CPE 325: Embedded Systems Laboratory
Laboratory #1 Tutorial
Introduction to TI Code Composer Studio (IDE)

Aleksandar Milenković
Email: milenka@uah.edu
Web: http://www.ece.uah.edu/~milenka

Objective:

This tutorial will help you get started with the TI's Code Composer Studio for MSP430. It includes the following topics:

Creating an application project
Debugging

Notes:

The latest version of Code Composer Studio can be downloaded for free from the TI's web site: http://www.ti.com/tool/ccstudio.

Contents:

1 Creating a Project

This section introduces you to TI's Code Composer Studio (CCStudio) Integrated Development Environment (IDE) that will be used for software development in the Embedded Systems Laboratory. In the form of a step-by-step tutorial, it demonstrates a typical development cycle and shows you how to use the CCStudio compiler and the CCStudio linker to create a small application for the MSP430 microcontroller. It includes topics such as creating a workspace, setting up a project with C source files, compiling and linking your application, and debugging.

1.1 Creating a New Workspace and a Project

Using the Code Composer Studio IDE, you can design advanced projects. For more information, read the Code Composer Studio wiki at [http://processors.wiki.ti.com/index.php/Category:CCS.](http://processors.wiki.ti.com/index.php/Category:CCS) Here, we will create a relatively simple project with several source files.

Step 1. Open Code Composer Studio (CCS).

Step 2. In the Eclipse Launcher select a directory to be used as the workspace (see Figure 1.) and press <Launch>. A *Getting Started* window will appear (Figure 2). This page provides links to example projects as well as documentation related to Code Composer and TI Microcontrollers.

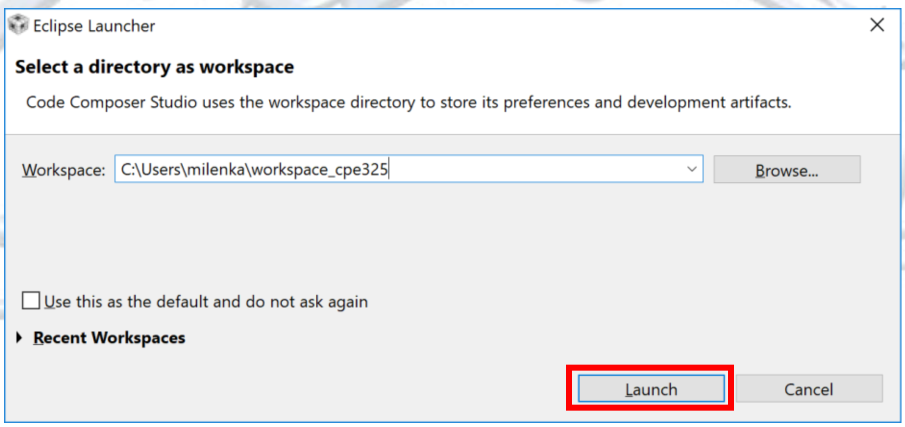


Figure 1. Eclipse Launcher

Step 3. Create a new project from the *Getting Started* menu. Click the <New Project> button. A New CCS Project window appears as shown in Figure 3.

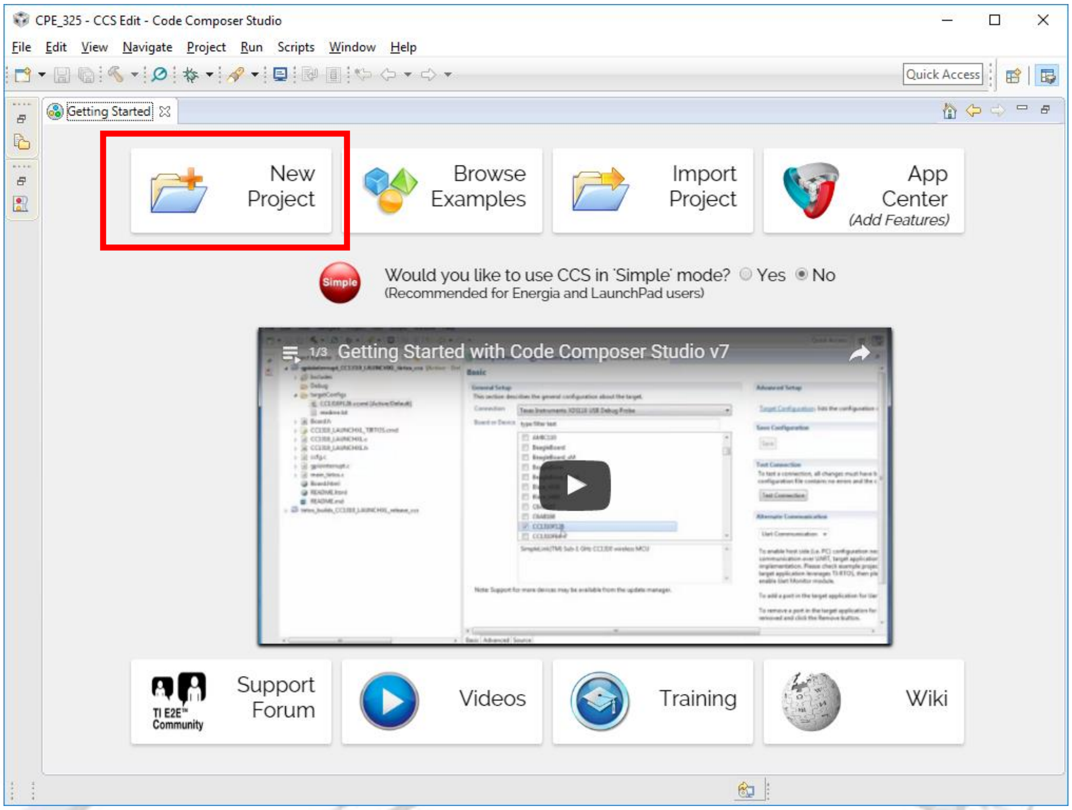


Figure 2. CCS Getting Started Page

Step 4. Name the project "Lab1_D1" and select "Empty Project" (see Figure 3.). For the target, select one of the following options depending on the target board: (a) MSP430FG4618 for the TI MSP430FG4618/2013 Experimenter's board; or (b) MSP430G2553 for the MSP430 Launchpad.

Step 5. Click <Finish>. The Project Explorer will show where all the project files are. Note that there are no source code files because we created an Empty Project (Figure 4).

