

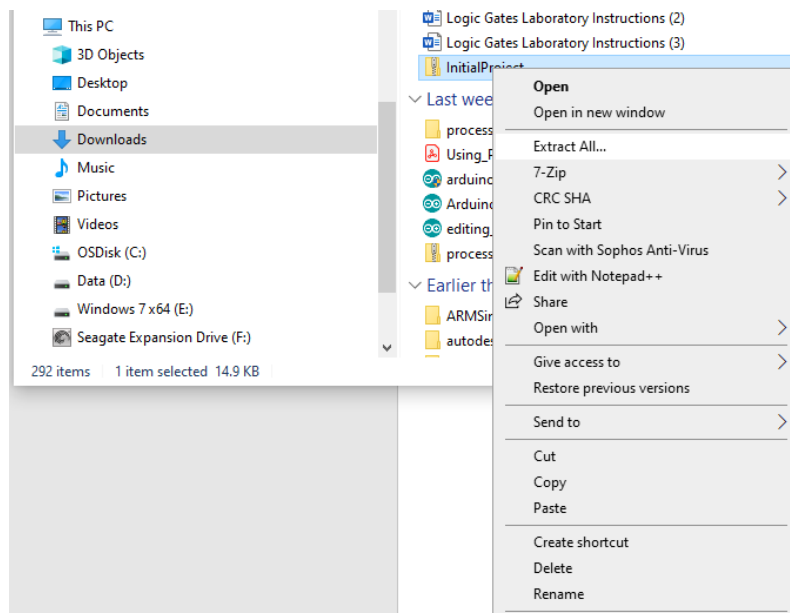
# Introduction to Structures and Pointers

## 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:

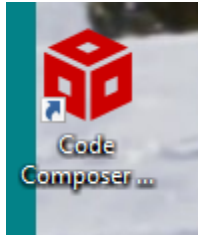
Let's import some example code to get started. To do this download the zip file InitialProject.zip from i-learn. Unzip the file by selecting the zip file and right clicking on the file name, then click Extract All... and chose where you want to extract the file. You can leave it in your Downloads directory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



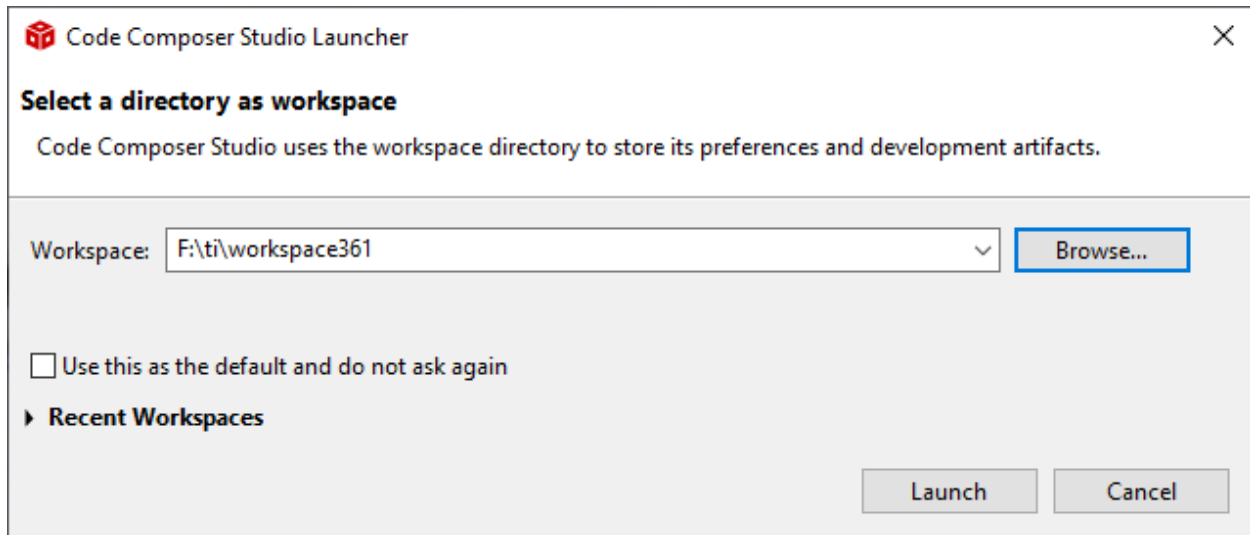
When you have the file unarchived, you can now import it into Code Composer.

