

CPE 325: Intro to Embedded Computer System

Lab01

First Lab Assignment

Submitted by: Michael Agnew

Date of Experiment: 1/10/2024

Report Deadline: 1/17/2024

Demonstration Deadline: 1/17/2024

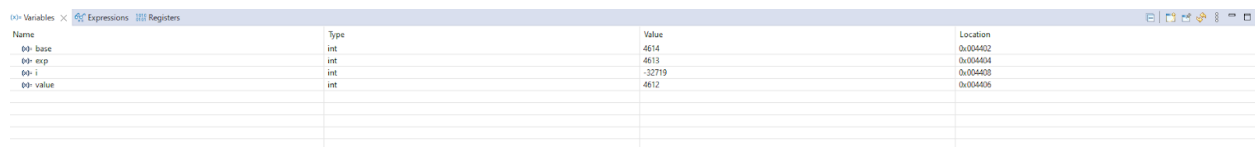
Introduction:

The aim of this lab was to learn how to use the Code Composer Studio software and use it to implement C code with embedded systems. There was a tutorial that was given to the students that participated in this lab; and with said tutorial, the students were able to fully replicate the steps that were laid out for them to complete. The tutorial for this lab talked extensively about how to set up Code Composer Studio (CCS) and configure it to the correct settings that would allow it to be properly implemented over the course of the semester for the purposes of this lab.

Theory:

There is a lot to learn about when it comes to CCS. There are a great many different features and tools that can be used for one to better understand code and how it acts in correlation to the rest of the system and memory. With this being said, the tutorial that was used for this first lab goes into great detail about the various tools that one will be using throughout the semester.

One of the more important tools that is able to be used in CCS is the memory and variable window.



The screenshot shows the 'Expressions' window in Code Composer Studio. It contains a table with four columns: Name, Type, Value, and Location. The table lists four variables: 'fp-base' (int, 4614, 0x004402), 'fp-exp' (int, 4613, 0x004404), 'fp-l' (int, -32719, 0x004408), and 'fp-value' (int, 4612, 0x004406). There are also several empty rows below the listed variables.

Name	Type	Value	Location
fp-base	int	4614	0x004402
fp-exp	int	4613	0x004404
fp-l	int	-32719	0x004408
fp-value	int	4612	0x004406

Figure 1

This allows the student to view the list of variables, their names, their type, their value, and their location in memory. This is very useful for understanding the given code and how the variables are used and is also very useful for debugging.

The next feature that is useful that was gone over by the tutorial is the console. This is where the output for various code is displayed when using print statements. The format of this tool is shown as such:

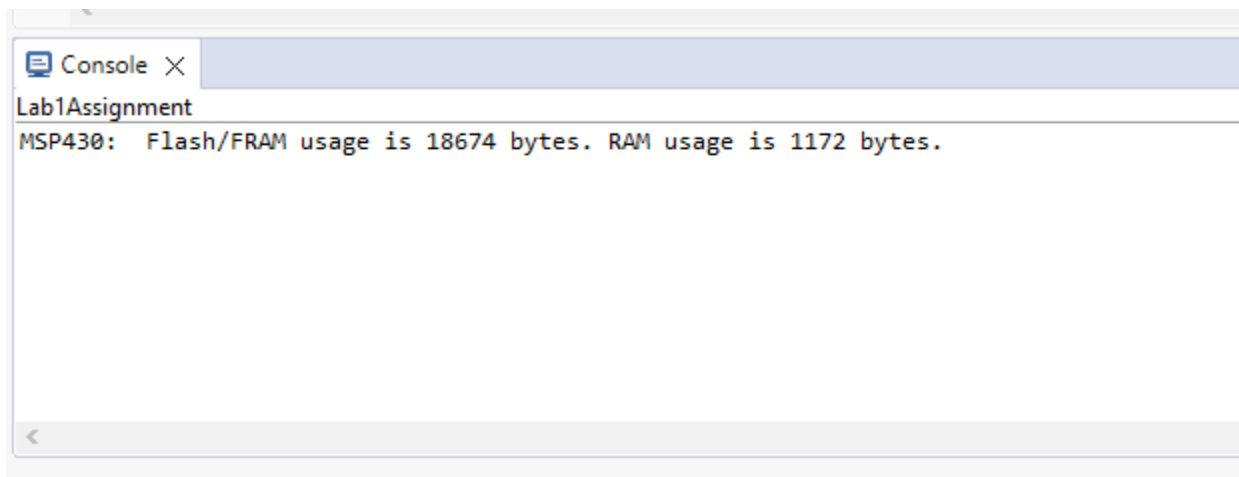


Figure 2

Another one of the extremely useful tools that is available in CCS is the debugging tool which allows one to set breakpoints, “run”, “step into”, “step over”, and “terminate”. The “run” tool allows one to load up a project and run it as well as debug it as shown:

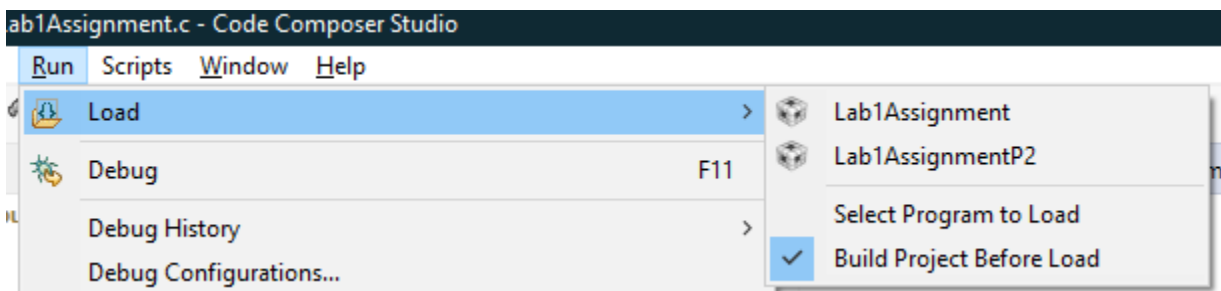
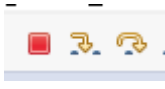


