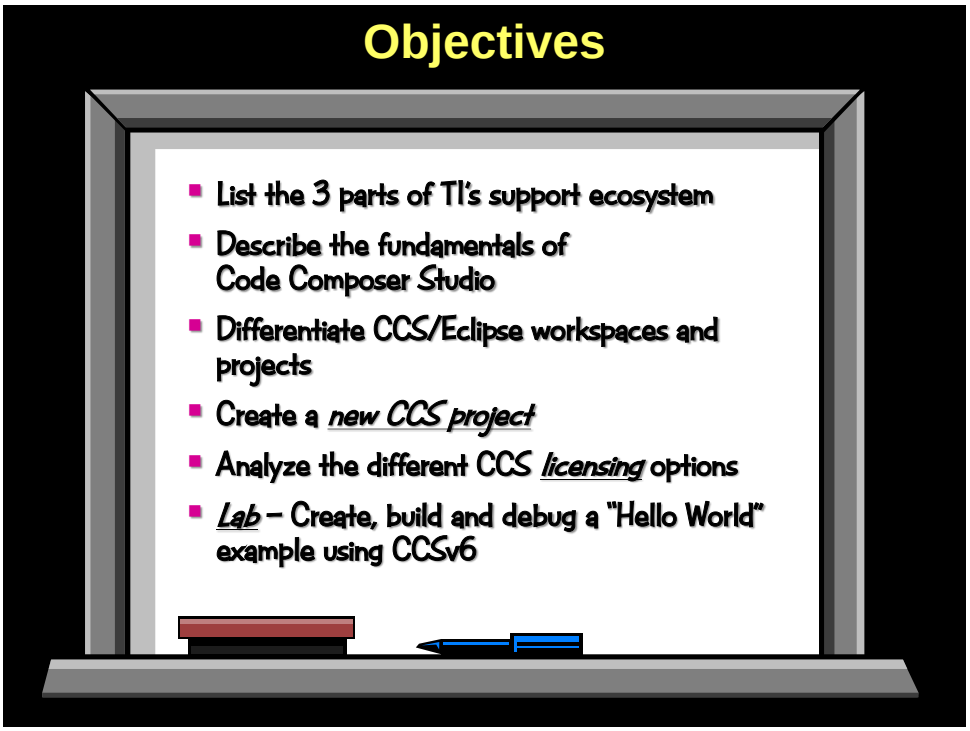
# Programming in C with CCS

## Introduction

This chapter will introduce you to Code Composer Studio (CCS).

In the lab, we will build our first project using CCS and then experiment with some useful debugging features. Even if you have some experience with CCS, we hope that you will find exercise to be a good review – and in fact, that you might even learn a few new things about CCS that you didn't already know.

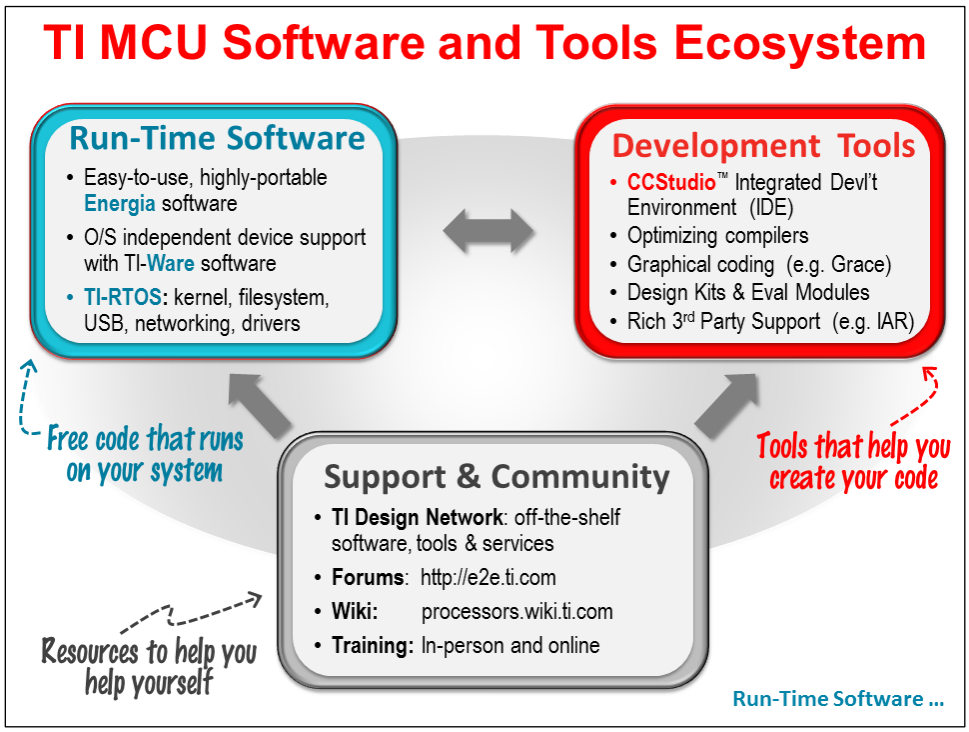
## Learning Objectives



# Chapter Topics

# TI Support Ecosystem

TI's goal is to provide an entire ecosystem of tools and support. Development tools, like Code Composer Studio are just the starting point; then add in software libraries that run on your target processor as well as wiki's and support forums.



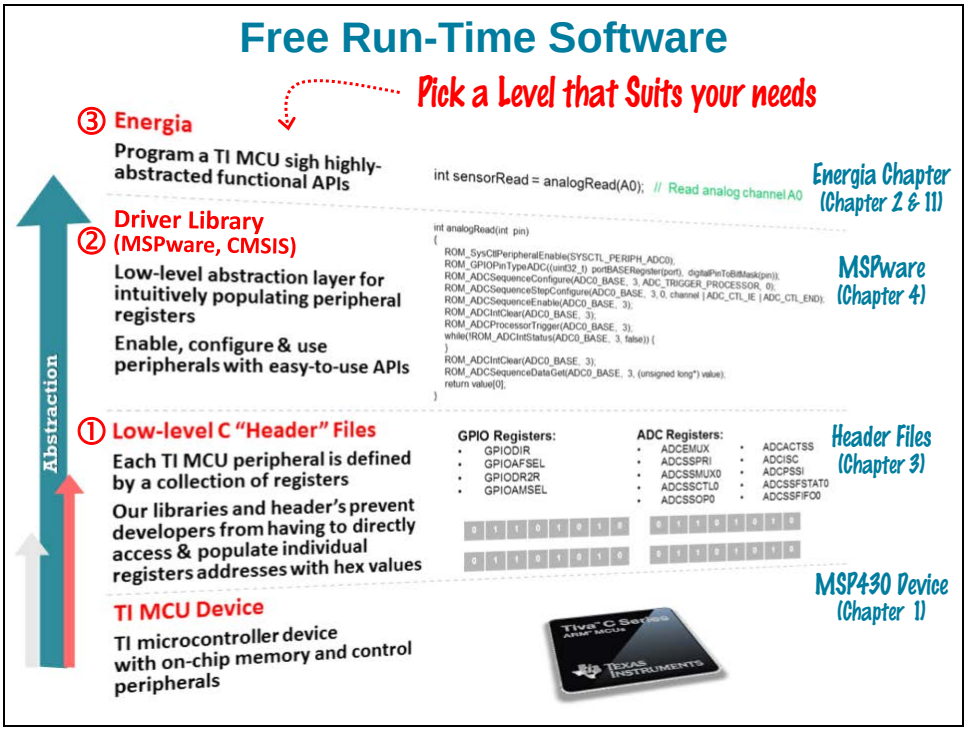

We'll take a brief look at all three parts of the Ecosystem:

- Run-Time Software
- Development Tools

Support and Community was examined back in Chapter 1.

# Run-Time Software

The MSP microcontroller, like most of TI's microcontroller (MCU) platforms, is supported by a rich, layered approach to foundational software.




## Register Layer - Low-level C Header Files

Working our way up from the bottom, the MSP family provides a custom C language header file (and linker command file) for each device. These header files provide symbols that define all the various registers, pointers and bitfields found on 'your' device. Not only do they minimize the number of times you'll need to pour through the user guide and datasheet (to figure out obsequious hex values), but they make your code more readable. We also hope that providing a common set of symbols will make it easier to share and reuse code. Finally, since these files primarily contain 'definitions', they don't add any 'bulk' to your code. *(We'll discuss these files further at the end of this chapter.)*

## MSPWare (DriverLib)

MSPWare is a collection of libraries, examples, and tools. We'll examine many of these items in the next chapter. What we want to call out here is the MSPWare Driver Library – also known as "DriverLib".

MSPWare DriverLib provides a low-level abstraction layer that makes it easier to write and maintain your code. MSPWare even builds upon the 'header' file *Register Layer*, allowing you to drill-down into the provided DriverLib source code, should you ever want to discover how an API is implemented. Furthermore, it means you can easily mix-and-match DriverLib with 'header' file code.

Our main goal is to help you improve the readability and maintenance of your MSP code; that said, we also strive to keep the library as small and efficient as possible.

If you've ever had to return to low-level code a year later – or port it to another device in the same MCU family – we hope you'll appreciate the convenience and ease-of-use of DriverLib.

***Note:*** CMSIS support will be discussed towards the end of this chapter.

## Energia

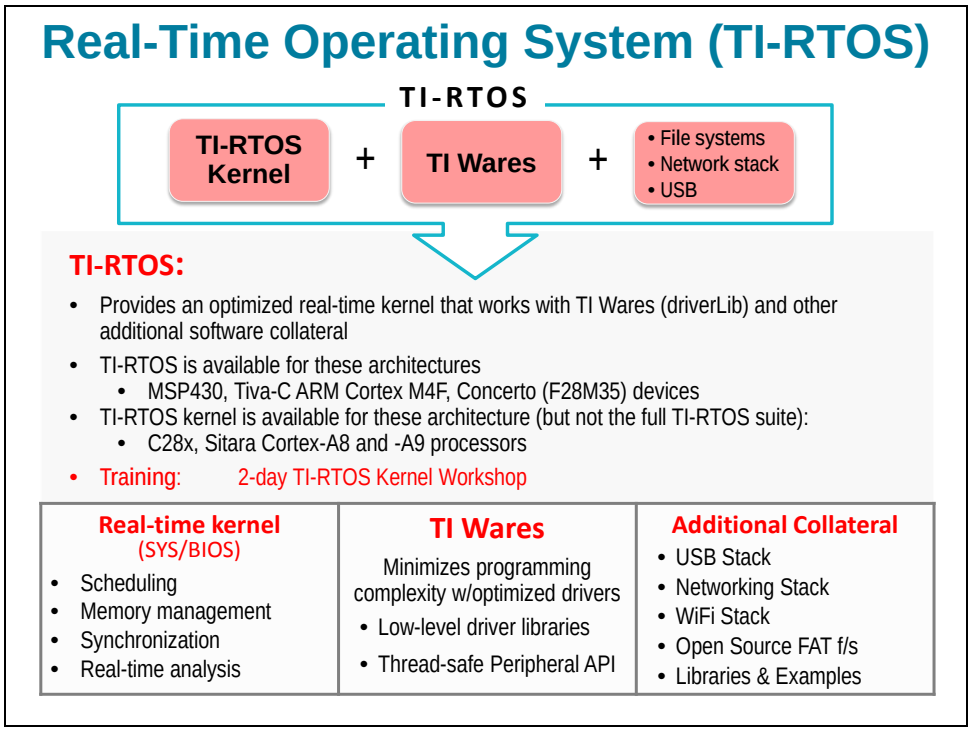
**Energia** is a community-based port of the ever-popular **Arduino**. This software makes it easy for users to grab code already available in the Arduino community and put it to good use on TI's MSP Launchpads. In other words, it puts the word "rapid" in rapid-prototyping.

In fact, Energia isn't just for prototyping anymore. We see many customers using this in small to midsize production systems. In any case, whether you use it for prototyping or otherwise, you'll find it an easy, fun way to get your ideas into hardware. (With good reason, Arduino helped coin the phrase, "Sketching with hardware".)

Like the other layers in our software model, Energia works well alongside the other layers. From your Energia sketch program, you can call DriverLib or Register-layer syntax as needed.

## TI-RTOS

TI's real-time operating system (TI-RTOS) is a highly capable package of system-building software. It's not just enough to package a bunch of software libraries together into a single executable; the TI-RTOS team validates all the components against each other – creating examples that utilize all the various libraries.




The soul of TI-RTOS is the TI-RTOS Kernel (formerly named SYS/BIOS). The kernel provides a broad set of embedded system services, most notably: Threads, Scheduling, Semaphores, Instrumentation, Memory Management, inter-thread communication and so on. It's been built with modularity in mind, so it's easy to take the parts that make sense for your application and exclude the parts that don't.

TI-RTOS includes the kernel plus a number of customized drivers built upon the TI-wares (i.e. MSPWare DriverLib). They've also thrown in a variety of other O/S level packages, such as: USB Stack, WiFi networking, FatFs. (The list will continue to grow, so keep your eye on the TI-RTOS webpage.)

# Development Tools

## Integrated Development Environments (IDE)

TI Code Composer Studio is a highly capable integrated development tool (IDE). Built on the popular Eclipse IDE platform, TI has both simplified and extended the Eclipse framework to create a powerful, easy-to-use development platform. In fact, the MSP was the first MCU inside TI to get the Eclipse treatment … but it's come a long way since then.

**Development Tools for MSP430**

| | IAR Systems | Code Composer Studio Desktop | Code Composer Studio Cloud | Energia |
|---|---|---|---|---|
| Evaluation License | ❑ 32KB code-size or 30-day limit<br>❑ Upgradeable | ❑ Full function<br>❑ JTAG limited after 90-days | No limitations | N/A |
| Compiler | IAR C/C++ | TI C/C++ or GCC | TI C/C++ GCC | GCC |
| Debugger and IDE | ❑ C-SPY<br>❑ Embedded Workbench | ❑ TI or GDB<br>❑ CCStudio (Eclipse-based) | TI Cloud Debugging (Beta) | Energia IDE (Arduino port) |
| Full Upgrade | $2700 | $445 | N/A | Free |
| JTAG Debugger | J-Link $299 | TI LaunchPad (free)<br>MSP-FET $115 | TI LaunchPad (free)<br>MSP-FET $115 | No JTAG<br>❑ serial.printf()<br>❑ LED or scope |

* Find CCS Cloud tools at: `http://dev.ti.com`

As highly as we value CCS, we know it may not be for every user. To that end, we work diligently with our 3rd parties and the open-source community to provide MSP compatibility in their ecosystems.

IAR Systems, for example, commands a huge fan base among MCU developers. Whenever the MSP team creates new tooling, they don't just think about how it can be integrated into CCS, but they also consider how it can be used by our IAR customers as well. With their highly regarded compiler, many of our customers think that the extra cost of IAR is easily worth it.

At the other end of the spectrum, we know that some of our customers cannot even afford the low-cost price-point of CCS. For hobbyists and folks needing to rapid-prototype systems, the Energia open-source port of Arduino is a great option.

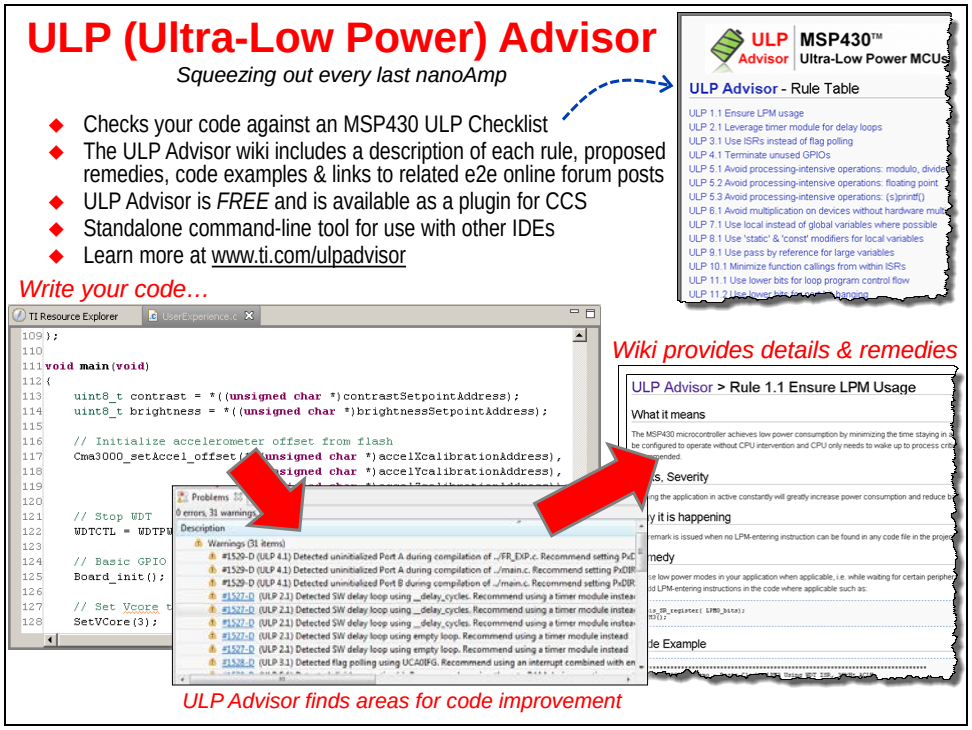
If you want to stay in the open-source domain, but step down from the abstraction provided by Energia, you can write C code using the open-source version of the Gnu Compiler (GCC).

It doesn't matter which tool suite you choose, in any case, you'll still have all the other MSP ecosystem components at your disposal. For example, MSPWare DriverLib works in all of these environments.

