
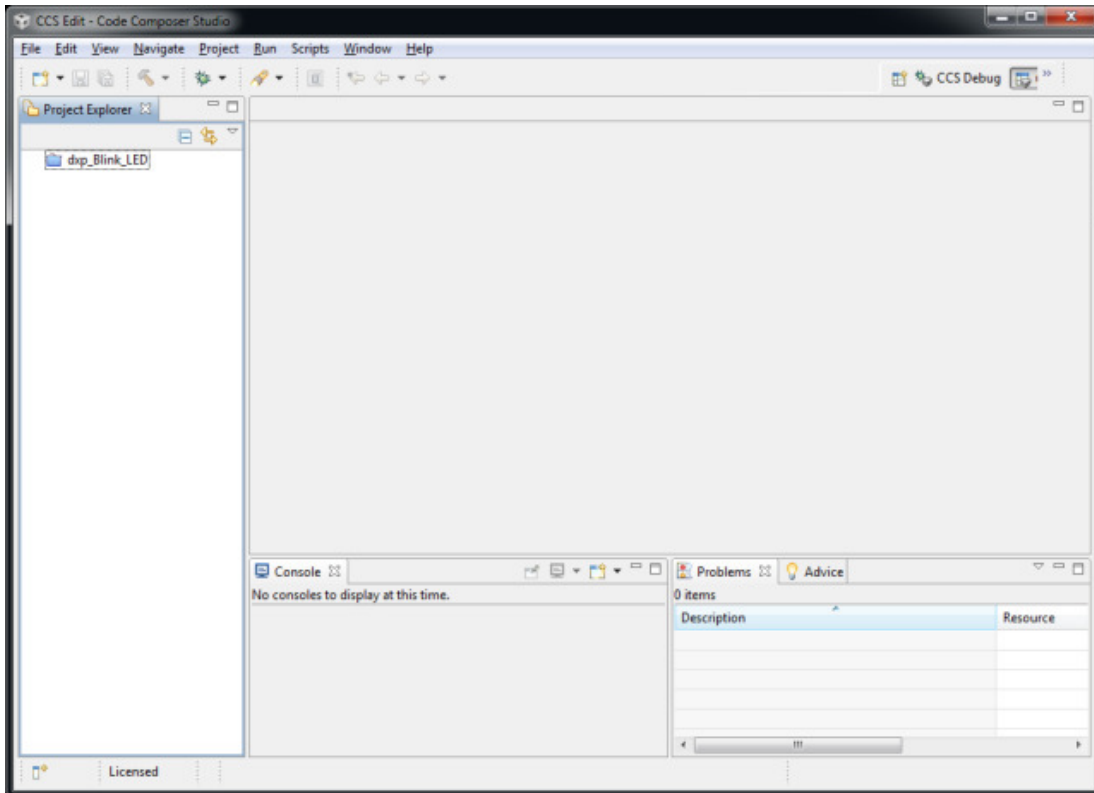
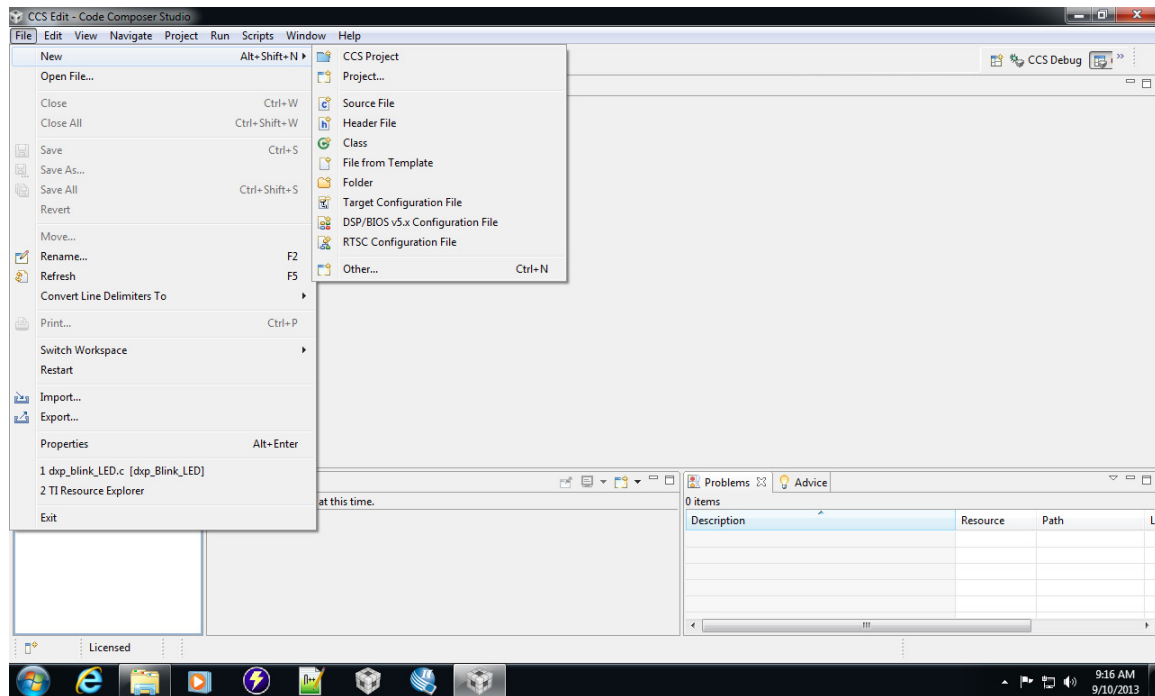


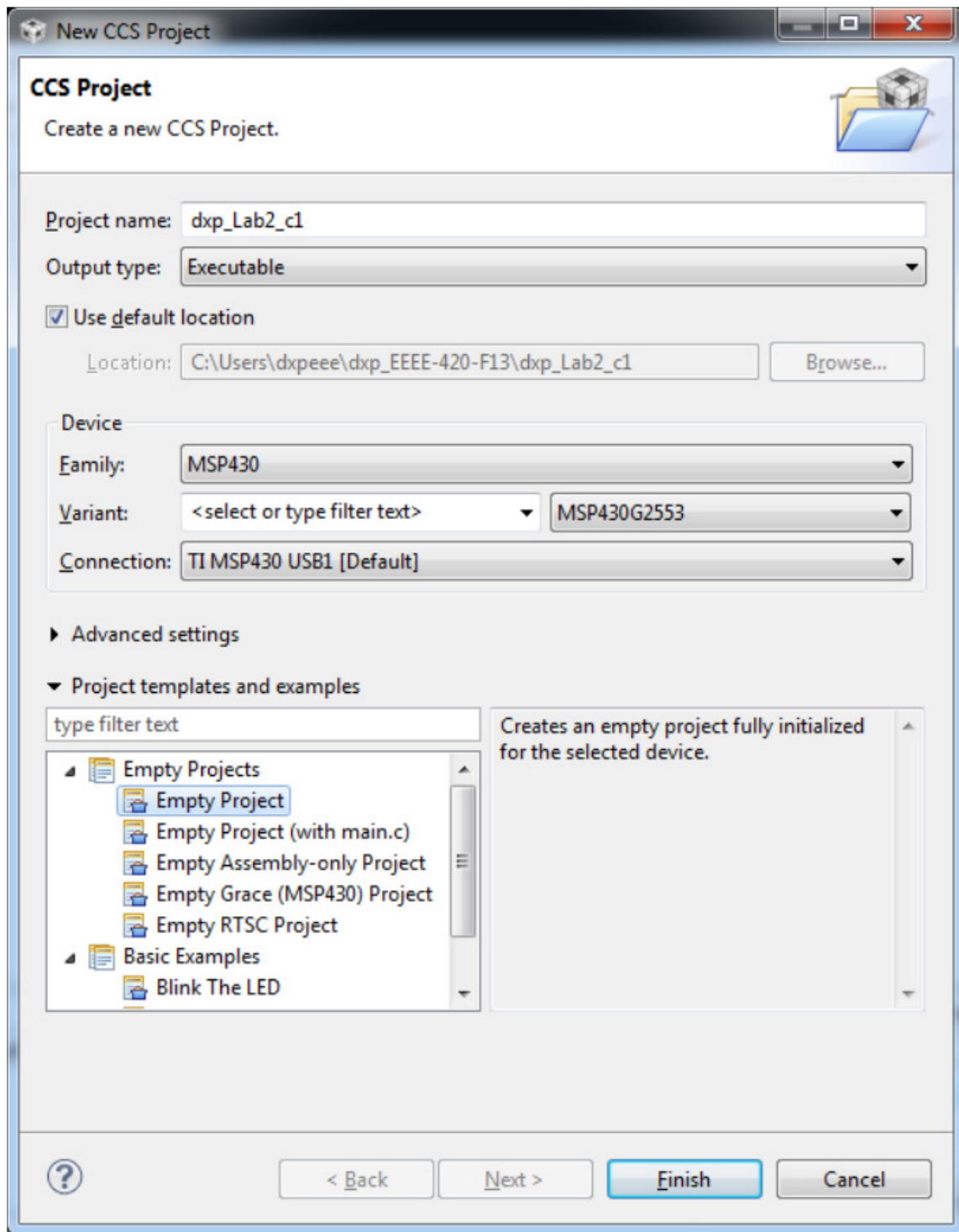
- 1) From mycourses download this pdf, dxp_Lab2_a1.c, and dxp_Lab2_c1.asm.
- 2) The purpose of this lab is to gain more familiarity with the LaunchPad board, which is designed around a TI MSP430 microcontroller, and utilization of the Code Composer Studio (CCS) Integrated Development Environment (IDE) - version 5.4. You will learn how to create a software Project that can be used to produce an executable program that allows interaction with the MSP430.
- 3) Part 1. Become familiar with the CCS IDE “build” process and debugger, and with C code for the MSP430.
- 4) Connect the hardware to the computer and open the CCS IDE by double-clicking on its icon: 
- 5) After a brief display of the application logo, a window appears in which you should select as your workspace the directory you have created last time: fmlxxxx_EEEE-420. In the lab handouts I will often refer to fmlxxxx_, but in the windows captures you will see the initials dxp_ or cab_. If the window below doesn't appear, you can still choose or switch the workspace by going to: File > Switch Workspace > Other. Use the same workspace every week, and just add your weekly projects to that single workspace.
- 6) NOTE: In general, the computers in the EE labs are “purged” of personal files on a weekly basis. You should make sure to back up your workspace after every Lab to some media (online or otherwise) that you have control over.
- 7) The application window is shown below. It looks overwhelming at first glance, but after you will start using and become familiar with it, it will not feel more difficult to use than any other CAD tool.



- 8) Let's explore for a few moments the application window above. On the top you have a selection of text-based menus, which combined with the dropdown menus cover every available command. The most often used commands in a specific context can be selected using icons, if you care to memorize these. Then the window is divided into panes, which number and orientation you can change from the View menu. The left hand-side (LHS) pane will display the project structure and associated files. The center pane will display your source files, C or assembly. The center lower left pane Console is equivalent to a terminal prompt. It echoes everything the application is doing. The center lower right pane Problems contains a list of warning and errors that the compiler/assembler may generate during their run. Finally, on the right hand-side (RHS) the panes will usually contain helpful, context related information. As we will continue next to create a project, build, and debug it, we will explain the meaning and use of each new pane.
- 9) We will next create a project. A project contains all source, include, link, debug, and any other necessary files to create a program for the MSP430. From the file menu, choose New and then CCS Project.

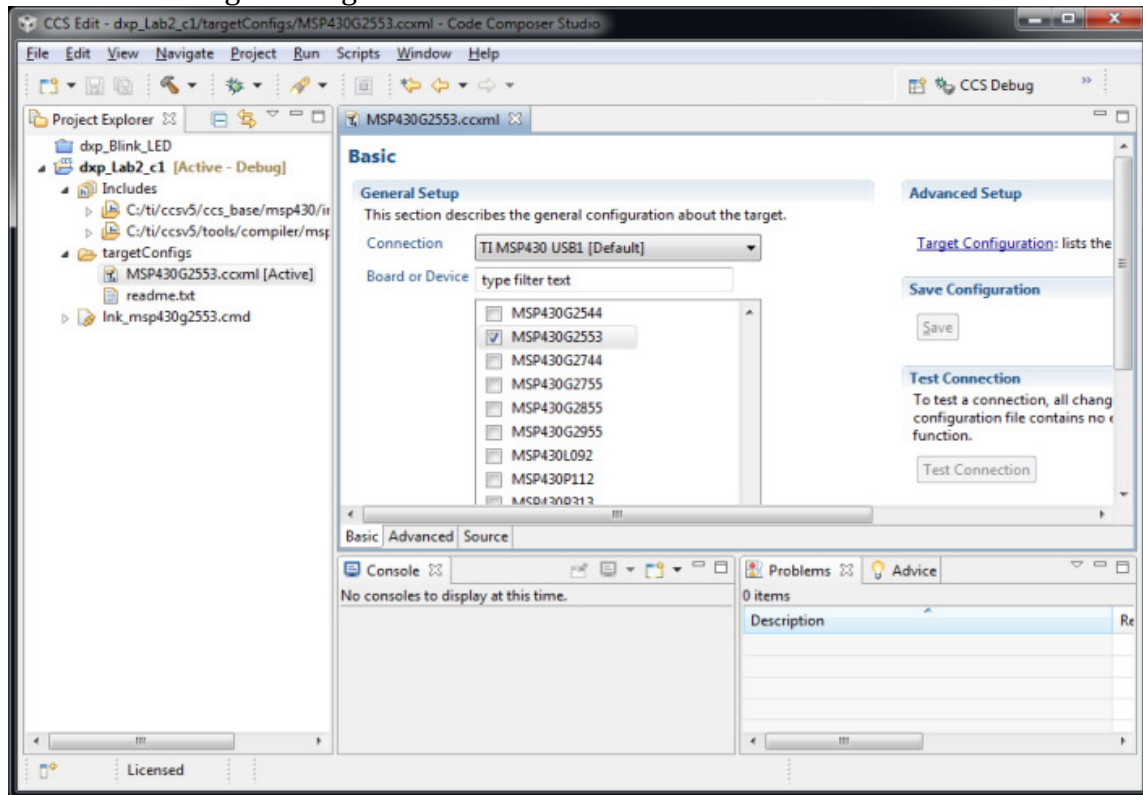


- 10) In the new window choose for the project title fmlxxxx_Lab2_c1. The 'c' stands for a C project, and we will use shortly 'a' for an assembly project. If the default location is your workspace, the project will be created in your workspace.
- 11) Leave Executable for Output type and Use default location checked. The Family is selected by default to be MSP430. **Make sure that the Variant is MSP430G2553** – by now it may show up as default too. Further, the connection is TI MSP430 USBx, where x is the current USB port number through which the board is connected.
- 12) Finally, before clicking Finish make sure you select an Empty Project from the Project templates pane. **For C only projects you could also choose Empty Project (with main.c).** **For assembly only projects in the future you will have to choose at this point Empty Assembly-only Project.** Click Finish.