## 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:

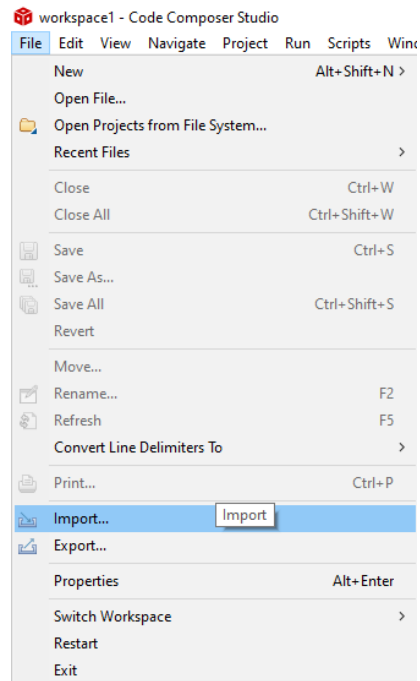


As the program starts you should be prompted for the workspace you want to use.

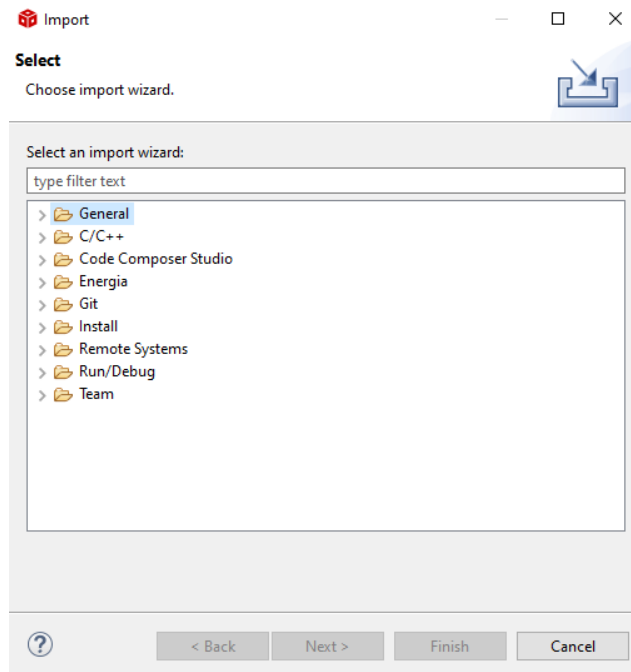


Remember this workspace is the directory where you project files will be placed. You can use several different workspaces if you want to work on very different projects. Click Launch when you have specified the work space directory (you can just use the default.)

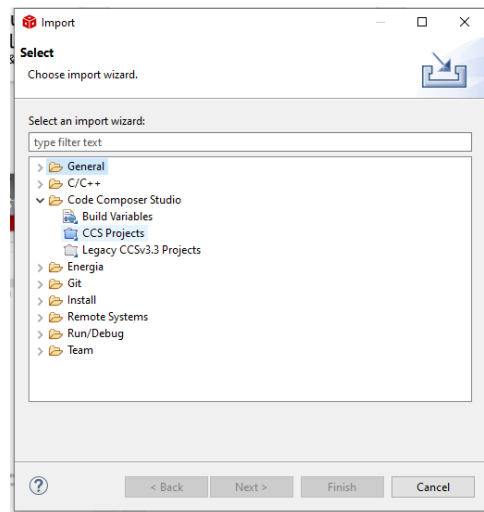
If you haven't already connect your MSP432P401R to the computer via a USB cable. Now let's import the project into Code Composer. To do this select File->Import



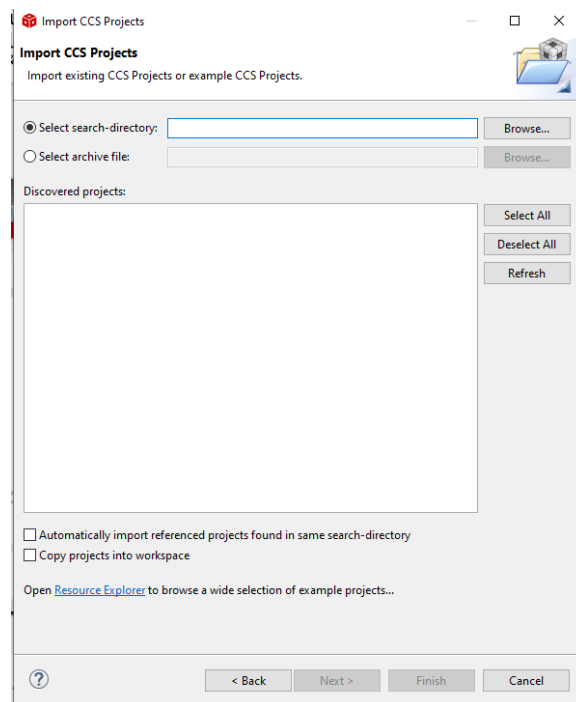
Then you should see this dialogue box:



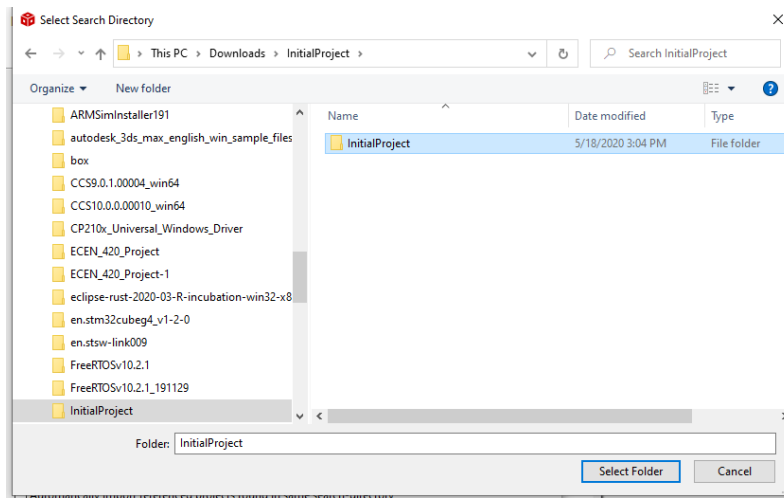
Now select Code Composer Studio, then CCS Projects:



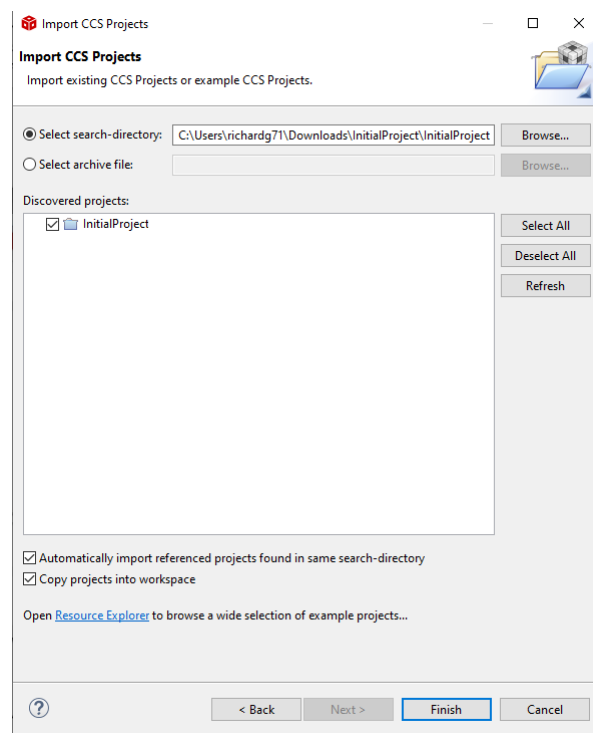
And you should see this dialogue box:



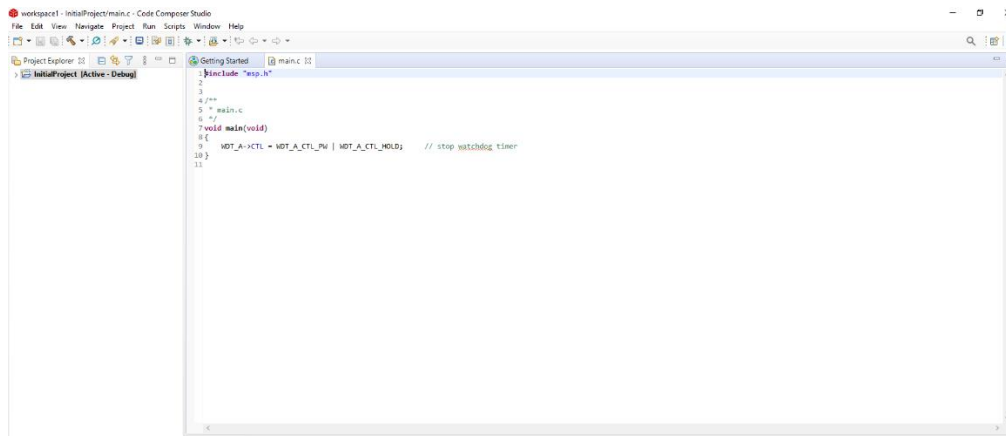
Browse to the directory where you unzip the archive. In my case it was my Downloads directory:



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:



