
Microchip University

Advanced C Lab Manual



Advanced C Lab Manual

Table of Contents

• LAB1	Page 3
• Nested Structures and Pointers to Unions and Structures	
• LAB2	Page 11
• Arrays of Pointers	
• LAB3	Page 18
• Arrays of Structure Pointers	
• LAB4	Page 28
• Arrays of Function Pointers	
• LAB5	Page 35
• Function Pointers Used in State Machines	
• LAB6	Page 43
• Function Pointers and Code Portability	
• LAB7	Page 50
• De-referencing Double Pointers	

****All labs tested with MPLABX v5.45 and XC32 v2.30****

Advanced C Programming

Lab1 – Nested Structures and Pointers to Unions and Structures

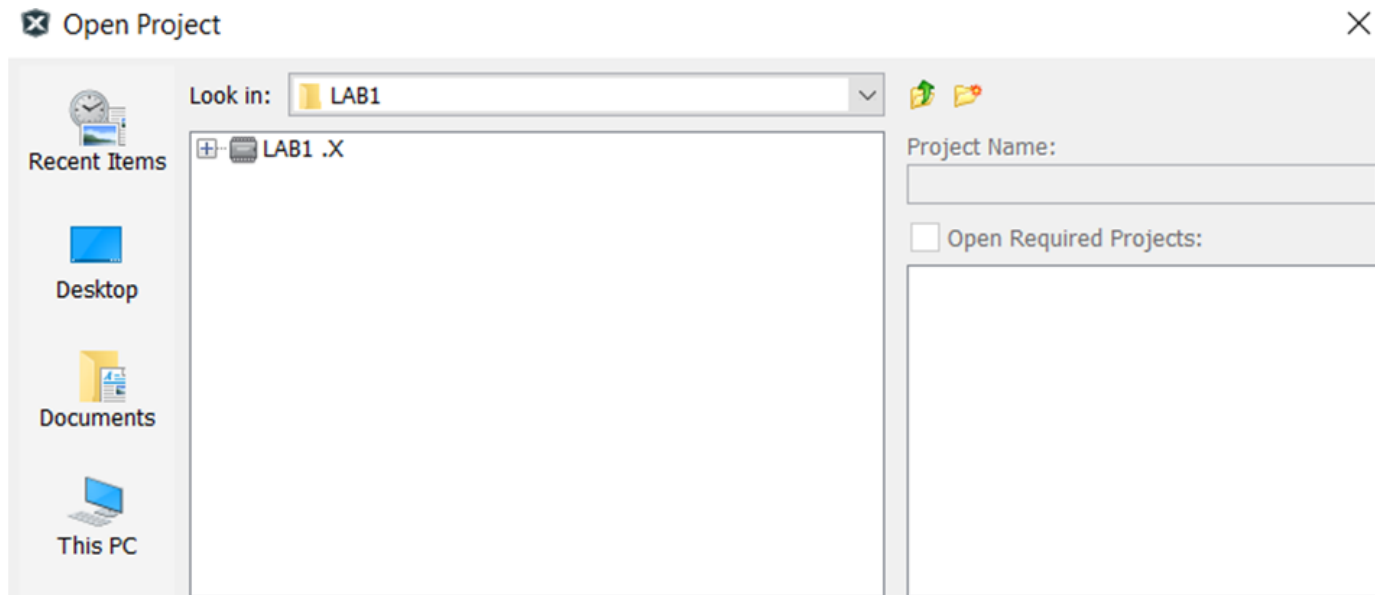
Tasks

In this section you will learn:

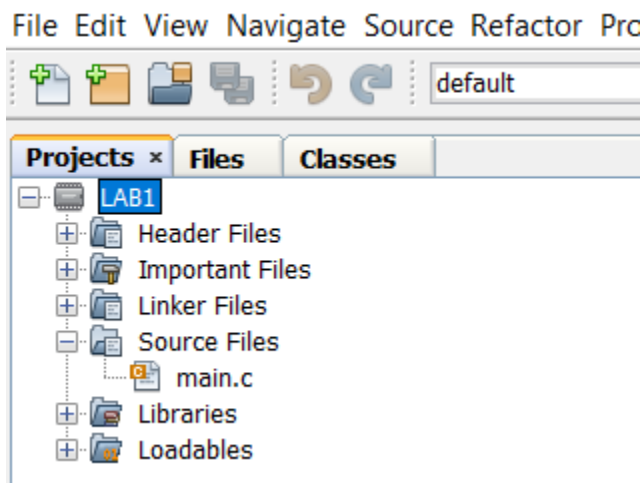
- How to nest structures and unions
- How to create pointers to unions and structures
- How to use structure and union pointers



Open MPLABX and select File -> Open Project. Navigate to the LAB1 folder and open LAB1.X:



Expand the 'Source Files' selection in the 'Projects' folder to show the "main.c" file:



Double click on the "main.c" file and the file will be opened in the code window. All the code for this project is included in this file.

This file contains the startup and initialization code for this project.

Code Analysis

There are three unions that are set up with a typedef. The first union contains a structure that has 4 char (8-bit) variables. The name of this structure is *member*. The union also contains an unsigned long (32-bit) variable named *reg*. The typedef of this union is named *LED_ROTATE_Type*:

```
typedef union {
    unsigned long reg;
    struct {
        unsigned char u8rotateForwardSpeed;
        unsigned char u8rotateReverseSpeed;
        unsigned char u8speedRateChange;
        unsigned char blrotateStatusFlags:    2;
    } member;
}LED_ROTATE_Type;
```

The second union contains a structure that has one unsigned int (16-bit) variable, one char (8-bit) variable, and one 2-bit variable. The name of this structure is *member*. The union also contains an unsigned long (32-bit) variable named *reg*. The typedef of this union is named *LED_BREATHE_Type*:

```
typedef union {
    unsigned long reg;
    struct {
        unsigned short ul6pwmLightIntensity;
        unsigned char u8breatheSpeed;
        unsigned char blbreatheFlags:        2;
    } member;
}LED_BREATHE_Type;
```

The third union contains a structure that has a four-member array of char (8-bit) values. The name of this structure is *member*. The union also contains an unsigned long (32-bit) variable named *reg*. The typedef of this union is named *LED_PATTERN_Type*:

```
typedef union {
    unsigned long reg;
    struct {
        unsigned char u8pattern[4];
    } member;
}LED_PATTERN_Type;
```

All three of the unions are now included in one structure. The name of this structure is *STRUCT_LEDVariables*:

```
typedef struct {
    volatile LED_ROTATE_Type    ledRotation;
    volatile LED_PATTERN_Type   ledPattern;
    volatile LED_BREATHE_Type   ledBreathe;
} STRUCT_LEDVariables;
```

The *volatile* keyword is added to make sure that the compiler does not optimize-out the code since the values are being changed but not all are used in this project. The names of the unions *ledRotation*, *ledPattern*, and *ledBreathe* are created based on the 'typedef' name of the specific union.

We now need to create an instance of this structure. So far, no memory has been reserved for any of the unions or structures. Only typedefs were created, so we need to create an instance of the unions and structures:

```
STRUCT_LEDVariables STRUCT_allLEDVariables;
```

Since this structure contains the definitions for all individual unions, memory is now allocated for all of them.

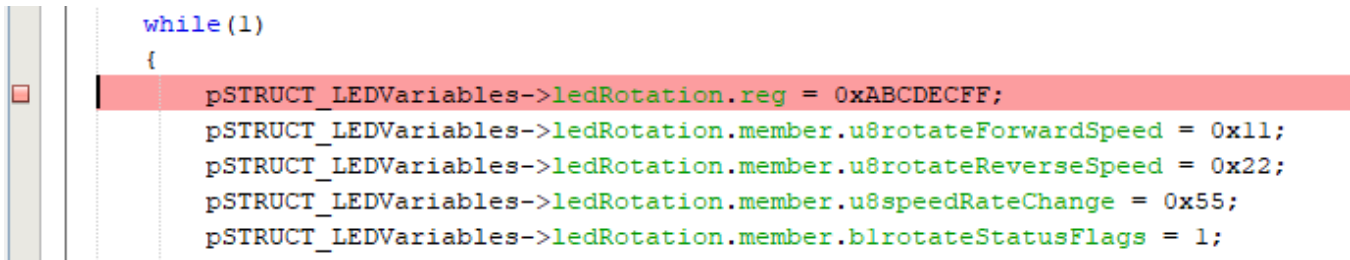
This lab will use pointers to access the variables in the unions, so we need to create a pointer to the *STRUCT_allLEDVariables* structure. It will also be initialized when it is declared:

```
STRUCT_LEDVariables *pSTRUCT_LEDVariables = &STRUCT_allLEDVariables;
```

The 'main()' function contains the 'while()' loop that just loads values into the individual union members by using the pointer that was previously created. The member access is easy to perform now that all three unions can be accessed using just one pointer.

Run the Project

We will analyze the variable access one instruction at a time. Set a breakpoint at the first code line within the 'while(1)' loop by clicking on the gray bar to the left of the code. A red square appears showing that a breakpoint is set:



Now click on the 'Debug Project' icon:



The project will build and the simulator will start code execution. The execution will stop at the breakpoint that was just set. A green arrow appears on top of the square:

```
while(1)
{
    pSTRUCT_LEDVariables->ledRotation.reg = 0xABCDECF;
    pSTRUCT_LEDVariables->ledRotation.member.u8rotateForwardSpeed = 0x11;
    pSTRUCT_LEDVariables->ledRotation.member.u8rotateReverseSpeed = 0x22;
    pSTRUCT_LEDVariables->ledRotation.member.u8speedRateChange = 0x55;
    pSTRUCT_LEDVariables->ledRotation.member.blrotateStatusFlags = 1;
}
```

We now need to look at the `pSTRUCT_LEDVariables` pointer because it will show the values of all the unions. Mouse over the `pSTRUCT_LEDVariables` name and right click, then select 'New Watch':

The screenshot shows a C code editor with a 'while(1)' loop. The code defines a 'pSTRUCT_LEDVar' array and a 'pSTRUCT_LEDVar' pointer. The context menu is open, showing options like 'Insert C Line Trace', 'Log Selected C Value', 'Navigate', 'New Watch...', 'New Runtime Watch...', 'New Data Breakpoint', 'Toggle Line Breakpoint', 'Remove All Breakpoints', 'Show Call Graph', 'Find Usages', and 'Find in Projects...'. The code also includes a 'CFF;' section with various assignments and a 'while(1)' loop.

```
while(1)
{
    pSTRUCT_LEDVar
    pSTRUCT_LEDVar
    pSTRUCT_LEDVar
    pSTRUCT_LEDVar
    pSTRUCT_LEDVar

    pSTRUCT_LEDVar
    pSTRUCT_LEDVar
    pSTRUCT_LEDVar
    pSTRUCT_LEDVar
    pSTRUCT_LEDVar

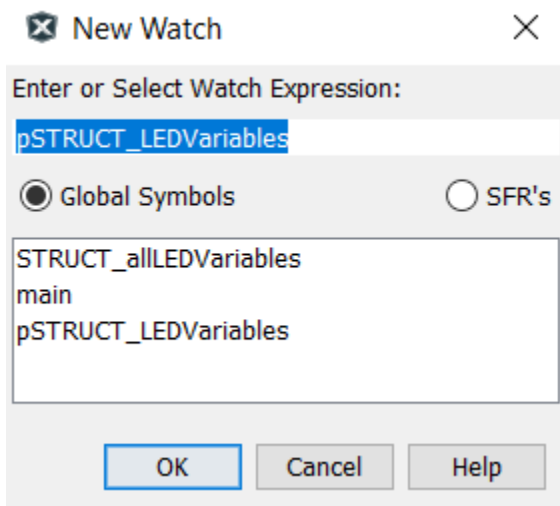
    pSTRUCT_LEDVar
    pSTRUCT_LEDVar
    pSTRUCT_LEDVar
    pSTRUCT_LEDVar

    CFF;
    teForwardSpeed = 0x11;
    teReverseSpeed = 0x22;
    dRateChange = 0x55;
    teStatusFlags = 1;

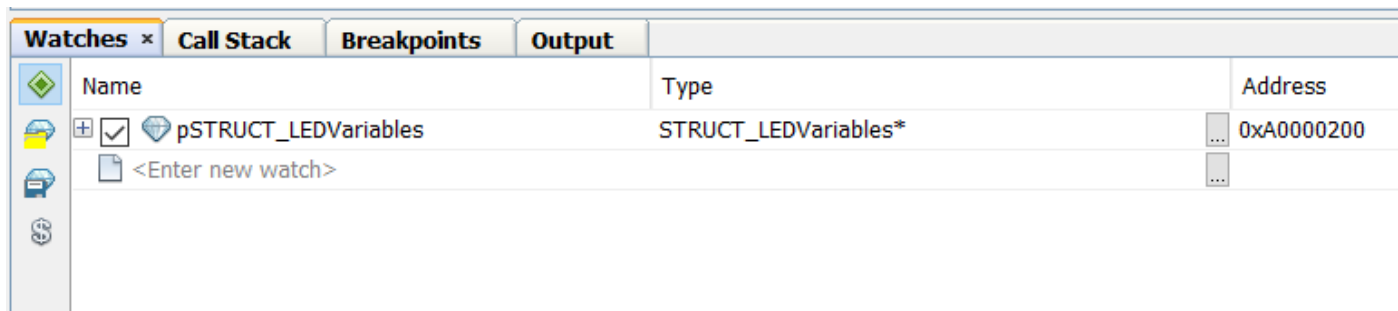
    11;
    rn[0] = 0xAA;
    rn[1] = 0xBB;
    rn[2] = 0xCC;
    rn[3] = 0xDD;

    78;
    heSpeed = 0x33;
    ightIntensity = 0xFEDC;
    heFlags = 0b11;
}
```

