Code Composer Studio v5 Fundamentals for MSP430

Agenda

- Overview
- Hands-on workshop with MSP430
  - Getting Started
  - Watchpoints
  - Build properties
  - Debugging programs existing in flash
  - Portable Projects
  - ULP Advisor
  - GRACE basics

What is Code Composer Studio?

- Integrated development environment for TI embedded processors
  - Includes debugger, compiler, editor, operating system…
  - The IDE is built on the Eclipse open source software framework
  - Extended by TI to support device capabilities
- CCSv5 is based on “off the shelf” Eclipse
  - Going forward CCS will use unmodified versions of Eclipse
    - TI contributes changes directly to the open source community
  - Drop in Eclipse plug-ins from other vendors or take TI tools and drop them into an existing Eclipse environment
  - Users can take advantage of all the latest improvements in Eclipse
- Integrate additional tools
  - OS application development tools (Linux, Android…)
  - Code analysis, source control…

Upgraded user interface for fast, intuitive and easy development

Simplified user interface	shows developers what and when features are needed. 
Resource Explorer	facilitates use of example code. 
Development tools	for Windows and now Linux operating systems.* 
Eclipse open source framework 3.7	enables customization via latest third-party plug-ins. 
Video tutorials	explain how to get the most out of features. 

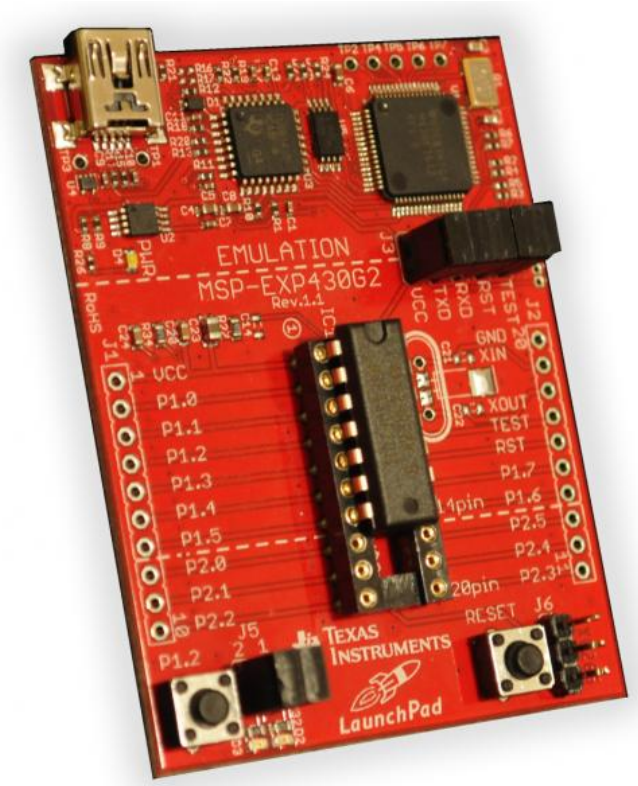
*MSP430, Stellaris MCU support available early 2012

Getting Started with CCSv5 and LaunchPad

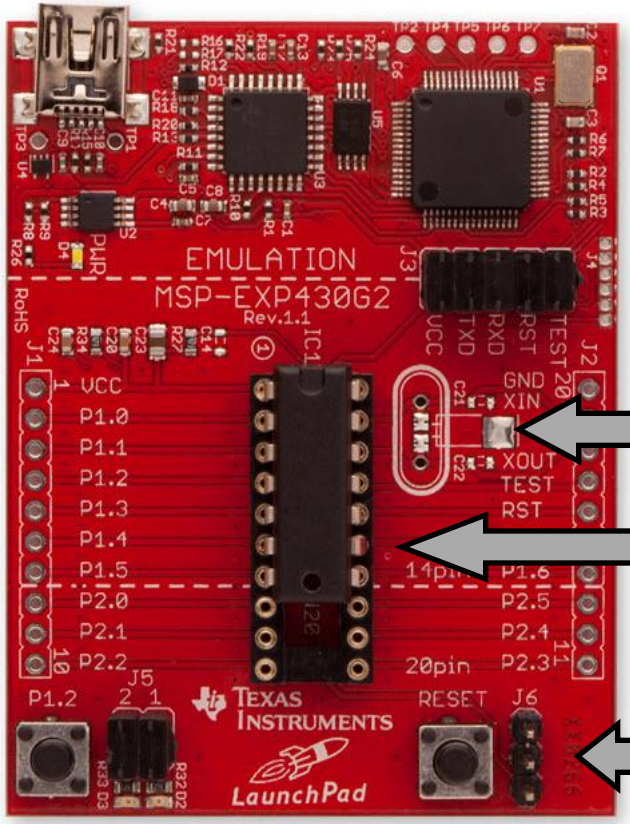
What is LaunchPad?

- Low cost (under $5), easy-to-use development tool intended for beginners and experienced users alike for the MSP430G2xx Value Line series devices

- Complete development environment that features integrated USB-based emulation and all of the hardware and software necessary to develop applications



LaunchPad: Development Board



USB Emulator Connection

Embedded Emulation

6-pin eZ430 Connector

Crystal Pads

Chip Pinouts

Part and Socket

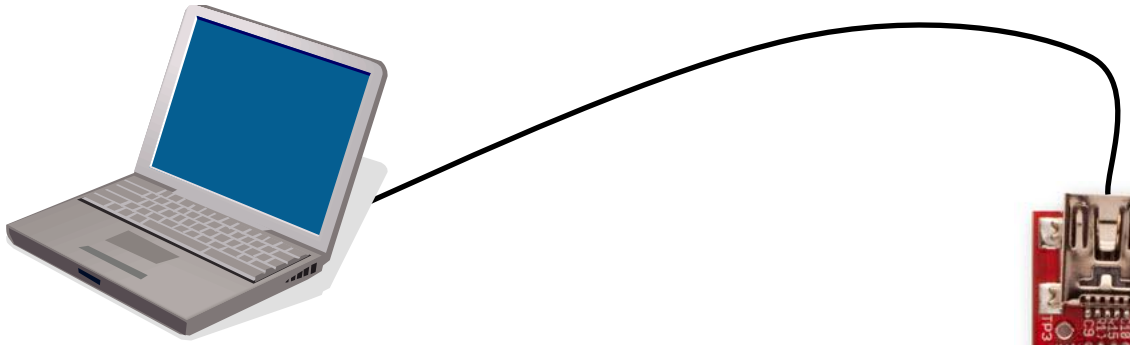
P1.3 Button

Power Connector

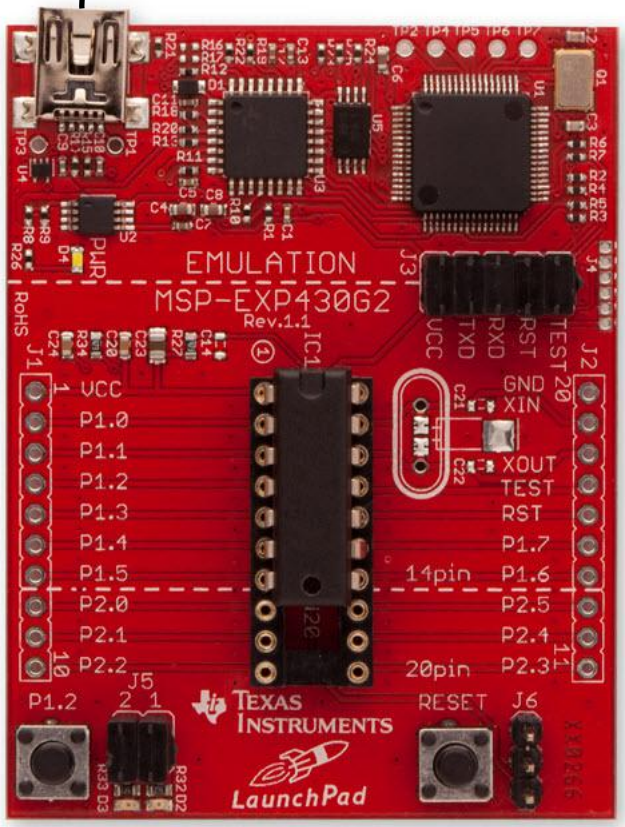
LEDs and Jumpers P1.0 & P1.6

Reset Button

LaunchPad: Hardware Setup



Connect mini-USB cable from PC to board



LaunchPad

- Want to learn more about LaunchPad?
  - Check out: http://processors.wiki.ti.com/index.php/MSP430_LaunchPad_%28MSP-EXP430G2%29

Workshop Instructions

- This workshop is a mix of presentation material and activities
- Whenever you see the “Let’s Do It” picture on a slide, there are actions for you to perform

- Some presentation slides may have other picture indicators to represent what the slide/bullet is for:
  - 

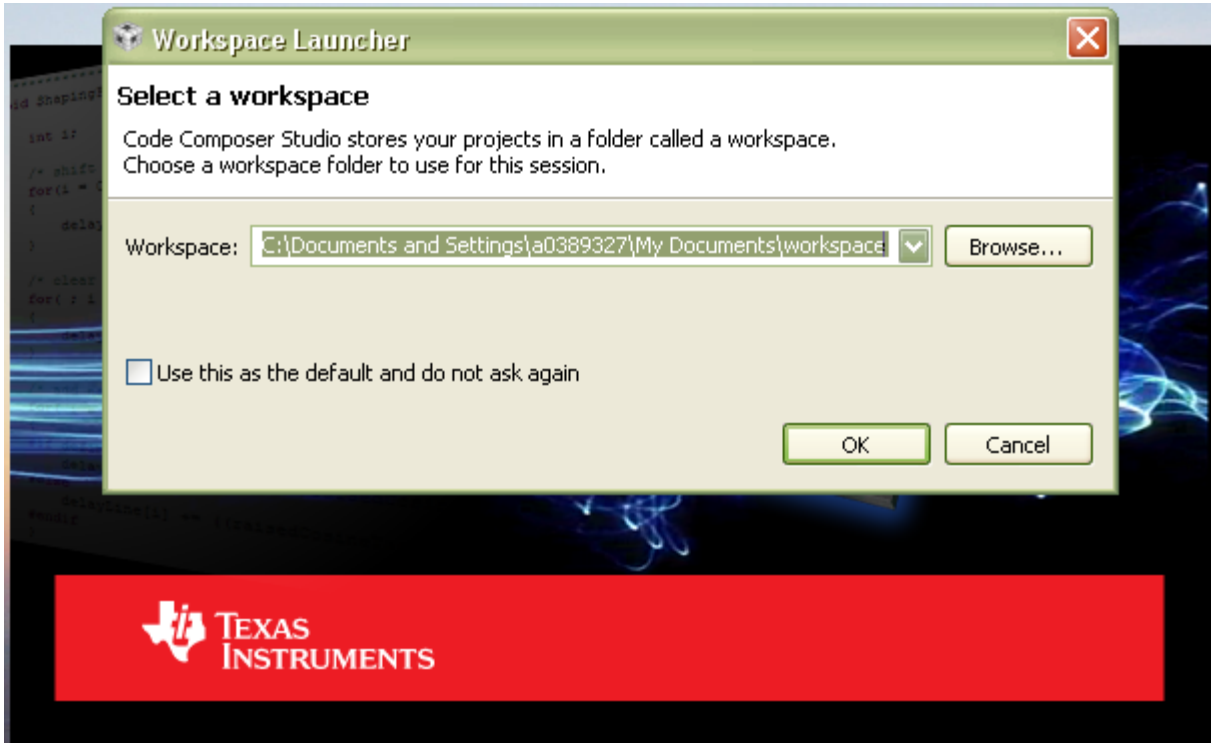
: indicates that this slide explains an Eclipse concept/term
  -  : indicates a comparison with CCSv3.x
  -  : indicates that this slide presents a CCS view/window

Blink LED1: Exercise Summary

- Key Objectives
  - Create and build a simple program to blink LED1
  - Start a debug session and load/flash the program on the LaunchPad
  - Run the program to blink LED1
- Tools and Concepts Covered
  - Workspaces
  - Welcome screen / Resource Explorer
  - Project Wizard
  - Template code
  - Project concepts
  - Basics of working with views
  - Debug launch
  - Debug control
  - Profile Clock

Workspace

- Launch CCS and select a workspace folder
  - Defaults to your user folder

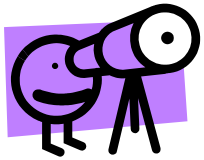


Eclipse Concept: Workspaces

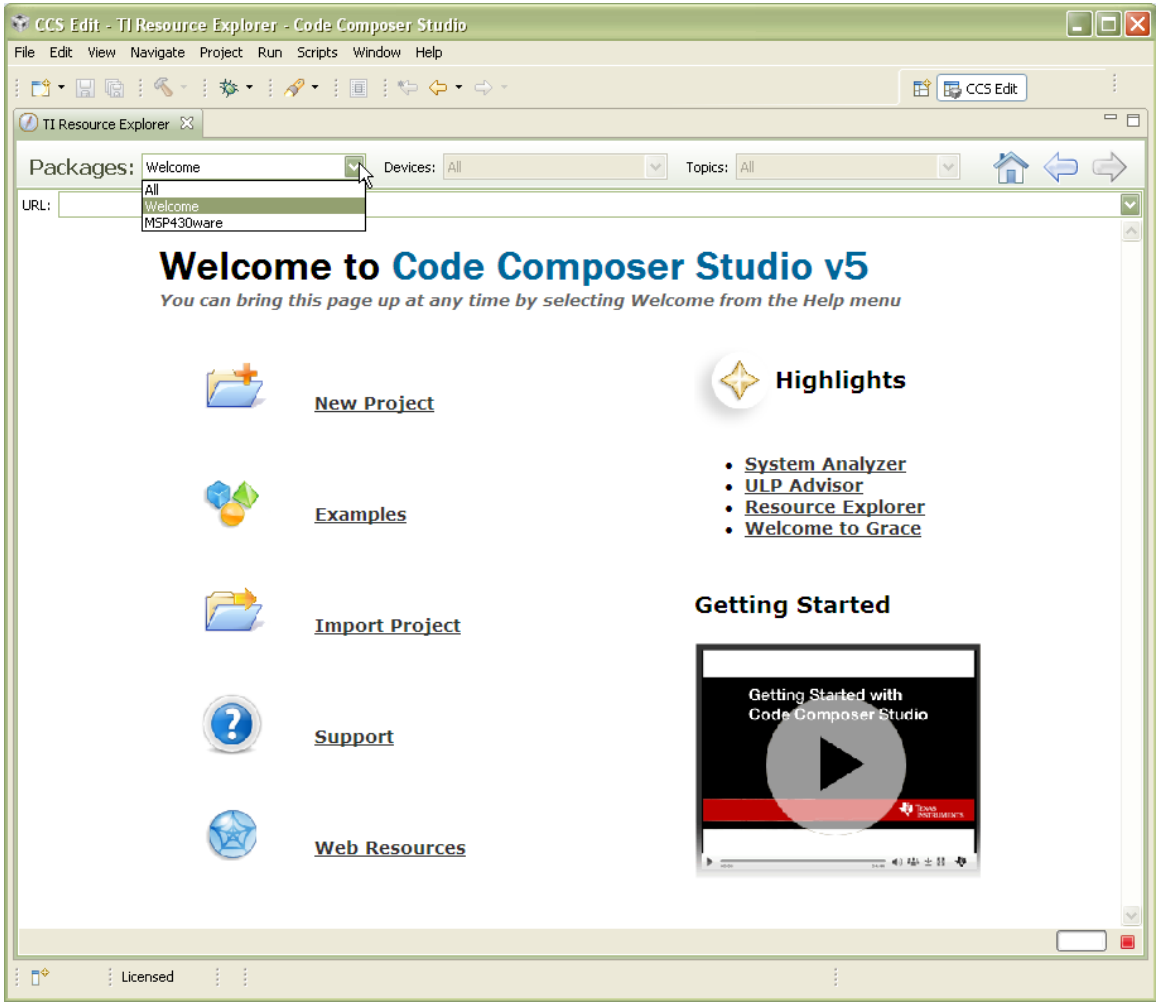
- Main working folder for CCS
- Contains information to manage all the projects defined to it
  - The default location of any new projects created
- User preferences, custom perspectives, cached data for plug-ins, etc all stored in the workspace

- Workspaces are not to be confused with CCSv3 workspace files (*.wks)
- Multiple workspaces can be maintained
  - Only one can be active within each CCS instance
  - The same workspace cannot be shared by multiple running instances of CCS
  - It is not recommended to share workspaces amongst users

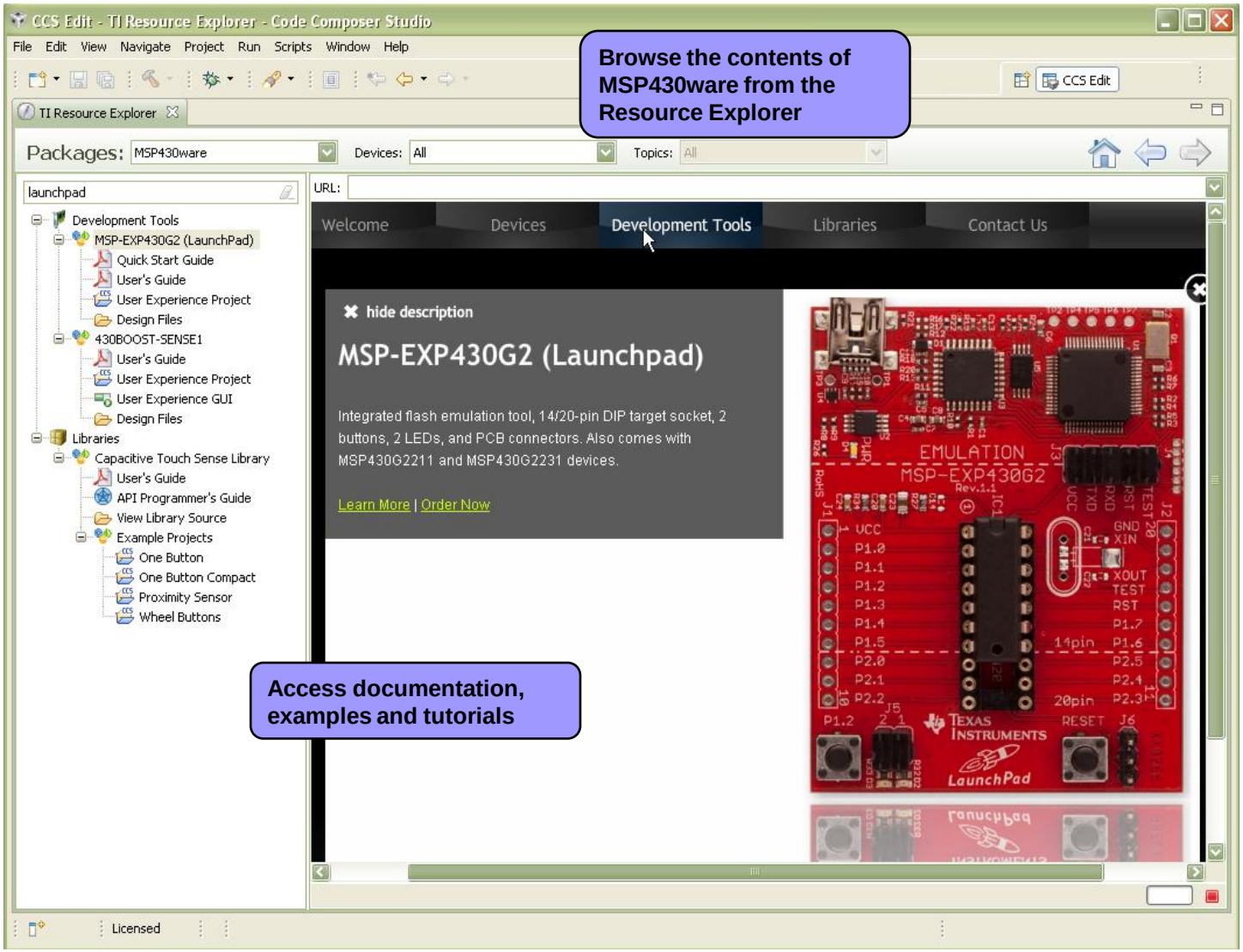
View: Resource Explorer



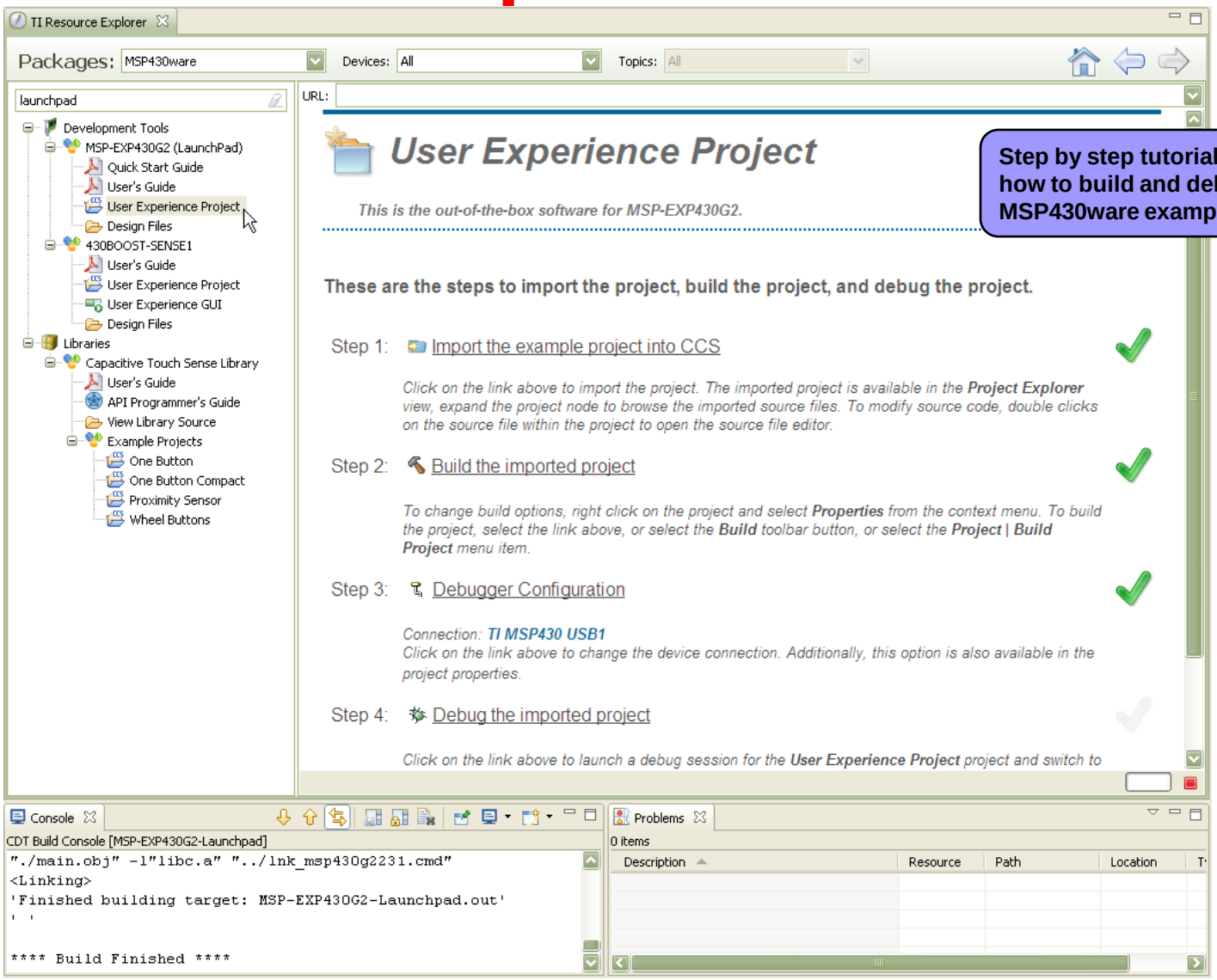
- The ‘Resource Explorer’ will display the ‘Welcome’ page the first time CCS is used with a new workspace
- Getting Started video introduces you to CCS
- Contains links to documentation, examples, support resources
- Buttons to create a new project or import an existing one into your workspace
- ‘Help->Welcome to CCS’ to return to the Resource Explorer in the future



