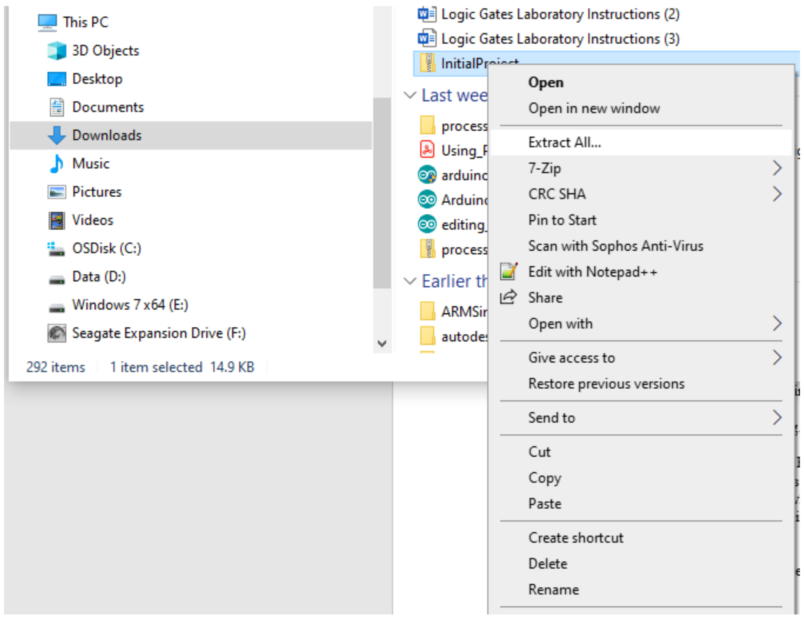
Introduction to the C language and Variables

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 typed variables 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:.

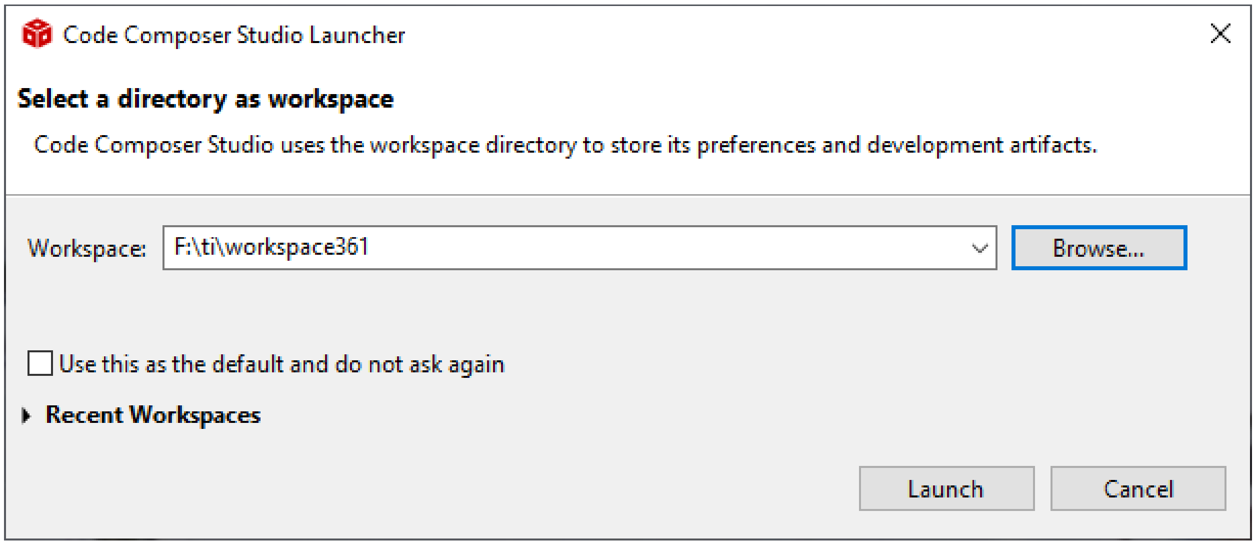


When you have the file unarchived, you can now import it into Code Composer.

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.


Code Composer Studio Launcher

Select a directory as workspace

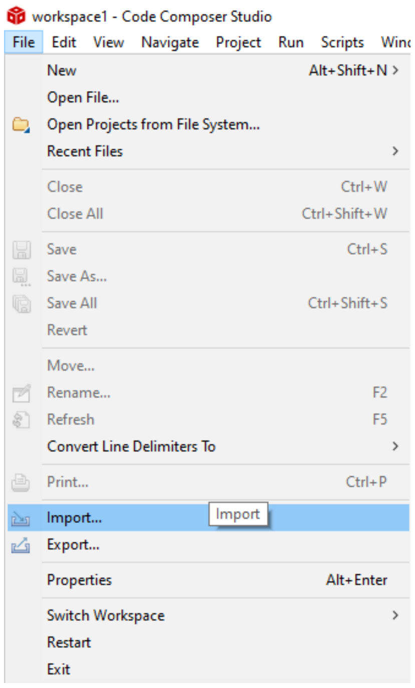
Code Composer Studio uses the workspace directory to store its preferences and development artifacts.

Workspace: F:\ti\workspace361

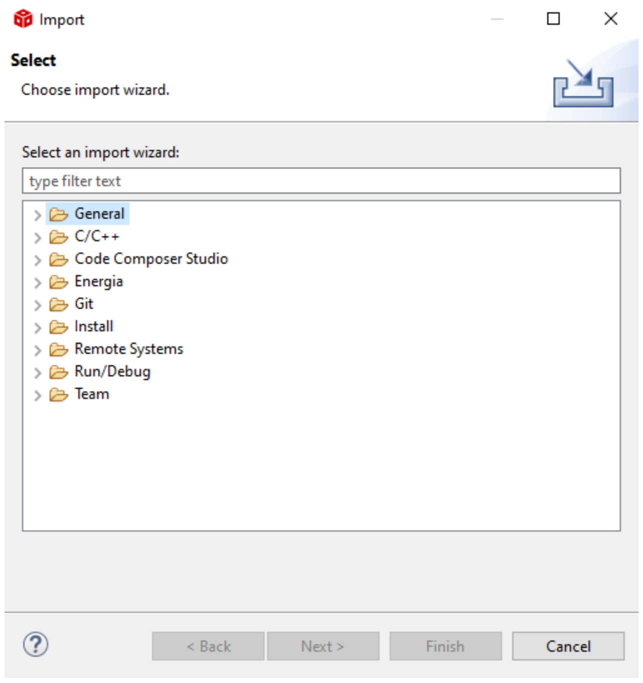
Browse...
☐ Use this as the default and do not ask again
Recent Workspaces
Launch
Cancel

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.)

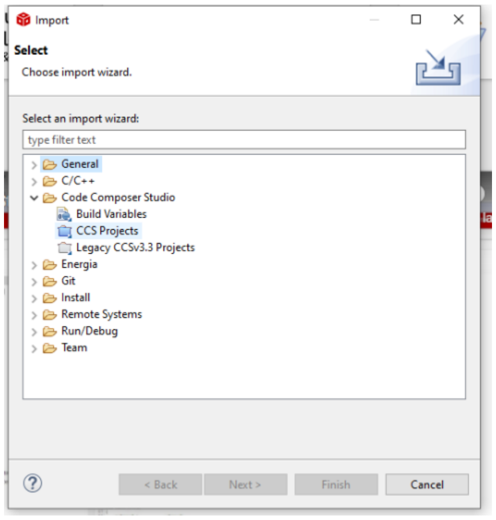
If you haven't already connect your MSP432P401R to the computer via a USB cable. Now let's import the project into Code Composer. To do this select File->Import



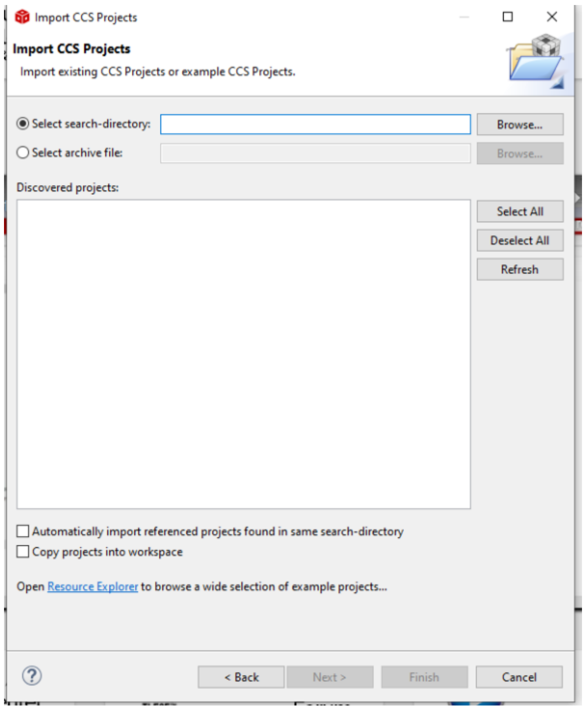
Then you should see this dialogue box:



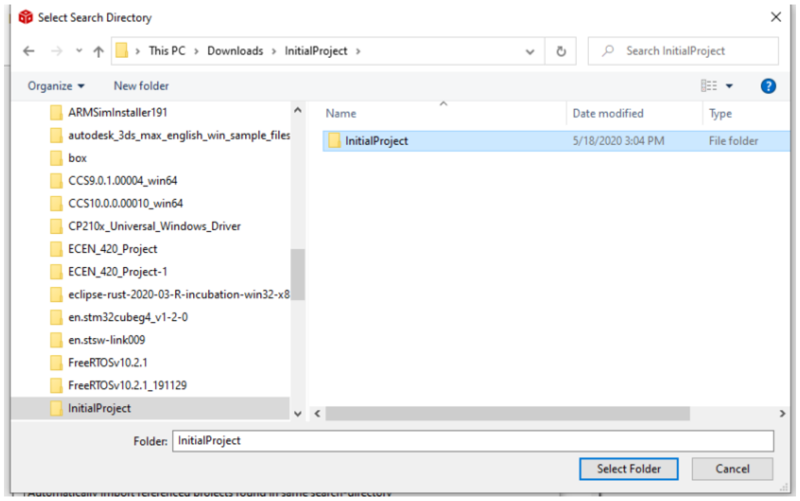
Now select Code Composer Studio, then CCS Projects:



And you should see this dialogue box:

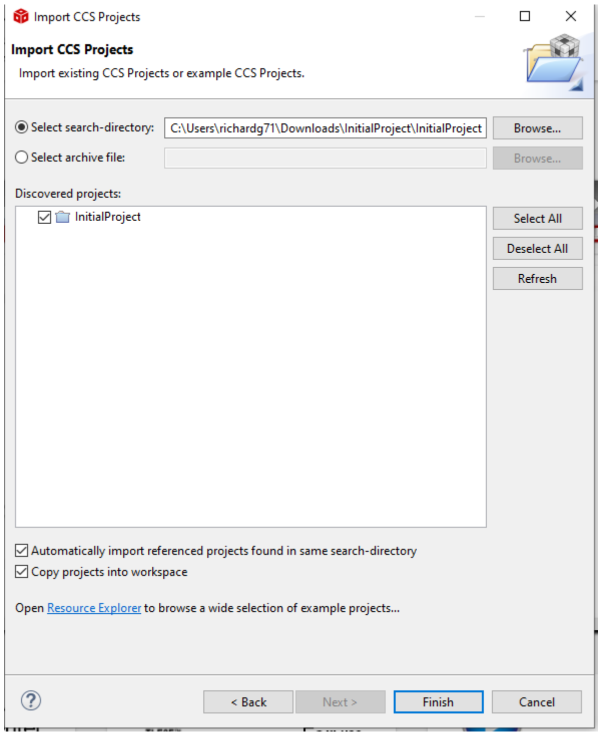

Import CCS Projects
Import CCS Projects
Import existing CCS Projects or example CCS Projects.
Select search-directory:
Browse...
Select archive file:
Browse...
Discovered projects:
Select All
Deselect All
Refresh
Automatically import referenced projects found in same search-directory
Copy projects into workspace
Open Resource Explorer to browse a wide selection of example projects...
< Back
Next >
Finish
Cancel

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



Folder: InitialProject

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



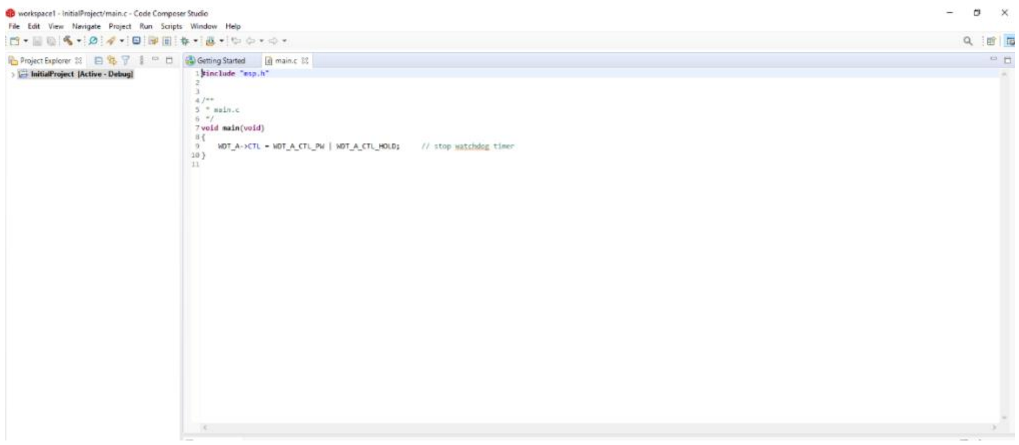
(●) Select search-directory: C:\Users\richardg71\Downloads\InitialProject\InitialProject
( ) Select archive file:

Discovered projects:
☑ InitialProject

☑ Automatically import referenced projects found in same search-directory
☑ Copy projects into workspace

Open Resource Explorer to browse a wide selection of example projects...

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


workspace1 - InitialProject/main.c - Code Composer Studio
File Edit View Navigate Project Run Scripts Window Help
Project Explorer
InitialProject [Active - Debug]
Getting Started
main.c

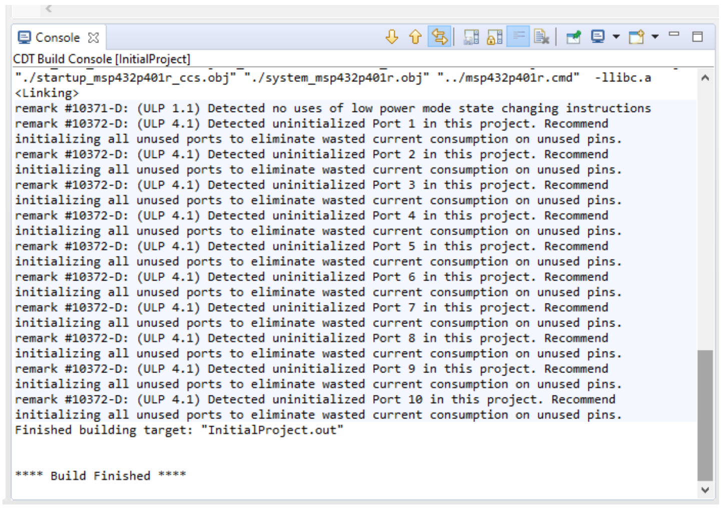
```
#include "msp.h"


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

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:



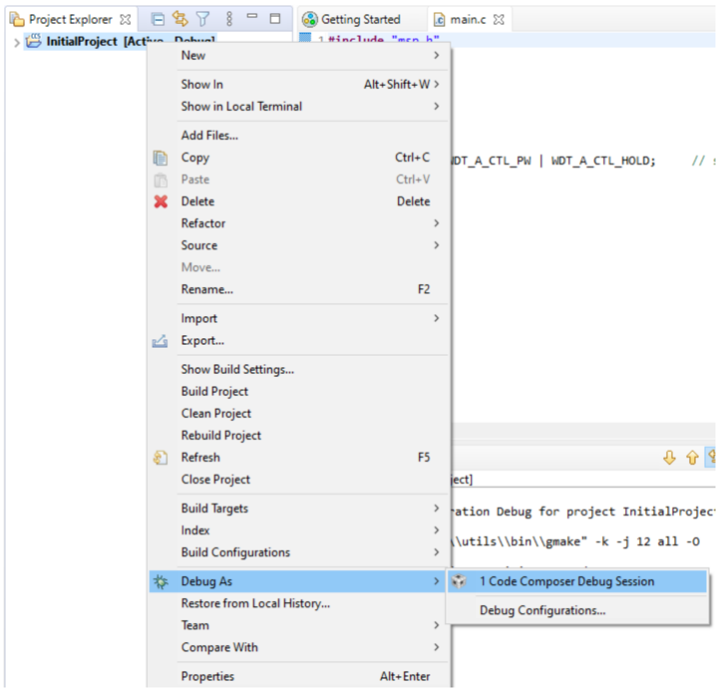
```
"./startup_msp432p401r_ccs.obj" "./system_msp432p401r.obj" "../msp432p401r.cmd"  -llibc.a
<Linking>
remark #10371-D: (ULP 1.1) Detected no uses of low power mode state changing instructions
remark #10372-D: (ULP 4.1) Detected uninitialized Port 1 in this project. Recommend
initializing all unused ports to eliminate wasted current consumption on unused pins.
remark #10372-D: (ULP 4.1) Detected uninitialized Port 2 in this project. Recommend
initializing all unused ports to eliminate wasted current consumption on unused pins.
remark #10372-D: (ULP 4.1) Detected uninitialized Port 3 in this project. Recommend
initializing all unused ports to eliminate wasted current consumption on unused pins.
remark #10372-D: (ULP 4.1) Detected uninitialized Port 4 in this project. Recommend
initializing all unused ports to eliminate wasted current consumption on unused pins.
remark #10372-D: (ULP 4.1) Detected uninitialized Port 5 in this project. Recommend
initializing all unused ports to eliminate wasted current consumption on unused pins.
remark #10372-D: (ULP 4.1) Detected uninitialized Port 6 in this project. Recommend
initializing all unused ports to eliminate wasted current consumption on unused pins.
remark #10372-D: (ULP 4.1) Detected uninitialized Port 7 in this project. Recommend
initializing all unused ports to eliminate wasted current consumption on unused pins.
remark #10372-D: (ULP 4.1) Detected uninitialized Port 8 in this project. Recommend
initializing all unused ports to eliminate wasted current consumption on unused pins.
remark #10372-D: (ULP 4.1) Detected uninitialized Port 9 in this project. Recommend
initializing all unused ports to eliminate wasted current consumption on unused pins.
remark #10372-D: (ULP 4.1) Detected uninitialized Port 10 in this project. Recommend
initializing all unused ports to eliminate wasted current consumption on unused pins.
Finished building target: "InitialProject.out"