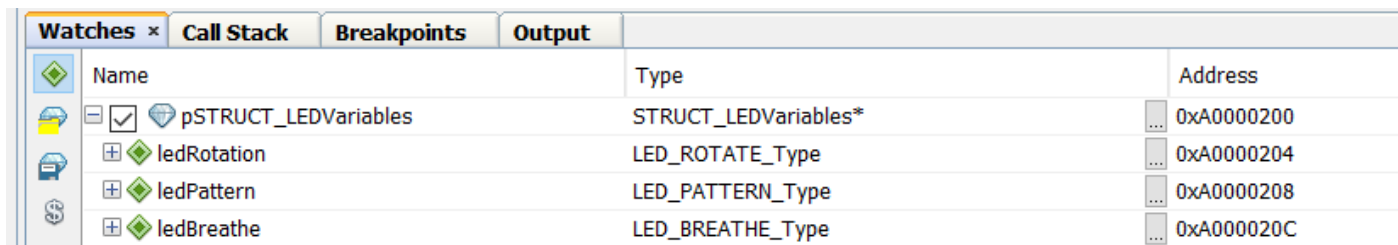
A pop-up will appear with the selected variable to watch. Select 'OK'.



The variable will appear in the watch window:



Click on the '+' next to the variable name to expand the pointer and see everything it is pointing to:



Each of the unions are now showing. Finally, expand each of the union instance names to see all the members:

Watches x	Call Stack	Breakpoints	Output	
Name	Type	Address	Value	
pSTRUCT_LEDVariables	STRUCT_LEDVariables*	0xA0000200	0xA0000204	
ledRotation	LED_ROTATE_Type	0xA0000204		
reg	unsigned long	0xA0000204	0xABCDECF	
member	struct	0xA0000204		
u8rotateForwardSpeed	unsigned char	0xA0000204	'y'; 0xff	
u8rotateReverseSpeed	unsigned char	0xA0000205	'I'; 0xec	
u8speedRateChange	unsigned char	0xA0000206	'f'; 0xcd	
b1rotateStatusFlags	unsigned char	0xA0000207	0x03	
ledPattern	LED_PATTERN_Type	0xA0000208		
ledBreathe	LED_BREATHE_Type	0xA000020C		
<Enter new watch>				

Remember that this is little Endian memory. The *reg* value is loaded with the value 0xABCDECF. The last value in the structure is the most significant memory location. Since it is a 2-bit value and a character is blocked off, the 0xAB value has both least significant bits set. Therefore, the 2-bit field is 0b11, or 3. The least significant byte in the value loaded into the 'reg' value is 0xFF and that value is loaded into *u8rotateForwardSpeed* that is listed first.



Now, click the 'Step Into' icon again:

The line of code that is executed is:

```
pSTRUCT_LEDVariables->ledRotation.member.u8rotateForwardSpeed = 0x11;
```

When we look at the Watch window, we can see that the *u8rotateForwardSpeed* value has changed. The *reg* value has also changed (both are now shown in red signifying that they changed):

pSTRUCT_LEDVariables	STRUCT_LEDVariables*	0xA0000200	0xA0000204
ledRotation	LED_ROTATE_Type	0xA0000204	
reg	unsigned long	0xA0000204	0xABCDEC11
member	struct	0xA0000204	
u8rotateForwardSpeed	unsigned char	0xA0000204	DC1; 0x11
u8rotateReverseSpeed	unsigned char	0xA0000205	'I'; 0xec
u8speedRateChange	unsigned char	0xA0000206	'f'; 0xcd
b1rotateStatusFlags	unsigned char	0xA0000207	0x03

You can use this same procedure to step through each line of code and look at the values in the Watch Window. The access to the unions is now simplified since only one pointer is required to access all the variables.

Advanced C Programming

Lab2 – Arrays of Pointers

Tasks

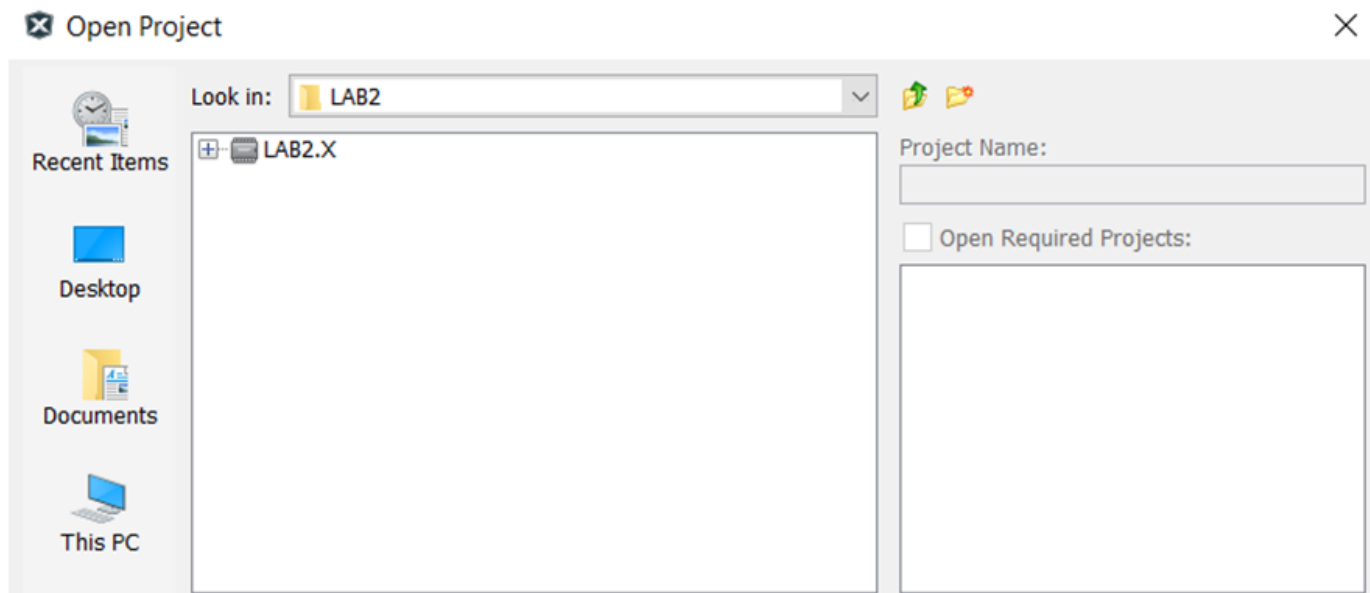
In this section you will learn:

- How to define arrays of pointers
- How to initialize the arrays of pointers to create and point to strings
- How to use pointers to access individual string characters
- How to place pointers in program memory

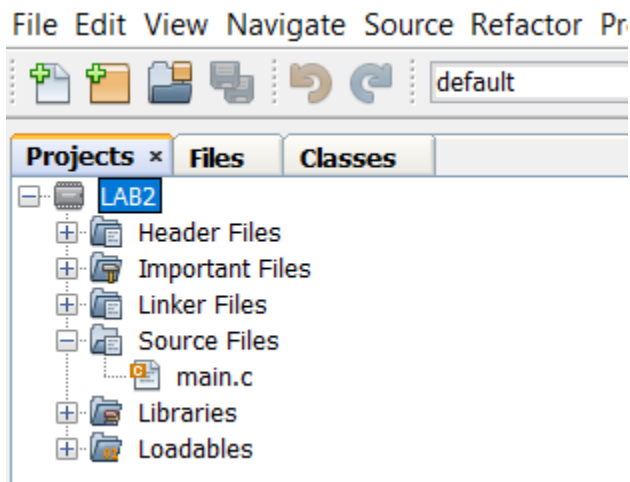


MICROCHIP

Open MPLABX and select File -> Open Project. Navigate to the LAB2 folder and open LAB2.X:



Expand the 'Source Files' selection in the 'Projects' folder to show the "main.c" file:



Double click on the "main.c" file and the file will be opened in the code window. All the code for this project is included in this file.

Code Analysis

The variables *i* and *j* are used in the 'for' loops for loop control. The variable *y* is used to hold each character that is accessed in the strings.

```
unsigned char i = 0, j = 0, y;
```

The variable **p[4]* is an array of pointers. The pointer is in program memory and is initialized to point to four different strings. These strings are placed in flash when the microcontroller is programmed. The *const* keyword is used to place the pointer in program memory:

```
char *const p[4] = {"a01", "b02", "c03", "d04"};
```

The variable **q[4]* is an array of pointers. The pointer is in data memory and it is initialized to point to four different strings. These strings are placed in flash when the microcontroller is programmed. This pointer is the same as the **p[]* pointer but does not contain the *const* keyword. The pointer is placed in data memory and the strings are placed in program memory:

```
const char *q[4] = {"e01", "f02", "g03", "h04"};
```

The variable **r[4]* is an array of pointers. The compiler attribute is used to place the pointers themselves in program memory. The strings are also placed in program memory:

```
const char * __attribute__((space(prog))) r[4] = {"i01", "j02", "k03", "l04"};
```

The pointers are initialized to point to the first character in each of the strings. *p[0]* points to the first character of "a01", *p[1]* points to the first character of "b02", and so on.

We are now going to use the array of pointers to access each string. The 'for' loops continually step through each string and place the value into the *y* variable. The first *for* loop runs through the *p[]* pointer array. The second loop runs through the *q[]* pointer array, and the last loop runs through the *r[]* pointer array. Each loop accesses each string variable, one at a time, and places that value into the *y* variable.

```

    while(1)
    {
//Step through p[ ] pointer
        for(i = 0; i < 4; i++, j=0)
        {
            do{
                y = *(p[i] + j);
                j++;
            } while(y != '\0');
        }

//Step through q[ ] pointer
        for(i = 0; i < 4; i++, j=0)
        {
            do{
                y = *(q[i] + j);
                j++;
            } while(y != '\0');
        }

//Step through r[ ]pointer
        for(i = 0; i < 4; i++, j=0)
        {
            do{
                y = *(r[i] + j);
                j++;
            } while(y != '\0');
        }

        while(1);
    }

```

In the first pass of the first 'for' loop, $i = 0$. The 'do ... while' loop will continuously step through each character string and place the value that is pointed to by the pointer into the y variable. The j variable is incremented in each pass of the 'do ... while' loop so that it increments the base pointer value to point to the next character in the string.

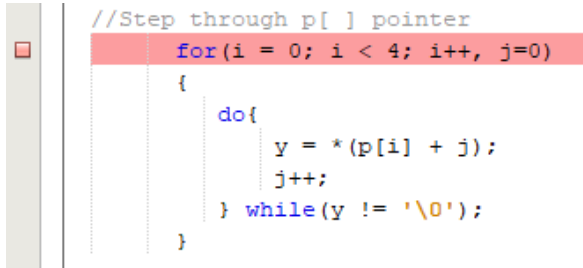
In the first pass of the first 'for' loop, i is equal to 0 for each character access. The variable j starts at 0 and is incremented to access each successive character value. When the string has been completely accessed, a NULL character is retrieved since this character is automatically inserted at the end of each character string by the compiler. The next iteration of the 'for' loop occurs with $i = 1$. The variable $p[1]$ points to the second string "b02". We can then access each character of this string.

The remaining two *for* loops perform the same functionality but using the arrays $q[]$ and $r[]$. The main difference in the data accesses is the placement of the pointer arrays themselves. In the $p[]$ and $q[]$ pointer arrays, the pointers are located in program and data memory, respectively. The $r[]$ pointer array is placed explicitly in program memory using compiler attributes.

The Watch window shows the memory addresses for the pointers. The $p[]$ array has addresses in program memory and the $q[]$ array has addresses in data memory. The $r[]$ array has addresses in program memory.

