

What Is a Function?

An opening word of caution. This is one of the longest lab manuals. Functions are very important to the C programming language, and they are very powerful. However, they can easily be misused, so again, we will go slowly through this, taking it step-by-step.

1. As we learned in the video, **functions** are named independent pieces of code written to perform a task that can also return a value. Although some constraints apply when you wish to create and use a **function**, they can easily serve as building blocks for most of the programs you write.
2. All **functions** in your program must have their own unique name. This is relatively self-explanatory. Much like variables, if two functions had the same name, the microcontroller wouldn't know which one to perform when it was called.
3. As stated earlier, functions are independent blocks of code. Therefore, they perform their specified task separate from the rest of your program. Because of this, they can be called without interfering with the rest of the program.
4. Most functions only perform a small discrete task, usually only a few lines of code at once. For example, you could create a function to turn off your watchdog timer or initialize a timer. By keeping functions short and simple, they become extremely reusable and can become building blocks for your future programs.
5. You can also create functions that have an input or output a value. Therefore, you can create functions that behave like mathematical functions. For example, you could create a function called **cubed** that would take an input value and multiply it 3 times and then return the solution. (We will actually create a **cubed** function shortly.)

6. For now, however, let us take a look at what a program with functions could look like. Below is a program we used when we first started working with the general purpose timers:

When you first started working with programs like this, it is easy to get confused about what things **TA0CTL**, **TAIFG**, and **ACLK** mean and how they are supposed to be used. Sometimes, if I take a week vacation or holiday, after I return, I have to refresh my memory on this stuff, too.

```
#include <msp430.h>

#define RED_LED      0x0001      // P1.0 is the Red LED
#define DEVELOPMENT  0x5A80      // Stop the watchdog timer
#define ENABLE_PINS  0xFFFE      // Required to use inputs and outputs
#define ACLK         0x0100      // Timer_A ACLK source
#define UP           0x0010      // Timer_A UP mode
#define TAIFG        0x0001      // Used to look at Timer A Interrupt FlaG

main()
{
    WDTCTL = DEVELOPMENT;          // Stop the watchdog timer
    PM5CTL0 = ENABLE_PINS;         // Enable inputs and outputs

    TA0CCR0 = 20000;               // We will count up from 0 to 20000
    TA0CTL = ACLK | UP;            // Use ACLK, for UP mode

    P1DIR = RED_LED;              // Set red LED as an output

    while(1)
    {
        if(TA0CTL & TAIFG)         // If timer has counted to 20000
        {
            P1OUT = P1OUT ^ RED_LED; // Then, toggle red P1.0 LED
            TA0CTL = TA0CTL & (~TAIFG); // Count again
        }

    } //end while(1)
} //end main()
```

7. Functions will allow us to rewrite the program to make it more understandable. For example, here is the same program rewritten with functions.

What do you think? Is this one easier to follow?

Most developers strongly believe that programs using functions are easier to work with.

```
main()
{
    stop_watchdog_timer();

    enable_inputs_and_outputs();

    timer0_will_count_up();
    timer0_will_count_for_500ms();

    make_P10_red_LED_and_output();

    while(1)
    {
        if(timer0_500ms_elapsed)
        {
            toggle_red_LED();
            clear_timer0_elapsed_flag();
        }

    } //end while(1)
} //end main()
```

8. Now that we have seen an example, let us take a look at how we create our own functions.

First, functions have three parts: a **function prototype**, a **function call**, and a **function definition**. To create or reuse a **function** in your program, you need to incorporate each of these three parts.

9. We begin with the **function prototype**. This part tells your program that a function exists and will be defined later. If you don't have a prototype and try to use the function, an error will occur. You can declare as many **function prototypes** as you need at the start of your program.

10. The simplest functions have a prototype with the following format:

```
void name(void);
```

Here, the function will be called **name**. The two uses of the word “**void**” refer to the fact that this simplest function will not have an input and it will not generate an output.

11. Here is another example for a function called **cubed**. This time, the function will need an input and it will generate an output. The type of input variable is enclosed in the parentheses after the function name. The type of output variable is listed before the name of the function.

```
signed int cubed(signed char);
```

In this example, the function **cubed** can take inputs of type **signed char** variables (-128 to +127) and will generate an output of type **signed int** (-32,768 to +32,767).

12. Next, we have the **function call**. This is what you use whenever you need to call the function to perform its task.

For example, you could create a function to toggle an LED. Then, every time you wanted to toggle the LED, you could call the function like this:

```
toggle_red_LED();
```

Similarly, we could use our **cubed** function like this. The input will be **6**, and the result will be sent to a **variable** called **answer**.

```
answer = cubed(6);
```

13. You can use a **function call** as many times as you'd like within your program. However, you cannot call a function without a **prototype** or **definition**.

14. Finally, we come to the **function definition**.

The definition is where you write out the instructions for the function's task when it is called. For example, a function that toggles an LED would have an instruction like this inside of its function definition.

```
P1OUT = P1OUT ^ RED_LED;
```

Similarly, our **cubed** function may have an instruction like this:

```
result = input * input * input;
```

15. Here is what the function definition might look like for our **toggle_red_LED();** function:

```
void toggle_red_LED(void)  
{  
    P1OUT = P1OUT ^ BIT0;           // Red LED is connected to P1.0  
}
```

16. To simplify things, you can also omit the word `void` from your function prototypes and function definitions. This is commonly done.

```
toggle_red_LED()  
{  
    P1OUT = P1OUT ^ BIT0;           // Red LED is connected to P1.0  
}
```

17. Similarly, our **cubed** function might look like this:

```
signed int cubed (signed char x)    // Input x is type signed char  
{                                  // Output will be type signed int  
  
    signed int result=0;            // Create variable to hold result  
    result = x*x*x;                 // result equals the cubed value of x  
    return result;                  // Return the value stored in result  
                                    // back to the program  
}
```

18. We will place our function definitions at the bottom of your program **OUTSIDE** of your `main()` function.

19. You can declare as many **function definitions** as you want, even if you don't plan on using them.

20. Let us go ahead and create a program that calls the **cubed** function. Create a new **CCS** project called **cubed**. Copy the following program into the new **main.c** file.

```
#include <msp430.h>                // Need this to easily stop watchdog

//*****
// Function Prototype
//*****
signed int cubed(signed char);      // Has an input  of type signed char
                                    // Has an output of type signed int

//*****
// Main program is here
//*****
main()
{
    WDTCTL = WDTPW | WDTHOLD;      // Stop watchdog timer

    signed int answer=0;            // We will store the result of the
                                    // cubed function in this variable

    answer = cubed(6);              // This is the function call.  It
                                    // sends the value of 6 to the function
                                    // and will put the function's output
                                    // into the variable called answer

    while(1);                      // Stay here forever after done
}

//*****
// Function Defintion
//*****
signed int cubed (signed char x)    // Has an input of type signed char that
{                                    // will be called "x" in the function
                                    // It also has an output of type signed int

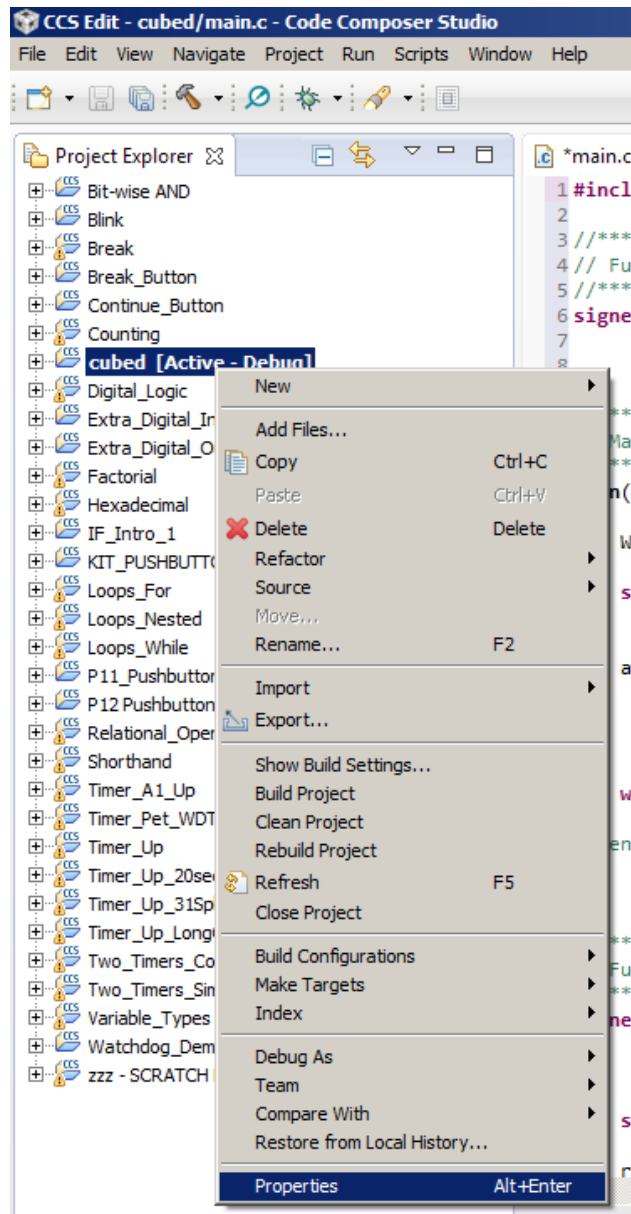
    signed int result=0;            // Clear a variable to hold the result

    result = x*x*x;                // result will be cubed value of input "x"

    return result;                 // send contents of result back to answer
}
```

21. **Caution.** Before you do anything else, turn off the optimization option. If you don't do this, **CCS** will make your program run more efficiently, but you will not be able to see everything happen step-by-step like this tutorial will show.

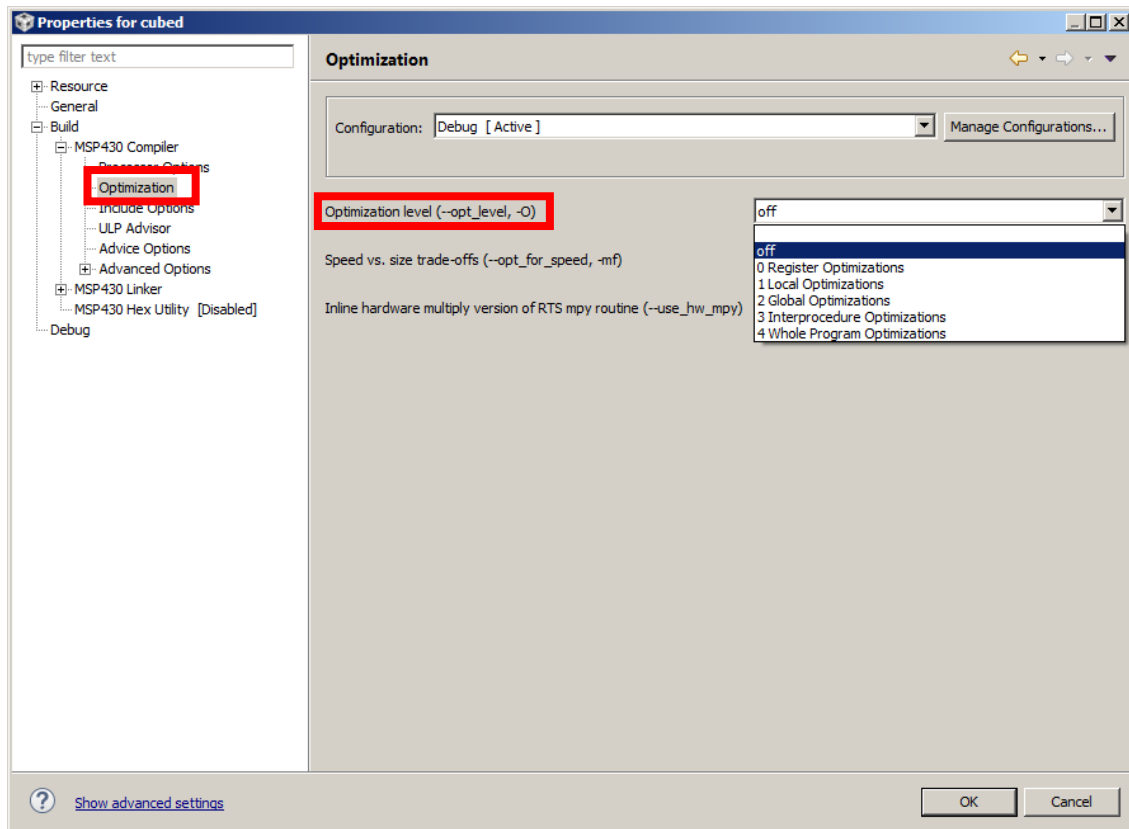
Right click on the project name and select **Properties**.