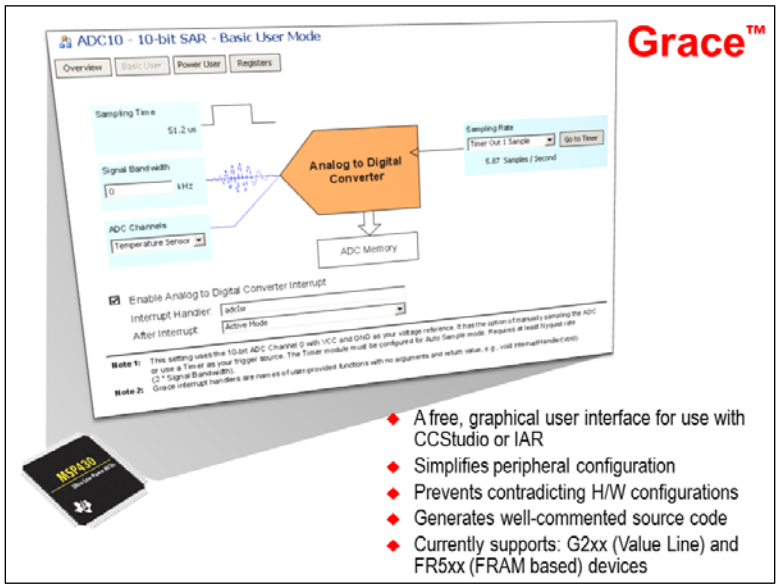
Finally, TI introduced **CCS Cloud** – an online IDE that runs in your browser. Develop your MSP program without needing to install any tools. It even supports all three software layers – Energia, DriverLib, and Register header files. Check it out today at: dev.ti.com

## Other MSP Tools

The MSP team has created a number of additional tools to support development of MSP applications. For example, since low-power designs are a major consideration for MSP users, the ***ULP Advisor*** tool provides static analysis of your code – from a power perspective – every time you compile. Novice and experienced users alike will find something they missed when trying to cut every nano-amp from their system.




***Grace***, on the other hand, provides a graphical development interface for TI's Value-Line and Wolverine series of devices. Just by selecting a few simple choices from the GUI interface, you can quickly build up your system. Grace outputs well commented DriverLib and/or Header file code. Use it to build up a custom set of drivers – or build your entire application – in Grace.

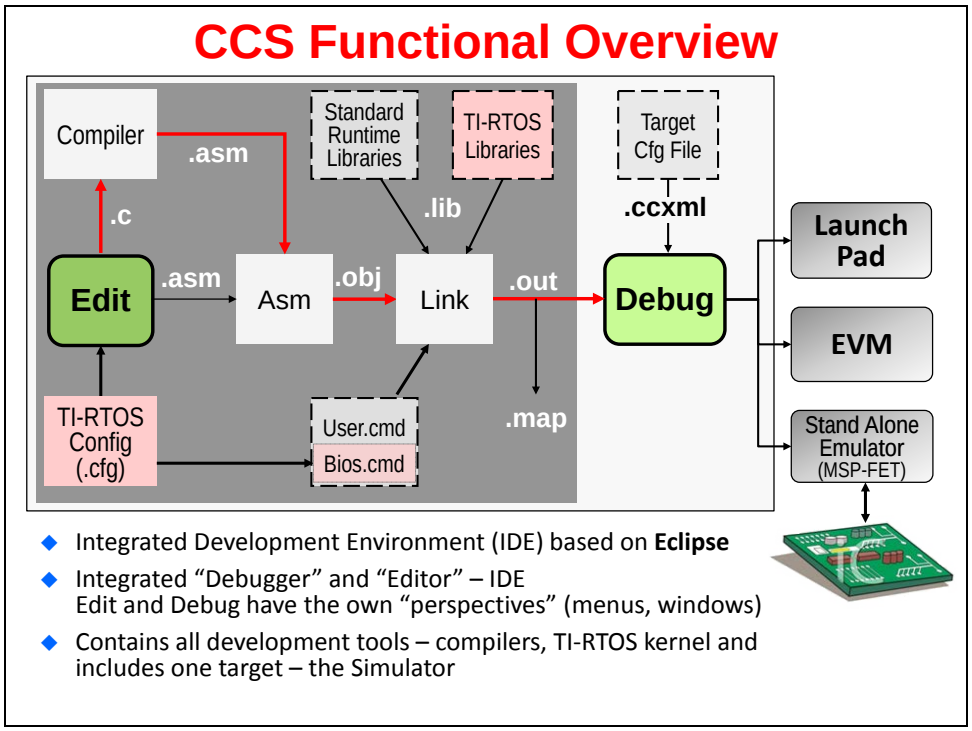


# Examining Code Composer Studio

## Functional Overview

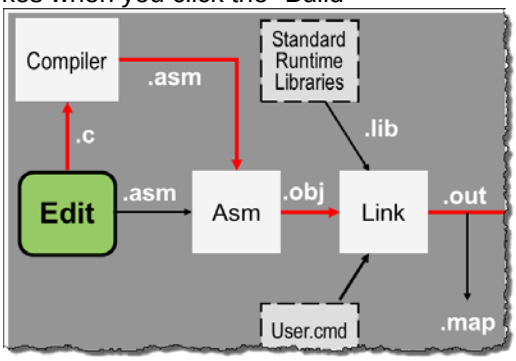
As described earlier, Code Composer Studio is TI's Eclipse based Integrated Development Environment (IDE). You might also think of IDE as meaning, "Integrated Debugger and Editor", since that's really what it provides. CCS is made up of a suite of tools that help you:

- Edit and Build your code
- Debug and Validate your code




### Editing

On the Editing side, you'll find the Compiler → Assembler → Linker tools combine to create the executable output file (.out). These are the tools that CCS invokes when you click the "Build" toolbar button.




Let's do a brief summary of the files shown here:

.c Your C (or C++) source code files

.asm Assembly files are created by the compiler. By default, they're considered temporary and deleted; though, you can tell CCS to retain them.

.obj Relocatable object files. Again thought of as temporary and deleted when build is complete.

.lib Any object library you want to reference in your code. By default, TI's compiler ships with a run-time support library (RTS) that provides standard C functions. See the compiler user's guide for more information. (slau132.pdf)

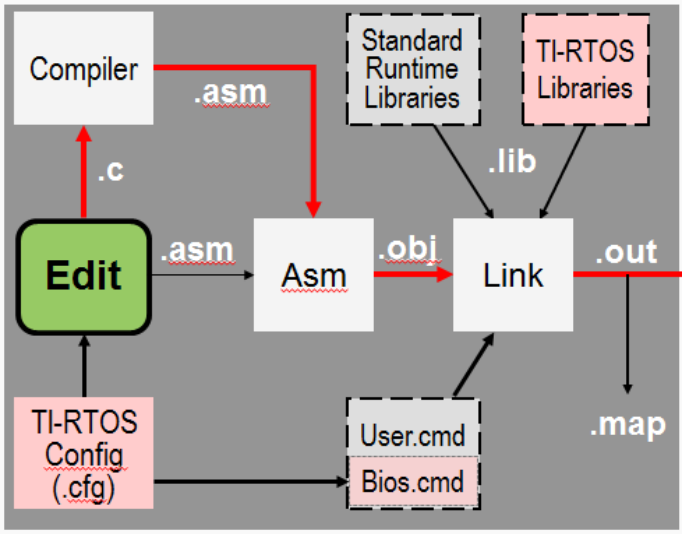
.cmd Linker command files tells the linker how to allocate memory and stitch your code and libraries together. TI provides a default linker command file specific to each MSP device; it is automatically added to your project when you create a new project. You can edit it, if needed, though most users get by without ever touching it.

.out The executable output file. This is the file that is loaded into Flash or FRAM on your MSP MCU whenever you click the "Debug" button on your CCS toolbar.

.map The map file is a report created by the linker describing where all your code and data sections were linked to in memory.

Please refer to the *MSP430 Compiler User's Guide* (slau132.pdf) and *MSP430 Assembly Language User's Guide* (slau131.pdf) for more information on the TI code generation tools.

The remaining "BUILD" tools shown in our diagram are related to the TI-RTOS kernel.




In essence, the TI-RTOS kernel is composed of many object code libraries. By creating a new project based on the TI-RTOS template, CCS will automatically:

- Link in the required libraries
- Add the TI-RTOS configuration file (.cfg)

The configuration file provides a GUI interface for specifying which parts of the kernel you want to use; helping you to create any static O/S objects that you want in your system; as well as creating a second linker command file that tells the linker where to find all the kernel's libraries.

While we briefly discuss TI-RTOS scheduling and threads during the Interrupts chapter of this workshop, we recommend you take a look at the TI-RTOS Kernel Workshop[1] if you want more information.

[1] http://processors.wiki.ti.com/index.php/Introduction_to_the_TI-RTOS_Kernel_Workshop

## Debugging

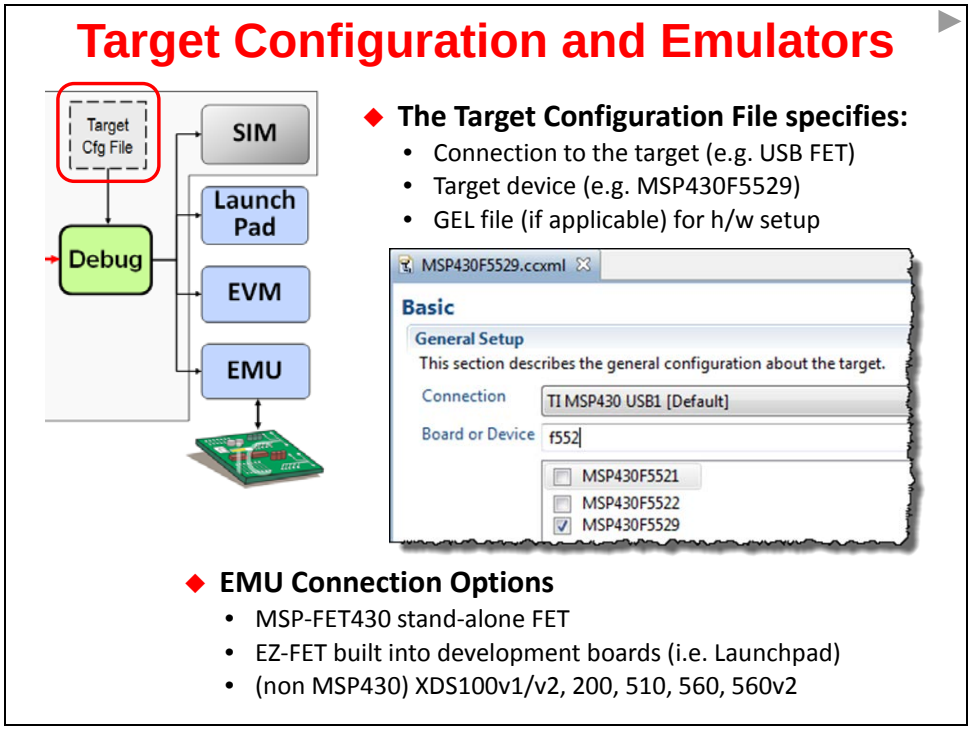
Once again, the “debug” side of the Code Composer Studio lets you download your executable output (.out) file onto your target processor (i.e. MSP device on your Launchpad) and then run your code using various debugging tools: breakpoint, single-step, view memory and registers, etc.

You will get a lot more detail and experience with debugging projects when running the upcoming lab exercises on your Launchpad.

# Target Config & Emulation

CCS needs to understand how to connect to your target. That is, which target processor do you want to download-to and run your code on?

Going back to older revisions of CCS (versions prior to CCSv4), TI provided a stand-alone tool where you would specify how the target board was connected to CCS. Nowadays, this feature has been integrated into CCS. The Target Configuration File (.ccxml) contains all the information CCS needs to connect and talk to your target (be it a board or a software simulator).




For the MSP, the CCXML file is automatically created when you create a new project. This file is based on your telling CCS which CPU variant you’ve chosen (i.e. MSP430F5529); as well as which “Connection” you are planning to use for connecting your PC to the target board.

The most common connection that MSP users choose is: **`TI MSP430 USB1 [Default]`**
In fact, this is the connection we’ll be using in the upcoming lab exercises.

---

***Note:*** If you ever get an error that indicates CCS doesn’t know how to connect to the target, you probably didn’t specify the “connection” when creating your project. You can easily fix this by editing the project’s properties.

---

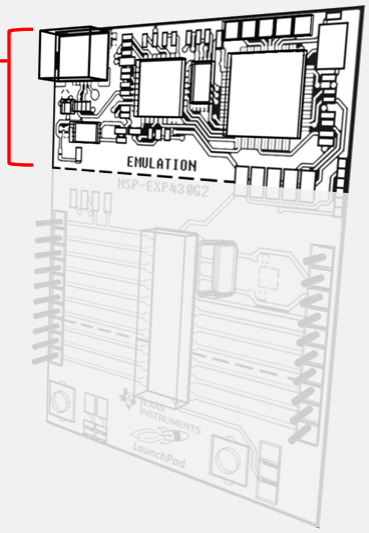
## Emulation Hardware

### MSP JTAG Emulators

MSP430 LaunchPad

**Integrated Flash Emulation Tool**

- Eliminates need for external tool
- Integrated USB-powered emulator - Mini USB cable
- Program & debug any MSP Value Line MCU through the Spy Bi-Wire (2-wire JTAG) protocol
- Use LaunchPad as a programmer ANY Spy Bi-Wire enabled MSP (not officially supported by TI)




### Flash Emulation Tool (MSP-FET)

*One tool to rule them all – Direct replacement to MSP-FET430UIF*

**Features:**

- USB debugging interface to connect any MSP MCU to a PC for real-time, in-system programming and debugging
- Enables EnergyTrace™ technology for energy measurement and debugging on all MSP devices
- Up to 4x faster than its predecessor (MSP-FET430UIF)
- Includes Backchannel UART for bi-directional communication between the MSP and a PC

Order now @ www.ti.com/tool/msp_fet

**Technical Specifications:**

- Software configurable supply voltage between 1.8 V and 3.6 V at 100 mA
- Supports JTAG Security Fuse blow to protect code
- Supports all MSP430 boards with JTAG header
- Supports both JTAG and Spy-Bi-Wire (2-wire JTAG) debug protocols

Texas Instruments

# MSP432: Cortex-M Debuggers


## XDS100v2, XDS200, XDS560 debuggers

- XDS100v2 JTAG Debug Probe (ARM version)
- XDS200 USB Debug Probe
- XDS560v2 System Trace USB Debug Probe
- XDS560v2 System Trace USB & Ethernet Debug Probe


## IAR Systems I-jet

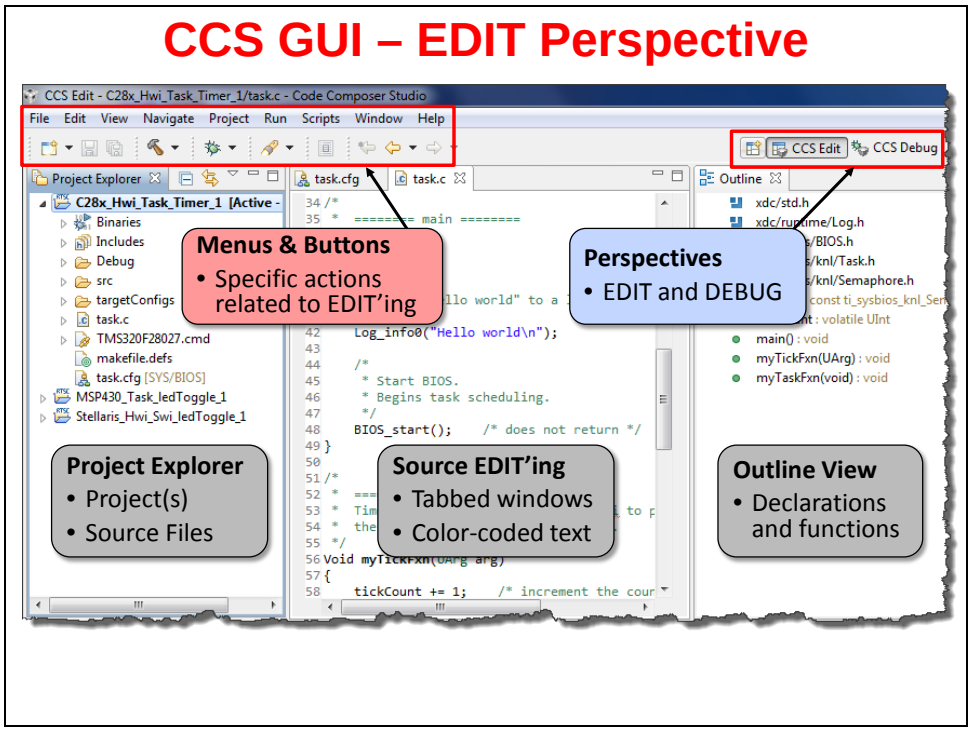
- https://www.iar.com/iar-embedded-workbench/arm/

## Segger J-link

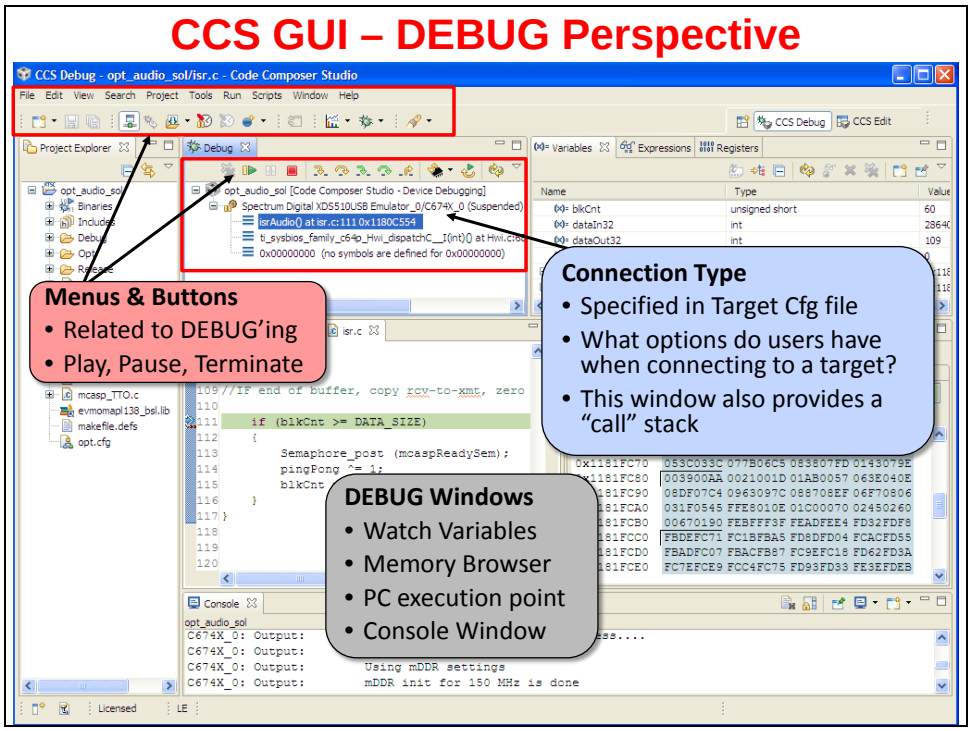
- https://www.segger.com/jlink-debug-probes.html

# Perspectives

In Eclipse, *Perspectives* describe an arrangement for toolbars and windows. ***CCS Edit*** and ***CCS Debug*** are the two perspectives that are used most often. Notice how the perspectives differ for each of the modes shown below.



