. 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_function(int variable1);

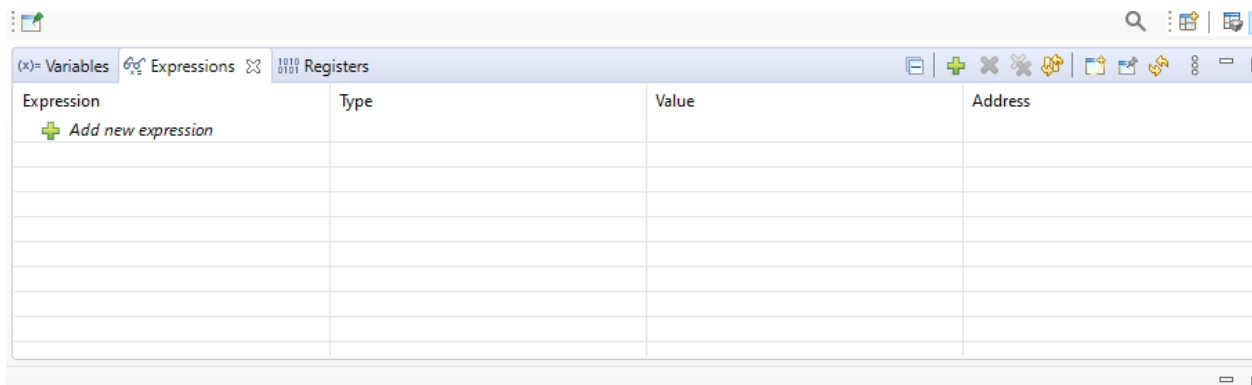
int global_variable = 0;

/**
 * main.c
 */
void main(void)
{
    int variable = 0;
    variable = 3;
    global_variable = 2;
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer
    variable = test_function(variable);
}













int test_function(int variable1)
{
    variable1 = 4;
    global_variable = 3;
    return variable1;
}
```

The `global_variable` is created by the compiler. It exists for the entirety of the execution of the program, and can be accessed by any function. Now you might be tempted to use all global variables, that way you won't have to pass or return values. However, your variables will be alive all the time, instead of only being placed on the stack when they are needed. This can cause real memory management problems.

Build this program and debug it. Now step through the program. To watch the value of your `global_variable`, because it is not on the task, you'll need to add it as an expression. In the watch window, select the Expressions tab.



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

<div>(x)= Variables  Expressions  Registers        </div>			
Expression	Type	Value	Address
 global_variable	int	0	0x20000000
 Add new expression			

As you walk through the code you'll see the value and that it changes as each of the functions access the variable.