**** Build Finished ****
```

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.

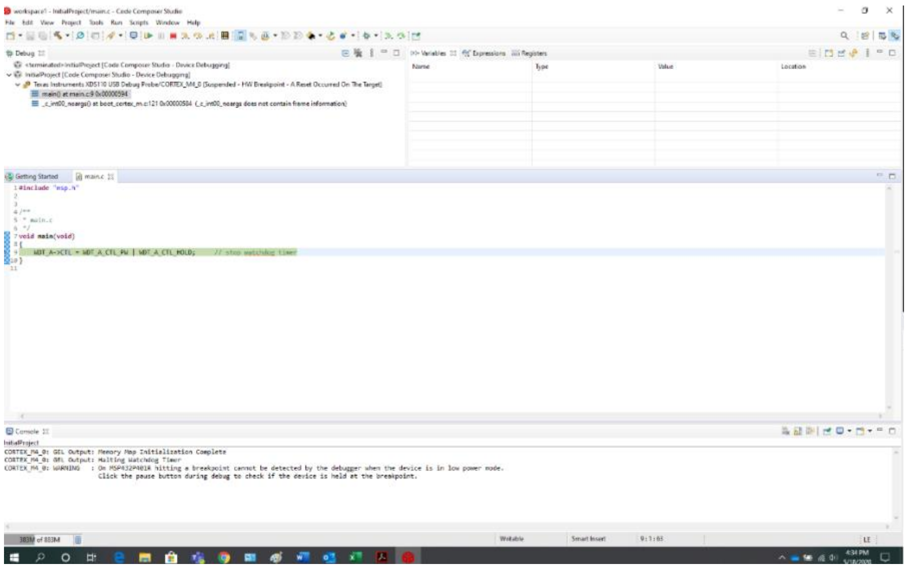
The Build process is something that all compiled languages, like C, must do prior to shipping or running. Some languages, like Python or JavaScript, are interpreted languages, means that there is a program running on the device that reads each command from the script file and interprets it for the computer. Thus you need not only your code, but the interpreter program to be running, for your program to execute successfully.

Since C, like many other languages, is a compiled programming language, 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:

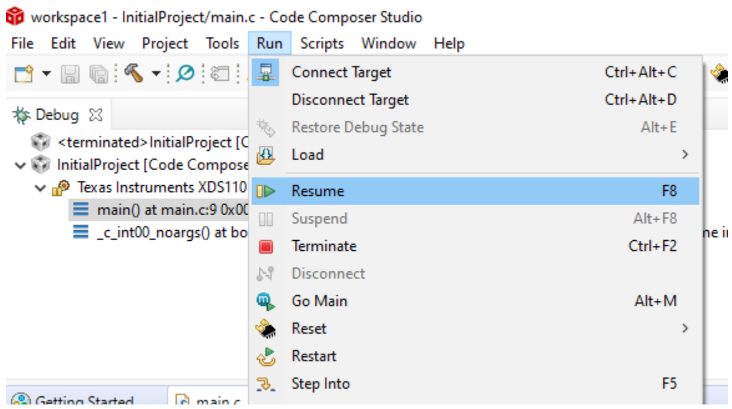

workspace1 - InitialProject/main.c - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Debug
Variables
Expressions
Registers
Name
Type
Value
Location
Getting Started
main.c
Console
Writable
Smart Insert

```
<terminated>InitialProject [Code Composer Studio - Device Debugging]
InitialProject [Code Composer Studio - Device Debugging]
  Texas Instruments XDS110 USB Debug Probe/CORTEX_M4_0 (Suspended - HW Breakpoint - A Reset Occurred On The Target)
    main() at main.c:9 0x00000594
    _c_int00_noargs() at boot_cortex_m.c:121 0x00000584  (_c_int00_noargs does not contain frame information)
```

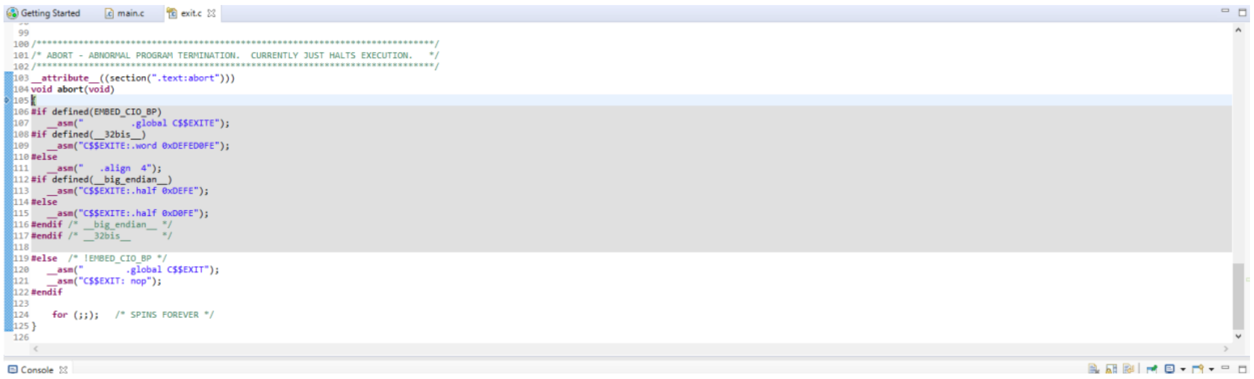
```c
#include "msp.h"


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

```
InitialProject
CORTEX_M4_0: GEL Output: Memory Map Initialization Complete
CORTEX_M4_0: GEL Output: Halting Watchdog Timer
CORTEX_M4_0: WARNING   : On MSP432P401R hitting a breakpoint cannot be detected by the debugger when the device is in low power mode.
                         Click the pause button during debug to check if the device is held at the breakpoint.
```

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


workspace1 - InitialProject/main.c - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Debug
<terminated>InitialProject [C
InitialProject [Code Compose
Texas Instruments XDS110
main() at main.c:9 0x00
_c_int00_noargs() at bo
Connect Target Ctrl+Alt+C
Disconnect Target Ctrl+Alt+D
Restore Debug State Alt+E
Load
Resume F8
Suspend Alt+F8
Terminate Ctrl+F2
Disconnect
Go Main Alt+M
Reset
Restart
Step Into F5
Getting Started
main.c