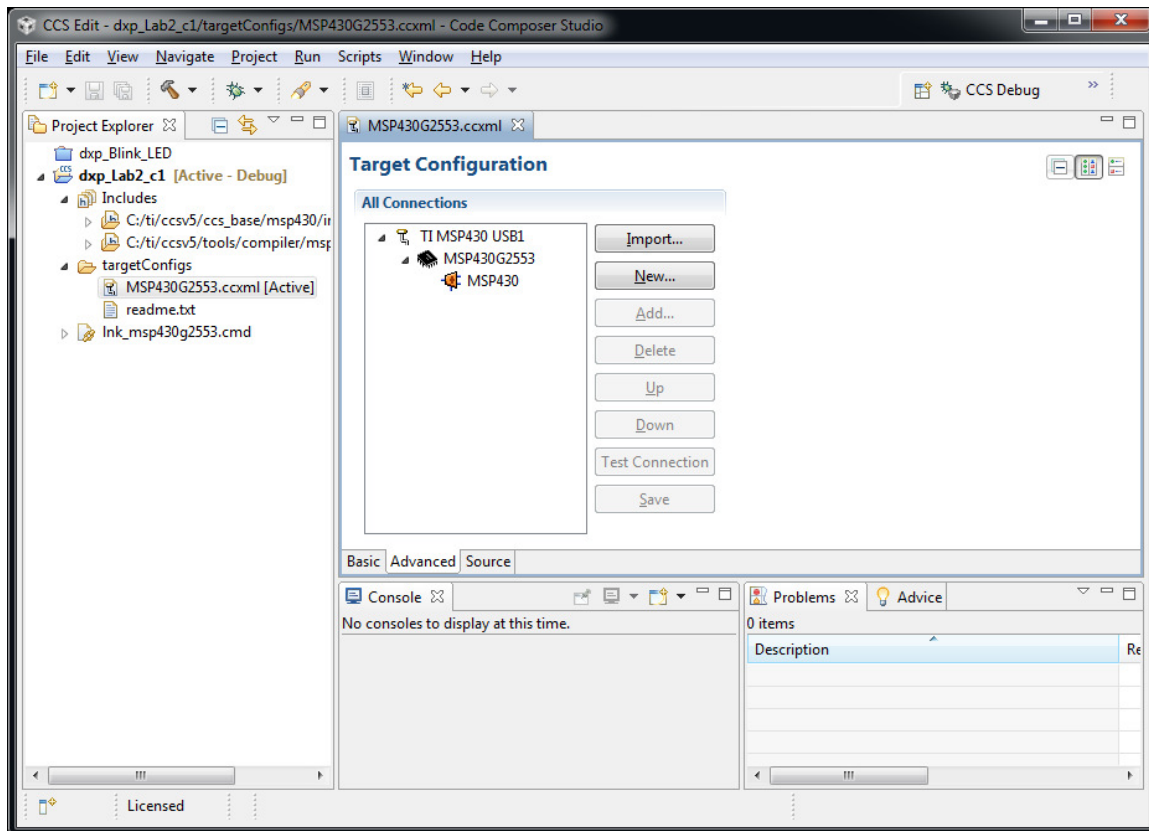


- 13) At this point, you should see in the LHS pane an active project with the name fmlxxxx_Lab2_c1. Click on the + sign to unroll information about its structure and associated files.
- 14) The Includes contains links to device specific include header files – remember we have already chosen the device MSP430G2553. If you are curious to see the include files, you can click on the + signs, but it will take a very long time to display them. Be patient, the application may not be responsive for some time (~min.).

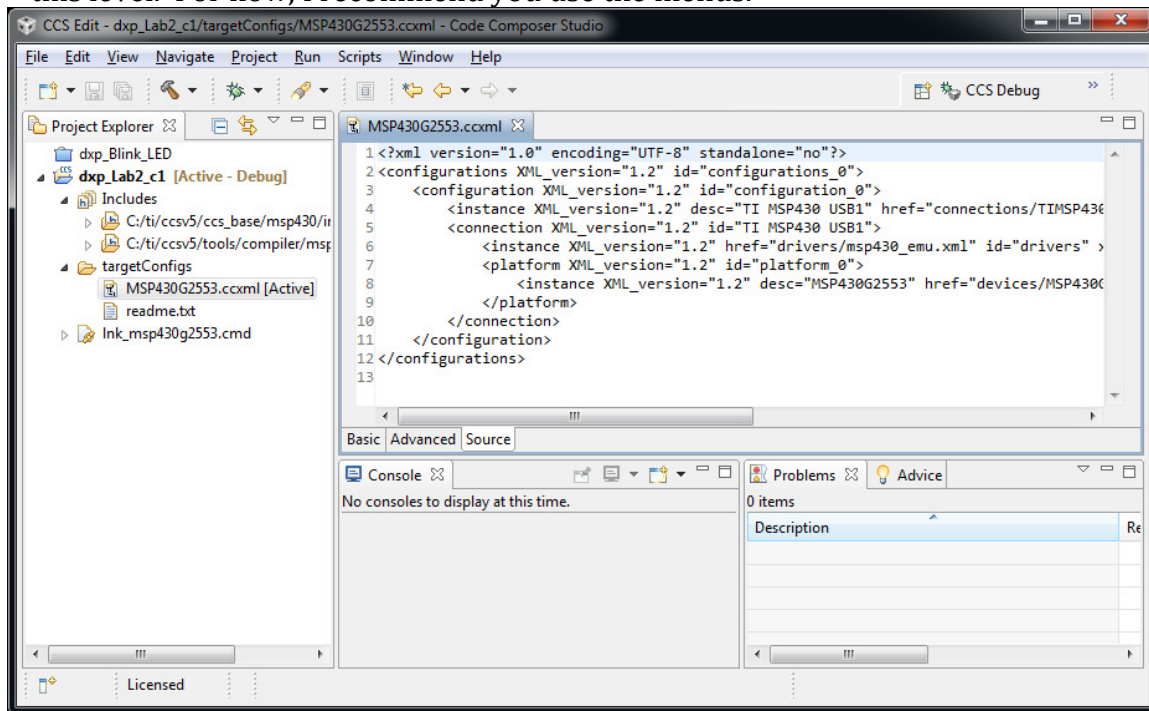
- 15) Information about the target device and emulation/debugging information is captured in the file MSP430G2553.ccxml. This is an *.xml file for Code Composer, hence the extension *.ccxml. You can edit it manually, i.e. using a text editor, or let the tool take care of that. Double-click the file name. In the RHS main pane you can now make target configuration selections.



- 16) Depending on your display resolution, you may need to scroll more or less. In the Board or Device box you should have MSP430G2553 selected. In the Connection box you see the USB port that was selected when you created the project. This is the place where you can change it if the hardware connection changes. Because you didn't change anything there's no need to Save at this point. Just for your information, before closing the MSP430G2553.ccxml pane, click the Advanced tab. Here you can see all available connections.



17) Now click on the Source tab. If you are xml proficient, you can change the setup at this level. For now, I recommend you use the menus.

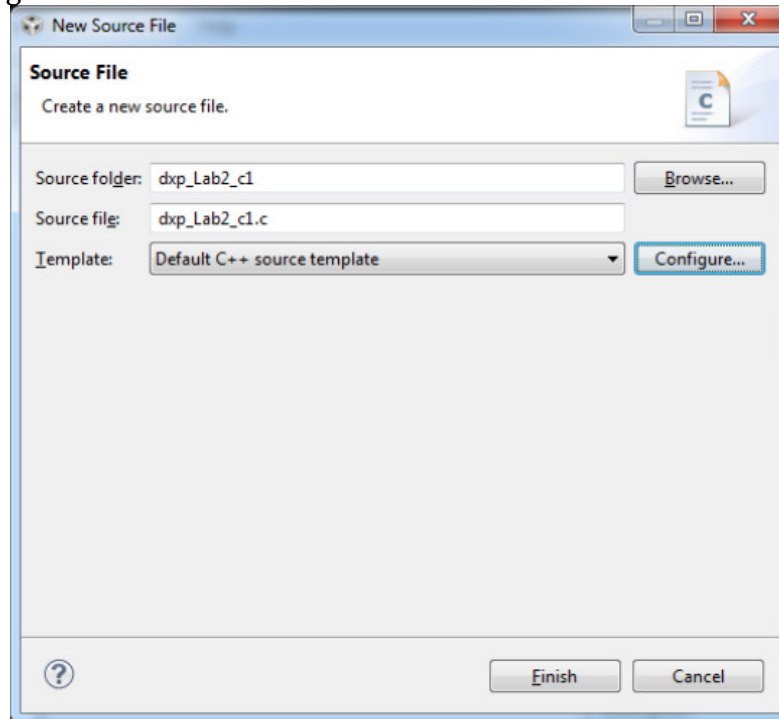


18) Close the MSP430G2553.ccxml file and pane, and the cheat sheet.

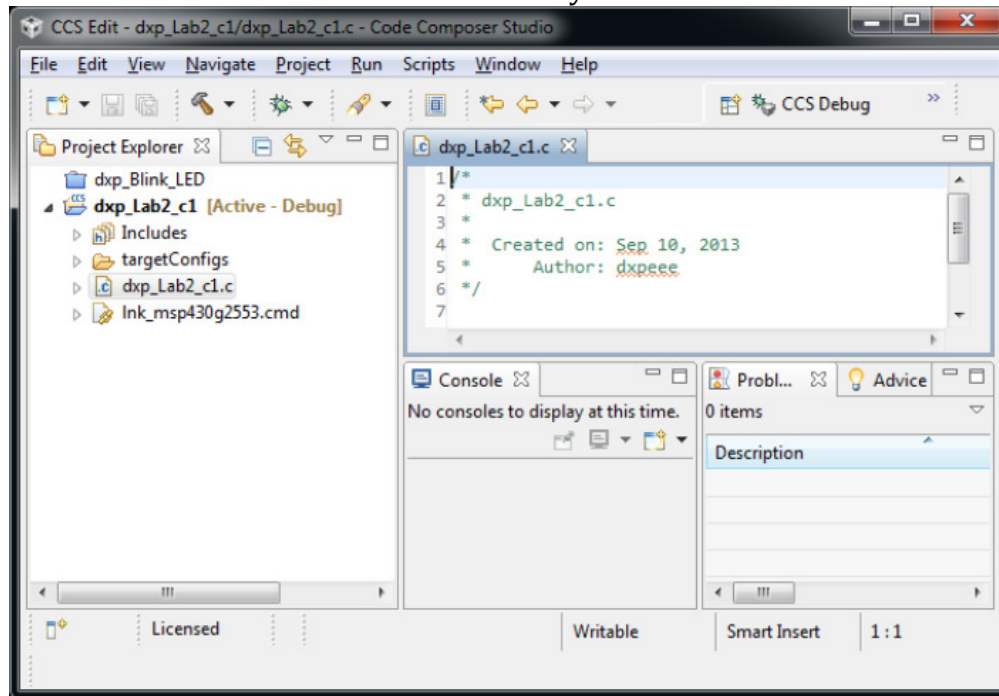
19) The second file of interest is lnk_MSP430G2553.cmd. Double click on it in the Project Explorer pane. This file was automatically generated during the project

creation step and contains device specific linking information, a process that we will explain later. Don't change anything and close the file and pane.

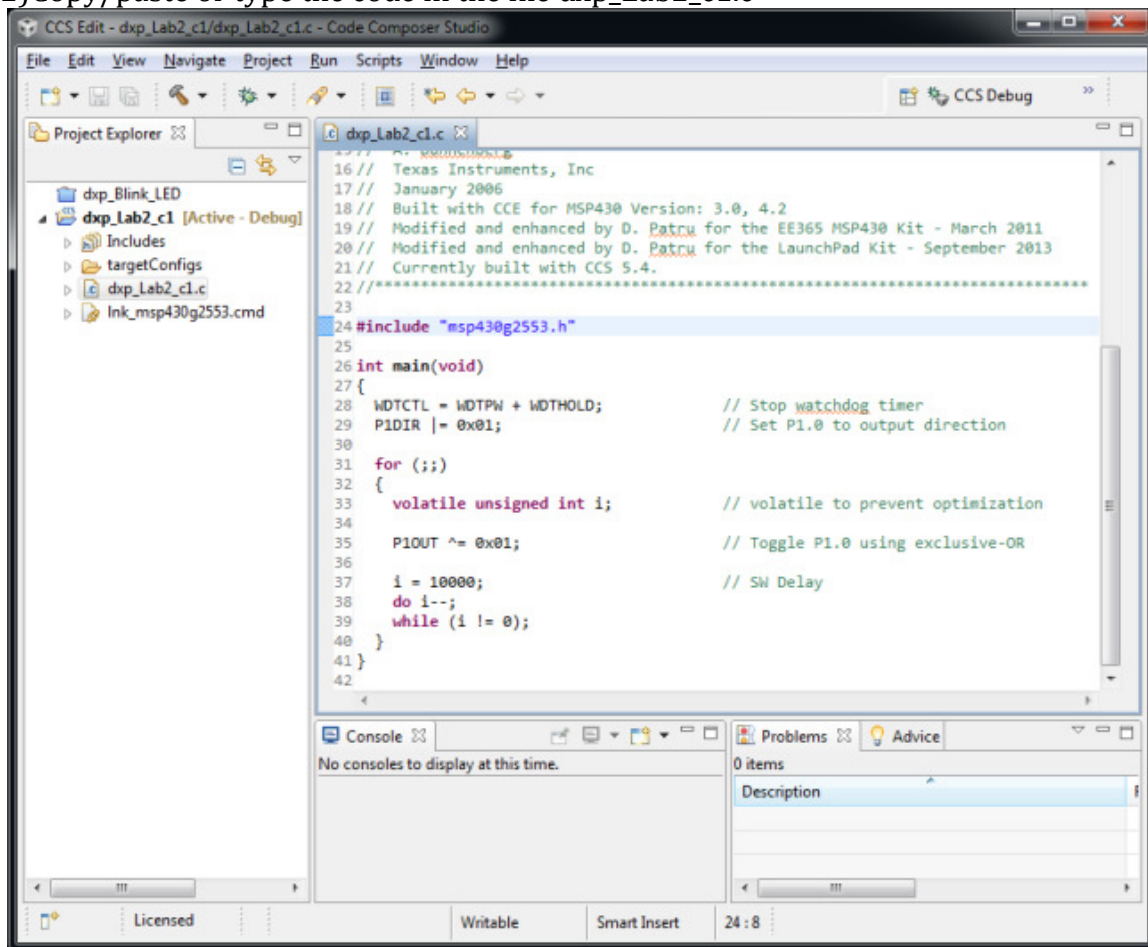
- 20) Well, at this point we are missing probably the most important file: one that contains the source code. Click on the project name in the left pane again. Now go to File > New > Source File and enter the name `fmlxxxx_Lab2_c1.c` for it. Leave everything else default.



- 21) Your window should look as the one below by now:



22) Copy/paste or type the code in the file dxp_Lab2_c1.c



```
16 // Texas Instruments, Inc
17 // January 2006
18 // Built with CCE for MSP430 Version: 3.0, 4.2
19 // Modified and enhanced by D. Patru for the EE365 MSP430 Kit - March 2011
20 // Modified and enhanced by D. Patru for the LaunchPad Kit - September 2013
21 // Currently built with CCS 5.4.
22 //*****
23
24 #include "msp430g2553.h"
25
26 int main(void)
27 {
28     WDTCTL = WDTPW + WDTHOLD;           // Stop watchdog timer
29     P1DIR |= 0x01;                      // Set P1.0 to output direction
30
31     for (;;)
32     {
33         volatile unsigned int i;        // volatile to prevent optimization
34
35         P1OUT ^= 0x01;                  // Toggle P1.0 using exclusive-OR
36
37         i = 10000;                      // SW Delay
38         do i--;
39         while (i != 0);
40     }
41 }
42
```

23) Aside from the commented header, your code should look like the one above.