Run the Project

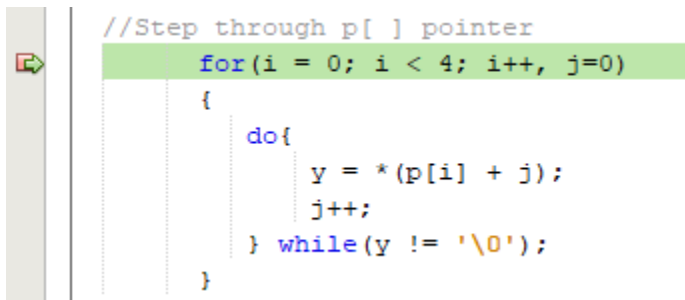
We will analyze the variable access one instruction at a time. Set a breakpoint at the 'for' loop by clicking on the gray bar to the left of the code. A red square appears showing that a breakpoint is set:



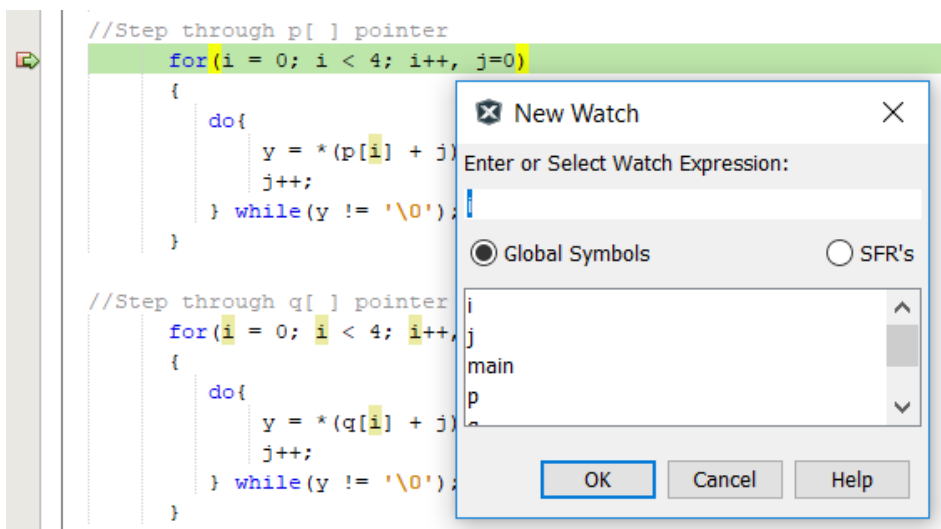
Now click on the 'Debug Project' icon:



The project will build and the simulator will start code execution. The execution will stop at the breakpoint that was just set. A green arrow appears on top of the square:



Let's look at the variables as we step through the code. We first need to include these variables in the Watch window. We will include these variables using the same method that we used in LAB1. Position the cursor over the *i* variable and right click. A pop-up window appears. Select 'New Watch':



Hit 'OK' to add the *i* variable to the Watch window.

The *i* variable is now displayed in the Watch window:

Watches x	Call Stack	Breakpoints	Output	
	Name	Type	Address	Value
	<input checked="" type="checkbox"/> i	unsigned char	0xA0000200	NUL; 0x0
	<Enter new watch>			

Use the same procedure to show the *j*, *y* and *p[]* variables in the Watch window.

Search Results


Watches x

Name	Type	Address	Value
<input checked="" type="checkbox"/> i	unsigned char	0xA0000200	NUL; 0x0
<input checked="" type="checkbox"/> j	unsigned char	0xA0000201	NUL; 0x0
<input checked="" type="checkbox"/> p	char*[4]	0x9D000AD4	
<input checked="" type="checkbox"/> y	unsigned char	0xA0000202	NUL; 0x0
<Enter new watch>			

Click the '+' sign next to the *p* variable to show each member of the array. Then click the '+' symbol for each of the individual '*p[]*' array pointers:

<input checked="" type="checkbox"/> p	char*[4]	0x9D000AD4	
<input checked="" type="checkbox"/> p[0]	char*	0x9D000AD4	'?'; 0xac4
<input checked="" type="checkbox"/> *p[0]	char	0x9D000AC4	'a'; 0x61
<input checked="" type="checkbox"/> p[1]	char*	0x9D000AD8	'?'; 0xac8
<input checked="" type="checkbox"/> *p[1]	char	0x9D000AC8	'b'; 0x62
<input checked="" type="checkbox"/> p[2]	char*	0x9D000ADC	'?'; 0xacc
<input checked="" type="checkbox"/> *p[2]	char	0x9D000ACC	'c'; 0x63
<input checked="" type="checkbox"/> p[3]	char*	0x9D000AE0	'\u0ad0'; 0xad0
<input checked="" type="checkbox"/> *p[3]	char	0x9D000AD0	'd'; 0x64



You can now see the physical addresses of the *p[]* pointers and the values that each *p[]* member is pointing to. For example, *p[]* is the actual pointer and the address is shown as the program memory address of the variable. The value that it contains is the address of the first character of the respective character string that was initialized for that pointer. The variable **p[]* shows the address it is pointing to along with the value of that flash location. The Watch window shows that the values for each pointer are the values of the first character in the respective strings. Also, the 'P' in the icon next to the pointer names shows that the pointer is in program memory.

Let's step through the code to see exactly what is happening. Now, click the 'Step Into' icon  and the arrow will be positioned to the left of the 'y=' instruction. This is the instruction that will be executed on the next step:


```
y = *(p[i] + j);
```

The *y* variable value in the Watch window is 0.













Now, click the 'Step Into' icon again. The arrow will point to the instruction that increments the *j* variable. The *y* variable value in the Watch window is 0x61 which is the ASCII value for 'a'. 'a' is the first character in 'a01'.

 <i>y</i>	unsigned char	 0xA0000202	 'a'; 0x61
--	---------------	--	---

Click the 'Step Into' icon three more times so that it executes the same instruction in the next loop of the 'do...while' loop. The *y* variable value shows the value of the string it is pointing to.

You can continue to step through the code to access the entire 'a01' string. When the NULL character is reached after accessing '1', the *i* variable is incremented so that it points to the second string. The variable *i* is now 1 and *j* is reset to 0. The second character string can now be accessed in the same manner as we demonstrated with accessing the 'a01' string.

Use the same procedure to run through the *q[]* and *r[]* pointer arrays. They will return the values of each string member that was initialized for the respective pointer. The *r[]* and *p[]* arrays reside in program memory:

Name	Type	Address	Value
  <i>p</i>	char*[4]	 0x9D000AD4	
  <i>q</i>	char*[4]	 0xA0000204	
  <i>r</i>	char*[4]	 0x9D000B04	

Advanced C Programming

Lab3 – Arrays of Structure Pointers and Passing Structure Addresses to Functions

Tasks

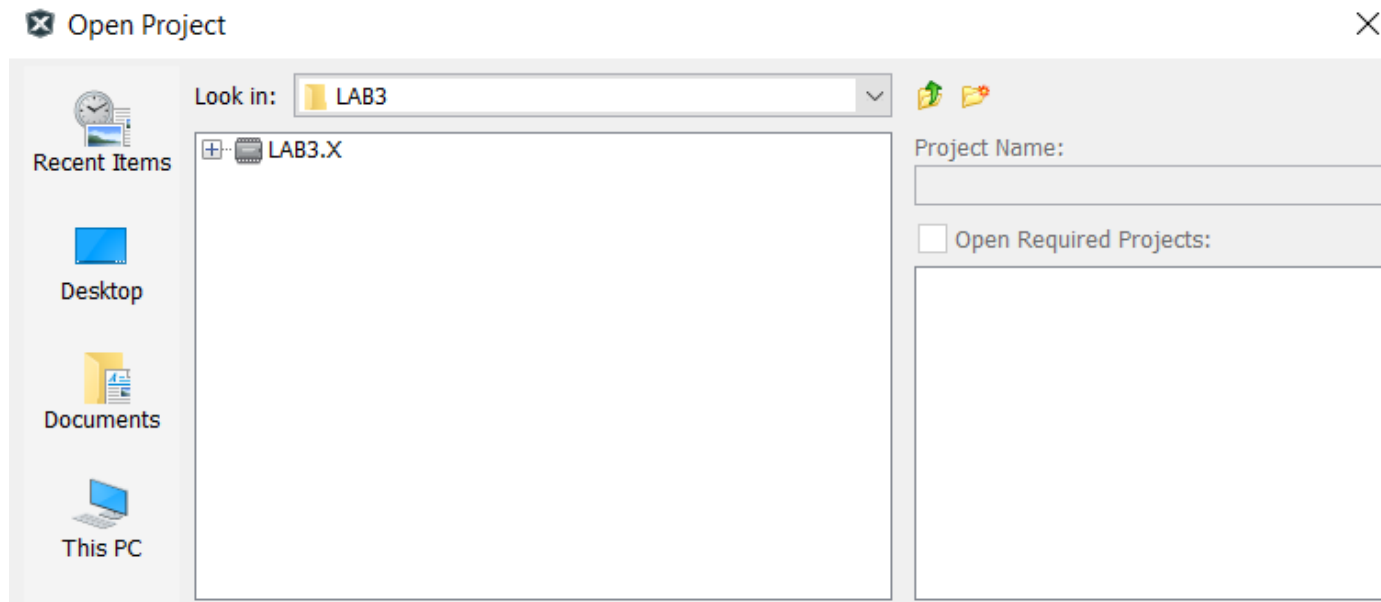
In this section you will learn:

- How to define arrays of structure pointers
- How to initialize the arrays of pointers to point to multiple structures
- How to use pass structure addresses to functions using structure pointers

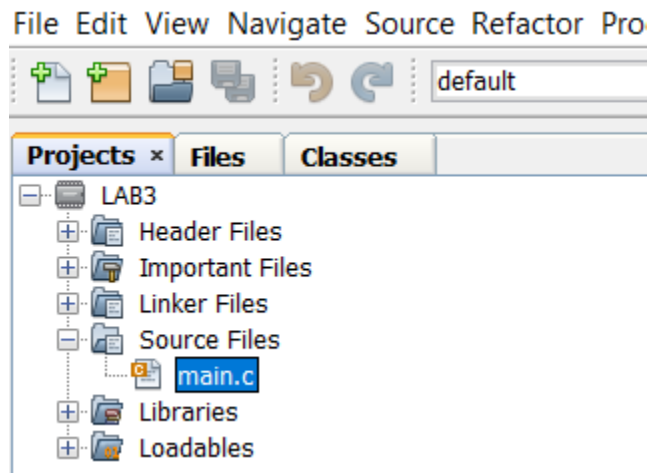


MICROCHIP

Open MPLABX and select File -> Open Project. Navigate to the LAB3 folder and open LAB3.X:



Expand the 'Source Files' selection in the 'Projects' folder to show the "main.c" file:



Double click on the "main.c" file and the file will be opened in the code window. All the code for this project is included in this file.

Code Analysis

The 'typedef' for the structure used in this lab is first created. It has three character members and is type `STRUCT_ALARM_Type`:

```
typedef struct{
    unsigned char u8alarmTimeSeconds;
    unsigned char u8alarmTimeMinutes;
    unsigned char u8alarmTimeHours;
}STRUCT_ALARM_Type;
```

Now, three variables are created. The first is the array of structures with five instances of the structure. The second variable is an array of pointers to the 5 structures. When the pointer is created, it is also initialized to point to the address of each of the structure instances. The third variable is simply a pointer that is initialized to point to the first member of the structure array:

```
STRUCT_ALARM_Type STRUCT_alarmDescriptor[5];
STRUCT_ALARM_Type *p_STRUCT_alarmDescriptor[5] = {&STRUCT_alarmDescriptor[0], &STRUCT_alarmDescriptor[1],
    &STRUCT_alarmDescriptor[2], &STRUCT_alarmDescriptor[3], &STRUCT_alarmDescriptor[4]};
STRUCT_ALARM_Type *p_STRUCT_singlePointer = &STRUCT_alarmDescriptor[0];
```

A function prototype is created for a function that will access the structures either directly or with a pointer. The parameters passed to this function are the pointer to the desired structure along with the individual member values to change:

```
void load_alarm_data_using_pointer(STRUCT_ALARM_Type *p_structure, unsigned char seconds,
    unsigned char minutes, unsigned char hours);
```

Another function prototype provides a method to send the array of function pointers to the function:

```
void load_alarm_data_using_pointer_array(STRUCT_ALARM_Type *p_structure[ ], unsigned char seconds,
    unsigned char minutes, unsigned char hours);
```