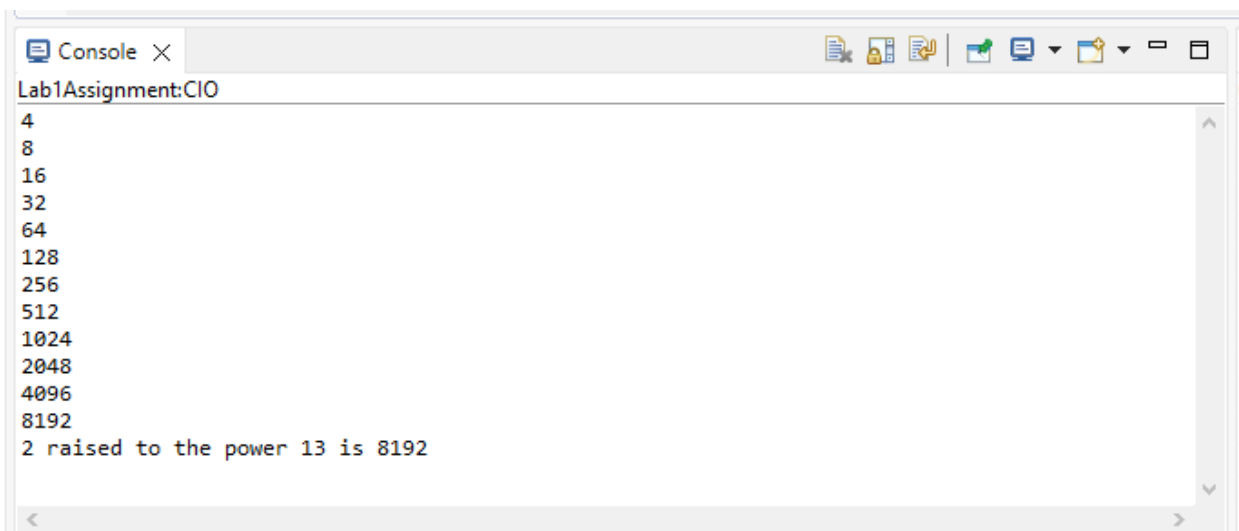
Figure 3

Breakpoints are available in debugging which lets one set points in the code where execution will stop until told to move on. “Step into” allows one to move into a function and check how it works and how it is being used. “Step over” allows one to move to the next line of the code and “terminate” allows the user to end the execution of the program at any step of which it currently

sits. These commands are shown here in the debug menu: . Figure 4

Program 01:

The program that was assigned to be made for this lab was a code that calculates a given number squared by a given exponent. It was specifically instructed for this code to not be recursive and so a for loop was used to complete it. The output of the code that was written is:



```
Lab1Assignment:CIO
4
8
16
32
64
128
256
512
1024
2048
4096
8192
2 raised to the power 13 is 8192
```

Figure 5

This output shows the value of the variable at each iteration of the loop and then prints the final result at the end. In the assignment, it asks the question of how many clock cycles it takes for this program to complete as opposed to the recursive version that was provided. What was found was

quite interesting; depending on what was used for executing the code, the clock speed was different. When entering into the debug menu and hitting “resume” in order to finish the execution of the code, the clock speed for both the for loop implementation and the recursive

implementation sat at 4,438 cycles.

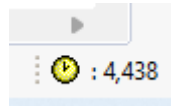


Figure 6.

Whereas, when entering into the debug menu and manually stepping through each of the steps using the “step over” tool, the clock speed for the for loop implementation was 59,305 as opposed to the recursive implementation which was 18,460 respectively.

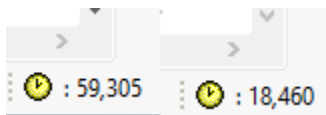


Figure 7.

This raises the question of whether or not CCS is possibly doing some sort of simplification or approximation when it comes to executing this code and that is why the clock speed is much lower when skipping to the end of the code as opposed to stepping through it one line by a time.

The last part of this section is meant to house the program flowchart for the first code that was written. This flowchart is show as such:

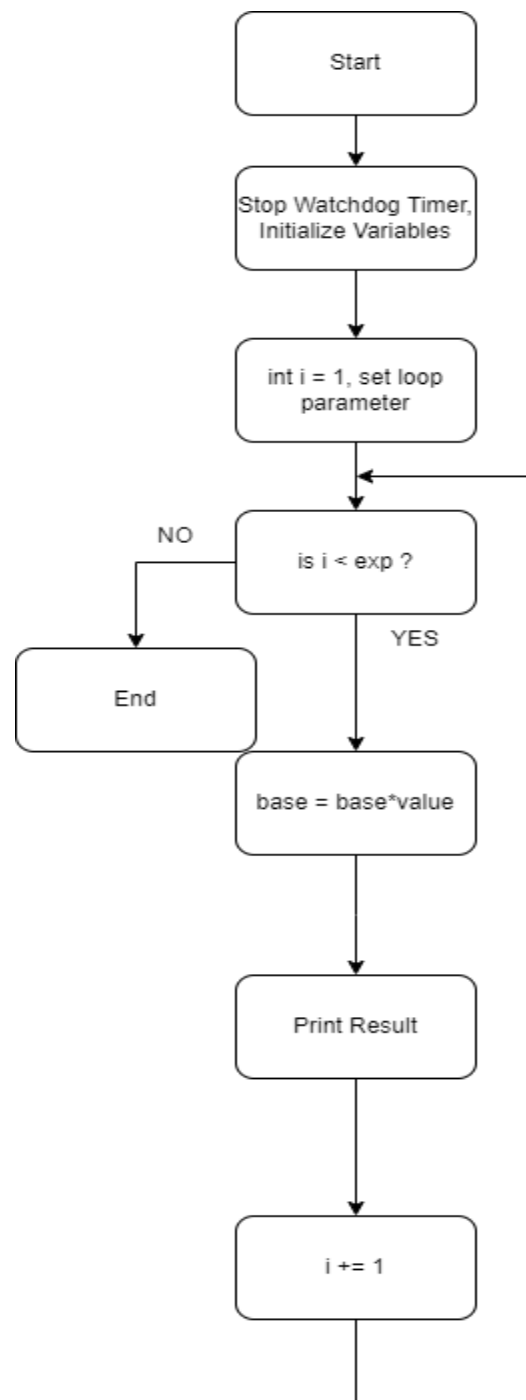


Figure 8

Program 02:

The second program that was configured for this lab was one that was previously written and was meant to be fixed by the student before being resubmitted. The program contained various errors which prevented it from working properly and so it was fixed in order to output the correct answer. These changes were documented throughout the code and a clock speed for the program was also captured.



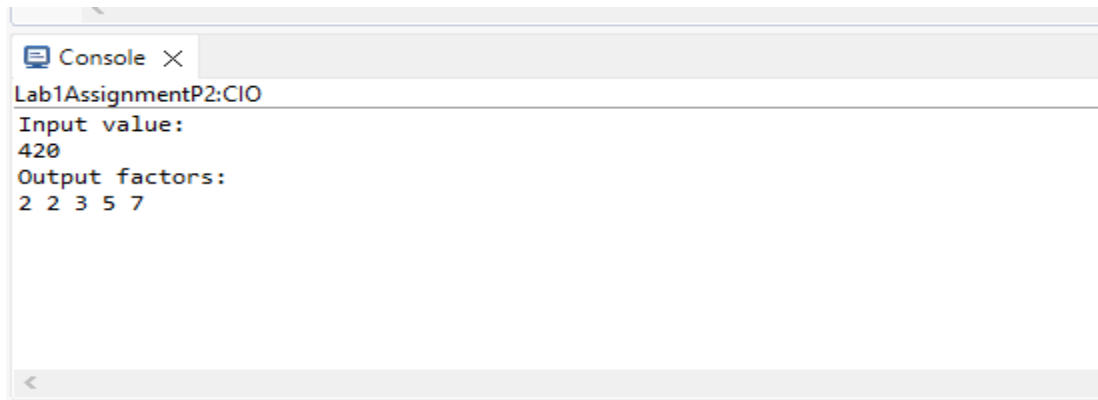
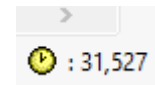
Figure 9

Figure 7 illustrates the correct output that was found for the fixed code that was given. The

number of clock cycles required for the completion of this code was  : 27,868 . Figure 8.

The input value can also affect the number of clock cycles required as the amount for a number

of size 84 was done in 27,868 while a number such as 420 was done in 31,527.



Figure(s) 10, 11

This would make sense because the larger a number is, the more factors it has which ultimately means that the code would take longer to run. This is supported by using a smaller number which takes an even lower amount of clock cycles to complete.



Figure 12

The amount of clock cycles it took to complete this program was 21,673

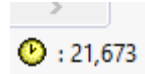


Figure 12.