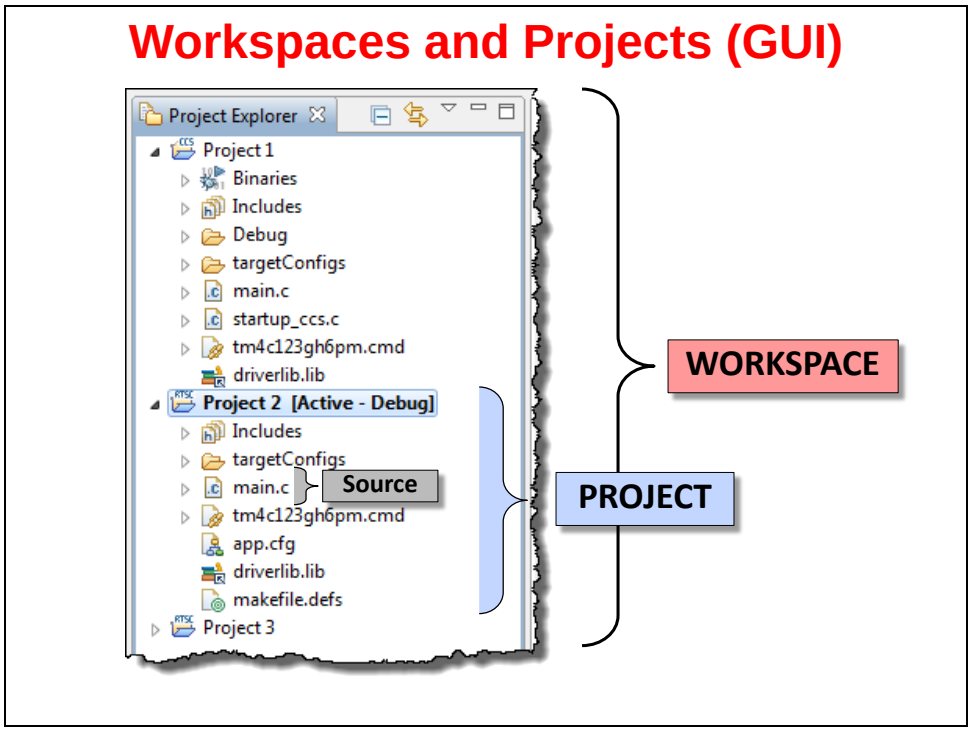

Eclipse even varies the toolbars and menus between perspectives.



# Workspaces & Projects

Eclipse based IDE's provide a hierarchy for storing program information. Experienced programmers are familiar with the concept of keeping all their programs source files in a ***Project***.

Eclipse goes one step further and also defines a ***Workspace***. In fact, whenever you open CCS (or any Eclipse IDE) you are asked to select a *workspace*. In essence, a *Workspace* is just the folder in which your projects reside. In the CCS/Eclipse, you can actually think of the *Project Explorer* window as a visual representation of your *Workspace*.




Every active project in your *workspace* will be displayed in the *Project Explorer* window, whether the project happens to be open or closed.

Some users like to only put only one project per *workspace*; others put every *project* into a single *workspace* – it doesn't matter to Eclipse.

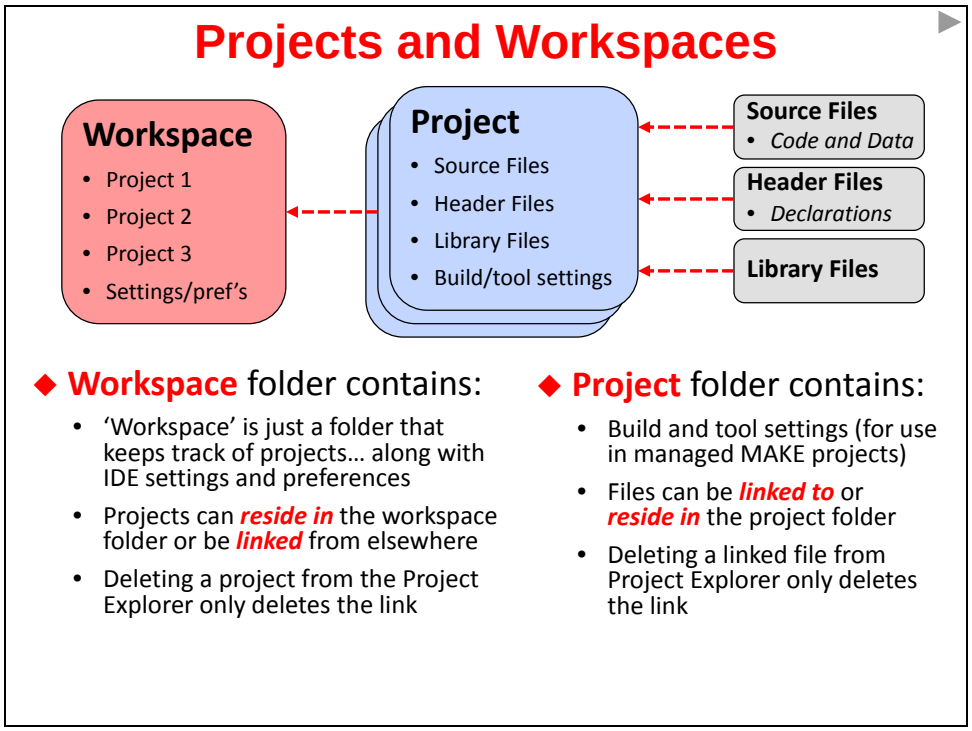
In our workshop, we have chosen to create one *workspace* which will hold all of our lab files. This makes it easy to switch back and forth between exercises, if you should want to do so.

As a final note, this hierarchy reflects how many settings are handled inside of Eclipse. Most settings are modified at the Project level – for example, you can pick the compiler per project.

Some settings, though, can be defined for the whole Workspace; for example, you can create path variables to point to library repositories. These almost always can be overridden in a given project, but this means you're not forced to define certain items over-and-over again.

Finally, there are some definitions that are globally setup in the Eclipse/IDE preferences. Unlike pre-Eclipse versions of CCS, they are not stored in the Windows registry. This makes the Linux version of the tools possible; but it also means it's easier to keep multiple versions of CCS on your computer (if you should need to do so).

Let's look at projects & workspaces from another perspective. The following diagram should confirm what we just discussed. ***Workspaces*** contain ***Projects*** which contain ***Source*** files.




Notice how the lines between the various objects are labeled "Link". This represents one way in which they can be connected. Reading the bullets on the above slide tells us that Source files can actually reside "inside" the project folder or be "linked" to the project.

As we'll see in a minute, when you add a file to a project, you have the option of "copying" the file into the project or "linking" it to the project. In other words, you have the option to decide how and where to store your files.

Within Projects, it's most common to see source files reside in the project folder; whereas, libraries are most often linked to the project. This is not a rule, but rather a style adopted by most users.

With regards to Projects and Workspaces: a project folder always resides inside of the workspace. At the very least, this is where Eclipse stores the metadata for each project (in a few different project-related XML files). The remaining project files can reside in a folder outside of the Workspace. Once again, Eclipse provides users with a lot of flexibility in how their files are stored.

## Some Final Notes about CCS/Eclipse

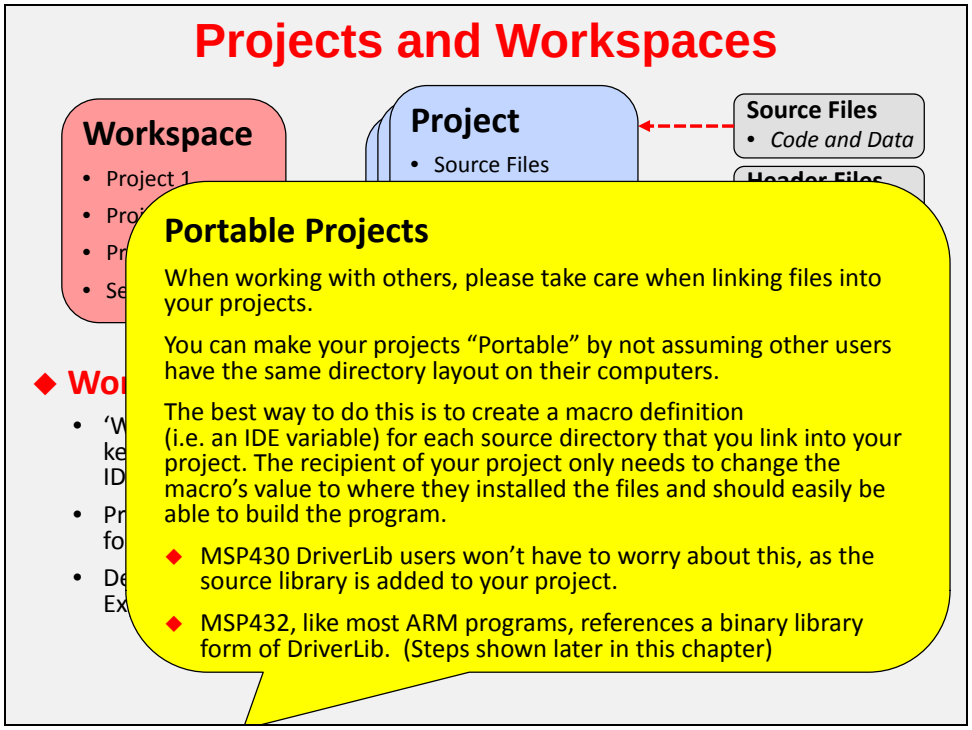
- If you create a new source file in CCS/Eclipse, it will automatically be stored in the project folder.
- If you copy a source file (e.g. C file) into the project folder using the O/S filesystem, it will automatically show up in the project. That is, if you copy a C file into the project folder using Windows explorer, it will be "in the project". Note, though, that CCS does provide a way to "exclude a file from build" – but this is not the default.
- You can export and import projects directly to/from archive (zip) files. Very nice!

## Portable Projects

While this will not be an issue when working with the MSP – at least in this workshop – you should be aware that build issues can arise when sources (files and/or libraries) are ***linked*** into a project. It isn't normally an issue on the system where the project is created, but rather, build problems can show up when sharing the project with other team members.

If your teammates do not have *exactly* the same file directory hierarchy as the person who created the project, the tools may not be able to find all of the sources – and thus, the build will fail.

This is not a TI specific problem; hence, the Eclipse IDE provides a solution.




As described here, the solution involves creating a "pointer" to each directory which contains linked source or library files. Officially, these "pointer" is called a "macro"; although it might be better described by the term "IDE variable".

Whatever you call this feature, a teammate who wants to build the project just needs to verify that the "pointer" macro contains the same directory path as the original user. If not, by updating any macro that differs in their system, the new user can easily build the project.

This is one of those problems that you might not realize is important… until you run into it.

### Note

Starting in the next chapter, we'll make heavy use of the DriverLib library to quickly build MSP applications. The MSP applications support team recommends:

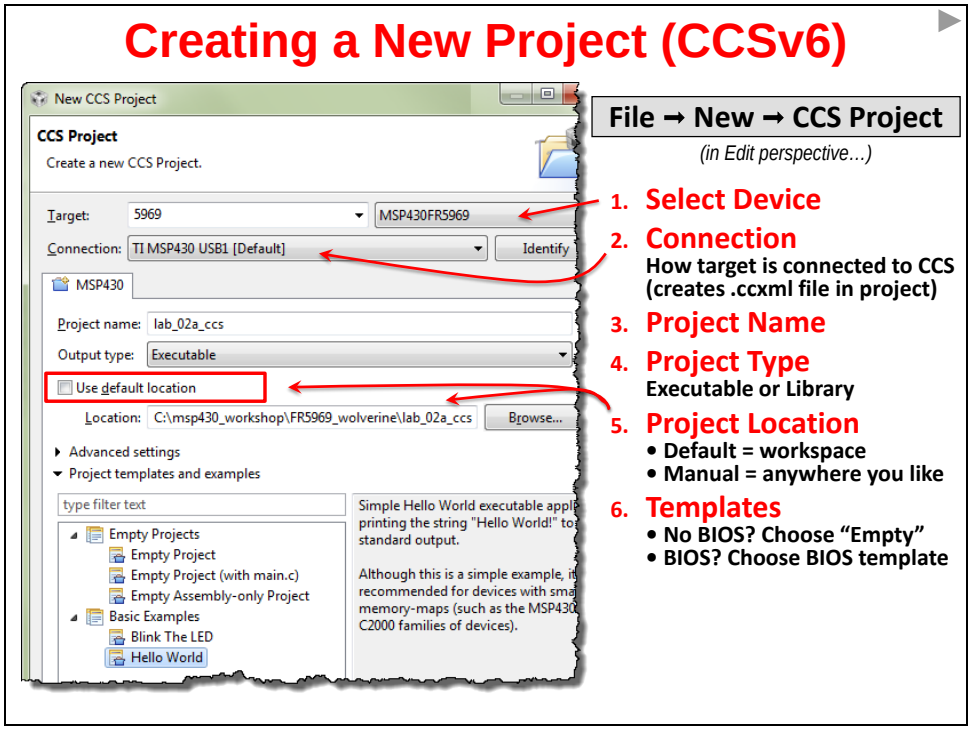
- MSP430: Import the entire MSP430 Driver Library into your project. This not only eliminates the problem of linked libraries, but it also means that the library will be built with the same compiler options as the rest of your project. (Caveat is that the first project build takes longer to complete.
- MSP432: Follow the traditional ARM style of linking the binary object DriverLib file.

# Creating a Project

There are many ways to create a new project, the easiest is to select:

```
File → New → CCS Project
```

TI defined their own C project type called “CCS Project”. This enhancement condenses the standard Eclipse “new project” wizard from 6 dialogs down to 1. *(Awesome!)*




When creating a new project you need to define:

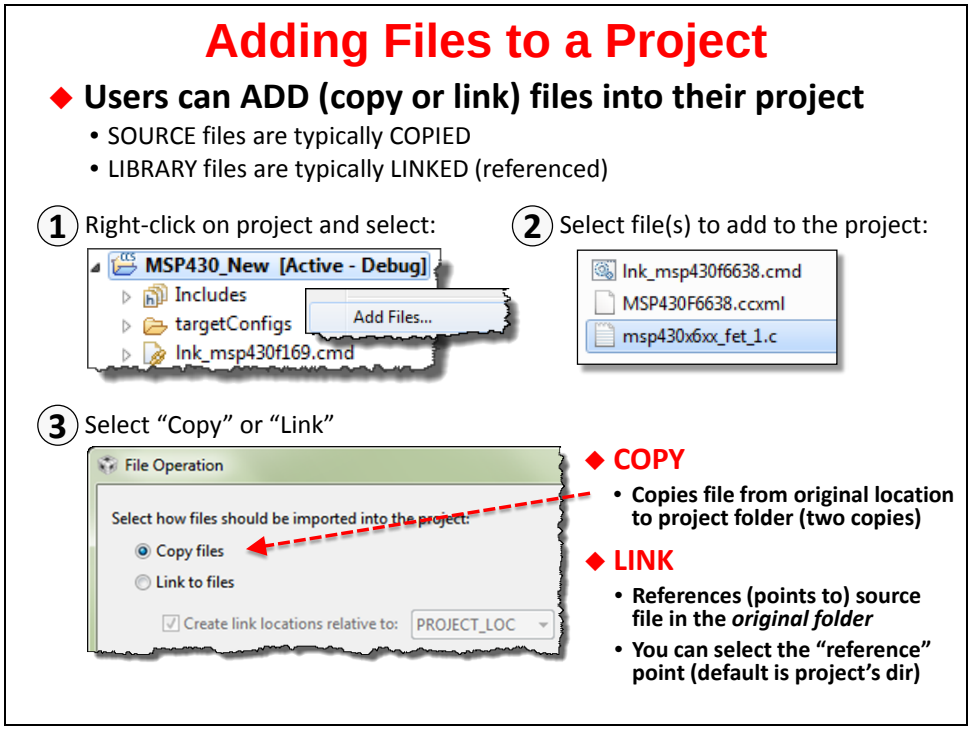
- *Project Name*
- Are you making an *Executable* program or a *Library*
- Where do you want your project to reside – by default, CCS puts it in the Workspace
- *Processor Family* (i.e. MSP)
- *Specific device* you’re using
- *Target Connection* (i.e. MSP430 USB 1)
- *Template* – CCS provides a number of project templates. The most common template is probably “Empty”. But some of the others may come in handy. For example, if you are creating a TI-RTOS based project, you will want to choose one of their project templates.

## Adding Files to a project

As we described earlier, when adding files to a project, you have the choice of copying them into the project folder or linking them to the project folder.

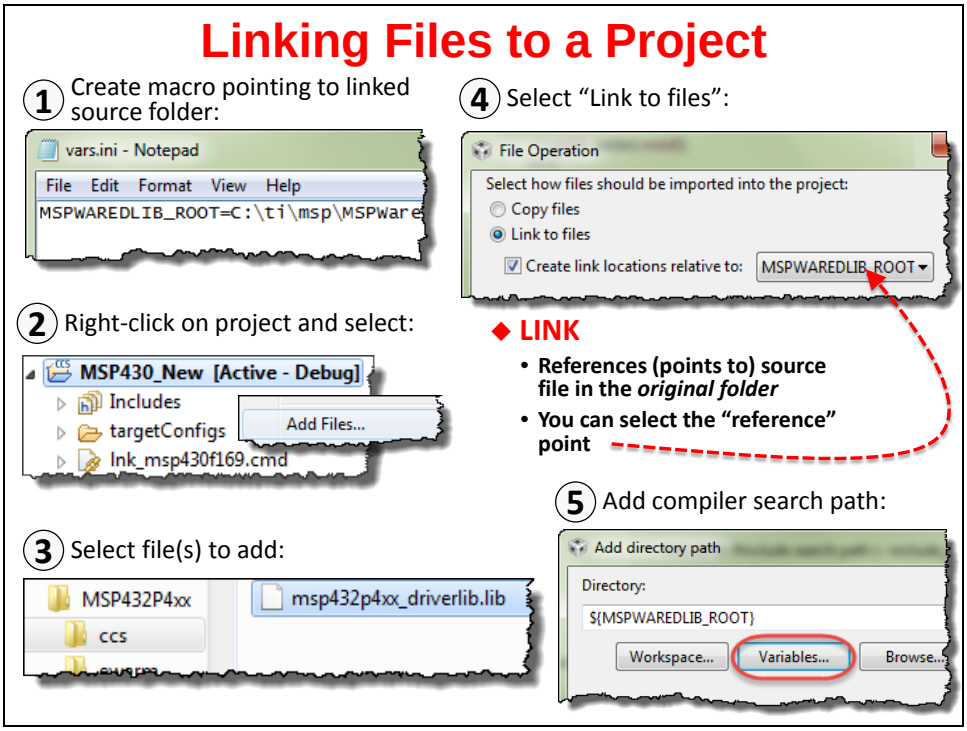
### Copying Files

Copying the files keeps them together inside the project folder. This is recommended for most individual source files – or for the MSP430 version of MSPWare DriverLib.