22. (continued)

From the **Properties for cubed** window, select **Optimization** and **off** for the **Optimization level**.

Click on **OK** when you are done.

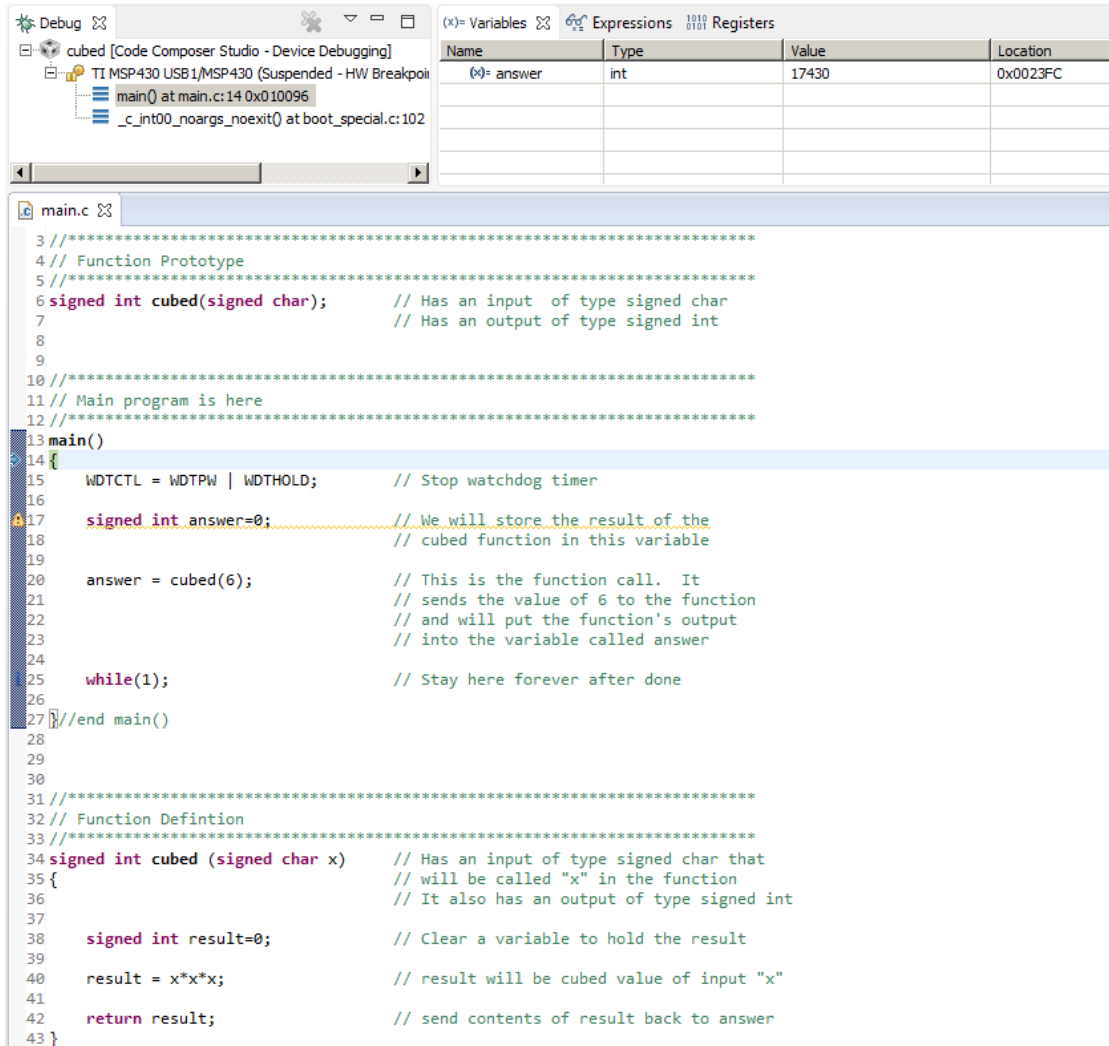


23. After ensuring that the **Optimization level** is **off**, **Save** and **Build** your project.

24. When you are ready, click **Debug**.

25. Make sure that the variable **answer** is shown in the **Variables** pane.

(If it is not shown, click **Terminate** and go back to the **CCS Editor**. Verify your **Optimization level** is **off** and **Build** and **Debug** your project again.)



The screenshot shows the CCS interface with the 'Variables' pane on the right. The variable 'x' (labeled 'answer') is listed with type 'int', value '17430', and location '0x0023FC'. The main.c source code is visible in the editor, showing the function definition for 'cubed' and its use in the 'main' function.

Name	Type	Value	Location
(x)= answer	int	17430	0x0023FC

```

3 //*****
4 // Function Prototype
5 //*****
6 signed int cubed(signed char);    // Has an input of type signed char
7                                  // Has an output of type signed int
8
9
10 //*****
11 // Main program is here
12 //*****
13 main()
14 {
15     WDCTL = WDTPW | WDTHOLD;    // Stop watchdog timer
16
17     signed int answer=0;        // We will store the result of the
18                                // cubed function in this variable
19
20     answer = cubed(6);          // This is the function call. It
21                                // sends the value of 6 to the function
22                                // and will put the function's output
23                                // into the variable called answer
24
25     while(1);                  // Stay here forever after done
26
27 } //end main()
28
29
30
31 //*****
32 // Function Definition
33 //*****
34 signed int cubed (signed char x) // Has an input of type signed char that
35 {                                // will be called "x" in the function
36     // It also has an output of type signed int
37
38     signed int result=0;        // Clear a variable to hold the result
39
40     result = x*x*x;            // result will be cubed value of input "x"
41
42     return result;             // send contents of result back to answer
43 }

```

26. Take a look inside of your function definition. Notice that you have two variables inside of your function definition (**result** and **x**) that are not presently shown in the **Variables** pane.

This may be a little confusing at first, but this is due to a feature of the C programming language called “scope.”

27. You may not have noticed, but every one of our programs has been contained in a function called **main()**.

```
main()
{

}
```

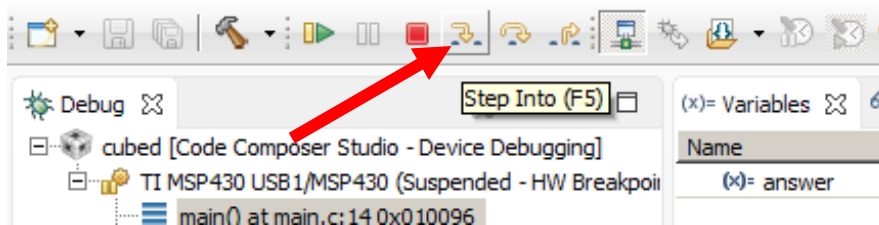
28. Variables are only available within their own function (within their own scope). Therefore, the variables **result** and **x** that are in the **cubed** function cannot be used (or even seen) by the **main()** function.

29. Similarly, we will see in a moment that when the program begins to execute the instructions in the **cubed** function, any variables in **main()**, such as **answer**, cannot be used (or even seen).

30. Let us step through the program line-by-line. As you can see below, the program is ready to start at the beginning of the **main()** function. This is always true. No matter how many functions you may add to your C program, the execution will always start at the beginning of **main()**.

```
3 //*****
4 // Function Prototype
5 //*****
6 signed int cubed(signed char);    // Has an input  of type signed char
7                                   // Has an output of type signed int
8
9
10 //*****
11 // Main program is here
12 //*****
13 main()
14 {
15     WDCTL = WDTNW | WDTOLD;      // Stop watchdog timer
16
17     signed int answer=0;          // We will store the result of the
18                                   // cubed function in this variable
19
20     answer = cubed(6);            // This is the function call. It
21                                   // sends the value of 6 to the function
22                                   // and will put the function's output
23                                   // into the variable called answer
24
25     while(1);                    // Stay here forever after done
26
27 } //end main()
```

31. Click the **Step Into** button.



32. The program moves on to the first instruction: stopping the watchdog. Because this instruction is highlighted, we know that it is the next to be executed.

```

10 //*****
11 // Main program is here
12 //*****
13 main()
14 {
15     WDTCTL = WDTPW | WDTHOLD;    // Stop watchdog timer
16
17     signed int answer=0;          // We will store the result of the
18                                   // cubed function in this variable
19
20     answer = cubed(6);            // This is the function call. It
21                                   // sends the value of 6 to the function
22                                   // and will put the function's output
23                                   // into the variable called answer
24
25     while(1);                    // Stay here forever after done
26
27 }//end main()

```

33. Click **Step Into** to stop the watchdog peripheral. The microcontroller is now ready to assign a value of 0 to the variable **answer**.

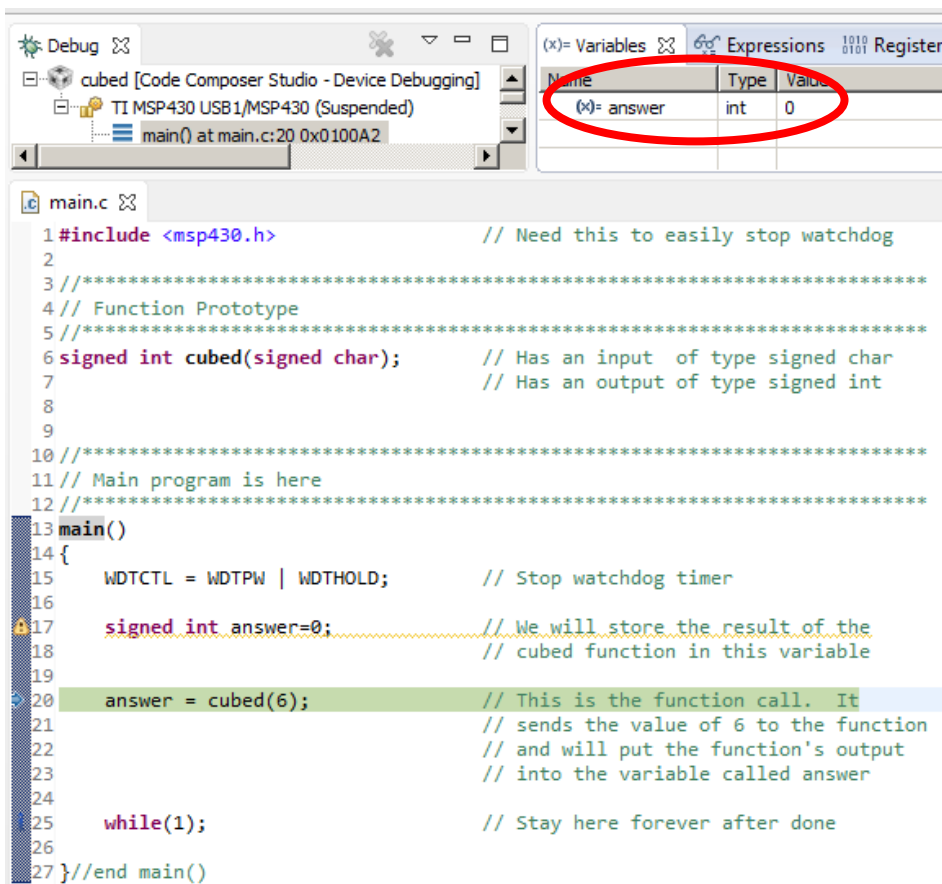
```

10 //*****
11 // Main program is here
12 //*****
13 main()
14 {
15     WDTCTL = WDTPW | WDTHOLD;    // Stop watchdog timer
16
17     signed int answer=0;          // We will store the result of the
18                                   // cubed function in this variable
19
20     answer = cubed(6);            // This is the function call. It
21                                   // sends the value of 6 to the function
22                                   // and will put the function's output
23                                   // into the variable called answer
24
25     while(1);                    // Stay here forever after done
26
27 }//end main()

```

34. Click **Step Into** again. If it was not previously, **answer** now has a value of 0.

The program is now ready to begin the **cubed** function.



The screenshot shows the Code Composer Studio interface. The top panel displays the debug console with the following text:

```

Debug
cubed [Code Composer Studio - Device Debugging]
TI MSP430 USB1/MSP430 (Suspended)
main() at main.c:20 0x0100A2

```

The bottom panel shows the source code for `main.c`. The line `signed int answer=0;` is highlighted, and the variable `answer` is circled in red in the variable window on the right.