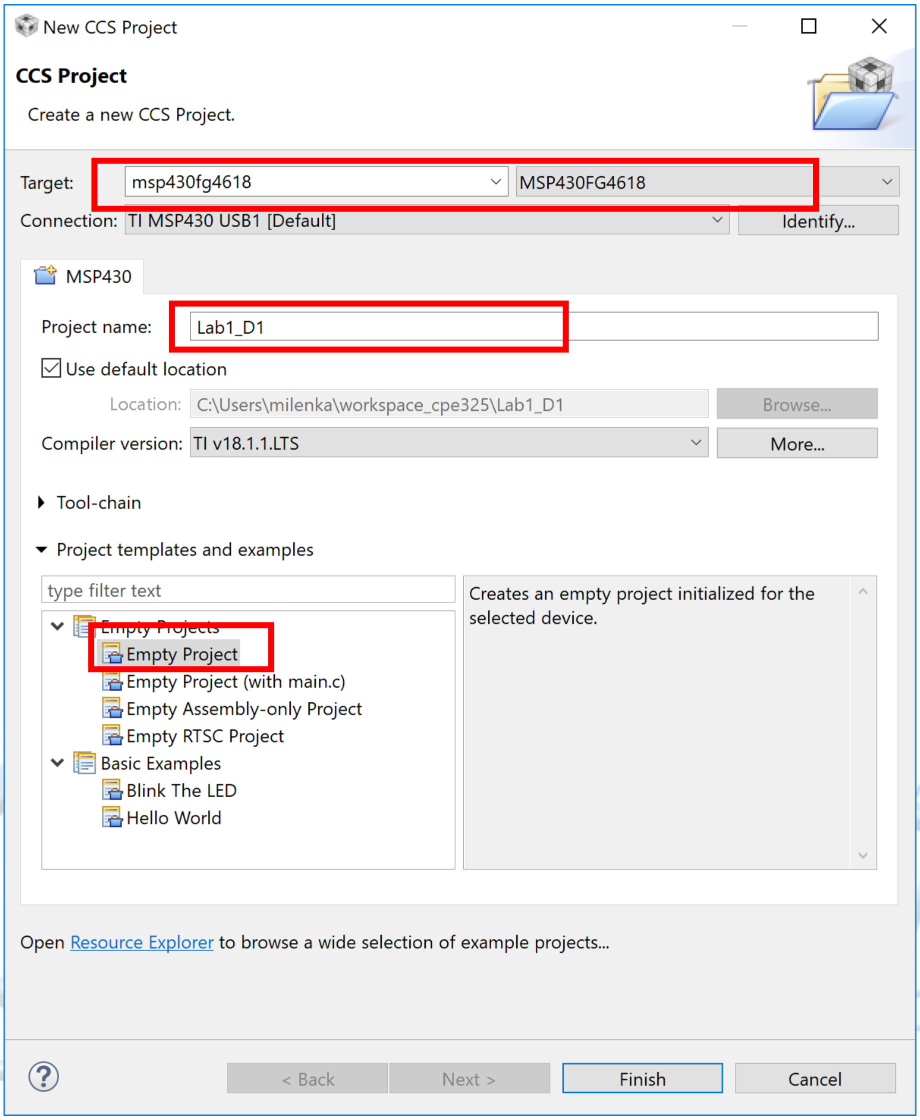


Figure 3. New CCS Project Setup

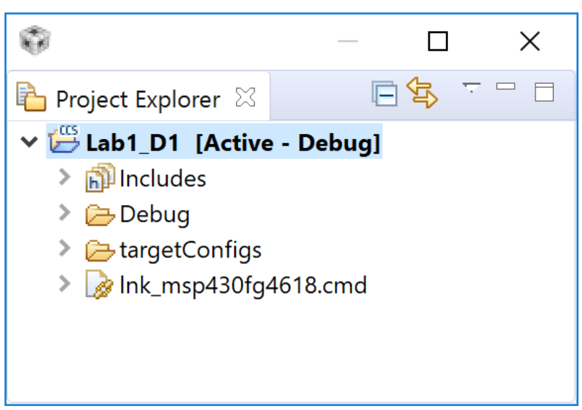


Figure 4 Project Explorer

Step 6. Copy the "Lab1_D1.c" and "twofact.c" from Windows Explorer to Lab_D1 in the project explorer. Right click on Lab1_D1 and select Add Files option. When prompted select "Copy Files" and click <OK>.

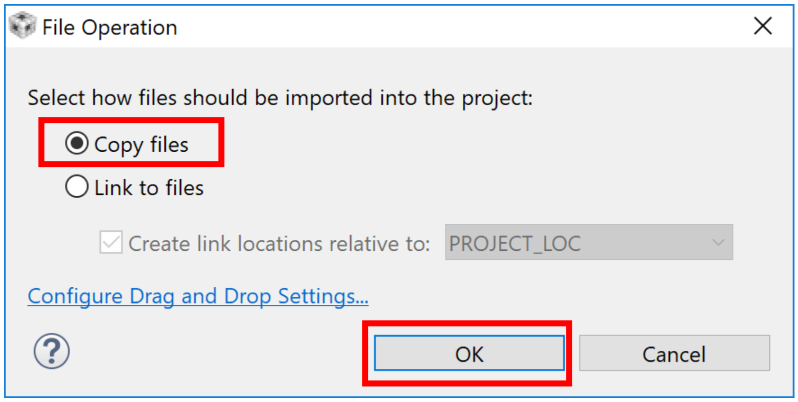


Figure 5. File Operation Dialog

Step 7. Double-click one of the files to see the source code (see Figure 6).

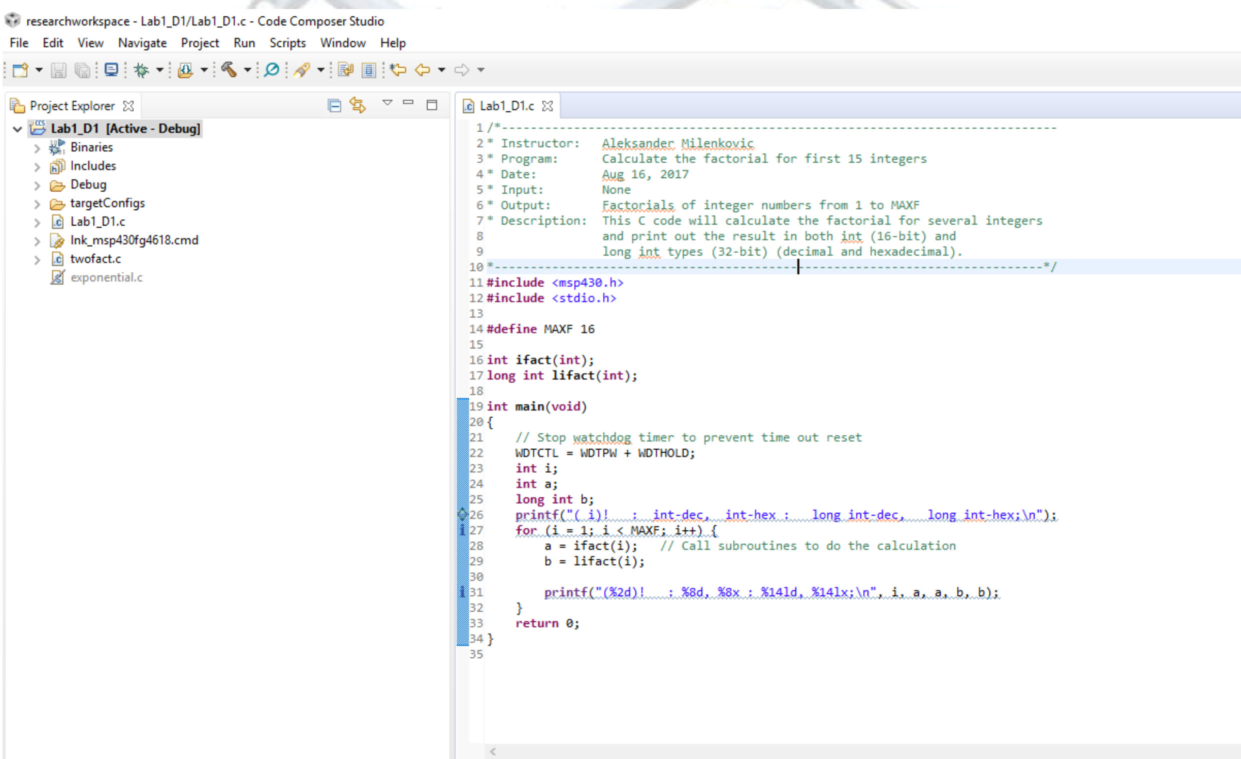


Figure 6. Code Composer Studio

1.2 Compiling and Linking the Application

You can now compile and link the application. You should also create a compiler list file and a linker map file and view both.

