Structures and Pointers

# Getting Started:

As you have done each week, download the zip file InitialProject.zip file, import it into Code Composer, compile it, and run it in the debugger on the MSP432P401R. You are now ready to add the code for this week’s learning.

# 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~~, lets you make more interesting applications that are also 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)

{

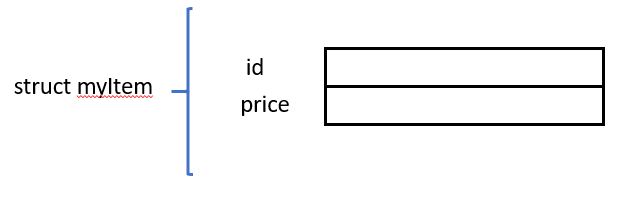
testItem.id = 2;

testItem.id = 3.22;

**return** testItem;

}

The introduction of structures allows 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:

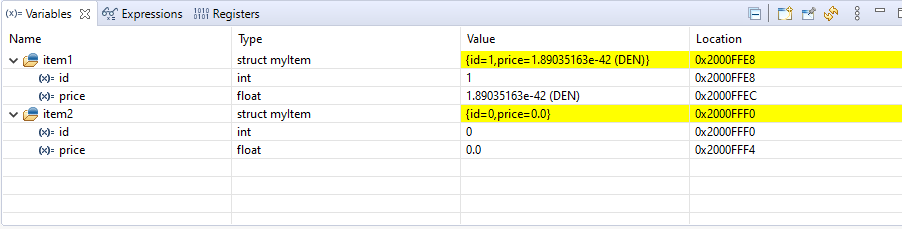


To access each individual element of the structure you’ll use the dot connotation. For example:

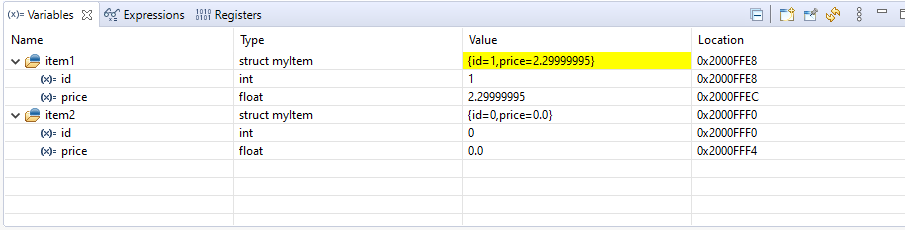
item1.id = 1

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:



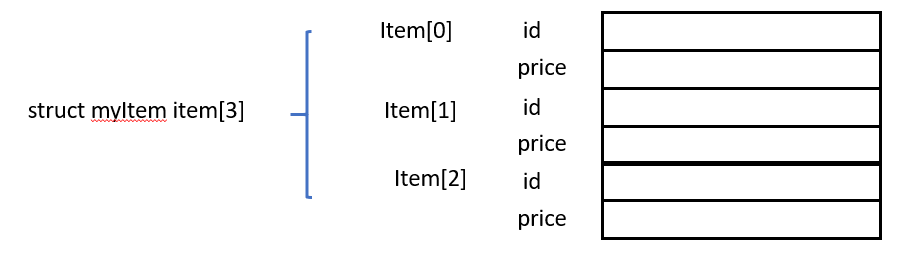
Notice how the variables item1 and item2 have two elements. At this point they haven’t been set to anything, but they do exist. Now step to the point where item1 has been set:



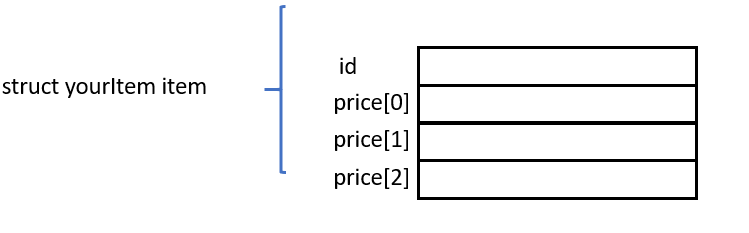
You can now see the values have been set for item1. Now step into the function test\_function. As you step through the function you can see them change. When you return from the function you can also see your original item2 change.