24) Next we will take a closer look at the source file.

25) Notice that it has a well-commented header section. Add a line after line 19 with your name in it. This commented header section is a good example of how you should introduce any of your future source files.

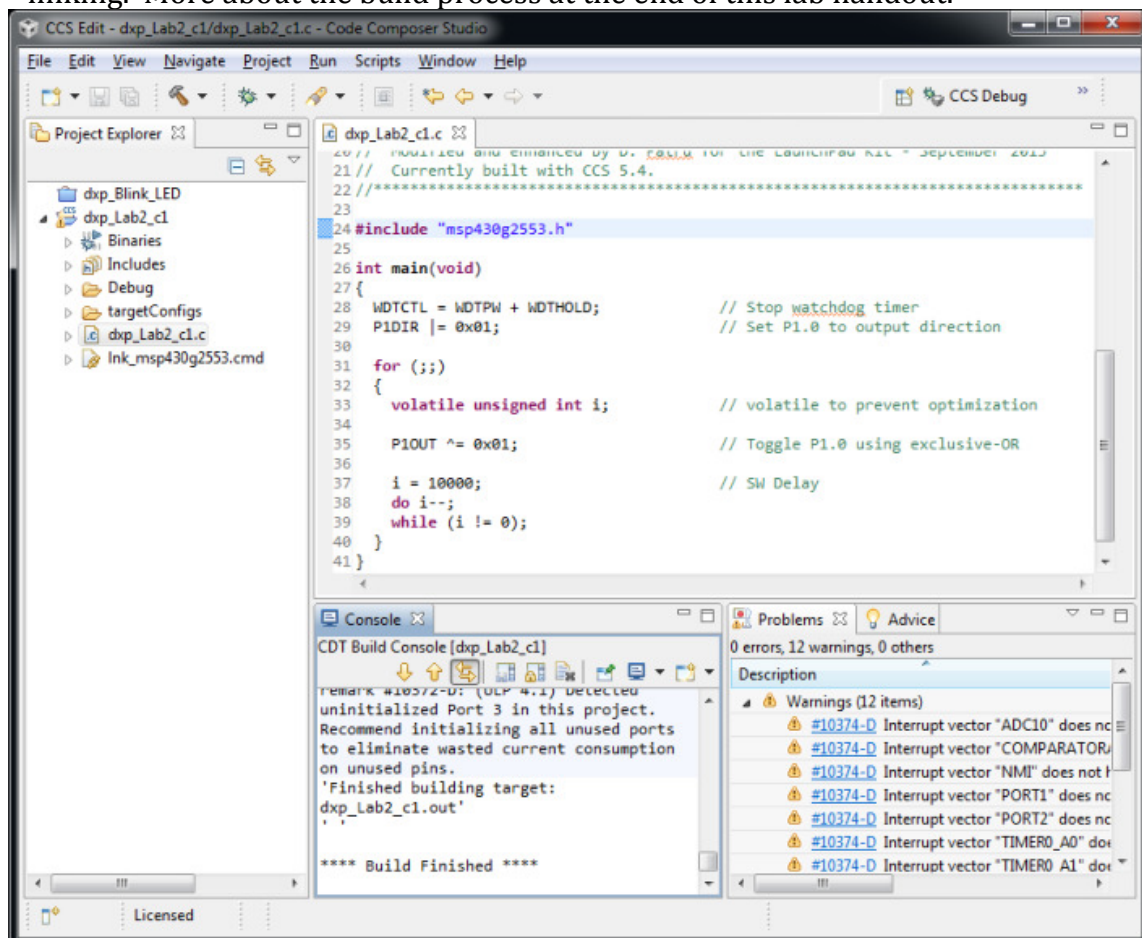
26) Save and scroll down. This is C code for the MSP430. I am sure it generally looks familiar, but it contains some constructs specific to the use of C for microcontrollers, which I will explain shortly.

27) The code contains a single function main. In line 28, the watchdog timer is stopped or disabled. Remember: if the watchdog timer is enabled, which is the default condition for MSP430, the microcontroller has to clear it before it rolls over, i.e. the WDT period expires, and triggers a system wide reset. The programs you will generate in this lab are not life-critical, so we will not use the WDT. Therefore, always remember to stop the WDT at the beginning of your code.

28) In line 29, the code sets the input/output port pin P1.0 to the output direction using the mask 0x01 in hex or 00000001B in binary. P1DIR is the name of the direction configuration register associated with port 1. The content of P1DIR is inclusive-OR-ed with the mask. An inclusive OR with 0 leaves the value unchanged, whereas an inclusive OR with 1 changes the value to 1. The inclusive OR operation symbol is: “

|= ". We will learn more about digital input/output ports, and their configuration later on in the course.

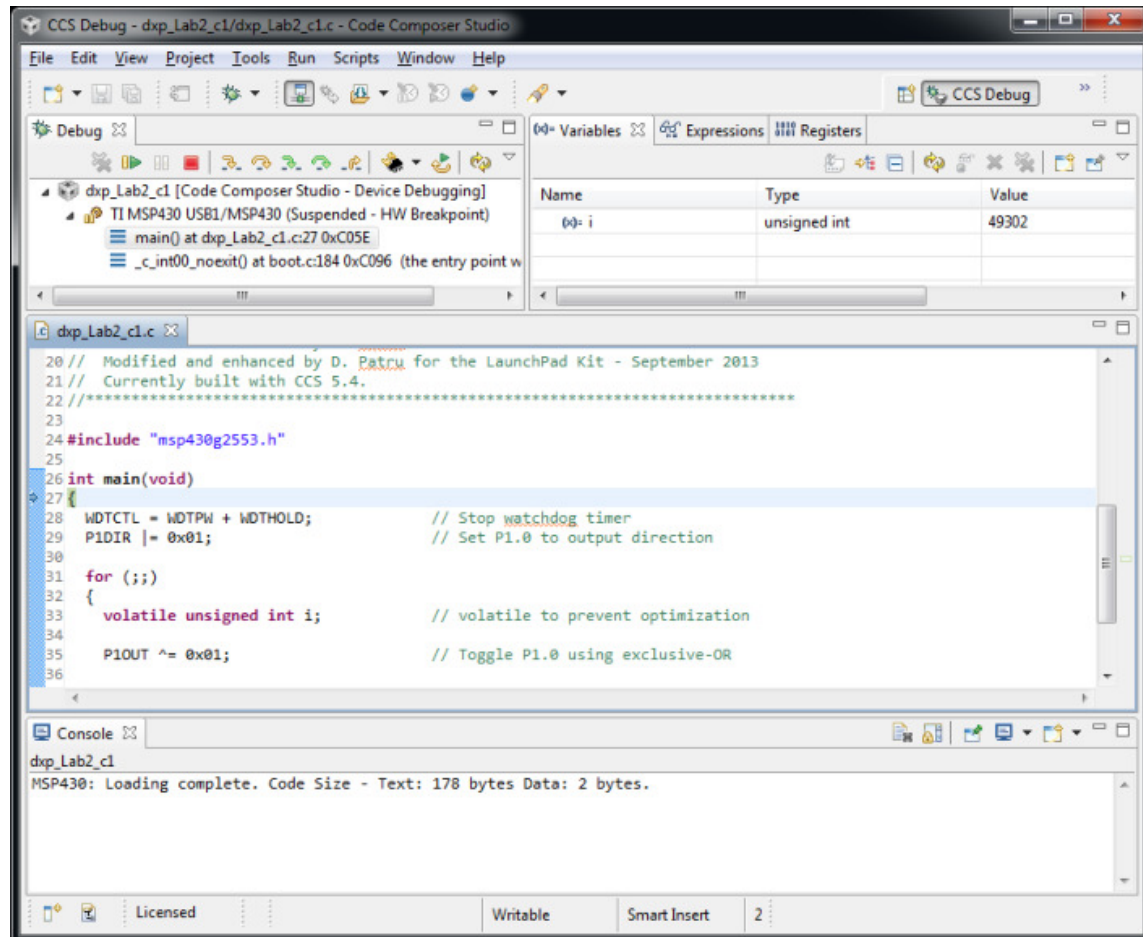
- 29) In line 33, the variable `i` is declared. The type `volatile` is specific for microcontrollers, and it indicates to the compiler that it is a variable that may appear to change "spontaneously", with no direct action by the user program. In this context, it prevents optimization by the compiler. More about this data type later.
- 30) In line 35, the value of `P1.0` is toggled by XORing it with the mask `0x01`. This output pin drives LED1 on your board. Toggling the value of the output pin will turn on and off the LED. The period or frequency of the toggle is set by a software delay loop, which is inserted in the iteration of the infinite 'for' loop.
- 31) At this point we are ready to build, i.e. compile and link the program. Select **Project > Build All** or click the hammer icon.
- 32) If everything goes well, you should see the following comments in the Console pane and some 12 warnings related to missing interrupt vector handlers. Ignore these warnings for now. However, if you see red comments, indicating errors during the build process, you need to go back to your code and fix them. These are also listed separately in the Problems pane. If you scroll up in the Console pane, you can follow all steps of the building process, which essentially include compilation and linking. More about the build process at the end of this lab handout.



- 33) So far we have worked in the CCS Edit Perspective. To download, run, and debug the program, you need to select **Run > Debug** or click the bug icon. CCS opens

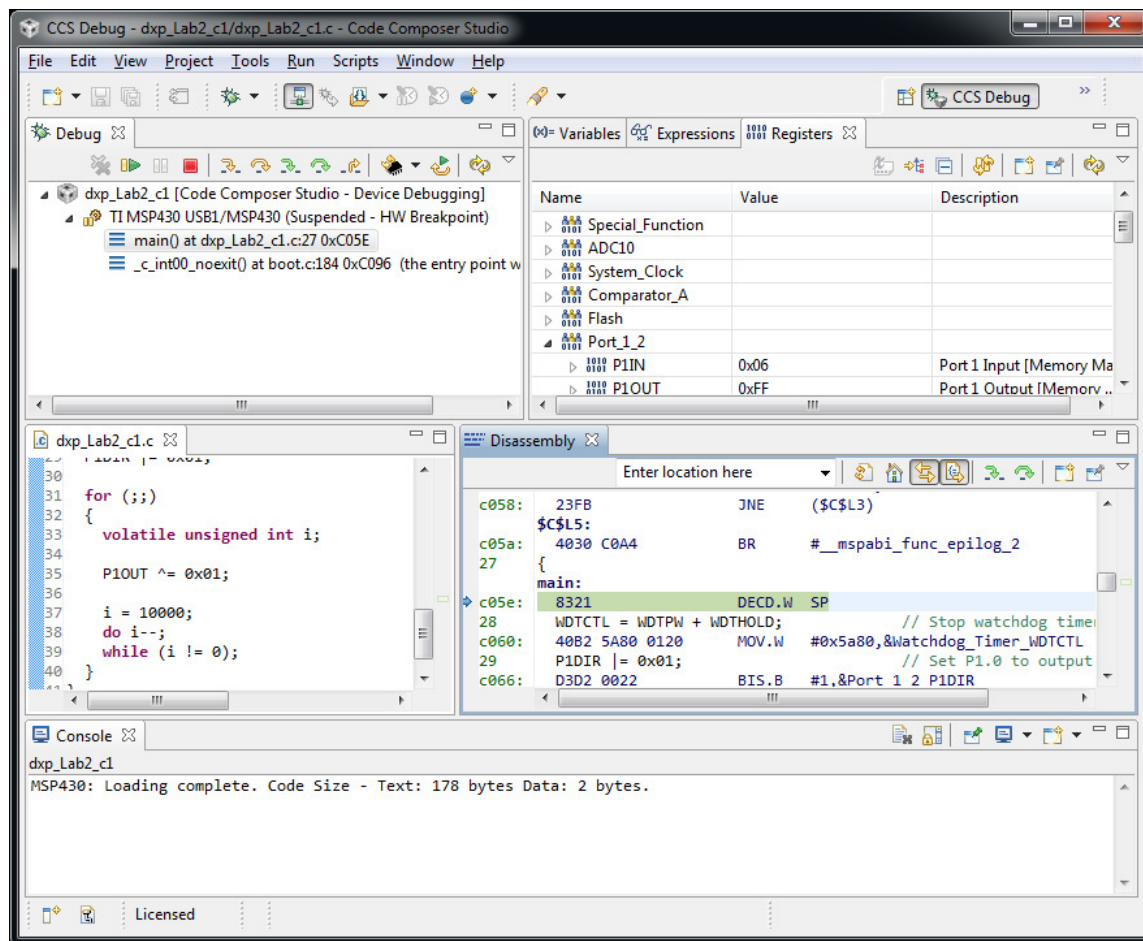
automatically the CCS Debug Perspective. In the upper RHS corner you can switch perspectives at any time. Before you hit debug make sure the board is connected.