Name	Type	Value
(x)= answer	int	0

The source code in the bottom panel is as follows:

```

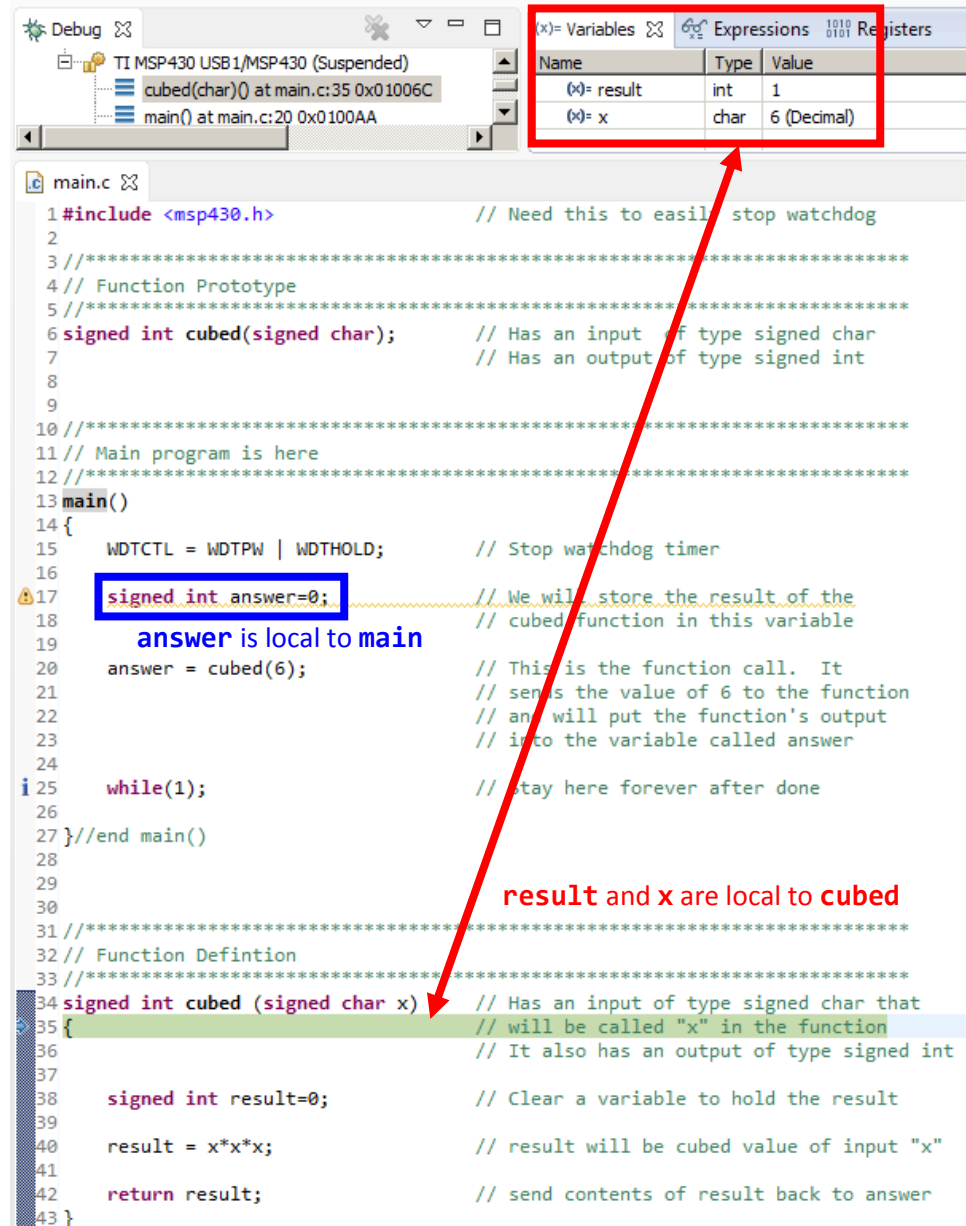
1 #include <msp430.h>                // Need this to easily stop watchdog
2
3 //*****
4 // Function Prototype
5 //*****
6 signed int cubed(signed char);      // Has an input  of type signed char
7                                     // Has an output of type signed int
8
9
10 //*****
11 // Main program is here
12 //*****
13 main()
14 {
15     WDTCTL = WDTPW | WDTHOLD;    // Stop watchdog timer
16
17     signed int answer=0;          // We will store the result of the
18                                   // cubed function in this variable
19
20     answer = cubed(6);            // This is the function call. It
21                                   // sends the value of 6 to the function
22                                   // and will put the function's output
23                                   // into the variable called answer
24
25     while(1);                    // Stay here forever after done
26
27 }//end main()

```

35. Click **Step Into** again. The microcontroller now leaves the **main()** function and goes to the **cubed** function.

Because we are now in the **cubed** function, the variables **result** and **x** are now in scope. We say that **result** and **x** are local to the **cubed** function. Also, notice that the function input from **main()**, **6**, has been assigned to the **cubed** function's input variable, **x**.

Similarly, since **answer** is local to **main()**, it is not presently in scope, and therefore, **CCS** does not show it in the **Variables** pane.



The screenshot shows the CCS IDE with the **Variables** pane open. The **Variables** pane displays the following:

Name	Type	Value
(x)= result	int	1
(x)= x	char	6 (Decimal)

The **main.c** source code is visible below. The **main()** function is shown with the following code:

```

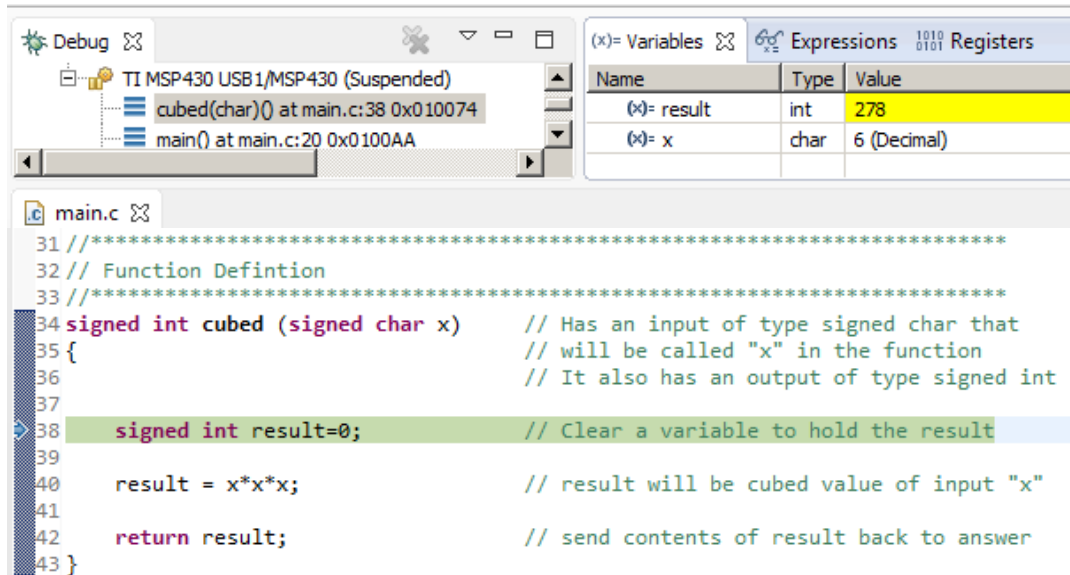
1 #include <msp430.h> // Need this to easily stop watchdog
2
3 //*****
4 // Function Prototype
5 //*****
6 signed int cubed(signed char); // Has an input of type signed char
7 // Has an output of type signed int
8
9
10 //*****
11 // Main program is here
12 //*****
13 main()
14 {
15     WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
16
17     signed int answer=0; // We will store the result of the
18                          // cubed function in this variable
19     answer is local to main
20     answer = cubed(6); // This is the function call. It
21                       // sends the value of 6 to the function
22                       // and will put the function's output
23                       // into the variable called answer
24
25     while(1); // stay here forever after done
26
27 } //end main()
28
29
30
31 //*****
32 // Function Definition
33 //*****
34 signed int cubed (signed char x) // Has an input of type signed char that
35 { // will be called "x" in the function
36     // It also has an output of type signed int
37
38     signed int result=0; // Clear a variable to hold the result
39
40     result = x*x*x; // result will be cubed value of input "x"
41
42     return result; // send contents of result back to answer
43 }

```

A red arrow points from the **Variables** pane to the **cubed** function definition in the code, highlighting the local variables **result** and **x**.

36. Click **Step Into** to begin executing the **cubed** function.

In my example below, the value of **result** was set to a random value as the variable is being created.



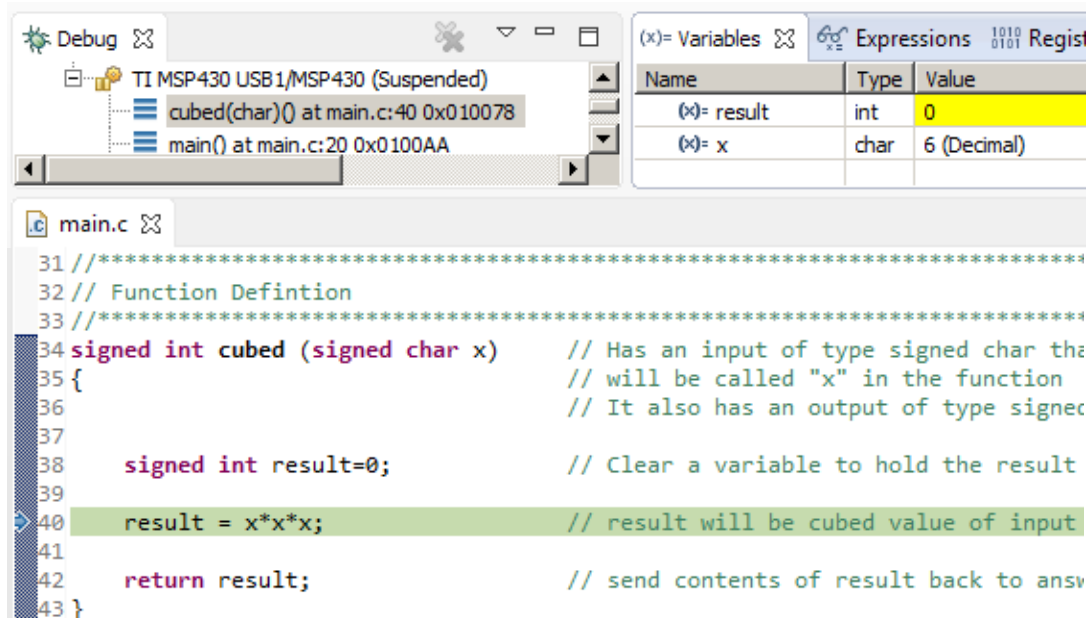
The screenshot shows the IDE with the 'cubed' function being executed. The variable 'result' has a value of 278. The code is as follows:

```

31 //*****
32 // Function Defintion
33 //*****
34 signed int cubed (signed char x)    // Has an input of type signed char that
35 {                                    // will be called "x" in the function
36                                     // It also has an output of type signed int
37
38     signed int result=0;            // Clear a variable to hold the result
39
40     result = x*x*x;                // result will be cubed value of input "x"
41
42     return result;                 // send contents of result back to answer
43 }

```

37. Click **Step Into** again to assign a value of **0** to result. The program is getting ready to calculate **6³** and store the value in **result**.