Step 1. Right-Click “Lab_D1” in the project explorer and select “Properties”. A new window will pop up as shown in Figure 7.

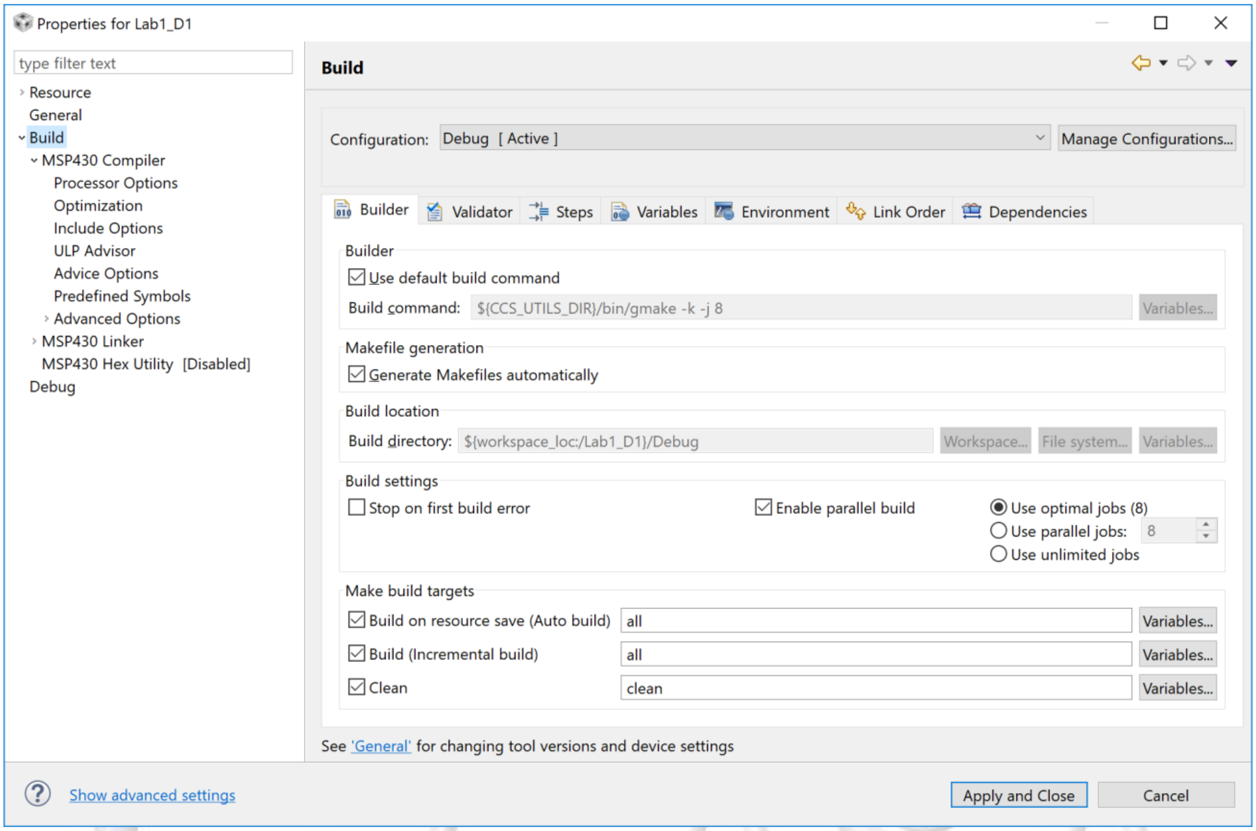


Figure 7. Project Properties

Step 2. From this menu, you can setup various project settings such as compiler optimizations, lst file generation, heap size, stack size, and the level of printf/scanf support. Below are several useful options.

- Select the “Build->MSP430 Compiler->Optimization.” This option allows you to set optimization settings. Change the “Optimization level” to off.
- Select the “Build->MSP430 Compiler->Advanced Options->Assembler Options” to setup the listing file options. Select the checkbox “Generate listing file (--asm_listing, -al)” (Figure 8).
- Select the “Build->MSP430 Compiler->Processor Options.” This allows you to set the code and memory model. Set the “Silicon version” to msp and the “code memory model” and “data memory model” to small. This will limit the instruction set to non-extended instructions.

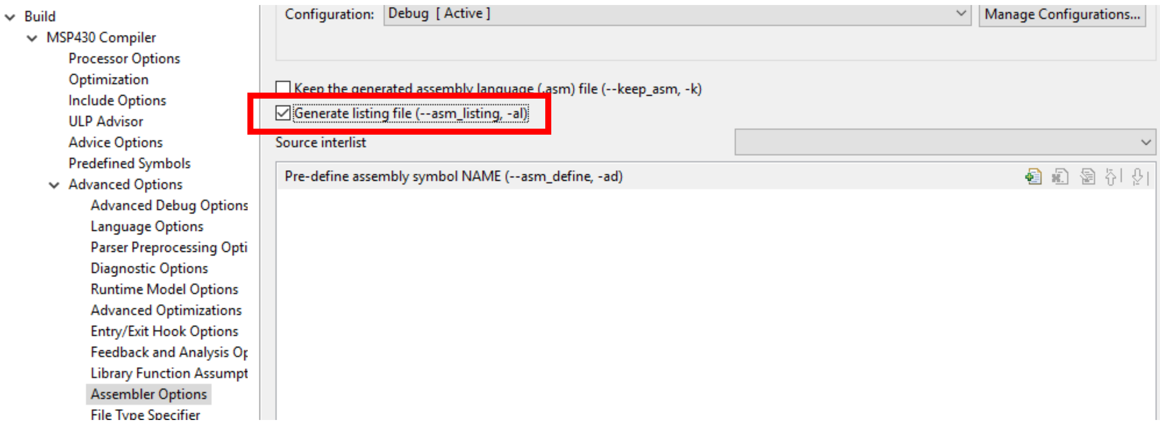


Figure 8 Project Assembler Options

- Select the "Build->MSP430 Compiler->Advanced Options->Language Options". Change "Level of printf/scanf support requires (--printf_support) to full.

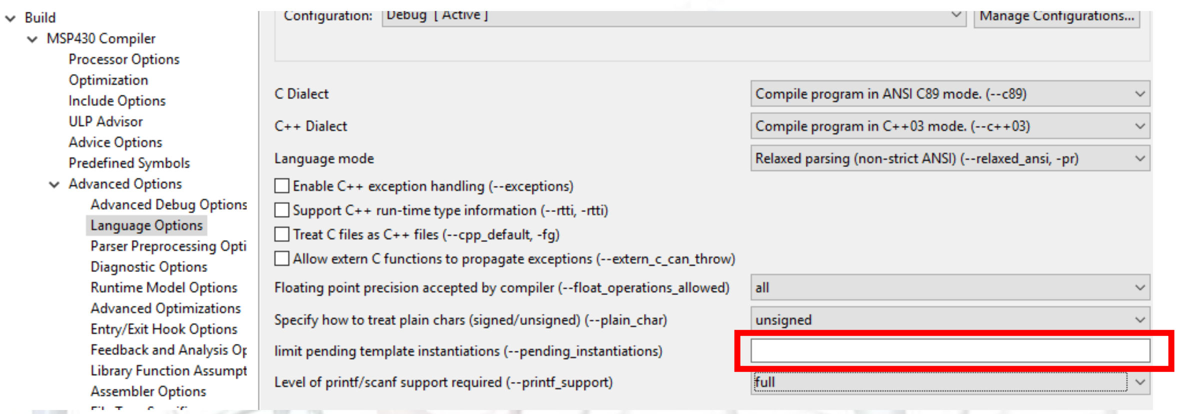


Figure 9. Compiler Language Options

- Select the "Build->MSP430 Linker->Basic Options" to set the heap size. Change "Heap size for C/C++ dynamic memory allocation (--heap_size, -heap) from 80 to 300. This change is necessary to get the printf debugging to work later in the lab. Click <Apply and Close> to close the properties windows.