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

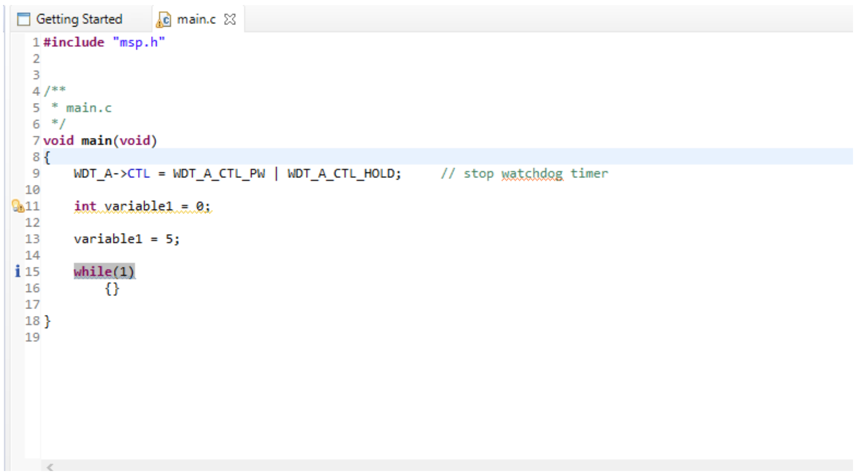


```
/*****************************************************************************/
/* ABORT - ABNORMAL PROGRAM TERMINATION.  CURRENTLY JUST HALTS EXECUTION.   */
/*****************************************************************************/
__attribute__((section(".text:abort")))
void abort(void)
{
#if defined(EMBED_CIO_BP)
   __asm("        .global C$$EXITE");
#if defined(__32bis__)
   __asm("C$$EXITE:.word 0xDEFED0FE");
#else
   __asm("   .align  4");
#if defined(__big_endian__)
   __asm("C$$EXITE:.half 0xDEFE");
#else
   __asm("C$$EXITE:.half 0xD0FE");
#endif /* __big_endian__ */
#endif /* __32bis__      */

#else  /* !EMBED_CIO_BP */
   __asm("       .global C$$EXIT");
   __asm("C$$EXIT: nop");
#endif

    for (;;);   /* SPINS FOREVER */
}
```

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

Simple Typed Variables

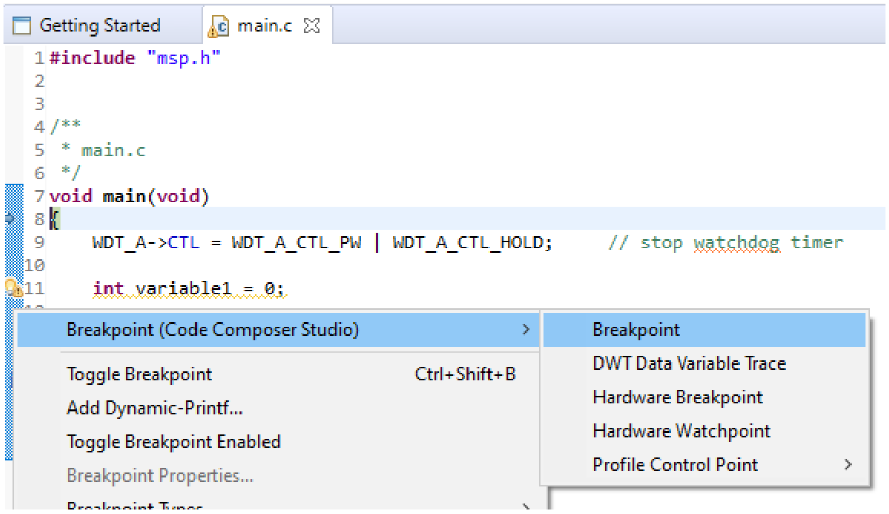
So let’s understand typed variables in the C coding language. Change your main.c add a variable, like this:



```
#include "msp.h"


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

    int variable1 = 0;

    variable1 = 5;

    while(1)
        {}

}
```

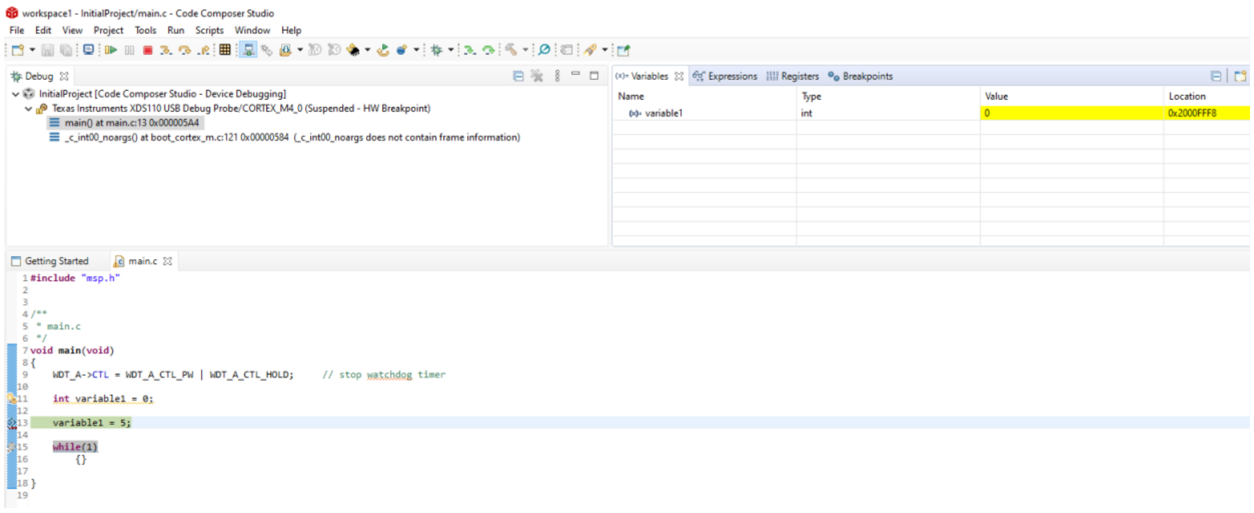
By the way, you've also added an infinite while loop, just so the program keeps running forever, and not reach the abort code. The declaration for any variable works like this; first you detail the type, then the name of the variable, and then, optionally, you'll can initialize the variable to a value.

Compile and load the code. Before running the code, however, let's put some break points in the code to see what is happening with our variable. To add a break point, after you've reached the debug view, go to the start of line 13, then right click and you should see the dialogue box pop up to add a breakpoint:



```
 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     int variable1 = 0;
```

This will stop the execution of the code just before line 13. Now go ahead and hit resume. Your code should run, but stop on line 13. Code Composer provides a way for you to look at the value associated with your variable. Up in the right hand corner of the display you can see the value of the variable:



workspace1 - InitialProject/main.c - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Debug
Variables
Expressions
Registers
Breakpoints
Getting Started
main.c

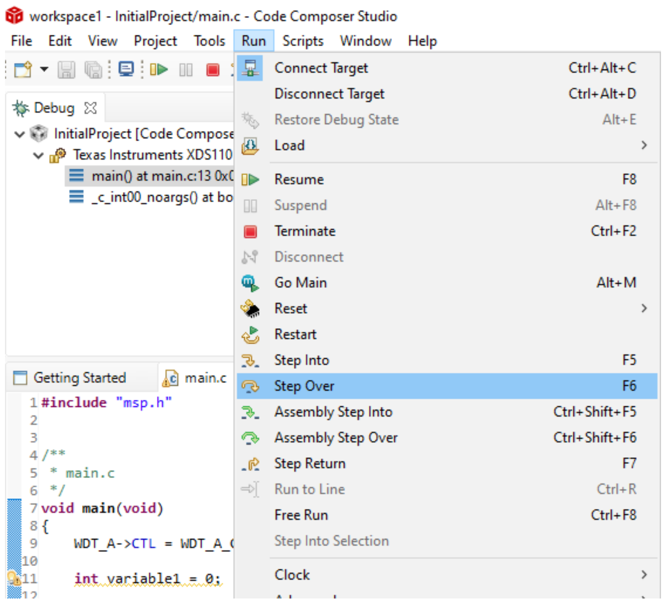
```
InitialProject [Code Composer Studio - Device Debugging]
  Texas Instruments XDS110 USB Debug Probe/CORTEX_M4_0 (Suspended - HW Breakpoint)
    main() at main.c:13 0x000005A4
    _c_int00_noargs() at boot_cortex_m.c:121 0x00000584  (_c_int00_noargs does not contain frame information)
```

Name	Type	Value	Location
variable1	int	0	0x2000FFF8

```
 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     int variable1 = 0;
12
13     variable1 = 5;
14
15     while(1)
16         {}
17
18 }
19
```

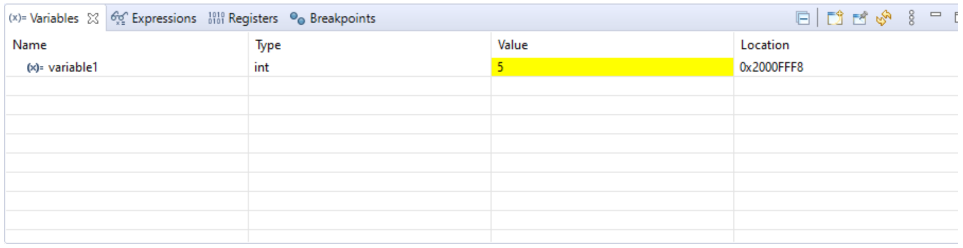
In this case you have a variable named variable1, it is of type int, it's current value is 0, and its Location is 0x2000FFF8 (your actual value may be different depending on where Code Composer has decided to allocate memory for your variable.)