These two functions are shown at the bottom of the C file. In the first function, the pointer is passed as one of the function parameters. In the second function, the base index of the array of pointers is sent:

```
void load_alarm_data_using_pointer(STRUCT_ALARM_Type *p_structure, unsigned char seconds,
    unsigned char minutes, unsigned char hours)
{
    p_structure->u8alarmTimeSeconds = seconds;
    p_structure->u8alarmTimeMinutes = minutes;
    p_structure->u8alarmTimeHours = hours;
}
```

```

void load_alarm_data_using_pointer_array(STRUCT_ALARM_Type *p_structure[ ], unsigned char seconds,
                                         unsigned char minutes, unsigned char hours)
{
    unsigned char j;

    for(j = 0; j < 5; j++)
    {
        p_structure[j]->u8alarmTimeSeconds = seconds;
        p_structure[j]->u8alarmTimeMinutes = minutes;
        p_structure[j]->u8alarmTimeHours = hours;
    }
}

```

The 'while()' loop now accesses each of the structure members using various methods:

```

int main(int argc, char** argv)
{
    unsigned char i;

    for(i = 0; i < 5; i++)
    {

        //Load values directly with the pointer
        p_STRUCT_alarmDescriptor[i]->u8alarmTimeMinutes = 1;
        p_STRUCT_alarmDescriptor[i]->u8alarmTimeSeconds = 2;
        p_STRUCT_alarmDescriptor[i]->u8alarmTimeHours = 3;

        //Pass the address to change the actual alarmDescriptor location
        load_alarm_data_using_pointer(&STRUCT_alarmDescriptor[i], 4, 5, 6);

        //Pass the address using the pointer
        load_alarm_data_using_pointer(p_STRUCT_alarmDescriptor[i], 7, 8, 9);

        //Set the structure values using a single pointer
        load_alarm_data_using_pointer(p_STRUCT_singlePointer, 0xA, 0xB, 0xC);

        //Increment the single pointer
        p_STRUCT_singlePointer++;
    }

    //Pass array of pointers to the function
    load_alarm_data_using_pointer_array(&p_STRUCT_alarmDescriptor[0], 77, 88, 99);

    while(1);
}

```

The 'for()' loop increments *i* on each pass and accesses instances 0 through 4 of the structures. The first three instructions load the three members with the values 1, 2, and 3 with the pointer.

The *load_alarm_data_using_pointer()* function call passes the address of the structure index for the current loop. It uses the '&' for the address locator. The first time through the loop, index 0 is accessed. Index 1 is accessed on the next loop, then index 2, etc... This function call loads the values 4, 5, and 6 into the three structure members.


The second *load_alarm_data_using_pointer()* function call passes the pointer to the structure that needs to be accessed. It loads the values 7, 8, and 9 to the structure members.

The third call to the same function passes the single pointer to set the structure values to 0xA, 0xB, and 0xC. This pointer is incremented after the function call so that the pointer will point to the next structure instance on the next pass.

Finally, the call to *load_alarm_data_using_pointer_array()* passes the base address of the array of pointers to the function. This function loads the values 77, 88, and 99 into the seconds, minutes, and hours variables for all the structures. This function demonstrates how to enable a function to see where the entire array resides in memory.

Run the Project

We will analyze the variable access one instruction at a time. Set a breakpoint at the 'for' loop by clicking on the gray bar to the left of the code. A red square appears showing that a breakpoint is set:

```
121   for(i = 0; i < 5; i++)
123  {
124
125  //Load values directly with the pointer
126      p_STRUCT_alarmDescriptor[i]->u8alarmTimeMinutes = 1;
127      p_STRUCT_alarmDescriptor[i]->u8alarmTimeSeconds = 2;
128      p_STRUCT_alarmDescriptor[i]->u8alarmTimeHours = 3;
129
130
131  //Pass the address to change the actual alarmDescriptor location
132      load_alarm_data_using_pointer(&STRUCT_alarmDescriptor[i], 4, 5, 6);
133
134  //Pass the address using the pointer
135      load_alarm_data_using_pointer(p_STRUCT_alarmDescriptor[i], 7, 8, 9);
136
137  //Set the structure values using a single pointer
138      load_alarm_data_using_pointer(p_STRUCT_singlePointer, 0xA, 0xB, 0xC);
139
140  //Increment the single pointer
141      p_STRUCT_singlePointer++;
142  }
143
144  //Pass array of pointers to the function
145      load_alarm_data_using_pointer_array(&p_STRUCT_alarmDescriptor[0], 77, 88, 99);
146
147
148  while(1);
149  }
```

Now click on the 'Debug Project' icon:



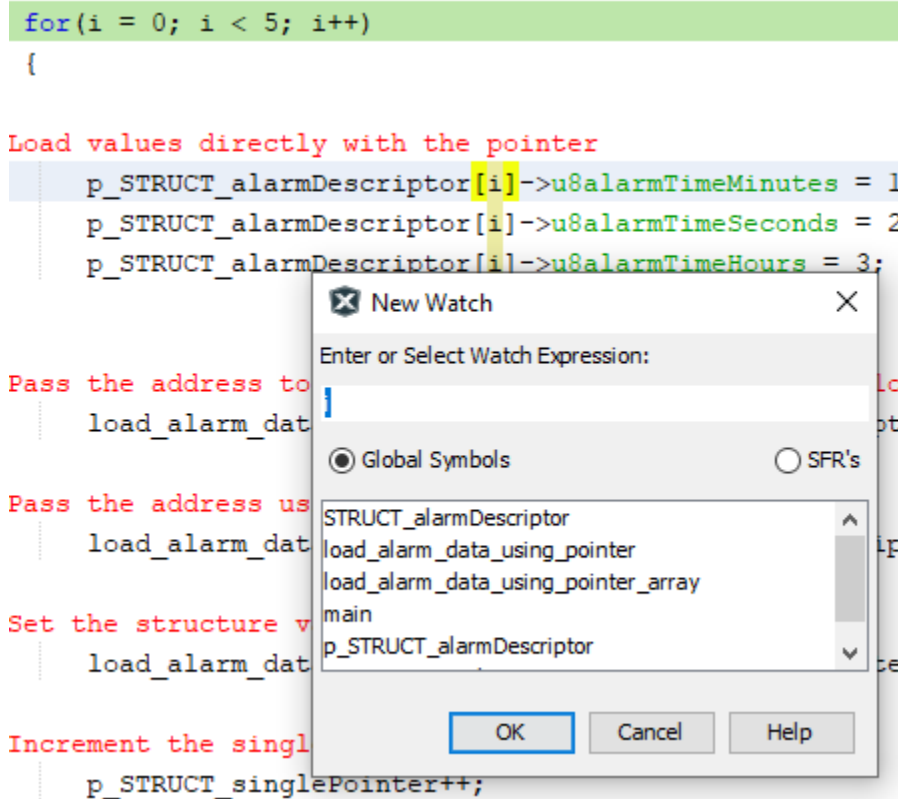
The project will build and the simulator will start code execution. The execution will stop at the breakpoint that was just set. A green arrow appears on top of the square:

```

121
122     for(i = 0; i < 5; i++)
123     {
124
125         //Load values directly with the pointer
126         p_STRUCT_alarmDescriptor[i]->u8alarmTimeMinutes = 1;
127         p_STRUCT_alarmDescriptor[i]->u8alarmTimeSeconds = 2;
128         p_STRUCT_alarmDescriptor[i]->u8alarmTimeHours = 3;
129
130
131         //Pass the address to change the actual alarmDescriptor location
132         load_alarm_data_using_pointer(&STRUCT_alarmDescriptor[i], 4, 5, 6);
133
134         //Pass the address using the pointer
135         load_alarm_data_using_pointer(p_STRUCT_alarmDescriptor[i], 7, 8, 9);
136
137         //Set the structure values using a single pointer
138         load_alarm_data_using_pointer(p_STRUCT_singlePointer, 0xA, 0xB, 0xC);
139
140         //Increment the single pointer
141         p_STRUCT_singlePointer++;
142     }
143
144     //Pass array of pointers to the function
145     load_alarm_data_using_pointer_array(&p_STRUCT_alarmDescriptor[0], 77, 88, 99);
146
147
148     while(1);
149 }

```

Let's look at the variables as we step through the code. We first need to include these variables in the Watch window. We will include these variables using the same method that we used in the previous labs. Position the cursor over the *i* variable and right click. A pop-up window appears. Select "New Watch":




Click 'OK'.

The 'i' variable is now displayed in the Watch window:


Watches x					Variables	Call Stack	Breakpoints	Output
Name		Type	Address		Value			
[x] i		unsigned char	0xA0000FC8		NUL; 0x0			
[+] <Enter new watch>								

Use the same procedure to show the *STRUCT_alarmDescriptor*, *p_STRUCT_alarmDescriptor* and *p_STRUCT_singlePointer* variables. When they appear in the Watch window, click the '+' sign next to the variables to expand them:


Watches x	Variables	Call Stack	Breakpoints	Output	
	Name	Type	Address	Value	
	<input checked="" type="checkbox"/> p_STRUCT_singlePointer	STRUCT_ALARM_Type*	0xA0000200	0xA0000204	
	<input checked="" type="checkbox"/> u8alarmTimeSeconds	unsigned char	0xA0000204	NUL; 0x0	
	<input checked="" type="checkbox"/> u8alarmTimeMinutes	unsigned char	0xA0000205	NUL; 0x0	
	<input checked="" type="checkbox"/> u8alarmTimeHours	unsigned char	0xA0000206	NUL; 0x0	
	<input checked="" type="checkbox"/> p_STRUCT_alarmDescriptor	STRUCT_ALARM_Type*[5]	0xA0000214		
	<input checked="" type="checkbox"/> p_STRUCT_alarmDescriptor[0]	STRUCT_ALARM_Type*	0xA0000214	0xA0000204	
	<input checked="" type="checkbox"/> p_STRUCT_alarmDescriptor[1]	STRUCT_ALARM_Type*	0xA0000218	0xA0000207	
	<input checked="" type="checkbox"/> p_STRUCT_alarmDescriptor[2]	STRUCT_ALARM_Type*	0xA000021C	0xA000020A	
	<input checked="" type="checkbox"/> p_STRUCT_alarmDescriptor[3]	STRUCT_ALARM_Type*	0xA0000220	0xA000020D	
	<input checked="" type="checkbox"/> p_STRUCT_alarmDescriptor[4]	STRUCT_ALARM_Type*	0xA0000224	0xA0000210	
	<input checked="" type="checkbox"/> STRUCT_alarmDescriptor	STRUCT_ALARM_Type[5]	0xA0000204		
	<input checked="" type="checkbox"/> STRUCT_alarmDescriptor[0]	STRUCT_ALARM_Type	0xA0000204	0xA0000204	
	<input checked="" type="checkbox"/> STRUCT_alarmDescriptor[1]	STRUCT_ALARM_Type	0xA0000207	0xA0000207	
	<input checked="" type="checkbox"/> STRUCT_alarmDescriptor[2]	STRUCT_ALARM_Type	0xA000020A	0xA000020A	
	<input checked="" type="checkbox"/> STRUCT_alarmDescriptor[3]	STRUCT_ALARM_Type	0xA000020D	0xA000020D	
	<input checked="" type="checkbox"/> STRUCT_alarmDescriptor[4]	STRUCT_ALARM_Type	0xA0000210	0xA0000210	









Let's step through the code to see exactly what is happening. Click the 'Step Into' icon  four times to step through the first three instructions in the 'for()' loop. Expand the *STRUCT_alarmDescriptor[0]* variable to see that the three member values of the structure have changed:

<input checked="" type="checkbox"/> STRUCT_alarmDescriptor[0]	STRUCT_ALARM_Type	0xA0000204	0xA0000204
<input checked="" type="checkbox"/> u8alarmTimeSeconds	unsigned char	0xA0000204	STX; 0x2
<input checked="" type="checkbox"/> u8alarmTimeMinutes	unsigned char	0xA0000205	SOH; 0x1
<input checked="" type="checkbox"/> u8alarmTimeHours	unsigned char	0xA0000206	ETX; 0x3


Now click the 'Step Over' icon  to step through the function call *load_alarm_data_using_pointer(&STRUCT_alarmDescriptor[i], 4, 5, 6)*. This icon will step through the function call and the program pointer arrow will point to the next instruction. The Watch window now shows that the values have changed to 4, 5, and 6:

<input checked="" type="checkbox"/> STRUCT_alarmDescriptor[0]	STRUCT_ALARM_Type	0xA0000204	0xA0000204
<input checked="" type="checkbox"/> u8alarmTimeSeconds	unsigned char	0xA0000204	EOT; 0x4
<input checked="" type="checkbox"/> u8alarmTimeMinutes	unsigned char	0xA0000205	ENQ; 0x5
<input checked="" type="checkbox"/> u8alarmTimeHours	unsigned char	0xA0000206	ACK; 0x6

Now click the 'Step Over' icon  to step through the function call *load_alarm_data_using_pointer* (*p_STRUCT_alarmDescriptor[i]*, 7, 8, 9). This icon will step through the function and the Watch window now shows that the values have changed to 7, 8, and 9:

  STRUCT_alarmDescriptor[0]	STRUCT_ALARM_Type	 0xA0000204	 0xA0000204
 u8alarmTimeSeconds	unsigned char	 0xA0000204	 BEL; 0x7
 u8alarmTimeMinutes	unsigned char	 0xA0000205	 BS; 0x8
 u8alarmTimeHours	unsigned char	 0xA0000206	 TAB; 0x9

Finally, click the same 'Step Over' icon to step through the function call that passes the *p_STRUCT_singlePointer* value. The values now change to 0xA, 0xB, and 0xC:

  STRUCT_alarmDescriptor[0]	STRUCT_ALARM_Type	 0xA0000204	 0xA0000204
 u8alarmTimeSeconds	unsigned char	 0xA0000204	 \n; 0xa
 u8alarmTimeMinutes	unsigned char	 0xA0000205	 VT; 0xb
 u8alarmTimeHours	unsigned char	 0xA0000206	 \f; 0xc

You can now continue to step through the code for structure indexes 1 through 4. Just expand those variables in the Watch window to see the change in variable values.