Resource Explorer: MSP430ware



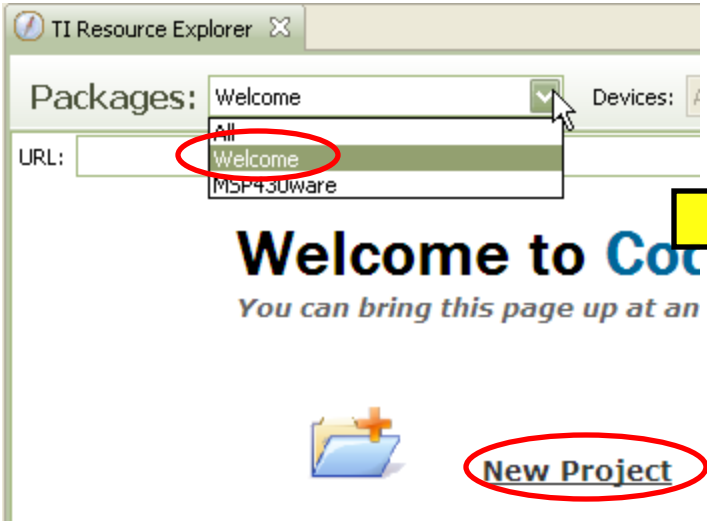
Resource Explorer: Tutorials

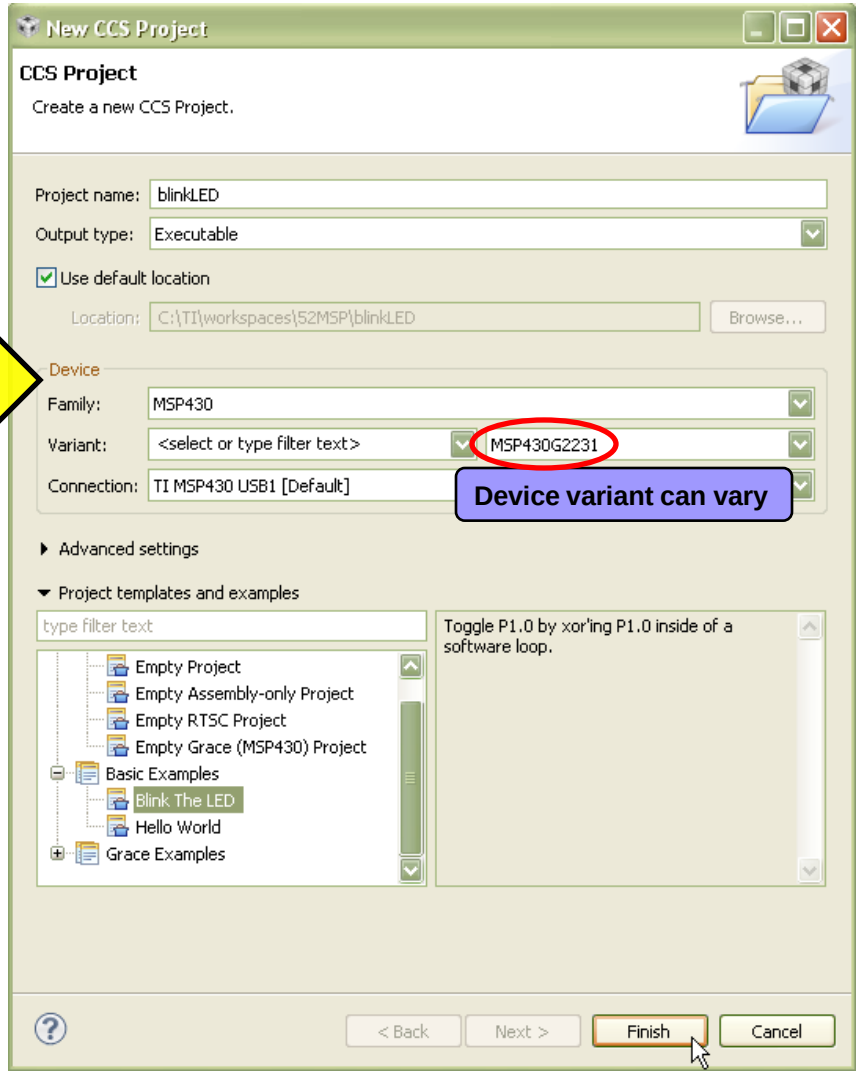


Step by step tutorials on how to build and debug MSP430ware examples

Create a New Project

- Start the Project Wizard
  - Select ‘New Project’ on the Welcome page

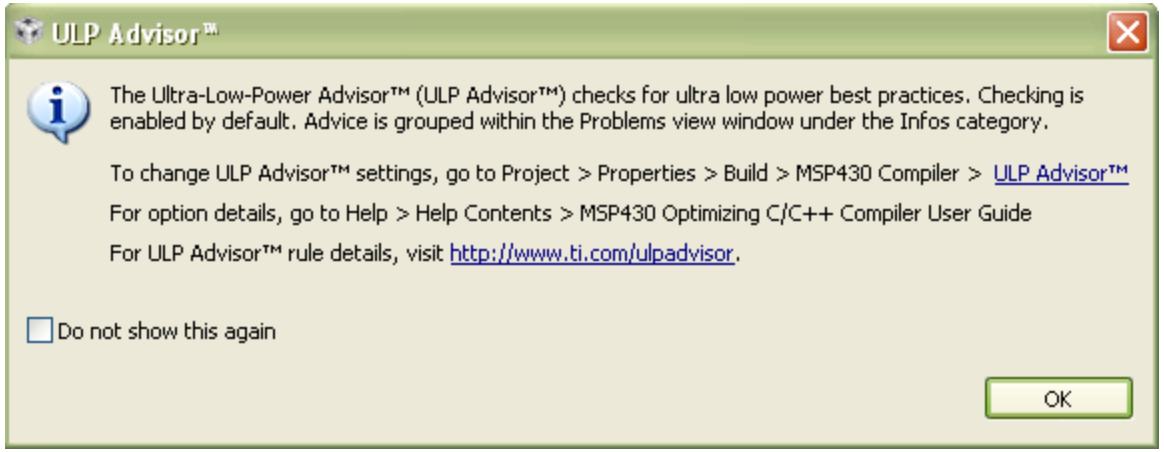




- Fill in the fields as shown on the right
  - Select the ‘Blink the LED’ template
  - Device variant for LaunchPad can vary
    - Check the device in the socket
- Select ‘Finish’ when done

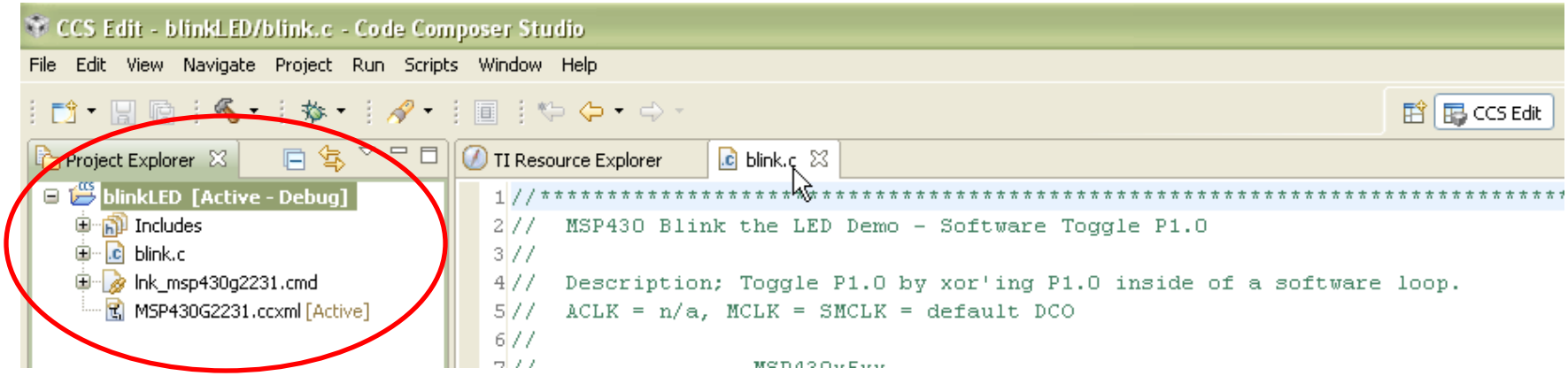
ULP Advisor Message

- A message highlighting the ULP Advisor will appear on project creation
- This message can be ignored for now as ULP will be covered later in this workshop
- Use the “Do not show this again” checkbox if you do not wish to see this message again



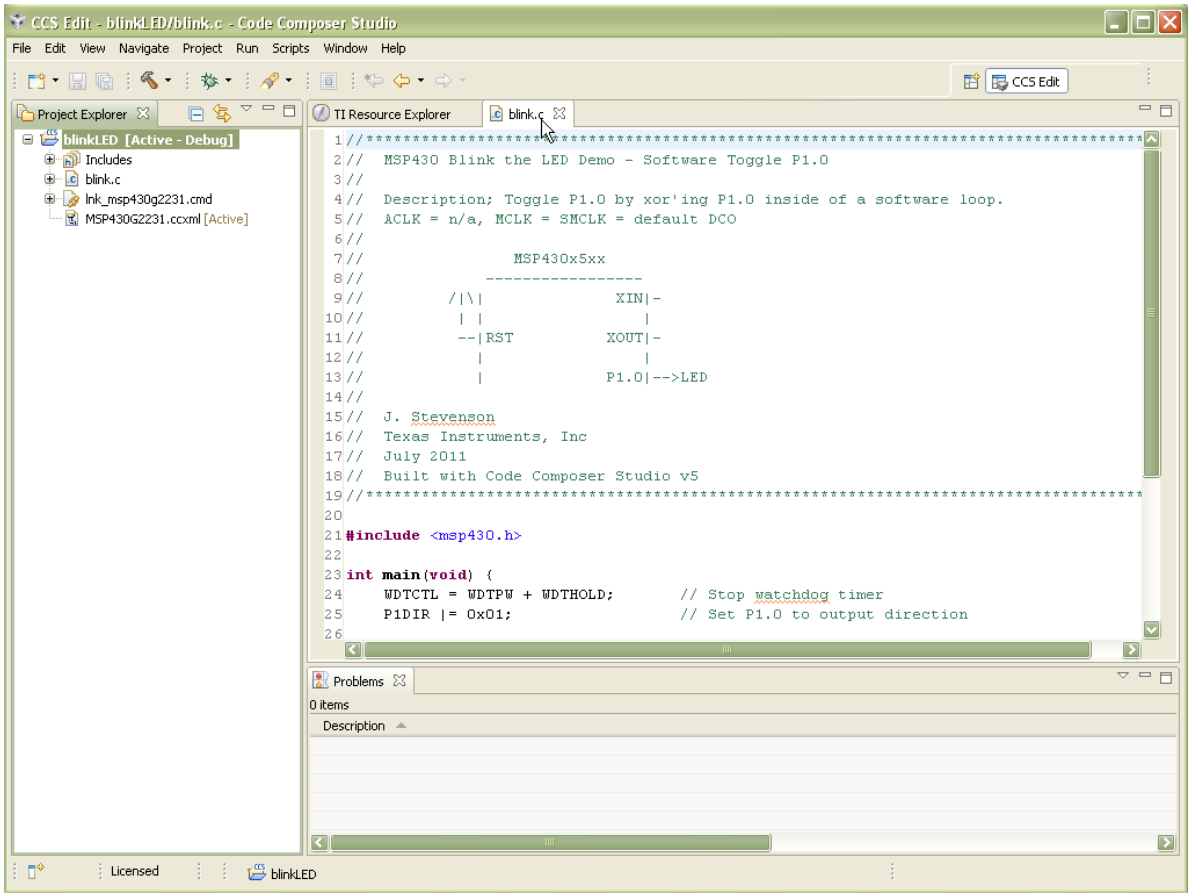
Create a New Project

- Once the project is created, a reference to it will be made in the workspace and the project is now available for use within the ‘Workbench’ and visible from the Project Explorer' view
- Expand the project by clicking on it in the Project Explorer to see its contents



Eclipse Concept: Workbench

- 'Workbench' refers to the main CCS GUI window
  - Equivalent to the 'Control Window' in CCS 3.x
- The Workbench contains all the various views and resources used for development and debug



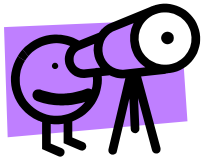
Workbench (Comparison with CCSv3.x)

- CCS 3.x
  - Only one Control Window can be opened for each debuggable CPU
  - Information is not shared between each Control Window
- CCS 4/5
  - Multiple Workbench windows can be opened ('Window->New Window')
  - Each Workbench window can differ visually (arrangement of views, toolbars and such), but refer to the same workspace and the same running instance of CCSv4/5
    - A project is opened from one Workbench will appear to be open in all the Workbench windows

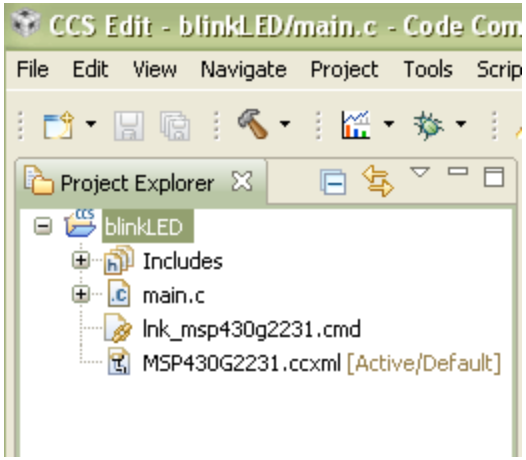
Eclipse Concept: Projects

- Projects map to directories in the file system
- Files can be added or linked to the project
  - Adding file to project
    - Copies the file into your project folder
  - Linking file to project
    - Makes a reference to the file in your project
    - File stays in its original location
    -  CCS 3.x project files used this concept exclusively
- Projects are either open or closed
  - Closed Projects:
    - Still defined to the workspace, but it cannot be modified by the Workbench
    - The resources of a closed project will not appear in the Workbench, but the resources still reside on the local file system
    - Closed projects require less memory and are not scanned during routine activity
    -  This differs from CCS 3.x, where closed projects do not appear at all in the CCS 3.x project view window.
- Projects that are not part of the workspace must be imported into the active workspace before they can be opened
  - Both CCSv4/5, CCE projects and legacy CCSv3 projects can be imported into the workspace

View: Project Explorer

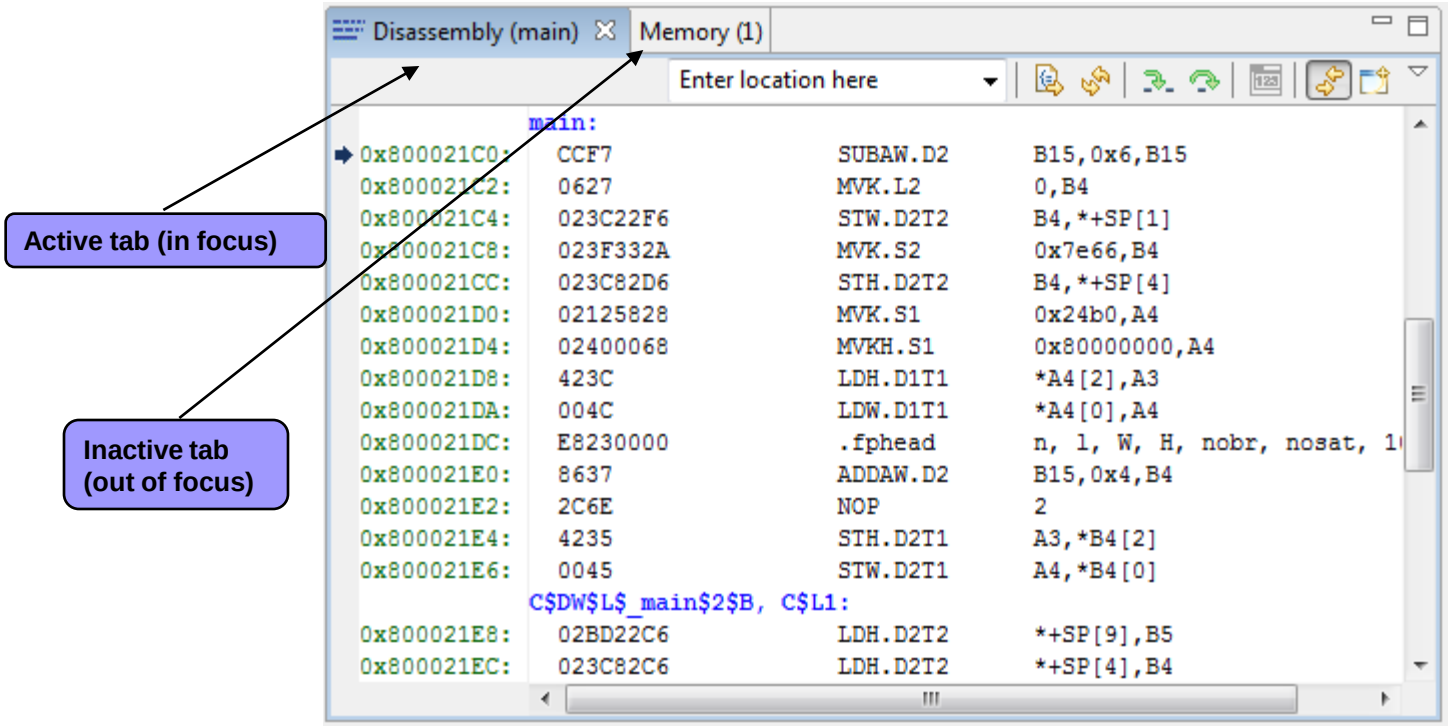


- Displays all projects defined in the active workspace
- The view is mostly a representation of the file system of the project folder
  - Linked files are marked with a special link graphic in the icon
- Use filters to hide various file types to reduce clutter in the view
  - Default is to filter CCS generated project files (.*)



Eclipse Concept: Views

- Views are windows within the main Workbench window that provide visual representation of some specific information
  - Most views can be accessed via the ‘View’ menu
  - Views are context sensitive

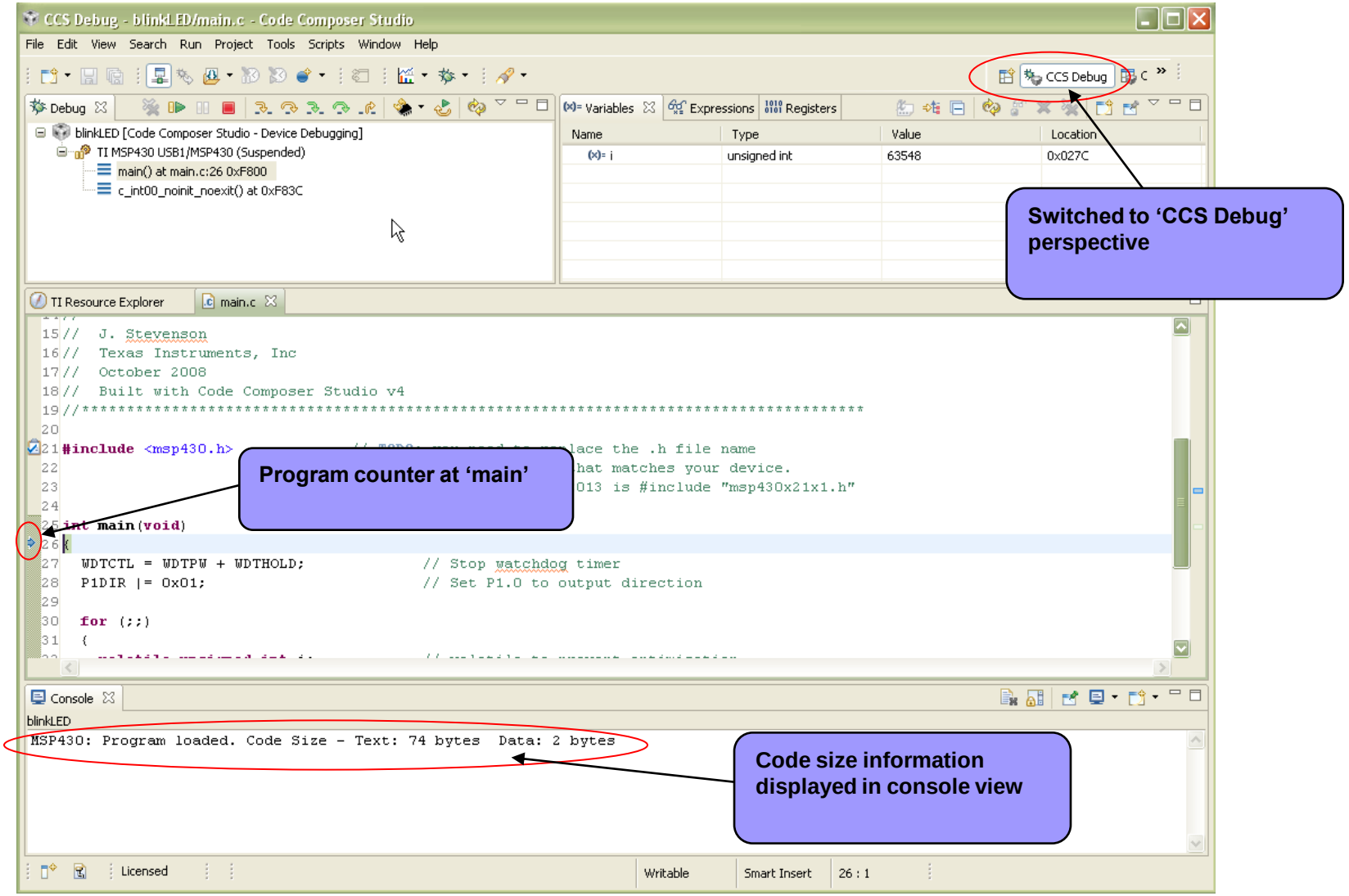


Build, Load/Flash the Program

- Click the ‘Debug’ button:

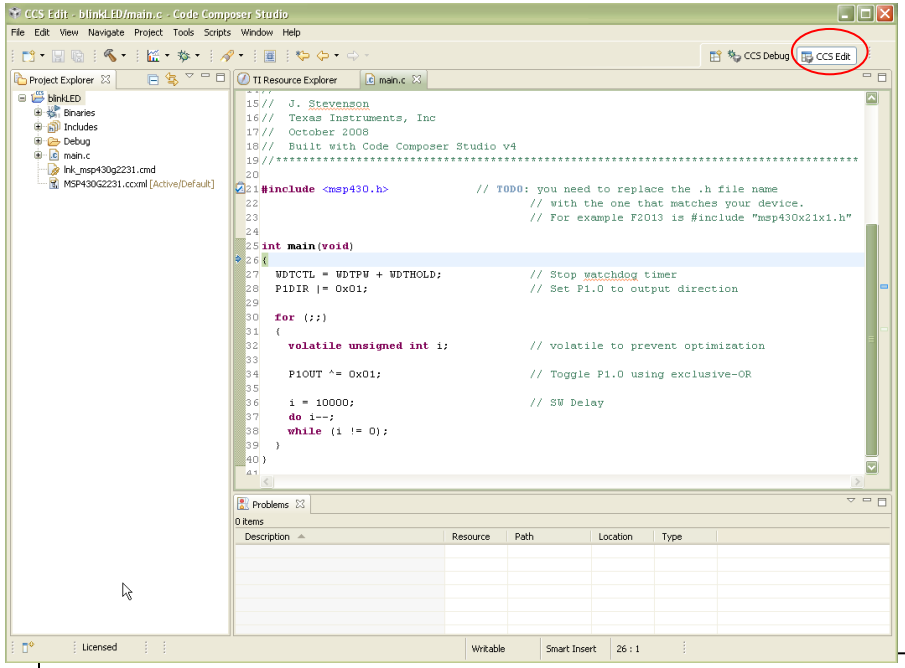
- ‘Debug’ button does multiple steps at once:
  - Prompt to save source files
  - Build the project
  - Start the debugger (CCS will switch to the ‘CCS Debug’ perspective)
  - Connect CCS to the target
  - Load (flash) the program on the target
  - Run to ‘main’
- Don’t want it to do all of the above at once? You can configure it to skip some steps (Debugger Options)