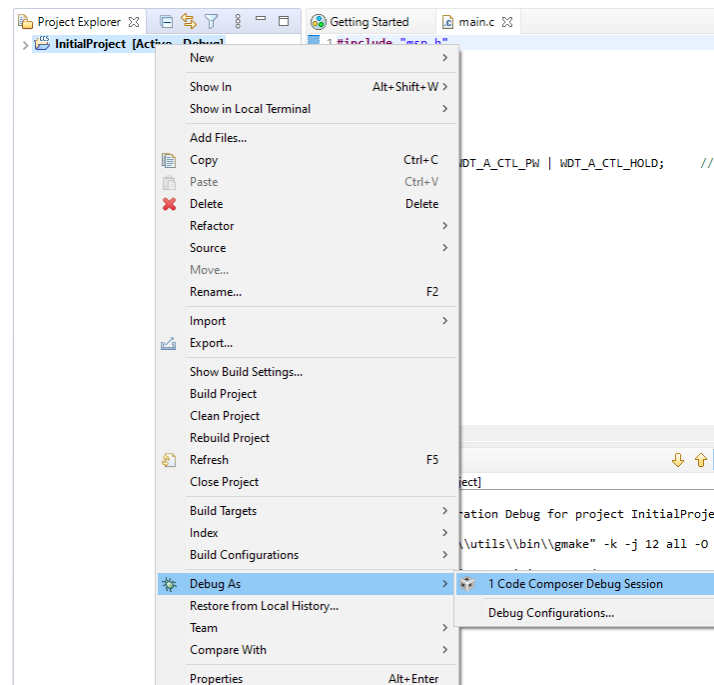
The first time you build a project you will often get suggested actions, like those shown above. They normally only occur the first time you build a new project.

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

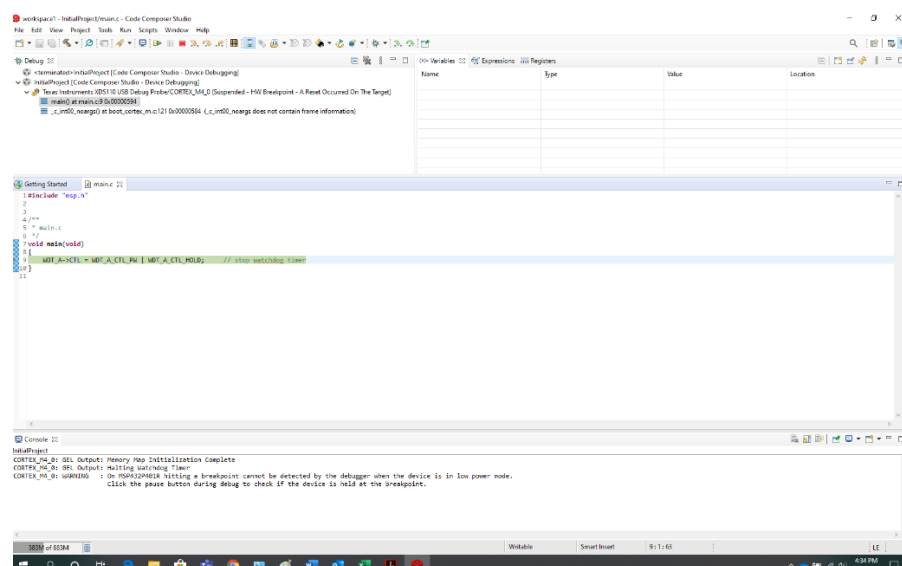
for the computer. Thus you need not only your code, but the interpreter program to be running, for your program to execute successfully.

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.

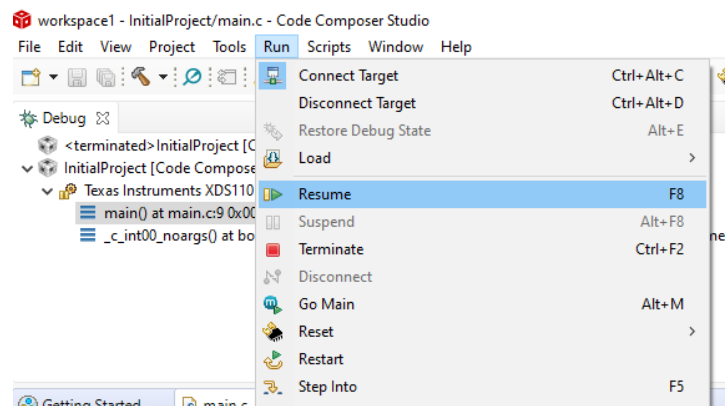
You can actually now run your program by right clicking on the project and hitting the Debug As ->Code Composer Debug Session;



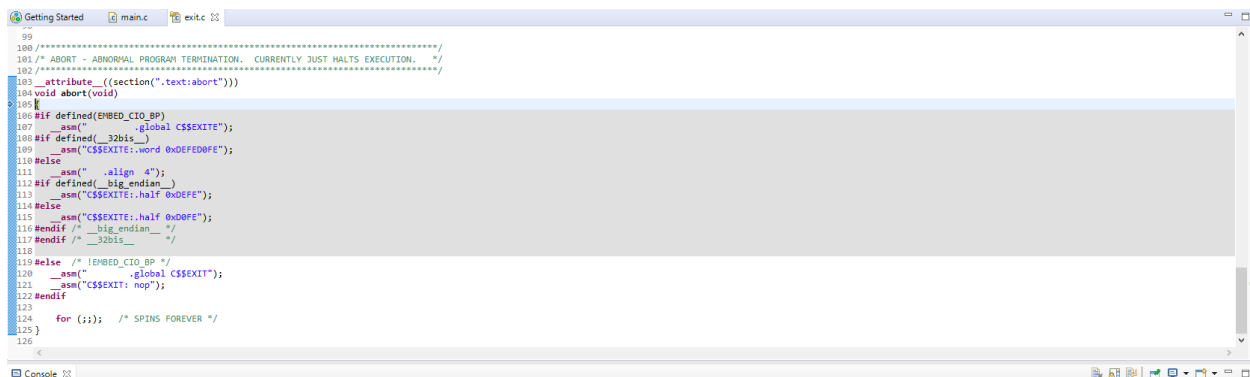
The program will now be uploaded to the MSP432P401R through the USB cable. The view will also switch to the Debug View, and you should see this:



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.