Now use the Step Over command to step over line 13:


workspace1 - InitialProject/main.c - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Debug
InitialProject [Code Compose
Texas Instruments XDS110
main() at main.c:13 0x0
_c_int00_noargs() at bo
Connect Target Ctrl+Alt+C
Disconnect Target Ctrl+Alt+D
Restore Debug State Alt+E
Load
Resume F8
Suspend Alt+F8
Terminate Ctrl+F2
Disconnect
Go Main Alt+M
Reset
Restart
Step Into F5
Step Over F6
Assembly Step Into Ctrl+Shift+F5
Assembly Step Over Ctrl+Shift+F6
Step Return F7
Run to Line Ctrl+R
Free Run Ctrl+F8
Step Into Selection
Clock
Getting Started
main.c

```
1 #include "msp.h"
2
3
4 /**
5  * main.c
6  */
7 void main(void)
8 {
9     WDT_A->CTL = WDT_A_(
10
11     int variable1 = 0;
12
```

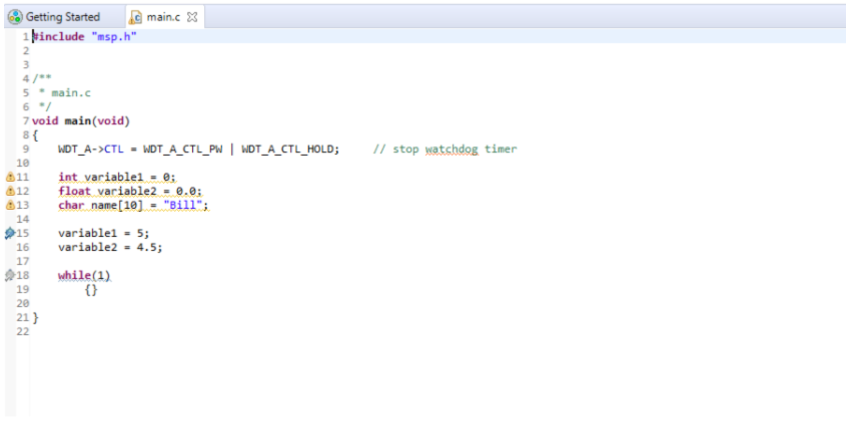
Now you should see this:



Name	Type	Value	Location
(x)= variable1	int	5	0x2000FFF8

Notice the value has changed to 5.

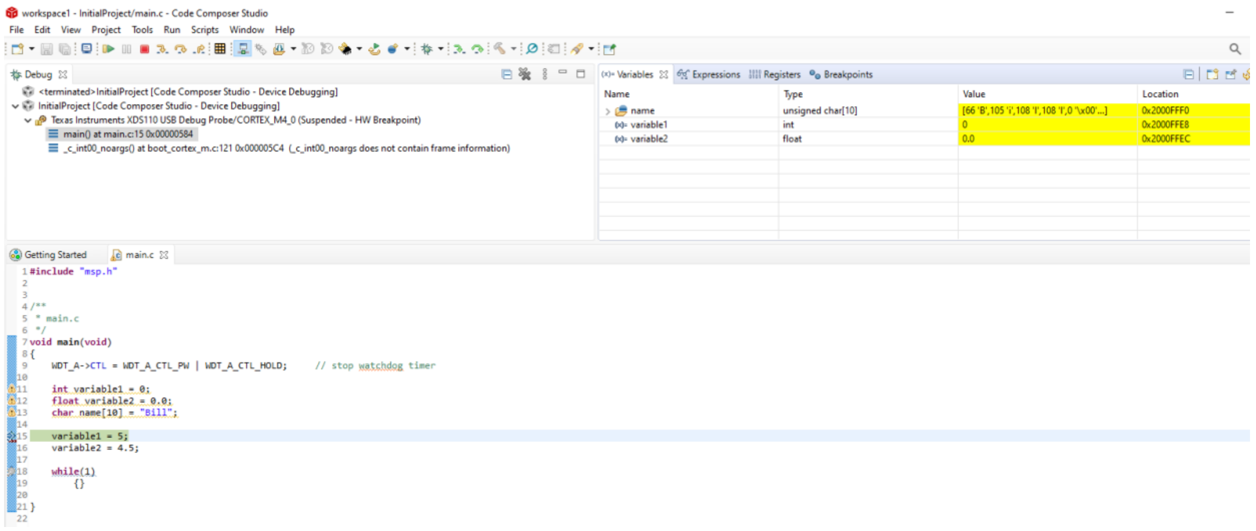
Let’s explore some other variable types. Change the code in main.c to look like this:



```
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     int variable1 = 0;
12     float variable2 = 0.0;
13     char name[10] = "Bill";
14
15     variable1 = 5;
16     variable2 = 4.5;
17
18     while(1)
19         {}
20
21 }
22
```

This introduces two new types of variable, a float and character array. Floats hold small values that have a decimal point. Character arrays hold string values. In this case you've allocated up to 10 characters for the *name* array. It is important to note that the NULL character, or the 0 value, must always be at the end of an array of characters, so if you want to store names of up to 10 characters in length, you'd actually need to allocate 11 characters for the array.

Now compile and upload the code. If you haven't changed it your breakpoint should still be at the *variable = 5;* statement, soResume the code and it should stop at that line. Then you should see something similar to this:


workspace1 - InitialProject/main.c - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Debug
(x)= Variables
Expressions
Registers
Breakpoints
Getting Started
main.c

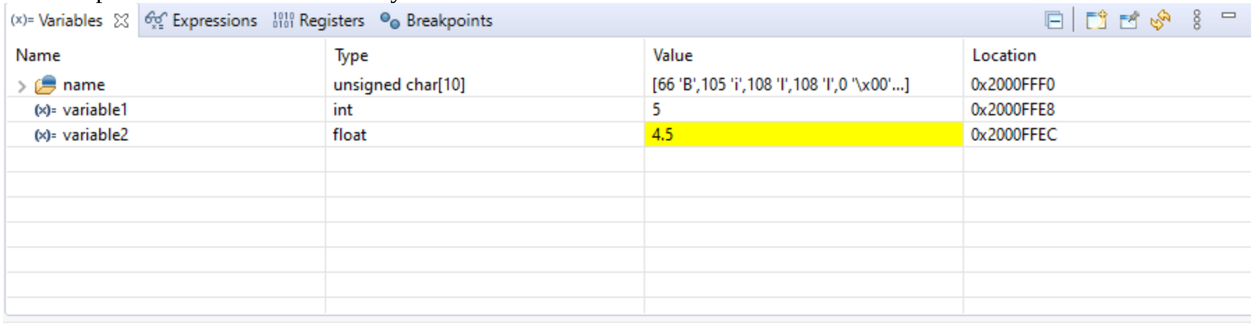
```
<terminated>InitialProject [Code Composer Studio - Device Debugging]
InitialProject [Code Composer Studio - Device Debugging]
  Texas Instruments XDS110 USB Debug Probe/CORTEX_M4_0 (Suspended - HW Breakpoint)
    main() at main.c:15 0x00000584
    _c_int00_noargs() at boot_cortex_m.c:121 0x000005C4  (_c_int00_noargs does not contain frame information)
```

Name	Type	Value	Location
name	unsigned char[10]	[66 'B',105 'i',108 'l',108 'l',0 '\x00'...]	0x2000FFF0
variable1	int	0	0x2000FFE8
variable2	float	0.0	0x2000FFEC

```
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     int variable1 = 0;
12     float variable2 = 0.0;
13     char name[10] = "Bill";
14
15     variable1 = 5;
16     variable2 = 4.5;
17
18     while(1)
19         {}
20
21 }
22
```

Notice all of our variables are shown in the Variables space. Each has a type, and an address (See the Location column in the table). If you look closely, you'll see that each character in the name array is stored as a numeric value as represented by the ASCII table. You'll also note that after the last "l" of the name is a NULL character to indicate that this particular name stops with the fourth character.