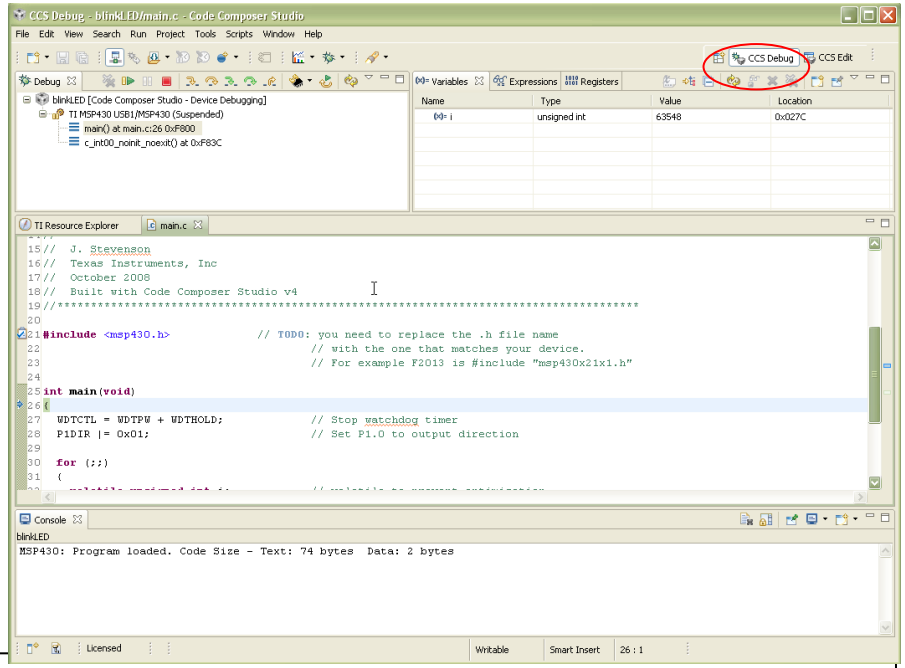
Build, Load/Flash the Program



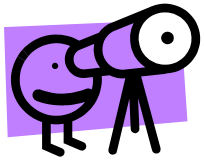
Eclipse Concept: Perspectives

- Defines the initial set and layout of views in the Workbench window
- Similar in concept to CCSv3 'workspaces' (*.wks) except that multiple perspectives are available from the Workbench window (though only one can remain active at a time)
- Each perspective provides a set of functionality aimed at accomplishing a specific type of task (‘CCS Edit’ for project development, ‘CCS Debug’ for debugging, etc)
- Can create custom perspectives

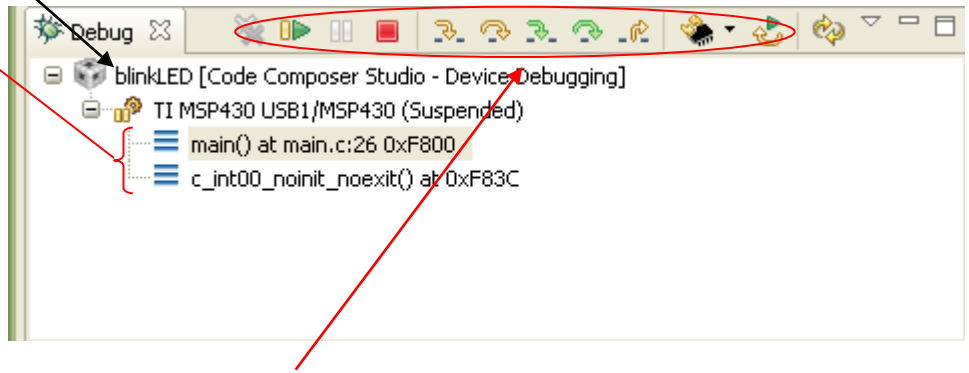




View: Debug

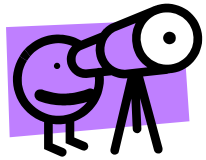


- The Debug view displays:
  - Target configuration or project
  - Call stack



- Buttons to ‘run, halt, terminate (debug session), source and asm stepping, reset CPU, restart program

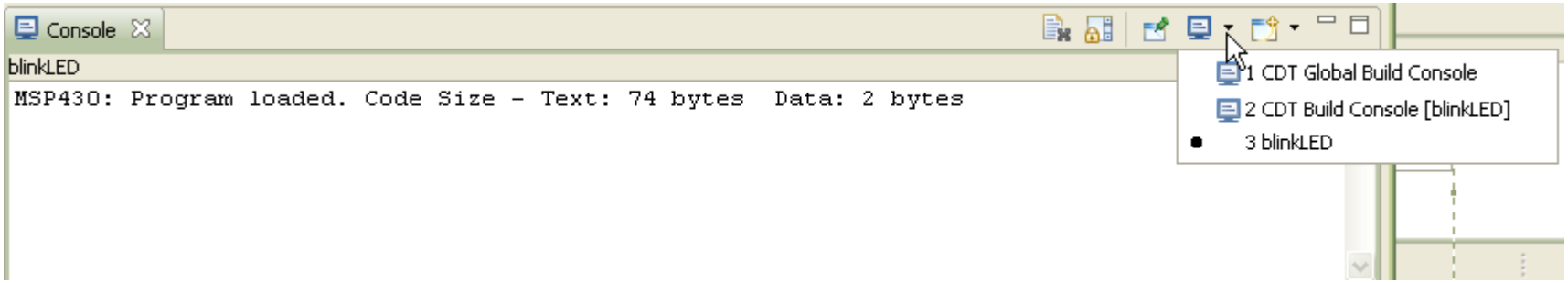
View: Console



- Multiple contexts
  - Can display build messages or debug messages (including CIO) depending on which console is selected
  - Automatically switches contexts when a new message occurs
    - Can use the “Pin” option to prevent this

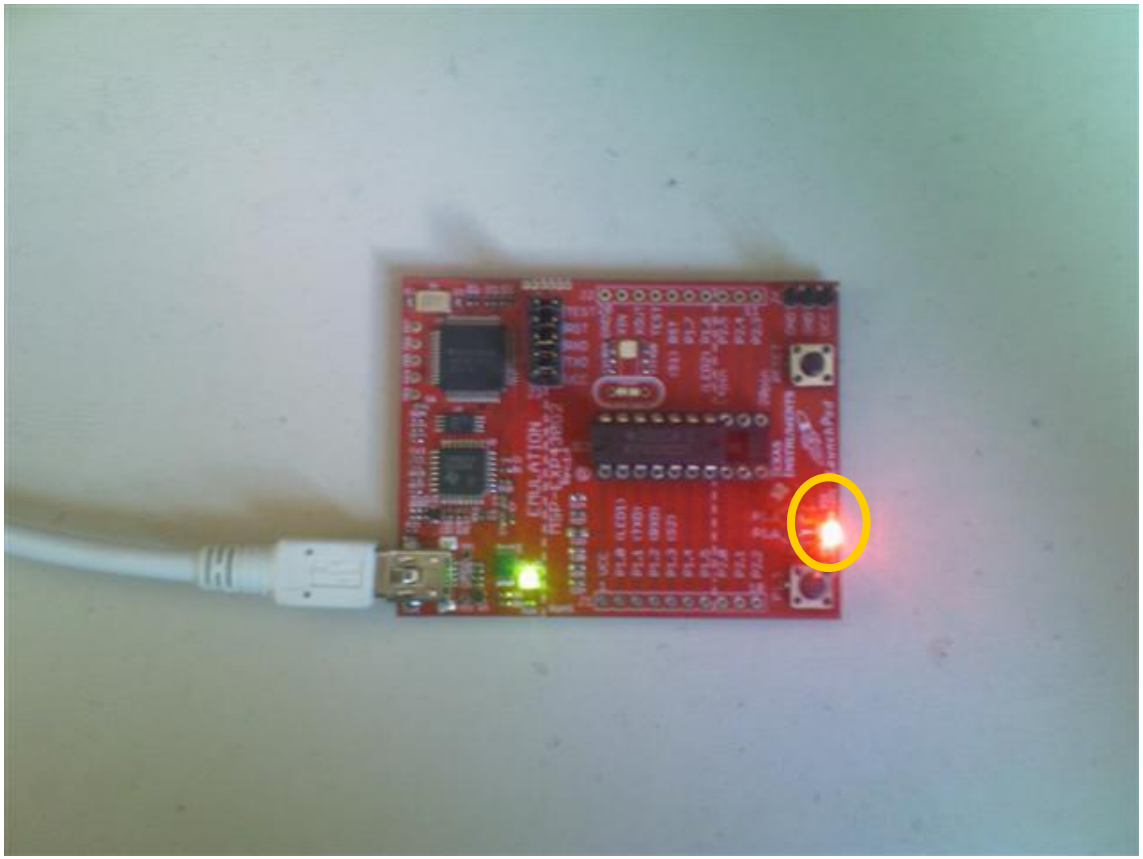
CCS 3.x had separate, dedicated views for build output, CIO output and debugger messages

- You can open multiple console windows
  - CIO output in one and build messages in the other



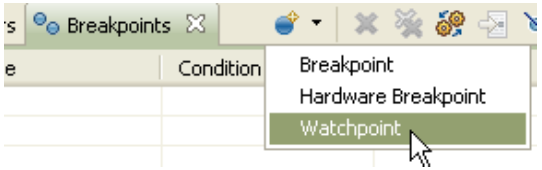
Blink LED1

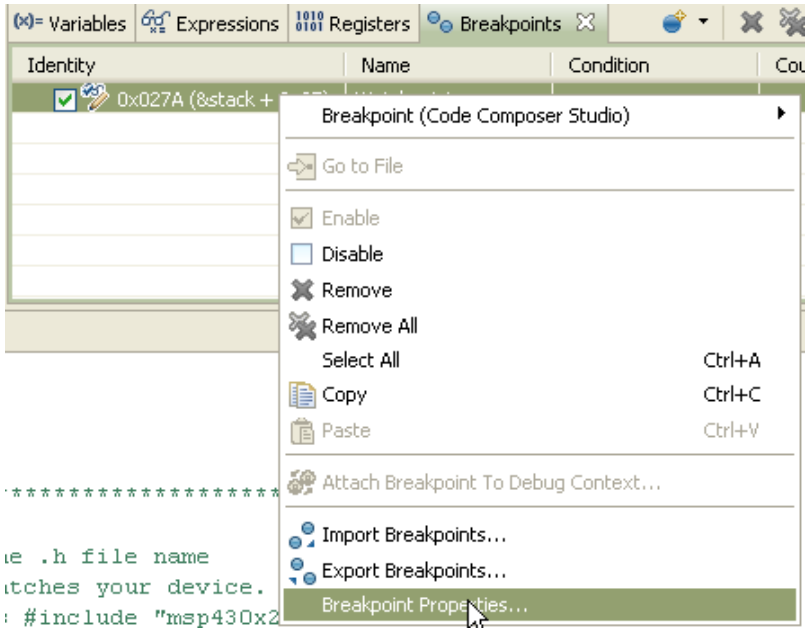
- Press the ‘run’ button  to run the program
  - LED1 on the LaunchPad should now be blinking



Debugging: Using Watchpoints

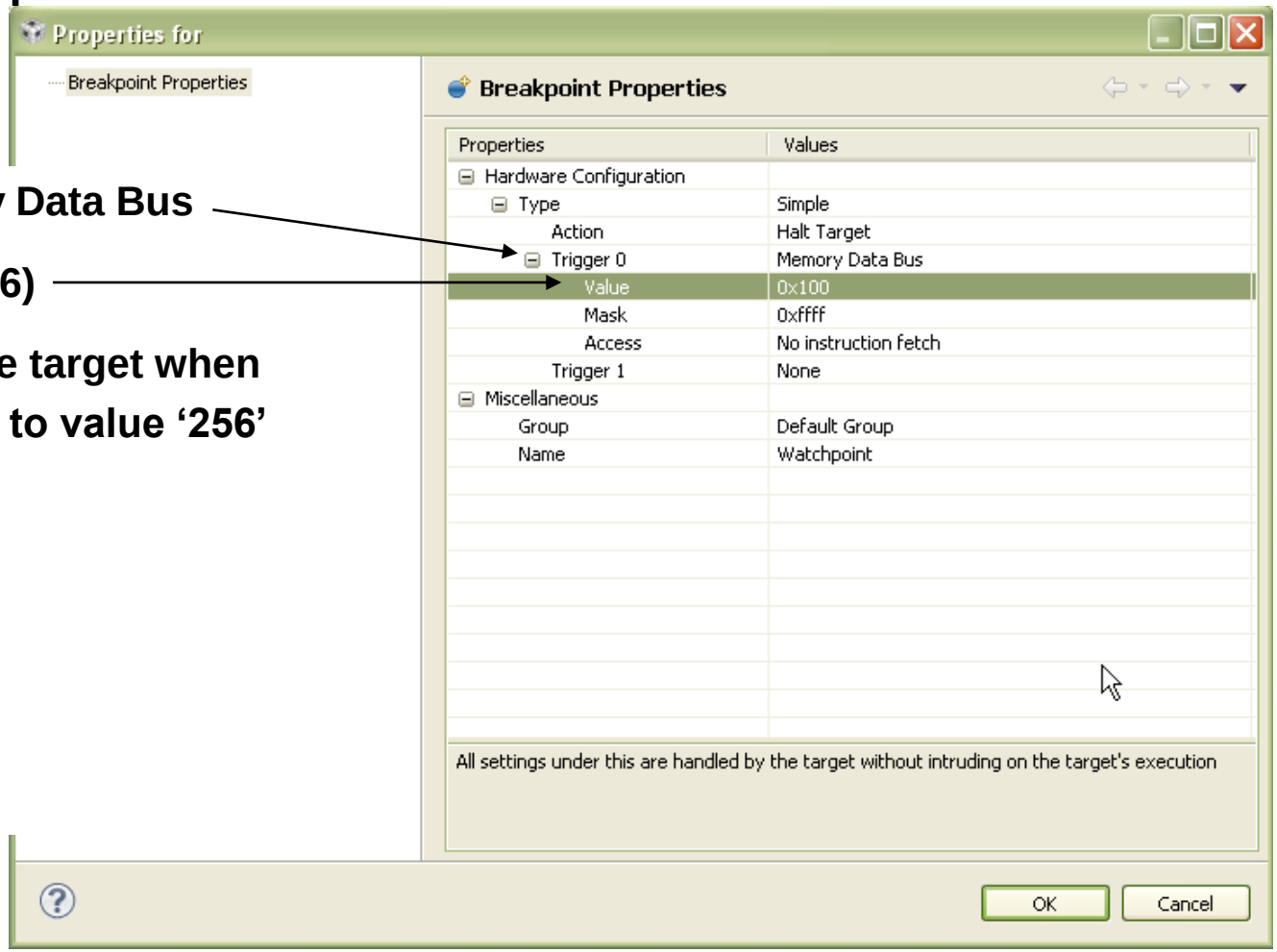
- Press the halt\suspend button to halt the running program
  - The code should stop somewhere inside the ‘for’ loop
- Open the ‘Breakpoints’ view
  - ‘View -> Breakpoints’
- Create a new Watchpoint and specify the variable ‘i’ as the location
- Right-click on the watchpoint and select ‘Breakpoint Properties’





Debugging: Using Watchpoints

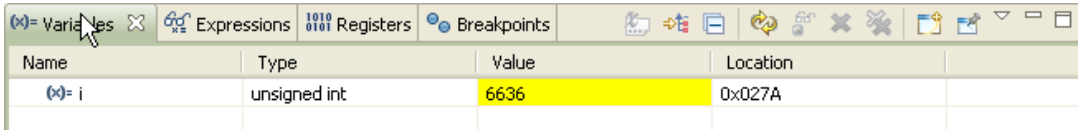
- Set the Properties as shown below:



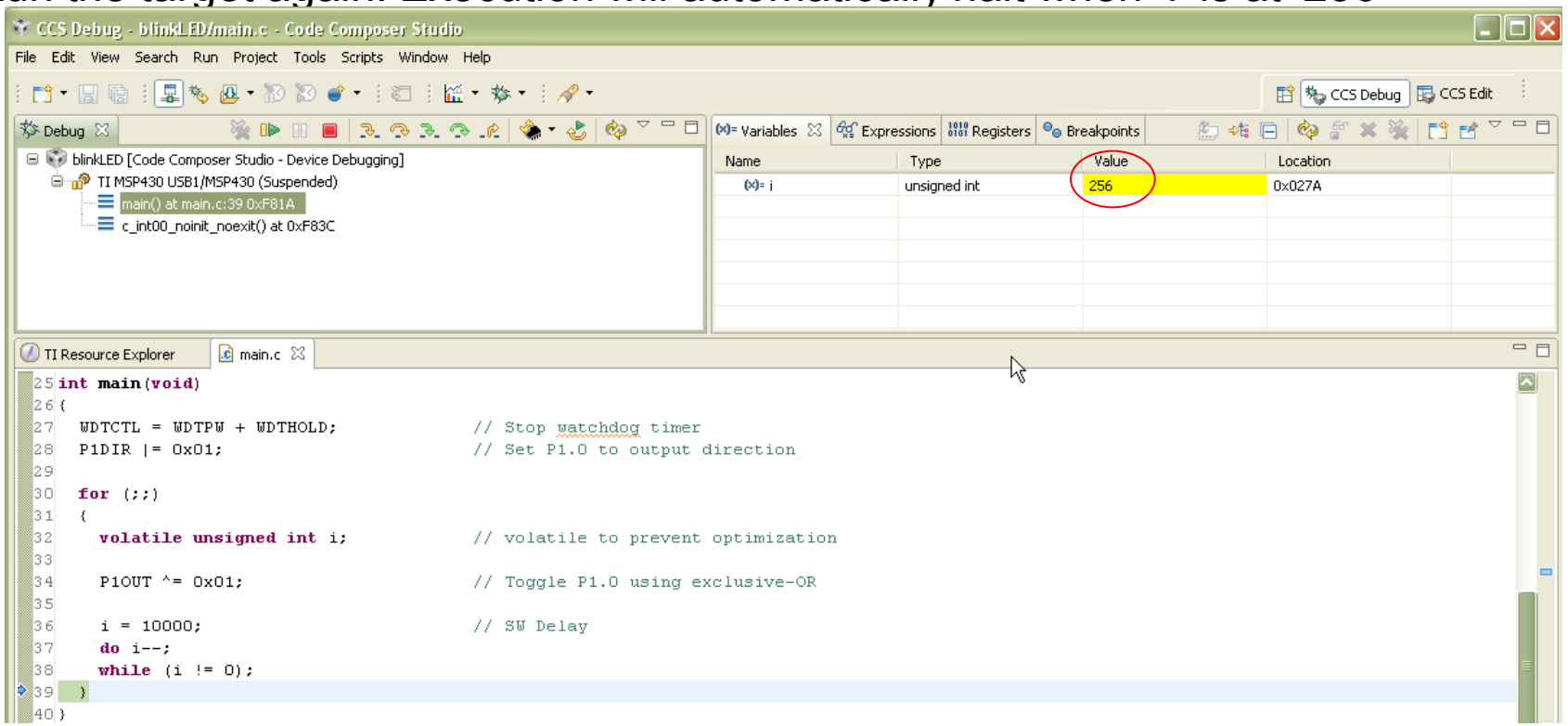
- **Trigger: Memory Data Bus**
- **Value: 0x100 (256)**
- **Basically halt the target when variable ‘i’ is set to value ‘256’**
- **Click OK**

Debugging: Using Watchpoints

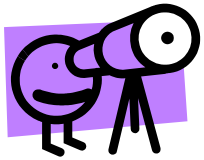
- Click on the ‘Variables’ view tab to bring it to the front



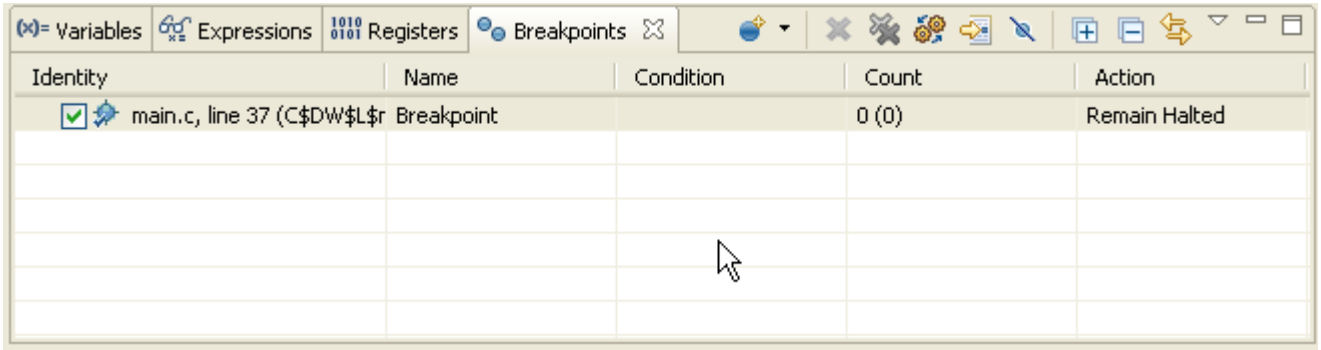
- Run the target again. Execution will automatically halt when ‘i’ is at ‘256’



View: Breakpoints

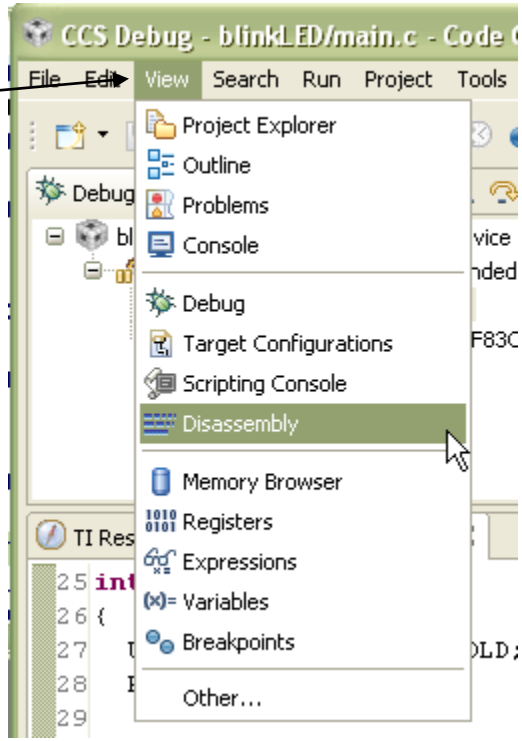


- View all available breakpoints
- Can group breakpoints by CPU (multi-core device)
- Specify various actions when the breakpoint is triggered
  - Refresh All Windows or update a specific view (replaces “Animate” in CCS 3.3)
  - Control Profiling (set profile halt/resume points)
  - File I/O (Probe Points in CCS 3.3)
  - Run a GEL expression
  - Set a Watchpoint
  - Control CPU trace (on selected ARM & DSP devices)