## Data Structures

In a previous lab you learned the basics of functions and variables. This lab will introduce you to some more complex concepts that, while a bit difficult to master, provide ways of making your program both more readable and more efficient. Let's start with some code:

```
#include "msp.h"
```

```
struct myItem{  
    int id;  
    float price;  
};
```

```
struct myItem test_function();
```

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



```

struct myItem item1;
struct myItem item2;

item1.id = 1;
item1.price = 2.30;

    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;           // stop watchdog timer
    item2 = test_function(item1);

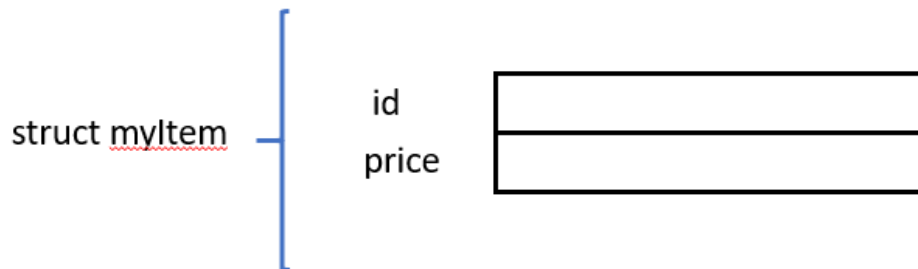
}

struct myItem test_function(struct myItem testItem)
{
    testItem.id = 2;
    testItem.id = 3.22;

    return testItem;
}

```

In this case the introduction of structures allow you to organize your data, when it makes sense, even if the data is not the same type. In this case you've introduced a new data type, the struct myItem which has an integer id and a float price associated with each variable of type struct myItem. When you declare of variable type struct myItem it will contain both an integer and a float. Here is a diagram that might help:



To access each individual element of the structure you'll use the dot connotation. For example:

```
item1.id = 1
```

assigns the id value of the variable name item1 to 1.

Now compile the program and then debug it. Step over the two variable creations, let's look at the variable view:

(x)= Variables Expressions Registers			
Name	Type	Value	Location
▼ item1	struct myItem	{id=1,price=1.89035163e-42 (DEN)}	0x2000FFE8
(x)= id	int	1	0x2000FFE8
(x)= price	float	1.89035163e-42 (DEN)	0x2000FFEC
▼ item2	struct myItem	{id=0,price=0.0}	0x2000FFF0
(x)= id	int	0	0x2000FFF0
(x)= price	float	0.0	0x2000FFF4

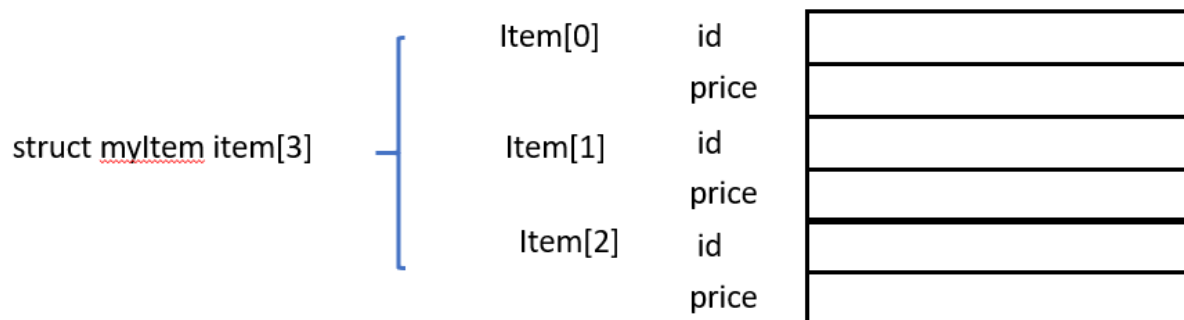
Notice how the variables item1 and item2 have two elements. At this point they haven't been set to anything, but they do exist. Now step to the point where item1 has been set:

(x)= Variables Expressions Registers			
Name	Type	Value	Location
▼ item1	struct myItem	{id=1,price=2.29999995}	0x2000FFE8
(x)= id	int	1	0x2000FFE8
(x)= price	float	2.29999995	0x2000FFEC
▼ item2	struct myItem	{id=0,price=0.0}	0x2000FFF0
(x)= id	int	0	0x2000FFF0
(x)= price	float	0.0	0x2000FFF4

You can now see the values have been set for item1. Now step into the function test\_function. As you step through the function you can see them change. When you return from the function you can also see your original item2 change.