Now Step Over lines 15 and 16 and you should see this:

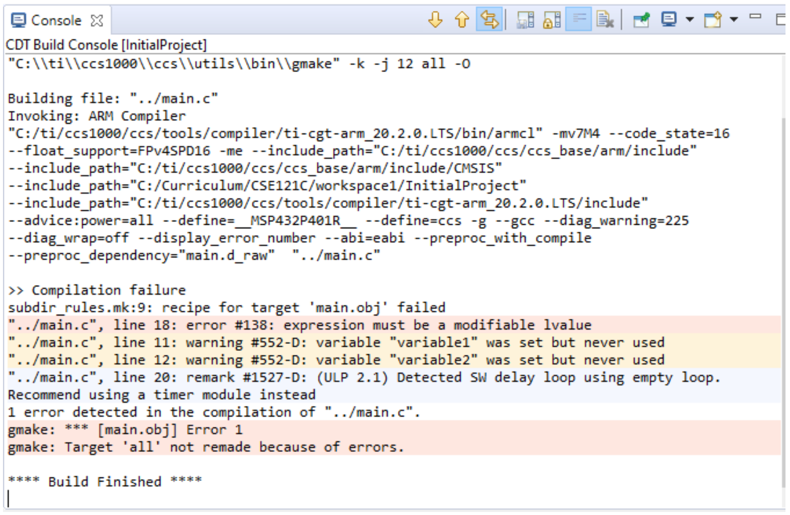


Name	Type	Value	Location
name	unsigned char[10]	[66 'B',105 'i',108 'l',108 'l',0 '\x00'...]	0x2000FFF0
variable1	int	5	0x2000FFE8
variable2	float	4.5	0x2000FFEC

Now the obvious question is why didn't you add a statement like this:

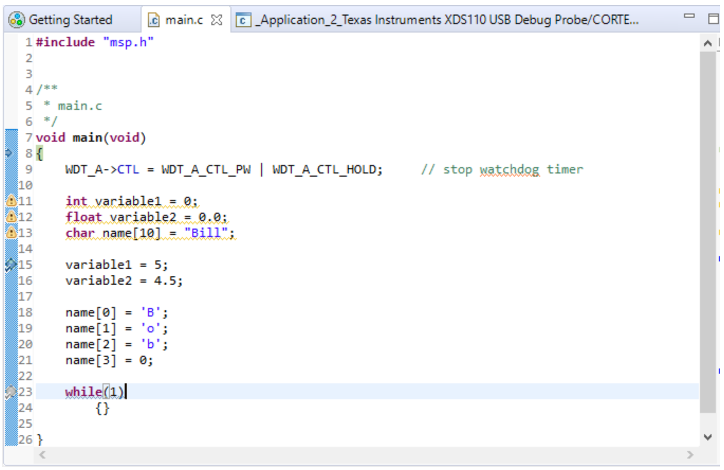
name = "Bob";

And the answer is this would have caused a compile error, like this:

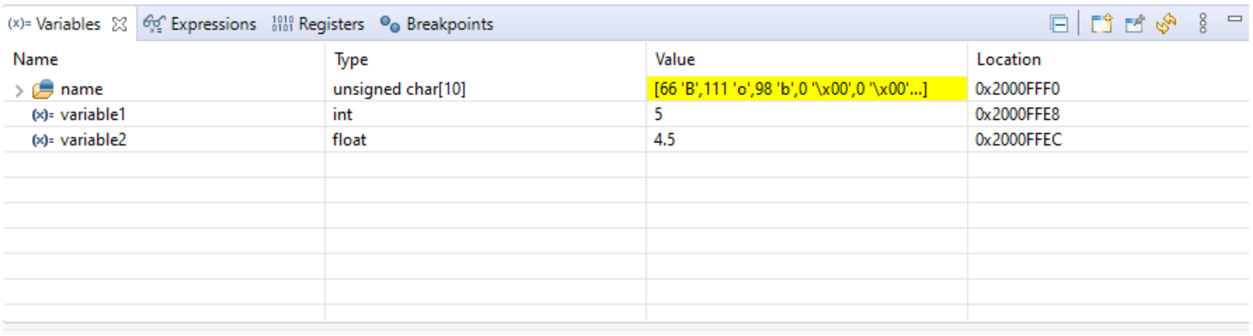

CDT Build Console [InitialProject]

```
"C:\\ti\\ccs1000\\ccs\\utils\\bin\\gmake" -k -j 12 all -O

Building file: "../main.c"
Invoking: ARM Compiler
"C:/ti/ccs1000/ccs/tools/compiler/ti-cgt-arm_20.2.0.LTS/bin/armcl" -mv7M4 --code_state=16
--float_support=FPv4SPD16 -me --include_path="C:/ti/ccs1000/ccs/ccs_base/arm/include"
--include_path="C:/ti/ccs1000/ccs/ccs_base/arm/include/CMSIS"
--include_path="C:/Curriculum/CSE121C/workspace1/InitialProject"
--include_path="C:/ti/ccs1000/ccs/tools/compiler/ti-cgt-arm_20.2.0.LTS/include"
--advice:power=all --define=__MSP432P401R__ --define=ccs -g --gcc --diag_warning=225
--diag_wrap=off --display_error_number --abi=eabi --preproc_with_compile
--preproc_dependency="main.d_raw"  "../main.c"

>> Compilation failure
subdir_rules.mk:9: recipe for target 'main.obj' failed
"../main.c", line 18: error #138: expression must be a modifiable lvalue
"../main.c", line 11: warning #552-D: variable "variable1" was set but never used
"../main.c", line 12: warning #552-D: variable "variable2" was set but never used
"../main.c", line 20: remark #1527-D: (ULP 2.1) Detected SW delay loop using empty loop.
Recommend using a timer module instead
1 error detected in the compilation of "../main.c".
gmake: *** [main.obj] Error 1
gmake: Target 'all' not remade because of errors.

**** Build Finished ****
```

Character arrays are a bit tricky, in C you can’t just use the assignment operator to change them. You can either write individual characters, like this:


```
#include "msp.h"


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

    int variable1 = 0;
    float variable2 = 0.0;
    char name[10] = "Bill";

    variable1 = 5;
    variable2 = 4.5;

    name[0] = 'B';
    name[1] = 'o';
    name[2] = 'b';
    name[3] = 0;

    while(1)
        {}

}
```

Now when you run and step through the program you should see this in the variable space:



Name	Type	Value	Location
name	unsigned char[10]	[66 'B',111 'o',98 'b',0 '\x00',0 '\x00'...]	0x2000FFF0
variable1	int	5	0x2000FFE8
variable2	float	4.5	0x2000FFEC

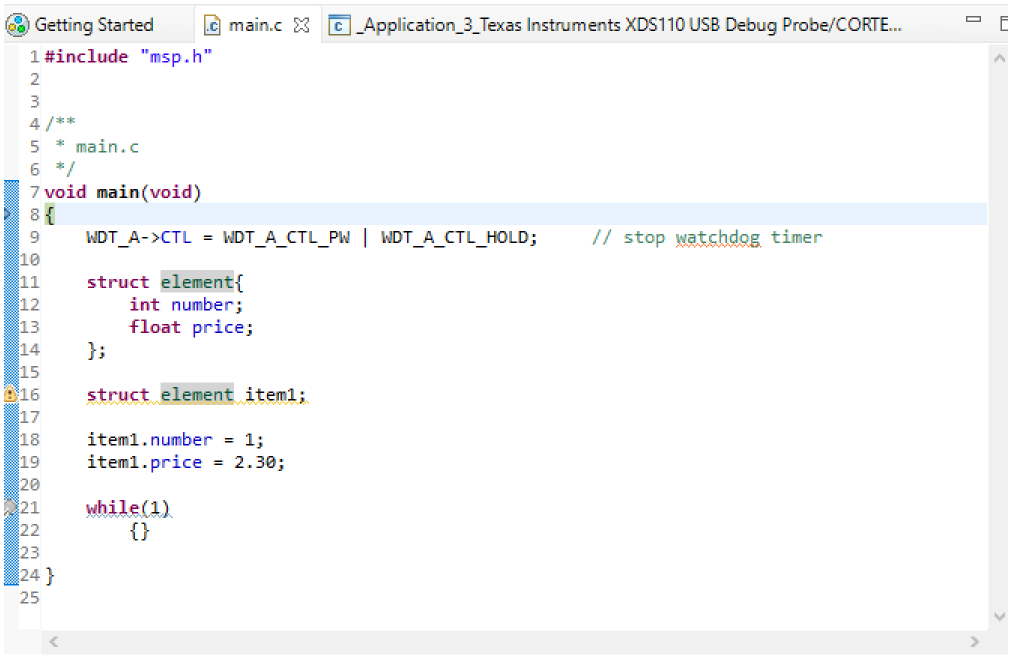
Notice line 21. It is important that you end your character array with the NULL character (value 0) or the computer won't know how many valid characters the array contains.