The screenshot shows the IDE with the 'cubed' function being executed. The variable 'result' has a value of 0. The code is as follows:

```

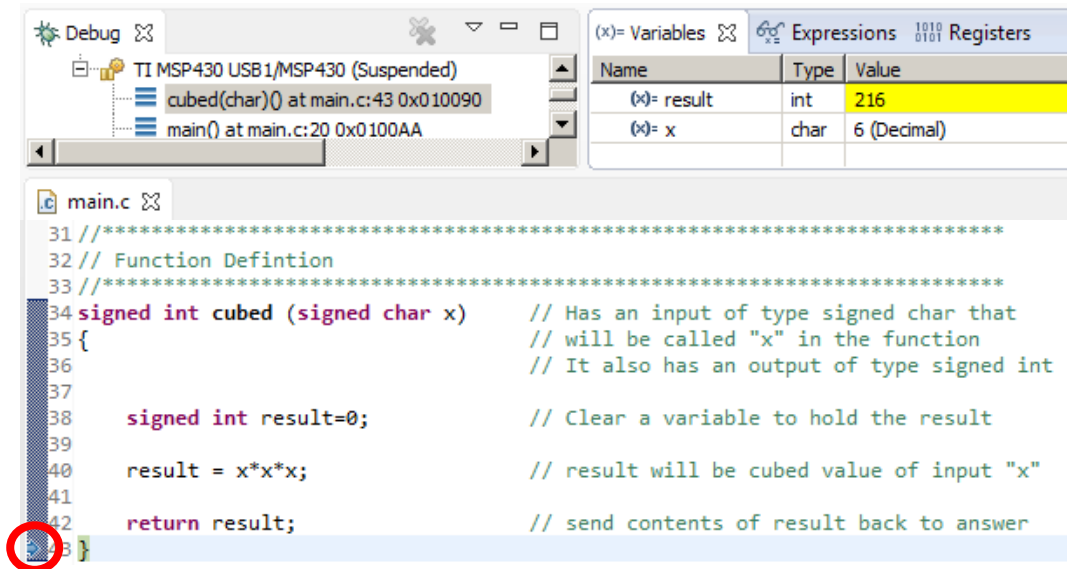
31 //*****
32 // Function Defintion
33 //*****
34 signed int cubed (signed char x)    // Has an input of type signed char tha
35 {                                    // will be called "x" in the function
36                                     // It also has an output of type signed
37
38     signed int result=0;            // Clear a variable to hold the result
39
40     result = x*x*x;                // result will be cubed value of input
41
42     return result;                 // send contents of result back to answ
43 }

```

38. Click **Step Into** again. Since 6^3 is equal to **216**, the correct value is stored in **result**.

Notice the highlighted line indicates that we are still in the **cubed** function. This is why **result** and **x** are still visible.

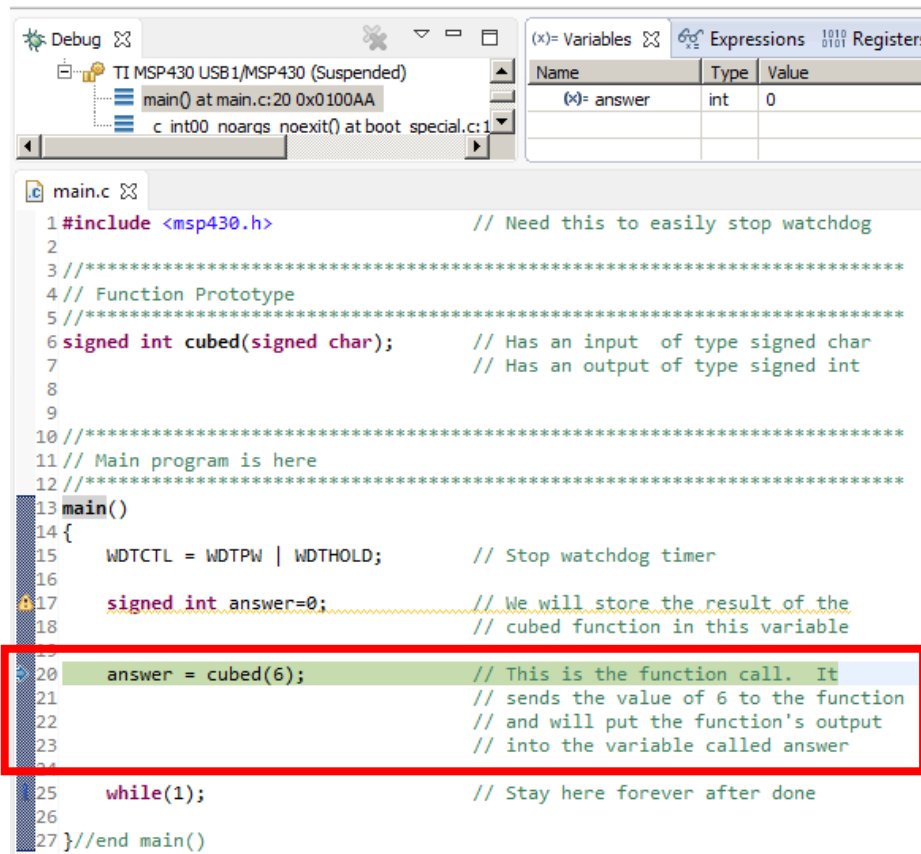
The next time we click **Step Into**, the program will return the value stored in **result** back to the **main()** function.



39. When we click **Step Into** again, the microcontroller has in fact jumped back up to **main()**.

answer is now in scope. **result** and **x** are no longer in scope.

Notice, however, that **answer** has not been updated yet. The Debugger is indicating that line 20 (in my example below) has not finished executing.



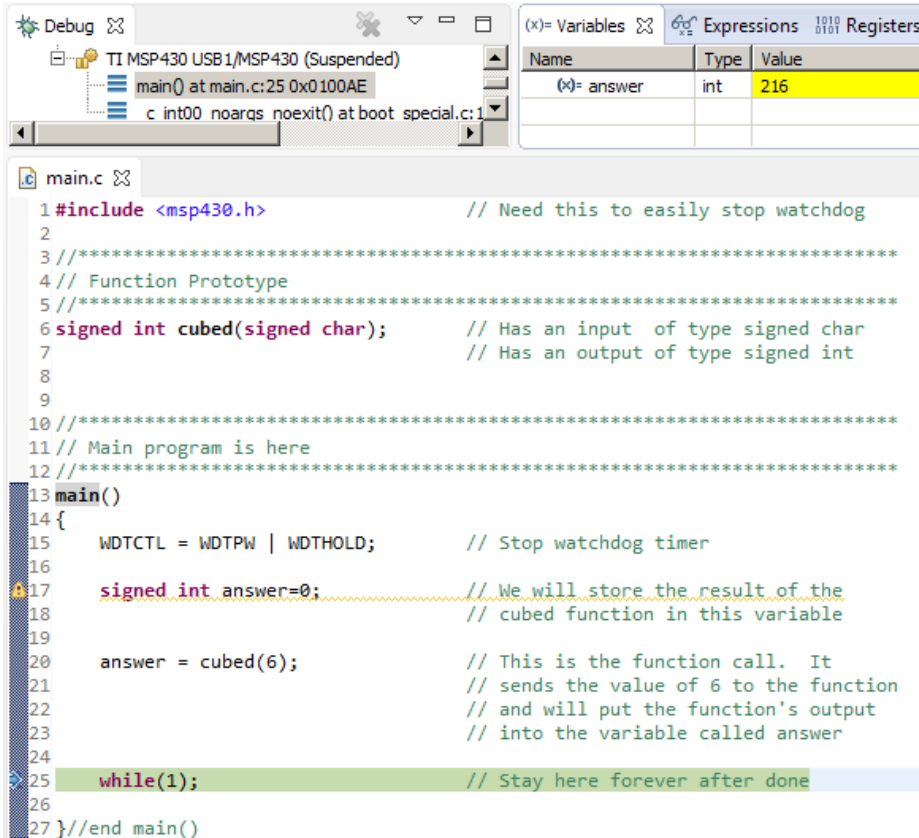
```

1 #include <msp430.h>           // Need this to easily stop watchdog
2
3 //*****
4 // Function Prototype
5 //*****
6 signed int cubed(signed char); // Has an input  of type signed char
7                               // Has an output of type signed int
8
9
10 //*****
11 // Main program is here
12 //*****
13 main()
14 {
15     WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
16
17     signed int answer=0;       // We will store the result of the
18                               // cubed function in this variable
19
20     answer = cubed(6);         // This is the function call. It
21                               // sends the value of 6 to the function
22                               // and will put the function's output
23                               // into the variable called answer
24
25     while(1);                 // Stay here forever after done
26 }
27 //end main()

```

40. We can click **Step Into** a final time and see that the output of the **cubed** function has been moved into the variable **answer**.

Continued clicks of **Step Into** at this point will simply keep the microcontroller in the **while(1);** loop.



The screenshot shows an IDE with the following components:

- Debug Console:** Shows the current execution state: TI MSP430 USB1/MSP430 (Suspended). The call stack indicates the program is at `main() at main.c:25 0x0100AE`.
- Variables Window:** Displays a table of variables:

Name	Type	Value
(x)= answer	int	216
- Source Code (main.c):**

```

1 #include <msp430.h>           // Need this to easily stop watchdog
2
3 //*****
4 // Function Prototype
5 //*****
6 signed int cubed(signed char); // Has an input  of type signed char
7                                // Has an output of type signed int
8
9
10 //*****
11 // Main program is here
12 //*****
13 main()
14 {
15     WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
16
17     signed int answer=0;       // We will store the result of the
18                                // cubed function in this variable
19
20     answer = cubed(6);         // This is the function call. It
21                                // sends the value of 6 to the function
22                                // and will put the function's output
23                                // into the variable called answer
24
25     while(1);                  // Stay here forever after done
26
27 }//end main()

```

41. You do not have to just use a constant (like **6**) as in input for your functions.

The next several steps will walk you through a modified **cubed** function and program, adding some new comments about how you can write your functions.

42. Below, we have modified the previous program slightly. The changes are highlighted.

Specifically, we are showing you that a variable (**q**) can be an input to a function.

Also, we have almost completely rewritten the function itself so that it is just one line long. Now the function does the calculation and returns the output all in the same line. We have eliminated the temporary value result (which can reduce your data memory usage) and a couple of lines of code (which can reduce your program memory usage).

```
#include <msp430.h>                // Need this to easily stop watchdog

//*****
// Function Prototype
//*****
signed int cubed(signed char);      // Has an input of type signed char
                                   // Has an output of type signed int

//*****
// Main program is here
//*****
main()
{
    WDTCTL = WDTPW | WDTHOLD;      // Stop watchdog timer

    signed char q=6;                // Will be the function input

    signed int answer=0;            // We will store the result of the
                                   // cubed function in this variable

    answer = cubed(q);              // This is the function call. It
                                   // sends the value of 6 to the function
                                   // and will put the function's output
                                   // into the variable called answer

    while(1);                       // Stay here forever after done
}

//*****
// Function Definition
//*****
signed int cubed (signed char abc) // Has an input of type signed char that
{                                   // will be called "abc" in this example
    // It also has an output of type signed int

    return abc * abc * abc;        // send contents of result back to answer
}
```

43. If you have not already done so, click **Terminate** in the **CCS Debugger** to return to the **Editor**.

Copy the program in the previous step into your **main.c** file.

Save, **Build**, and **Debug** your project.

44. Try single-stepping through the modified program and verify that it works as before.

When you are ready, click **Terminate** to return to the **CCS Editor**.

45. Ok? That did not seem too bad.

Yes, you could have just written the line of code inside of main to calculate the cubed value of your input. However, as our functions become larger, and as they become more microcontroller peripheral centric, there are real advantages to using functions.

Let me give you my sales pitch....

46. I believe that functions help you better plan, write, and debug your program.

We can use functions to create structured programs – programs that are broken down into small, independent sections of code. Not only that, but the functions can be used and reused over and over again.

47. Not convinced? Let's take a look at a hardware centric example. Here are the two programs we started this lab manual with.

The first uses our original C instructions to toggle the red LED every 20,000 counts on Timer 0.

```
#include <msp430.h>

#define RED_LED      0x0001      // P1.0 is the Red LED
#define DEVELOPMENT  0x5A80      // Stop the watchdog timer
#define ENABLE_PINS  0xFFFE      // Required to use inputs and outputs
#define ACLK          0x0100      // Timer_A ACLK source
#define UP            0x0010      // Timer_A UP mode
#define TAIFG         0x0001      // Used to look at Timer A Interrupt FlaG

main()
{
    WDTCTL = DEVELOPMENT;          // Stop the watchdog timer
    PM5CTL0 = ENABLE_PINS;         // Enable inputs and outputs

    TA0CCR0 = 20000;                // We will count up from 0 to 20000
    TA0CTL = ACLK | UP;            // Use ACLK, for UP mode

    P1DIR = RED_LED;               // Set red LED as an output

    while(1)
    {
        if(TA0CTL & TAIFG)          // If timer has counted to 20000
        {
            P1OUT = P1OUT ^ RED_LED; // Then, toggle red P1.0 LED
            TA0CTL = TA0CTL & (~TAIFG); // Count again
        }

    } //end while(1)
} //end main()
```

47. (continued)

The second uses functions to toggle the red LED every 20,000 counts on Timer 0.

```
main()
{
    stop_watchdog_timer();

    enable_inputs_and_outputs();

    timer0_will_count_up();
    timer0_will_count_for_500ms();

    make_P10_red_LED_and_output();

    while(1)
    {
        if(timer0_500ms_elapsed)
        {
            toggle_red_LED();
            clear_timer0_elapsed_flag();
        }

    } //end while(1)
} //end main()
```

48. Now, for you experts out there, this example of using functions is a little extreme. But, it does illustrate the point of using functions to perform small, independent tasks which can be used and reused.