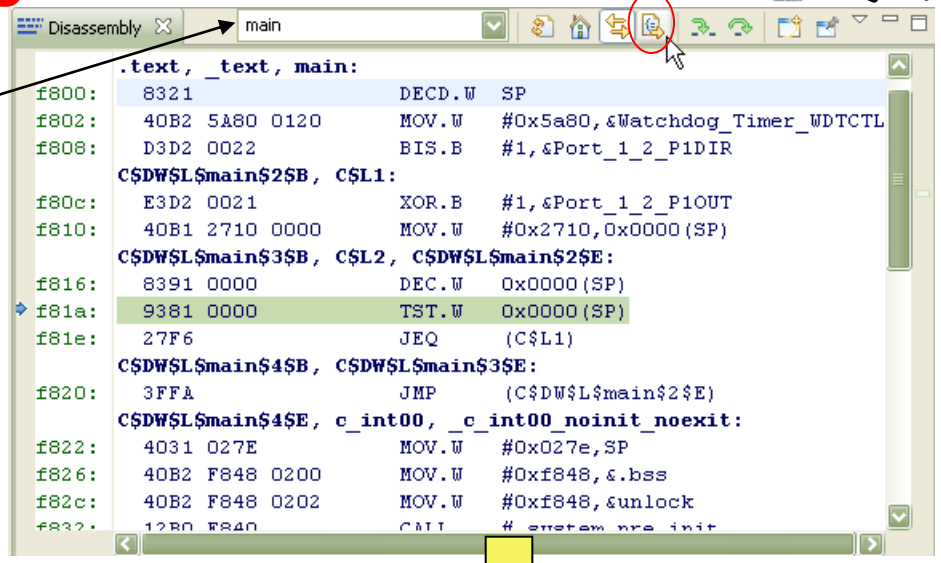
More Debugging

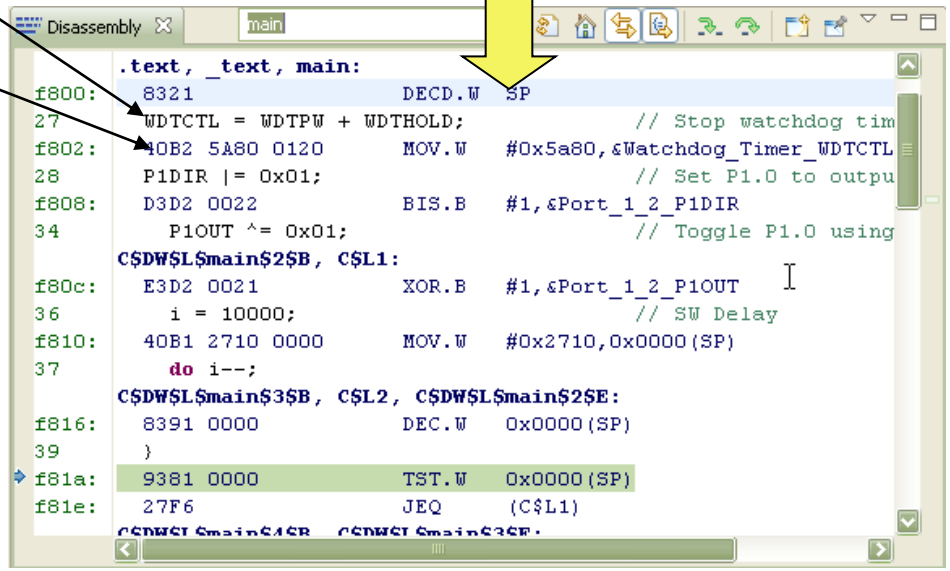
- Investigate other debugging views (Open via ‘View’ menu)
  - Memory Browser
  - Registers
  - Disassembly (see next slide)
- Use the buttons in the ‘Debug’ view to:
  - Restart the program
  - Source stepping
  - Assembly stepping



View: Disassembly

- Go to the ‘main’ symbol in the Disassembly view by typing it in the address field and hit ‘ENTER’
- Toggle the ‘Show Source’ button
- Note how interleaved source with the disassembly is toggled on/off





Debugger Options

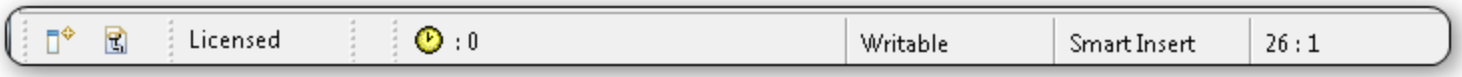
- Many debugging features can be enabled/disabled from the Debugger Options
- During a debug session in the ‘CCS Debug’ perspective:
  - ‘ Tools->Debugger Options->Generic Debugger Options’
- Configure a variety of debug options like enable/disable:
  - auto-run to main
  - auto-connect to a HW target
  - real-time options (on supported HW)
  - program verification on load
  - etc…
- Use the “Remember My Settings” option to have the settings apply for subsequent debug sessions

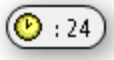
Remove the Watchpoint

- Clear breakpoints
  - Go to the Breakpoints view
  - Click on the button to “Remove all Breakpoints” , say yes to the prompt
- Restart execution
  - Go to the Debug View
  - Click on the Restart button
  - The program counter should be back at main

Counting Cycles

- The profile clock
  - Available on most devices and can be used to count cycles
  - On some targets it can be used to count other events as well
- Enable the Clock
  - Run -> Clock -> Enable
  - The clock will now be displayed on the status bar



- Place a breakpoint and run to it
  - Double click in the selection margin on line 35 (“}” for the for loop) to add a breakpoint
  - Click the run button to run to this point
  - Clock should now show 24 cycles 

Remove the Breakpoint

- Clear breakpoints
  - We need to clear the breakpoint or it will get set for our next lab as well as we are using the same target
  - Go to the Breakpoints view
  - Click on the button to “Remove all Breakpoints” , say yes to the prompt

Terminate the Debug Session

- Go to the Debug view
- Click on the terminate button 
- This will kill the debugger and return you to the CCS Edit perspective

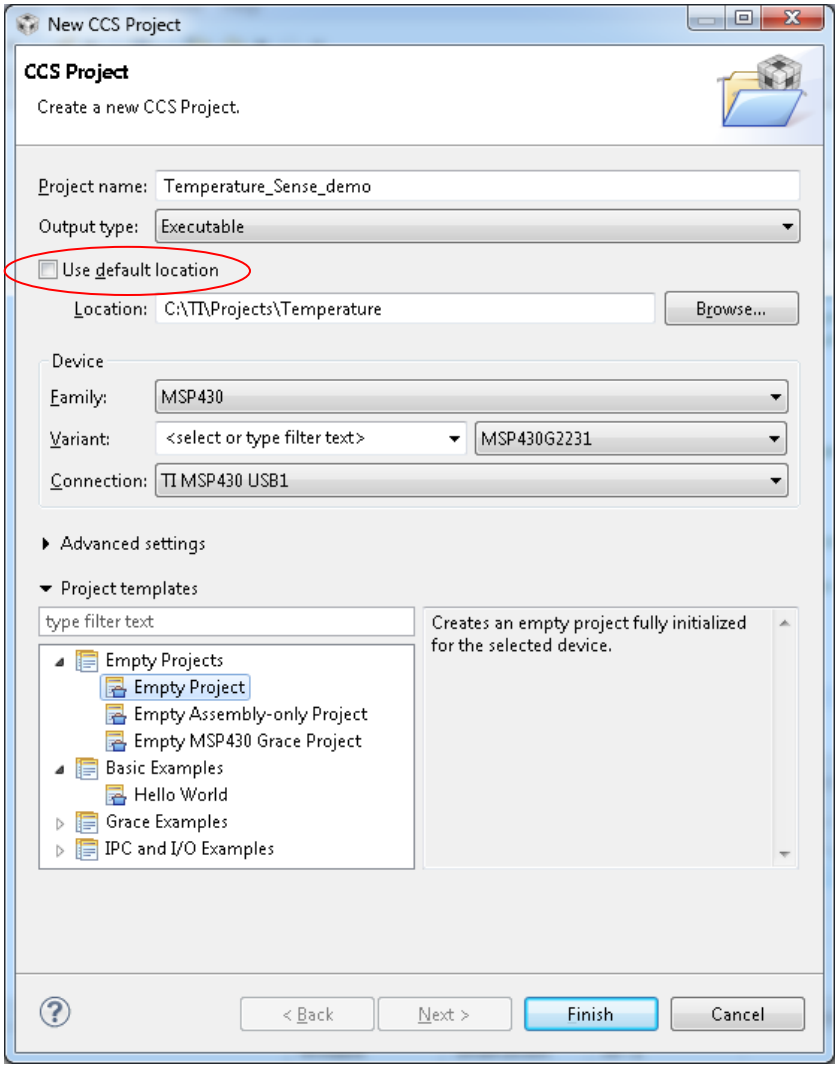
TEMPERATURE SENSE DEMO

Temperature Sense Demo: Briefing

- Key Objectives
  - Create and build the Temperature Sense Demo
  - Start a debug session and load/flash the program on the LaunchPad
  - Run the program to start Temperature Sense Demo
- Tools and Concepts Covered
  - Adding source files
  - “Focus” concept
  - Loading Symbols
  - Changing Build Options
  - Changing Compiler Version

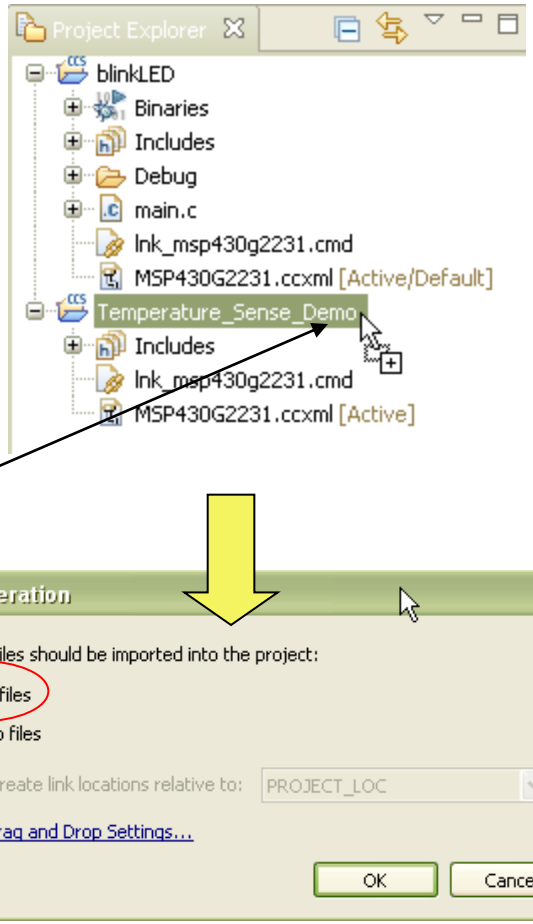
Create a New Project

- Launch ‘New CCS Project’ Wizard
  - In ‘CCS Edit’ perspective, Project -> ‘New CCS Project’
- Project Location
  - Note that this time we will be creating this project outside the workspace
  - Uncheck the box to use default location
  - Specify **C:\ti\Projects\Temperature**
- Select ‘Finish’ when done



Add Temperature Sensor Source Code

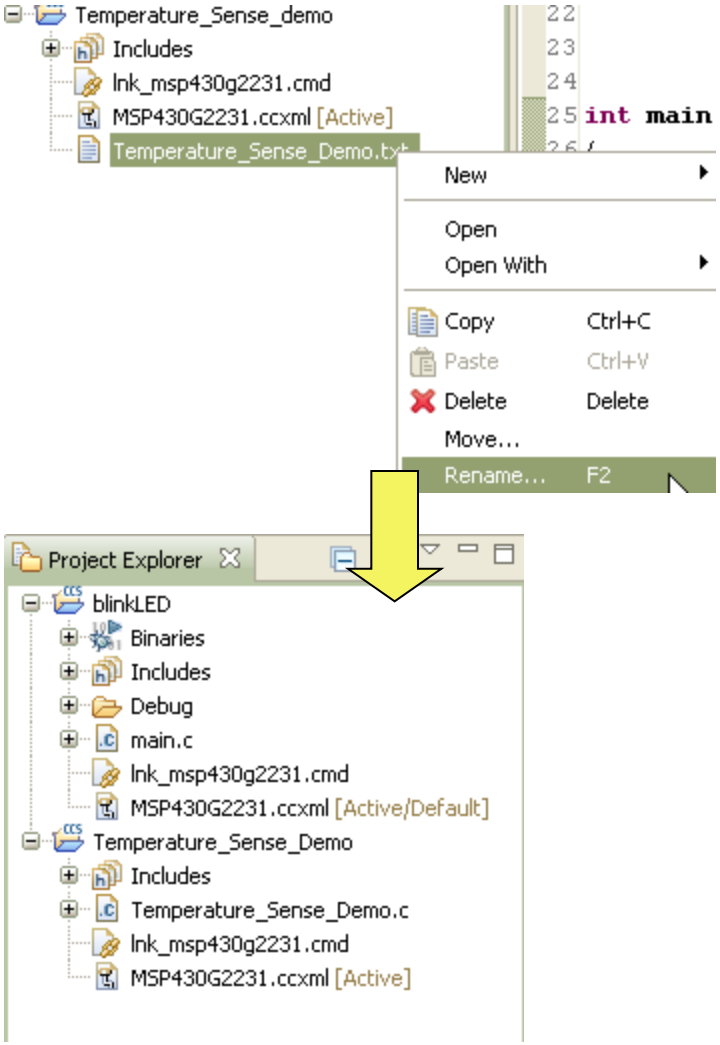
- Remove the generated ‘main.c’ file from the project
  - Select it in the project explorer and press delete
- Open Windows Explorer and browse to:
  - C:\TI\LaunchPad\temperature_sensor
- Drag and drop ‘Temperature_Sense_Demo_G2xxx.txt’ to the ‘Temperature_Sense_Demo’ project
  - Use the file that matches the MSP430 device being used
    - For G2553, use: **Temperature_Sense_Demo_G2553.txt**
  - Make sure the file is dragged to the ‘Temperature_Sense_Demo’ project
- In the dialog popup, select the option to ‘Copy Files’ and hit ‘OK’



Add Temperature Sensor Source Code

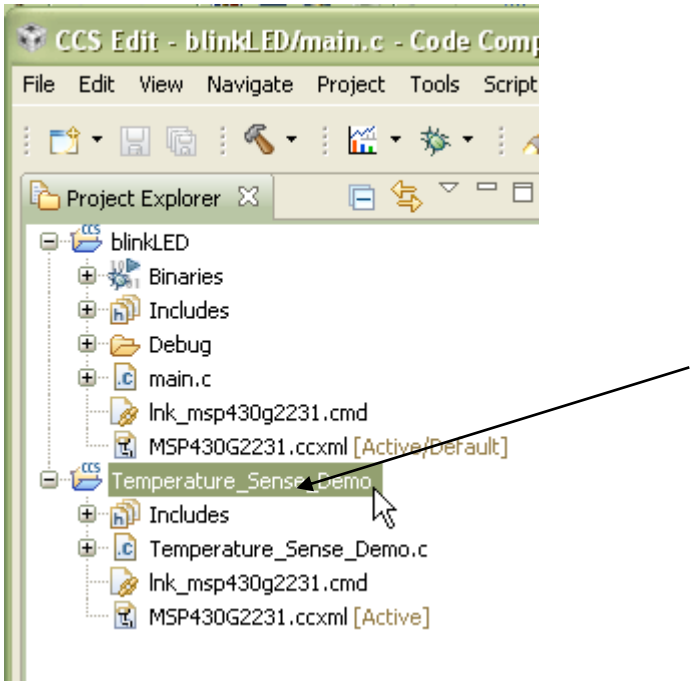
- Right-click on the ‘Temperature_Sense_Demo_G2xxx.txt’ file in the Project Explorer and select the option to ‘Rename..’ the file
  - Rename the file so that the ‘.txt’ extension is renamed to ‘.c’

- Project is ready to build



Eclipse Concept: Focus

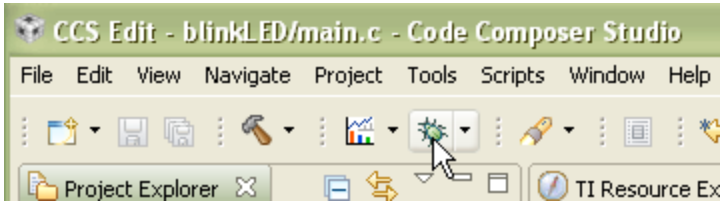
- Focus refers to the highlighted portion of the workbench
  - Can be an editor, a view, a project, etc.
- This concept is important since several operations inside Eclipse are tied to the element in focus
  - Project build errors, console, menu and toolbar options, etc.



'Temperature_Sense_Demo' project is in 'Focus' since it has been selected. So pressing the 'Debug' button will build the project and start the debugger for the 'Temperatrue_Sense_Demo' project

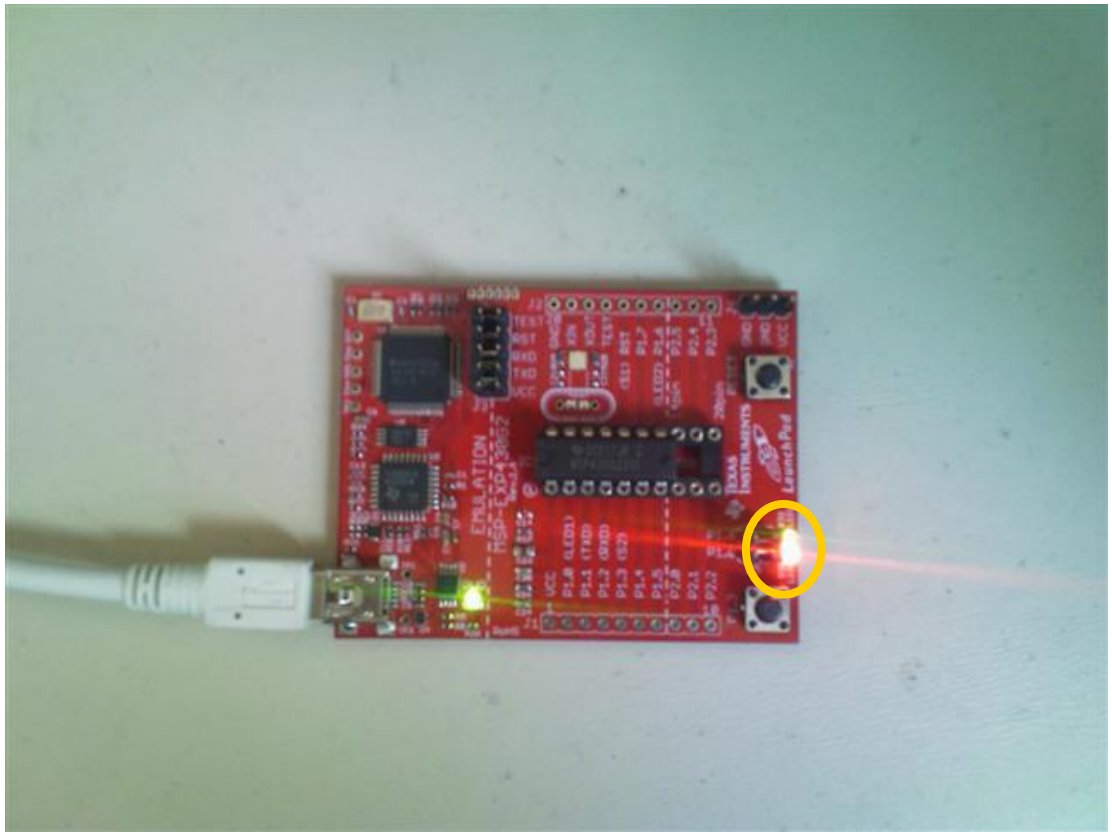
Build and Load/Flash the Program

- Make sure the ‘Temperature_Sense_Demo’ project is in ‘Focus’. Then use the ‘Debug’ button



Temperature Sense Demo

- Press the ‘run’ button  to run the program
  - LED1 (red) and LED2 (green) on the LaunchPad should now alternate blinking

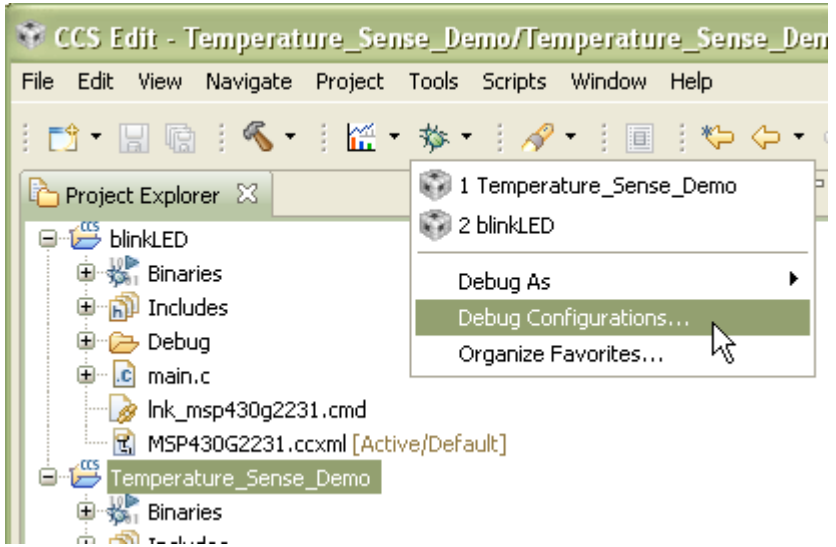


Temperature Sense Demo: Debugging

- Press the halt\suspend button to halt the running program
  - The code should stop in the PreApplicationMode() function.
- Step-into the code once and it will enter the timer ISR for toggling the LEDs (ta1_isr)
- Step-over a few more times and notice that the red and green LEDs alternate on and off
- When you are done, terminate the debug session

Loading Symbols for Flashed Program

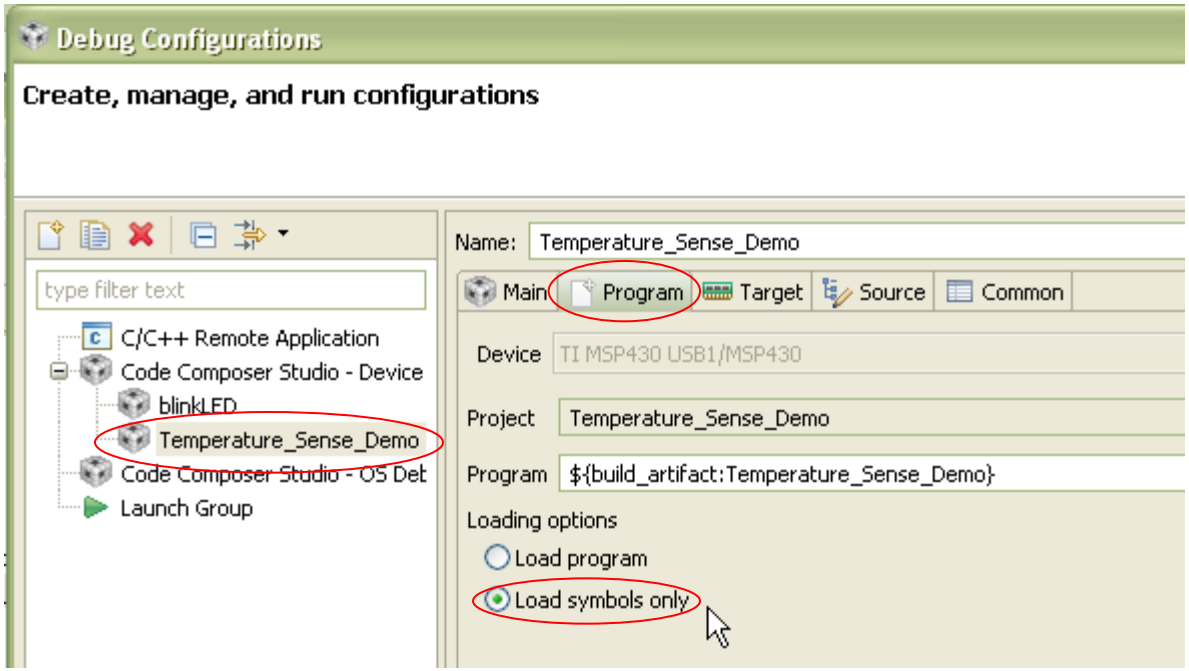
- If the program is already flashed in CCS and just wish to debug the existing code flashed on the target, you can configure CCS to debug the project by loading symbols only
- Select the drop-down menu next to the ‘bug’ button and select ‘Debug Configurations..’