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:



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**;

}

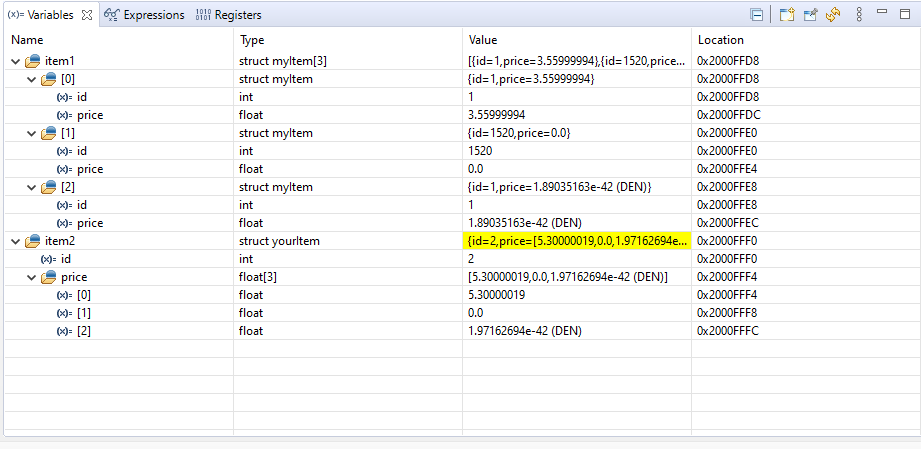
Notice for item1, which is the array of structures, to access a single item you use the syntax:

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:



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 other things you can do to make your C programs easier to read and debug. The first is using the pre-compiler directive *#define*. As an example you can define some text to stand in for some “magical” number you need throughout your code. This means you won’t have to remember what the number stands for, and you won’t have to use up memory in your device if you, mistakenly, chose to try to do the same thing using a variable. It also means that if you choose to change the value of the number, you wouldn’t have to change the number everywhere it’s used in your code, much like using a variable. Here is a common example.

**#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 your compiler will replace *FLASH\_BASE* everywhere it is found with the number.

The second thing you can do to make your code more readable and debugable is to create enumerations. Enumerations allow you to have multiple indicators grouped together without the overhead in memory and CPU use required by an array. Here is an example that creates an enumeration for the days of the week. Notice the use of the *enum* keyword.

**#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 keyword and definition at the top of the code 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 makes it clearer to yourself and others 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 it 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.

Enter this code into this week’s project.

**#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;

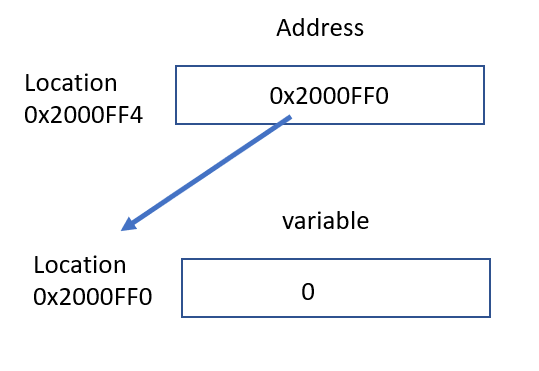
**return**;

}

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 an operator you haven’t seen before. When used this way it tells the compiler 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 such as a -------.

The ----- library provides an interface that points ot a -------‘s hardware address. Once the pointer is assigned to “point to” the hardware you want to access (normally done in an include file) you can then simply read information from the -----(hardware name) as if it were a memory address.

1. 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**;

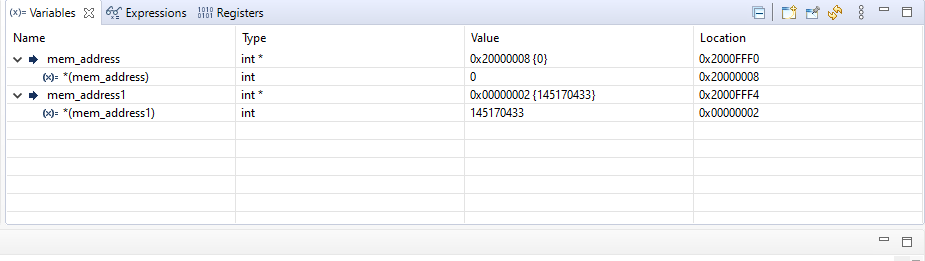
}

In this code you create a memory address that points to the start address of the block of the memory you allocated. The allocation is done using the malloc function (Memory ALLOCate). It’s job is to inform the system that a certain amount of memory is needed and return a pointer to the beginning of the memory block.

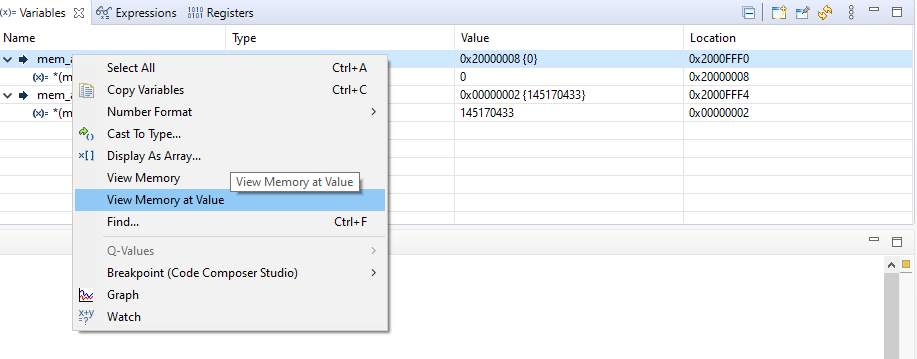
WARNING: if you request too much memory, your application will crash and may also crash the entire system.