After the 'for' loop executes, the *load_alarm_data_using_pointer_array()* function is called. As you step through this function, you will see that all five instances of the structure are changed. The seconds, minutes, and hours variables are changed to 77, 88, and 99. The changes occur for all the structure instances.

Advanced C Programming

Lab4 – Arrays of Function Pointers

Tasks

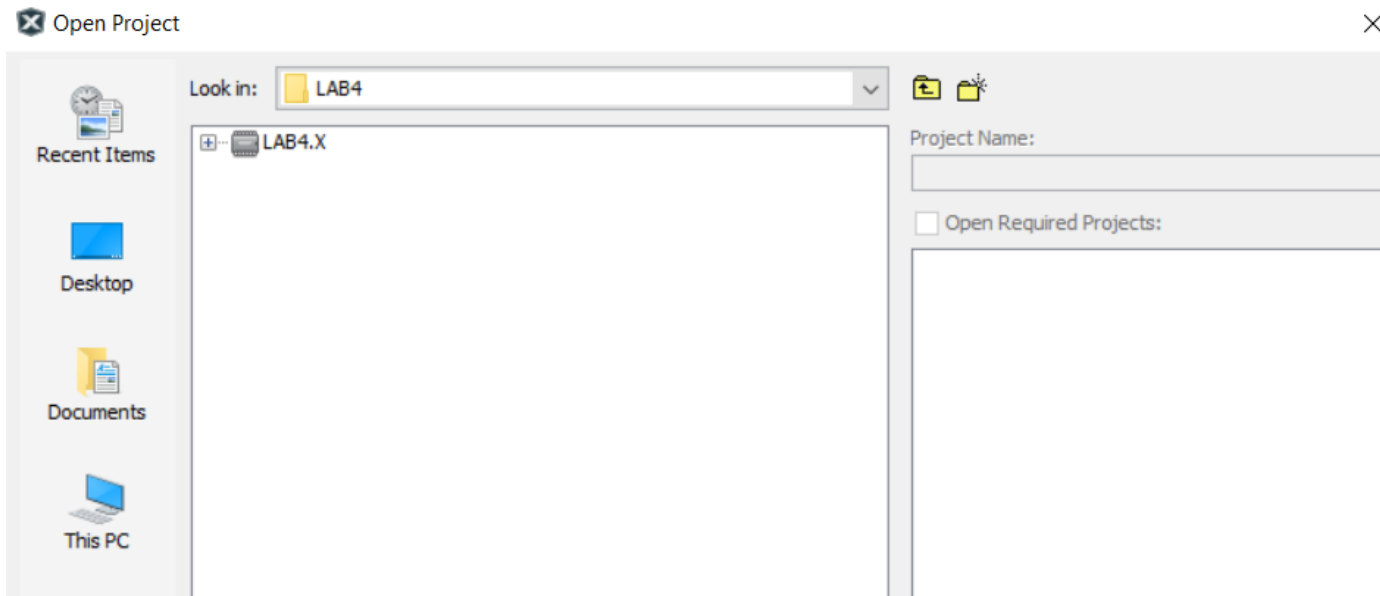
In this section you will learn:

- How to define arrays of function pointers
- How to initialize the arrays of function pointers to point to function locations
- How to use the function pointers to call functions

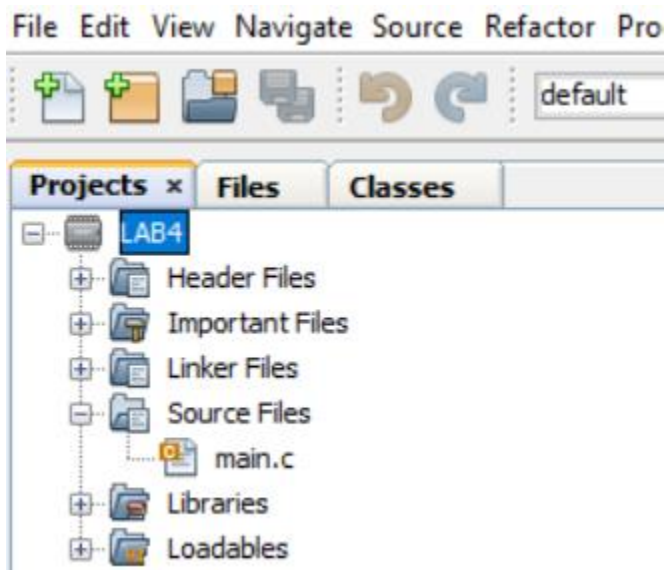


MICROCHIP

Open MPLABX and select File -> Open Project. Navigate to the LAB4 folder and open LAB4.X:



Expand the 'Source Files' selection in the 'Projects' folder to show the "main.c" file:



Double click on the "main.c" file and the file will be opened in the code window. All the code for this project is included in this file.

Code Analysis

Two variables are created for the state machine value and the returned value:

```
unsigned char functionState = 0;
unsigned int  functionResult = 0;
```

The prototypes for the three functions are defined. They each are passed one integer value and return one integer value:

```
int function1(int x);
int function2(int y);
int function3(int z);
```

The array of function pointers is created and initialized so that it contains the addresses of the three functions that it will access. The array has three members and the asterisk signifies that it is a pointer. This pointer is located in data memory:

```
int (*funPtr[3])(int a) = { &function1, &function2, &function3 };
```

Another pointer is set up to demonstrate using pointers that are located in program memory:

```
int (* const funPtr[3])(int a) = { &function1, &function2, &function3 };
```

The three functions shown at the bottom of the file are the functions that are called using the function pointer. Each function returns a different value:

```
//Returns the value passed to the function
int function1(int x)
{
    return x;
}

//Returns the square of the value passed to the function
int function2(int y)
{
    return y * y;
}

//Returns the cube of the value passed to the function
int function3(int z)
{
    return z * z * z;
}
```

The 'main()' routine contains the 'while(1)' loop that calls the different functions using the pointer:

```
//Main routine
int main(void)
{
    //The functionState variable is incremented after each indirect function call so that
    // the succeeding function is called next. In each function call, a value of 2 is passed
    // to the function
    while(1)
    {
        //Calls function1
        functionState = 0;
        functionResult = funPtr[functionState] (2);

        //Calls function 2
        functionState++;
        functionResult = funPtr[functionState] (2);

        //Calls function 3
        functionState++;
        functionResult = funPtr[functionState] (2);

        functionState = 0;

        //Now use function pointers in program memory

        //Calls function1
        functionState = 0;
        functionResult = funPtrFlash[functionState] (2);

        //Calls function 2
        functionState++;
        functionResult = funPtrFlash[functionState] (2);

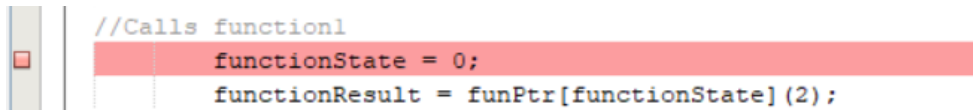
        //Calls function 3
        functionState++;
        functionResult = funPtrFlash[functionState] (2);

        functionState = 0;
    }
}
```

The *functionState* variable starts at 0 and is incremented after each function call. This variable is the index of the function pointer array. The function pointer first points to function1 and calls that function. It then calls function2, then function3. The next three function calls use function pointers that are located in flash and call the same three functions.

Run the Project

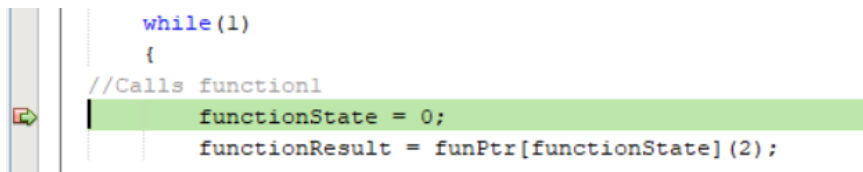
We will analyze the variable access one instruction at a time. Set a breakpoint at the first instruction within the 'while(1)' loop by clicking on the gray bar to the left of the code. A red square appears showing that a breakpoint is set:



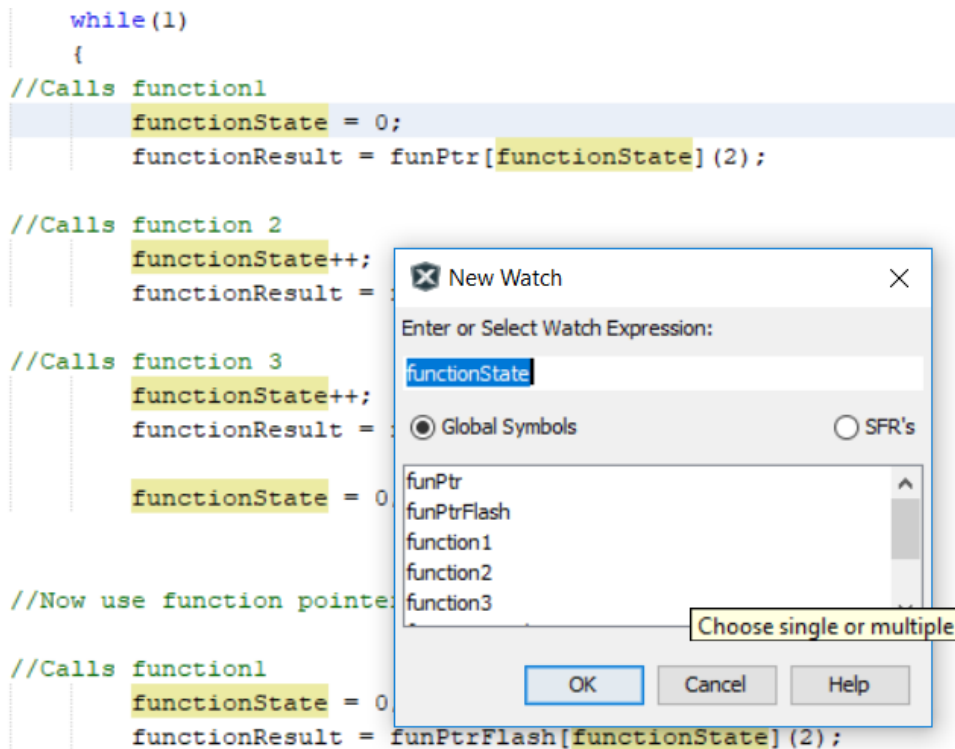
Now click on the 'Debug Project' icon:



The project will build and the simulator will start code execution. The execution will stop at the breakpoint that was just set. A green arrow appears on top of the square:








Let's look at the variables as we step through the code. We first have to include these variables in the Watch window. We will include these variables using the same method that we used in the previous labs. Position the cursor over the *functionState* variable and right click. A pop-up window appears. Select "New Watch":
























Click 'OK'.