## Linking Files

On the other hand, if you're sharing libraries or files between projects (or with other users), it might make more sense to link them. This is recommended by the MSP apps team for MSP432 DriverLib-based projects.




Here's a quick explanation of these steps:

1. Create a macro (i.e. IDE variable) which points to the directory where your library resides. The easiest way to do this is to create a file name "macros.ini" that includes the macro (or macros) specifying your library folders. Then import the file into CCS using the command:

   `File → Import → Code Composer Studio → Build Variables`

2. Right-click on your project and select "Add Files…" to project.
3. Using the file browser, select the file or library you want to add to your program.
4. Select "Link to files" and choose the appropriate macro IDE variable CCS should use to reference the file.
5. Add the location for your libraries 'include' files to the compilers search path.

   `Open project properties → Build → ARM Compiler → Include Options`

6. Make sure to include macros.ini with your project.

## Receiving and Building a Project with Linked Files

Whenever you receive a project with linked files, you should verify that the macro (IDE variable) definitions are consistent with your file system.

- If the project contains a “macros.ini” (or “vars.ini”) file, the easiest solution is to verify it is correct. If you need to make any changes, YOU MUST reimport the file as described above.
- Try building – if a file reference is incorrect, it will lead to build errors.
- Find and verify the references manually. Verify the macros found in both locations shown:

  `Open project properties → Resource → Linked Resources → Path Variables`
  `→ Build → Variables (tab)`

  If you find an incorrect folder reference, use/create a “macros.ini” file and import the new correct values. Make sure you check the “Update existing” option, when requested.

### Portable Projects Summary

*This is not an issue for this workshop because the MSP430 team recommends that you add a copy of DriverLib to each project. That said, you will likely run into this issue in the future, so we wanted to bring it to your attention.*

The phrase ***Portable Projects*** signifies that projects can be built in a portable fashion. That is, with a little consideration, it is easy to build projects that can be moved from one user to another – or from one computer environment to another.

When a source file or library is contained inside of a project folder, it is easy for the tools to find and use it. Eclipse automatically knows how to find files inside the project folder.

The biggest headache in moving projects relates to "linked" source files and libraries. When a file is located outside of the project folder, the build will fail unless the person receiving the project user places all the referenced (i.e. linked) files into exactly the same locations inside their filesystem. This is a very common problem!!!

The best solution is to use Eclipse *Path Variables* to point to each directory where you have linked resources. Since this is not a problem encountered in this workshop, we suggest you refer to these locations for more info:

http://processors.wiki.ti.com/index.php/Portable_Projects

You may also want to reference the *Tiva-C Workshop* or the *TI-RTOS Kernel Workshop* for additional examples in dealing with *Portable Projects*.

# Licensing/Pricing

Many users will find that they can use Code Composer Studio free of charge.

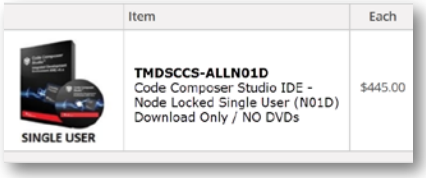
For example, there is no charge when using CCS with most of the available TI development boards – with the MSP, they allow you to use it for free (with any tool), as long as your program is less than 16KB.

Furthermore, TI does not charge for CCS licenses when you are connecting to your target using the low-cost XDS100 JTAG connection.

## CCStudio Licensing and Pricing

**Licensing**

- Wide variety of options (node locked, floating, time based)
- All versions (full, DSK, free tools) use same image
- Annual subscription - $99 *($159 for floating)*
- Updates available online




| Item | Description | Price | Annual |
|---|---|---|---|
| Platinum Eval Tools | Full tools with 90 day limit (all EMU) | FREE | |
| Platinum Bundle | XDS100; Simulators; many TI dev'l boards (such as Tiva-C Launchpad); MSP430 when using GNU Compiler | FREE | |
| 16K Code-Size Limited | MSP430 when using TI C Compiler | FREE | |
| Platinum Node Lock | Full tools tied to a machine | $445* | $ 99 |
| Platinum Floating | Full tools shared across machines | $795 | $159 |

* Download version; $495 when disc is shipped to you

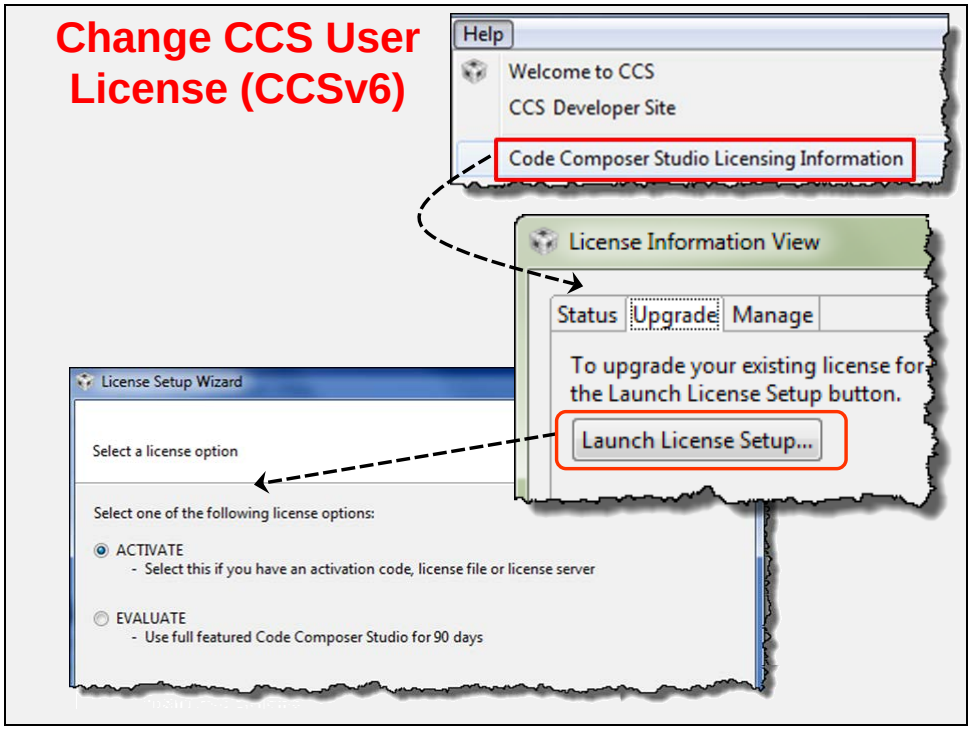
For those cases where you need to use a more sophisticated (i.e. faster) JTAG development connection, TI provides a 90-day free evaluation license. After that, you need to purchase the tool. Thankfully, when you encounter one of these cases, CCS for only costs $445.

## Changing a CCS User Licence

In this workshop, we can use the free license options. For CCSv5 you would choose the "***16K Code Size Limited (MSP430)"*** option; you don't have to do anything for CCSv6, it defaults to the free option.

It is a little bit tricky to change the licensing method. That is, it's hard to find the following dialog.

As shown, choose *Code Composer Studio Licensing Information* from the *Help* menu. When that dialog appears, choose the *Upgrade* tab, then click the *Launch License Setup…* button.



# Writing MSP C Code

As part of the prerequisites for the workshop, we stated that you should be familiar with the C language; therefore, in this section we do not plan to cover general C language syntax. Rather, this section is dedicated to implementation details of the MSP C Compiler.

## Build Config & Options

TI C compilers offer nearly a hundred different build options. We plan to focus on just a few options so that you're aware of the most common ones.

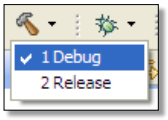
You should find the table below broken into two sets of options:

### Compiler Build Options

- Almost 100 compiler options let you tune your code's performance, size, etc.
- The following table lists the most commonly used options:

| | Options | Description |
|---|---|---|
| Debug | -ss | Interlist C statements into assembly listing |
| Optimize (Release) | -o3 | Invoke optimizer (-o0, -o1, **-o2**/-o, -o3, -o4) |
| | -mf | Speed/code size tradeoff (-mf0 thru -mf5) |
| | -k | Keep asm files, but don't interlist |

- To make things easier, CCS creates two BUILD CONFIGURATIONS:
  - *Debug* (no optimization) which is great for LOGICAL debug
  - *Release* which is good for PERFORMANCE/Size
  - Users can create their own custom build configurations



## *Debug Options*

Until recently, you were required to use the **–g** option when you wanted source-level debugging turned on. The drawback to this option was that it affected the code performance and size. This has changed… since source-level debugging does not affect the optimizer's efficiency, it is always enabled.

On the other hand, if you want to see your C code interlisted with its associated assembly code, then you should use the **–ss** option. Be aware, though, that this does still affect the optimizer – which means that you should turn off this option when you want to minimize the code size and maximize performance such as when building your production code.

## Optimize Options (aka "Release" Options)

We highlight 3 optimization options:

- **-o** turns on the optimizer. In fact, you can enable the optimizer with different levels of aggressiveness; from –o0 up thru –o4. When you get to –o3, the compiler is optimizing code across the entire C file. Recently, TI has added the –o4 level of optimization; this provides link-time optimizations, on top of all those performed in level –o3.
- **-mf** lets the compiler know how to tradeoff code size versus speed.
- **-k** does not change the optimizer; rather, it tells the tools to *keep* the assembly file (.asm). By default the asm file is deleted, since it's only an intermediate file. But, it can be handy if you're trying to debug your code and/or want to evaluate how the compiler is interpreting your C code. ***Bottom Line:*** *When optimizing your code, replace the –ss option with the –k option!*

## Build Configurations

Early in development, most users always use the *Debug* compiler options.

Later in the development cycle, it is common to switch back and forth between *Debug* and *Release* (i.e. optimize) options. It is often important to optimize your code so that it can perform your tasks most efficiently … and with the smallest code footprint.

Rather than forcing you to continuously tweak options by hand, you can use *Build Configurations*. Think of these as 'groups' of options.

When you create a new project, CCS automatically creates two Build Configurations:

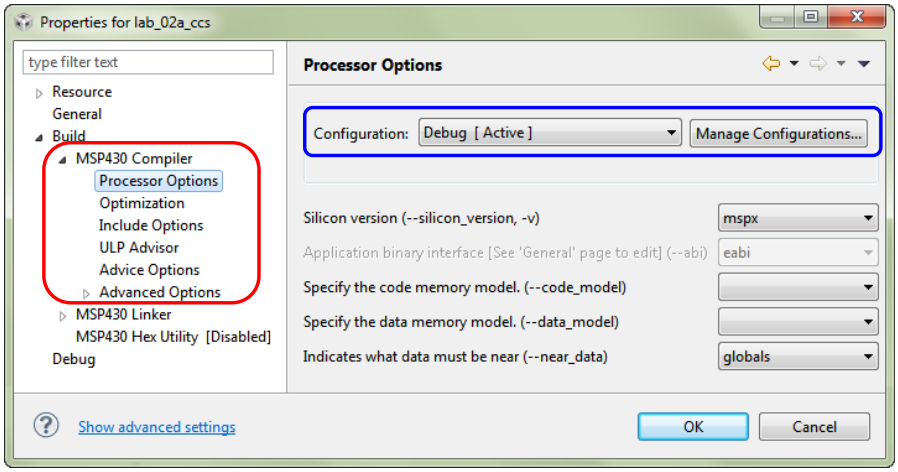
- Debug
- Release

This makes it easy for you to switch back and forth between these two sets of options.

Even further, you can modify each of these option sets … or create your own.

### Modifying Build Configurations

- Right-click on the project and select *Properties*
- Select the build configuration: *Debug or Release*
- Then click "*Processor Options*" or any other category (like *Optimization*):




**Hint:** If you modify a Project build option, it only affects the <u>active</u> build configuration.

This is a common source of errors. For example, when you add a new library search path to your project options during Debug, it only affects that configuration. This means that it's common to run into errors whenever you switch to the Release build configuration.

CCS is trying to help – and often asks if you want to update both/all configurations. But, this is a new feature and only works for some of the options. This means that when an option should apply to all configurations, you should (manually) change them both at the same time … or be prepared to tweak the Release build options the first time you use it.

# Data Types

The following data types are specified in the C Compiler Users Guide. We've circled the types that best describe this processor.

With the MSP's ability to perform byte-wide addressing, it follows that *char*'s are 8-bits.

As one might expect, though, being a 16-bit CPU, both the *short* and *int* data types are 16-bits wide.

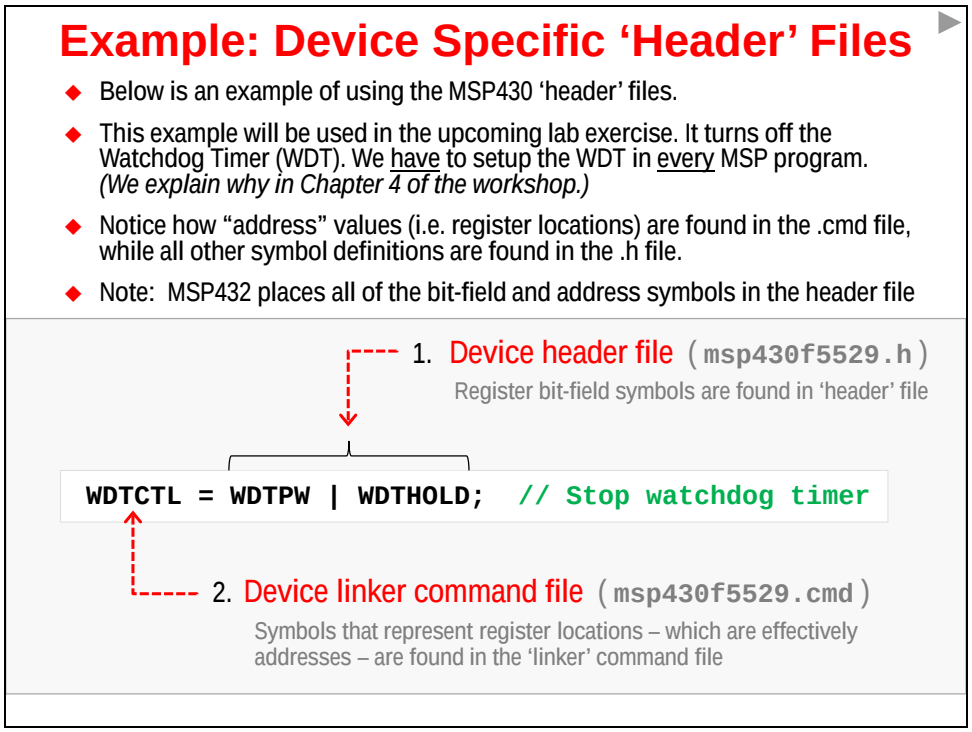
## MSP430 C Data Types (ELF format)

| Type | Bits | Representation |
|---|---|---|
| char | 8 | (aligned to 8-bit boundary) |
| short | 16 | Binary, 2's complement |
| int | 16 | Binary, 2's complement |
| long | 32 | Binary, 2's complement |
| long long | 64 | Binary, 2's complement |
| float | 32 | IEEE 32-bit |
| double | 64 | IEEE 64-bit |
| long double | 64 | IEEE 64-bit |

- Data are aligned to 16-bit address boundary (except where noted)
- 8-bit values are stored in bits 0-7 of a register
- 32- and 64-bit types require 2 and 4 registers, respectively

# Device Specific Files (.h and .cmd)

TI has created a device-specific header file (.h) and linker command file (.cmd) for each specific MSP device. With the MSP430F5529 device as an example, if you look through the files installed with the MSP compiler, you'll find: `msp430f5529.h` and `msp430f5529.cmd`




As described in the above diagram, these two files provide symbolic definitions for all registers and bitfields found in each CPU and its peripherals.

What's the simple key to figure out which file contains a given symbol?

- If the symbol relates to an address, such as the symbol for a memory-mapped register (e.g. WDTCTL), you'll find it defined in the .CMD file. This is because the linker (and its associated linker command file) specifies memory allocations and addresses.
- All the other device-specific symbols are described in the header (.h) file, as is common practice for the C language.

To make programming easier for you, CCS automatically adds these two device-specific files to your project.

- You'll find a linker command file added to your project folder; in fact, it should be listed in the Project Explorer window within your project.
- Most new CCS projects include an "empty" `main.c` file. The header file is #included at the top of this file.

## Device Specific Files (.h/.cmd)

- New CCS projects automatically contain two files based upon the "Target CPU" selection:
  1. Device header file (e.g. **`msp430f5529.h`**)
     - Symbols defined for bit fields, reg's, etc.
     - Structs/union's also defined for bit fields, if you prefer
     - You shouldn't have to use hard-coded bit locations, etc.
     - Your code should #include msp430.h, this points to the device specific .h file
  2. Device linker command file (e.g. **`msp430f5529.cmd`**)
     - Device specific addresses defined in dev specific .cmd file
     - Creating a new CCS project automatically includes a project .cmd file … which includes the device specific .cmd file
     - You shouldn't have to ever look up the address of a register
     - Default linker command file in your project points to device specific .cmd file
- You should use these symbols in your code, rather than specifying hard values/addresses
- MSPWare also uses these symbolic definitions; that is, these definitions represent the lowest-level abstraction layer for C code

In the next chapter we introduce the MSPWare Driver Library. It utilizes these device-specific header (and linker command) files, though it is automatically included by including the Driver Library's own header file `<driverlib.h>`.

# MSP Compiler Intrinsic Functions

Along with the symbols defined in the device specific header & linker files, it's common to see programmers use the compiler's intrinsic functions. Think of these as functions that are "built-in" to the TI compiler. In most cases, intrinsic functions correlate to hardware specific features found in processors.