One of the most useful ways to use structures is in Arrays. There are, however, two ways to do this, and they can be confusing, so let's look at each.

First you can create an array of the struct myItem. This would look like this in memory:



You can also create a struct that has an array for one of the elements:

struct yourItem item



id

price[0]

price[1]

price[2]


Let's show some code with an example of both.

```
#include "msp.h"
```

```
struct myItem{  
    int id;  
    float price;  
};
```

```
struct yourItem{  
    int id;  
    float price[3];  
};
```

```
/**
```

```
 * main.c
```

```
 */
```

```
void main(void)
```

```
{
```

```
    struct myItem item1[3];
```

```
    struct yourItem item2;
```

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

```
    item1[0].id = 1;
```

```
    item1[0].price = 3.56;
```

```
    item2.id = 2;
```

```
    item2.price[0] = 5.30;
```

```
    return;
```

```
}
```

Notice for item1, which is the array of structures, to access a single item you use the syntax:

```
item1[0].id = 1;
```

For item2, which contains an array of prices, to access a single item in the array you use the syntax:

```
item2.price[0] = 5.30;
```

Now build and debug this code. When you reach the end of the program and look at the Variable viewer you should see this:

(x)= Variables Expressions Registers			
Name	Type	Value	Location
▼ item1	struct myItem[3]	{id= 1, price= 3.55999994}, {id= 1520, price=...	0x2000FFD8
▼ [0]	struct myItem	{id= 1, price= 3.55999994}	0x2000FFD8
(*)= id	int	1	0x2000FFD8
(*)= price	float	3.55999994	0x2000FFDC
▼ [1]	struct myItem	{id= 1520, price= 0.0}	0x2000FFE0
(*)= id	int	1520	0x2000FFE0
(*)= price	float	0.0	0x2000FFE4
▼ [2]	struct myItem	{id= 1, price= 1.89035163e-42 (DEN)}	0x2000FFE8
(*)= id	int	1	0x2000FFE8
(*)= price	float	1.89035163e-42 (DEN)	0x2000FFEC
▼ item2	struct yourItem	{id= 2, price= [5.30000019, 0.0, 1.97162694e-42]}	0x2000FFF0
(*)= id	int	2	0x2000FFF0
▼ price	float[3]	[5.30000019, 0.0, 1.97162694e-42 (DEN)]	0x2000FFF4
(*)= [0]	float	5.30000019	0x2000FFF4
(*)= [1]	float	0.0	0x2000FFF8
(*)= [2]	float	1.97162694e-42 (DEN)	0x2000FFFC

You can see the item1 is an array of structs, while item2 is a single item that contains an array of prices.

## #define and ENUM

There are two additional data related commands that make C programs easier to read. One of those is #define. In this case you can, instead of using a “magical” number throughout your code, and then having to change it throughout your code every time that number changes, introduce the following command:

```
#define FLASH_BASE ((uint32_t)0x00000000) /*!< Main Flash memory start address */
```

This statement allows you to use the term FLASH\_BASE throughout your code and it will, at compile time, be replaced with the number in this statement. That way you only need to change it once, and it will be changed throughout the code.

The other additional data command is the ENUM command. It is also used to make your code more readable. Here is some example code using the ENUM command:

```
#include "msp.h"

enum Days { SUN, MON, TUE, WED, THU, FRI, SAT };

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

```

{

    int dayAttendance[] = {1232,
                          500,
                          643,
                          343,
                          872,
                          900,
                          1589};

    int myDayAttendance;
    myDayAttendance = dayAttendance[MON];

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

    return;
}

```

The enum statement at the top defines a set of expressions that can later be used in place of numbers. In this case SUN can be used for 0, MON can be used for 1, TUE can be used for 2, and so on. Now notice in the code that instead of using the command:

```
myDayAttendance = dayAttendance[1];
```

you can use the command:

```
myDayAttendance = dayAttendance[MON];
```

This can make it clearer to someone reading your code that you are looking for the attendance associated with Monday.

## Pointers

One of the most powerful, but also most confusing aspect of C programming is the use of Pointers. Pointers, most simply defined, are a way to access memory using the address instead of the name of a location. These are useful mainly in three situations. The first is when accessing hardware that looks like memory. The second is in passing large chunks of information. The third is when trying to dynamically (during run time) decide what function to execute. Let's go over the basics of pointers.

Here is some code that creates and uses a pointer:

```

#include "msp.h"

void test_function(int *variable1);

/**
 * main.c
 */
void main(void)
{
    int variable = 0;
    int *address = &variable;