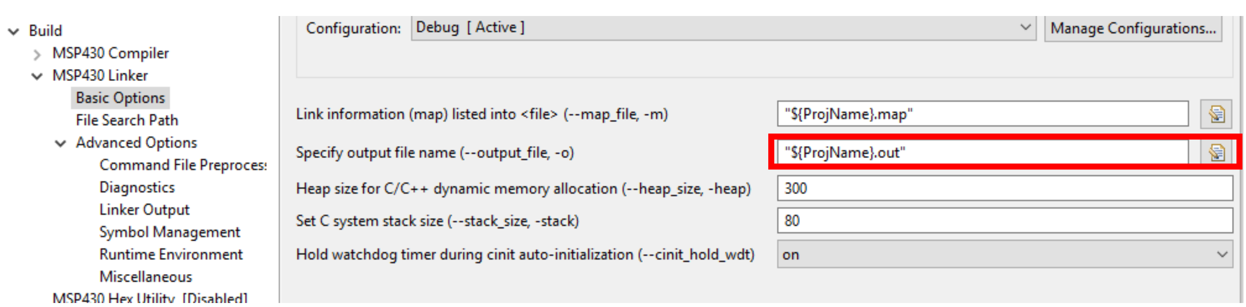


Figure 10 Project Linker Options

Step 3. To build a single file right-click the file in the Project Explorer and select "Build Selected File(s)" as shown in Figure 11. The Console window will show you if there are any build errors (Figure 12).

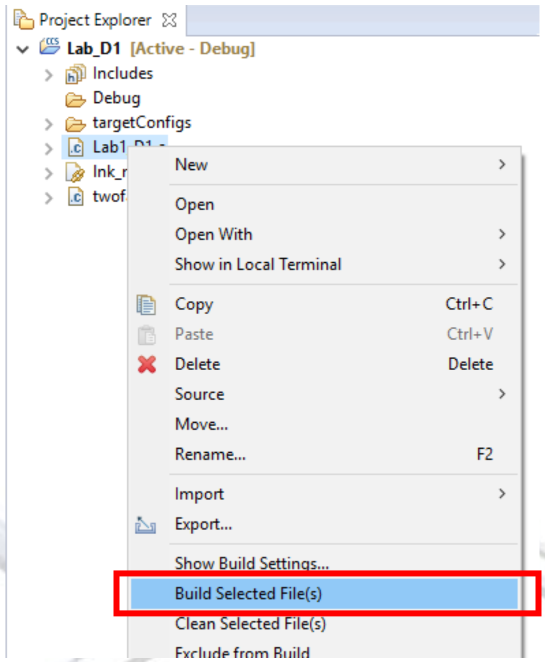


Figure 11. File Options Menu

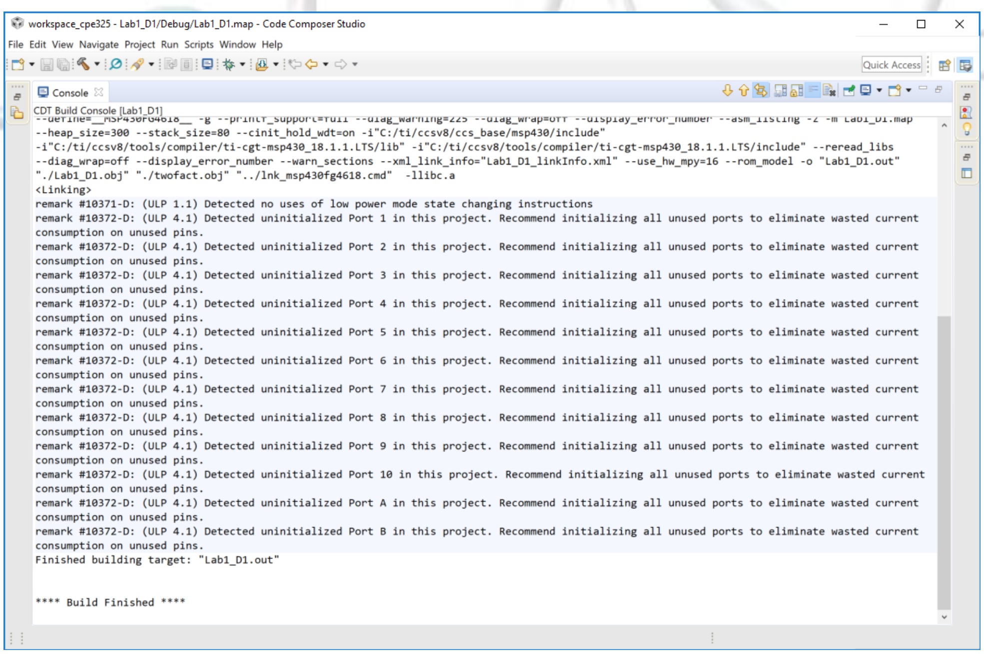


Figure 12. CDT Build Console

Step 4. Repeat the steps to build the "twofact.c" file. Click the  icon to build and link the project.

Step 5. Open the Debug folder in the Project Explorer to see the generated list files. The list files have the extension .lst. For example, twofact.lst is generated containing a C source code with the corresponding assembly language code for twofact.c source file. The files with extension .obj will be used as inputs to the linker. The debug executable has the extension .out and will be used as an input to the Debugger.

Step 6. Open the map file from the debug folder and examine the different sections and the space they will use in memory. The formatting can take some getting used to but there are several tools available for graphical analysis, including one inside of CCS. From the menu bar select "Memory Allocation" (Figure 13). Examine the Memory Allocation table to see which sections are placed into which type of memory (Figure 14).