Eclipse Concept: Debug Configurations - Cached information created when a debug session is first launched for a project or target configuration. Information cached includes which target configuration to use, debug settings…

Loading Symbols for Flashed Program

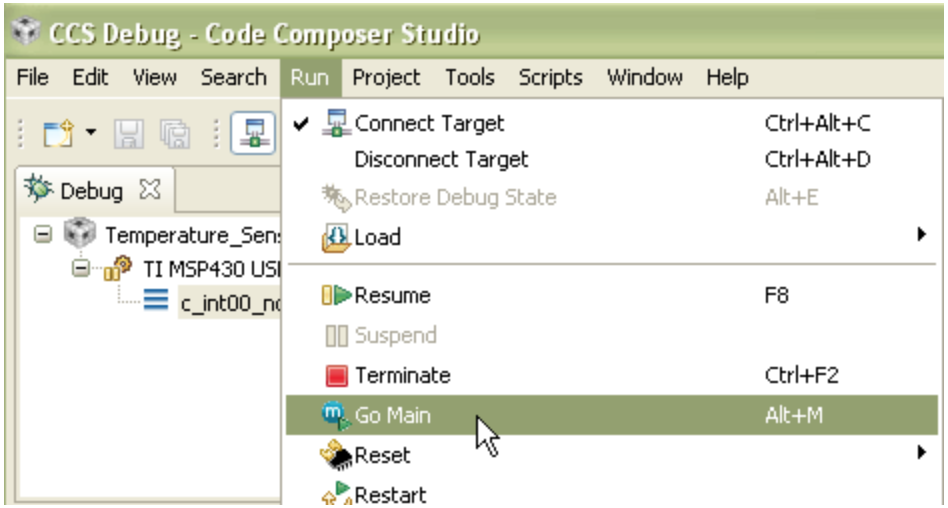
- Select ‘Temperature_Sense_Demo’ in the left panel and the ‘Program’ tab in the right panel
- Under ‘Loading options’, select ‘Load symbols only’



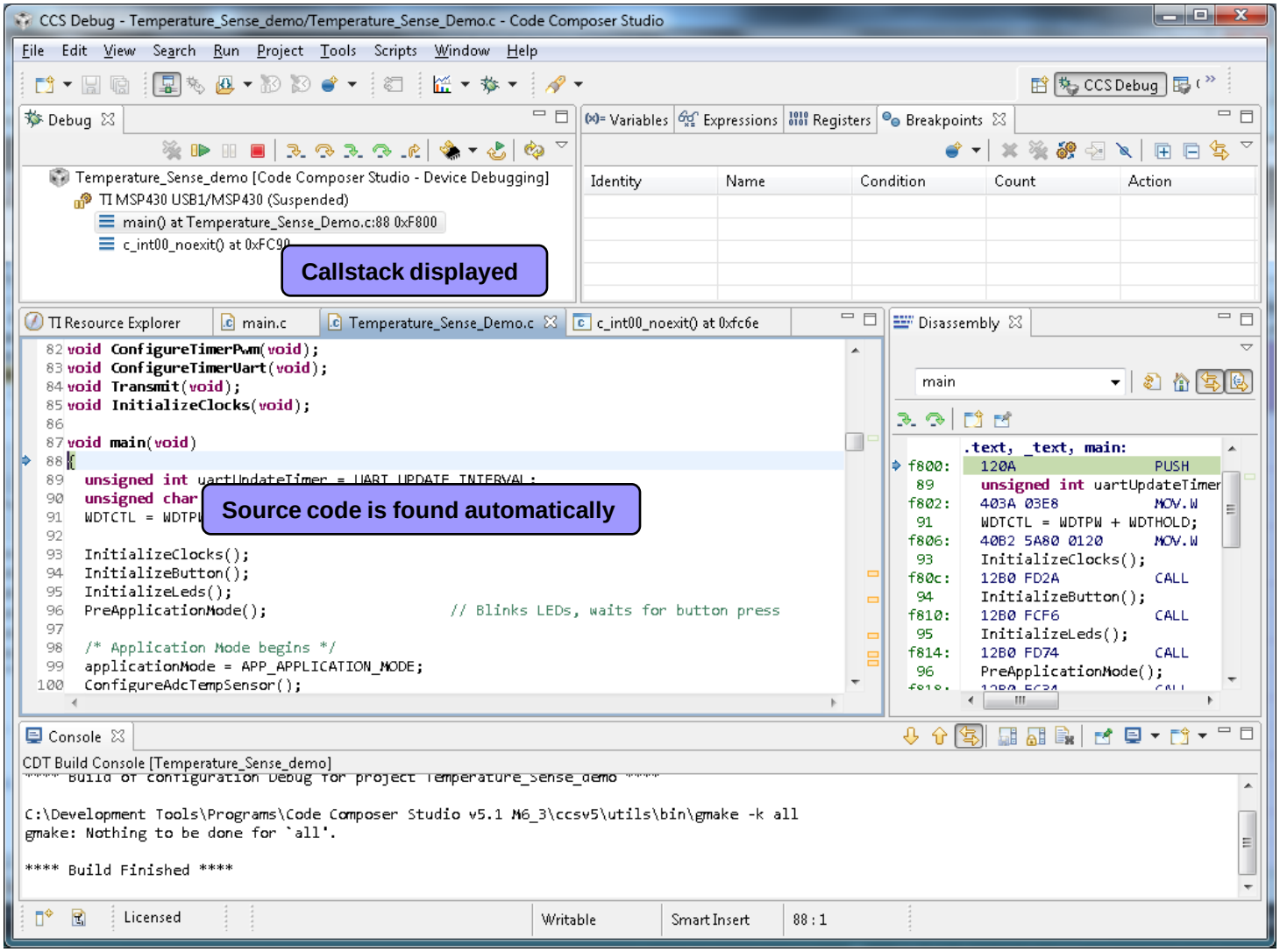
- Then select ‘Apply’ and then ‘Debug’

Loading Symbols for Flashed Program

- The debugger will start up, connect to the target, and load only the symbols for the program for the debugger (no code is loaded/flashed on the target)
- The program counter will be set to the entry point of the code and not at ‘main’
  - ‘Run->Go Main’ will run the target to ‘main’



Loading Symbols for Flashed Program



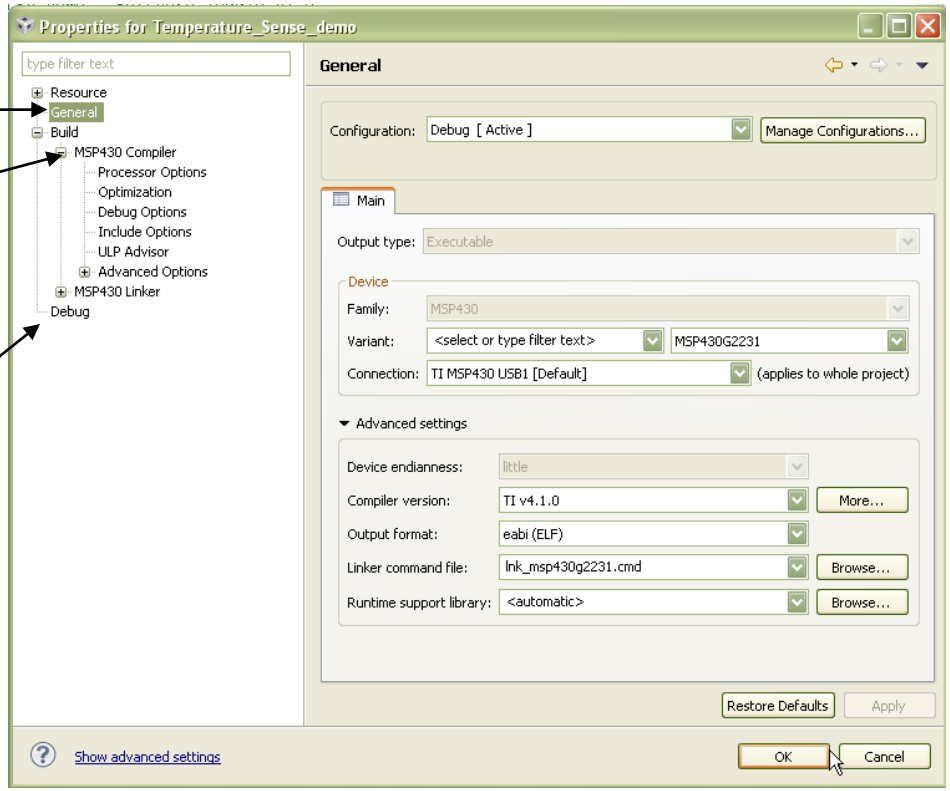
Changing Build Options

- Terminate the debug session
- Right click on the ‘Temperature_Sense_demo” project and select ‘Properties’

Device and high level settings

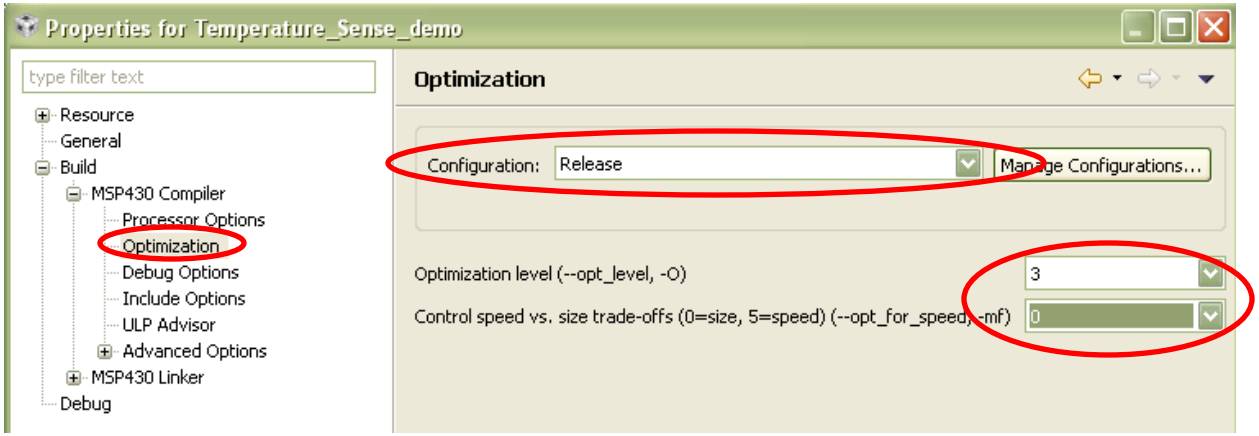
Compiler Options

Linker Options



Changing Build Options

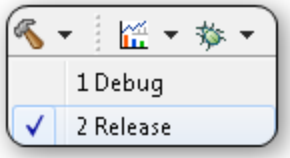
- Build options are set per build configuration
- Change your Configuration to “Release”
- Change the optimization settings
  - Go to the Basic Option Group
  - Change the optimization level to 3
  - Change speed vs size to 0 (optimize for size)



- Click OK

Changing Build Options

- Change the active configuration to ‘Release’
  - Right click on the Project
  - Select Build Configurations -> Set Active -> Release
- Build the project by clicking the build button
  - In the console view you will see that the Release configuration has been built
- You can also change the configuration and build it by click on the arrow beside the build button and selecting the configuration you want to build
  - Select ‘Debug’ and it will build the Debug configuration
  - The active configuration is indicated by the Checkmark

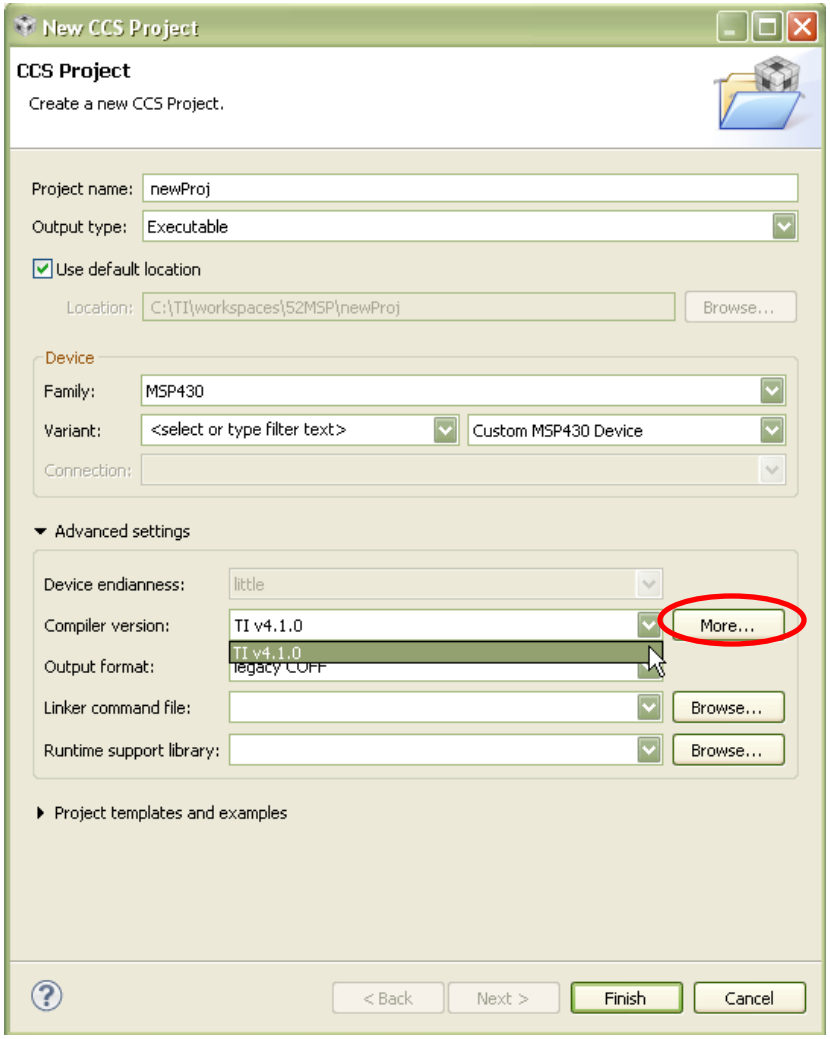


Compiler Versions

- Each project specifies which version of the compiler to use
  - Actually set on a per configuration level
  - I.e. Debug can use one version and release another
- When you download a new compiler if you want to use it you have to change the compiler version specified by your project
- CCS will allow you to select from all the compilers that it knows about on your computer
- When you install a new compiler via the Update Manager in CCS it will automatically know where the computer is
  - However you can also tell CCS where another version of the compiler is located

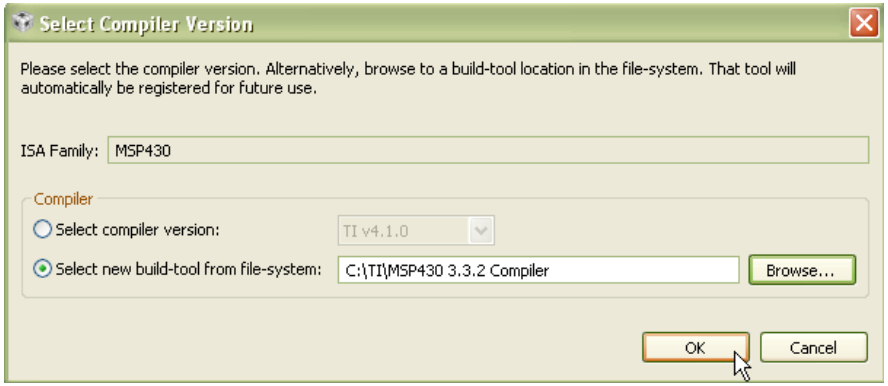
Compiler Versions

- Launch ‘New CCS Project’ Wizard
  - In ‘CCS Edit’ perspective, Project -> ‘New CCS Project’
- Fill in the fields as shown on the right
- Expand the ‘Advanced Settings’ and check the ‘Compiler version:’ dropdown menu
  - Note there is only one option (v4.1.0)
  - Select ‘More…’ button next to the ‘Compiler version:’ field



Changing the Compiler Version

- Browse to the path shown in the dialog and click OK

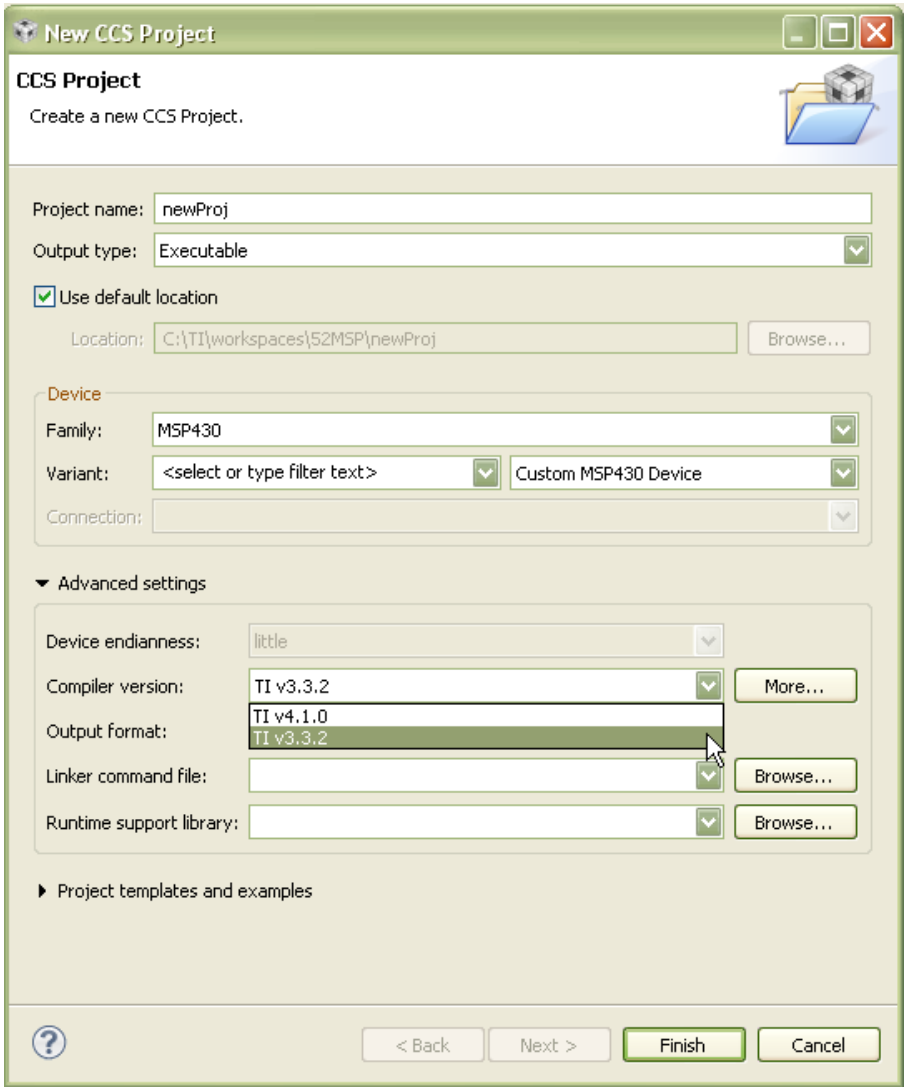


- CCS will determine what compiler is located there and select it for your active configuration. You will see that TI v3.3.2 is now specified

- Hit ‘OK’ to return to the New Project Wizard

Compiler Versions

- The ‘Compiler version:’ field will be updated to use version 3.3.2 of the TI MSP430 compiler
- Note how both v3.3.2 and v4.1.0 (default version that ships with CCSv5.2) now appear in the dropdown list
- The location of v3.3.2 is now known to CCS and will be available as an option for all projects using the same workspace
- Note that compiler versions can be changed for existing projects via
  - Right click on your project and select ‘Properties’
  - Click on ‘General’



SHARING PROJECTS

Sharing Projects

- Sharing “Simple” projects (all source/header files are contained in the project folder)
- Sharing “Linked file” projects and all source (project uses linked files)
- Sharing “Linked file” projects only (no source)
  - Only the project folder is shared (person receiving project already has all source)
  - Effort involves making the project “portable”
    - Eliminate absolute paths in the project
  - **This is the most common use case**

Sharing Projects – “Linked file” Projects

- USE CASE(S):
  - Wish to share (give) a project folder only. The person receiving the project folder already has a copy of the external source files
  - Wish to check the project folder/files into source control
- Most use cases involve sharing JUST the project folder instead of bundling all the source files
  - People will have their own local copies of the external source files
- Need to make the project portable to make sure the project is easily shared
- Portable projects avoid any absolute paths
- Ideal portable projects should be usable without modifying any of the project files
  - This is ideal for projects maintained in source control

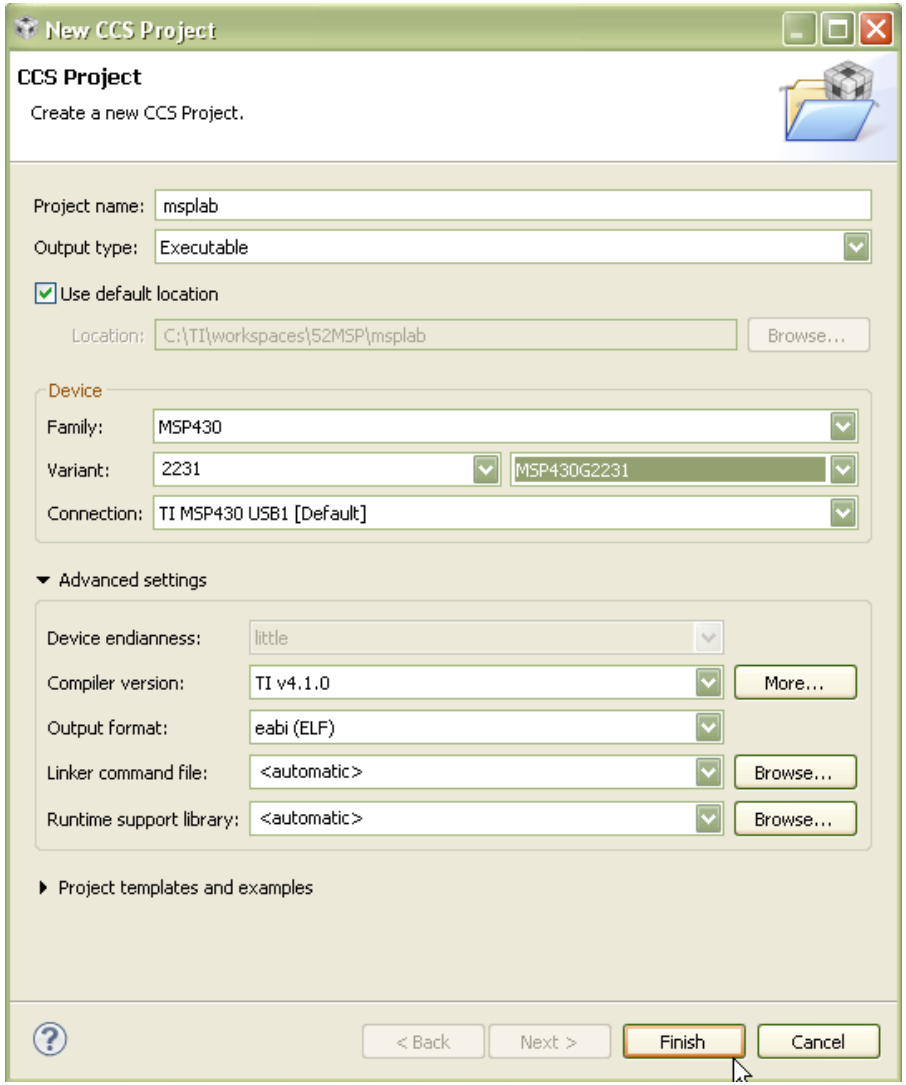
CREATING A PORTABLE PROJECT

Portable Projects: Briefing

- Key Objectives
  - Create a project that uses a linked source file
  - Make the project portable
- Tools and Concepts Covered
  - Linked Resources
  - Linked Resource Path Variables
  - Build Variables