    *address = 3;
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer
}

```

```

        test_function(address);
    }

    void test_function(int *variable1)
    {
        *variable1 = 4;

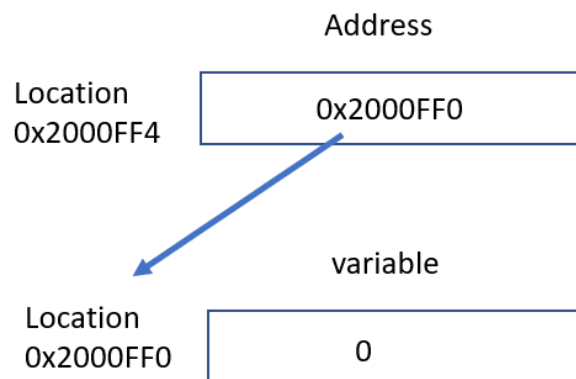
        return;
    }

```

There are several new operators here that you will need to understand. Let's look at the line:

```
int *address = &variable;
```

The `int *address` is a new operator. The `*` operator used in this way indicates that the variable address is of type physical memory address of an integer. The address variable will hold the physical address that will "point to" an integer stored somewhere else in memory. Perhaps a diagram would help:



Again, you might be asking why you need such a structure, so let's review the three different usages of pointers:

- 1) The first is when accessing hardware that looks like memory.

Sometimes you will use hardware that provides an interface that looks like a hardware address. Once you have initialized the pointer to "point to" the hardware you want to access (normally done in an include file) you can then simply read the data as if it were a memory address.

- 2) The second is in passing large chunks of information.

Occasionally you'll need to manage large chunks of memory. You may be used to doing this using the array structure built into C. But you can also use the pointer process to manage this data. This is particularly true when you are using dynamically allocated memory. This is memory that you manage during the execution of the program.

Here is some example code:

```

#include "msp.h"

void test_function(int *variable1);

```

```

/**
 * main.c
 */
void main(void)
{
    int *mem_address;
    int *mem_address1;

    mem_address = (int *)malloc(4 * sizeof(int));
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD; // stop watchdog timer
    test_function(mem_address);
    free(mem_address);
    mem_address1 = (int *)malloc(4 * sizeof(int));
    test_function(mem_address1);
    free(mem_address1);
}

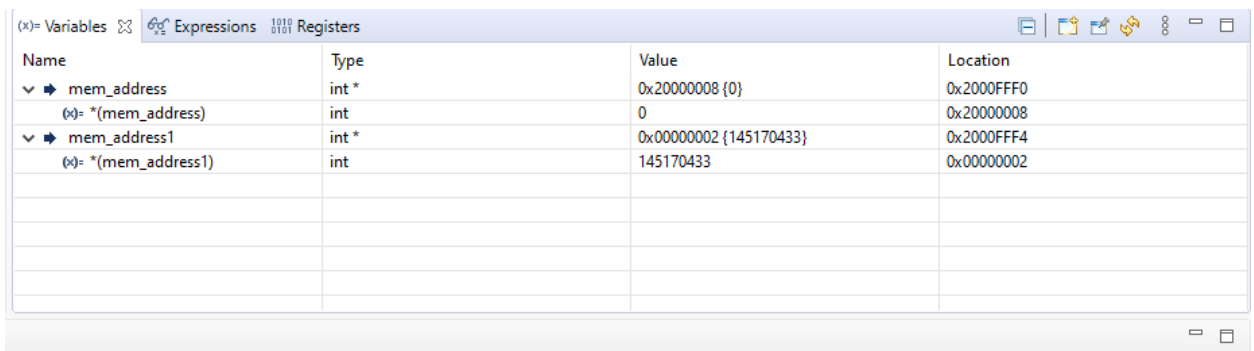
void test_function(int *variable1)
{
    int i;
    for (i = 0; i < 4; i++)
        variable1[i] = 4 * i;