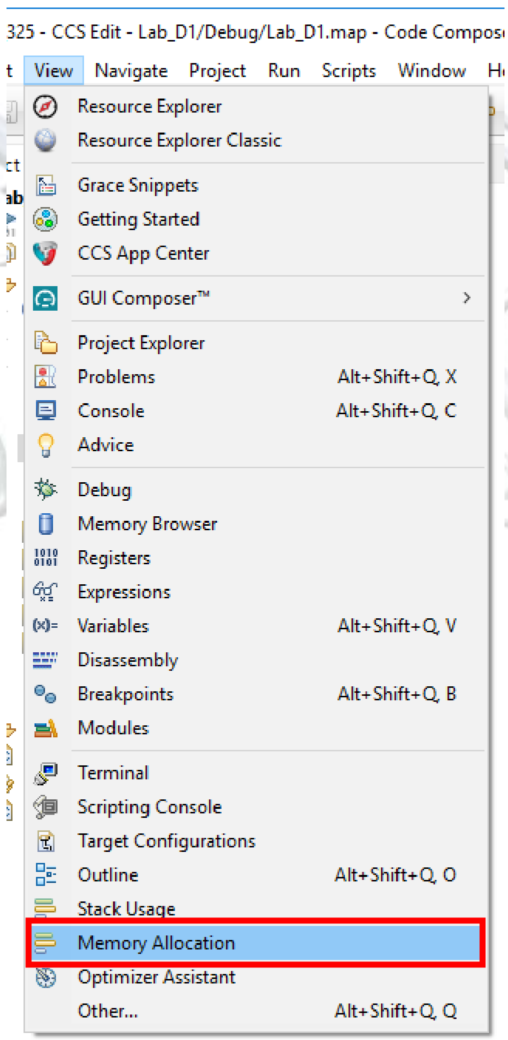


Figure 13. View Menu

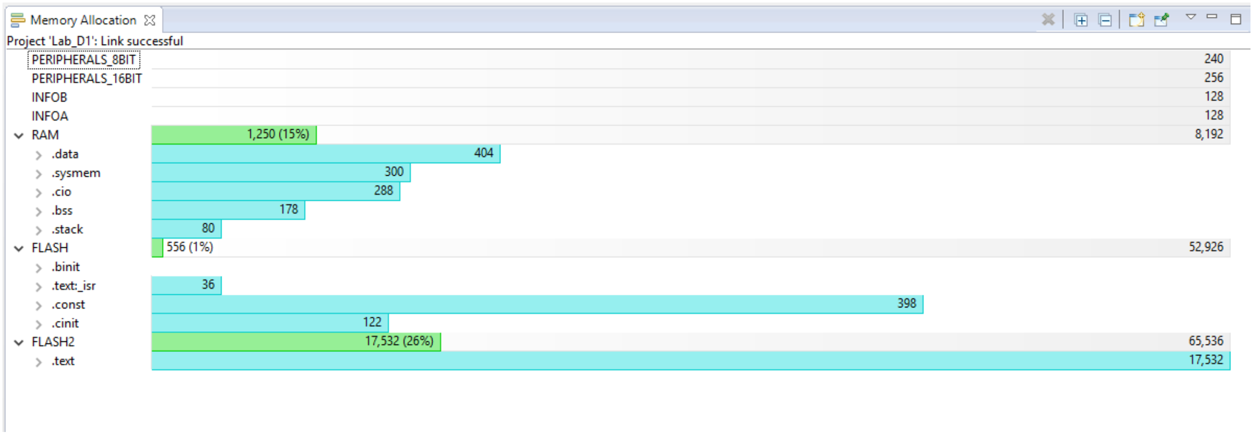


Figure 14. Memory Allocation Table

Step 7. You can also examine the estimated stack usage by selecting the "Stack Usage" option from the View menu. Change the optimization settings and example how the Memory Allocation and Stack Usage change from the various options. Reset the optimization level back to the off setting.

2 Debugging

This section continues the development cycle started in the previous section and explores the basic features of the Debugger.

2.1 Starting the Debugger

The correct debugger options were setup during the project creation, TI MSP430 USB1, and will work from both the Experimenter board and the Launchpad. Make sure the correct device is connected to your workstation.

Step 1. Click the  icon from the menu bar to start debugging.

Step 2. In the UPL Adviser Dialog (Figure 15) click the <Proceed> button.

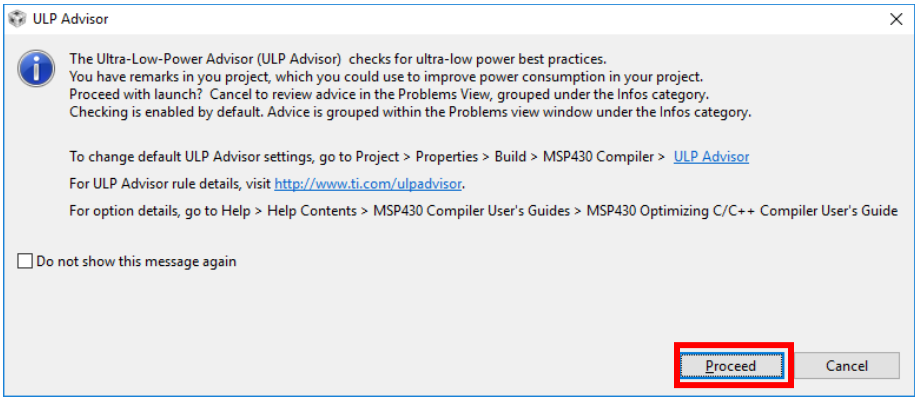


Figure 15. ULP Advisor

Step 3. After downloading the code, the application will be halted at the start of main as shown in Figure 16.

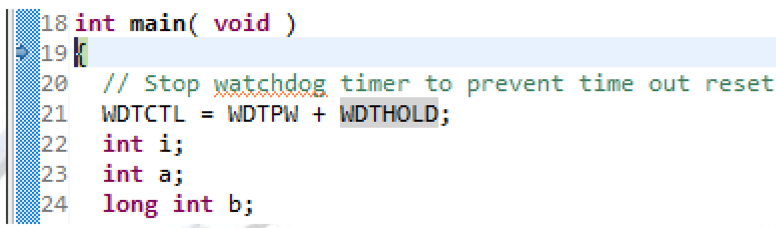


Figure 16. Debugger Entry Location

2.2 Inspecting Source Statements

The following debugging options are available for inspecting the code while it runs on the target hardware.

- Resume (F8). Run the program.
- Terminate (CTRL + F8). Stop execution of the program.
- Suspend (Alt + F8). Pause execution of the program.
- Step Into (F5). Step into a function.
- Step Over (F6). Step over the next line.
- Step Out (F7). Step out of a function.
- Restart . Start execution of the program from the beginning.

Step 1. Use the Step Over command until you have passed the ifact function as shown in Figure 16.

```
int main(void)
{
    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTHOLD;
    int i;
    int a;
    long int b;
    printf("( i)!    :  int-dec,  int-hex :    long int-dec,     long int-hex;\n");
    for (i = 1; i < MAXF; i++) {
        a = ifact(i);   // Call subroutines to do the calculation
        b = lifact(i);

        printf("(%2d)!    : %8d, %8x : %14ld, %14lx;\n", i, a, a, b, b);
    }
    return 0;
}
```

Figure 16. Passing the ifact function

2.3 Inspecting Variables

CCS allows you to watch variables or expressions in the source code, so that you can keep track of their values as you execute your application. You can look at a variable in many ways. For example, you can view a variable by pointing at it in the source window with the mouse pointer, or by opening one of the Variables, or Watch windows.

Note: When optimization level off is used, all non-static variables will live during their entire scope and thus, the variables are fully debuggable. When higher levels of optimizations are used, variables might not be fully debuggable. A good rule of thumb is to increase optimization until the program meets design constraints. For this lab turning off optimization will meet these constraints. By default, CCStudio shows you the Variables window which gives you a list of the current variables in the scope of the executing code (e.g., Figure 17).

(x)= Variables | Expressions | Registers

Name	Type	Value	Location
(x)= a	int	1	Register R9
(x)= b	long	1	R13:16,R12:16
(x)= i	int	1	Register R10

Figure 17. Local Variables Window

2.3.1 Setting a Watchpoint

Next you will use the Watch window to inspect variables.

Step 1. Double click one of the variables to highlight it in the source code. Next right-click it and select “Add Watch Expression…” (Figure 18).