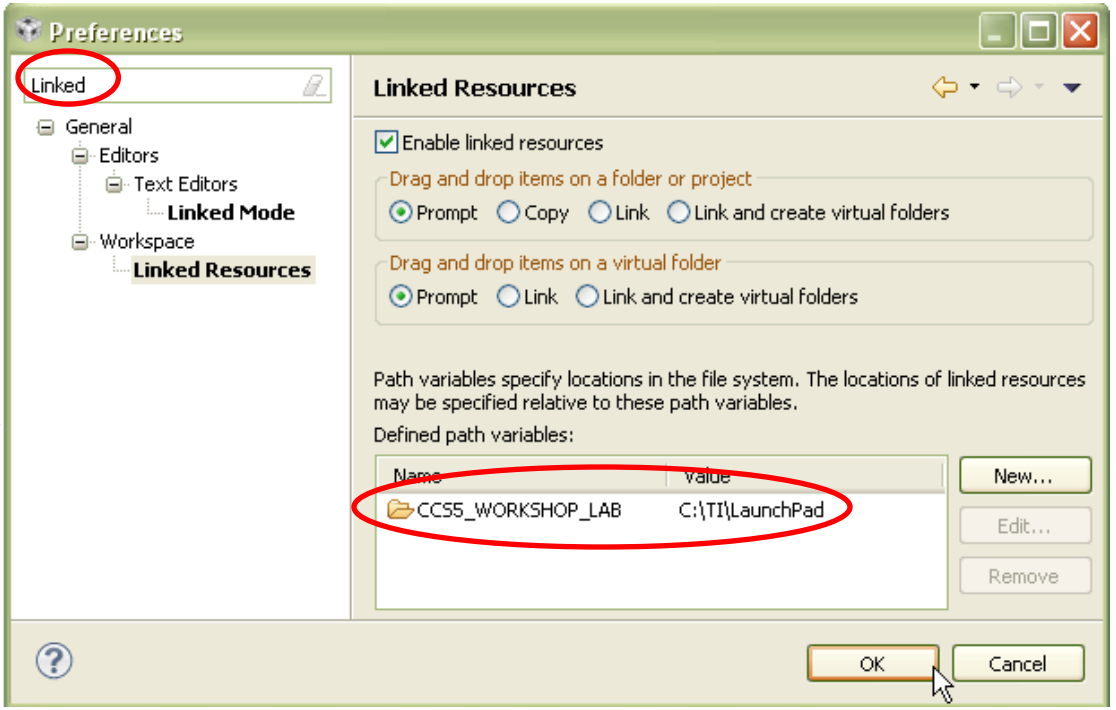
Create a New Project

- Launch ‘New CCS Project’ Wizard
  - Select ‘New Project’ from the Welcome page
- Fill in the fields as shown in the right
  - Use compiler version **v4.1.0**
- Select ‘Finish’ when done
- Generated project will appear in the Project Explorer view
- Remove the generated ‘main.c’ file from the project



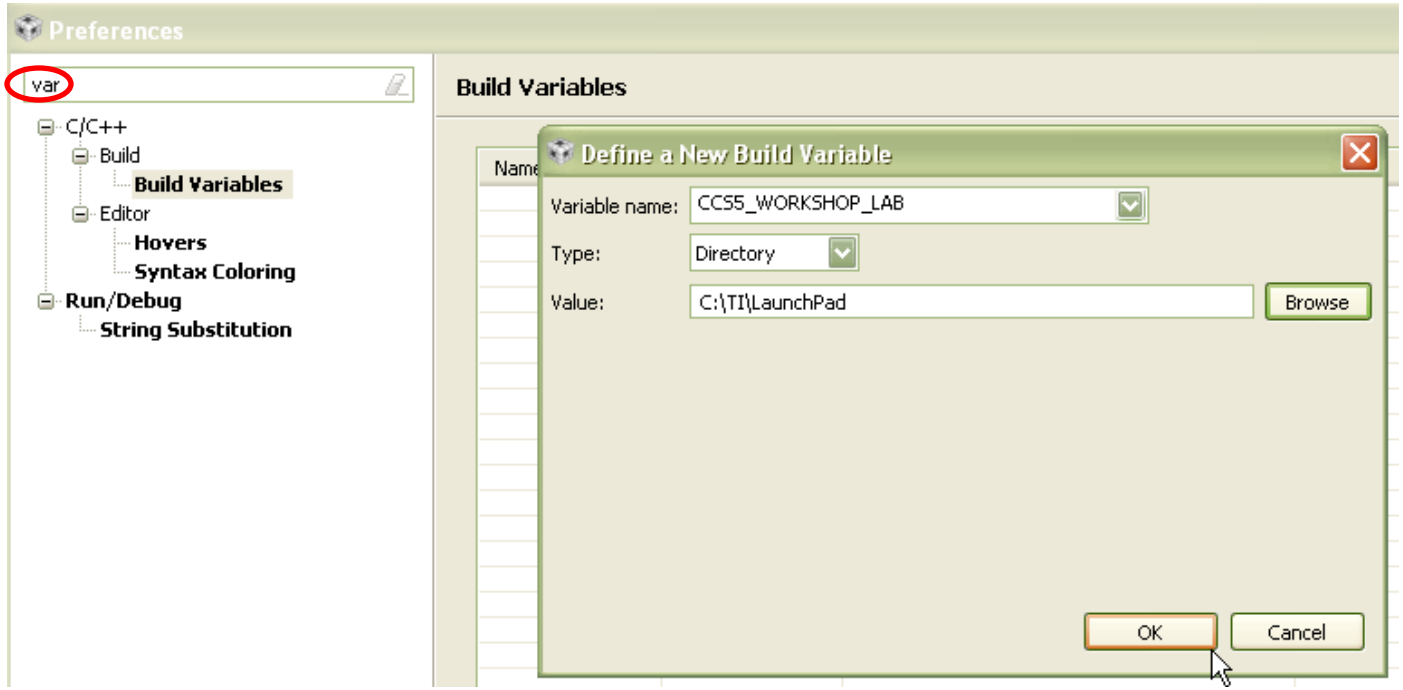
Create a Linked Resource Path Variable

- Open the workspace preferences
  - Window -> Preferences
- Go to the ‘Linked Resources’ preferences
  - Type ‘Linked’ in the filter field to make it easier to find
- Use the ‘New’ button to create a ‘Linked Resource Variable’ that points to the root location of the workshop LaunchPad labs
- Hit ‘OK’ when finished



Create a Build Variable

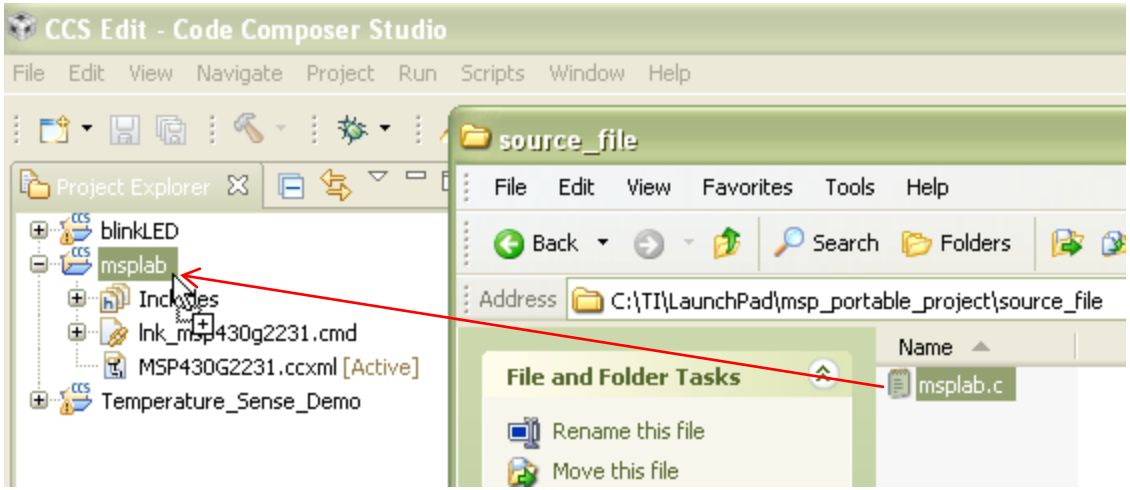
- Go to the ‘Build Variables’ preferences
  - Type ‘Variables’ in the filter field to make it easier to find
- Build Variables allow you to use variables in the project properties
  - Linked Resource variables are only used for linked files
- Use the ‘Add’ button to create a ‘Build Variable’ that points to the root location of the workshop LaunchPad labs
  - Set the ‘Type:’ to ‘Directory’ to browse to a directory
- Hit ‘OK’ when done



Link Source Files to Project

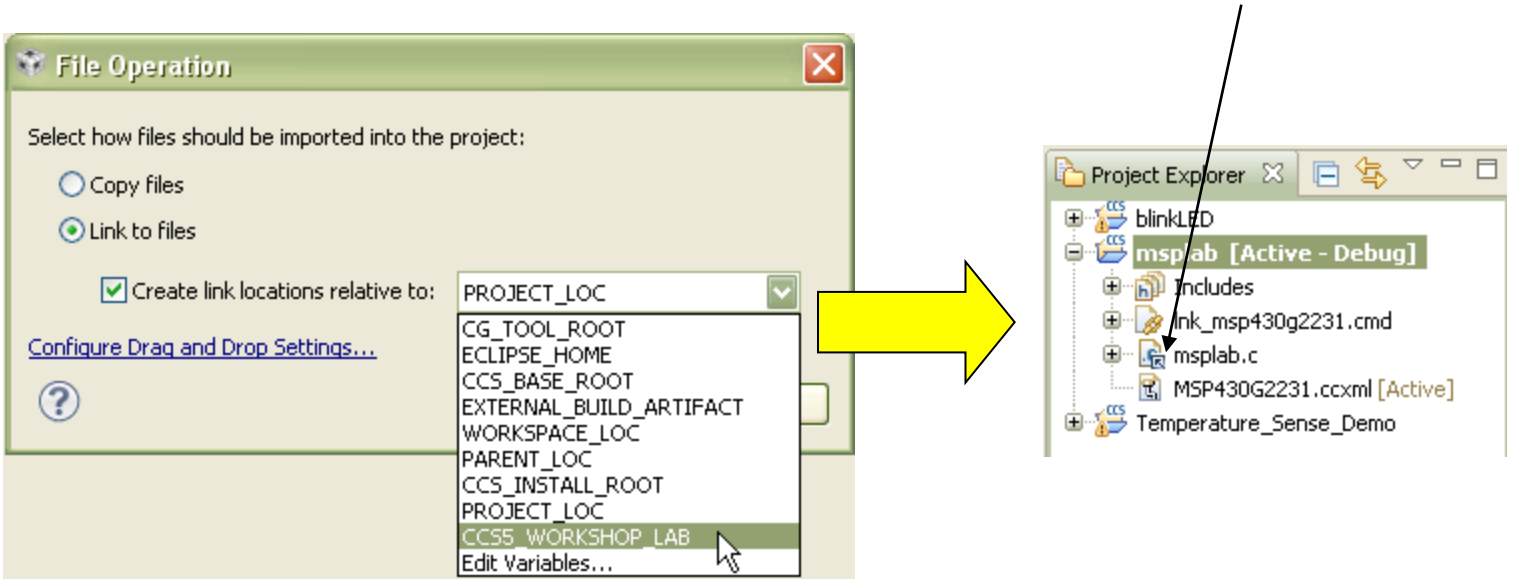
- Open Windows Explorer and browse to:
  - C:\TI\LaunchPad\msp_portable_project\source_file

- Drag and drop the ‘msplab.c’ file in to the ‘msplab’ project



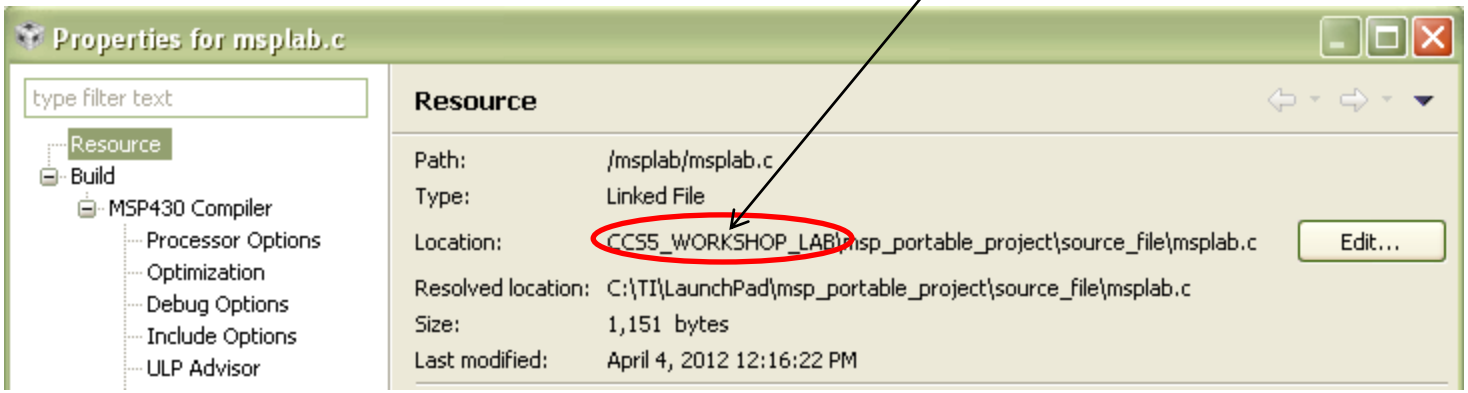
Link Source Files to Project

- A dialog will appear asking if you wish to Copy or Link the files:
  - Select ‘Link to files’
  - Select ‘Create link locations relative to:’
    - Use the new Linked Resource variable we created (CCS5_WORKSHOP_LAB)
  - Hit ‘OK’
- Files will now appear in the Project Explorer with the ‘link’ icon



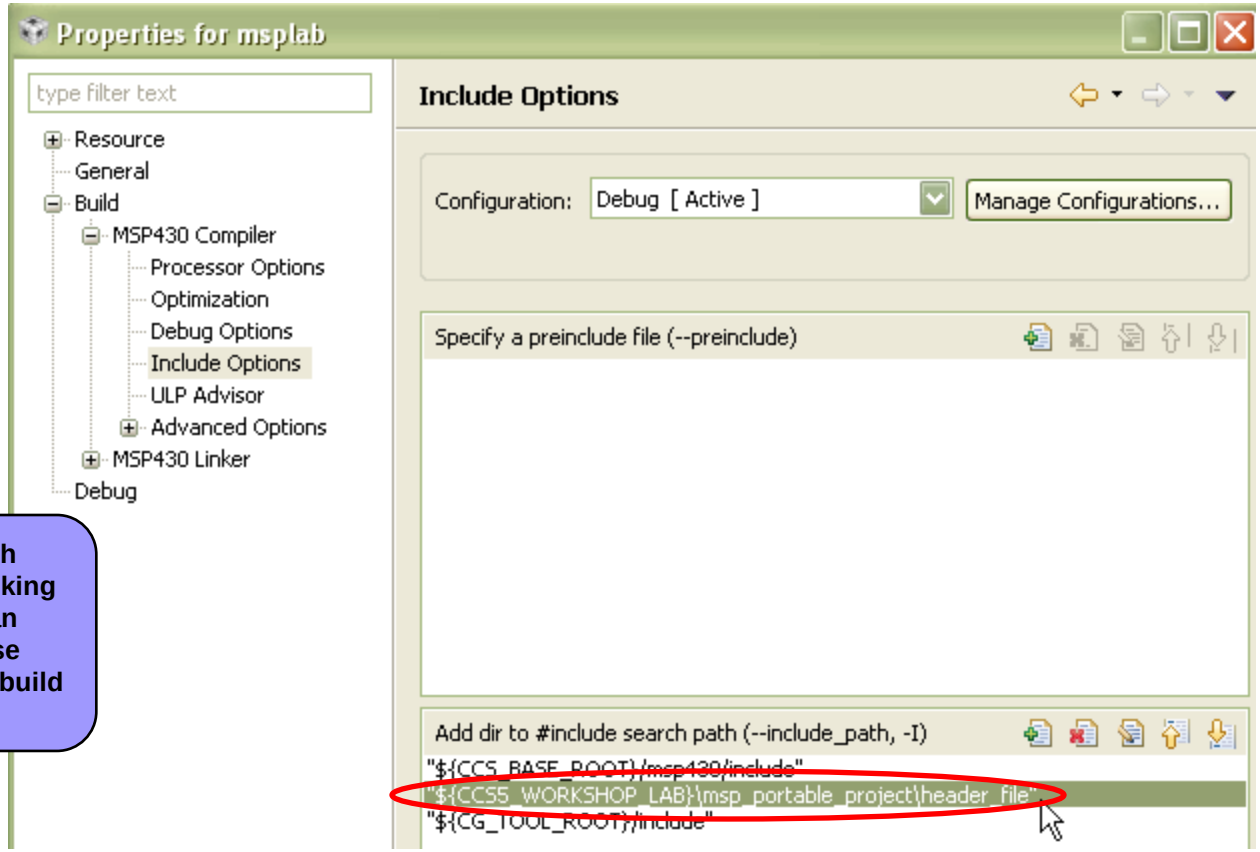
Link Files to Project

- Right-click on ‘msplab.c’ file and check the ‘Properties’
  - See how the ‘Location’ parameter references the Linked Resource Variable



Modifying Project Properties

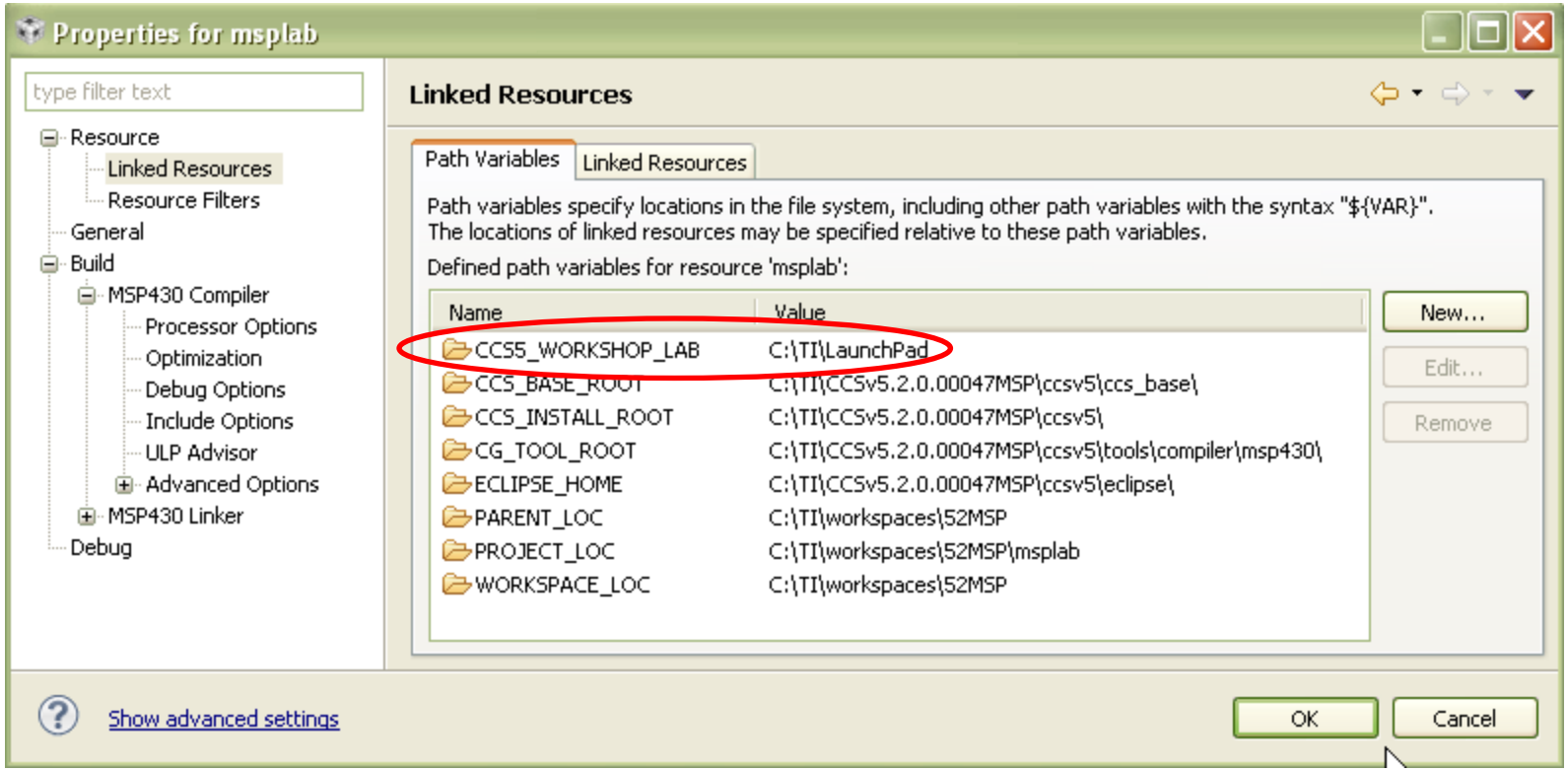
- Right-click on the project and select ‘Properties’
- In the compiler ‘Include Options’, add the following entries to the list of include search paths:
  - **${CCS5_WORKSHOP_LAB}\msp_portable_project\header_file**
- ‘${<BUILD VARIABLE>}’ is the syntax to use a Build Variable in the project properties



WARNING: Linked Resource Path Variables are only used when linking source files to a project. They can not be used for build options. Use Build Variables when modifying build options

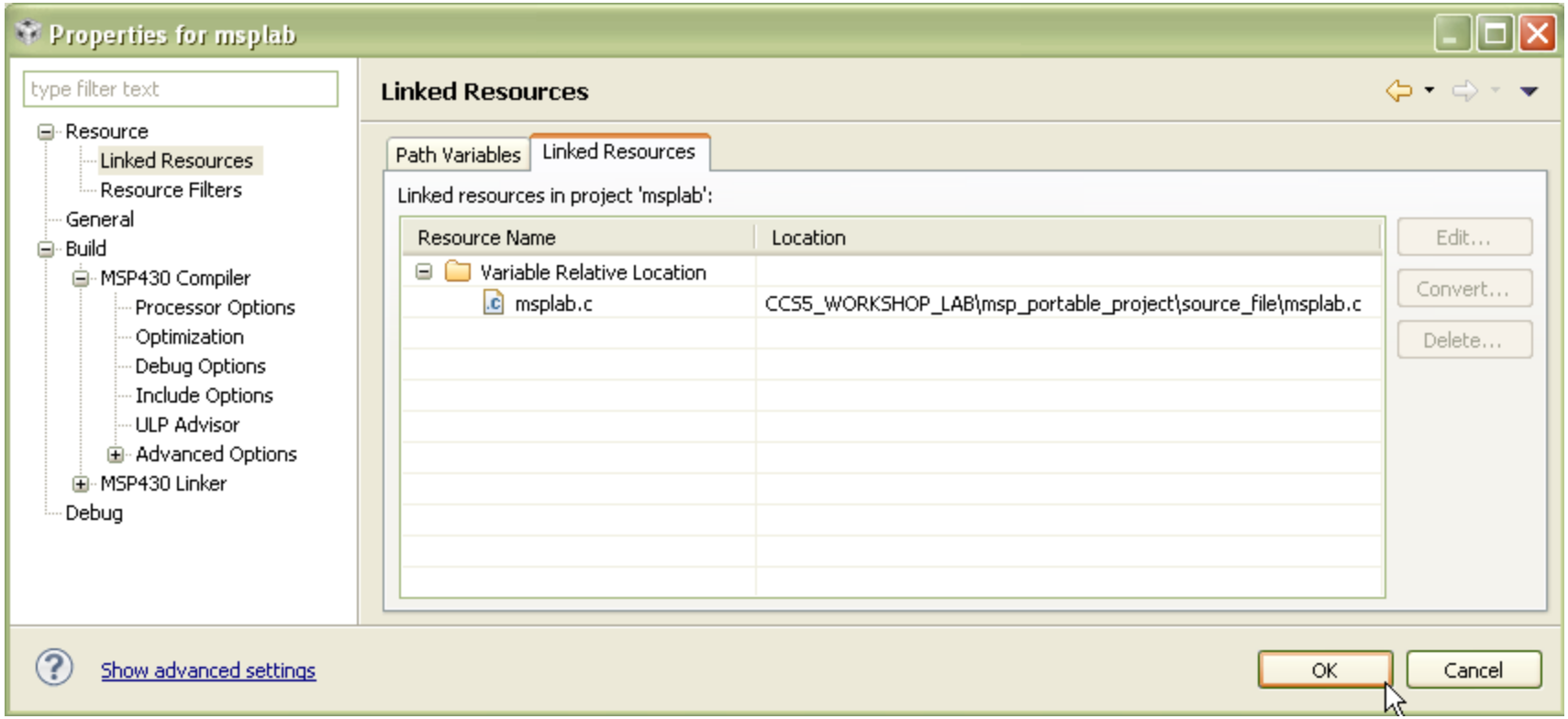
Project Properties

- Go to ‘Resource -> Linked Resources’ to see all the Linked Resource Path Variables that is available to the project
  - This will show all variables created at the project level and workspace level
- See the workspace level Linked Resource Path Variable that was created appears in the list
- Variables may be edited here but changes will only be recorded at the project level (stored in the project files)