    return;
}

```

In this code you create a memory address that will hold the start address of the block of memory you wish to allocate. Then you use the malloc function call to request the memory from the system. This returns a pointer to the first location of the memory allocated. Now you can pass this pointer to the test\_function subroutine, and it can access the data much like an array.

Build and Debug the code. Step until you read the first free command. You should see something like this in the memory space:



The screenshot shows a debugger's 'Variables' window with two tabs: 'Variables' and 'Expressions'. The 'Variables' tab is active, displaying a table of variables. The table has four columns: 'Name', 'Type', 'Value', and 'Location'. There are two main variable entries, each with a sub-entry for its dereferenced value.

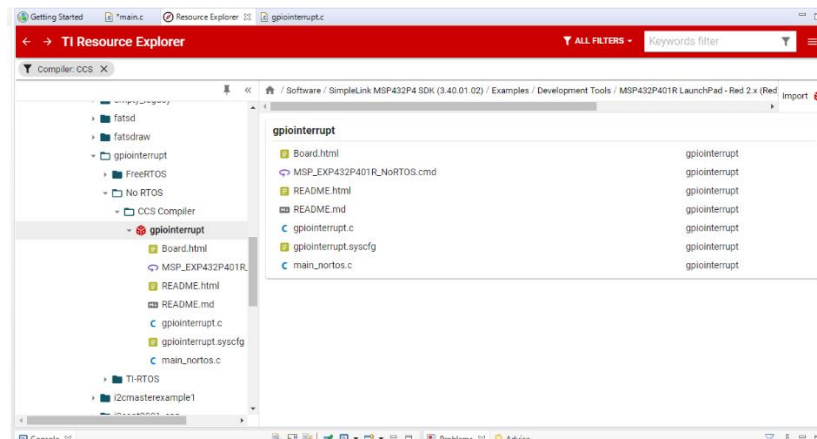
Name	Type	Value	Location
mem_address	int *	0x20000008 {0}	0x2000FFF0
(*)= *(mem_address)	int	0	0x20000008
mem_address1	int *	0x00000002 {145170433}	0x2000FFF4
(*)= *(mem_address1)	int	145170433	0x00000002

mem\_address is at 0x2000FFF0 but is holding the value 0x20000008, which is point to a value 0. If you want to see the entire block of memory, right click on the mem\_address Value and then select View Memory at Value





- 3) The third time when you may want to use a Pointer is when trying to dynamically (during run time) decide what function to execute. So let's look at some quite different code. To access this code import the gpiointerrupt project from the Resource Explorer:



Now select to edit the gpiointerrupt.c file:

```
/*
 * ===== gpiointerrupt.c =====
 */
#include <stdint.h>
#include <stddef.h>

/* Driver Header files */
#include <ti/drivers/GPIO.h>

/* Driver configuration */
#include "ti_drivers_config.h"

/*
 * ===== gpioButtonFxn0 =====
 * Callback function for the GPIO interrupt on CONFIG_GPIO_BUTTON_0.
 */
void gpioButtonFxn0(uint_least8_t index)
{
    /* Clear the GPIO interrupt and toggle an LED */
    GPIO_toggle(CONFIG_GPIO_LED_0);
}

/*
 * ===== gpioButtonFxn1 =====
 * Callback function for the GPIO interrupt on CONFIG_GPIO_BUTTON_1.
 * This may not be used for all boards.
 */
void gpioButtonFxn1(uint_least8_t index)
{
    /* Clear the GPIO interrupt and toggle an LED */
    GPIO_toggle(CONFIG_GPIO_LED_1);
}
```

```

/*
 * ===== mainThread =====
 */
void *mainThread(void *arg0)
{
    /* Call driver init functions */
    GPIO_init();

    /* Configure the LED and button pins */
    GPIO_setConfig(CONFIG_GPIO_LED_0, GPIO_CFG_OUT_STD | GPIO_CFG_OUT_LOW);
    GPIO_setConfig(CONFIG_GPIO_LED_1, GPIO_CFG_OUT_STD | GPIO_CFG_OUT_LOW);
    GPIO_setConfig(CONFIG_GPIO_BUTTON_0, GPIO_CFG_IN_PU | GPIO_CFG_IN_INT_FALLING);

    /* Turn on user LED */
    GPIO_write(CONFIG_GPIO_LED_0, CONFIG_GPIO_LED_ON);

    /* install Button callback */
    GPIO_setCallback(CONFIG_GPIO_BUTTON_0, gpioButtonFxn0);

    /* Enable interrupts */
    GPIO_enableInt(CONFIG_GPIO_BUTTON_0);

    /*
     * If more than one input pin is available for your device, interrupts
     * will be enabled on CONFIG_GPIO_BUTTON1.
     */
    if (CONFIG_GPIO_BUTTON_0 != CONFIG_GPIO_BUTTON_1) {
        /* Configure BUTTON1 pin */
        GPIO_setConfig(CONFIG_GPIO_BUTTON_1, GPIO_CFG_IN_PU |
GPIO_CFG_IN_INT_FALLING);

        /* Install Button callback */
        GPIO_setCallback(CONFIG_GPIO_BUTTON_1, gpioButtonFxn1);
        GPIO_enableInt(CONFIG_GPIO_BUTTON_1);
    }

    return (NULL);
}

```

You'll notice the two functions declared above the main function. Each of these is a function you will want to execute when an interrupt occurs. Configuring this happens using this command:

```

/* install Button callback */
GPIO_setCallback(CONFIG_GPIO_BUTTON_0, gpioButtonFxn0);

```

Notice that the gpioButtonFxn0 does not have any parenthesis. This is an example of using the function name as a pointer. The system now knows that when the GPIO\_BUTTON\_0 is pressed (the switch on the side of the MSP432P401R), then the function gpioButtonFxn0 is to be executed.

You can build and Debug As this program and see the LEDs change when you press the switches.