49. As you may already be saying, the functions program is interesting, but we actually need our function prototypes and definitions to make it all work. Good point! Let us take a look at the whole thing.

To fit the whole program including the function definitions into one longer **main.c** file, we need two pages.

```
#include <msp430.h>

#define RED_LED      0x0001      // P1.0 is the Red LED
#define DEVELOPMENT  0x5A80      // Stop the watchdog timer
#define ENABLE_PINS  0xFFFE      // Required to use inputs and outputs
#define ACLK          0x0100      // Timer_A ACLK source
#define UP            0x0010      // Timer_A UP mode
#define TAIFG         0x0001      // Used to look at Timer A Interrupt FlaG

/*****
*** Function Prototypes *****/
/*****
void stop_watchdog_timer(void);           // These functions do not have
void enable_inputs_and_outputs(void);     // an input or an output
void timer0_will_count_up_for_500ms(void);
void make_P10_red_LED_an_output(void);
void toggle_red_LED(void);
void clear_timer0_elapsed_flag(void);
/*****
unsigned int timer0_500ms_elapsed(void);  // Has an output, but no input
/*****

main()
{
    stop_watchdog_timer();

    enable_inputs_and_outputs();

    timer0_will_count_up_for_500ms();

    make_P10_red_LED_an_output();

    while(1)
    {
        if( timer0_500ms_elapsed() )
        {
            toggle_red_LED();
            clear_timer0_elapsed_flag();
        }

    } //end while(1)

} //end main()
```

```

/** Function Definitions ****
void stop_watchdog_timer(void)
{
    WDTCTL = DEVELOPMENT;    // Disables watchdog timer for development
}
/**
void enable_inputs_and_outputs(void)
{
    PM5CTL0 = ENABLE_PINS;    // Enables inputs and outputs
}
/**
void timer0_will_count_up_for_500ms(void)
{
    TA0CCR0 = 20000;          // ACLK will increment every 25us (0.000025)
    TA0CTL = UP | ACLK;       // 20000 * 25us = 0.5 seconds
}
/**
void make_P10_red_LED_an_output(void)
{
    P1DIR = RED_LED;          // Makes pin P1.0 an output
}
/**
void toggle_red_LED(void)
{
    P1OUT = P1OUT ^ RED_LED;  // Toggles the red LED on pin P1.0
}
/**
void clear_timer0_elapsed_flag(void)
{
    TA0CTL = TA0CTL & (~TAIFG); // Like we have seen before, this first looks
                                // at the value of TAIFG which we defined:
                                //     TAIFG = 0x0001 = 0000 0000 0000 0001

                                // Then, it bit-wise inverts the value
                                //     ~TAIFG = 0xFFFE = 1111 1111 1111 1110

                                // Then, it bit-wise ANDs the 0xFFFE value with
                                // the contents of TA0CTL. This clears the
                                // TAIFG bit (bit 0 of TA0CTL) without
                                // modifying any of the other bits
}
/**
unsigned int timer0_500ms_elapsed(void)
{
    return TA0CTL & TAIFG;    // This takes the bit-wise logic AND of
                                // the value we defined for TAIFG
                                //     TAIFG = 0x0001 = 0000 0000 0000 0001
                                // and the contents of the TA0CTL register

                                // The result will be returned as the output
                                // back to the main program
                                //     0x0000 If TAIFG is LO and the timer has
                                //             not yet counted up to TA0CCR0
                                //     0x0001 If TAIFG is HI and the timer has
                                //             counted up to TA0CCR0
}
/**

```


50. Create a new **CCS** project called **Timer_Up_Functions**. Copy the above program (including all of the function prototypes and function definitions) into your new **main.c**.

Save and **Build** your new project. If there are any errors, please go back and verify that you did not omit any of the lines.

Do not **Debug** the program yet. We will want to take a look at it.

51. Let us begin by looking at the function prototypes.

The first 6 function prototypes indicate that each function will have no (**void**) inputs and no output (also **void**). This is because the functions will simply be setting or clearing the necessary bits in the peripherals' registers to complete their appointed tasks.

The last function, **timer0_500ms_elapsed()**, again has no input (**void**), but it will have an output (type **unsigned int**) that it will return to the main program. The output will be used as the tested condition in an **if** statement.

Finally, it is worth a note of caution. Function prototypes DO have a semicolon at the end.

```
//*****  
/** Function Prototypes *****/  
//*****  
void stop_watchdog_timer(void);           // These functions do not have  
void enable_inputs_and_outputs(void);      // an input or an output  
void timer0_will_count_up_for_500ms(void);  
void make_P10_red_LED_an_output(void);  
void toggle_red_LED(void);  
void clear_timer0_elapsed_flag(void);  
//*****  
unsigned int timer0_500ms_elapsed(void);    // Has an output, but no input  
//*****
```

52. Next, let us look at the **main()** function itself.

The program consists of almost nothing but function calls. The first four will initialize the timer to count for 500 milliseconds (500ms or 0.5 seconds) and make pin P1.0 an output.

The program then goes into an infinite **while(1)** loop. Inside the loop, the program determines **if** the timer has finished counting to 500ms. If it has not, the program continues to check.

Once 500ms has elapsed, the functions will toggle the red LED and then clear the status of the timer. The program will then wait for another 500ms to elapse.

```
main()
{
    stop_watchdog_timer();

    enable_inputs_and_outputs();

    timer0_will_count_up_for_500ms();

    make_P10_red_LED_an_output();

    while(1)
    {
        if( timer0_500ms_elapsed() )
        {
            toggle_red_LED();
            clear_timer0_elapsed_flag();
        }

    } //end while(1)
} //end main()
```

53. Finally, we get to the function definitions. The first five are shown below.

Each of these functions should start to look a little more understandable. They have no input, and they do not have an output. Each, however, changes the values of the appropriate peripheral register to accomplish its appointed task.

A short comment has been added, but you may note that the comment really does not add much more information than the function name.

It is also worth mentioning that function definitions do NOT have a semicolon at the end of their top line.

```

/** Function Definitions ****
void stop_watchdog_timer(void)
{
    WDTCTL = DEVELOPMENT;    // Disables watchdog timer for development
}
/*****
void enable_inputs_and_outputs(void)
{
    PM5CTL0 = ENABLE_PINS;    // Enables inputs and outputs
}
/*****
void timer0_will_count_up_for_500ms(void)
{
    TA0CCR0 = 20000;          // ACLK will increment every 25us (0.000025)
    TA0CTL = UP | ACLK;       // 20000 * 25us = 0.5 seconds
}
/*****
void make_P10_red_LED_an_output(void)
{
    P1DIR = RED_LED;          // Makes pin P1.0 an output
}
/*****
void toggle_red_LED(void)
{
    P1OUT = P1OUT ^ RED_LED;   // Toggles the red LED on pin P1.0
}
/*****

```

54. Now, let us look at the next function definition.

Here, the function is simply clearing the **TAIFG** bit in the **TA0CTL** register. This is something we have seen several times before.

Unlike the previous function definitions, I have chosen to add a more detailed comment to this function. As I have mentioned in previous lab manuals, this particular operation has a way of confusing people. By spending a few moments writing out all the comments here, I will always have them in front of me if I come back to the program 6 months from now and I don't remember how it works.

In addition, if I want to reuse this function in another program, I have all of the details I will ever need right there in my code.

```
/**
 * *****
void clear_timer0_elapsed_flag(void)
{
    TA0CTL = TA0CTL & (~TAIFG); // Like we have seen before, this first looks
                                // at the value of TAIFG which we defined:
                                //      TAIFG = 0x0001 = 0000 0000 0000 0001

                                // Then, it bit-wise inverts the value
                                //      ~TAIFG = 0xFFFFE = 1111 1111 1111 1110

                                // Then, it bit-wise ANDs the 0xFFFFE value with
                                // the contents of TA0CTL. This clears the
                                // TAIFG bit (bit 0 of TA0CTL) without
                                // modifying any of the other bits
}
 * *****
 */
```

55. Now, we come to the last function definition. We saved it for last, because it is just a little different from the others – it has an output.

```
/** *****  
unsigned int timer0_500ms_elapsed(void)  
{  
    return TA0CTL & TAIFG;    // This takes the bit-wise logic AND of  
                               // the value we defined for TAIFG  
                               //    TAIFG = 0x0001 = 0000 0000 0000 0001  
                               // and the contents of the TA0CTL register  
  
                               // The result will be returned as the output  
                               // back to the main program  
                               //    0x0000 If the TAIFG bit in TA0CTL is LO and  
                               //          the timer has not yet counted up to  
                               //          the value in TA0CCR0  
                               //    0x0001 If the TAIFG bit in TA0CTL is HI and  
                               //          the timer has counted up to  
                               //          the value in TA0CCR0  
}  
/** *****
```

56. Previously, we have seen timer programs that included an **if** statement like this:

```
if(TA0CTL & TAIFG)
```


All we are doing in this program is replacing the (**TA0CTL & TAIFG**) condition with a function, **timer0_500ms_elapsed()**.

```
if( timer0_500ms_elapsed() )
```

Here, the function call is actually the test condition of the **if** statement.

57. Take a moment to notice the two sets of parentheses. The outer set of parentheses sets the start and end of the condition. The inner parentheses is indicating that the function does not have an input.

```
if( timer0_500ms_elapsed() )
```



Start of test condition

Function has no inputs

End of test condition

58. So, what does `timer0_500ms_elapsed()` actually do?

When the program reaches the `if` statement, it begins by looking at the test condition.

This test condition is the function name, so the program will “jump” to the function to get the test condition value.

The only instruction inside of the function is:

```
return TA0CTL & TAIFG;
```

Therefore, the function returns to the main program a value equal to `TA0CTL & TAIFG`. This return value then becomes the value tested by the `if` statement.

59. Ok, that was a lot of explanation for just a couple lines of code. We hope you feel at least a little comfortable with it so far. If not, take a look again, and then let us know if you still want a further explanation.

60. It is probably a good idea to **Save** and **Build** your program one more time. Sometimes, it is possible to accidentally modify your program after this lengthy of a discussion. It is better to find that out now than later....
61. When you are ready, click **Debug**.
62. From the **CCS Debugger**, click **Step Into** several times to walk through the first several functions.

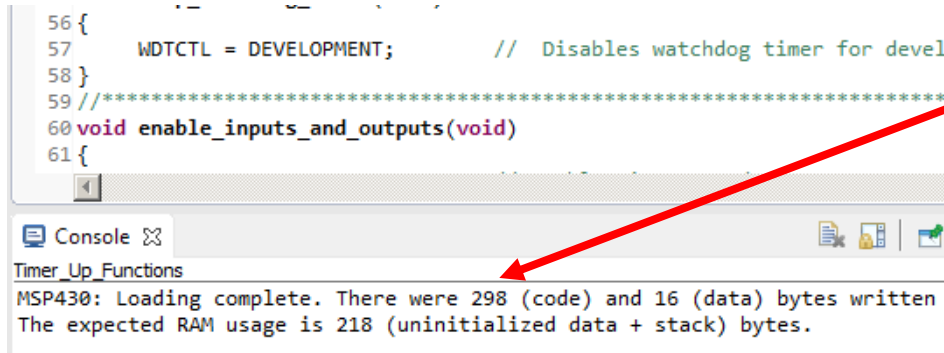
You will be able to watch as the program switches back-and-forth with the **main()** function and the functions we have defined.

Stop when you get to the **if** statement.

```
28 main()
29 {
30     stop_watchdog_timer();
31
32     enable_inputs_and_outputs();
33
34     timer0_will_count_up_for_500ms();
35
36     make_P10_red_LED_an_output();
37
38     while(1)
39     {
40         if( timer0_500ms_elapsed() )
41         {
42             toggle_red_LED();
43             clear_timer0_elapsed_flag();
44         }
45     } //end while(1)
46
47
48 } //end main()
49
```

63. From here, go ahead and click **Play** to run your program. The microcontroller takes over running at regular speed, and the LED starts to blink. We know, however, that the program is continuing to switch between **main()** and the functions we defined to continuously check the status of the timer, and to toggle the LED and reset the timer's **TAIFG** flag when it is appropriate.