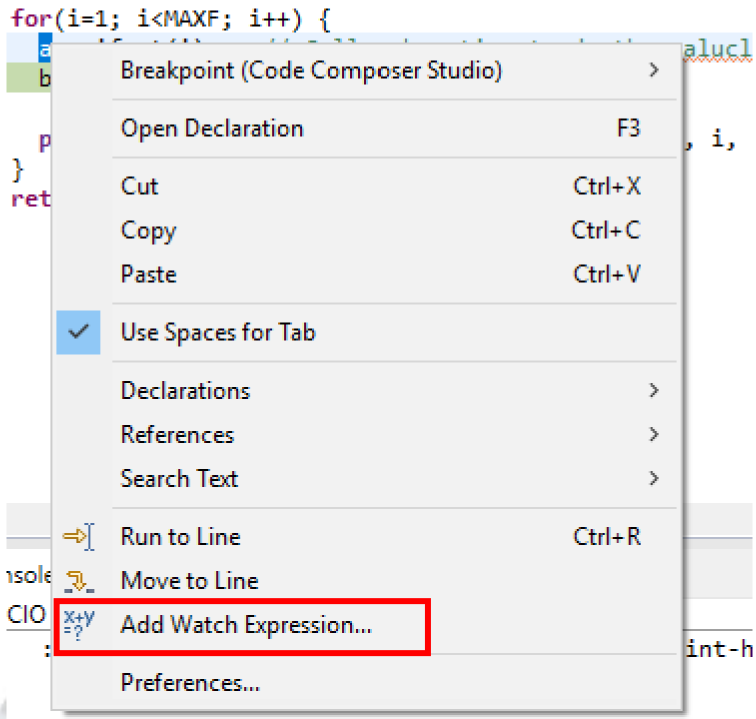


Figure 18. Text Options Menu

Step 2. In the Add Watch Expression Dialog box (Figure 19) click the <OK> button.

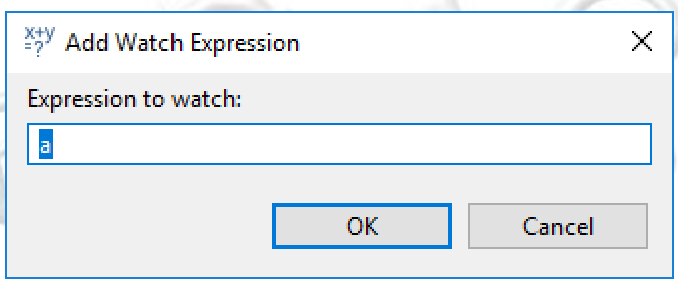


Figure 19. Add Watch Expression Dialog

The variable should now appear in the Watch Expressions Window as shown in Figure 20.

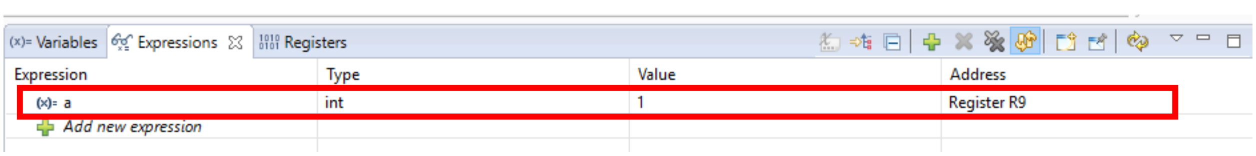
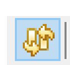


Figure 20. Watch Expressions Window

The Continuous Refresh  button can also be used to update the value while the program is running.

2.3.2 Setting and Monitoring Breakpoints

Breakpoints can be added to the code by double-clicking the blue shaded area next to the line number where you want the breakpoint to stop (e.g., Figure 21 Line 28).

```
26     printf("( i)!    :  int-dec,  int-hex :    long int-dec,    long int-hex;\n");
27     for (i = 1; i < MAXF; i++) {
28         a = ifact(i);    // Call subroutines to do the calculation
29         b = lifact(i);
30
31         printf("(%2d)!    : %8d, %8x : %14ld, %14lx;\n", i, a, a, b, b);
32     }
33     return 0;
```

Figure 21. Breakpoint in code (Line 28)

2.3.3 Executing up to a Breakpoint

Pressing the Resume button or F8 will run the program until the breakpoint is reached.

2.4 Debugging in Disassembly Mode

Debugging in the disassembly can be done by selecting “View->Disassembly” (Figure 22) from the menu bar.

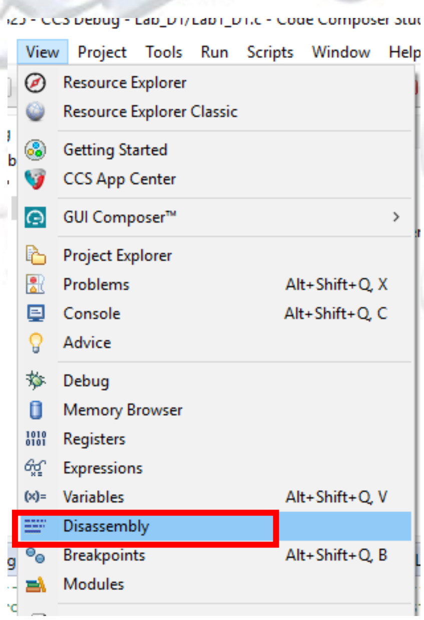


Figure 22. View Menu

In this view as shown in Figure 23, you can see how the compiler has translated your C code to the machine language of the processor.

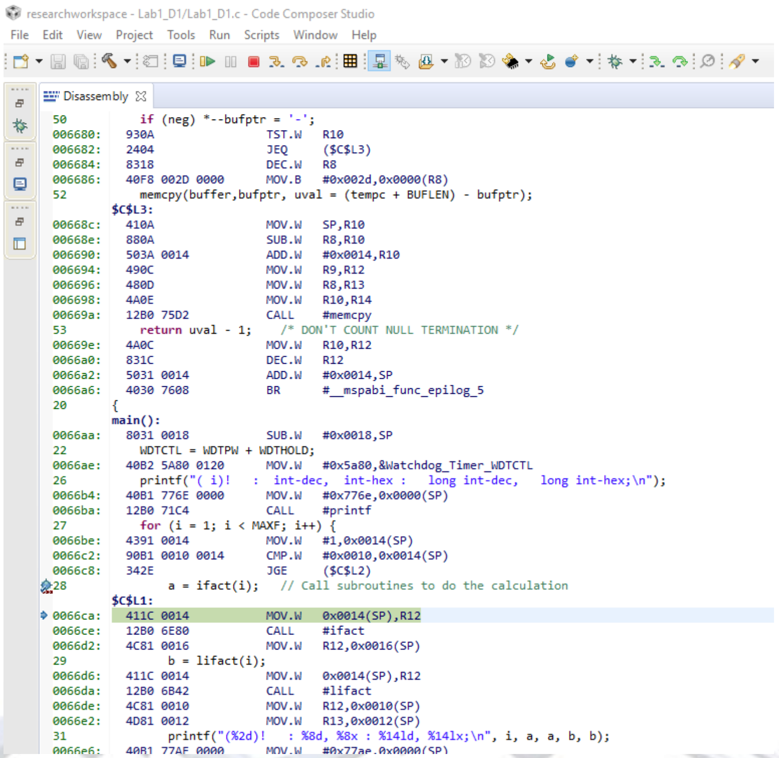


Figure 23. Disassembly View

2.5 Monitoring Registers

The Register window lets you monitor and modify the contents of the processor registers. Notice registers PC (Program Counter), SP (Stack Pointer), SR (Status Register), and R4-R15 (general-purpose registers) in Figure 24.