- 34) During the switch to the Debug Perspective you may be asked to look at or use the ULP advisor. Click Proceed for now. Once in Debug mode your window will look similar to this:

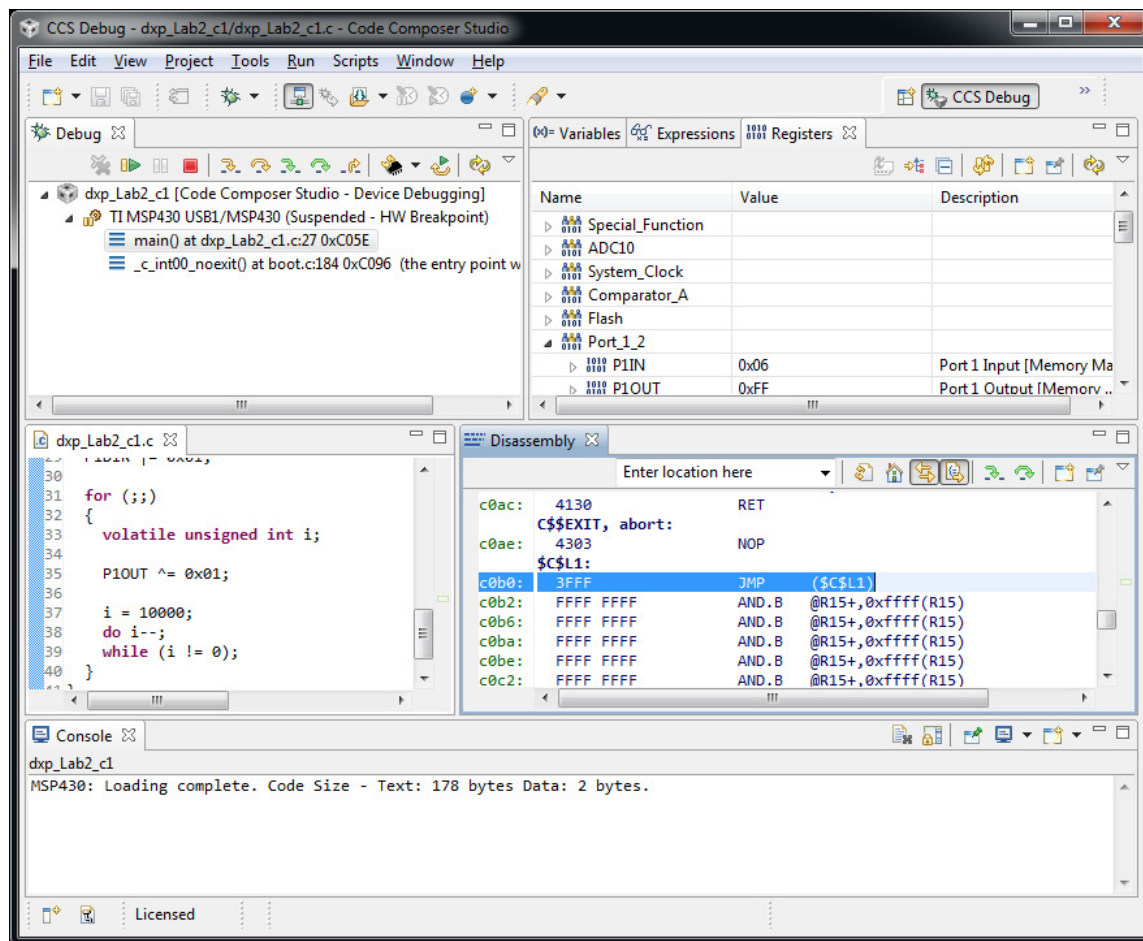


- 35) In this perspective new panes are opened. The Debug pane contains information about the debug session, similar to the C/C++ Projects pane in the C/C++ perspective. The central pane contains the code ready to be run. Notice a blue arrow on the LHS of line 27, pointing to the current next code line ready to be executed. The Console pane is used again to echo the steps the debugger goes through as a result of a specific user command. The panes on the upper RHS will contain debugging information.

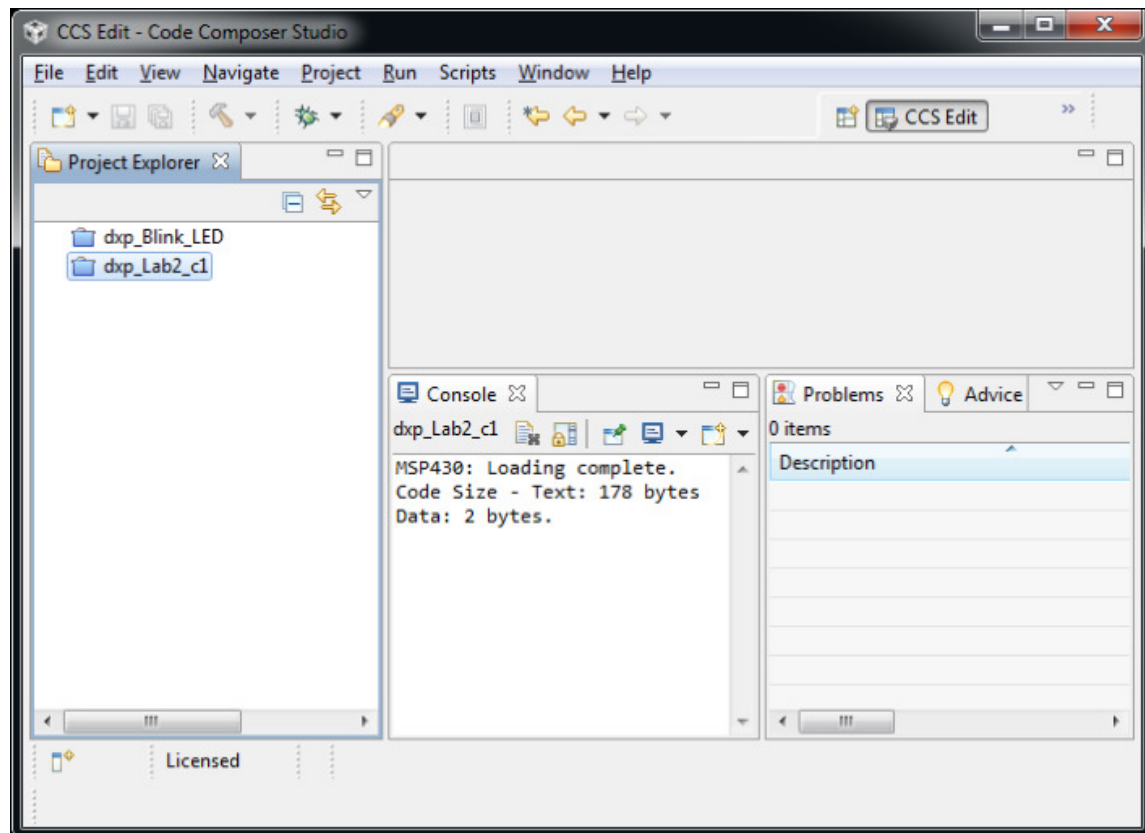
- 36) Before we run the program we need to do a few more preparatory steps. First, click the Registers tab. Second, select View > Disassembly. A new pane opens to the left of dxp_Lab2_c1.c. In this, the green left column contains the address in memory, in the center we see the contents of memory locations, and on the RHS column the equivalent assembly code. The latter was “reverse engineered” by the debugger, from the machine code generated by the builder (compiler and linker). The black arrow points to the first instruction to be executed, so that means that the location of the first instruction to be executed is 0xc05e. There is some auto-initialization cde further up, but we’ll ignore that for now.



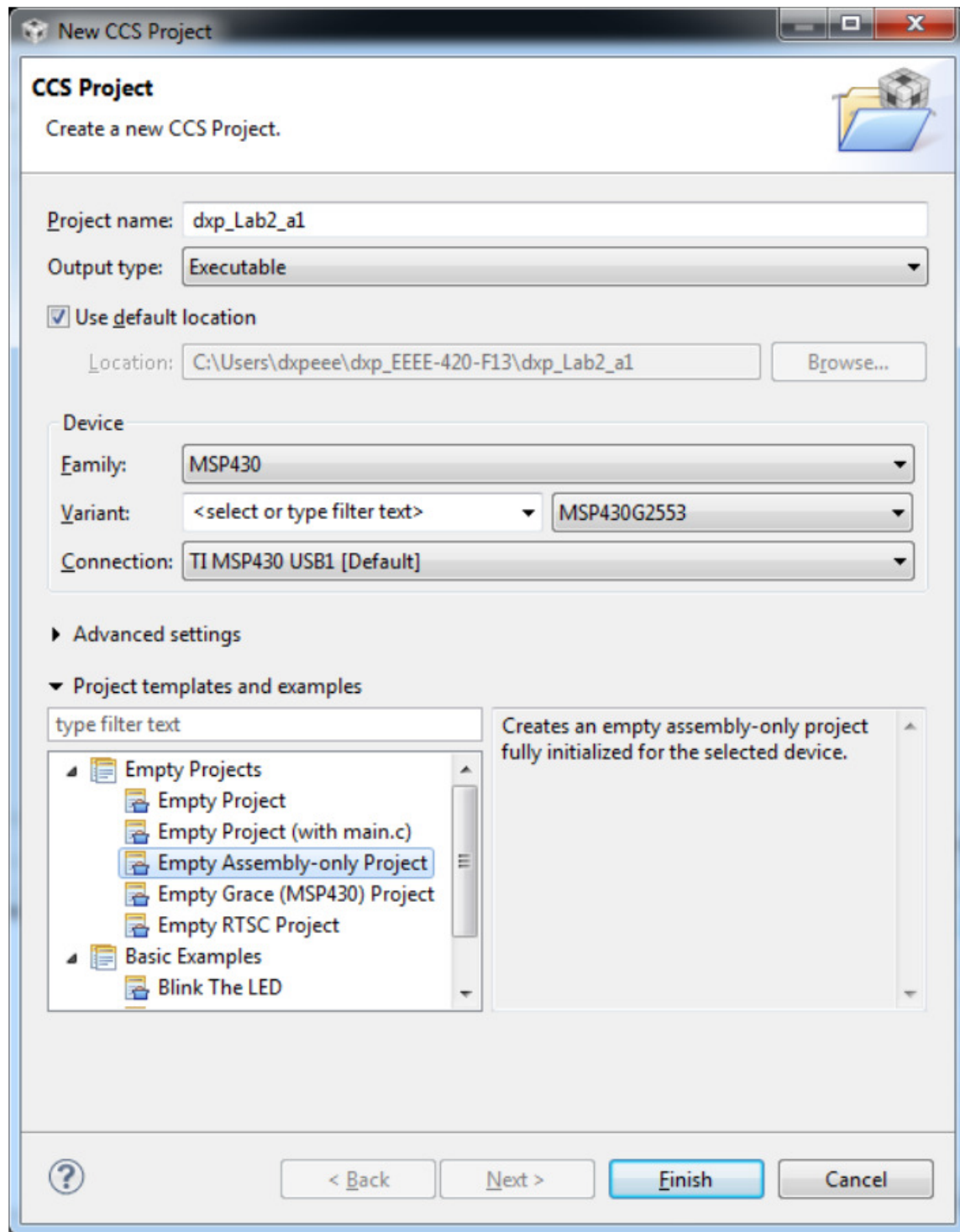
37) Now, scroll down and you will arrive at the end of the program at location 0xc0b0. This means that this program is 0x52 bytes long, or 82 in decimal. We will keep this in mind for later.



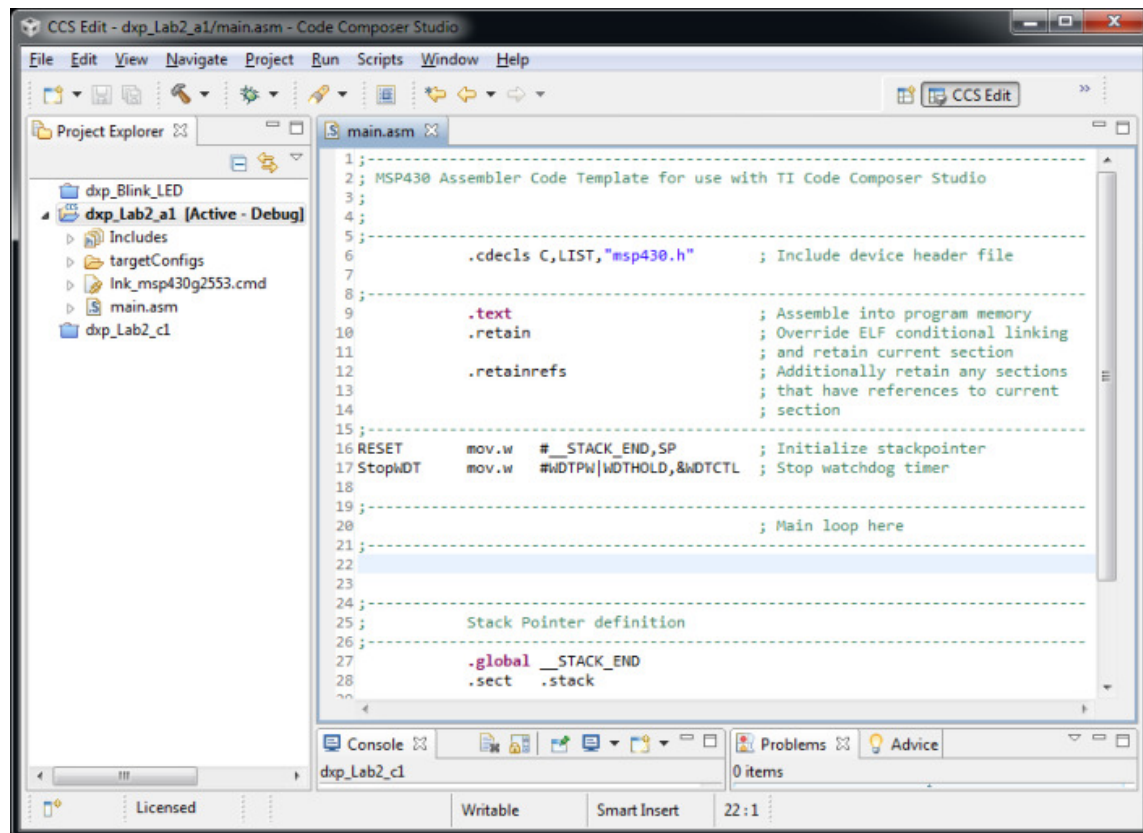
- 38) The disassembly view is very useful during the debugging process because it shows exactly what has been written to the controller's memory.
- 39) The program can be run step-by-step, up to a breakpoint, or launched all together. Because it has been a long way up to this point, we will let it run and enjoy the blinking of the LED. Click on the green play button to let it run free.
- 40) At this point, the LED should blink at a frequency that the human eye can follow. We will explore the features of the Debug perspective in the next project. Now, click on the red stop button to Terminate running the program.
- 41) Notice that the Debug session closes automatically, but the LED keeps blinking. This means that CCS is not in contact with the uC anymore, but the latter is still running the program that has been written to its flash memory. In fact, because the program was burned into the non-volatile flash memory, even if you disconnect and reconnect the power, or reset the microcontroller, it will still run the same program.
- 42) Right click on the project name in the Projects pane, and close it. Your display should now look similar to this:



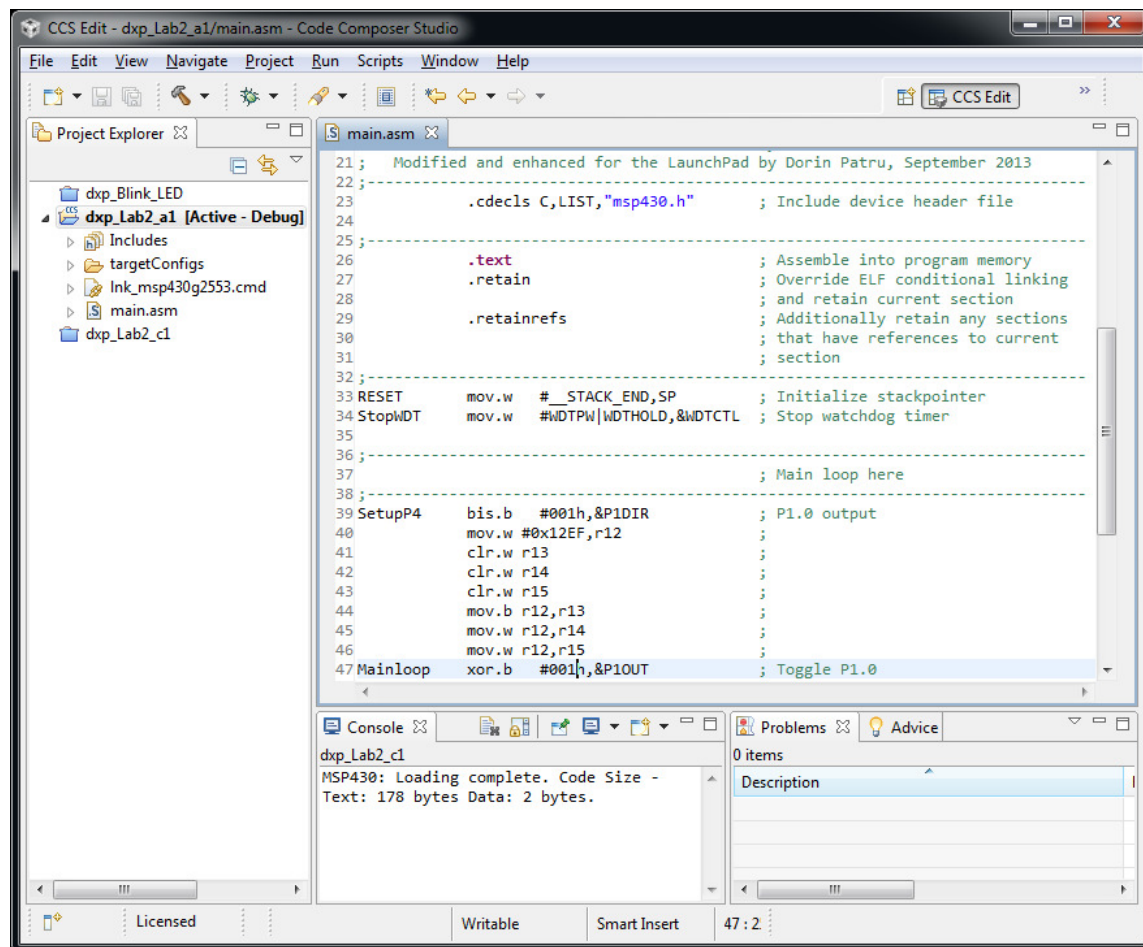
- 43) **Part 2.** Become familiar with the CCE IDE “build” process and debugger, and with assembly code for the MSP430.
- 44) At this point, we will create a second project, in which we will import assembly code that turns on and off the LED. To create a new project, follow the same steps from part 1, and name the new project dxp_Lab2_a1. Make sure to **select Empty Assembly-only Project** as shown below.