## Intrinsics for MSP C Compiler

- Compiler intrinsic functions are essentially "built-in" C functions
- They usually provide access to underlying hardware features of a processor; often mapping closely to specific asm instructions
- We will use some of these in today's workshop:

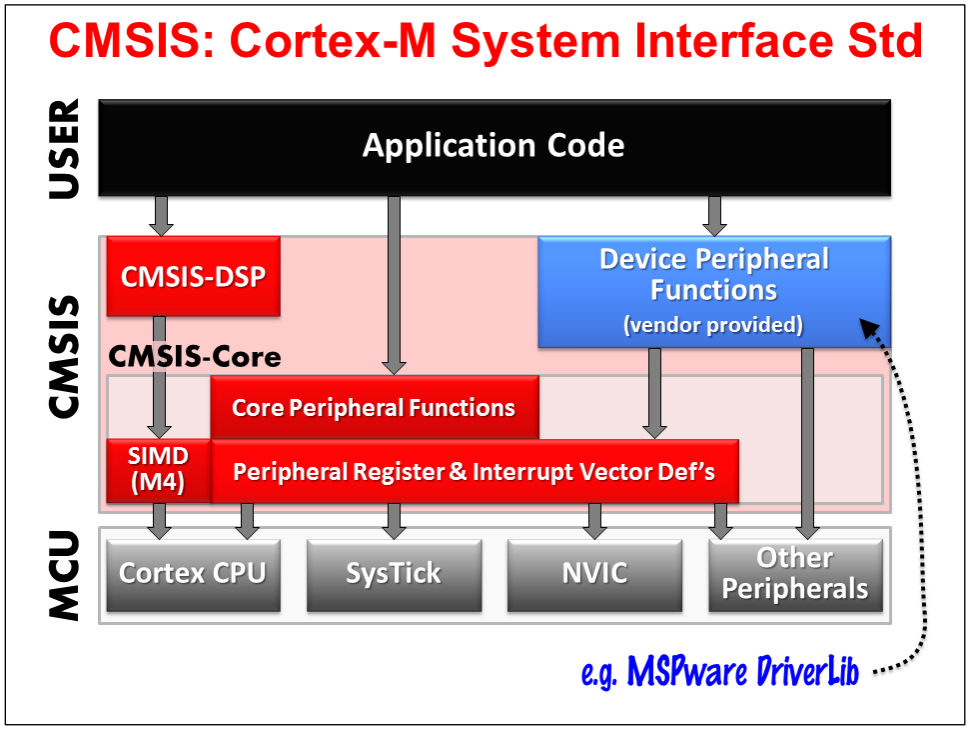
```
__bcd_add_short();
__bcd_add_long();
__bic_SR_register();
__bic_SR_register_on_exit();
__bis_SR_register();
__bis_SR_register_on_exit();
__data16_read_addr();
__data16_write_addr ();
__data20_read_char();
__data20_read_long();
__data20_read_short();
__data20_write_char();
__data20_write_long();
__data20_write_short();
__delay_cycles();
__disable_interrupt();
__enable_interrupt();
__even_in_range();
__get_interrupt_state();
__get_R4_register();
__get_R5_register();
__get_SP_register();
__get_SR_register();
__get_SR_register_on_exit();
__low_power_mode_0();
__low_power_mode_1();
__low_power_mode_2();
__low_power_mode_3();
__low_power_mode_4();
__low_power_mode_off_on_exit();
__never_executed();
__no_operation();
__op_code();
__set_interrupt_state();
__set_R4_register();
__set_R5_register();
__set_SP_register();
__swap_bytes();
```

We've circled some of the intrinsic functions we'll use in this class; from setting and/or clearing bits in the Status Register (SR) to putting the processor into low-power modes.

# ARM CMSIS-Core Files (MSP432 only)

CMSIS (Cortex-M System Interface Standard) is an effort made by ARM to provide a standardized way to program specific component of an ARM Cortex-M based system.

Shown in the diagram below, the "CMSIS-Core" layer makes up a central part of the CMSIS API. These definitions and routines provide a standardized interface for elements that are in the ARM Cortex-M CPU (i.e. ARM Cortex-M4F in the case of the MSP432).




While the ARM intellectual property provides a definition for the CPU core of a processor, it does not specify its entire feature set – that must be done by the device vendor. For this reason, CMSIS describes a "vendor provided" block shown as "Device Specific Functions".

Great minds think alike – the CMSIS standard actually fits quite well with TI's long-term software strategy, part of which we discussed earlier in this chapter.

- CMSIS *Device Peripheral Functions* correlate to the MSP432's *MSPWare Driver Library*
- CMSIS-*Core* matches up with TI's *Register Layer* header files as well as a number of intrinsic functions supported by the TI (and GCC) compilers.

# Comparing Coding Styles – MSP vs CMSIS

## Register Layer

As noted, both software styles provide bitwise definitions for all device resources. MSP style primarily uses #define definitions for the various registers, fields and enumerations. CMSIS tends towars bit-wise structure definitions.

Bottom-line: TI supports both coding styles. Pick whichever suits your needs or perferences.

### ① Register Layer (Low-Level C Header Files)

**MSP definition**

```
#define WDTCTL (HWREG16(0x4000480C))

/* WDTCTL Control Bits */
#define WDTIS0       (0x0001)
#define WDTIS1       (0x0002)
#define WDTIS2       (0x0004)
#define WDTCNTCL     (0x0008)
#define WDTTMSEL     (0x0010)
#define WDTSSEL0     (0x0020)
#define WDTSSEL1     (0x0040)
#define WDTHOLD      (0x0080)
#define WDTPW        (WDTPW_VAL)

#define __WDT_BASE__ (0x40004800)
```

**CMSIS definition**

```
typedef struct {
  uint8_t RESERVED0[12];
  union {    /* WDT_CTL Register */
    __IO uint16_t reg;
    struct { /* WDT_CTL Bits */
      __IO uint16_t IS    :      3;
      __O  uint16_t CNTCL :      1;
      __IO uint16_t TMSEL :      1;
      __IO uint16_t SSEL  :      2;
      __IO uint16_t HOLD  :      1;
      __IO uint16_t PW    :      8;
    } bit;
  } CTL;
} WDT_Type;
#define WDT ((WDT_Type *) __WDT_BASE__)
```

```
// Stop watchdog timer
WDTCTL = WDTPW | WDTHOLD;            // Traditional MSP style
WDT_A->rCTL.r = 0x5A0000 | (1<<7);   // CMSIS syntax
WDT_A->rCTL.r = WDTPW | WDTHOLD;     // Hybrid syntax
```

## Driver Library / Functional Layer

Likewise, TI has chosen to support the functional layer with both MSPWare style DriverLibrary as well as the CMSIS style intrinsics. (In many cases, you also have the option to use TI or GCC intrinsics, as well.)

Once again, you are free to choose what best suits your needs.

**② Functional Layer (Driver Library)**

**MSPware DriverLib**

```
:
// Interrupt Functions
Interrupt_enableInterrupt()
Interrupt_disableInterrupt()
Interrupt_isEnabled()
Interrupt_enableMaster()
Interrupt_disableMaster()
Interrupt_pendInterrupt()
Interrupt_unpendInterrupt()
Interrupt_setPriority()
Interrupt_setPriorityGrouping()
Interrupt_setPriorityMask()
Interrupt_setVectorTableAddress()
```

**CMSIS Intrinsics**

```
:
// NVIC (Interrupt) Functions
NVIC_EnableIRQ()
NVIC_DisableIRQ()
NVIC_GetPendingIRQ()
NVIC_SetPendingIRQ()
NVIC_ClearPendingIRQ()
NVIC_SetPriority()
NVIC_GetPriority()
NVIC_EncodePriority()
NVIC_DecodePriority()
NVIC_GetPriorityGrouping()
NVIC_SetPriorityGrouping()
```

```
// Put the mcu to sleep or into deep_sleep
__wfi();                          // CMSIS syntax (sleep)
__deep_sleep();                   // CMSIS
PCM_gotoLPM0();                   // MSP DriverLib style
PCM_gotoLPM3();                   // MSP 'deep sleep'
```

## Coding Styles – Why both?

The popularity of the MSP432 draws users from two different groups:

- Traditional MSP microcontrollers users who want access to full 32-bit performance, without giving up the ultra-low power nature of MSP devices
- ARM Cortex-M uses looking to maintain (or improve) their performance while drastically reducing the energy requirements of their applications.

With users from both groups moving towards the MSP432, TI realized the need to support both styles of coding – knowing that users would want to minimize the impact on code written for their previous (MSP or ARM Cortex-M) designs.

In the end, since both are supported by Texas Instruments, you can pick your preference. If coming to the MSP432 from either avenue, choose the style that best maximizes your incumbent code. On the otherhand, if you're new to microcontroller programming, pick the style which will make you most productive.

# Lab 2 – CCStudio Projects

The objective of this lab is to learn the basic features of Code Composer Studio. In this exercise you will create a new project, build the code, and program the on-chip flash on the MSP device.

## Lab 2 – Creating CCS Projects

- **Lab 2a – Hello World**
  - Create a new project
  - Build program, launch debugger, connect to target, and load your program
  - printf() to CCS console
- **Lab 2b – Blink the LED**
  - Explore basic CCS debug functionality
    Restart, Breakpoint, Single-step, Run-to-line
- **Lab 2c – Restore Demo to Flash**
  - Import CCS project (for original demo)
  - Load program to device's flash memory
  - Verify original demo program works
- **(Optional) Lab 2d**
  - Create binary TXT file of your program
  - Use MSP430 Flasher to program original demo's binary file to device's flash


**Time: 45 minutes**

# Lab Outline

# Lab 2a – Creating a New CCS Project

In this lab, you create a new CCS project that contains one source file – `hello.c` – which prints "Hello World" to the CCS console window.

The purpose of this lab is to practice creating projects and getting to know the look and feel of CCS. If you already have experience with CCS (or the Eclipse) IDE, this lab will be a quick review. The workshop labs start out very basic, but over time, they'll get a bit more challenging and will contain less "hand holding" instructions.

**Hint:** In a *real-world* MSP program, you would **NOT want to call *printf()***. This function is slow, requires a great deal of program and data memory, and sucks power – all bad things for any embedded application. (Real-world programs tend to replace *printf()* by sending data to a terminal via the serial port.)

We're using this function since it's the common starting point when working with a new processor. Part B of this lab, along with the next chapter, finds us programming what is commonly called, the "embedded" version of "hello world". This involves blinking an LED on the target board.

## Intro to Workshop Files

**1. Find the workshop lab folder.**

Using Windows Explorer, locate the following folder. In this folder, you will find at least two folders – aptly named for the two launchpads this workshop covers – F5529_USB, FR5969_FRAM, FR4133_FRAM, FR6989_FRAM, and MSP432P401R:

```
C:\msp_workshop\F5529_usb          C:\msp_workshop\FR4133_fram
C:\msp_workshop\FR5969_fram        C:\msp_workshop\FR6989_fram
C:\msp_workshop\msp432p401r
```

Click on YOUR specific target's folder. Underneath, you'll find many subfolders

```
C:\msp_workshop\F5529_usb\lab_02a_ccs
C:\msp_workshop\F5529_usb\lab_02b_blink
      ...
C:\msp_workshop\F5529_usb\solutions
C:\msp_workshop\F5529_usb\workspace
```

From this point, we will usually refer to the path using the generic ***<target>*** so that we can refer to whichever target board you may happen to be working with.

e.g. `C:\msp_workshop\`***<target>***`\lab_02a_ccs`

So, when the instructions say "navigate to the Lab2 folder", this assumes you are in the tree related to YOUR specific target.

Finally, you will usually work within each of the `lab_` folders but if you get stuck, you may opt to import – or examine – a lab's archived (.zip) solution files. These are found in the `\solutions` directory.

**Hint:**
- This lab does not contain any "starter" files; rather, we'll create everything from scratch.
- The readme file provides the solution code that you can copy/paste, if necessary. That said, you won't need to do that in this lab exercise.

## Start Code Composer Studio and Open a Workspace

***Note:*** CCSv6 should already be installed; if not please refer to the workshop installation guide.

**2. Start Code Composer Studio (CCS).**

Double-click CCS's icon on the desktop or select it from the Windows Start menu.

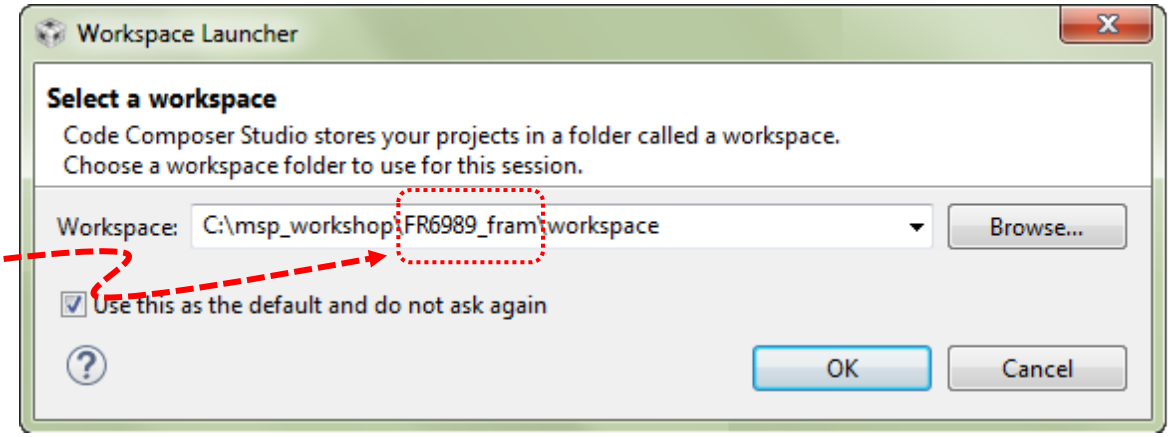
**3. Select a *Workspace* – don't use the default workspace location !!**

When CCS starts, a dialog box will prompt you for the location of a workspace folder. We suggest that you select the workspace folder provided in our workshop labs folder. *(This will help your experience to match our lab instructions.)*

**Select either one of:** (to match your target)

`C:\msp_workshop\`***<target>***`\workspace`

Make sure to select the correct location for your LaunchPad




Most importantly, the workspace provides a location to store your projects … or links to your projects. In addition to this, the workspace folder also contains many CCS preferences, such as perspectives and views. The workspace is saved automatically when CCS is closed.

**`Hint:`** If you check the *"Use this as the default…"* option, you won't be asked to choose a workspace every time you open CCS. At some point, if you need to change the workspace – or create a new one – you can do this from the menu: `File → Switch Workspace`

**4. Click OK to close the "Select a workspace" dialog.**

**5. After quickly examining the "Getting Started" window, you can close it, too.**

When CCS opens to a new workspace, the *Getting Started* window is automatically opened and you're greeted with a variety of options. We want to mention two items:

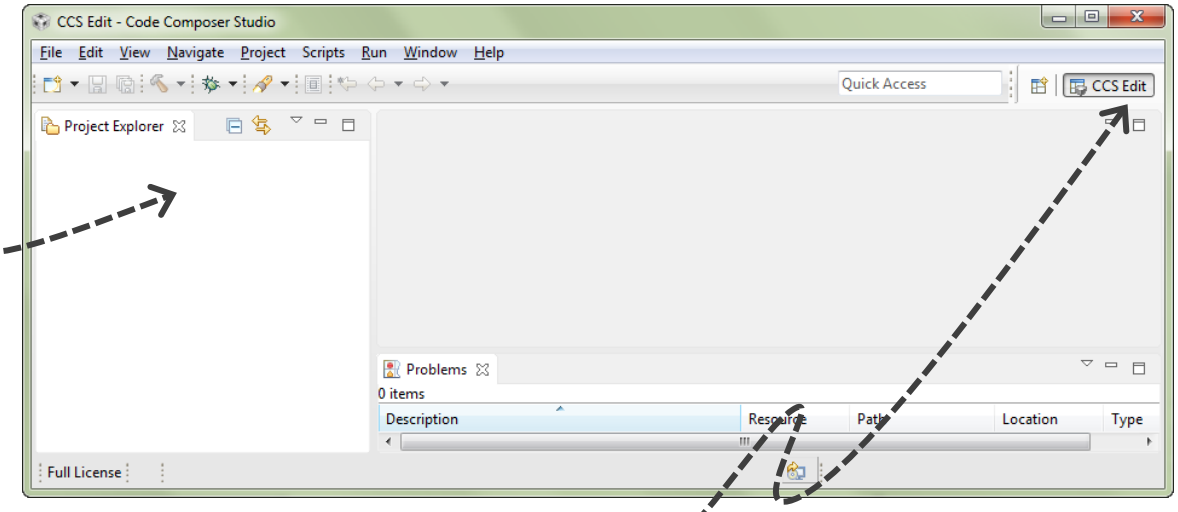
- **App Center** – you can download additional TI tools and content here. For example, this is one way to install MSPWare or TI-RTOS.
- **Simple Mode** – We suggest that you do not put CCS into Simple Mode when following our lab instructions, as we've chosen to use the full-featured interface.

Later on, you may want to come back and check out the remaining links and videos.

## "CCS Edit" Perspective

**6. At this point you should see an empty CCS workbench.**

The term *workbench* refers to the desktop development environment.




Notice Project Explorer is empty – this matches our empty Workspace folder

The workbench will open in the "CCS Edit" view.

```
Maximize CCS to fill your screen
```

Notice the tab in the upper right-hand corner…

Perspectives define the window layout views of the workbench, toolbars, and menus – as appropriate for a specific type of activity (i.e. editing or debugging). This minimizes clutter of the user interface.

- The "CCS Edit" perspective is used to when creating, editing and building C/C++ projects.
- CCS automatically switches to the "CCS Debug" perspective when a debug session is started.

You can customize the perspectives and save as many as you like.