64. While the program is running, take a look at the bottom-left corner of the **Debug** window for the Console pane. We see that this version of the program (in the example below) with all of our functions has used 298 bytes for the code (instructions).



```

56 {
57     WDTCTL = DEVELOPMENT;    // Disables watchdog timer for devel
58 }
59 //*****
60 void enable_inputs_and_outputs(void)
61 {

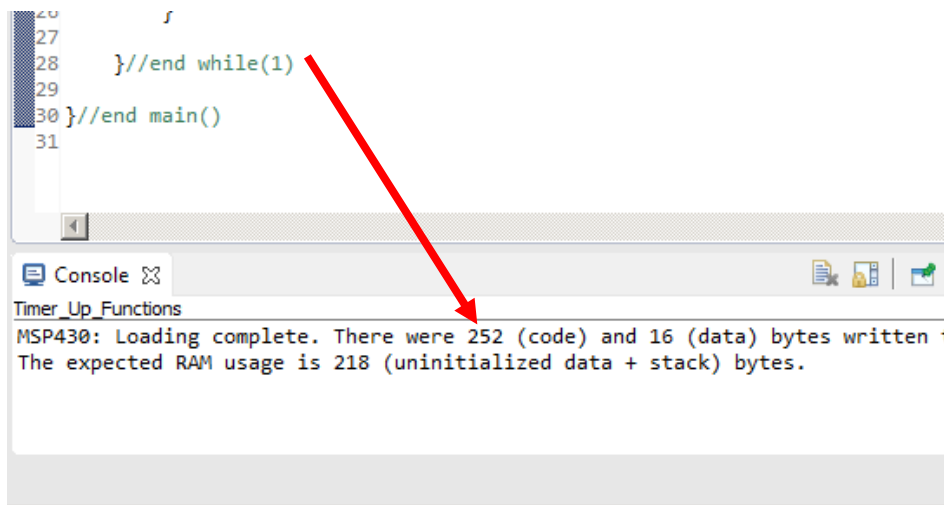
```

Console

Timer_Up_Functions

MSP430: Loading complete. There were 298 (code) and 16 (data) bytes written
The expected RAM usage is 218 (uninitialized data + stack) bytes.

If you were to **Build** the non-function version of the program from step 7 above, you would see that the version without functions is slightly smaller. It only uses 252 bytes.



```

27
28 } //end while(1)
29
30 } //end main()
31

```

Console

Timer_Up_Functions

MSP430: Loading complete. There were 252 (code) and 16 (data) bytes written
The expected RAM usage is 218 (uninitialized data + stack) bytes.

This is relatively common. Most of the time we use functions, our programs will be slightly larger. However, most developers are willing to suffer a small increase in program size for the convenience of using functions.

65. Now, let us have a function call another function. Previously, we defined one of our functions as follows:

```
void timer0_will_count_up_for_500ms(void)
{
    TA0CCR0 = 20000;           // ACLK will increment every 25us (0.000025)
    TA0CTL = UP | ACLK;       // 20000 * 25us = 0.5 seconds
}
```

66. To demonstrate how to have a function call another function, we are going to modify this to be:

```
void timer0_will_count_up_for_500ms(void)
{
    timer0_count_for_500ms();    // ACLK will increment every 25us (0.000025)
    timer0_in_up_mode();
}
```

67. To do this, we will need to add the two function prototypes at the beginning of your **main.c** file before the **main()** function.

```
void timer0_count_for_500ms(void);
void timer0_in_up_mode(void);
```

68. Finally, we will also have to add their function definitions at the end of the main() file.

```
void timer0_count_for_500ms(void)
{
    TA0CCR0 = 20000;           // 20000 * 25us = 0.5s
}

void timer0_in_up_mode(void)
{
    TA0CTL = UP | ACLK;       // Count up to 20000 with the ACLK
}
```

69. Go ahead and try to modify your previous program and see if you can get all the parts for the project to **Build** successfully.

If you run into trouble, the correct working program is shown on the next three pages with modifications highlighted.

```
#include <msp430.h>

#define RED_LED      0x0001      // P1.0 is the Red LED
#define DEVELOPMENT  0x5A80      // Stop the watchdog timer
#define ENABLE_PINS  0xFFFE      // Required to use inputs and outputs
#define ACLK         0x0100      // Timer_A ACLK source
#define UP           0x0010      // Timer_A UP mode
#define TAIFG        0x0001      // Used to look at Timer A Interrupt Flag

/***** Function Prototypes *****/
void stop_watchdog_timer(void);      // These functions do not have
void enable_inputs_and_outputs(void); // an input or an output
void timer0_will_count_up_for_500ms(void);
void make_P10_red_LED_an_output(void);
void toggle_red_LED(void);
void clear_timer0_elapsed_flag(void);
void timer0_count_for_500ms(void);    // Newly added prototype
void timer0_in_up_mode(void);         // Newly added prototype
/*****
unsigned int timer0_500ms_elapsed(void); // Has an output, but no input
*****/

main()
{
    stop_watchdog_timer();

    enable_inputs_and_outputs();

    timer0_will_count_up_for_500ms();

    make_P10_red_LED_an_output();

    while(1)
    {
        if( timer0_500ms_elapsed() )
        {
            toggle_red_LED();
            clear_timer0_elapsed_flag();
        }

    } //end while(1)
} //end main()
```

```

/** Function Definitions ****
void stop_watchdog_timer(void)
{
    WDTCTL = DEVELOPMENT;    // Disables watchdog timer for development
}
/*****
void enable_inputs_and_outputs(void)
{
    PM5CTL0 = ENABLE_PINS;    // Enables inputs and outputs
}
/*****
void make_P10_red_LED_an_output(void)
{
    P1DIR = RED_LED;          // Makes pin P1.0 an output
}
/*****
void toggle_red_LED(void)
{
    P1OUT = P1OUT ^ RED_LED;   // Toggles the red LED on pin P1.0
}
/*****
void clear_timer0_elapsed_flag(void)
{
    TA0CTL = TA0CTL & (~TAIFG); // Like we have seen before, this first looks
                                // at the value of TAIFG which we defined:
                                //     TAIFG = 0x0001 = 0000 0000 0000 0001

                                // Then, it bit-wise inverts the value
                                //     ~TAIFG = 0xFFFE = 1111 1111 1111 1110

                                // Then, it bit-wise ANDs the 0xFFFE value with
                                // the contents of TA0CTL. This clears the
                                // TAIFG bit (bit 0 of TA0CTL) without
                                // modifying any of the other bits
}
/*****
unsigned int timer0_500ms_elapsed(void)
{
    return TA0CTL & TAIFG;      // This takes the bit-wise logic AND of
                                // the value we defined for TAIFG
                                //     TAIFG = 0x0001 = 0000 0000 0000 0001
                                // and the contents of the TA0CTL register

                                // The result will be returned as the output
                                // back to the main program
                                //     0x0000 If TAIFG is LO and the timer has
                                //         not yet counted up to TA0CCR0
                                //     0x0001 If TAIFG is HI and the timer has
                                //         counted up to TA0CCR0
}
/*****

```

```
void timer0_will_count_up_for_500ms(void)
{
    timer0_count_for_500ms();    // Timer0 will count for 0.5seconds
    timer0_in_up_mode();        // in up mode
}
//*****
void timer0_count_for_500ms(void)
{
    TA0CCR0 = 20000;            // 20000 * 25us = 0.5s
}
//*****
void timer0_in_up_mode(void)
{
    TA0CTL = UP | ACLK;        // Count up to value in TA0CCR0 with the ACLK
}
//*****
```

70. There is no limit to how many functions you can have call other functions. They really allow you to develop your program as you desire.

A final note of caution, however. As we saw previously, using functions, we saw how the original program grew from 252 bytes of program memory to 298 bytes of program memory. When I added the modifications for the program shown in the previous step, the code grew to 310 bytes of program memory.

71. Next, let us look at an example of a function that has multiple inputs and has multiple output possibilities.

Below is a function called **max_of**. It has two inputs (of type **signed int**) and one output (also of type **signed int**).

After initializing the variables **first** (-4) and **second** (13219), the program immediately will call the **max_of** function.

The function only consists of three statements: an **if** statement and two different **return** statements. Your function may have as many different **return** statements as you want, but as soon as the program executes one of the **return** statements, the program will immediately leave your function and return to **main()** (or from wherever your function was called).

```
//*****
/** Function Prototype *****/
//*****
signed int max_of (signed int, signed int);
//*****

main()
{
    signed int first, second, output;

    first = -4;
    second = 13219;

    output = max_of(first,second);

    while(1);
}

//*****
/** Function Definition *****/
//*****
signed int max_of(signed int a, signed int b)
{
    if (a > b)                // If a>b,
    {
        return a;           // then return a, and immediately
    }                       // leave the max_of function

    return b;               // You only get here if a>b if false,
                           // so return b and leave max_of function
}
//*****
```

72. Create a new **CCS Project** called **Max_Of** and copy and paste the above program into your new **main.c** file.

Make sure you turn off the **Optimization** in the **Project Properties**.

Save, Build, and Debug the project.

73. In the **Debugger**, click Step Into to watch your microcontroller go line-by-line through your code.

74. When you are ready, click **Terminate** to return to the **CCS Editor**.

75. In the **CCS Editor**, change the value of **first** to be **22000** (or at least larger than **second**).

76. **Save, Build, and Debug** your project.

77. Click **Step Into** again to watch your program execute again. Notice how after you reach the first **return** statement, the program leaves the function and returns to **main()**.

78. Click **Terminate** when you are ready to return to the **CCS Editor**.

79. There is one more thing that we want to show you about functions.

Please be careful as you walk through these last steps, because they do cause some confusion in developers and this has been a rather long lab manual.

Below, we have modified the **Max_Of** program to add two additional variables (also called **a** and **b**) to the **main()** function. Now we have variables called **a** and **b** in both the **main()** function and also the **max_of** function.

```
//*****
/** Function Prototype *****
//*****
signed int max_of (signed int, signed int);
//*****

main()
{
    signed int a, b, first, second, output;

    a = 5;
    b = -2;

    first = -4;
    second = 13219;

    output = max_of(first,second);

    while(1);
}

//*****
/** Function Definition *****
//*****
signed int max_of(signed int a, signed int b)
{
    if (a > b)                // If a>b,
    {
        return a;            //      then return a, and immediately
    }                        //      leave the max_of function

    return b;                // You only get here if a>b if false,
    // so return b and leave max_of function
}
//*****
```


80. While having multiple variables by the same name might initially seem confusing, the C programming language can take care of this easily **IF THE VARIABLES HAVE DIFFERENT SCOPES**.

In the modified **max_of** program, we now have two variables called **a** and two variables called **b**.