45) Once you create the project, CCS will automatically create for you an assembly template source code file called main.asm.

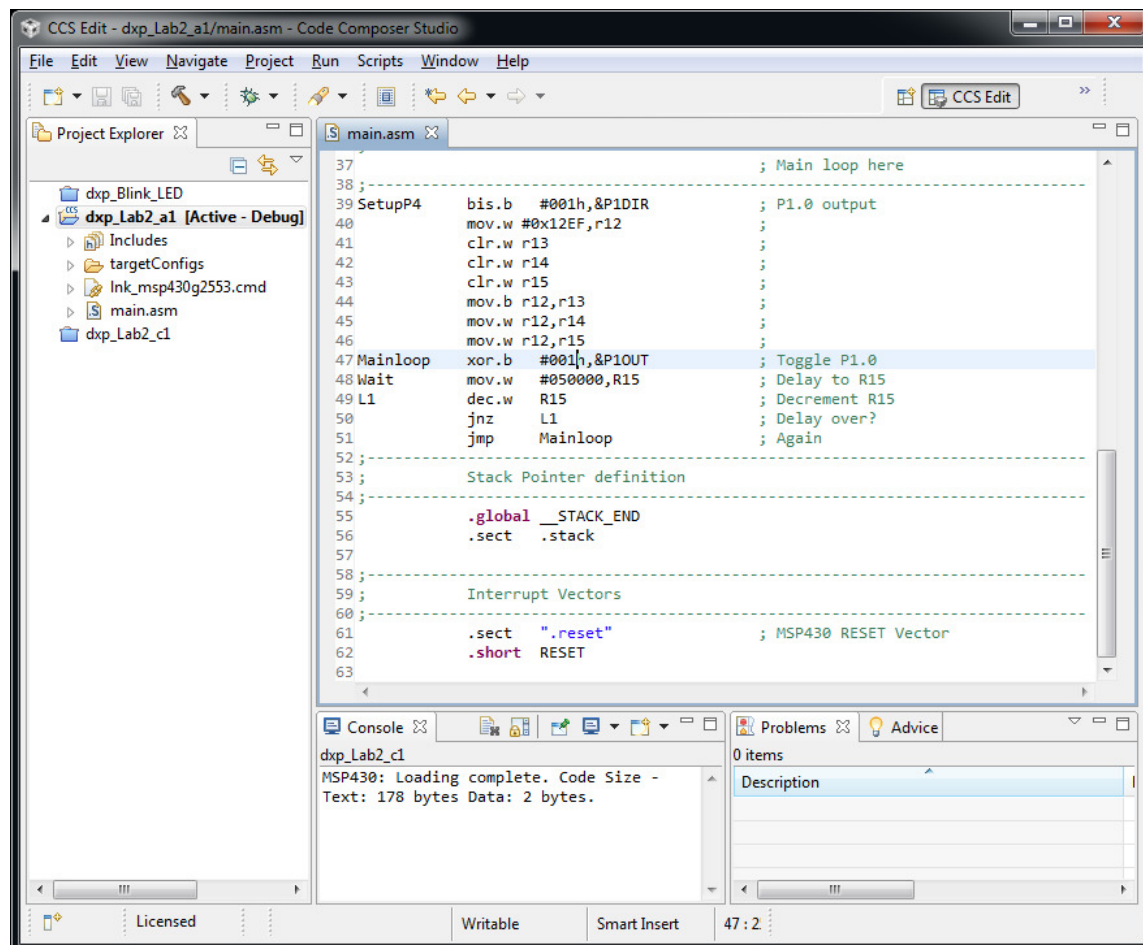


46) Copy/paste or type the main loop code as shown below. Make sure you copy/paste all the way down to the last line (line 63 - .short RESET). For completeness copy also the header. Notice that the comments in the assembly source file follow a semicolon, and not two forward slashes. The first line of actual code is line 23.



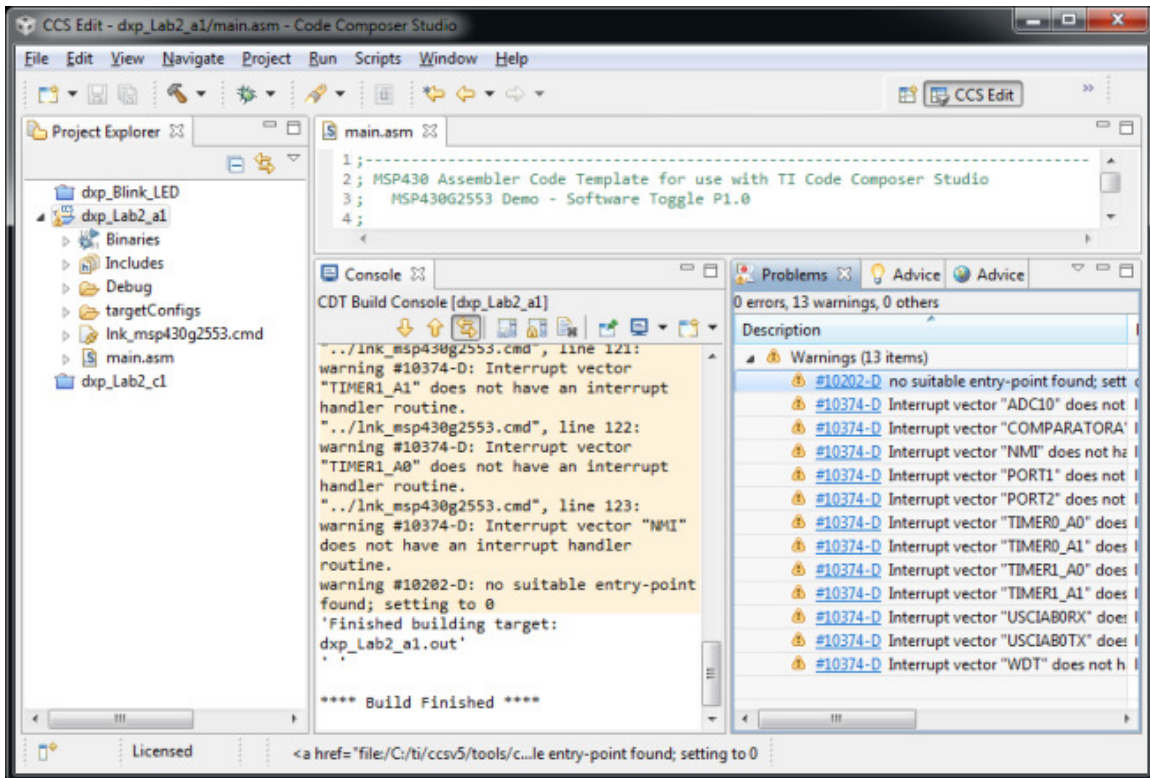
- 47) As you can see, assembly language code is more rigid than C. The code is divided into sections. In CCE the program is listed in the section .text. This is called an assembler directive. We will learn about other sections and directives later.
- 48) Note: Other assemblers may have different directives and conventions. A good and immediate example is the IAR assembler, for which most assembly code in your textbook has been written. IAR is a third party that provides assemblers and compilers for a very large number of processors and microcontrollers, including the MSP430. As you can see, the generation of assembly code is not standardized, which is a drawback for the user/programmer. Porting the code from one IDE to another requires slight modifications.
- 49) As discussed in the lecture, a line of code usually consists of four parts: a label, an operation, operands, and comments. With reference to line 33, the label is RESET, the operation is indicated by the instruction mnemonic mov.w, the one source operand is the immediate value #300h, the destination operand is the Stack Pointer (SP) register, followed by comments after the semicolon. The label is optional. It is most often used as a target for jump/branch instructions. For example, the jmp on line 51 will jump back to the instruction on line 47. The assembler will replace the label Mainloop with the address of xor.b. The instruction mnemonic is essentially an abbreviated code name, meant to be easily understood by the programmer and reader of the code.

- 50) Let us now analyze the code line by line.
- 51) In line 33, the stack pointer is initialized. Details about its use are given in the lectures.
- 52) In line 34, the watchdog timer is stopped or disabled, for reasons explained before and in the lecture.
- 53) In line 39, port pin P1.0 is set to output direction. The uC UG contains on pages 62 and 63 the list of all MSP430 instructions, core and emulated, in alphabetical order. In this list you'll find the bis.b instruction, which effectively implements the bitwise OR function. It is called bis and not or, because the architects of the instruction set considered it easier to remember it in the context it is usually used: to set bits. Its counterpart instruction is bic.b, which is used to clear bits, using a bitwise AND.
- 54) In line 47, the Mainloop starts with the toggling of P1.0 through an XOR with the mask 001h. To toggle with a frequency the human eye can follow, the toggling is delayed using a software-based timer/counter. This is implemented as follows:
- 55) In line 48, register R15, which is a general data register in the register file, is loaded with the value 50000 in decimal.
- 56) Then, in line 49 R15 is decremented, i.e. 1 is subtracted from its current content.
- 57) In line 50, its current content is compared to 0. If it is not 0, the execution jumps back to line 49. This two-instruction sequence will therefore be executed 50,000 times. Once R15 has been decremented down to 0, the execution of the program will continue past the jnz instruction. You can read about its operation in the instruction set list.



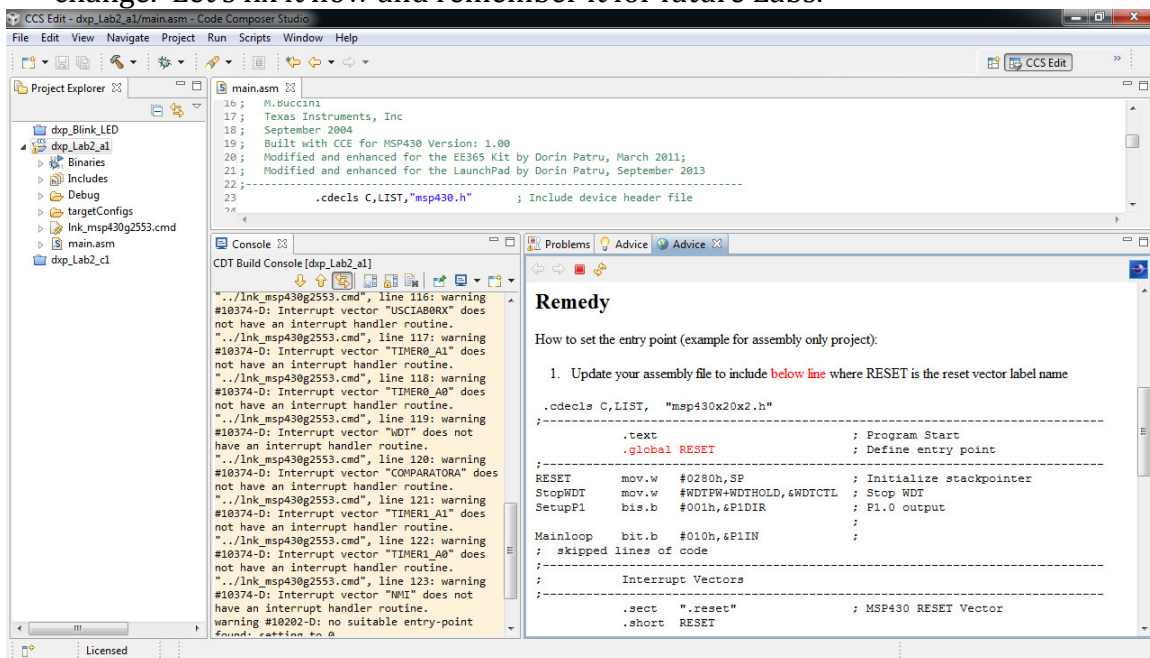
58) The above three lines of code implement a software based timer/counter. We will see later how one can use the hardware-based timer/counter to achieve the same delay effect. You may want to answer the following question: How can the toggle period be increased or decreased?

59) Finally, the `jmp` in line 51 closes the Mainloop. While the C and assembly source codes seem similar in size, we will see shortly that the size of the generated machine code is very different.

