Code Composer Studio™

Fundamentals Workshop with the MSP430 Lab Material
LaunchPad: Hardware Setup

Connect mini-USB cable from PC to board
LAB conventions

• Lab steps are numbered for easier reference
  1. ...
  2. ...

• Explanations, notes, warnings are written in blue
  – Warnings are shown with 🚨
  – Information is marked with 🔄
  – Tips and answers are marked with💡
  – Questions are marked with❓
LAB 1: BLINK LED
Blink LED Example: 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 concepts
  – Basics of working with views
  – Debug launch
  – Debug control
  – Profile Clock
  – Local History
  – Build Properties
  – Changing compiler versions
Workspace

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

1. Create a new project in the CCS Workspace by going to menu File → New → CCS Project
2. Fill in the boxes as shown in the image. Select the Blink LED template and double check the exact device variant with that in the launchpad socket
3. Click Finish
ULP Advisor Message

- A message highlighting the ULP Advisor may 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
Build ‘BlinkLED’ Project

1. In the *Project Explorer* view, select the BlinkLED project (it should appear [Active – Debug]) to make it active

2. Click on the Build icon in the toolbar. The project will start the building process.
   
   Warning: When using a newly installed CCS, the tool will take extra time to build the Runtime Support Library (RTS) at this time. This is normal and will only happen once.

3. The *Console* view will appear at the bottom with build messages (information, warnings, errors) as the project builds

4. The *Problems* view will also appear at the bottom to highlight any possible build errors.
   
   Warning: When building the RTS, some warning messages will appear in the problems view and can be ignored.

5. If the build is successful, the *Problems* view will contain no errors (warnings can still be seen)
Debug ‘BlinkLED’ Project

1. Click on the “green bug” button – make sure the project is selected!

When pressing the debug button, several actions are done automatically:
- Prompt to save source files
- Build the project (incrementally)
- Start the debugger (CCS will switch to the CCS Debug perspective)
- Connect CCS to the target
- Load the program on the target
- Run to main()
Build, Load/Flash the Program

Switched to ‘CCS Debug’ perspective

Program counter at ‘main’

Code size information displayed in console view
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
Blink LED1

1. Press the Run button to run the program

LED1 on the Launchpad should now be blinking
Debugging: Using Watchpoints

A Watchpoint is a type of breakpoint that monitors activity on a memory address. In this step we will set a watchpoint to halt the CPU anytime the LED will toggle.

1. Press the `halt/suspend` button ⏸️ to halt the running program
   - The code should stop somewhere in the `for` loop
2. Open the Breakpoints view
   - `View → Breakpoints`
3. Create a new Watchpoint
   - `Location`: set it to “i”
4. 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

1. Run the target again. Execution will automatically halt when i is 256.

You may need to bring the variables tab into focus.
More Debugging

- Investigate other debugging views (Open via View menu)
  - Memory Browser
  - Registers
  - Disassembly (see next slide)

- Set breakpoints
  - Double click on a source line to set/clear
  - See list of breakpoints with the Breakpoints view

- Use the buttons in the Debug view to:
  - Restart the program
  - Source stepping
  - Assembly stepping
View: Disassembly

1. Go to the main() symbol in the Disassembly view by typing “main” in the address field and hit <ENTER>
   - You can see the current location of the PC (small blue arrow) and any breakpoints (small blue circles)

2. Toggle the **Show Source** button. Note the toggling of interleaved source with the disassembly
Remove the Watchpoint

1. Go to the Breakpoints view
   - Select *Remove all Breakpoints*
   - Select “Yes” when prompted

2. Restart execution
   - Go to the Debug View
   - Click on the Restart button
   - The program counter should be back at main
Debugging: Using Profile Clock

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 like cache hits/misses, bus stalls, etc.