This program also required a flowchart to be made which is here:

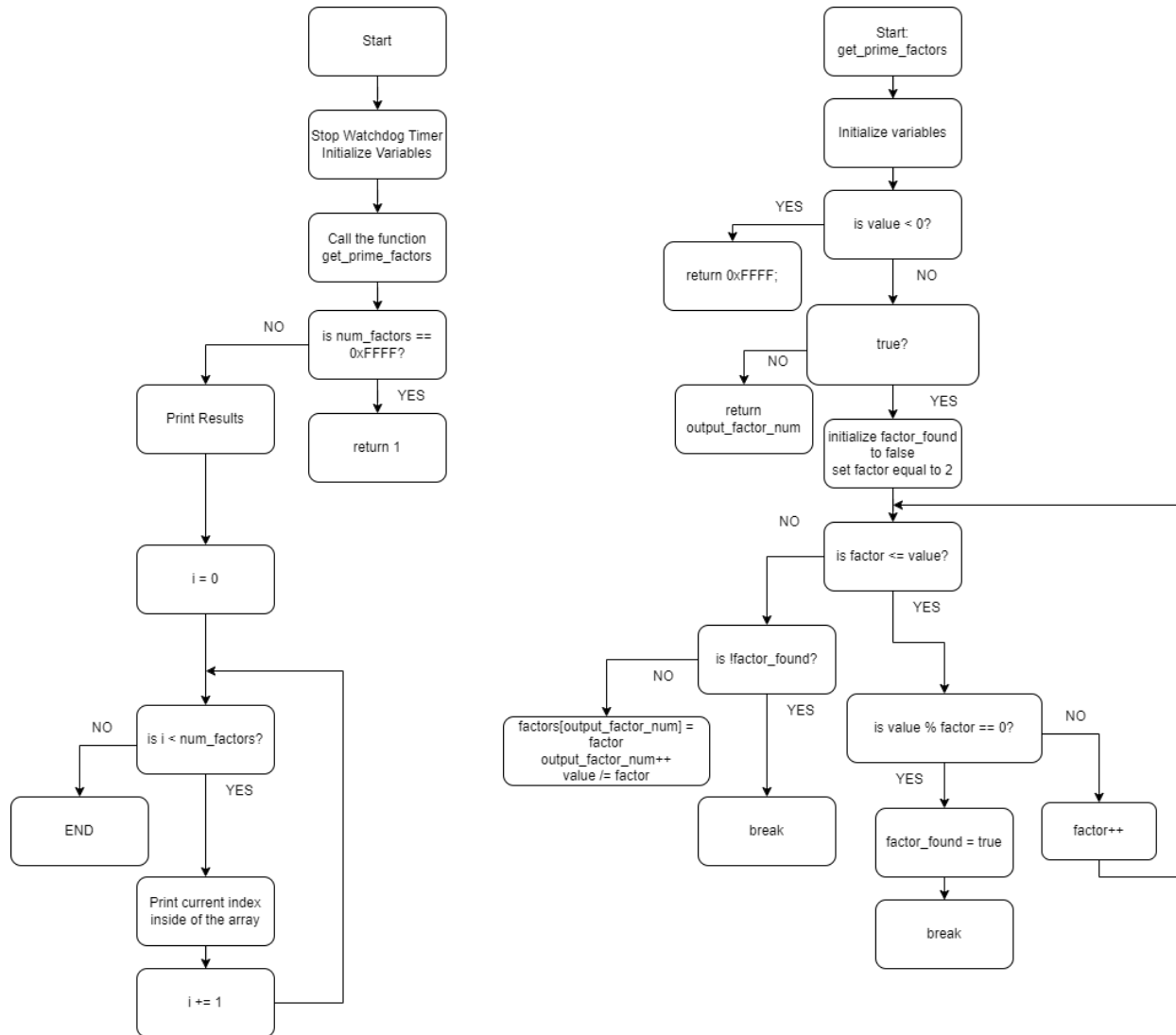


Figure 13

Conclusion:

The conclusion that can be made is that those who closely followed the tutorial for this lab are now prepared to use CCS for the coming semester and for any projects that are assigned in the future. There are many tools that can be used to help one understand the code and perhaps the issues that reside in it that can be fixed. The students now also understand the clock cycles and how some codes can take longer than others and how the type of code or perhaps the input influences this. This can help students optimize their code in the future which is beneficial for employers. There were no issues to report over the course of the lab. The only issues run into were growing pains of using new software and trying to understand how all of it works. However, these have been alleviated.