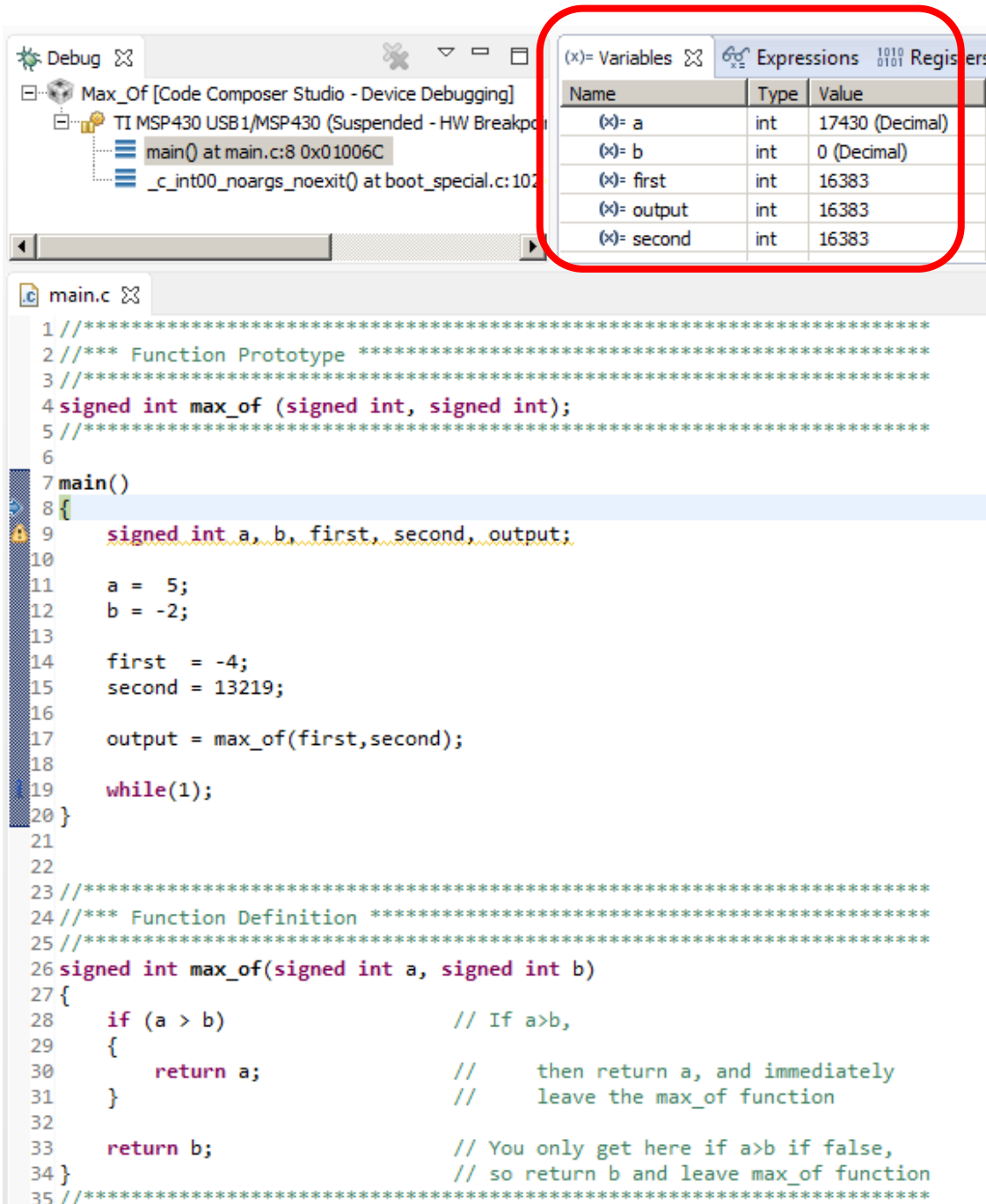
One of the **a** variables and one of the **b** variables is local to the **main()** function and can only be seen in the **main()** function.

One of the **a** variables and one of the **b** variables is local to the **max_of()** function and can only be seen in the **max_of()** function.

81. Sound confusing? Let us try it out. Copy the modified **max_of** program into your **CCS** project's **main.c** file.

Save, **Build**, and **Debug** your project.

82. In the **Debugger**, you should see **a**, **b**, **first**, **second**, and **output** in the **Variables** pane. Remember, we have not started the program yet to assign their values, so the values you see may be different than the example below.



The screenshot shows the Code Composer Studio Debugger interface. The **Variables** pane is highlighted with a red rectangle, displaying the following data:

Name	Type	Value
(*)= a	int	17430 (Decimal)
(*)= b	int	0 (Decimal)
(*)= first	int	16383
(*)= output	int	16383
(*)= second	int	16383

The **main.c** source code is visible below the debugger, showing the following code:

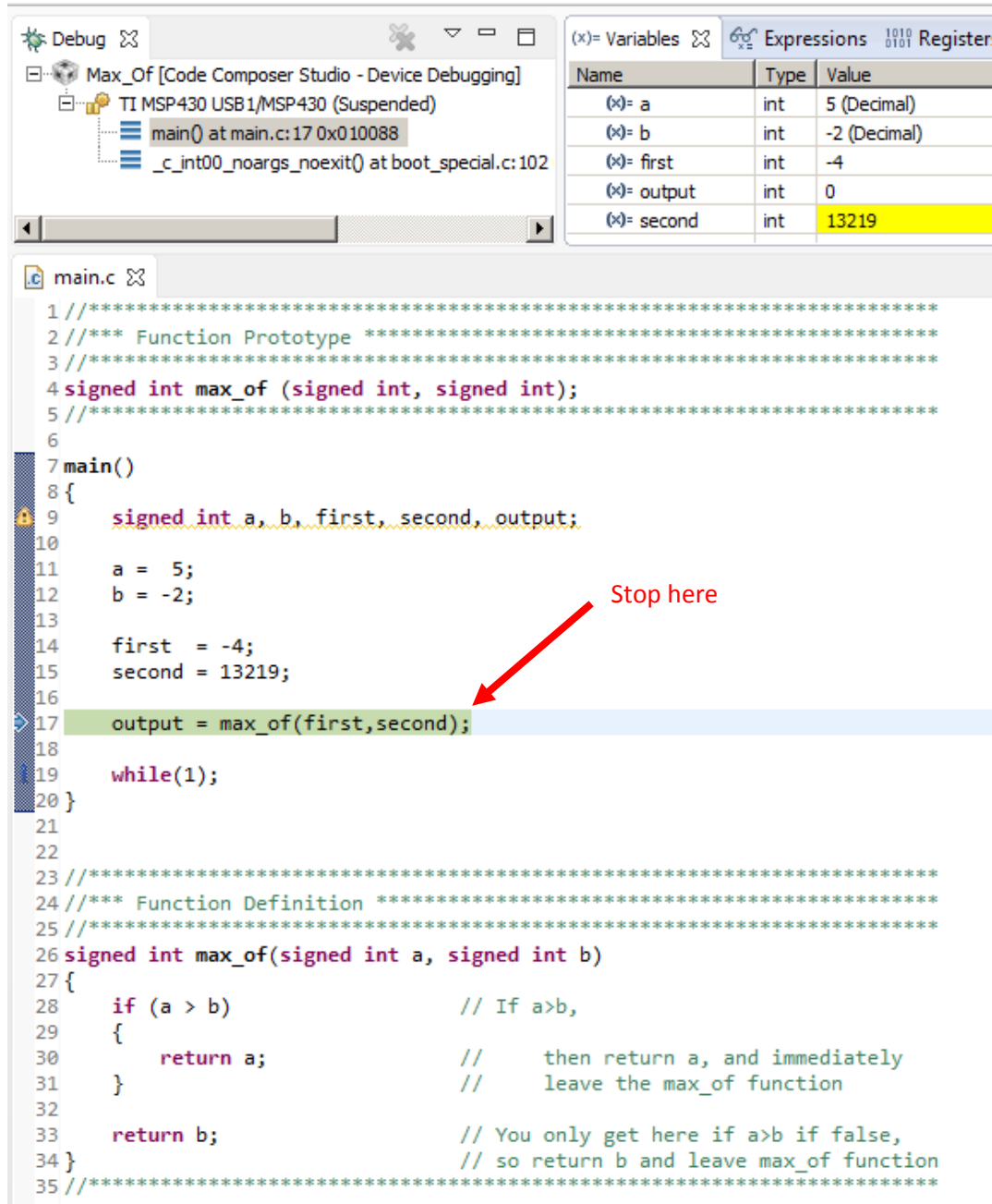
```

1 //*****
2 /** Function Prototype *****
3 //*****
4 signed int max_of (signed int, signed int);
5 //*****
6
7 main()
8 {
9     signed int a, b, first, second, output;
10
11     a = 5;
12     b = -2;
13
14     first = -4;
15     second = 13219;
16
17     output = max_of(first, second);
18
19     while(1);
20 }
21
22
23 //*****
24 /** Function Definition *****
25 //*****
26 signed int max_of(signed int a, signed int b)
27 {
28     if (a > b)           // If a>b,
29     {
30         return a;       // then return a, and immediately
31     }                   // leave the max_of function
32
33     return b;           // You only get here if a>b if false,
34 }                       // so return b and leave max_of function
35 //*****

```

83. Click **Step Into** until you reach (have not yet performed) the function call.

As we should expect, since we are still performing instructions in the **main()** function, the local values of the variables **a** and **b** are 5 and -2, respectively.



Debug [Code Composer Studio - Device Debugging]

TI MSP430 USB1/MSP430 (Suspended)

main() at main.c:17 0x010088

_c_int00_noargs_noexit() at boot_special.c:102

Name	Type	Value
(x)= a	int	5 (Decimal)
(x)= b	int	-2 (Decimal)
(x)= first	int	-4
(x)= output	int	0
(x)= second	int	13219

main.c

```

1 //*****
2 /** Function Prototype *****
3 //*****
4 signed int max_of (signed int, signed int);
5 //*****
6
7 main()
8 {
9     signed int a, b, first, second, output;
10
11     a = 5;
12     b = -2;
13
14     first = -4;
15     second = 13219;
16
17     output = max_of(first, second);
18
19     while(1);
20 }
21
22
23 //*****
24 /** Function Definition *****
25 //*****
26 signed int max_of(signed int a, signed int b)
27 {
28     if (a > b)                // If a>b,
29     {
30         return a;            // then return a, and immediately
31     }                        // leave the max_of function
32
33     return b;                // You only get here if a>b if false,
34                             // so return b and leave max_of function
35 //*****

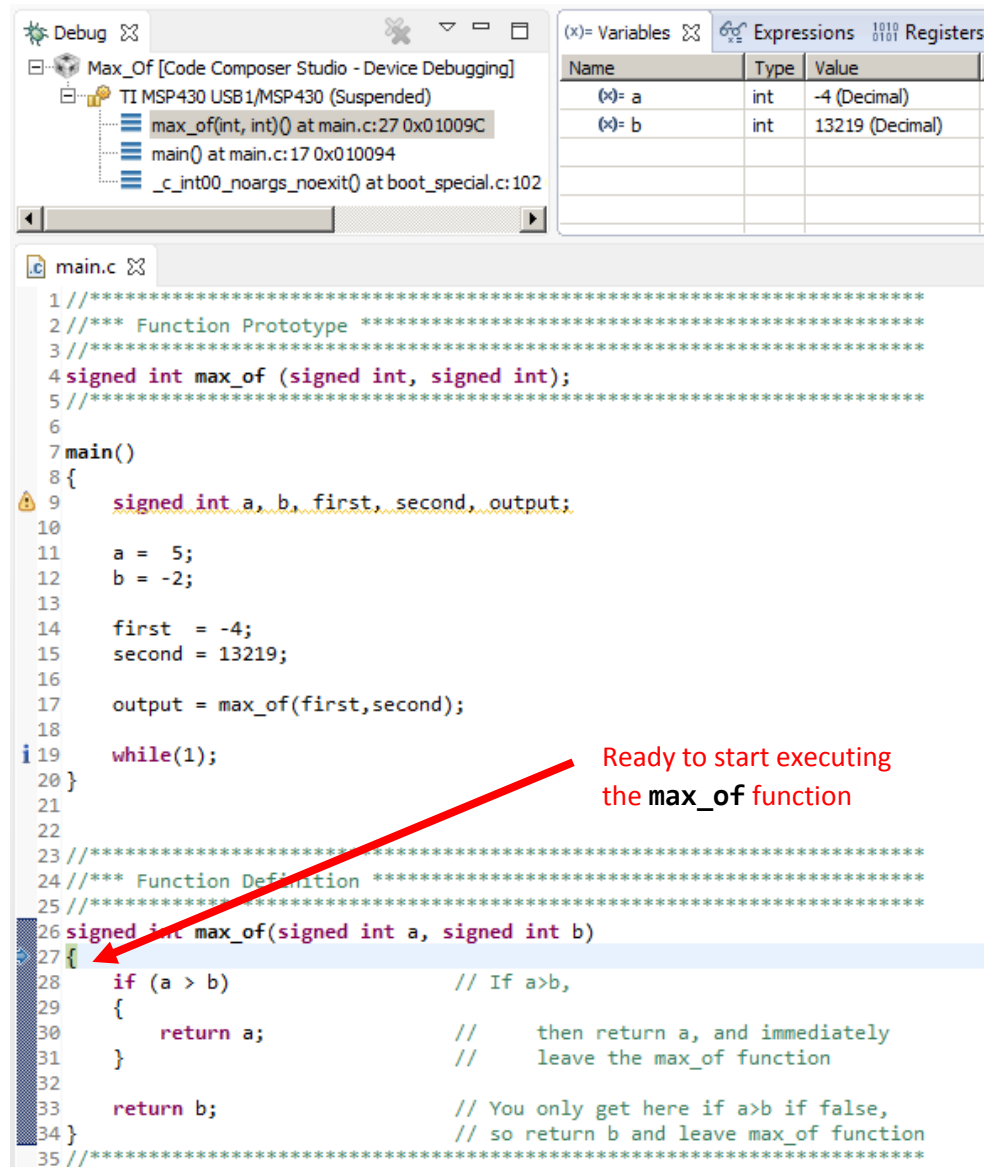
```

Stop here

84. Click **Step Into** and your program will jump to the **max_of** function.

Now, the variables shown in the **Variables** pane are different. **first**, **second**, and **output** are no longer shown because they are local to **main()**.