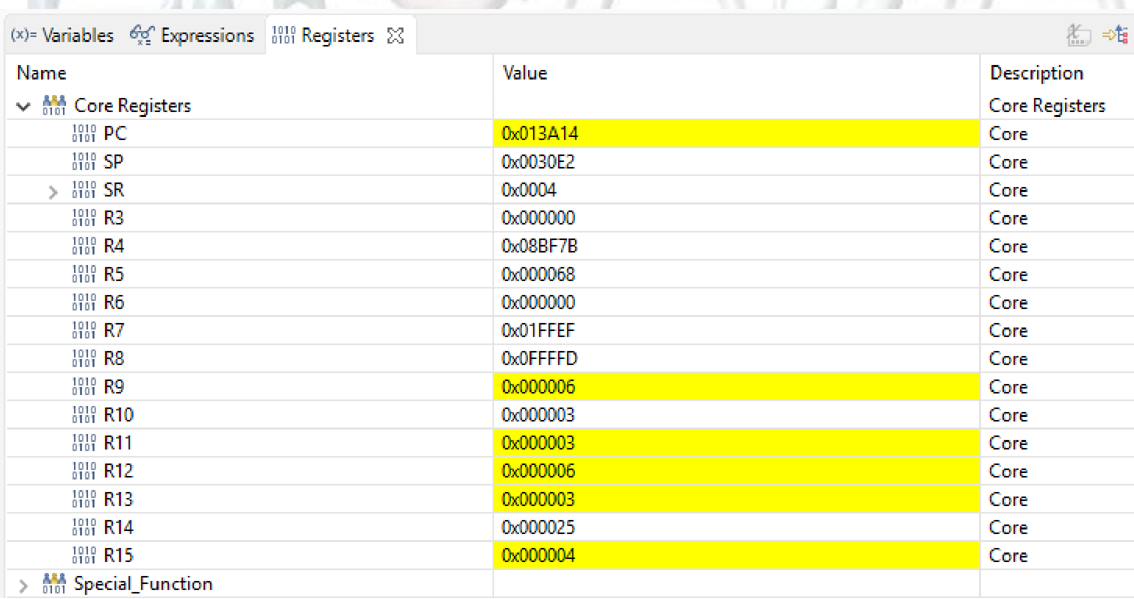


Figure 24. Registers Window

Step 1. Step Over to execute the next instructions and watch how the values change in the Register window.

Step 2. Close the Register window. To see the cycle count, click on Run->Clock->Enable (Figure 25) from the debug perspective.

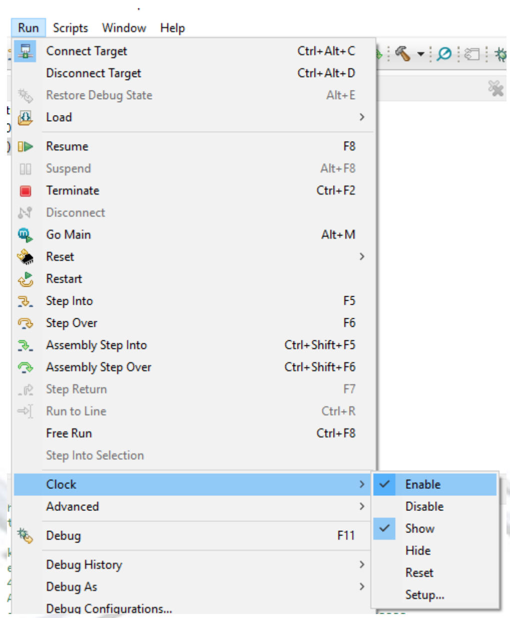


Figure 25. Run Menu

This icon () will show in either the bottom of CCS or in the status bar. More information about the profile clock can be found at http://processors.wiki.ti.com/index.php/Profile_clock_in_CCS

2.6 Monitoring Memory

The Memory window (Figure 26) lets you monitor selected areas of memory. You can select RAM, flash, or SFR portion of the memory.

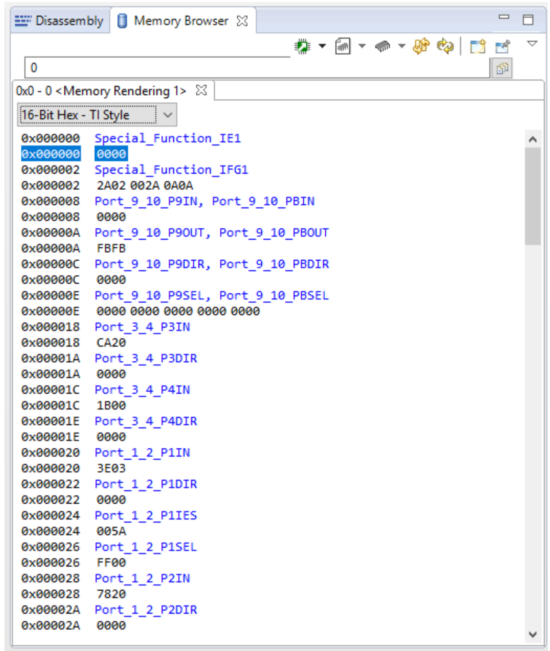


Figure 26. Memory View

Step 1. To open the Memory Browser, select "View->Memory Browser" (Figure 27) from the menu bar.

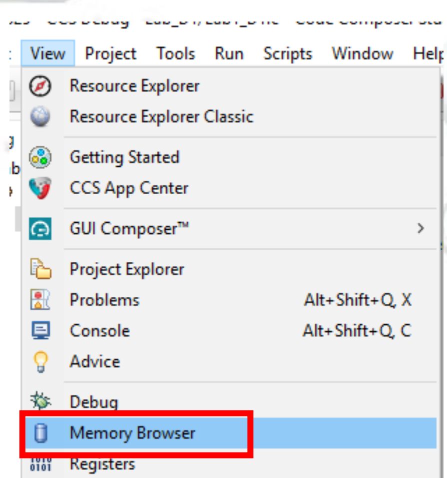


Figure 27. View Menu

Step 2. If all of the memory units have not been initialized yet, continue to step over and you will notice how the memory contents will be updated.

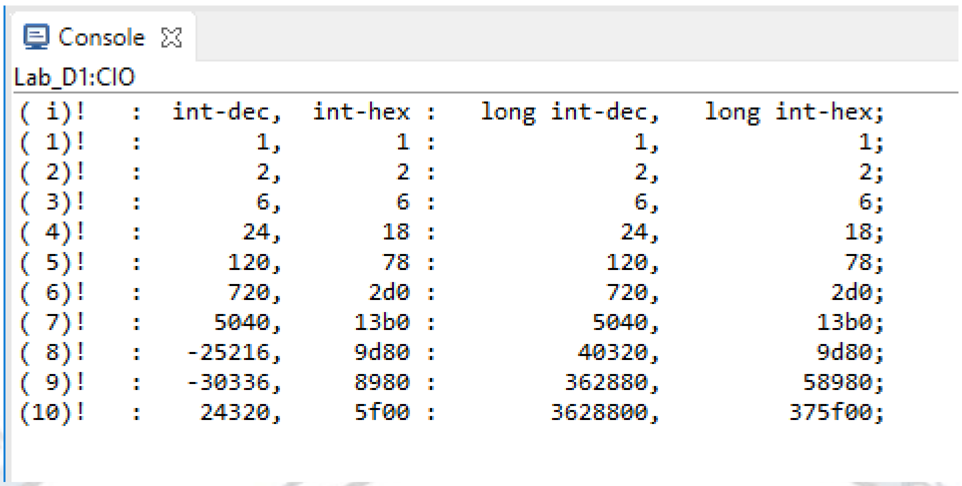
Step 3. You can change the memory contents by editing the values in the Memory window. Just place the insertion point at the memory content that you want to edit and type the desired value.