Appendix:

Source Code Project 01:

```
/*-----  
  
* File:    Lab1Assignment.c  
  
* Function: This C code will calculate the power of a given base.  
  
* Description: This program calculates the power of a given  
*             base using a for loop  
  
* Input:    None  
  
* Output:   Base raised to the given power.  
  
* Author(s): Michael Agnew, ma0133@uah.edu  
  
* Date:     Jan 17, 2024  
  
*-----*/
```

```
#include <stdio.h>
```

```
#include <msp430.h>
```

```
int main(void)
```

```
{
```

```
    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
```

```

int base = 2; // initialize base

int exp = 13; // initialize exp

int value = base; // set value var to store value of base

int i = 1; // initialize loop parameter

for (i = 1; i < exp; i++) // loop while the loop parameter is less than the value of exp
{
    base = base*value; // multiply the current base by the given value

    printf("%d", base); // print the result

    printf("\n"); // extra space for format
}

printf("%d raised to the power ", value); // printing the results

printf("%d is ", exp);

printf("%d", base);

printf("\n");

return 0;

}

```

Source Code Project 02:

/*-----

* File: Lab1AssignmentP2.c

* Description: This program finds the prime factorization of a hardcoded value

*

* Board: 5529

* Input: Hardcoded short int number

* Output: Factors of input value printed to console (ordered low to high)

* Author: Douglas Marr, Fixed and submitted by Michael Agnew

* Date: 1/17/2024

-----/

```
#include <stdio.h>
```

```
#include <stdbool.h> // Added to use boolean data type
```

```
#include <msp430.h>
```

```
#define true 1
```

```
#define false 0
```

```
// This function finds all of the factors in `value`
```

```
// Factors of `value` are output as elements of `factors`
```

```
// Function return value is the number of factors found (0xFFFF for error)
```

```
int get_prime_factors(int *factors, int value)
```

```
{ // start of getprimefactors
```

```
    int output_factor_num = 0;
```

```

int factor;

//char factor_found;

if (value < 0)
{
    return 0xFFFF;
}

// Loop while remaining value is prime
while (true) // changed from !true to true
{ // start of while

    bool factor_found = false; // added "bool" to define this variable

    factor = 2;

    // Get the lowest remaining factor of `value`
    while (factor <= value) // changed to (factor <= value)
    { // start of while

        if (value % factor == 0)

        { // Is `factor` a factor of `value`?

            // Factor found

            factor_found = true;

            break;

        } // end of if

```

```

        // Factor not found

        factor++;

    } // end of while

    if (!factor_found) // changed to !factor_found from factor_found
    {
        break;
    }

    // Add factor to array, and divide it out of the working value

    factors[output_factor_num] = factor; // added a semi-colon

    output_factor_num++;

    value /= factor;

} // end of while

return output_factor_num;

} // end of getprimefactors

int main(void)

{

    // Stop watchdog timer

    WDTCTL = WDTPW + WDTHOLD; // added stop watchdog timer

    // Input value for factorization

    const int INPUT_VALUE = 10;

    // Output array

```

```

// Array must have at least as many elements as factors of INPUT_VALUE

int prime_factors[10] = {};

int num_factors;

int i;

// Calculate prime factors, and check for function error

num_factors = get_prime_factors(prime_factors, INPUT_VALUE);

if (num_factors == 0xFFFF)

{

    return 1;

}

// Print input value & output prime factors separated by spaces

printf("Input value: \n%d\n", INPUT_VALUE);

printf("Output factors: \n");

for (i = 0; i < num_factors; i++)

{

    printf("%d ", prime_factors[i]);

}

printf("\n");

fflush(stdout);

return 0;} // added bracket

```