1. Enable the Clock
   – Menu Run → Clock → Enable
   – The clock will now be displayed on the status bar

2. Add a breakpoint on line 35 “}” of the for loop by double clicking

3. Click the Run button
   – Clock should now show 24 cycles

Tip: Double-clicking on the clock icon will reset the count to ‘0’
Remove the Watchpoint

1. Go to the Breakpoints view
   – Select Remove all Breakpoints
   – Select “Yes” when prompted

2. Restart execution
   – Go to the Debug View
   – Click on the Restart button
   – The program counter should be back at main
View: Local History

 CCS keeps a local history of source changes
 1. Switch to the CCS Edit perspective
 2. Right-click on a file in the editor and select Team → Show Local History

 CCS also keeps project history
 1. Recover files deleted from the project
 2. Right-click on the project and select Recover from Local History in the context menu
Terminate the Debug Session

1. Go to the Debug View
2. Click on the terminate button
3. This will kill the debugger and return you to the Edit perspective
LAB 2 : 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 the Project

1. Create a new project in the CCS Workspace by going to menu *File → New → CCS Project*
2. Fill in the boxes as shown in the image. This time the project will be created outside the workspace
   - Uncheck the box to use default location
   - Specify `C:\ti\Projects\Temperature`
3. Click *Finish*
Add Temperature Sensor Source Code

1. Open Windows explorer and browse to
   - C:\TI\LaunchPad\temperature_sensor

2. Drag and drop ‘Temperature_Sense_Demo_G2xxx.txt’ to the ‘Temperature_Sense_Demo’ project. Use the file that matches the MSP430 device being used (G2231 or G2553) Make sure the file is dragged to the ‘Temperature_Sense_Demo’ project

3. In the dialog popup, select the option to ‘Copy Files’ and hit ‘OK’
Add Temperature Sensor Source Code

1. 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’

2. 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.
Build and Load/Flash the Program

1. Make sure the ‘Temperature_Sense_Demo’ project is in ‘Focus’. Then use the ‘Debug’ button.
Temperature Sense Demo

1. Press the ‘run’ button to run the program
   - LED1 (red) and LED2 (green) on the LaunchPad should now alternate blinking
Temperature Sense Demo: Debugging

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

1. 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.

2. 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

1. Select ‘Temperature_Sense_Demo’ in the left panel and the ‘Program’ tab in the right panel
2. Under ‘Loading options’, select ‘Load symbols only’
3. Then select ‘Apply’ and then ‘Debug’
Loading Symbols for Flashed Program

1. 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)

2. 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

Callstack displayed

Source code is found automatically
Changing Project Properties

1. Make sure you are in the *CCS Edit* perspective
2. Right click on the Temperature_Sense_demo project and select *Properties*
Changing Build Options

Build options are set per build configuration

1. Change your Configuration to *Release*

2. Change the optimization settings
   - Go to the *Build → MSP430 Compiler → Optimization*
   - Change the optimization level to 3
   - Change the speed vs size to 0

3. Click OK
Changing Build Options

1. Change the active configuration to *Release*
   - Right click on the Project
   - Select *Build Configurations* → *Set Active* → *Release*

2. 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 clicking on the arrow beside the build button and selecting the configuration you want to build*
   - Select *Release* and it will build this configuration
   - The active configuration is indicated by the Checkmark
1. Launch ‘New CCS Project’ Wizard --
   - In ‘CCS Edit’ perspective, Project -> ‘New CCS Project’
2. Fill in the fields as shown on the right
3. Expand the ‘Advanced Settings’ and check the ‘Compiler version:’ dropdown menu
   - Note there is only one option (v4.1.x)
   - Select ‘More…’ button next to the ‘Compiler version:’ field
Changing the Compiler Version

1. Click on General
2. Click on the More… button beside the Compiler version Ti v4.1.x
3. Check the option Select new build-tool from file-system
4. Browse to the location of the new compiler tools and click OK
   C:\TI\MSP430 3.3.2 Compiler

强迫

CCS will determine what compiler is located there and select it for your active configuration.
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.x (default version that ships with CCSv5.4) 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’
Blink LED Example: Exercise Summary

- At this point you experimented the following concepts:
  - Workspaces
  - Welcome screen / Resource Explorer
  - Project concepts
  - Basics of working with views
  - Debug launch
  - Debug control
  - Profile Clock
  - Local History
  - Build Properties
  - Changing compiler versions
  - Debug symbols
  - Profile
LAB 3: SHARING PROJECTS
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

1. Launch ‘New CCS Project’ Wizard
   - Select ‘New Project’ from the Welcome page
2. Fill in the fields as shown in the right
   - Use compiler version **4.1.x (not 3.2)**
3. Select ‘Finish’ when done
4. Generated project will appear in the Project Explorer view
5. Remove the generated ‘main.c’ file from the project
Create a Linked Resource Path Variable