The 'functionState' variable is now displayed in the Watch window:

Search Results	Output	Watches x	Variables	Breakpoints
	Name	Type	Address	Value
	<input checked="" type="checkbox"/>  functionState	unsigned char	0xA0000200	NUL; 0x0
	 <Enter new watch>			

Use the same procedure to show the *funPtr* variable. Click the '+' sign next to the variable to expand it to show each member of the array:


Search Results	Output	Watches x	Variables	Breakpoints	
	Name	Type	Address	Value	
	<input checked="" type="checkbox"/>  functionState	unsigned char	 0xA0000200	 NUL; 0x0	
	<input checked="" type="checkbox"/>  funPtr	int*[3]	 0xA0000208		
	 funPtr[0]	int*	 0xA0000208	 0x9D000840	
	 funPtr[1]	int*	 0xA000020C	 0x9D000868	
	 funPtr[2]	int*	 0xA0000210	 0x9D00089C	

Use the same procedure to show the *function1*, *function2*, and *function3* addresses in the Watch window. This would not normally be used for debugging purposes but is only done here so that addresses can be compared:

Search Results	Output	Watches x	Variables	Breakpoints	
	Name	Type	Address	Value	
<input checked="" type="checkbox"/>	functionState	unsigned char	... 0xA0000200	... NUL; 0x0	
<input checked="" type="checkbox"/>	funPtr	int*[3]	... 0xA0000208	...	
<input checked="" type="checkbox"/>	funPtr[0]	int*	... 0xA0000208	... 0x9D000840	
<input checked="" type="checkbox"/>	funPtr[1]	int*	... 0xA000020C	... 0x9D000868	
<input checked="" type="checkbox"/>	funPtr[2]	int*	... 0xA0000210	... 0x9D00089C	
<input checked="" type="checkbox"/>	function1	function	... 0x9D000840	... 0x27BDFFF8	
<input checked="" type="checkbox"/>	function2	function	... 0x9D000868	... 0x27BDFFF8	
<input checked="" type="checkbox"/>	function3	function	... 0x9D00089C	... 0x27BDFFF8	

Note that *funPtr[0]* value is the same as the address of *function1*. *funPtr[1]* contains the address of *function2* and *funPtr[2]* contains the address of *function3*.

Let's step through the code to see exactly what is happening. Add the *functionResult* variable to the Watch

window so that we can see the return value from each function. Now, click the 'Step Into' icon  two times and the *function1* function is called. When it returns to the calling program, the value of 2 is returned. The *functionState* variable is incremented to point to the next function. Continue to step through the instructions to see each function that is called by the function pointer.

To place the array of function pointers in program memory, comment out the function pointer declaration with the attribute that places it in program memory. Comment out the existing function pointer declaration.

The array of function pointers *funPtrFlash[]* is placed in program memory. You can add this variable to the Watch window and step through the function calls to see that they call the same functions that we just described.

Advanced C Programming

Lab5 –Function Pointer State Machines

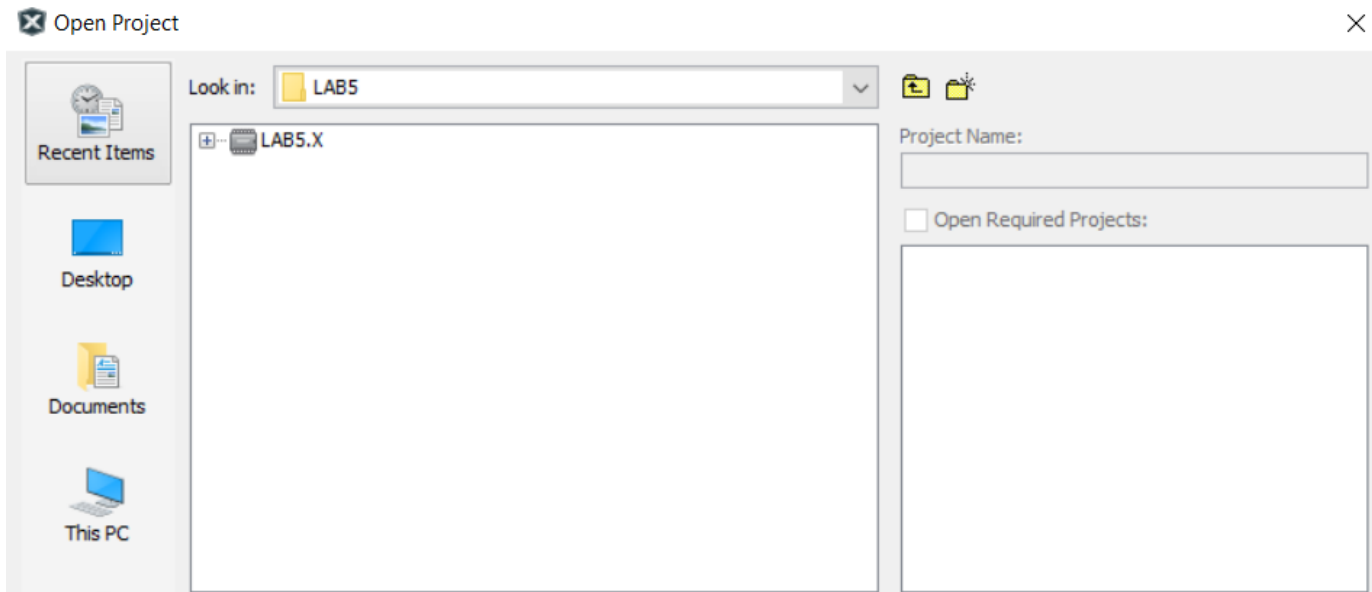
Tasks

In this section you will learn:

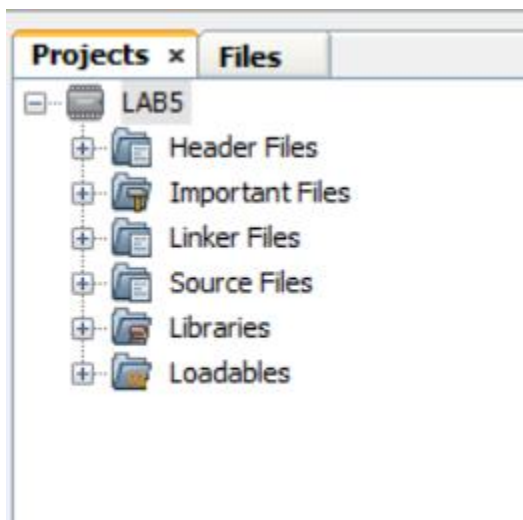
- How to define an enumeration as a state machine selection
- How to use one function for state machine control



Open MPLABX and select File -> Open Project. Navigate to the LAB5 folder and open LAB5.X:



Expand the 'Source Files' selection in the 'Projects' folder to show the "main.c" file:



Double click on the "main.c" file and the file will be opened in the code window. All the code for this project is included in this file.

Code Analysis

An enumeration is defined to create the condition for the state machine. First, the typedef is created for `SYSTEM_STATE`, then the actual instance is created:

```
typedef enum
{
    STATE0 = 0,
    STATE1,
    STATE2
} SYSTEM_STATE;

SYSTEM_STATE gSystemState;
```

The three functions that will be called from the state machine are now defined. Each function is passed two parameters: the first is an unsigned integer value and the second is the state machine variable. If the *InputBits* value in *function1* is 0 and the *State* variable is greater than 1, the *InputBits* value is returned. Otherwise a value of 99 is returned. Function 2 returns the square of the *InputBits* value if it has a value of 2 and the *State* variable is greater than 1. Otherwise a value of 88 is returned. Finally, in *function3*, the cube of the *InputBits* value is returned if its' value is 0 and the *State* variable is greater than 1. Otherwise, a value of 77 is returned.

In all three functions, the *State* variable is incremented so that the next state in the state machine is executed.

```

unsigned int function1(unsigned int InputBits, SYSTEM_STATE *State)
{
    (*State)++;

    if(InputBits == 0 && *State > 1)
        return InputBits;
    else
        return 99;
}

```

```

unsigned int function2(unsigned int InputBits, SYSTEM_STATE *State)
{
    (*State)++;

    if(InputBits == 2 && *State > 1)
        return InputBits * InputBits;
    else
        return 88;
}

```

```

unsigned int function3(unsigned int InputBits, SYSTEM_STATE *State)
{
    (*State)++;

    if(InputBits == 0 && *State > 1)
        return InputBits * InputBits * InputBits;
    else
        return 77;
}

```

An array of function pointers is now created. This pointer points to functions that return an unsigned integer variable and are passed an unsigned integer value and a variable that points to a *SYSTEM_STATE* variable. It is also initialized with the addresses of *function1*, *function2*, and *function3*:

```

unsigned int ( * pStateFunction[] ) ( unsigned int InputBits, SYSTEM_STATE *State ) =
{
    function1,
    function2,
    function3
};

```

Now the *RunStateMachine()* function is created. This function will use the function pointer to call the proper function in order. The pointer calls the function based on the *gSystemState* variable. It also passes the address of the *gSystemState* variable so that the function that is called can increment the state. This function also passes the *InputBits* variable to the function being called. The value from each function is returned to the *main()* function running the state machine.

```
unsigned int RunStateMachine( unsigned int InputBits )
{
    return (*pStateFunction[ gSystemState ])( InputBits, &gSystemState );
}
```

The *main()* function runs the state machine. The *gSystemState* variable is initialized to 0. The *RunStateMachine()* function is now called and the returned value from each function is saved in a result register. Each state execution passes a value to the *RunStateMachine()* function. It is now simple to add functions to the state machine.

```
void main(void)
{
    volatile unsigned int result1;
    volatile unsigned int result2;
    volatile unsigned int result3;

    while(1)
    {
        gSystemState = STATE0;
        result1 = RunStateMachine( 1 );

        //gSystemState should be incremented by function
        result2 = RunStateMachine( 2 );

        //gSystemState should be incremented by function
        result3 = RunStateMachine( 3 );
    }
}
```

Run the Project

We will analyze the variable access one instruction at a time. Set a breakpoint at the first *RunStateMachine()* function call instruction within the 'while(1)' loop by clicking on the gray bar to the left of the code. A red square appears showing that a breakpoint is set:

```
while(1)
{
    gSystemState = STATE0;
    result1 = RunStateMachine( 1 );

    //gSystemState should be incremented by function
    result2 = RunStateMachine( 2 );

    //gSystemState should be incremented by function
    result3 = RunStateMachine( 3 );
}
```

Now click on the 'Debug Project' icon: 

The project will build and the simulator will start code execution. The execution will stop at the breakpoint that was just set. A green arrow appears on top of the square:

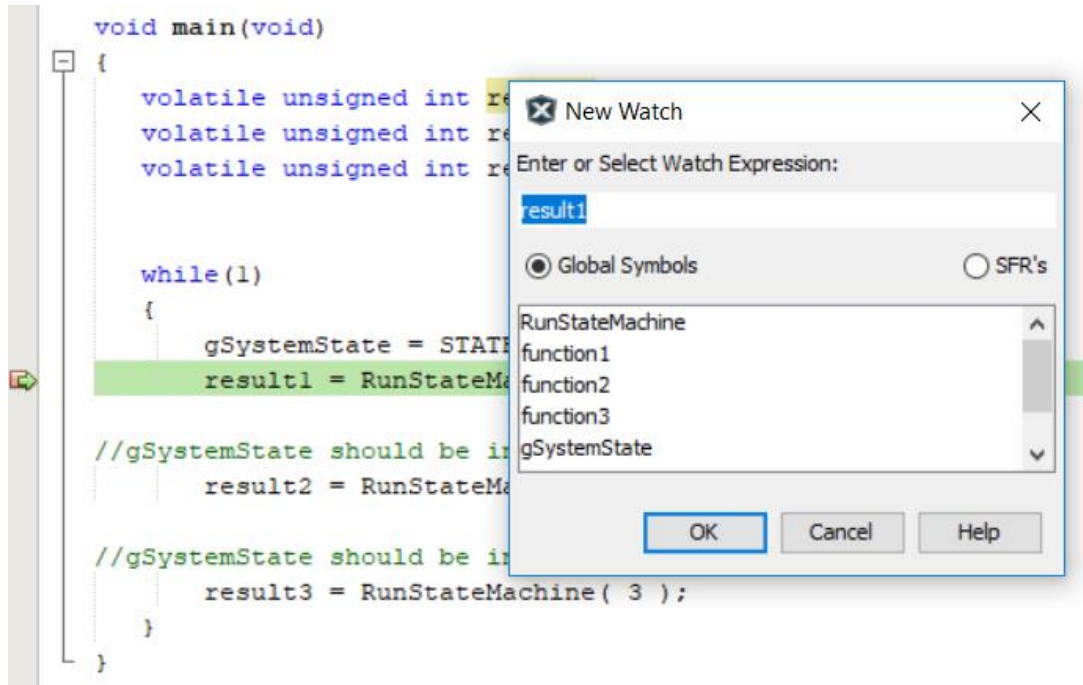
```
void main(void)
{
    volatile unsigned int result1;
    volatile unsigned int result2;
    volatile unsigned int result3;

    while(1)
    {
        gSystemState = STATE0;
        result1 = RunStateMachine( 1 );

        //gSystemState should be incremented by function
        result2 = RunStateMachine( 2 );

        //gSystemState should be incremented by function
        result3 = RunStateMachine( 3 );
    }
}
```

Let's look at the variables as we step through the code. We first include these variables in the Watch window. We will include these variables using the same method that we used in the previous labs. Position the cursor over the *result1* variable and right click. A pop-up window appears. Select "New Watch":



Click 'OK'.