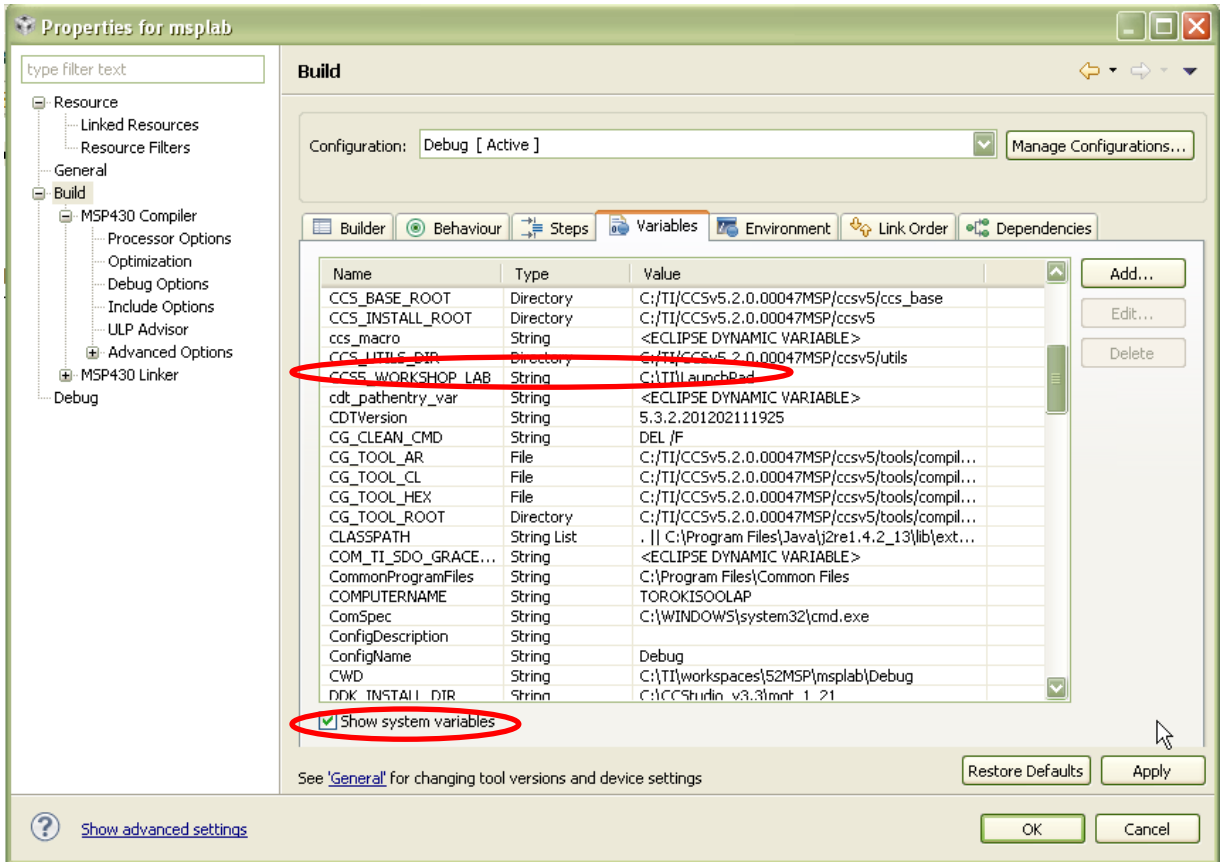
Project Properties

- The ‘Linked Resources’ tab will show all the files that have been linked to the project
  - It will sort them by files linked with a variable and files linked with an absolute path
- Links can be modified here with the ‘Edit…’ button
- Links can be converted to use an absolute path with the ‘Convert…’ button



Project Properties

- Go to ‘Build’ to see all the Build Variables that is available to the project
  - Only project level variables will be listed by default
  - Enable the “Show system variables” checkbox to see variables set at the workspace and system level
- See how the workspace level Build Variable that was created appears in the list



Project vs Workspace Level Variables

- The last few slides shows that ‘Linked Resource Path Variables’ and ‘Build Variables’ can be set at the project level
- This current lab set these variables at the workspace level
- What is the benefit of setting these variables at the workspace level instead of the project level?

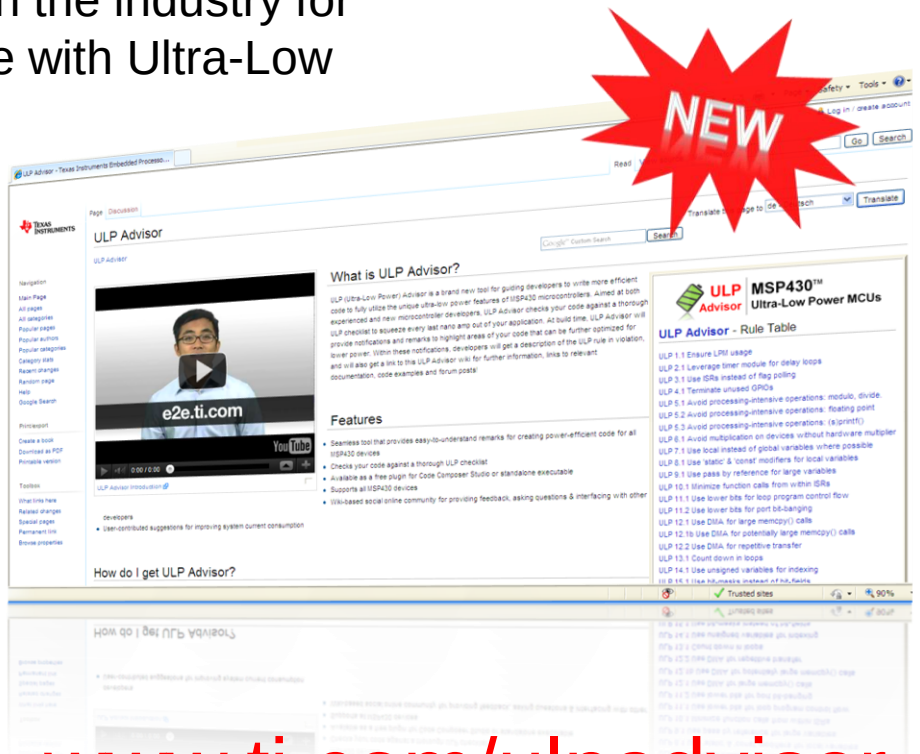
Project vs Workspace Level Variables

- What is the benefit of setting these variables at the workspace level instead of the project level?
  - All projects can reuse the same variable (set it once)
  - Do not need to modify the project!
    - This is important for projects checked into source control and to avoid constant checkouts so the project can be written to!

ULP (Ultra-Low Power) Advisor

ULP (Ultra-Low Power) Advisor

- Guide developers to write more efficient code to fully utilize the unique ULP features of MSP430.
- First true software-based teaching tool in the industry for enabling customers to always write code with Ultra-Low Power in mind.
- Offers unique and not-easily-discovered ULP tips and tricks.
- Customers will understand where they can improve their code line-by-line for ULP performance.
- BETA available now



www.ti.com/ulpadvisor

ULP Advisor | Integrated SW

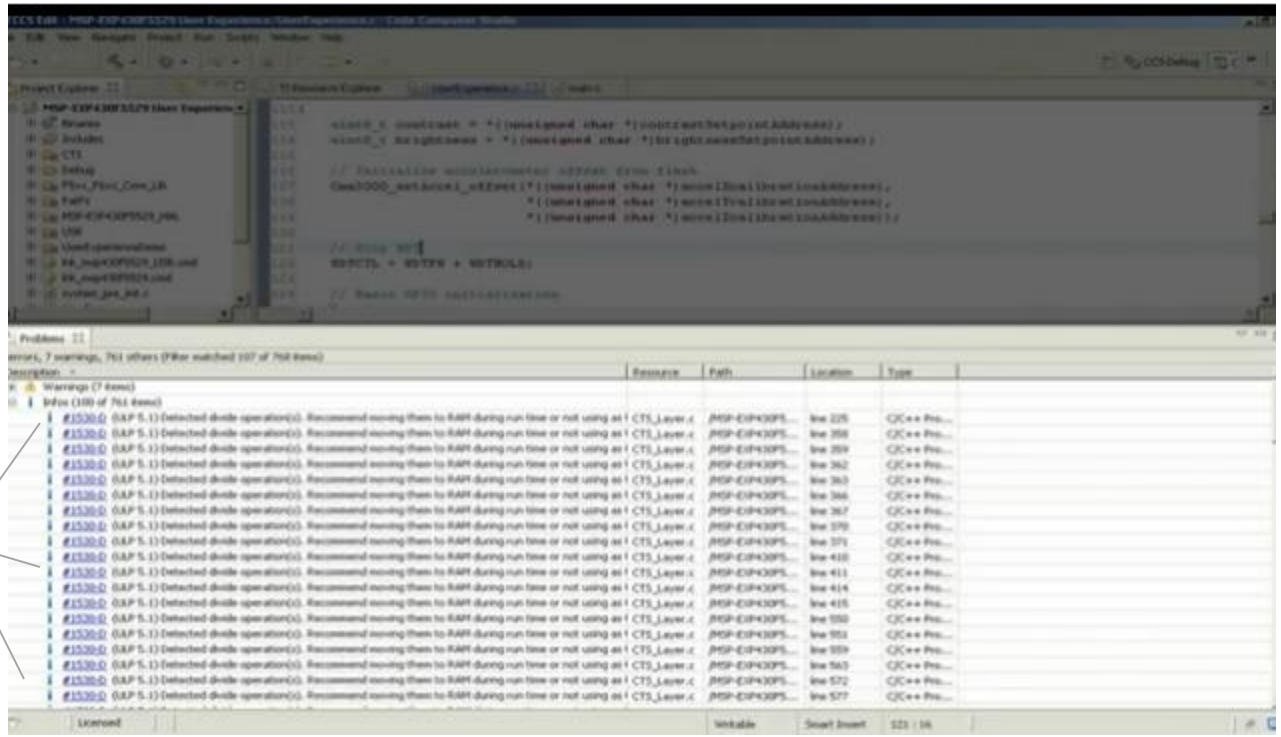
- IDE Integration
  - Comes with Code Composer Studio v5.2
  - Pre-installed as a plug-in for IAR (coming April)

- Stand-alone version coming soon
  - Support for other IDEs, such as Open Source MSPGCC
- Automatically enabled for new projects

ULP Advisor | Operation

- Activated when compiling a project
- Scans through project's code files against a ULP checklist
- Highlights ULP violations and reports in **Problem View**

ULP Remarks



ULP Advisor | Wiki

- Each remark provides:
  - Brief suggestion
  - Hyperlink to wiki with detailed information on the rule
- Customers can find:
  - Background information, why the rule is important
  - What ULP Advisor checked to issue this remark
  - Now that I know why and whether or not:
    - what is wrong with my code or
    - understand why my code intentionally consumes such power
  - Code examples
  - Videos
  - Tutorials
  - Links to relevant documentation
  - Leverages TI's huge e2e online community.

ULP Demo and Lab

Import and Build ULP Demo Project

- Import ULP Demo project into the workspace
  - Import -> Existing CCS Eclipse Project
  - Project is located in:
    - C:\ti\LaunchPad\ulp_demo
- Explore the one source file ‘ulp_demo.c’
- Build it

ULP Diagnostics

- ULP Diagnostics appear in problems view
- Maximize ‘Problems’ view
- Click on link to open advice window for that diagnostic
- Experiment with the links
- Can position Advice window anywhere

Problems

0 errors, 18 warnings, 4 others

Description	Resource	Path	Locatio
Warnings (18 items)			
Infos (4 items)			
#10371-D (ULP 1.1) Detected no uses of low power mode state changes using LPMx or _bis_SR_register() or __low_po	ulp_demo		
#1535-D (ULP 8.1) variable "const_local" is used as a constant. Recommend declaring variable as either 'static const' or '	ulp_demo.c	/ulp_demo	line 47
#1539-D (ULP 11.1) Loop program control flow variable "flags_var" compared against higher bits. Recommend compare	ulp_demo.c	/ulp_demo	line 72
#1544-D (ULP 13.1) Detected loop counting up. Recommend loops count down as detecting zeros is easier	ulp_demo.c	/ulp_demo	line 88

Click here!

Advice

ULP Advisor > Rule 8.1 Use 'static' & 'const' modifiers for local variables

What it means

In an MSP430 C function, local variables without any modifiers are dynamically allocated upon each function call. This requires additional code & RAM space and the impact increases depending on the function call frequency. On the other hand, when declared as 'static', the variables are only generated once and remain available throughout the lifetime of the application. This minimizes the amount of code needed to re-allocate/re-initialize the variables every time the function is invoked. Alternatively, when the const modifier is used, the variable is stored as data in flash as part of the function, hence requiring no further re-allocation for each function entry.

Risks, Severity

Using local variables without 'static' or 'const' modifier requires additional code execution to reallocate & reinitialize the variables each time the function is invoked.

Why it is happening

The project code contains a function with local variables that are not but can be declared with the 'static' and/or 'const' modifiers.

Remedy

ULP Advisor | MSP430™ Ultra-Low Power MCUs

ULP Advisor - Rule Table

ULP 1.1 Ensure LPM usage
ULP 2.1 Leverage timer module for delay loops
ULP 3.1 Use ISRs instead of flag polling
ULP 4.1 Terminate unused GPIOs
ULP 5.1 Avoid processing-intensive operations: modulo, divide.
ULP 5.2 Avoid processing-intensive operations: floating point
ULP 5.3 Avoid processing-intensive operations: (s)printf()
ULP 6.1 Avoid multiplication on devices without hardware multiplier

ULP 7.1 Use local instead of global variables where possible
ULP 8.1 Use 'static' & 'const' modifiers for local variables
ULP 9.1 Use pass by reference for large variables
ULP 10.1 Minimize function calls from within ISRs
ULP 11.1 Use lower bits for loop program control flow

Fix Rule 8.1

- For function ulp_demo_rule_8_1
- Static and const variables are preferred over plain local variables
  - Add `static const` before `const_local`
- Make change directly
- Or cheat …
  - Change `#define BREAK_RULE_8_1` to 0
- Build
- Diagnostic about rule 8.1 no longer in problems view

Rule 8.1 – After Fix

```
39
40 //-----------------------------------------------------------------------------
41 // Demonstrate rule 8.1 - Use 'static' and 'const' modifiers for local
42 // variables
43 //-----------------------------------------------------------------------------
44 int ulp_demo_rule_8_1()
45 {
46     static const int const_local = 10;
47     return const_local;
48 }
49
```

Problems

0 errors, 18 warnings, 3 others

Description

- Warnings (18 items)
- Infos (3 items)
  - #10371-D (ULP 1.1) Detected no uses of low power mode state changes using LPMx or _
  - #1539-D (ULP 11.1) Loop program control flow variable "flags_var" compared against hi
  - #1544-D (ULP 13.1) Detected loop counting up. Recommend loops count down as detect

Fix Rule 11.1

- For function ulp_demo_rule_11_1
- Change #define F1 … F4 to 1, 2, 4, 8
  - MSP430 can use those constant values cheaply
- Make change directly
- Or cheat …
  - Change `#define BREAK_RULE_11_1` to 0
- Build
- Diagnostic about rule 11.1 no longer in problems view

Rule 11.1 – After Fix

```
50 //-----------------------------------------------------------------------------
51 // Demonstrate rule 11.1 - Use lower bits for loop program control flow
52 //-----------------------------------------------------------------------------
53 #define F1 0x01
54 #define F2 0x02
55 #define F3 0x04
56 #define F4 0x08
57
58 unsigned flags_var = 0;
59 void ulp_demo_rule_11_1()
60 {
61    while (flags_var & (F2 | F3))
62       do_some_processing();
63 }
```

Problems

0 errors, 18 warnings, 2 others

Description

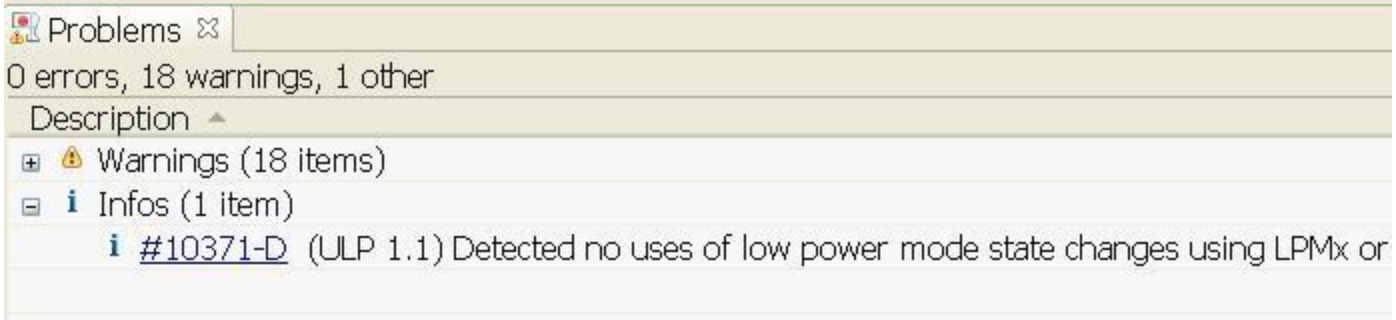
- Warnings (18 items)
- Infos (2 items)
  - #10371-D (ULP 1.1) Detected no uses of low power mode state changes using LPMx
  - #1544-D (ULP 13.1) Detected loop counting up. Recommend loops count down as d

Fix Rule 13.1

- For function ulp_demo_rule_13_1
- Change to loop that counts from N-1 to 0
  - Comparisons against 0 are cheaper
- Make change directly
- Or cheat …
  - Change `#define BREAK_RULE_13_1` to 0
- Build
- Diagnostic about rule 13.1 no longer in problems view

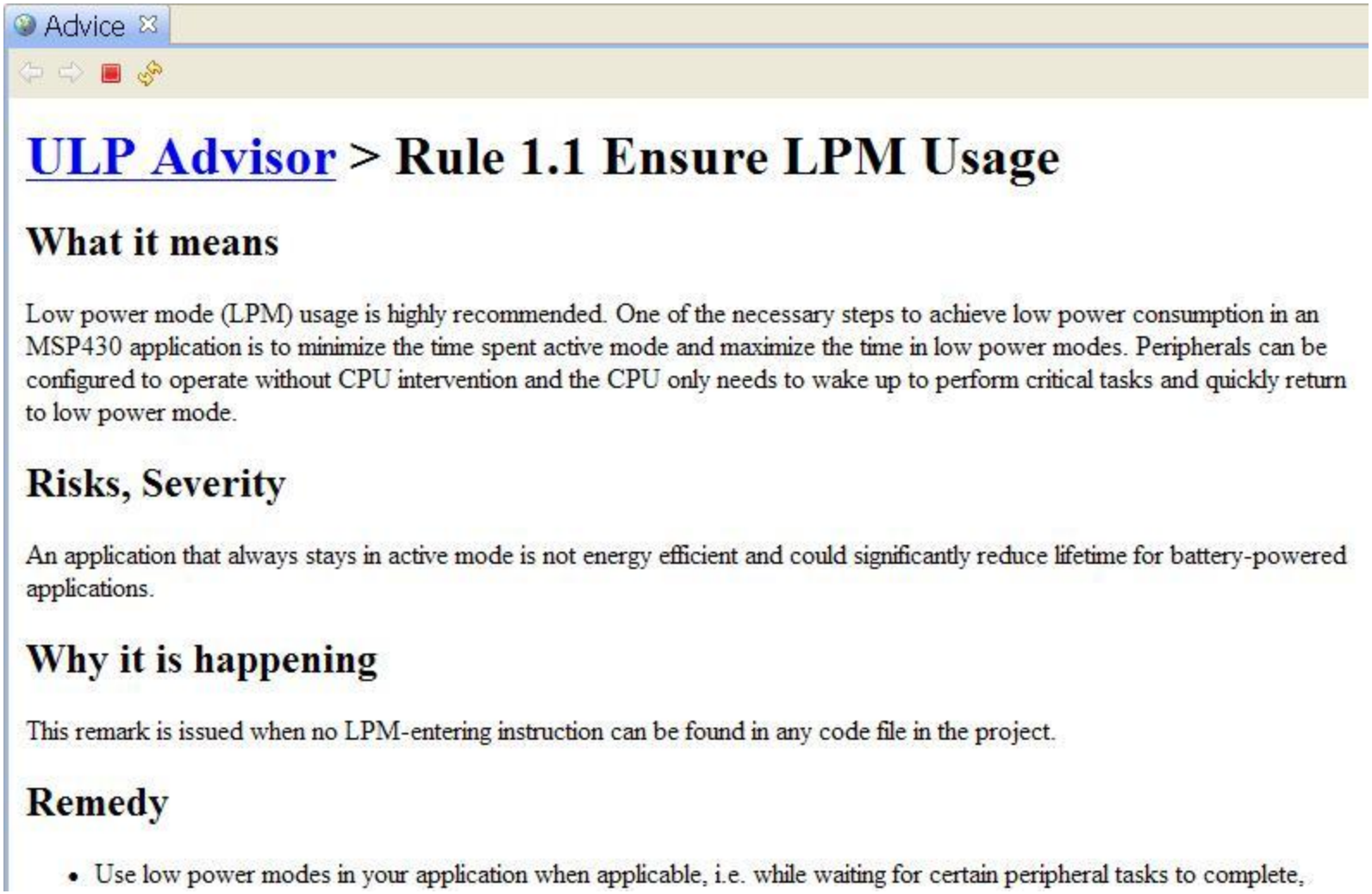
Rule 13.1 – After Fix

```
69 //-----------------------------------------------------------------------------
70 // Demonstrate rule 13.1 - Count down in loops
71 //-----------------------------------------------------------------------------
72 void ulp_demo_rule_13_1(int input)
73 {
74    int i;
75    for (i = input-1; i >= 0; i--)
76        do_some_processing();
77 }
78
```



- Rule 1.1 problem remains
- Click the link to see related advice

Importance of Rule 1.1



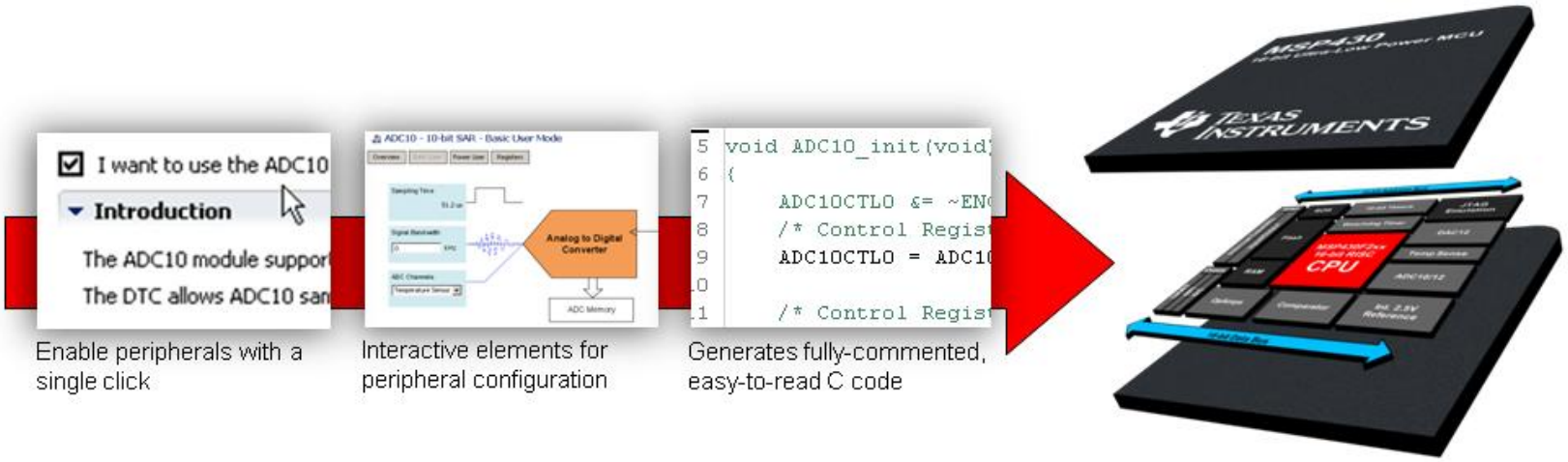
Importance of Rule 1.1

- Remain in low power mode (LPM) as long as possible
- Cannot run any code while in LPM
- Come out of LPM with an interrupt
- Primary technique for conserving power
- More details beyond scope of this demo

Grace

What is Grace?