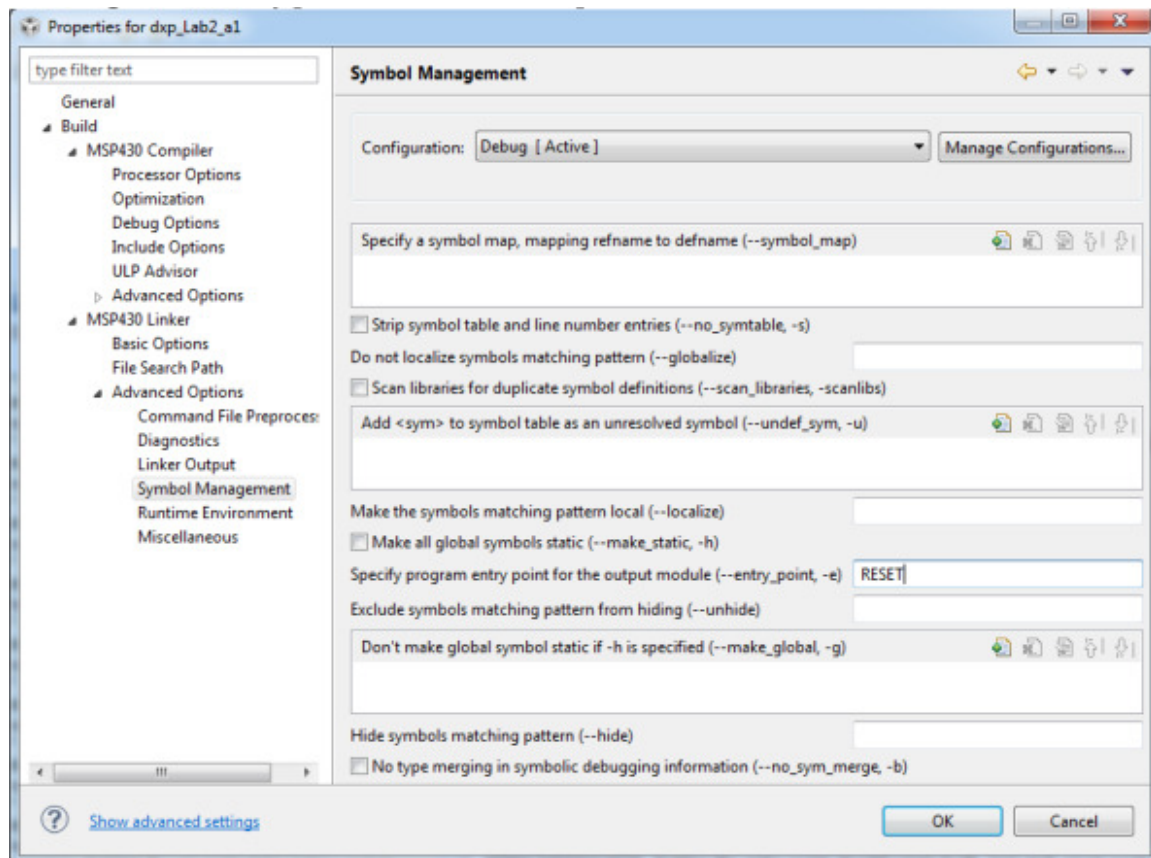


60) The first warning as highlighted above refers to “no suitable entry point – into the program – found”. Let’s click on the blue warning message number.

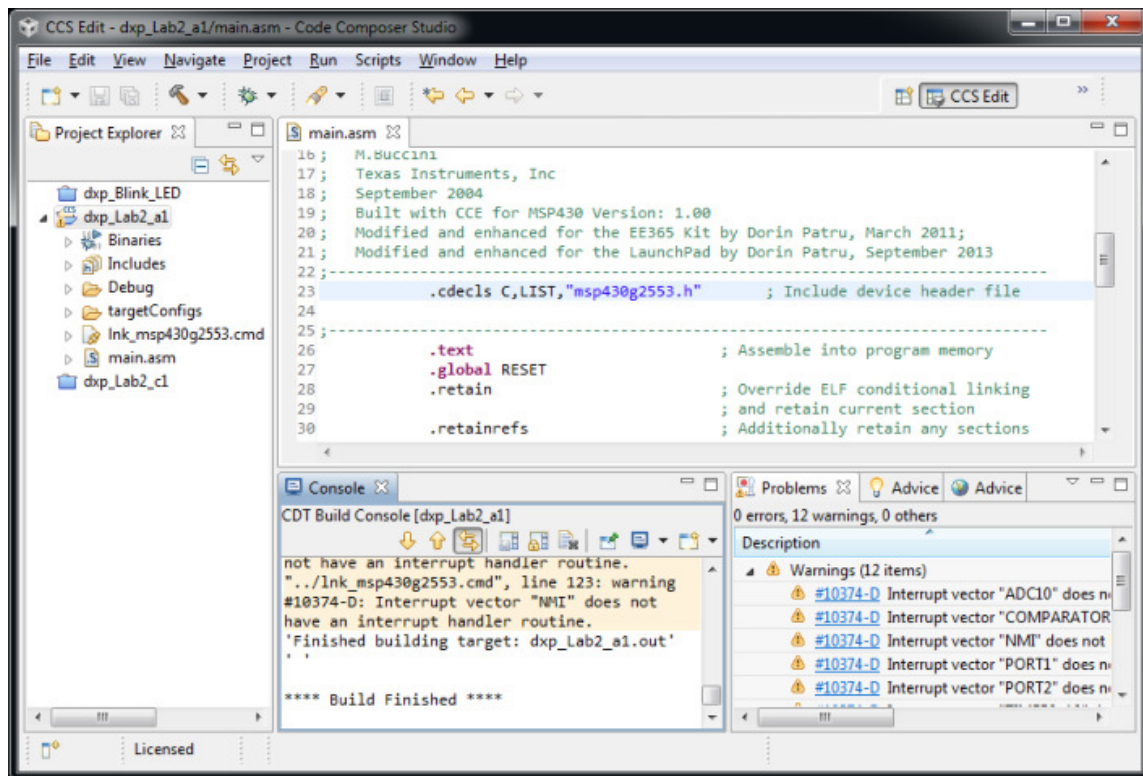
61) You can scroll through the explanation. In an assembly-only project, we need to define the entry point, so that the linker knows where to have the reset vector pointing to (similar to "main()" in a C program). We should make the recommended change. Let’s fix it now and remember it for future Labs.



62) Now, right-click on the project name in the LHS pane and select ... Collapse MSP430 Linker and then Advanced Options. Select now Symbol Management. Type RESET with capital letters in the box as shown below. This entry point name ("RESET") needs to match the label used in your code. Essentially, the reset vector (at location 0xFFFFE in the vector table), will contain the address of where the first instruction (mov.w #0280h, SP) is located. When the MSP430 is reset, the PC will be loaded with that address, so that the instruction word can be fetched.



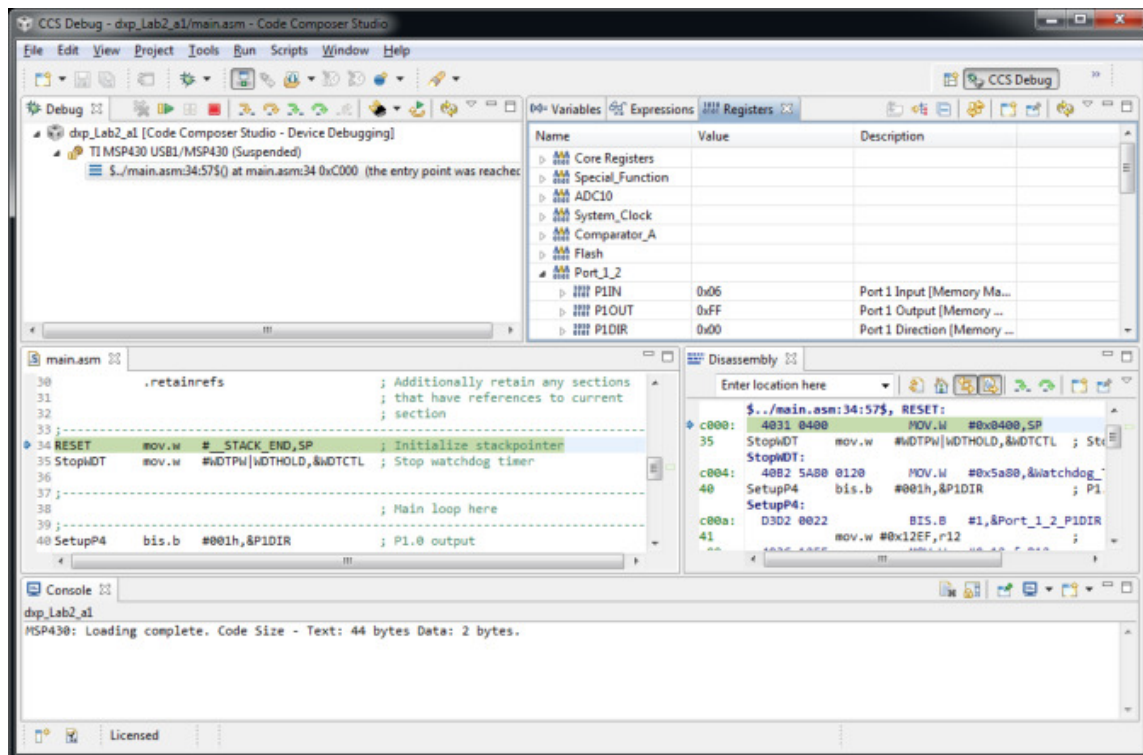
63) Before proceeding, let's change the include file name to the one specifically written for our device: msp430g2553.h. Make the change and re-build the project, this time by going to Project > Clean. This will erase all build, link, and debug related files and perform a clean build. In large projects, if we don't clean, the assembler or compiler will only rebuild files that have been changed/saved since the last build. The linker usually has to do the whole work again. The results should look as in the capture below.



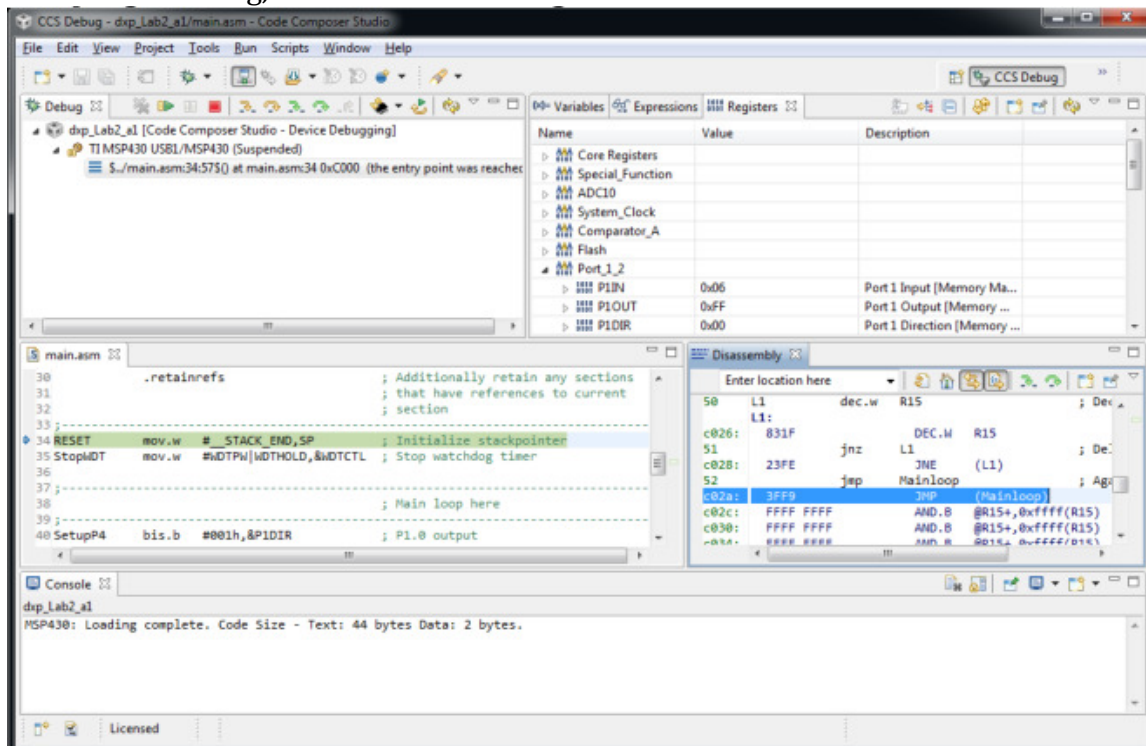
64) Well so far so good. It seems we got rid of the warning related to the entry point.

65) Note: if the number of comments in the console is too high, and scrolling becomes difficult, you can clear the console by clicking on the second icon from the left in the Console pane header.

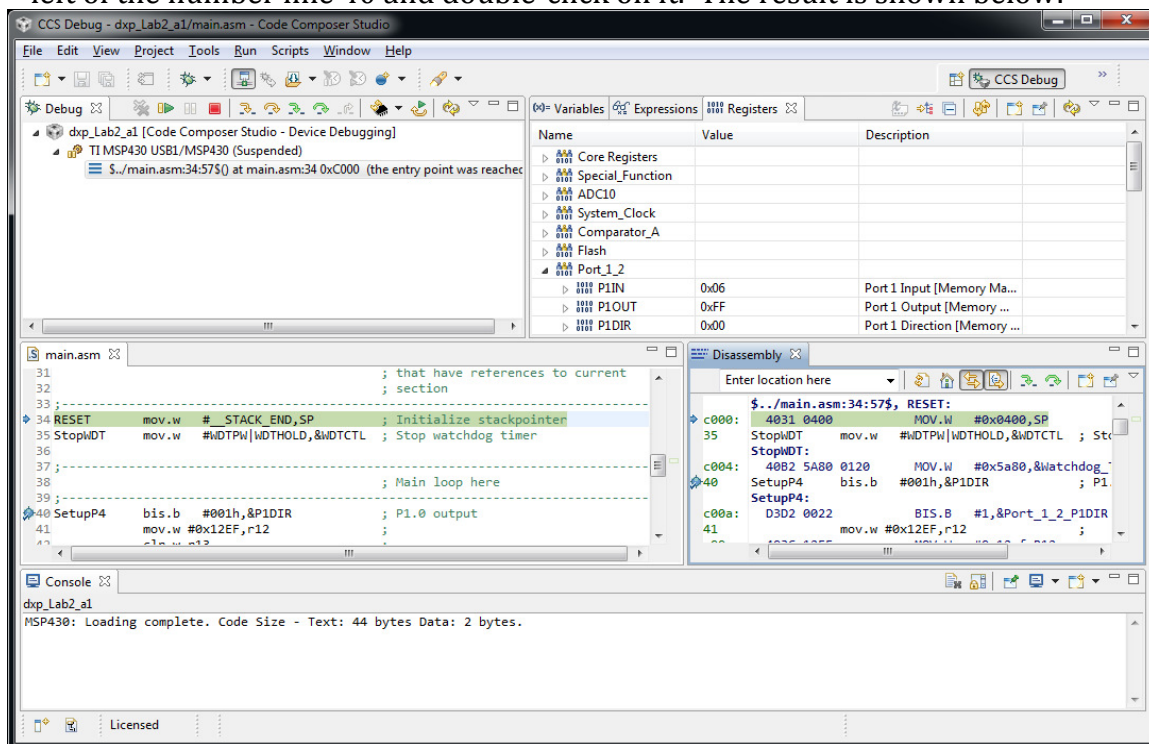
66) As in part 1, click the bug icon to start the Debug session and open the Debug Perspective.