If malloc completes successfully, the *mem­\_address*  pointer can be passed to test\_function ~~subroutine~~, *test\_function* can access the block of memory as if the block was an array (since it actually is).

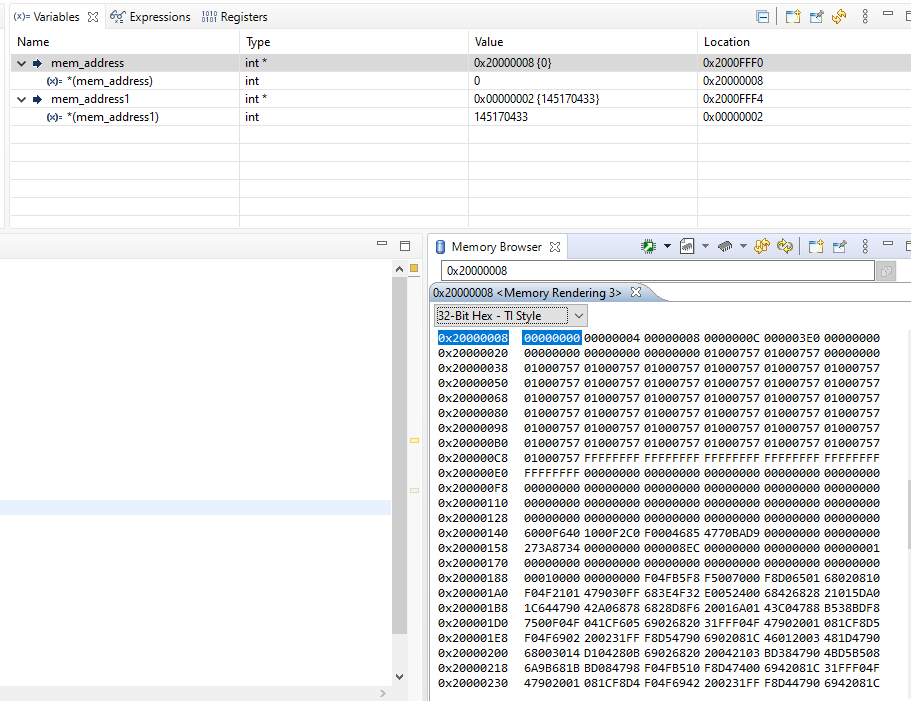
Build and Debug the code. Step until you read the first free command. You should see something like this in the memory space:



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



You can now view the entire block of memory:

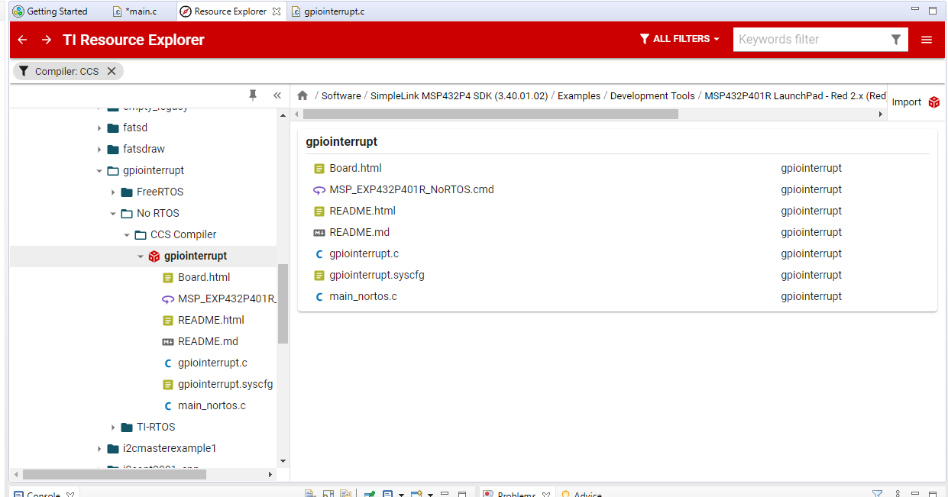


Now step past the Free command. You will see nothing change in the Memory views, even though the Memory has been returned to the system.

Now Step over the next malloc command. If you look at the Memory View you will see that mem\_address1 is pointing to the same location that mem\_address used. The system re-allocated the memory because it was free. If you step past the test\_function it will fill the memory the exact same way.

It is important to note that there is no indication from the system that both mem\_address and mem\_address1 are still pointing to the same memory. If you as a programmer don’t realize this you can cause all kinds of defects (as I noted above, Pointers can be the cause of many serious defects.)

1. 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 *mainThread* 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.