The Code Composer compiler supports a significant number of other standard data types, including these:

int1
Defines a 1 bit number
int8
Defines an 8 bit number
int16
Defines a 16 bit number
int32
Defines a 32 bit number
char
Defines a 8 bit character
float
Defines a 32 bit floating point number
short
By default the same as int1
Int
By default the same as int8
long
By default the same as int16
void
Indicates no specific type

Combining Data Types using Data Structures

There are times when a data item might have more than one number associated with it. For example, you might be building an inventory of electronic parts. You might want to capture not only the number of parts available, but also the price that each part cost. To do this you can create a new data structure that contains both elements. To do this create the following code:

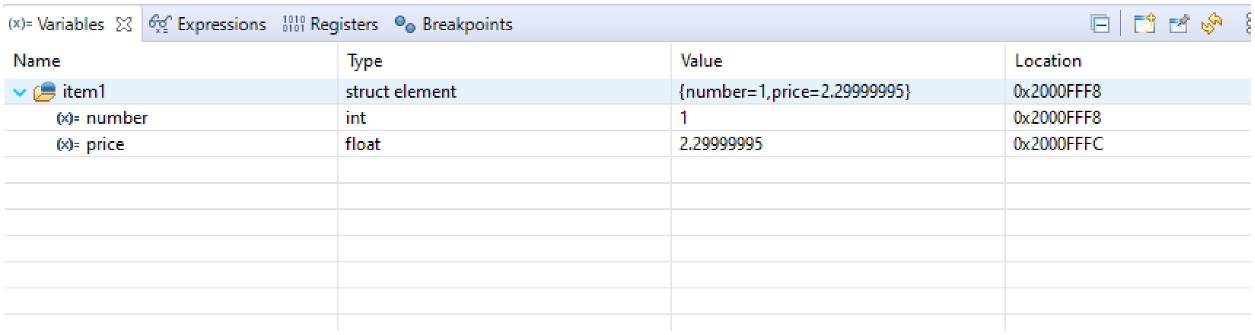

Getting Started
main.c
_Application_3_Texas Instruments XDS110 USB Debug Probe/CORTE...

```c
#include "msp.h"


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

    struct element{
        int number;
        float price;
    };

    struct element item1;

    item1.number = 1;
    item1.price = 2.30;

    while(1)
        {}

}
```

The *struct* keyword tells the compiler that you are going to create a custom, combined data type. In this case it has two elements, an integer variable named number, and a float variable named price. Be aware that the struct keyword used like this does not create a variable, it simply creates a definition of a new type you can use later.

The *struct element item1* line of code creates an instance of the custom combined variable of type *element* and assigns it to the variable named *item1*. To use the variable requires the use of .(dot) notation. The line of code *item1.number = 1;* sets *item1*'s number. The next line of code, *item1.price = 2.30;* sets *item1*'s price.

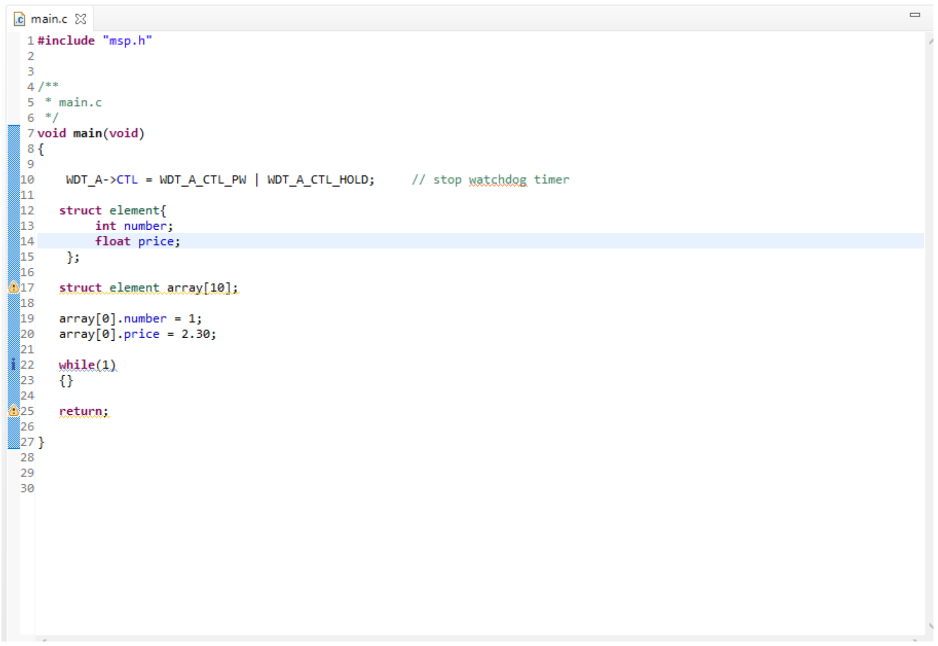
Compile and run this code. Put a breakpoint on line 18. Then Resume the code, and step over lines 18 and 19. In the variable selection click on the arrow key just to the left of the item1 variable name, and you should be able to expand the item to see its individual parts.



Name	Type	Value	Location
item1	struct element	{number=1,price=2.29999995}	0x2000FFF8
(x)= number	int	1	0x2000FFF8
(x)= price	float	2.29999995	0x2000FFFC

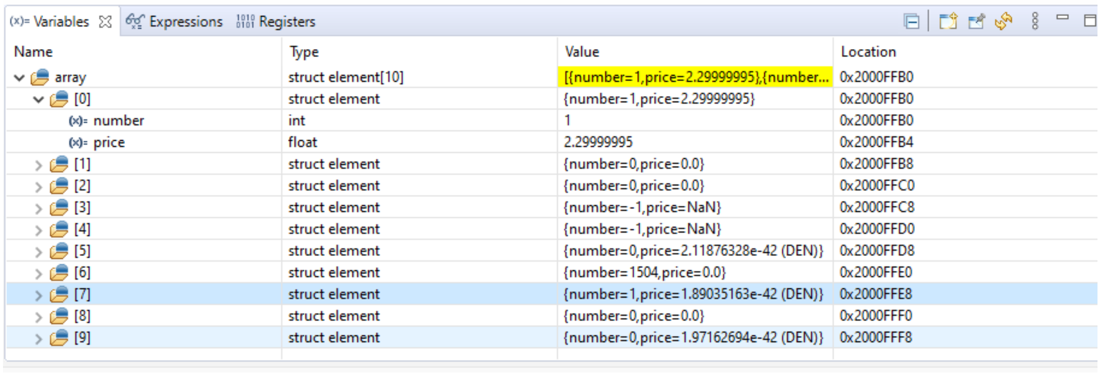
It is interesting to note that the price is not stored as exactly 2.30. The actual value stored is as close as possible to 2.30 based on the accuracy of the storage location.

An inventory of a single item doesn't make much sense, So let's create an array of *elements* called *array:*



```c
#include "msp.h"


/**
 * main.c
 */
void main(void)
{

    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;     // stop watchdog timer

   struct element{
        int number;
        float price;
    };

   struct element array[10];

   array[0].number = 1;
   array[0].price = 2.30;

   while(1)
   {}

   return;

}
```

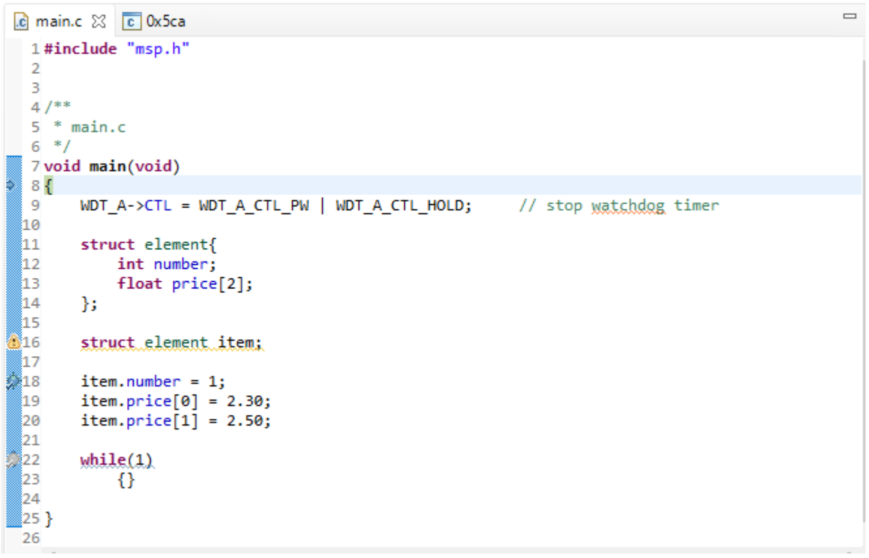
Now compile and debug the program. If you step over until line 21 and look at the variables you should see something like this:



Name	Type	Value	Location
array	struct element[10]	[{number=1,price=2.29999995},{number...	0x2000FFB0
[0]	struct element	{number=1,price=2.29999995}	0x2000FFB0
number	int	1	0x2000FFB0
price	float	2.29999995	0x2000FFB4
[1]	struct element	{number=0,price=0.0}	0x2000FFB8
[2]	struct element	{number=0,price=0.0}	0x2000FFC0
[3]	struct element	{number=-1,price=NaN}	0x2000FFC8
[4]	struct element	{number=-1,price=NaN}	0x2000FFD0
[5]	struct element	{number=0,price=2.11876328e-42 (DEN)}	0x2000FFD8
[6]	struct element	{number=1504,price=0.0}	0x2000FFE0
[7]	struct element	{number=1,price=1.89035163e-42 (DEN)}	0x2000FFE8
[8]	struct element	{number=0,price=0.0}	0x2000FFF0
[9]	struct element	{number=0,price=1.97162694e-42 (DEN)}	0x2000FFF8

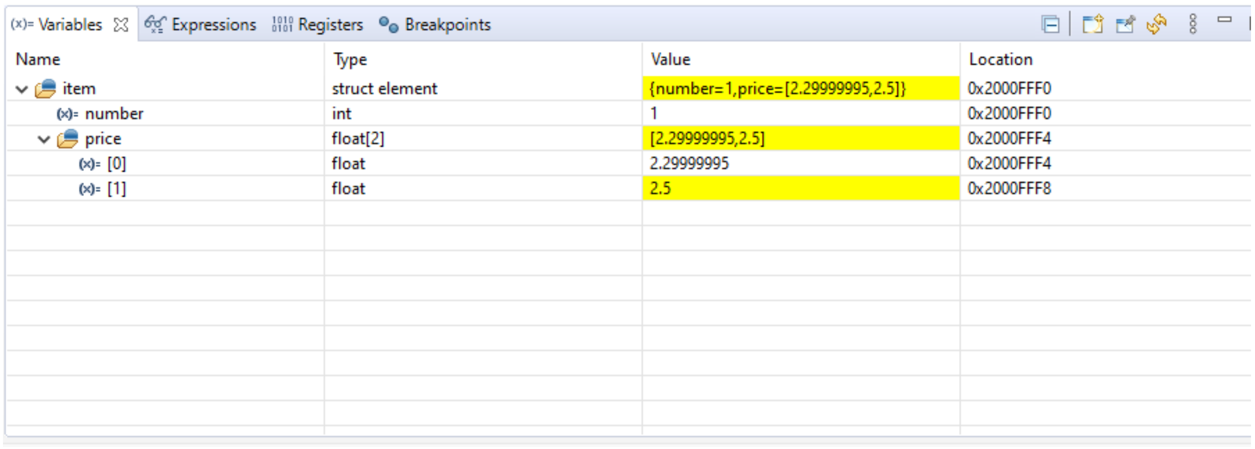
Notice that the number and price values for the first element in the array have been set. Also notice that the other elements in the array have values that are not zero, but hold whatever garbage the memory used happened to hold when it was allocated. It is important to note that the C programming language does not initialize variables. If you forget this, it can cause you real problems when running programs. That’s why you should always initialize all variables your create. Otherwise, when your code runs, the results will seem random. when really your code is using memory that wasn’t cleaned up and holds data from the last time the memory was used.

Also notice where the index operator, [], is used when accessing the array.. It is after the variable name *arrau*, not after *the array*’s *number*, array.number, since it is array’s that are in the array, not *arrau numbers*. This is an important distinction since you can also have arrays that are part of a structure, like this:



```c
#include "msp.h"


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

    struct element{
        int number;
        float price[2];
    };

    struct element item;

    item.number = 1;
    item.price[0] = 2.30;
    item.price[1] = 2.50;

    while(1)
        {}

}
```

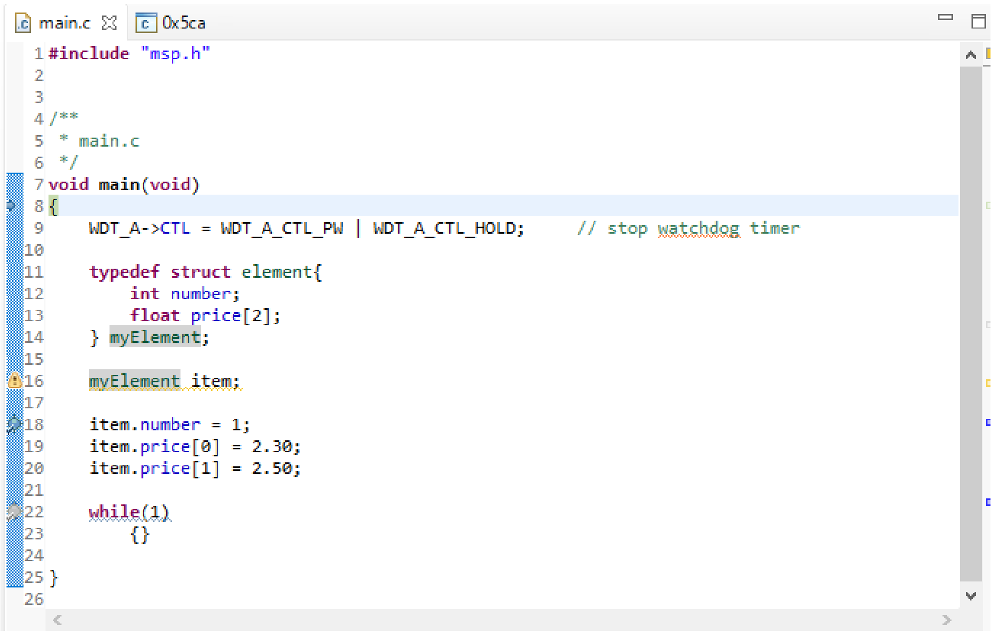
You might need to do this if you have to capture the max and min price of an item. In this case, because the array is part of the structure, you put the index operator, [], after the item.price. This tells the compiler that price is an array of floats instead of a single float. Build and debug this code, step over the assignment operators, and you should see this in the variable space:



Name	Type	Value	Location
item	struct element	{number=1,price=[2.29999995,2.5]}	0x2000FFF0
number	int	1	0x2000FFF0
price	float[2]	[2.29999995,2.5]	0x2000FFF4
[0]	float	2.29999995	0x2000FFF4
[1]	float	2.5	0x2000FFF8

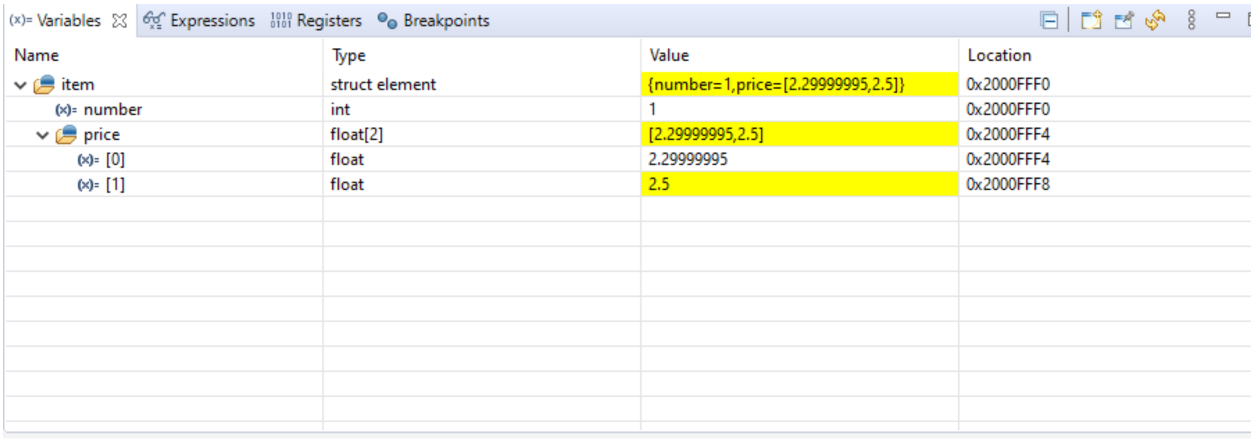
Creating Your Own Data Types using typedef

You can also create your own data types using typedef to make your programming easier to read. This is often used in conjunction with the struct command. If you change your code to look like this:



```
#include "msp.h"


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

    typedef struct element{
        int number;
        float price[2];
    } myElement;

    myElement item;

    item.number = 1;
    item.price[0] = 2.30;
    item.price[1] = 2.50;

    while(1)
        {}

}
```

You will now notice that you’ve create a new type called myElement. It is associated with the data structure element, and you can now use it as you would any data type to declare new variables. Now build and run this code. Resume the code and step over the assignments and you should see this in the variable space:



Name	Type	Value	Location
item	struct element	{number=1,price=[2.29999995,2.5]}	0x2000FFF0
number	int	1	0x2000FFF0
price	float[2]	[2.29999995,2.5]	0x2000FFF4
[0]	float	2.29999995	0x2000FFF4
[1]	float	2.5	0x2000FFF8