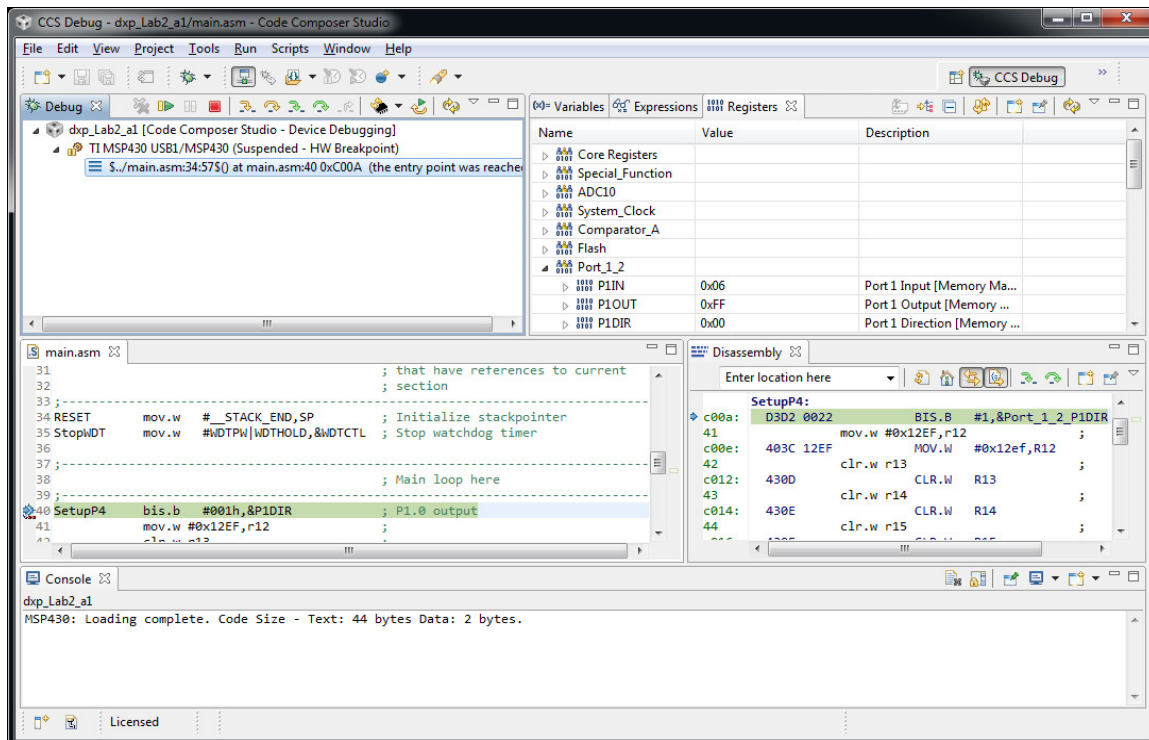
67) The debug perspective displays the same panes you used last. Note that the blue arrow indicates the first (next) instruction to be executed in the source code pane. A similar arrow indicates the same in the disassembly pane. The location of the first instruction is now 0xc000. If you scroll down, you'll see that the last byte of the last instruction is stored in location 0xc02a. Thus, the program is 0x2A instructions long, or 42 in decimal.



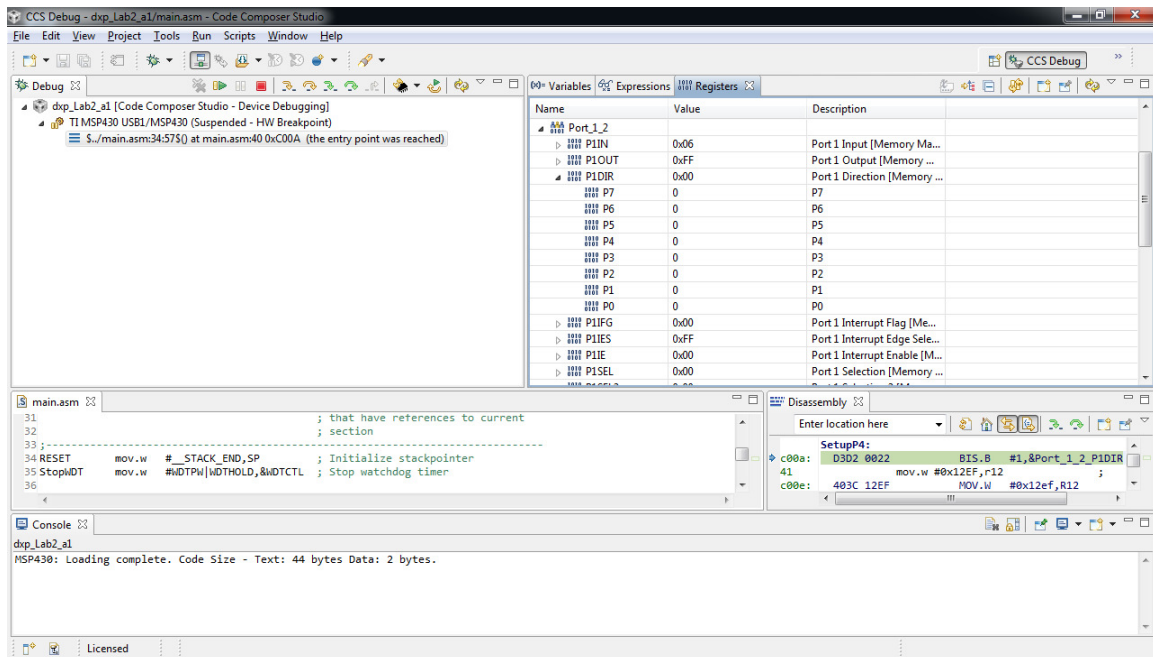
- 68) The machine code generated by the compiler from C code was 82 bytes long, not counting the initialization part. If we count the latter, which started at location 0xc00, then the length of the C program is 0xb0 or 176 bytes. Thus, at least from a size point of view, the machine code generated in this example from assembly source code is about four times smaller, i.e. more efficient, than the machine code generated from C source code. While the relative machine code size varies for different programs, and also between different programmers and programming styles, the balance is always in favor of assembly. The reduction in machine code size has further implications: a smaller code size will require a smaller program memory. In turn, a smaller program memory allows the selection of a smaller and cheaper microcontroller family member, which eventually will also consume less power. These are highly desirable design goals in embedded systems.
- 69) On the other hand, writing C code is faster and in general easier to port. Throughout this lab we will compare the two programming styles, and highlight the advantages and disadvantages of one and the other approach. Finally, we will show that assembly can be mixed with C code, producing the best results of both worlds.
- 70) At this point we could launch the program to run free. Instead, let's learn a little bit more about the use and capabilities of the debugger. Place the mouse cursor to the left of the number line 40 and double-click on it. The result is shown below:



- 71) You have just set a breakpoint, which is also visible in the Disassembly window. If we now let the program run, it will run up to this breakpoint only. Click the green play button and you'll see that only two instructions are executed, i.e. up to the breakpoint you have set on line 40.

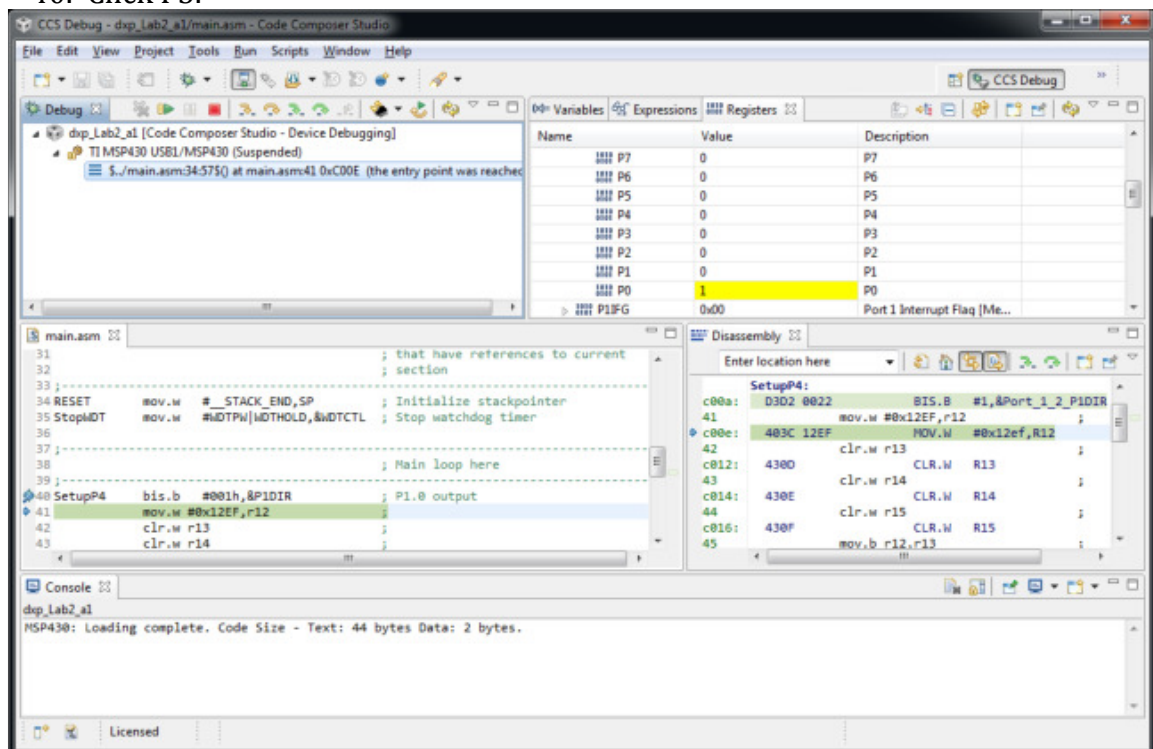


- 72) We can also execute one instruction only, without setting a breakpoint. You can do this in three ways:
- 73) Select Run > Step Into from the text menus, or
- 74) Use the command icon, which is the first from the left in the second set of icons in the Debug pane header, or
- 75) Click F5.
- 76) Because the execution of the next instruction will eventually change the content of register P1DIR, let's check it before and after the execution of this instruction.
- 77) To view the content of register P1DIR, select View > Registers from the text menus. A new pane called Registers is opened in upper right corner. You may have it already open from last time. Click on the + sign to the left of Port_1_2. A list of all registers associated with these two digital input / output ports drops down. Towards the bottom you'll see the value 0x00 in register P1DIR.



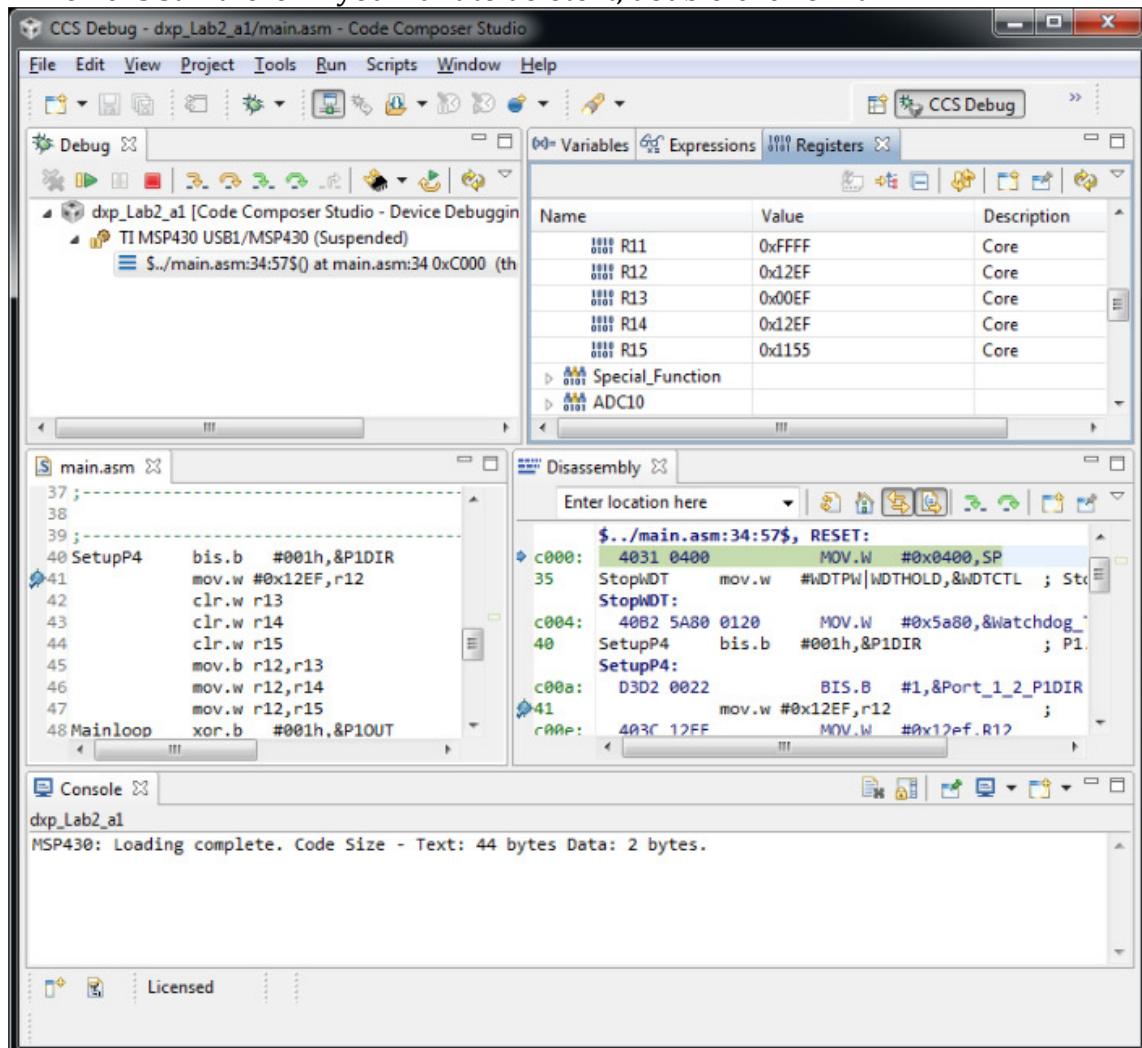
78) If you want to see it in binary format, i.e. bit by bit, you can click on the + sign to its left.

79) Now we are ready to step into, and see the effects of instruction bis.b on line 40. Click F5.



80) You notice that the program has advanced to the next instruction on line 41, and P0 of P1DIR, highlighted in yellow, has changed from 0 to 1. Registers or memory locations affected during the last run of the program are highlighted in yellow.

- 81) At this point, we can take a break and enjoy the blinking of the LED, so simply click Run and the LED should continuously turn on and off.
- 82) Stop the debug session by clicking on the red stop button.
- 83) **Part 3:** Learn what the 'mov' and 'clr' instructions do.
- 84) At this point, we will concentrate on the instruction sequence on lines 41 through 47. You can find out about the effect of each of these instructions in the instruction set list. Insert this instruction sequence in the source file opened in the debug perspective, after you Target > Reset CPU.
- 85) Re-start a debug session.
- 86) First, set a breakpoint on line 41 and run the program to it. The old breakpoint on line 40 is still there. If you want to delete it, double-click on it.