In addition, there are variables, **a** and **b**, now shown with new values that are local to **max_of**.



The screenshot shows the Code Composer Studio interface. The top pane displays the project structure with 'Max_Of [Code Composer Studio - Device Debugging]' selected. The 'Variables' pane on the right shows the current state of variables: 'a' is an integer with value -4 (Decimal), and 'b' is an integer with value 13219 (Decimal). The bottom pane shows the source code for 'main.c'. The code includes a function prototype for 'max_of' and its definition. A red arrow points to the start of the 'max_of' function definition, indicating the current execution point. The code is as follows:

```

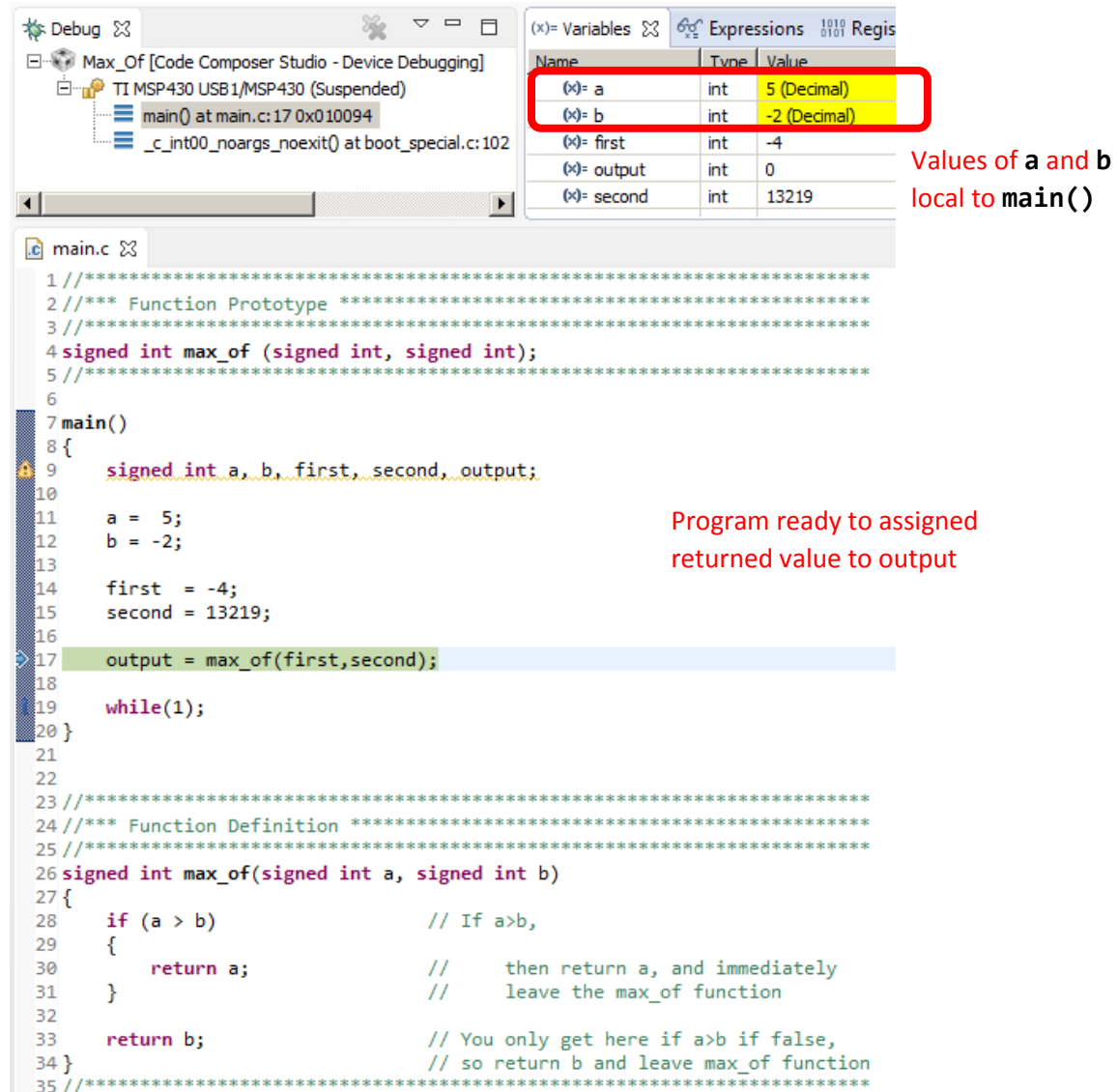
1 //*****
2 /*** Function Prototype *****/
3 //*****
4 signed int max_of (signed int, signed int);
5 //*****
6
7 main()
8 {
9     signed int a, b, first, second, output;
10
11     a = 5;
12     b = -2;
13
14     first = -4;
15     second = 13219;
16
17     output = max_of(first,second);
18
19     while(1);
20 }
21
22
23 //*****
24 /*** Function Definition *****/
25 //*****
26 signed int max_of(signed int a, signed int b)
27 {
28     if (a > b)                // If a>b,
29     {
30         return a;            // then return a, and immediately
31     }                        // leave the max_of function
32
33     return b;                // You only get here if a>b if false,
34                             // so return b and leave max_of function
35 }
36 //*****

```

85. If you continue to click **Step Into**, the program identifies that **13219** is larger than **-4** and the execution returns to the **main()** function.

Note, as soon as the program returns to **main()**, the local values of **a** and **b** are updated. The original values were never discarded or erased – they just are not available in **max_of** or anywhere else outside of their scope.

Also, since the program has just returned to **main()**, you will notice that the value **returned** from **max_of** has not yet been assigned to **output**.



Name	Type	Value
(x)= a	int	5 (Decimal)
(x)= b	int	-2 (Decimal)
(x)= first	int	-4
(x)= output	int	0
(x)= second	int	13219

Values of **a** and **b** local to **main()**

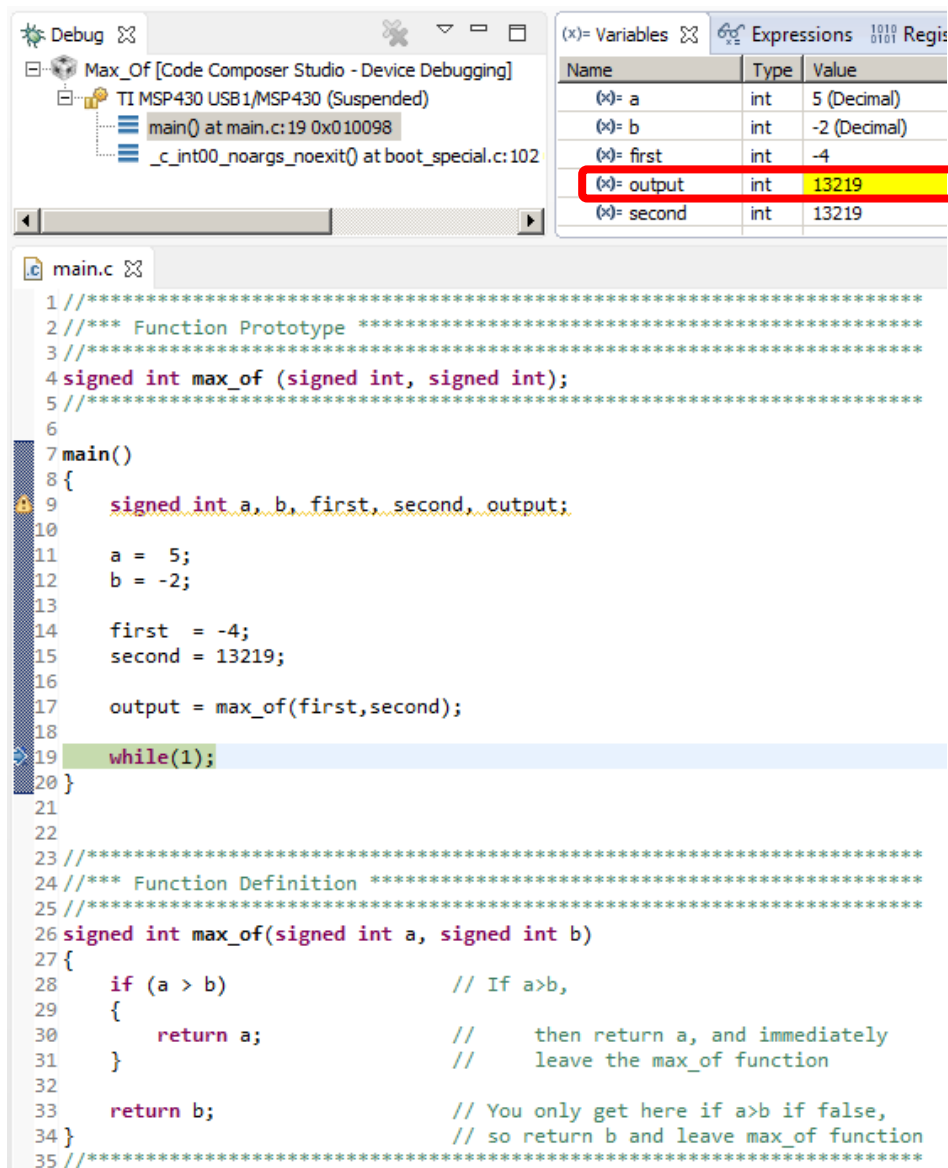
```

1 //*****
2 /*** Function Prototype *****/
3 /***
4 signed int max_of (signed int, signed int);
5 /***
6
7 main()
8 {
9     signed int a, b, first, second, output;
10
11     a = 5;
12     b = -2;
13
14     first = -4;
15     second = 13219;
16
17     output = max_of(first, second);
18
19     while(1);
20 }
21
22
23 /***
24 /*** Function Definition *****/
25 /***
26 signed int max_of(signed int a, signed int b)
27 {
28     if (a > b)                // If a>b,
29     {
30         return a;            // then return a, and immediately
31     }                        // leave the max_of function
32
33     return b;                // You only get here if a>b if false,
34                             // so return b and leave max_of function
35 /***

```

Program ready to assigned
returned value to output

86. Finally, you can click **Step Into** and the **returned** value is assigned to **output**. This is reflected in the **Variables** pane.



The screenshot shows the Code Composer Studio interface. The top pane displays the **Variables** window with the following data:

Name	Type	Value
(x)= a	int	5 (Decimal)
(x)= b	int	-2 (Decimal)
(x)= first	int	-4
(x)= output	int	13219
(x)= second	int	13219

The bottom pane shows the source code of **main.c**:

```

1 //*****
2 /*** Function Prototype *****/
3 //*****
4 signed int max_of (signed int, signed int);
5 //*****
6
7 main()
8 {
9     signed int a, b, first, second, output;
10
11     a = 5;
12     b = -2;
13
14     first = -4;
15     second = 13219;
16
17     output = max_of(first,second);
18
19     while(1);
20 }
21
22
23 //*****
24 /*** Function Definition *****/
25 //*****
26 signed int max_of(signed int a, signed int b)
27 {
28     if (a > b)           // If a>b,
29     {
30         return a;       // then return a, and immediately
31     }                   // leave the max_of function
32
33     return b;           // You only get here if a>b if false,
34 }                       // so return b and leave max_of function
35 //*****

```

87. Whew. That was a lot, and we have only just begun to scratch the surface of the function in the C programming language.

We will continue to use functions as we move forward throughout the course, so you will get lots of practice using them.

Congratulations on finishing this lab manual! :)

All tutorials and software examples included herewith are intended solely for educational purposes. The material is provided in an “as is” condition. Any express or implied warranties, including, but not limited to the implied warranties of merchantability and fitness for particular purposes are disclaimed.

The software examples are self-contained low-level programs that typically demonstrate a single peripheral function or device feature in a highly concise manner. Therefore, the code may rely on the device's power-on default register values and settings such as the clock configuration and care must be taken when combining code from several examples to avoid potential side effects. Additionally, the tutorials and software examples should not be considered for use in life support devices or systems or mission critical devices or systems.

In no event shall the owner or contributors to the tutorials and software be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.