The *result1* variable is now displayed in the Watch window:

Watches				
Name	Type	Address	Value	
<input checked="" type="checkbox"/> result1	unsigned int	0xA0000FC8	...	0x00000000
<input type="checkbox"/> <Enter new watch>	

Use the same procedure to show the *result2* and *result3* variables. Also show the *pStateFunction[]* variable and expand the listing to show all members of the array:

Output x Watches x				
	Name	Type	Address	Value
<input checked="" type="checkbox"/>	result1	unsigned int	0xA0000FC8	0x00000000
<input checked="" type="checkbox"/>	result2	unsigned int	0xA0000FCC	0x00000000
<input checked="" type="checkbox"/>	result3	unsigned int	0xA0000FD0	0x00000000
<input checked="" type="checkbox"/>	pStateFunction	unsigned int*[3]	0xA0000204	...
<input checked="" type="checkbox"/>	pStateFunction[0]	unsigned int*	0xA0000204	0x9D000780
<input checked="" type="checkbox"/>	pStateFunction[1]	unsigned int*	0xA0000208	0x9D0007EC
<input checked="" type="checkbox"/>	pStateFunction[2]	unsigned int*	0xA000020C	0x9D00086C

As we saw in the previous lab, the function pointer array contains the addresses of each function.

Now, click the 'Step Into' icon . The simulator will step into the *RunStateMachine()* function.

```

18  unsigned int RunStateMachine( unsigned int InputBits )
19  {
20      return (*pStateFunction[ gSystemState ])( InputBits, &gSystemState );
21  }

```

Step two more times until the *function1()* is called. .

```

    unsigned int function1(unsigned int InputBits, SYSTEM_STATE *State)
    {
        (*State)++;

        if(InputBits == 0 && *State > 1)
            return InputBits;
        else
            return 99;
    }

```

The function *function1()* was called since it is the first address in the function pointer array. This function will increment the *State* variable. A value of 0 was passed to the *InputBits* variable but *State* is equal to one. Therefore, the value of 99 will be returned. Step through the code until execution returns to the 'main()' function. The variable *result1* will be equal to 99. Step through the remainder of the code to see the returned values from *function2()* and *function3()*. *function2()* will return a value of 4 and *function3()* will return a value of 77.

Advanced C Programming

Lab6 –Function Pointers for Code Portability

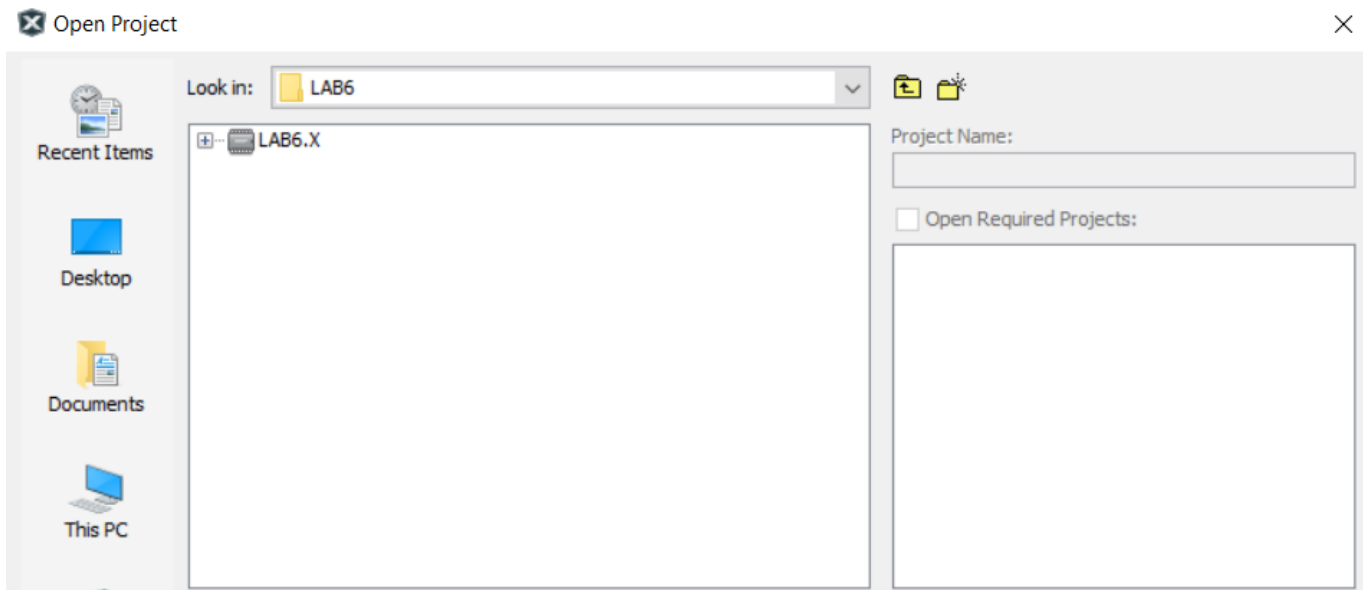
Tasks

In this section you will learn:

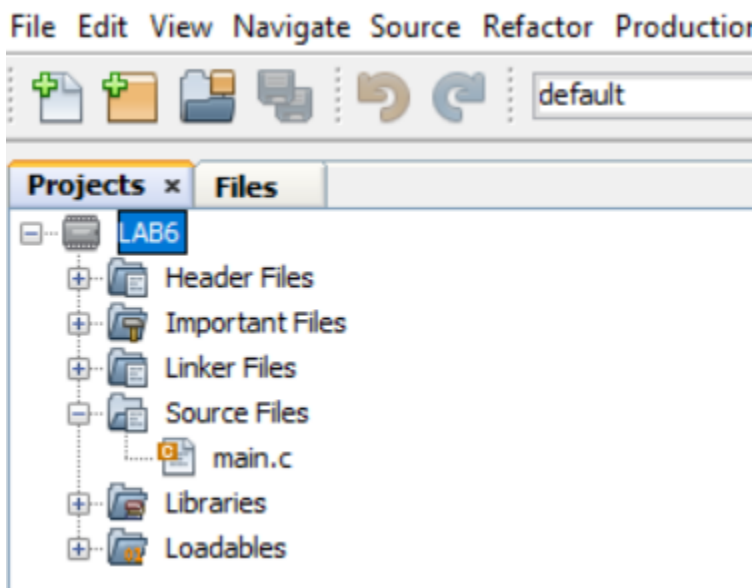
- How to create structures of function pointers
- How to associate each structure member with different function calls
- How to use arrays to group function addresses
- How to use arrays with structure members to call functions



Open MPLABX and select File -> Open Project. Navigate to the LAB5 folder and open LAB6.X:



Expand the 'Source Files' selection in the 'Projects' folder to show the "main.c" file:



Double click on the "main.c" file and the file will be opened in the code window. All the code for this project is included in this file.

Code Analysis

Three variables are created. Two of the variables are arrays that hold the *BUFFER1* and *BUFFER2* values (*u16buffer1[16]* and *u16buffer2[16]*). The third variable is just used to read returned values from functions later on.

```
unsigned short u16buffer1[16];
unsigned short u16buffer2[16];
unsigned short u16dataBufferRead;
```

Three functions are defined for each of the two buffers. Here are the functions for *BUFFER1*. The functions for *BUFFER2* are identical. The *BUFFER1_Initialize()* function takes one parameter and programs all sixteen integer locations with this value. The *BUFFER1_Read()* function takes one parameter which is the index of the buffer to read and the function returns the read value. The *BUFFER1_Write()* function takes one index parameter and the value to write.

```
void BUFFER1_Initialize(unsigned short u16bufferFill)
{
    unsigned char i;

    for(i = 0; i <= 15; i++)
    {
        u16buffer1[i] = u16bufferFill;
    }
}

//Read one 16-bit value from BUFFER1 at the address specified by u8index
unsigned short BUFFER1_Read(unsigned char u8index)
{
    return u16buffer1[u8index];
}

//Write one 16-bit value to BUFFER1 at the address specified by u8index
void BUFFER1_Write(unsigned char u8index, unsigned short u16value)
{
    u16buffer1[u8index] = u16value;
}
```

An enumeration is created to provide the buffer references. The members will be used later to access each buffer:

```
enum { BUFFER1, BUFFER2} buffer_configurations_t;
```

The structure typedef contains three members and each are function pointers. We will see later how the first member is the pointer to the *BUFFERx_Initialize()* functions, the second member is the pointer to the *BUFFERx_Write()* functions, and the third member points to the *BUFFERx_Read()* functions.

```
typedef struct {
    void (*DataInit)(unsigned short bufferFill);
    void (*DataWrite)(unsigned char index, unsigned short value);
    unsigned short (*DataRead)(unsigned char index);
} buffer_functions_t;
```

Now we need to create a way to access all the functions with a simple interface. An array is created to hold the addresses for all the functions for both buffers. This is a 2 x 3 array with the '2' representing locations for each buffer location and the '3' representing the addresses for each function of the individual buffers. The *BUFFER1* functions are included in the first braced entry and the *BUFFER2* functions are included in the second braced entry. Note that this array is type *buffer_functions_t* which is the same as the structure of function pointers just discussed. So this array is the same as two instances of the above structure. In one instance of the structure, the **DataInit* pointer corresponds to the *BUFFER1_Initialize* function, the **DataWrite* pointer corresponds to the *BUFFER1_Write* function, and the **DataRead* pointer corresponds to the *BUFFER1_Read* function. The second instance of this structure applies to the *BUFFER2* functions.

```
const buffer_functions_t buffer_access[] = {
    {BUFFER1_Initialize, BUFFER1_Write, BUFFER1_Read },
    {BUFFER2_Initialize, BUFFER2_Write, BUFFER2_Read }
};
```

The 'main()' routine simply accesses the individual functions for each buffer:

```
int main(void)
{
    //Initialize both buffers
    buffer_access[BUFFER1].DataInit(0x55);
    buffer_access[BUFFER2].DataInit(0xAA);

    while(1)
    {
        //Access the functions for each buffer using the buffer_access[ ] array
        buffer_access[BUFFER1].DataWrite(4, 0x1234);
        ul6dataBufferRead = buffer_access[BUFFER1].DataRead(4);

        buffer_access[BUFFER2].DataWrite(8, 0x9876);
        ul6dataBufferRead = buffer_access[BUFFER2].DataRead(8);

        while(1);
    }
}
```

It is simple to access each buffer function. The array *buffer_access[]* will access each instance of the structure so that the pointers for one buffer are accessed. *buffer_access[0]* will access buffer 1. *buffer_access[1]* will access buffer 2. To access each of the buffer functions, the '.' is used to access structure members for the initialize, read, and write functions for each buffer.

Run the Project

Set a breakpoint at the first line of code within the 'main()' function by clicking on the gray bar to the left of the code. A red square appears showing that a breakpoint is set:



Now click on the 'Debug Project' icon:



The project will build and the simulator will start code execution. The execution will stop at the breakpoint that was just set. A green arrow appears on top of the square:

```

int main(void)
{
    //Initialize both buffers
    buffer_access[BUFFER1].DataInit(0x55);
    buffer_access[BUFFER2].DataInit(0xAA);

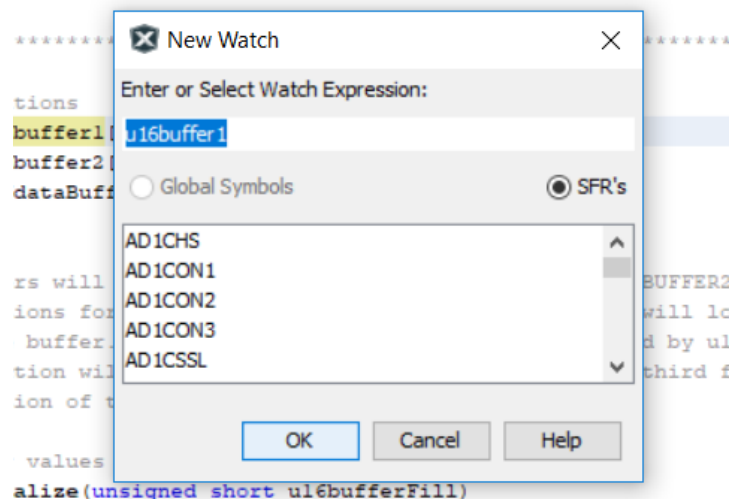
    while(1)
    {
        //Access the functions for each buffer using the buffer_access[ ] array
        buffer_access[BUFFER1].DataWrite(4, 0x1234);
        ul6dataBufferRead = buffer_access[BUFFER1].DataRead(4);

        buffer_access[BUFFER2].DataWrite(8, 0x9876);
        ul6dataBufferRead = buffer_access[BUFFER2].DataRead(8);



        while(1);
    }
}

```


Let's look at the variables as we step through the code. We first will include these variables in the Watch window. We will include these variables using the same method that we used in the previous labs. Position the cursor over the *u16buffer1* variable and right click. A pop-up window appears. Select "New Watch":



Click 'OK'. Do the same for *u16buffer2*.