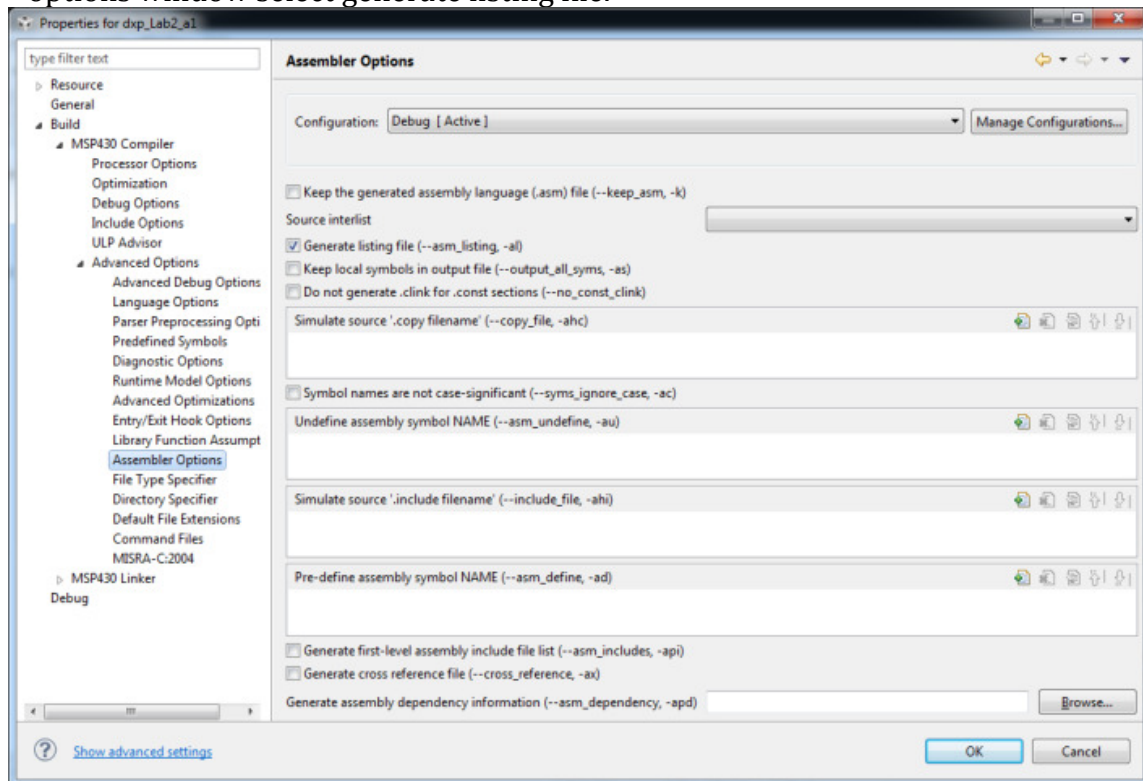


- 87) Now, "single-step" through the additional assembly language statements and record the contents of the registers indicated in Table 1 (found on the last page of this document) as each instruction is executed. Include this table in your report, which in this case may and can exceed the usual one page.
- 88) To see the content of these registers, scroll all the way up in the already open Registers pane, and select the Core Registers drop-down list.
- 89) Terminate the debug session by clicking on the red stop button.

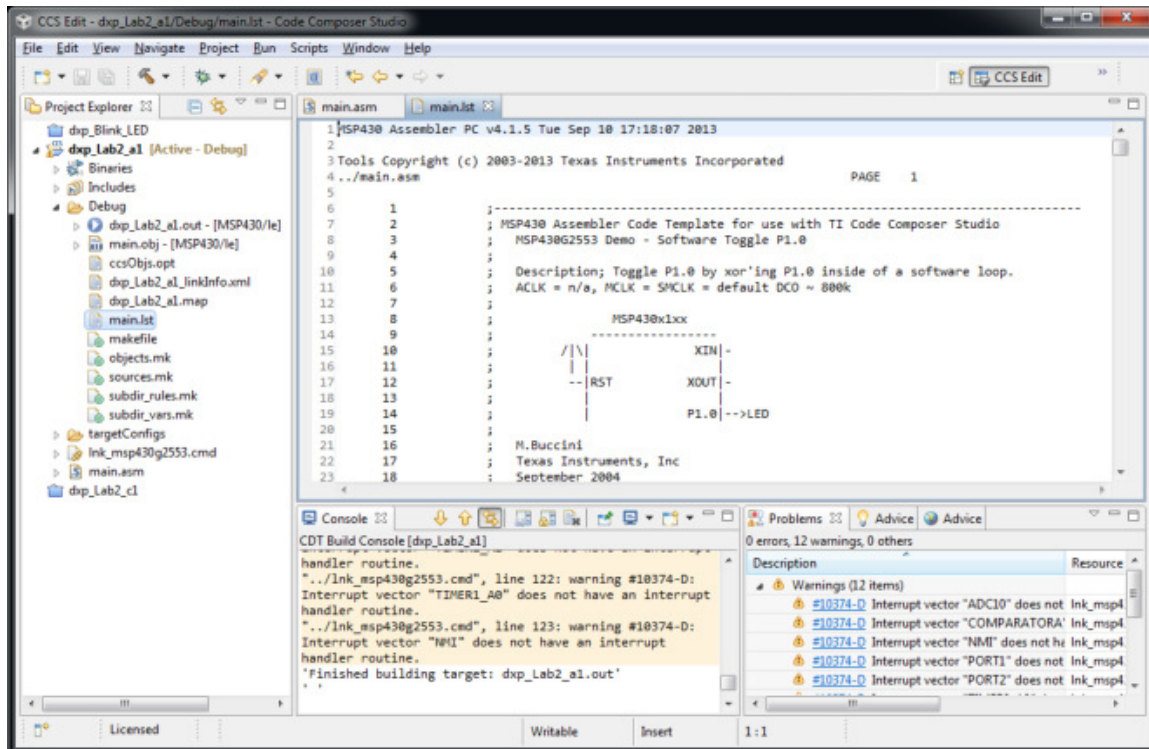
90) **Part 4.** Listing files, map files and disassembly

91) It is possible to create something referred to as a “listing” file when you assemble and link your program. This will actually indicate the binary object code (usually in hexadecimal format) and relevant address locations in memory that result from the assembly and linking process.

92) Assuming you are now in the CCS Edit Perspective, select the active project name. You can produce a listing file via Project > Properties, and then Build > MSP430 Compiler > Advanced Options > Assembler Options. Now in the assembler options window select generate listing file.

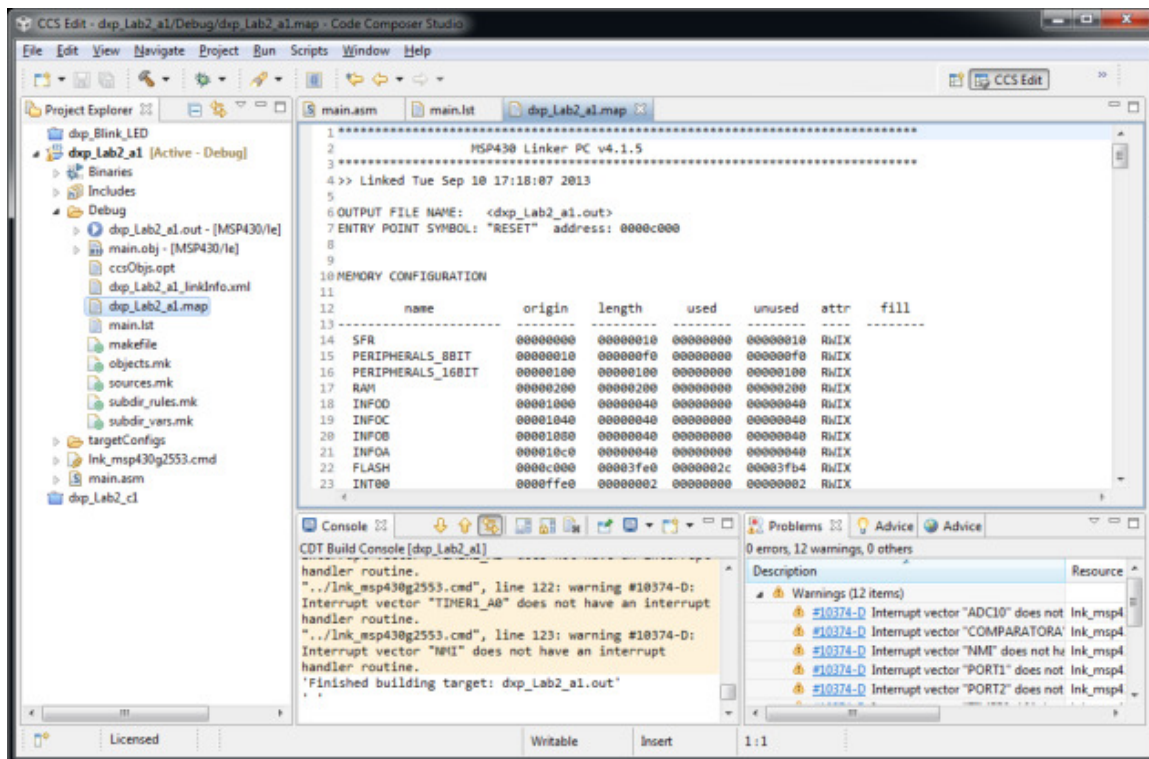


93) Click Apply and OK. Now rebuild the project, and then open the file fmlxxxx_Lab2_a1.lst (or main.lst), as shown below. The listing file contains header information, and towards the end of page 13 (yes it is formatted for printing), the code and corresponding machine code.



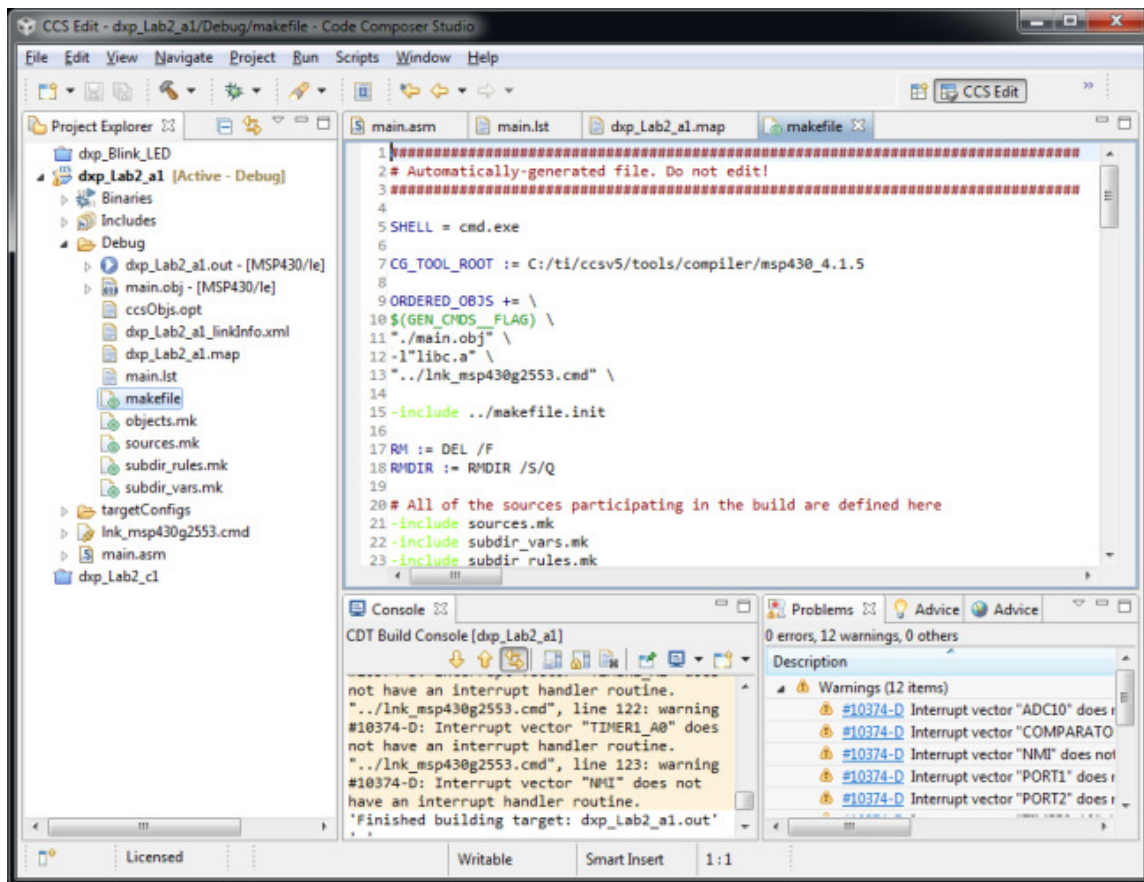
94) You can also view/find this information in the Disassembly window of the Debug Perspective.

95) Another useful file is the map file, i.e. fmlxxxx_Lab2_a1.map. Compare the contents of these two files with the information displayed in the Disassembly window in the CCE IDE, Debug perspective. Note what a “disassembler” does and why it would be useful information in the debugging process. The notes you take in this portion of the lab should be included in your lab report.



96) Appendix:

- 97) A behind the scenes “gentle” introduction into how an assembly level application is “built”, or “What is the make file and how is the executable (to be downloaded) produced?” (Hint: make is a program designed to run other programs.)
- 98) In general, the “build” process involves the following steps:
- 99) A source file containing assembly language instructions is created and edited.
- 100) The assembly language file is “assembled” into a re-locatable binary object file (machine code) that the target CPU can process.
- 101) The object file and any other binary support files are combined into a target specific executable file that can be run on the target system. A program called a “linker” carries out the process of combining and resolving location references.
- 102) The steps can be automated by a program called make, which reads its instructions from a “build” specific file referred to as a makefile (which is usually what it is named). This is shown below. You shouldn’t need to change anything in the make file, unless you know what you are doing, which requires significantly more reading and experience.



- 103) **NOTE:** A makefile is a very important file because it contains all the rules a compiler or assembler and linker needs to follow in order to do the job they are expected to. The makefile also describes the dependencies of your own code, and when you change your code or add new files to it, which is what we are doing now, i.e., your makefile needs to be edited. (Sophisticated makefiles can be written, which will recognize that new files have been added, but we are starting with a simple one.) In essence, a makefile is a list of commands that you could type in, one after another, from a command line prompt. Since there are number of commands with a variety of parameters as well as variety of different source files, it makes sense to just place all the appropriate information to “build an executable target” together in a single makefile. By creating a suitable makefile, we only need to invoke a single utility, logically enough named make, to accomplish the same objective.
- 104) See <http://www.gnu.org/software/make/> for more information regarding the make process.
- 105) Close your project and CCS.
- 106) **Credits:**
- 107) Dr. Dan Phillips and his graduate student Jonathan Barnard originally developed the original sequence of EE365 labs based on the TI MSP430 microcontroller family during 2005-2007. Until spring 2012 these labs used a MSP430 prototyping board developed by Jason Mann and Dr. Dan Phillips.
- 108) In 2007 and 2008, Dr. Dorin Patru upgraded the last two labs to use the ADC.

- 109) In 2011, Dr. Dorin Patru updated and upgraded all labs to Windows7, and CCE 4.2.1.
- 110) In 2013, Dr. Dorin Patru updated and upgraded all labs to CCS 5.4 and the TI LaunchPad board.
- 111) Countless TAs and students have and continue to contribute with valuable suggestions and corrections. The main authors wish to thank all of them.
- 112) This concludes this week's lab. Remember to write the brief report and upload it along with the project archive created above in the corresponding dropbox on mycourses.

Table 1 - Result of Register Manipulations

	Enter HEX values of specified register after each instruction is completed					
	PC	SR	R12	R13	R14	R15
<code>mov.w #0x12EF,r12</code>						
<code>clr.w,r13</code>						
<code>clr.w,r14</code>						
<code>clr.w,r15</code>						
<code>mov.b r12, r13</code>						
<code>mov.w r12, r14</code>						
<code>mov.w r12, r15</code>						