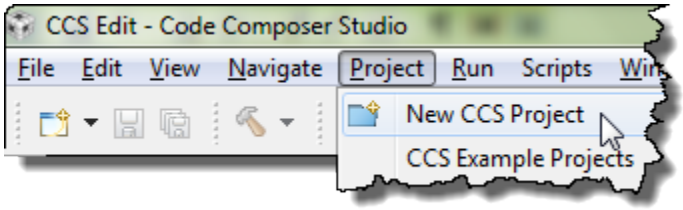
| | |
|---|---|
| **Hint:** | The `Window` → `Reset Perspective`… is handy for those times when you've changed the windows and want to get back to the original view. |

## Create a New Project

**7. Select New CCS Project from the menu.**

A *project* contains all the files you will need to develop an executable output file (.out) which can be run on the MSP hardware. To create a new project click:

`File → New → CCS Project`




**8. Make project choices as shown here:**

*Note: Your dialog may look slightly different than this one. This is how it looked for CCSv6.1 (build 104).*

a) Type "**5529**", "**5969**", "**4133**", **"6989"** or **"P401"** into *variant* to quickly select **Target CPU**

b) Use *Default* debugger connection *(this creates the .ccsxml file for you)*

c) Name: `lab_02a_ccs`

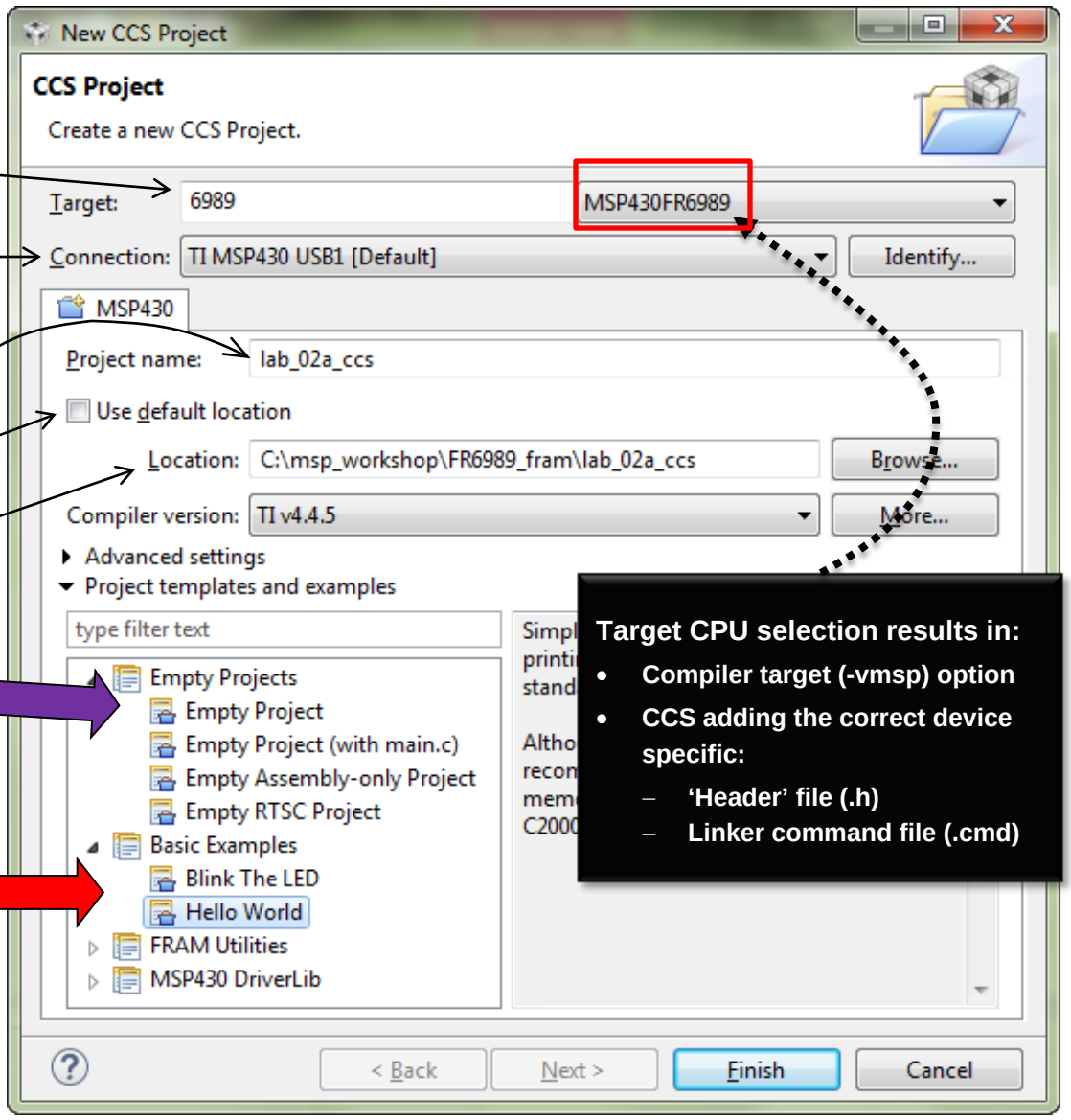
d) Don't use default location

e) Choose your *target's* `lab_02a_ccs` folder

**MSP432**
*Pick "Empty Project" because there isn't a "Hello World" template*

f) Select template: `Hello World`

g) Click 'Finish' when done.




**Target CPU selection results in:**

- **Compiler target (-vmsp) option**
- **CCS adding the correct device specific:**
  - **'Header' file (.h)**
  - **Linker command file (.cmd)**

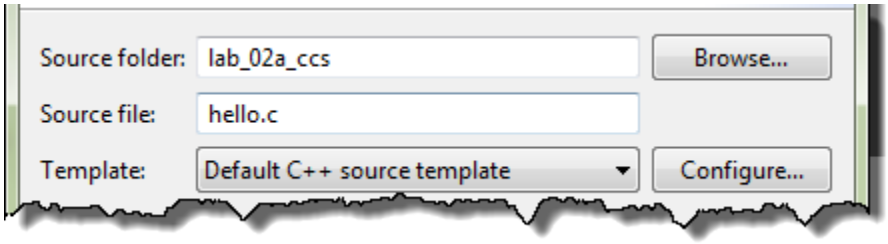
MSP432

**9. MSP432 (only) Add hello.c source file.**

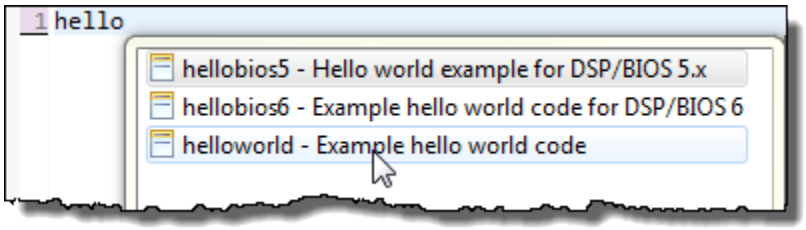
Since there wasn't a "Hello World" template, you will need to add a source file to the project.

a) Create a new source file in the project.

`File → New → Source File`



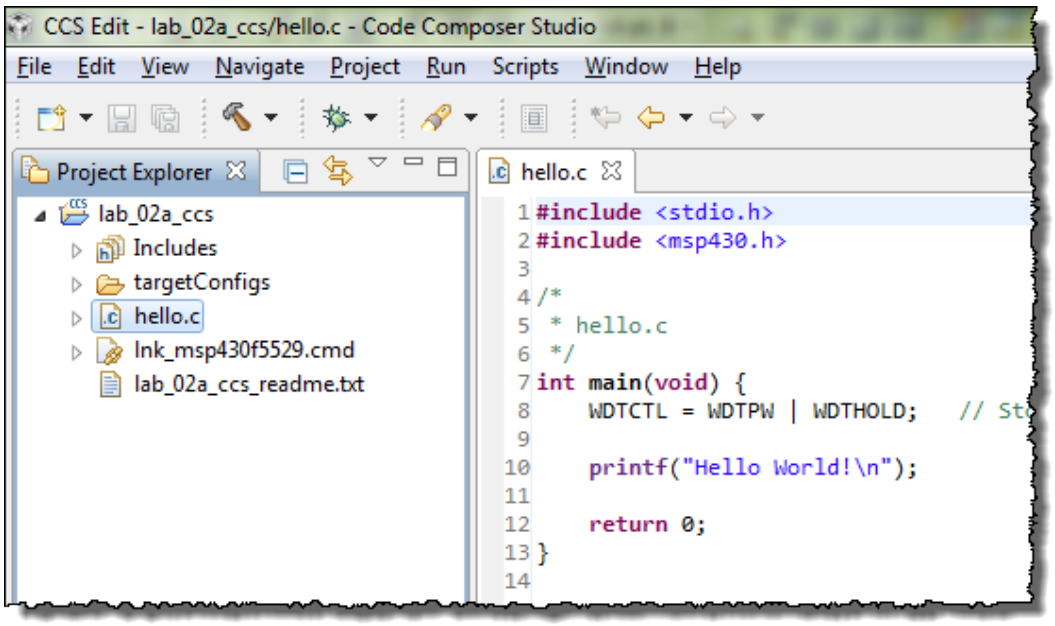

b) Delete the contents of the source file, then type "hello" and hit CTRL-TAB and CCS will let you select the "helloworld" template. CCS will auto-complete the hello world program.




**10. Code Composer will add the named project to your workspace.**

View the project in the Project Explorer pane.

`Click on the ▷ left of the project name to expand the project`




CCS includes other items based upon the ***Template*** selection. These might include source files, libraries, etc. (When choosing the *Hello World* template, CCS adds the file `hello.c` to the new project.)

**11. Open and view lab_02a_ccs_readme.txt.**

During installation, we placed the readme file into the project folder.

By default, Eclipse (and thus CCS) adds any file it finds within the project folder to the project. This is why the readme text file shows up in project explorer. Go ahead and open it up:

Double-click on: `lab_02a_ccs_readme.txt`

You should see a description of this lab similar to the abstract found in these lab directions.

> **Hint:** Be aware of this Eclipse feature. If – say in Windows Explorer – you absent-mindedly add a C source file to your project folder, it will become part of your program the next time you build.
>
> If you want a file in the project folder, but not in your program, you can exclude files from build:
>
> `Right-click on the file → Exclude from Build`

**12. Explore source code in `hello.c`.**

Open the file, if it's not already open.

`Double-click on hello.c in the Project Explorer window`

We hope most of this code is self-explanatory. Except for one line, it's all standard C code:

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

/*
 * hello.c
 */
int main(void) {
    WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer
    printf("Hello World!\n");
    return 0;
}
```

The only MSP-specific line is the same one we examined in the chapter discussion:

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

As the comment indicates, this turns off the watchdog timer (WDT peripheral). As we'll learn in Chapter 4, the WDT peripheral is always turned on (by default) in MSP devices. If we don't turn it off, it will reset the system – which is not what we usually want during development (especially during 'hello world').

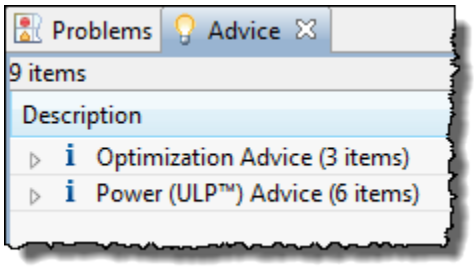
## Build The Code (ignore advice)

**13. Build your project using "the hammer" and check for errors.**

At this point, it is a good time to build your code to check for any errors before moving on.

`Just click the "hammer" icon:` 

It should build without any *Problems*, although you should see two sets of Advice: Optimization Advice and Power (ULP™) Advice.

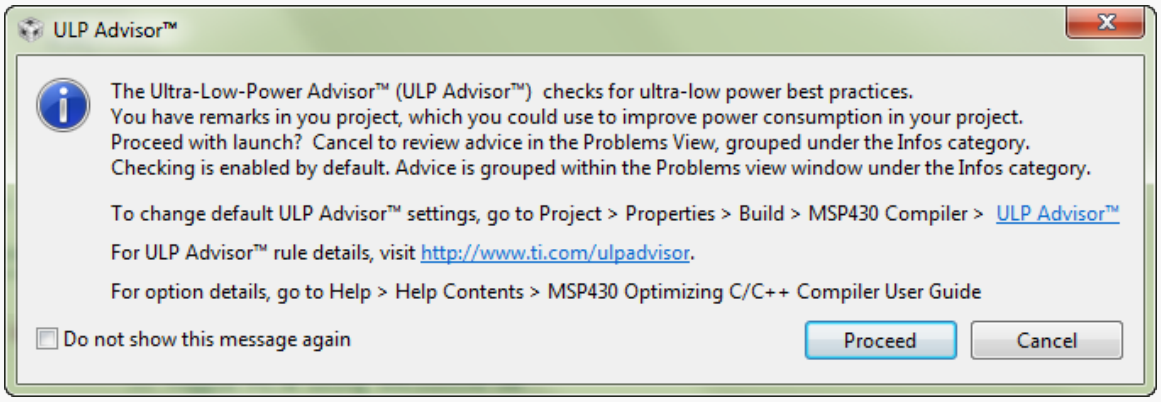



At this point, we're just going to ignore their advice. It's better to get code running first. Later, we return and investigate some of these items further.

If the program builds successfully, move to the next page to begin debugging. If you have problems getting it to build, please ask a neighbor, or your instructor for help.

### Sidenote: ULP Advisor

Sometime, when you launch the debugger (as we will soon), CCS will warn you that your code could be better optimized for lower power.




While we like the ULP Advisor tool, this usually comes up a long time before we are ready to start optimizing our performance. We recommend that you click the box:

☑ `Do not show this message again`

As the dialog above indicates, you can always go into your project's properties and enable or disable this advice. We will do this in a later chapter, when we're ready to focus on driving our every last Nano amp.

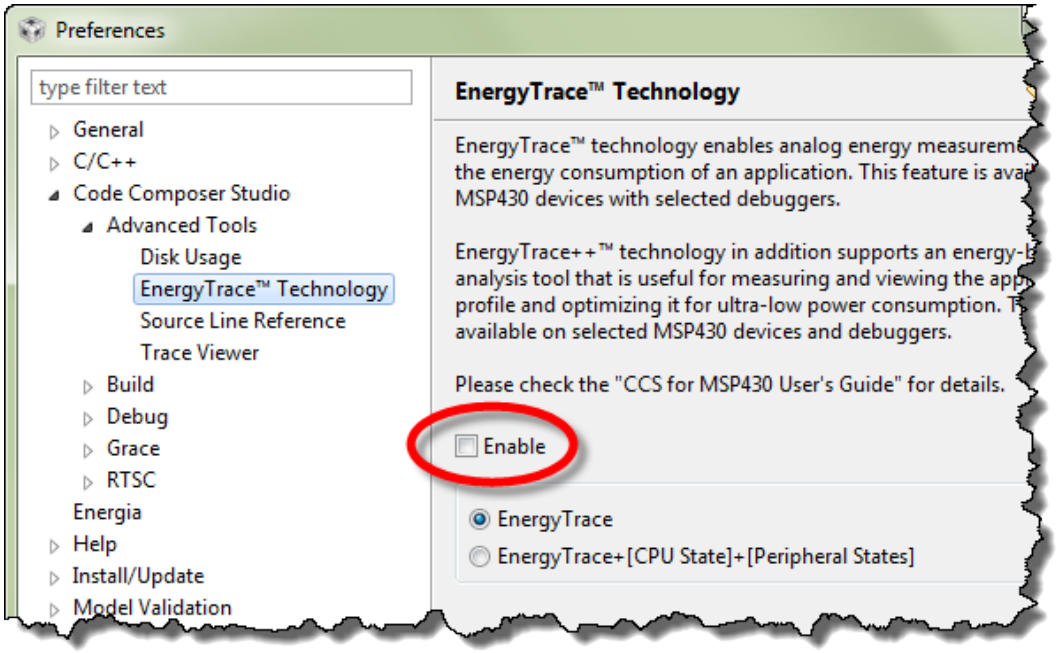
## Verify Energy Trace is 'Off'

We really like the new EnergyTrace features in CCS. It provides an incredible amount of information – but, we really don't need all of that info when we're just trying to get an LED to blink. Some versions of CCS turn this new feature 'on' by default. We suggest turning it off – for now. We'll re-enable it during the Low Power Optimzation chapter.

**14. Disable EnergyTrace (or verify it's disabled).**

`Window → Preferences`

`Code Composer Studio → Advanced Tools → EnergyTrace™ Technology`



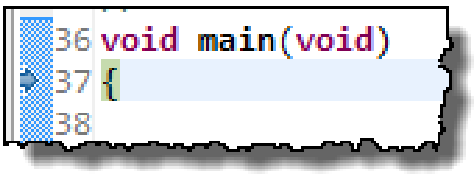

## Debug The Code

**15. Debug your program.**

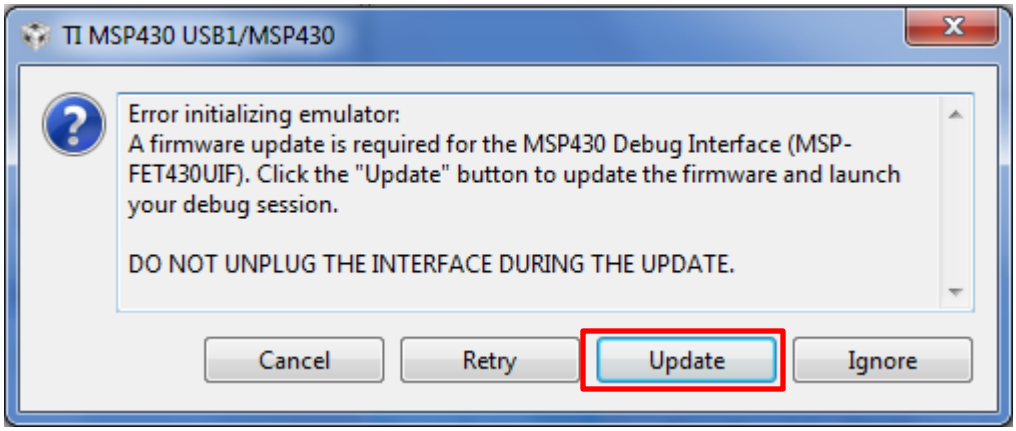
Clicking the Debug button will: Build the program (if needed); Launch the debugger; Connect to Target; and Load your program

`Click the BUG toolbar button:` 

Your program will now download to the target board and the PC will automatically run until it reaches `main()`, then stop as shown:



***Note:*** The first time you Launch a debugger session, you may encounter the following dialog:




This occurs when CCS finds that the FET firmware – that is, the firmware in your Launchpad's debugger – is out-of-date. We recommend that you choose to update the firmware. Once complete, CCS should finish launching the debugger.

## Connection Problems - Troubleshooting

If the error "*cannot connect to target*" appears, the problem is most likely due to:

- No target configuration (.ccxml) file
- Wrong board/target config file or both – i.e. board does not match the target config file
- Bad USB cable
- Windows USB driver is incorrect – or just didn't get enumerated correctly

If you run into this, check for each of these possibilities. In the case of the Windows USB driver try:

- Unplugging the USB cable and trying it in a different USB port. (Just changing ports can often get Windows to re-enumerate the device.
- Open Windows Device Manager and verify the board exists and there are no warnings or errors with its driver.
- If all else fails, ask your neighbor (or instructor) for assistance.

**16. Run the code.**

Now, it's finally time to RUN or "Play". ► Hit the **Resume** button: 

*The button is called 'Resume', though we may end up calling it 'Play' since that's what the icon looks like.*

**17. Pause the code.**

To stop your program running, ► click the **Suspend** button to pause): 

---

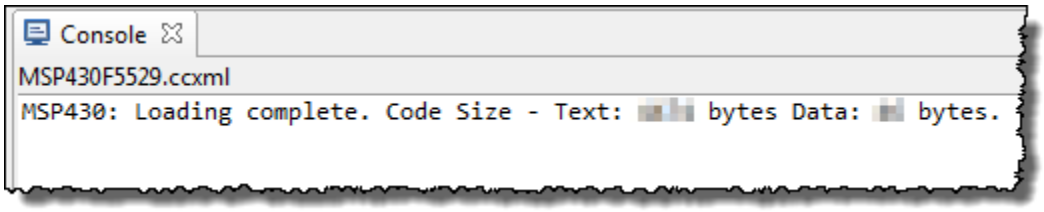
**Warning: Suspend** is different than **Terminate** !!!

If you click the Terminate button, the debugger – and your connection to the target – will be closed. If you're debugging and just want to view a variable or memory, you will have to open a new debug session all over again. Remember to ***pause*** and think, before you halting your program.

---

**18. Did printf work?**

Did "Hello World!" show up in your console window?




It works for the MSP432, but not for the other LaunchPads. ☹

**19. Let's Terminate the debug session and go fix "their" project.**

This time we really want to terminate our debug session.

`Click the red Terminate button:` 

This closes the debug session (and Debug Perspective). CCS will switch back to the *Edit* perspective. You are now completely disconnected from the target.

MSP432

**Note:** *Since "Hello World" for the MSP432 worked fine, go ahead and skip to step 23.*

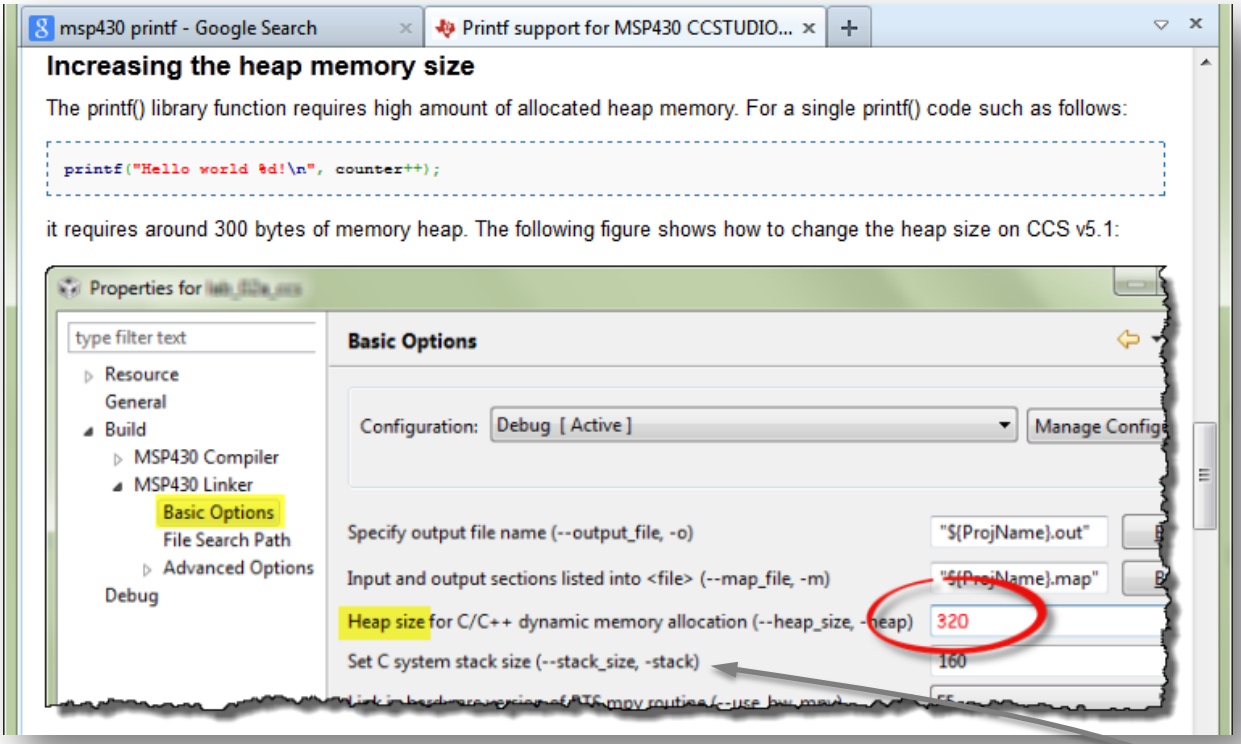
## Fix The Example Project

**20. What is wrong? Increase the heap size.**

Per the wiki suggestion, let's increase the heap size to 320 bytes.

`Right-click project → Properties → MSP430 Linker → Basic Options`

Increase *Heap size* to: **320**




You can find a description of this problem by searching the internet for: "msp430 printf"

From that, you should find a MSP wiki page that describes how to get printf() to work:

http://processors.wiki.ti.com/index.php/Printf_support_for_MSP430_CCSTUDIO_compiler

(In fact, this is how we figured out how to solve the problem.)

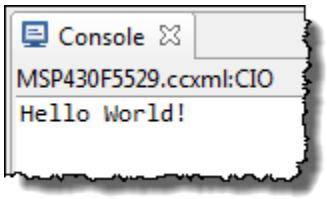
**Hint:** As a side note, if you look just below the entry for setting the Heap size, you will see the setting for Stack size. This is where you would change the stack size of you system, if you ever need to do that.

## Build, Load, Connect and Run … using the Easy Button

**21. Rebuild and Reload your program.**

First, make sure you terminated your previous debug session and you are in the Edit perspective.

**22. Once the program has successfully loaded, ► run it.**




---

**FR4133**

***Note:*** The ‘FR4133 may stop half-way through the printf() routine – if this happens, just click the Run/Resume button again and it should continue.

You can avoid this unintended breakpoint by setting the FRAM waitstates to 0. The default waitstates value on the ‘FR4133 is 1, which covers running the processor up to its full speed. If you stay at or below 8MHz, then they can be set to 0.

Eliminating this pause isn’t really necessary for this lab, though we’ll need to employ this trick for `lab_4b_wdt`. By Lab 4, we’ll have learned how to change waitstates using Driver Library; for now, adding this line of code somewhere before the call to printf() will solve the problem:

```
FRCTL0 = FRCTLPW | NWAITS_0;
```

---

**23. Terminate and Close the `lab_02a_ccs` project.**

Terminate the debug session and then close the project. Closing a project is both handy and prevents errors.

`Right-click project → Close Project`

If your source file (hello.c) was open, notice how closing the project also closes most source files. This can help prevent errors. *(Wait until you’ve spent an hour editing a file – with it not working – only to find you were editing a file with the same name, but from a different project. Doh!)*

You can quickly reopen the project, when and if you need to.

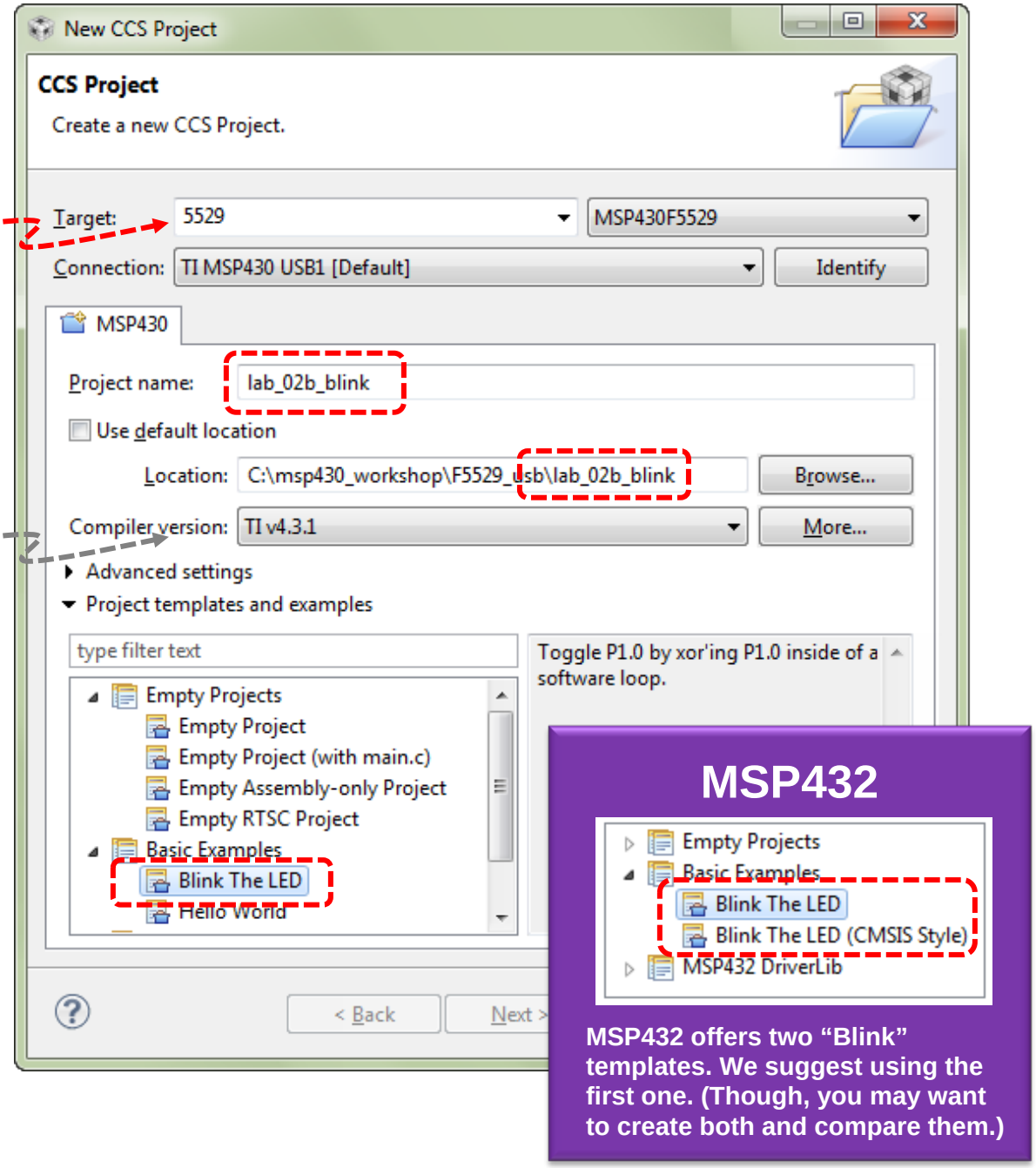
# Lab 2b – My First Blinky

We plan to get into all the details of how GPIO (general purpose input/output) works in the next chapter. At that time, we will also introduce the MSPWare DriverLib library to help you program GPIO, as well as all the other peripherals on the MSP.

In the lab exercise, we want to teach you a few additional debugging basics – and need some code to work with. To that end, we're going to use the Blink template found in CCS. This is generic, low-level MSP code, but it should suite our purposes for now.

## Create and Examine Project

1. **Create a new project (`lab_02b_blink`) with the following properties:**




Make sure to select the target for the LaunchPad you are using

*Use the default compiler version (which will be more recent than 4.3.1)*

2. **Let's quickly examine the code that was in the template.**

   This code simply blinks the LED connected to Port1, Pin0 (often shortened to P1.0).