Name	Type	Address	Value
<input checked="" type="checkbox"/>  u16buffer1	unsigned short[...]	0xA0000228	...
<input checked="" type="checkbox"/>  u16buffer2	unsigned short[...]	0xA0000208	...



Now, click the 'Step Into' icon . The simulator will step into the *BUFFER1_Initialize()* function. You can step through this function and look at the *u16buffer1* entries to see that each location is loaded with the variable 0x55. Step through the code until the execution returns back to the *main()* function.

```

int main(void)
{
    //Initialize both buffers
    buffer_access[BUFFER1].DataInit(0x55);
    buffer_access[BUFFER2].DataInit(0xAA);

    while(1)
    {
        //Access the functions for each buffer using the buffer_access[ ] array
        buffer_access[BUFFER1].DataWrite(4, 0x1234);
        u16dataBufferRead = buffer_access[BUFFER1].DataRead(4);
    }
}

```

You can step through this instruction in the same way to see that buffer2 is loaded with 0xAA.

In both cases, *buffer_access[]* is used to access the desired buffer, then the '.' Notation is used to access the specific function for that buffer. The 'while(1)' loop contains instructions to perform the write and read for each buffer. The access method is similar for all three functions and both buffers.

Advanced C Programming

Lab7 –Double Pointers

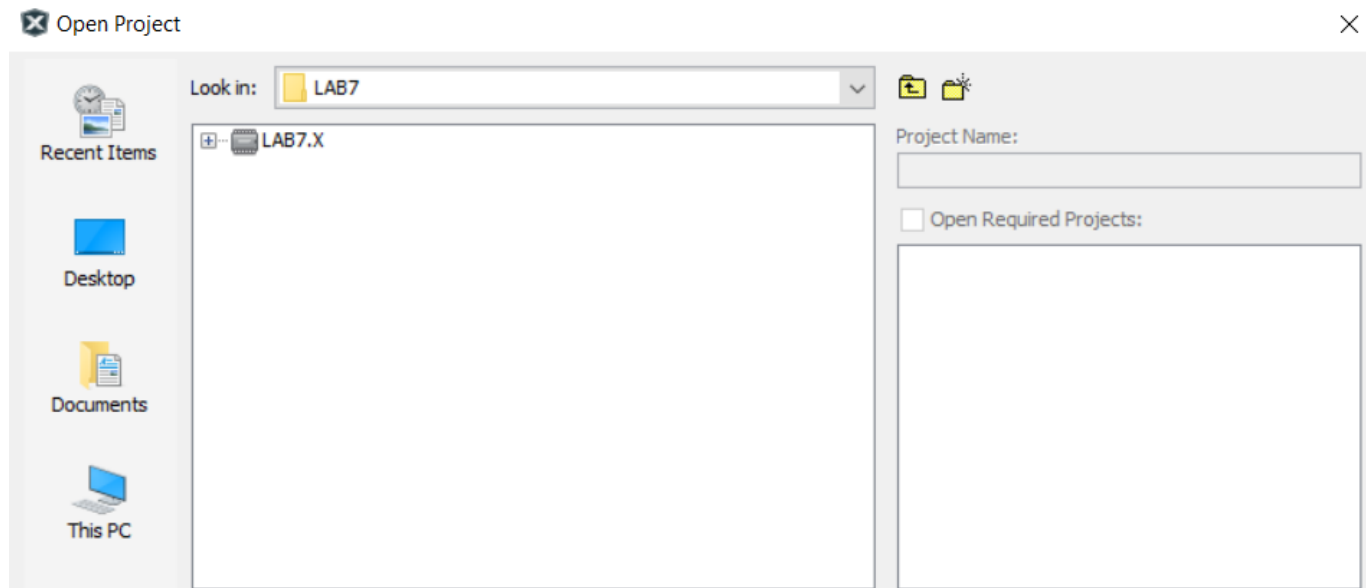
Tasks

In this section you will learn:

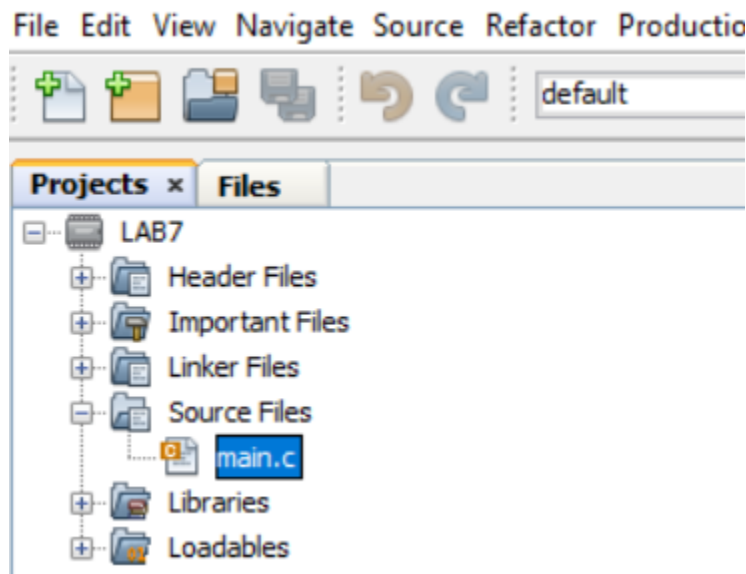
- How to create pointers to pointers (double pointers)
- How to dereference double pointers



Open MPLABX and select File -> Open Project. Navigate to the LAB7 folder and open LAB7.X:



Expand the 'Source Files' selection in the 'Projects' folder to show the "main.c" file:



Double click on the "main.c" file and the file will be opened in the code window. All the code for this project is included in this file.

Code Analysis

The pointer array variable **p[4]* is created to point to 4 different character strings. The array is initialized by declaring the values of each of the four strings:

```
const char *p[4] = {"ALARM", "FAULT", "SENSOR", "UNLOCK"};
```

The “pointer-to-pointer” variable ***pp* is created and initialized to point to the *p* pointer:

```
const char **pp = &p[0];
```

The variable *y* is used to hold the value that holds the characters accessed in the function *string_access()*:

```
unsigned char y;
```

The function *string_access()* takes as a parameter the double pointer that points to the pointer array *p*. The prototype and function definition are shown here:

```
void string_access(const char **message_pointer);
```

```
void string_access(const char **message_pointer)
{
    unsigned char i;

    i = 0;
    do{
        y = *(*message_pointer + i);
        i++;
    } while(y != '\0');
}
```

The *string_access()* function uses the double pointer that it receives and dereferences it to access each character of the character strings. On each pass of the ‘do ... while()’ loop, the *y* variable is set equal to the character that is accessed from the string. Please note how the character is accessed. The **message_pointer* is equal to the *p* pointer index being accessed. The addition of 1 on each loop simply accesses the next character in the string. Finally, the ‘*’ outside of the parentheses dereferences the pointer so that the actual character in the string is accessed. The function returns when a NULL character is accessed at the end of the string.

The 'while(1)' loop sets the value of the double pointer before calling the *string_access()* function. The *pp* double pointer is first initialized with the address of *p[0]* so that it points to the first character string. The address of the double pointer is then passed to the function. Since *pp* is equal to the address that the pointer *p* is pointing to, then **pp* points to the address of *p*. We will send the address of the pointer *p* using the double pointer. This loop increments the double pointer between each function call then calls the function that accesses the string.

Could we have simply incremented the pointer *p* between each function call? Yes, if the function does not need to know the address of the pointer itself. This is a simple example of using the double pointer, but there are many instances where the function may want to change the value of the single pointer itself. It would need to know the address of that pointer, and the double pointer is a proper way to do that.

```
while(1)
{
    //Access "ALARM"
    pp = &p[0];
    string_access(&*pp);

    //Access "FAULT"
    pp++;
    string_access(&*pp);

    //Access "SENSOR"
    pp++;
    string_access(&*pp);

    //Access "UNLOCK"
    pp++;
    string_access(&*pp);

    while(1);
}
```

Run the Project

Set a breakpoint at the first line of code within the 'while(1)' loop by clicking on the gray bar to the left of the code. A red square appears showing that a breakpoint is set:



Now click on the 'Debug Project' icon:



The project will build and the simulator will start code execution. The execution will stop at the breakpoint that was just set. A green arrow appears on top of the square:

```

while(1)
{
//Access "ALARM"
pp = &p[0];
string_access(&*pp);

//Access "FAULT"
pp++;
string_access(&*pp);

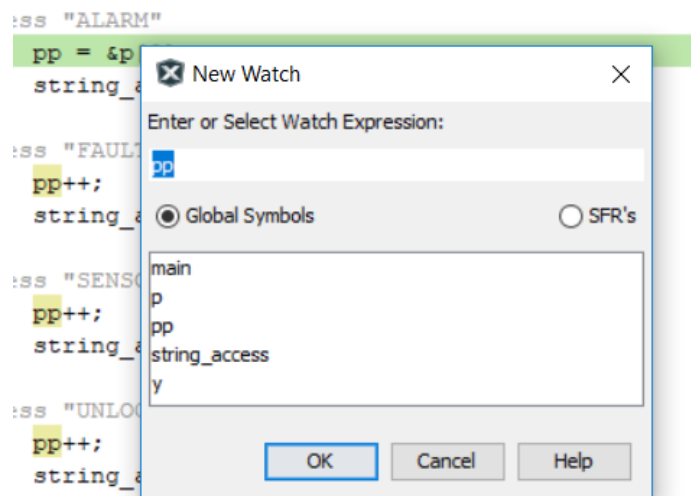
//Access "SENSOR"
pp++;
string_access(&*pp);

//Access "UNLOCK"
pp++;
string_access(&*pp);

while(1);
}

```

Let's look at the variables as we step through the code. We first will include these variables in the Watch window. We will include these variables using the same method that we used in the previous labs. Position the cursor over the *pp* variable and right click. A pop-up window appears. Select "New Watch":



Click 'OK'. Do the same for *p* and *y* variables:

Watches x					Variables	Breakpoints	Output
	Name	Type	Address	Value			
	pp	char**	0xA0000200	0xA0000208			
	p	char*[4]	0xA0000208				
	y	unsigned char	0xA0000204	NUL; 0x0			
	<Enter new watch>						

Click the '+' sign at the front of the *p* and *pp* variables to see all values for these pointers:

Watches x	Variables	Breakpoints	Output	
	Name	Type	Address	Value
	pp	char**	0xA0000200	0xA0000208
	pp	char	0xA0000208	0x9D000B6C
	**pp	char	0x9D000B6C	'A'; 0x41
	p	char*[4]	0xA0000208	
	p[0]	char*	0xA0000208	0x9D000B6C
	*p[0]	char	0x9D000B6C	'A'; 0x41
	p[1]	char*	0xA000020C	0x9D000B74
	*p[1]	char	0x9D000B74	'F'; 0x46
	p[2]	char*	0xA0000210	0x9D000B7C
	*p[2]	char	0x9D000B7C	'S'; 0x53
	p[3]	char*	0xA0000214	0x9D000B84
	*p[3]	char	0x9D000B84	'U'; 0x55
	y	unsigned char	0xA0000204	NUL; 0x0
	<Enter new watch>			

Each of the *p* pointer index values point to the first character of each character string ('A', 'F', 'S', 'U'). **pp* points to the address of the *p[0]* variable. The dereferenced ***pp* shows the value of the 'A' character which is the first character that is being pointed to by *p[0]*.

Now, click the 'Step Into' icon



to step into the function *string_access()* until the pointer is on the instruction:

```
y =>(*message_pointer + i);
```

Now, click the 'Step Into' icon



again and note that the *y* variable is equal to the 'A' ASCII value for the first character of the first string 'ALARM':

<input checked="" type="checkbox"/>		y	unsigned char	0xA0000204	'A'; 0x41
-------------------------------------	--	---	---------------	------------	-----------

Continue to step through the function to see that each character of 'ALARM' is accessed. After the NULL character is accessed, the control returns to the main loop. *pp* is incremented and the function is called again. Step through the code until you reach the same *y = (*message_pointer + i)* instruction. Step into this instruction and note the *y* variable now contains the ASCII 'F' that is the first character of 'FAULT':

<input checked="" type="checkbox"/>		y	unsigned char	0xA0000204	'F'; 0x46
-------------------------------------	--	---	---------------	------------	-----------

Also, note that the *pp* pointer is pointing to *p[1]*:

  *pp	char*	 0xA000020C	 0x9D000B74
  p[1]	char*	 0xA000020C	 0x9D000B74

You can continue to step through the code to access all four strings and note the addresses of the pointers.