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

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

1. Open Windows Explorer and browse to C:\TI\LaunchPad\msp_portable_project\source_file
   Drag and drop the ‘msplab.c’ file into the ‘msplab’ project
Link Source Files to Project

1. 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’
2. Files will now appear in the Project Explorer with the ‘link’ icon
Link Files to Project

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

1. Right-click on the project and select ‘Properties’
2. In the compiler ‘Include Options’, add the following entries to the list of include search paths:
   - `${CCS5_WORKSHOP_LAB}\msp_portable_project\header_file`
3. ‘${<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 cannot be used for build options. Use Build Variables when modifying build options.
Project Properties

1. 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
2. See the workspace level Linked Resource Path Variable that was created appears in the list
3. Variables may be edited here but changes will only be recorded at the project level (stored in the project files)
1. 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.
2. Links can be modified here with the ‘Edit…’ button.
3. Links can be converted to use an absolute path with the ‘Convert…’ button.
Project Properties

1. 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
2. See how the workspace level Build Variable that was created appears in the list
LAB 4: Ultra Low Power Advisor
Import and Build ULP Demo Project

1. Import ULP Demo project into the workspace
   - Import -> Existing CCS Eclipse Project
   - Project is located in:
     • C:\ti\LaunchPad\ulp_demo

2. Explore the one source file ‘ulp_demo.c’

3. Build it
ULP Diagnostics

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

```c
// Demonstrate rule 8.1 - Use 'static' and 'const' modifiers for local
// variables

int  ulp_demo_rule_8_1()
{
    static const int const_local = 10;
    return const_local;
}
```

Problems

0 errors, 18 warnings, 3 others

- Warnings (18 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 dete
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

```c
// Demonstrate rule 11.1 - Use lower bits for loop program control flow

#define F1 0x01
#define F2 0x02
#define F3 0x04
#define F4 0x08

unsigned flags_var = 0;

void ulp_demo_rule_11_1()
{
    while (flags_var & (F2 | F3))
    {
        do_some_processing();
    }
}
```

Problems

- 0 errors, 18 warnings, 2 others

Description

- **Warnings (18 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 discussed
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

```c
void ulp_demo_rule_13_1(int input)
{
    int i;
    for (i = input-1; i >= 0; i--)
        do_some_processing();
}
```

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

ULP Advisor > Rule 1.1 Ensure LPM Usage

What it means

Low power mode (LPM) usage is highly recommended. One of the necessary steps to achieve low power consumption in an MSP430 application is to minimize the time spent active mode and maximize the time in low power modes. Peripherals can be configured to operate without CPU intervention and the CPU only needs to wake up to perform critical tasks and quickly return to low power mode.

Risks, Severity

An application that always stays in active mode is not energy efficient and could significantly reduce lifetime for battery-powered applications.

Why it is happening

This remark is issued when no LPM-entering instruction can be found in any code file in the project.

Remedy

- Use low power modes in your application when applicable, i.e. while waiting for certain peripheral tasks to complete,
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
LAB 5 : GRACE
Create a New (Grace) Project

1. Start the Project Wizard
   – Select ‘New Project’ on the Welcome page

2. Fill in the fields as shown on the right
   – Select the ‘Blink LED from the CPU’ template under ‘Grace Examples’

3. Select ‘Finish’ when done
Grace – Configuration File
Grace – Building/Running the Example

1. Build and load the Grace example using the ‘Debug’ button
   – HINT: Make sure the new grace example project is in focus
2. Run the example
3. LED1 should be blinking
   – Note the blink rate
4. 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

Overview - This view describes the peripheral and shows you the most important use cases for it.

Select the ‘Basic User’ mode to configure the clock frequency.
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

Configure Clock Source

- Internal High Speed Clock Source
  - Internal DCO
  - Pre-calibrated DCO Values
  - Disable DCO

Select Clock Source

- Clock Source: DCOCLK
- Divider: Divide by 1
- Main System Clock (MCLK) 200.48 kHz
- Output MCLK: No MCLK Pins

- Clock Source: DCOCLK
- Divider: Divide by 1
- Sub System Clock (SMCLK) 200.48 kHz
- Output SMCLK: SMCLK Output OFF

- Clock Source from Low Speed External Clock Source 1
  - Select Clock Source
  - 12 kHz

- Divider: Divide by 1
- Auxiliary Clock (ACLK) 12 kHz

Power User Mode. This view includes all the configuration settings of the peripheral.
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