```
#include <msp430.h>

int main(void) {
    WDTCTL = WDTPW | WDTHOLD;     // Stop watchdog timer

    P1DIR |= 0x01;                // Set P1.0 to out-put direction

    for(;;) {
        volatile unsigned int i; // volatile to prevent optimization

        P1OUT ^= 0x01;            // Toggle P1.0 using exc

        i = 10000;                // SW Delay
        do i--;
        while(i != 0);
    }
```

**MSP432 CMSIS style will look a bit differently, although with the same end result.**

   Other than standard C code which creates an endless loop that repeats every 10,000 counts, there are three MSP-specific lines of code.

   - As we saw earlier, the Watchdog Timer needs to be halted.
   - The I/O pin (P1.0) needs to be configured as an <u>output</u>. This is done by writing a "1" to bit 0 of the Port1 direction register (P1DIR).
   - Finally, each time thru the for loop, the code toggles the value of the P1.0 pin.
     *(In this case, it appears the author didn't really care if his LED started in the on or off position; just that it changed each time thru the loop.)*

| | |
|---|---|
| **Hint:** | As we mentioned earlier, we will provide more details about the MSP GPIO features, registers, and programming in the next chapter. |

## Build, Load, Run

3. **Build the code. Start the debugger. Load the code.**

   If you don't remember how, please refer back to *lab_02a_ccs*.

4. **Let's start by just running the code.**

   `Click the` ***`Resume`*** `button on the toolbar (or press` ***`F8`***`)`

   You should see the LED toggling on/off.

5. **Halt the debugger by clicking the "Suspend" button … <u>don't</u> terminate!**

## Restart, Single-Step, Run To Line

**6. Restart your program.**

Let's get the program counter back to the beginning of our program.

`Run` → `Restart` - or - use the Restart toolbar button: 

Notice how the arrow, which represents the Program Counter (PC) ends up at main() after your restart your program. This is where your code will start executing next.

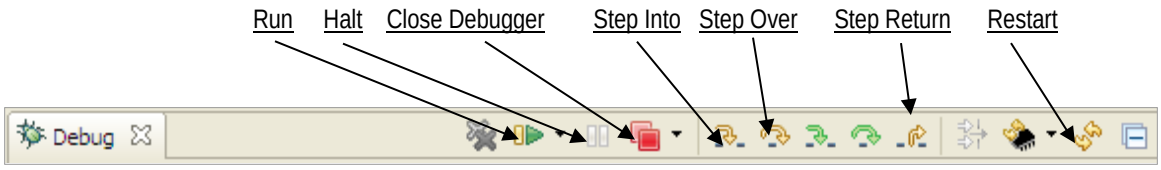
```
21 #include <msp430.h>
22
23 int main(void) {
24     WDTCTL = WDTPW | WDTHOLD;
25     P1DIR |= 0x01;
26
```

In CCS, the default is for execution to stop whenever it reaches the main() routine.

By the way, ***Restart*** starts running your code from the entry point specified in the executable (.out) file. Most often, this is set to your reset vector. On the other hand, ***Reset*** will invoke an actual reset. *(Reset will be discussed further in Chapter 4.)*

**7. Single-step your program.**

With the program halted, click the **Step Over (F6)** toolbar button (or tap the F6 key):




Notice how one line of code is executed each time you click *Step Over*; in fact, this action treats functions calls as a single point of execution – that is, it steps *over* them. On the other hand *Step Into* will execute a function call step-by-step – go *into* it. Step Return helps to jump back out of any function call you're executing.

| | |
|---|---|
| **Hint:** | You probably won't *see* anything happen until you have stepped past the line of code that toggles P1.0. |

**8. Single-step 10,000 times**

`Try stepping over-and-over again until the light toggles again…`

Hmmm… looking at the count of 10,000; we could be single-stepping for a long time. For this, we have something better…

**9. Try the *Run-To-Line* feature.**

Click on the line of code that toggles the LED.

Click on the line: `P1OUT ^= 0x01;`

Then `Right-click and select` ***`Run To Line`*** (or hit Ctrl-R)

`Single-step once more to toggle the LED`

10. **Set a breakpoint.**

    There are many ways to set a breakpoint on a line of code in CCS. You can right-click on a line of code to toggle a Breakpoint. But the easiest is to:

    `Double-click the blue bar on the line of code`

    For example, you can see we have just set a breakpoint on our toggle LED line of code:

    Once a breakpoint is set, there will be a blue marker that represents it. By **double-clicking** in this location, we can easily add or remove breakpoints.


11. **Run to breakpoint.**

    Run the code again. Notice how it stops at the breakpoint each time the program flow encounters it.

    `Press F8` (multiple times)

    You should see the LED toggling on or off each time you run the code.

12. **Terminate your debug session.**

    When you're done having fun, terminate your debug session.

13. **Close the project.**

    If any edit windows are still open after closing the project, we recommend closing them, too.

# (Optional) Lab 2c – Restoring the OOB

Do you want to go back and run the original Out-Of-Box (OOB) demo that came on your Launchpad board?

Unfortunately, we overwrote the Flash memory on our microcontroller as downloaded our code from the previous couple lab exercises. In this part of the lab, we will build and reload the original demo program. Note: sometimes the Out-Of-Box demo is also referred to as the UE (User Experience) demo.

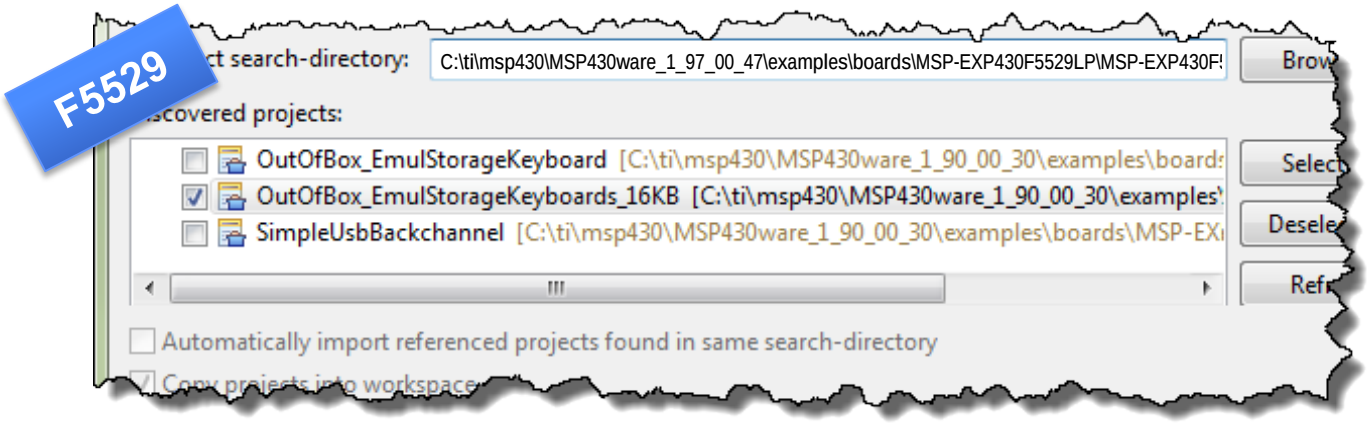
1. **Import OOB demo project.**

   The out-of-box demo can be found in the latest version of MSPWare.

   `Project → Import CCS Projects...`

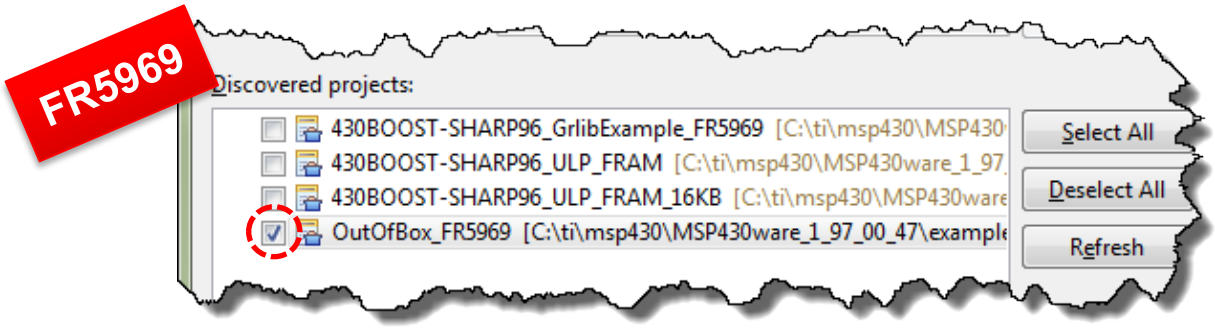
For '**F5529** users, import the project **`OutOfBox_EmulStorageKeyboards_16KB`** from the following:

`MSPWare_2_21_00_39\examples\boards\MSP-EXP430F5529LP\MSP-EXP430F5529LP Software Examples`



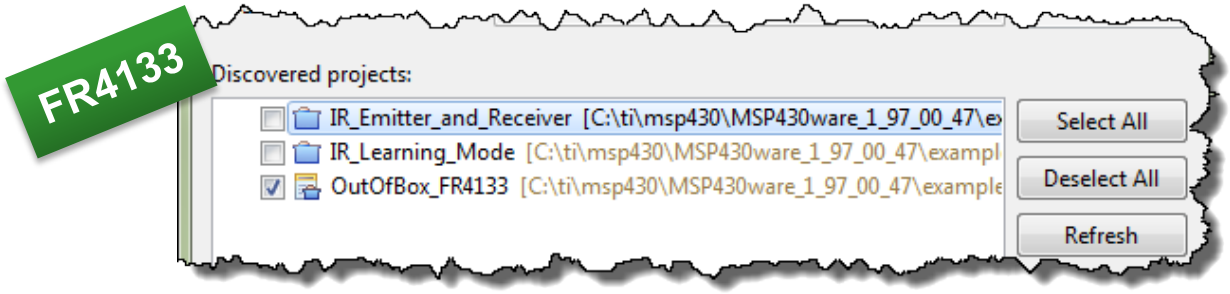

For '**FR5969** users, import the project **`OutOfBox_MSP430FR5969`** from:

`MSPWare_2_21_00_39\examples\boards\MSP-EXP430FR5969\MSP-EXP430FR5969 Software Examples`



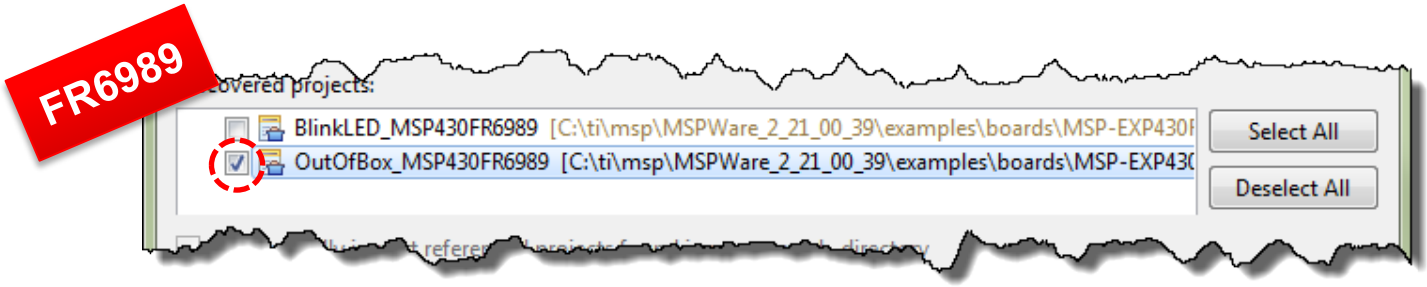

For '**FR4133** users, import the project **`OutOfBox_MSP430FR4133`** from:

`MSPWare_2_21_00_39\examples\boards\MSP-EXP430FR4133\MSP-EXP430FR4133_Software_Examples\`



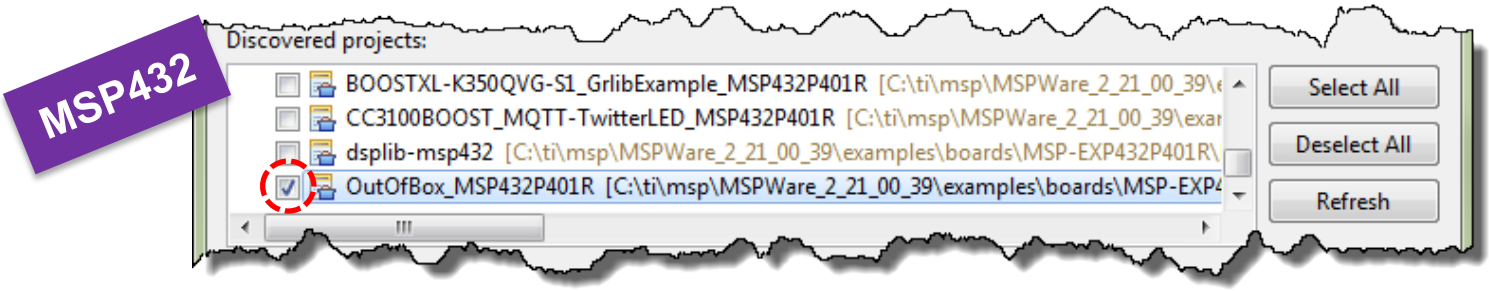
For **'FR6989** users, import the project **`OutOfBox_MSP430FR6989`** from:

`MSPWare_2_21_00_39\examples\boards\MSP-EXP430FR6989\MSP-EXP430FR6989_Software_Examples`




For **MSP432** users, import the project **`OutOfBox_MSP432P401R`** from:

`MSPWare_2_21_00_39\examples\boards\MSP-EXP432P401R\MSP-EXP432P401R_Software_Examples`




In all cases, if you have a choice, check *"Copy projects into workspace"* and then hit the *Finish* button.

2. **Build the out-of-box demo project that you just imported.**

3. **Click the Debug button to launch the debugger, and load the program to flash.**

   In this excercise, we're not that interested in running the code within the debugger, rather we're just using the debug button as an easy way to program our device with the demo program. Later labs will explore the various features on display in the demos.

4. **Terminate the debugger and close the project.** (You can run it within the debugger, but running it outside the debugger 'proves' the program is actually in Flash or FRAM memory.)

5. **Unplug the Launchpad from your PC and plug it back in.**

   The original demo, which was just re-programmed into Flash/FRAM, should now be running. *(You can refer back to Lab1 if you have questions on how to use the demo.)*

# (Optional) Lab 2d – MSP430Flasher

MSP432

As of this writing, the MSP430Flasher does not yet support the MSP432.

The MSP430Flasher utility lets you program a device without the need for Code Composer Studio. It can actually perform quite a few more tasks, but writing binary files to your board is the only feature that we explore in this exercise. The tool is documented at:

http://processors.wiki.ti.com/index.php/MSP430_Flasher_-_Command_Line_Programmer

***Note:*** The MSP430Flasher utility is quite powerful; with that comes the need for caution. With this tool you could – if you are being careless – lock yourself out of the device. This is a feature that is appreciated by many users, but not during development. The batch files we're provide should'nt hurt your Launchpad – but you should treat this tool with caution.

## Programming the OOB demo using MSP430Flasher

**1. Verify MSP430Flasher installation.**

Where did you install the *MSP430Flasher* program? Please write down the path here:

_____/MSP430Flasher.exe

```
Hint: If you have not installed this executable, either return to
      the installation guide to do so, or you may skip this optional
      lab exercise.
```

**2. Edit / Verify DOS batch program in a text editor.**

We created the `ue.bat` file to allow you to program the User Experience OOB demo to your Launchpad without CCS. Open the following file in a text editor:

```
C:\msp_workshop\<target>\lab_02d_flasher\oob.bat
```

Verify – and modify, if needed – the two directory paths listed in the .bat file. For example:

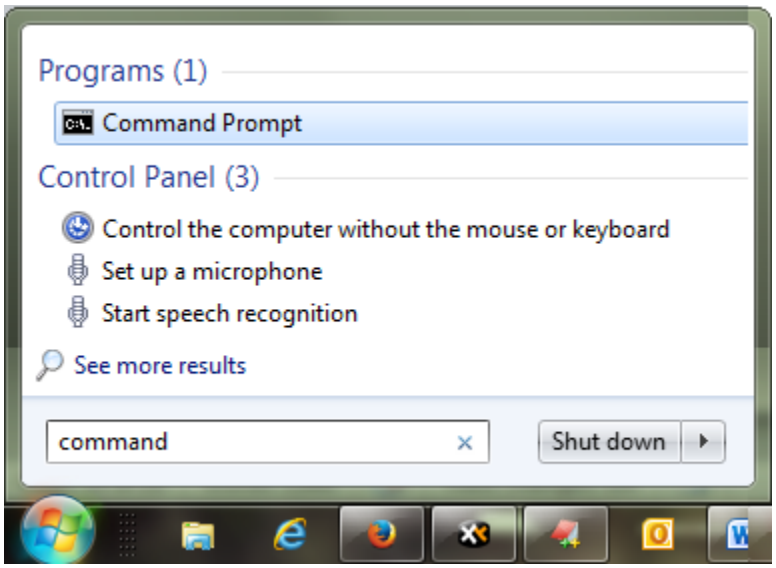
```
CLS
C:\ti\MSP430Flasher_1.3.4\MSP430Flasher.exe -n MSP430FR6989 -w
"C:\msp_workshop\FR6989_fram\workspace\OutOfBox_MSP430FR6989\Debug\OutOfBox_MSP430FR6989.txt" -v
```

```
Where: -n is the name of the processor to be programmed
       -w indicates the binary image
       -v tells the tool to verify the image
```

We used the default locations for MSP430Flasher and our lab exercises. You will have to change them if you installed these items to other locations on your hard drive.

**3. Open up a DOS command window.**

One way to do this is by typing “command” in Windows “Start” menu, then hitting Enter.




After starting command, it should open to something similar to this:


**4. Navigate to your `lab_02d_flasher` folder.**

The DOS command for changing directories is: “*cd*”

```
cd C:\msp_workshop\<target>\lab_02d_flasher\
```

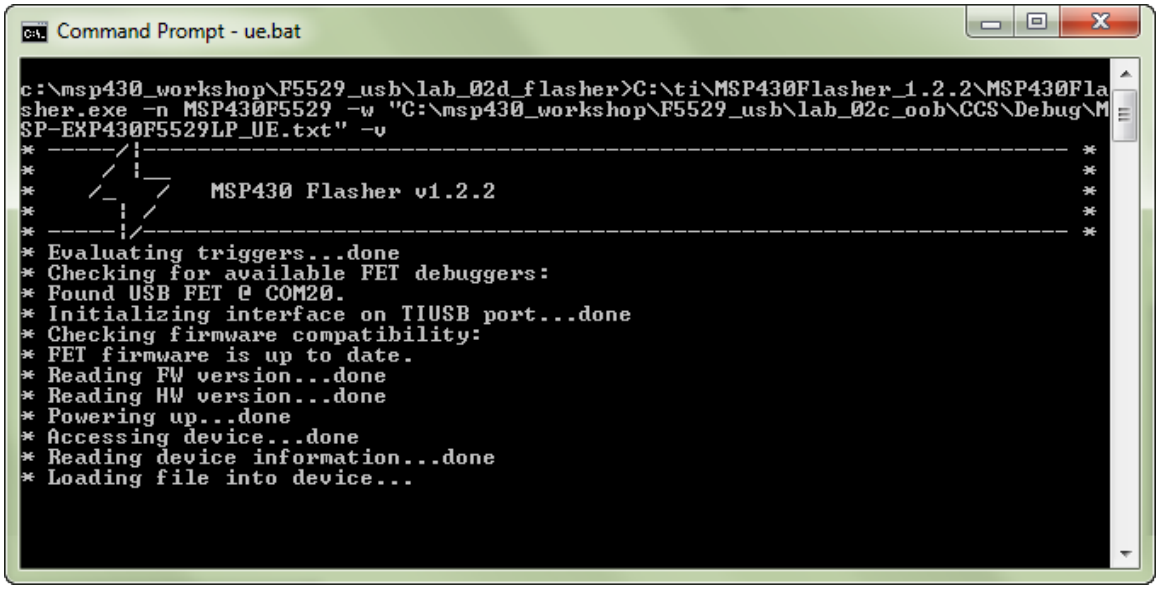
Once there, you should be able to list the directories contents using the *dir* command.

```
dir
```

**5. Run the batch file to program the out-of-box executable to your board.**

```
oob.bat ↵
```

You should see it running … here's a screen capture we caught mid-programming:




If the information echoed by MSP430Flasher went by too fast on the screen, you can review the log file it created. Just look for the 'log' folder inside the directory where you ran MSP430Flasher.

**6. Once again, verify the Launchpad program works.**

| | |
|---|---|
| **Hint:** | If you have trouble finding the binary hex file (or in the next section, creating the binary hex file), we created a subdirectory in Lab2c called "local_copy" and placed the two binary files along with their respective .bat files. |

# Notes:

## Programming Blinky with MSP430Flasher

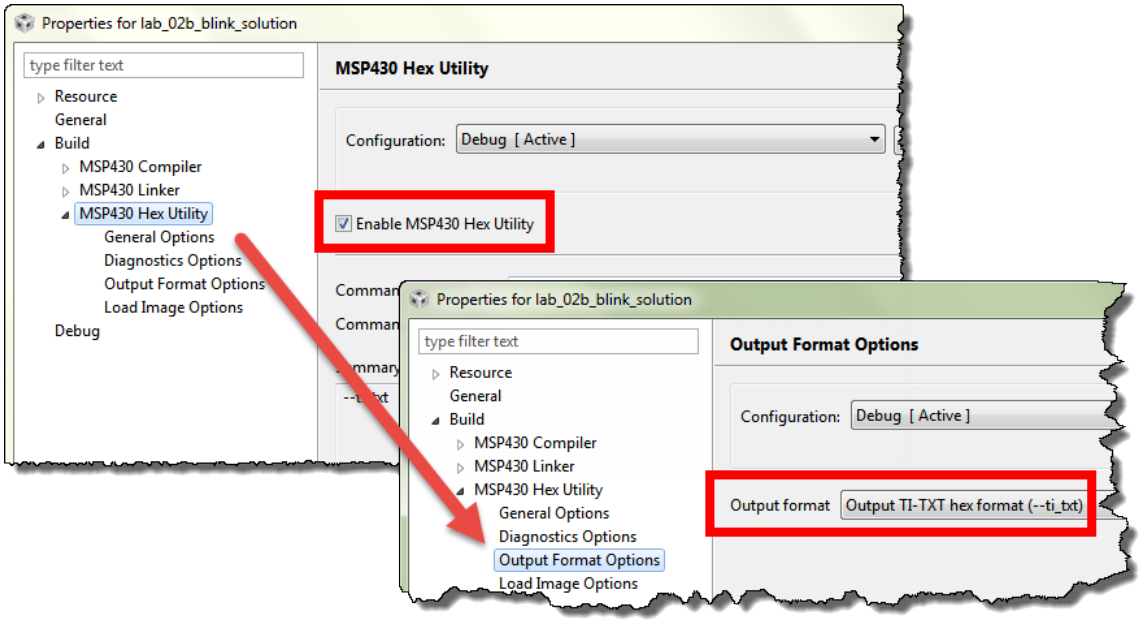
We can use this same utility to burn other programs to our target. Before we can do that, though, we need to create the binary file of our program. The UE app already did this as part of their build process, but we need to make a quick modification to our project to have it build the correct binary format for the flasher tool.

7. **Open your `lab_02b_blink` project.**

8. **Open the project properties for you project.**

   With the project selected, hit *`Alt-Enter`*.

9. **(CCSv6) Change the following settings in your project, as shown below:**




> **Hint:** This procedure is documented at: [http://processors.wiki.ti.com/index.php/Generating_and_Loading_MSP430_Binary_Files](http://processors.wiki.ti.com/index.php/Generating_and_Loading_MSP430_Binary_Files).

10. **Rebuild the project.**

    If you don't rebuild the project, the .txt binary might not be generated if CCS thinks the program is already built.

    ```
    Clean the project
    Build the project
    ```

11. **Verify that `lab_02b_blink.hex (or lab_02b_blink.txt)` was created in the `/Debug` directory.**

    ```
    lab_02b_blink.txt
    ```

12. **Open `blink.bat` with a text editor and verify all the paths are correct.**

    ```
    C:\msp_workshop\<target>\lab_02d_flasher\blink.bat
    ```

    Note that you may need to change the name of the file in .bat depending on the file extension needed for your program (either .hex or .txt).

13. **Run `blink.bat` from the DOS command window.**

    When done programming, you should see the LED start blinking.

## Cleanup

14. **Close your `lab_02b_blink` project.**
15. **You can also close the DOS command window, if it's still open.**