- Grace is a tool that allows you as an MSP430 developer to generate the peripheral's set up code within minutes



Enable peripherals with a single click

Interactive elements for peripheral configuration

Generates fully-commented, easy-to-read C code

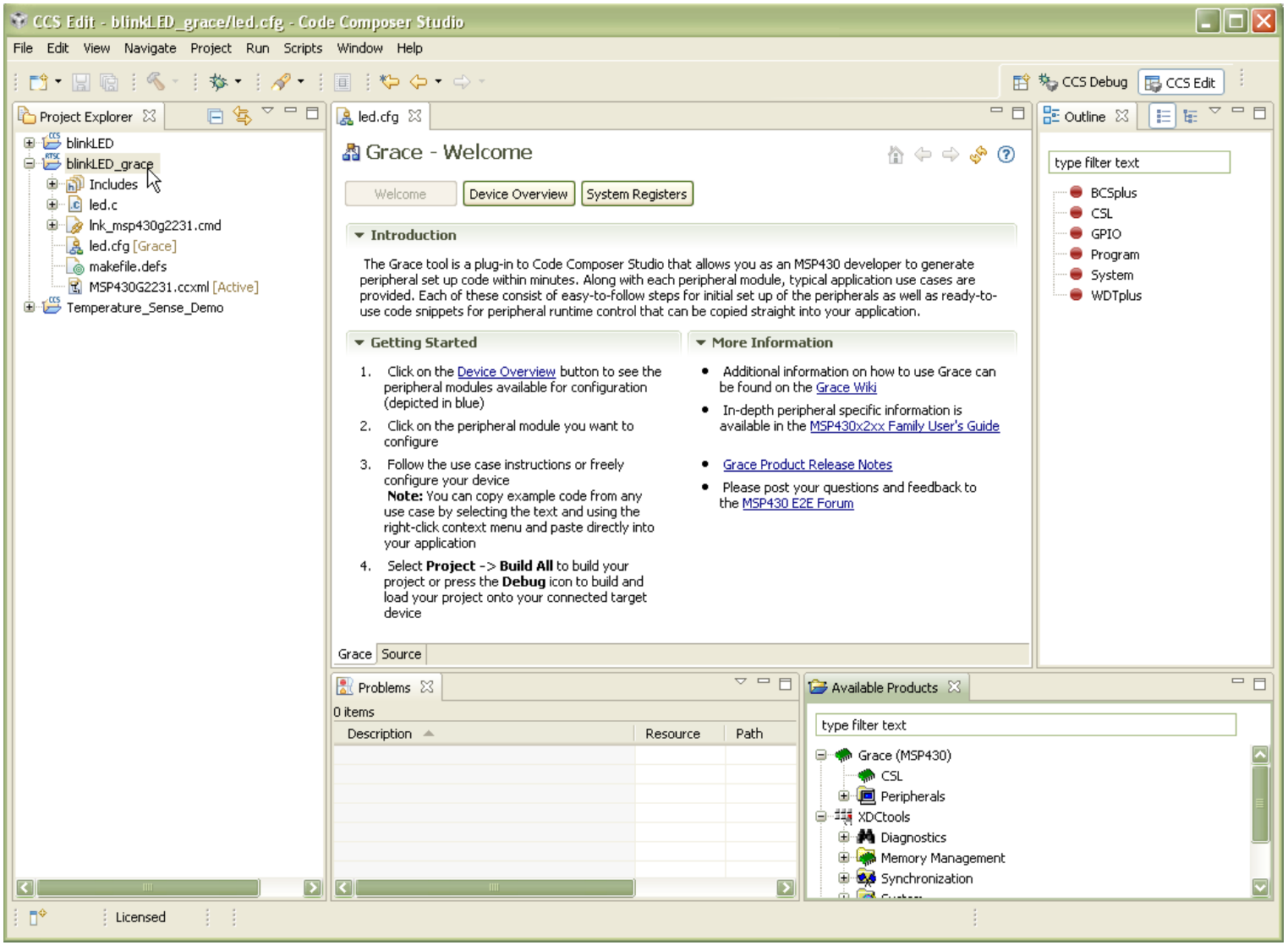
Grace

- Fully harness MSP430 MCUs integrated analog and digital peripherals with Grace Software
- Grace software supports all MSP430F2xx and G2xx Value Line MCU devices
- Seamless integration into TI's eclipse-based Code Composer Studio v5 software
  - Grace comes with Code Composer Studio v5

Create a New (Grace) Project

- Start the Project Wizard
  - Select ‘New Project’ on the Welcome page
- Fill in the fields as shown on the right
  - Select the ‘**Blink LED from the CPU**’ template under ‘**Grace Examples**’
- Select ‘Finish’ when done

Grace – Configuration File



Grace – Building/Running the Example

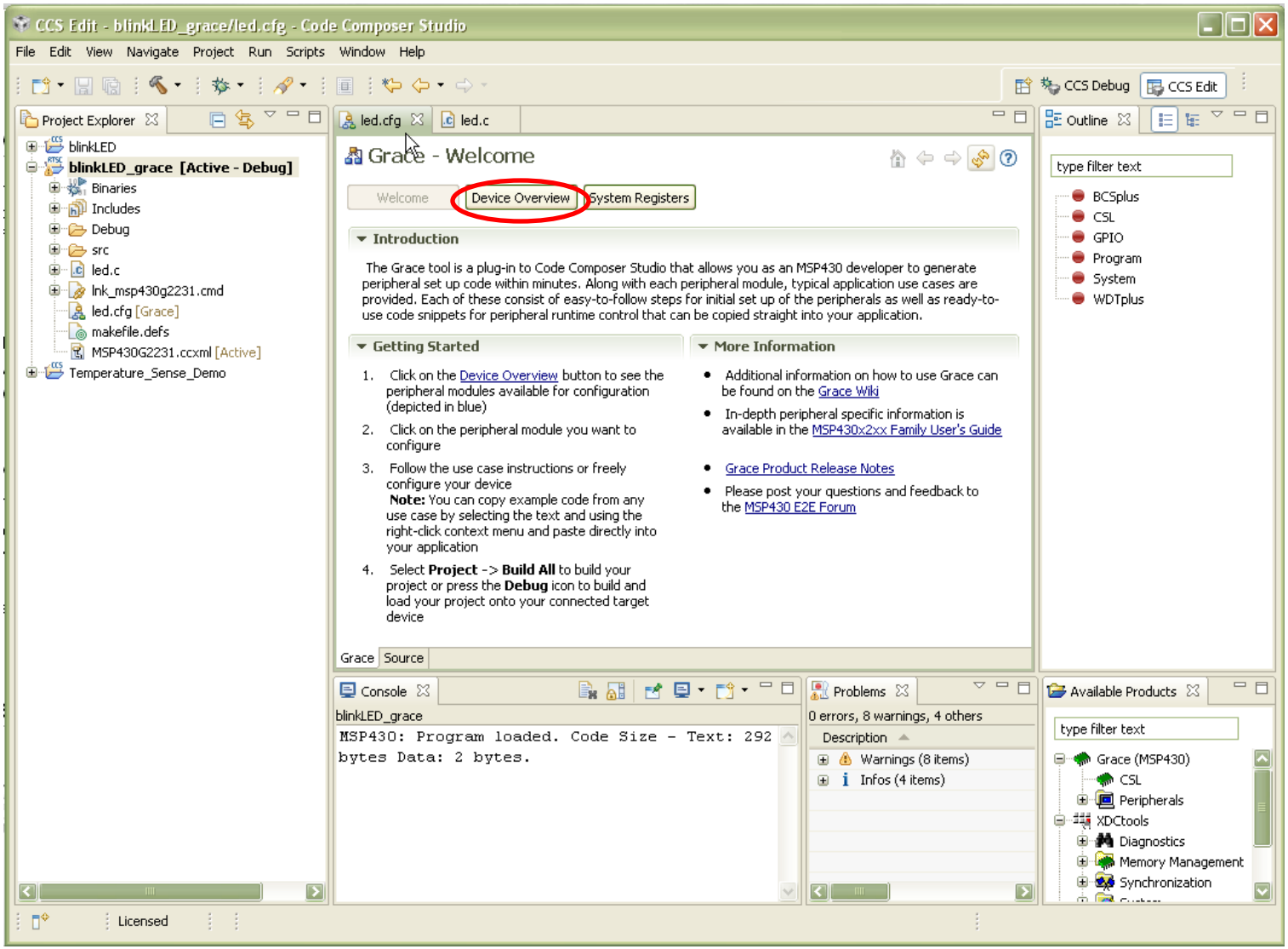
- Build and load the Grace example using the ‘Debug’ button
  - HINT: Make sure the new grace example project is in focus
- Run the example
- LED1 should be blinking
  - Note the blink rate
- Terminate the debug session and return to the ‘CCS Edit’ perspective

Grace – Modifying the Example

- Let's change the blink rate. How?
  - Modify the source code… or..
  - Tweak register values… or..
  - **Use Grace to change the clock value!**

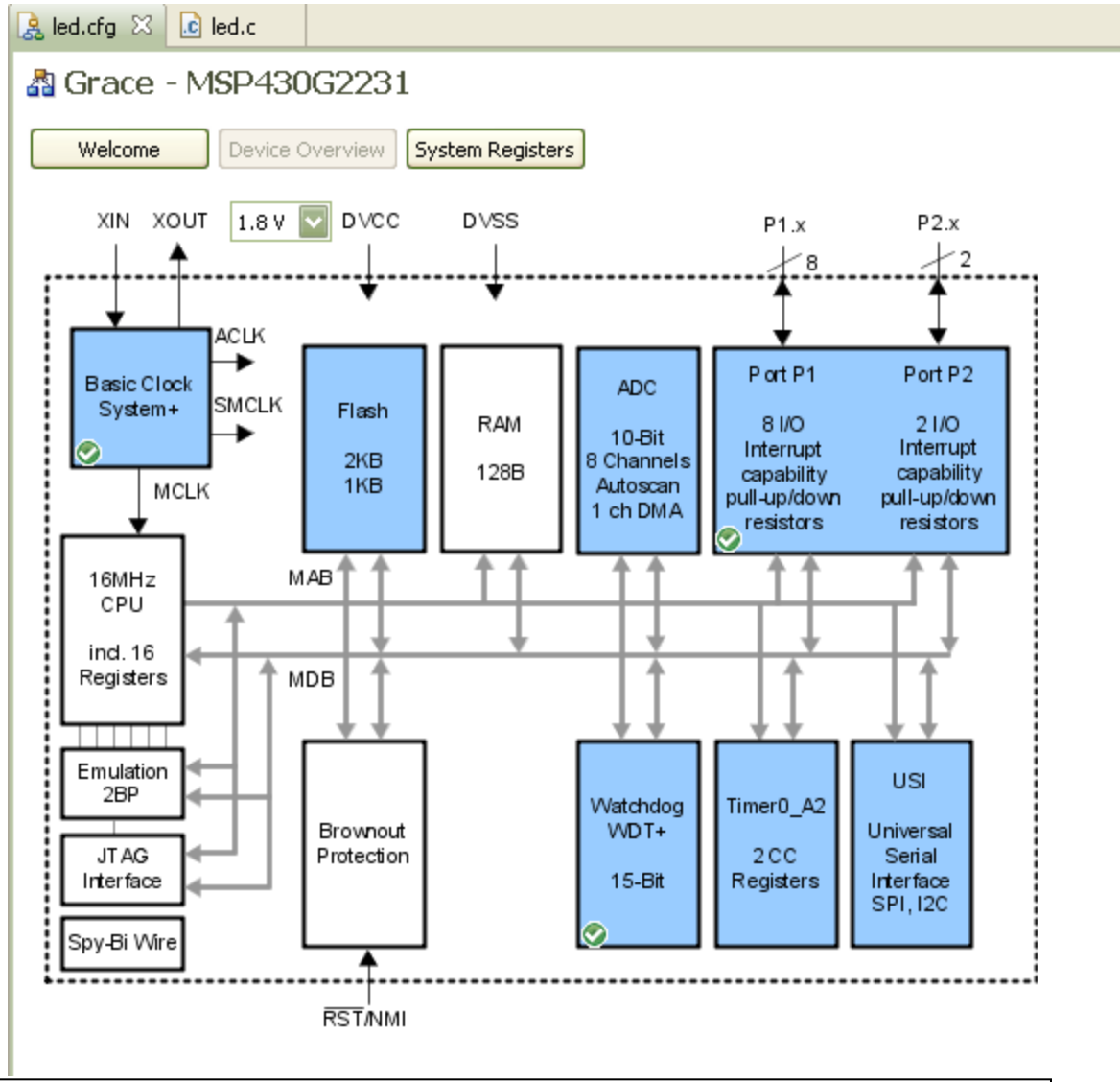
- In the 'led.cfg' file, select the 'Device Overview' button to see the peripheral modules available for configuration

Grace – Modifying the Configuration



Grace – Device Overview

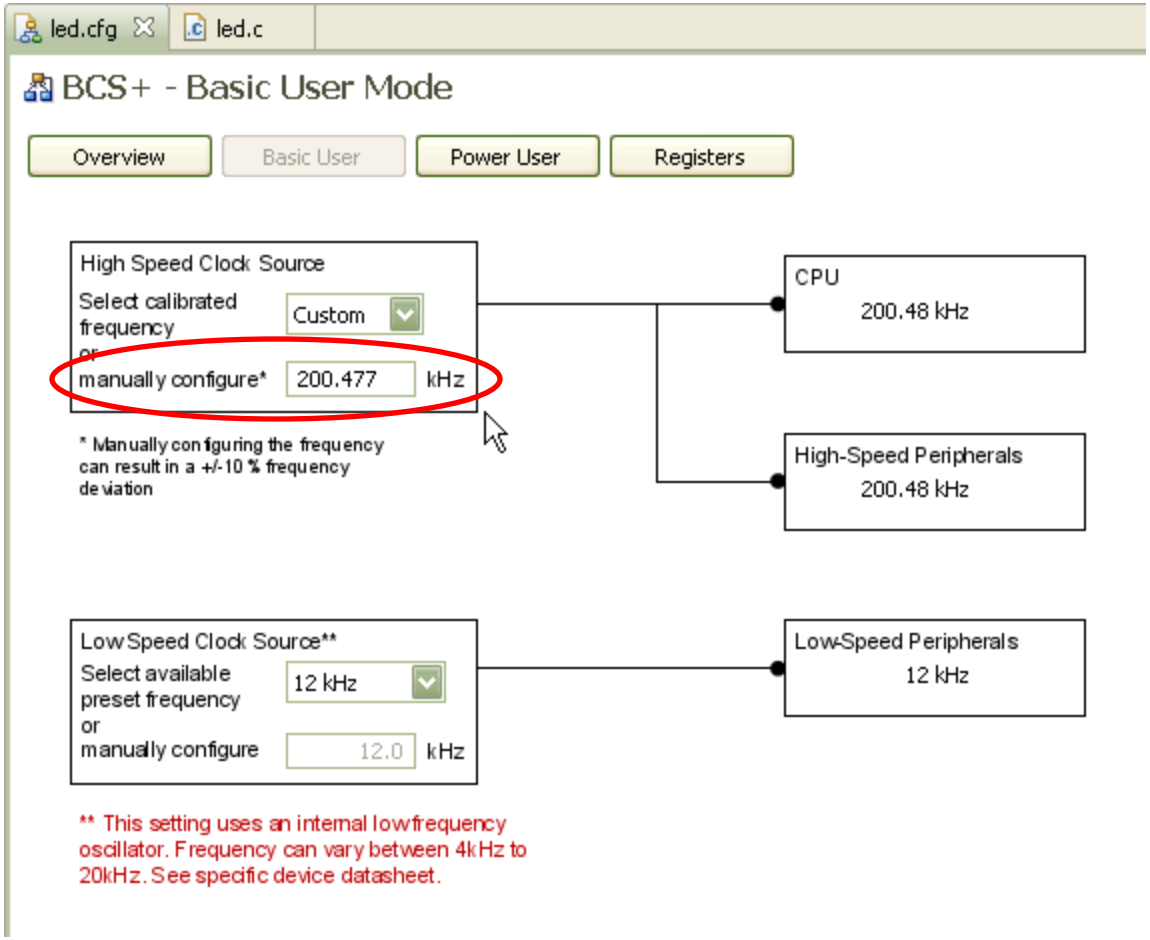
- Interactive block diagram of the peripheral modules available for configuration
- Select the ‘Basic Clock System+’ (BCS+) peripheral to configure it
  - NOTE: The name of the peripheral can vary depending on the device
    - **G2553**: ‘Oscillators Basic Clock System+’



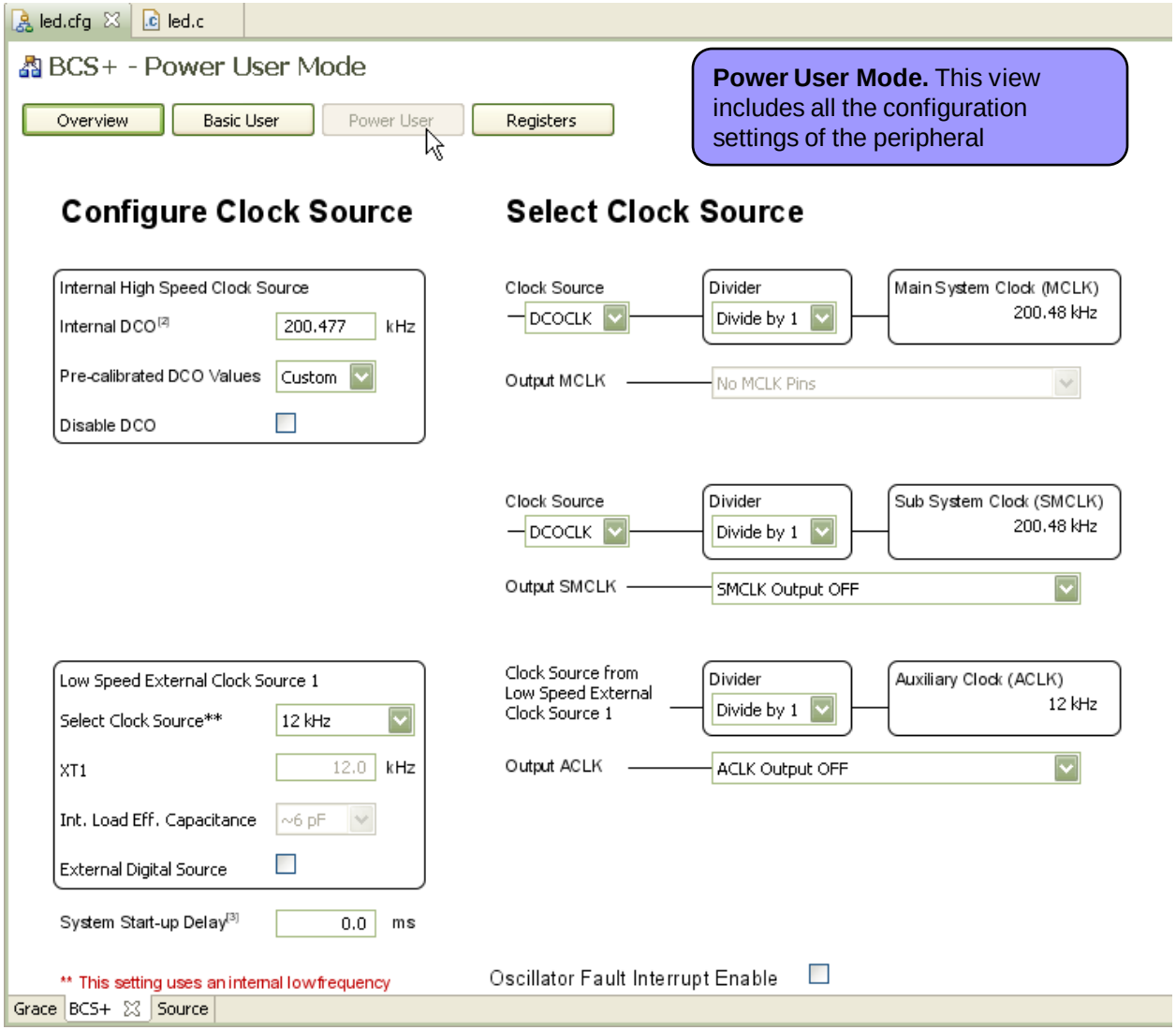
Grace – BCS+ Overview

Grace – BCS+ Basic User Mode

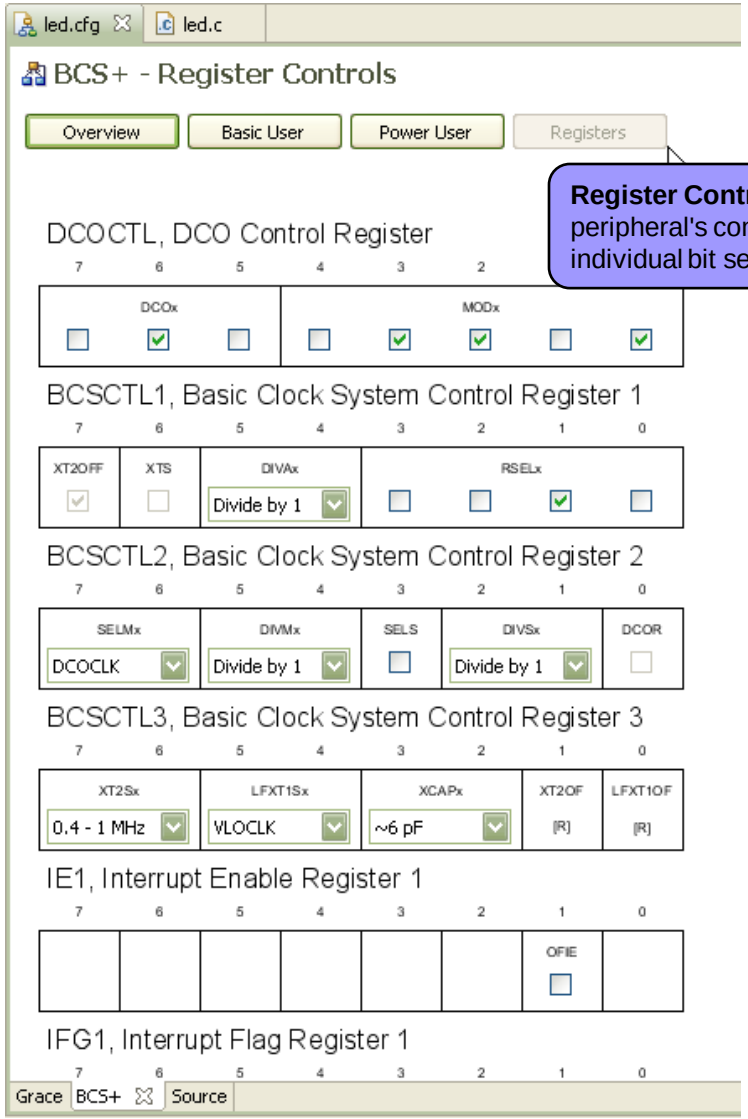
- This view includes most of the configuration settings that most users will need
- Change the frequency of the ‘High Speed Clock Source’ from 1 MHz to 200 kHz
- Save the led.cfg file



Grace – BCS+ Power User Mode



Grace – BCS+ Register Controls



Register Controls. This view depicts the peripheral's control registers and individual bit settings.

Grace

- Rebuild and reload the Grace example using the ‘Debug’ button 
  - HINT: Make sure the new Grace example project is in focus
- Run the example 
- LED1 should be blinking
  - Note that new blink rate is 5x slower