Step 4. Close the memory window.

2.7 Viewing Terminal I/O

Sometimes you might need to debug instructions in your application that make use of stdin and stdout without the possibility of having hardware support. CCS lets you simulate stdin and stdout by using the Console (e.g., Figure 28).

Note: While the correct settings were set above for printf support in this program, some programs may need more heap memory to correctly print to the console. More information about printf can be found at http://processors.wiki.ti.com/index.php/Tips_for_using_printf



```
Console
Lab_D1:CIO
( i)!   :  int-dec,  int-hex :   long int-dec,   long int-hex;
( 1)!   :        1,        1 :              1,              1;
( 2)!   :        2,        2 :              2,              2;
( 3)!   :        6,        6 :              6,              6;
( 4)!   :       24,       18 :             24,             18;
( 5)!   :      120,       78 :            120,             78;
( 6)!   :      720,      2d0 :            720,            2d0;
( 7)!   :     5040,     13b0 :           5040,           13b0;
( 8)!   :   -25216,     9d80 :          40320,           9d80;
( 9)!   :   -30336,     8980 :         362880,          58980;
(10)!   :    24320,     5f00 :        3628800,         375f00;
```

Figure 28. Console Output of Execution Program

Note: The contents of the window depend on how far you have executed the application.

3 Software Documentation

Maintaining good software documentation is of key importance, especially when your code may be used by others. A special focus should be placed on your header and comments, code formatting, and software diagrams or flowcharts.

3.1 Code Formatting and Organization

Part of good programming is ensuring that the code is neatly formatted and organized. This not only helps others who may need to access your code, but it also helps in debugging and maintaining your own code. Here are some general guidelines to assist in organizing your code in the Code Composer Studio IDE:

- Ensure all your function prototypes are declared after your #include statements at the top of the program.
- Declare all your global variables directly after your function prototypes in one area
- Consistently organize your functions. A good way to organize functions is to have your main function, followed by the other functions in order of call, followed by your interrupt functions. You can choose a different way, but make sure to organize the types of functions and maintain consistency.

- Keep track of your indention. In the CCStudio IDE, a tab is four spaces. Each function, loop, or other “nest” should be indented appropriately and consistently.
- It is especially important to keep track of where your code is located on your workstation. CCStudio uses workspaces to help you keep track of your code. It is recommended that you create a directory where you will keep all your projects. Create subdirectories for each lab assignment. Your code is your creative expression and thus take care of it. Your engineering reputation will depend on the quality of your code.

3.2 Code Headers and Comments

By now, you have become familiarized with commenting in your programs. Comments help you and others keep up with the flow of the code, and it is important to maintain good comments. In this Laboratory, you will be programming in C and assembly language. In assembly language, you generally should comment every line of code to explain its purpose. The reason for this is that the code is much less self-explanatory than common coding languages. In C, there are generally a few guidelines to remember when commenting:

- You should always include a header at the top of your code that gives basic information about your program (what it does and how), author, and when the code was written.
- Each variable declaration should be commented.
- Each function declaration should be properly noted.
- Any segment of code, whether it is to initialize hardware, perform a calculation, or do another task, should have concise comments that explain it.

3.3 Software Flowcharts

A flowchart is a helpful way for you to decide on an approach to your program before you begin. It is also an extremely effective way of concisely relaying how your code works to others. A flowchart does not contain information about every line of code, but it is a slightly higher-level picture that shows logically how problems are addressed. Hardware initializations and variable declarations should be documented. Also, any logical steps, function calls, or loops should be noted as well as their respective conditions.

A flowchart for calculating the factorial can be seen below in Figure 29. Note that it does not include every line of code, but it does capture the main steps in the program.

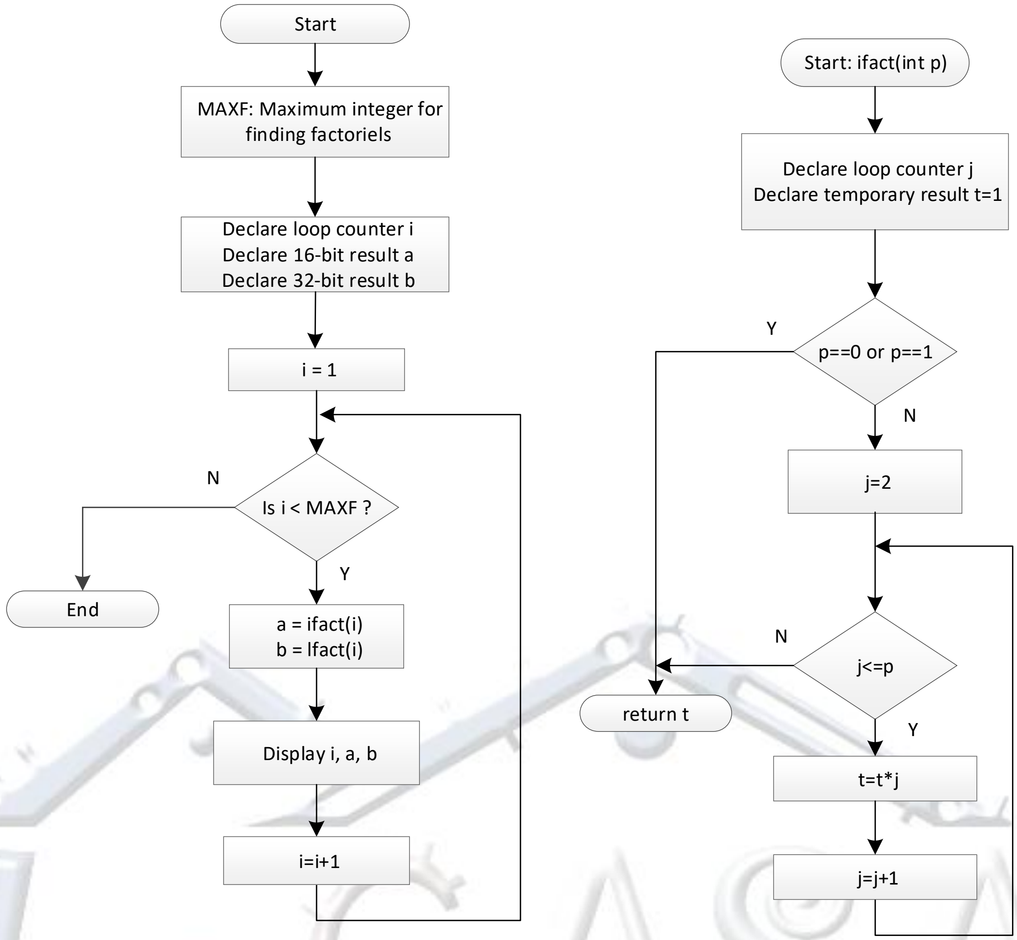


Figure 29. Flowcharts for Calculating the Factorial