# TMS320F2837xD Microcontroller Workshop – Driverlib Supplement

## Lab Manual

**Kenneth W. Schachter**

***Revision 1.0***
***November 2020***

# Important Notice



# Revision History

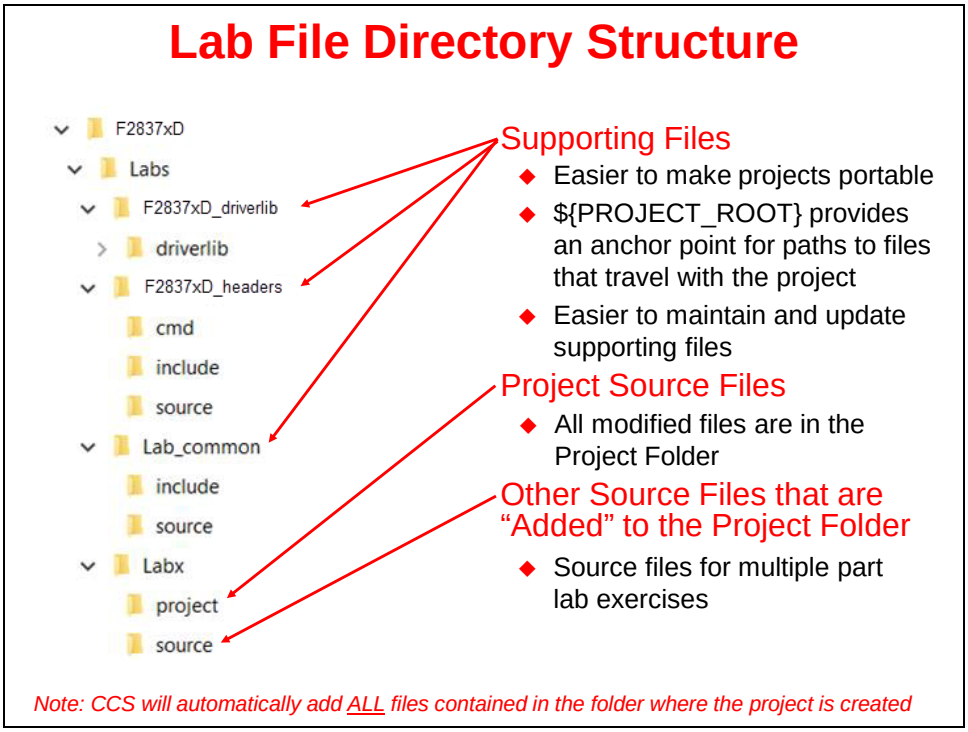
November 2020 – Revision 1.0

# Mailing Address

Texas Instruments
C2000 Training Technical
13905 University Boulevard
Sugar Land, TX 77479

# TMS320F2837xD Microcontroller Workshop – Driverlib Supplement

This supplement is intended to be used with the "TMS320F2837xD Microcontroller Workshop", revision 2.0 January 2018. The lab exercise files in this supplement are the equivalent Driverlib implementation of the bit-field header files used in the workshop lab exercises. All lab exercise directions have been updated in this manual for use with Driverlib. Each of the lab exercise source files has been completed and learning is accomplished by reviewing these files. However, creating some of the projects and configuring build options still remain as an exercise to reinforce the needed steps for using Driverlib. All lab exercises have been converted to Driverlib with the exception of Lab 2 exercise. This lab exercise does not use the bit-field header files or Driverlib. Therefore, no need exists for including it in this supplement. For additional details on how Driverlib is used and implemented with the lab exercises, reference the "TMS320F28004x Microcontroller Workshop", revision 1.0 July 2019. This workshop was originally developed with Driverlib and it is a good resource for learning more about the various Driverlib functions. For Driverlib functions specific to the TMS320F2837xD device, see the latest version of C2000Ware. The figure below is an outline of the lab file directory structure used in this supplement.




# Modification to F2837xD_device.h

***Note:*** An error exists in the 'F2837xD_device.h' file in C2000Ware_3_02_00_00 which was used to create this supplement. The 'F2837xD_device.h' file included with this supplement has been modified to fix this error and this modification will be included in the next version of C2000Ware.

Specifically, the 'F2837xD_device.h' file is missing the following #define which will cause build errors when combining the bit-field header files with Driverlib in a project:

```
//
// Common CPU Definitions:
//

#ifndef _TMS320C28XX_   ← MISSING IN FILE (added in supplement)
#define __cregister     ← MISSING IN FILE (added in supplement)
#endif //_TMS320C28xx_  ← MISSING IN FILE (added in supplement)

extern __cregister volatile unsigned int IFR;
extern __cregister volatile unsigned int IER;
```

Without this modification, errors in the 'F2837xD_device.h' for "#838 unrecognized cregister name..." for 'IER' and 'IFR' will be generated.

# Target Configuration

Details for setting up the target configuration file can be found the “TMS320F2837xD Microcontroller Workshop”, revision 2.0 January 2018 in Lab 2 exercise on page 2-20, steps 3 to 5. For completeness in this supplement, the procedure is also included below:

1. Open the target configuration dialog box. On the menu bar click:

   `File → New → Target Configuration File`

   In the file name field type **F2837xD.ccxml**. This is just a descriptive name since multiple target configuration files can be created. Leave the “Use shared location” box checked and select `Finish`.

2. In the next window that appears, select the emulator using the “Connection” pull-down list and choose “Texas Instruments XDS100v2 USB Debug Probe”. In the “Board or Device” box type **TMS320F28379D** to filter the options. In the box below, check the box to select “TMS320F28379D”. Click `Save` to save the configuration, then close the “F2837xD.ccxml” setup window by clicking the `X` on the tab.

3. To view the target configurations, click:

   `View → Target Configurations`

and click the plus sign (‘+’ or ‘>’) to the left of “User Defined”. Notice that the F2837xD.ccxml file is listed and set as the default. If it is not set as the default, right-click on the .ccxml file and select “Set as Default”. Close the Target Configurations window by clicking the `X` on the tab.

# Chapter Topics

# Lab 5: System Initialization

## ➢ Objective

The objective of this lab exercise is to perform the processor system initialization. Additionally, the peripheral interrupt expansion (PIE) vectors will be initialized and tested using the information discussed in the previous module. This initialization process will be used again in all of the lab exercises throughout this workshop.

The first part of the lab exercise will setup the system initialization and test the watchdog operation by having the watchdog cause a reset. In the second part of the lab exercise the interrupt process will be tested by using the watchdog to generate an interrupt. This lab will make use of the F2837xD Driver Library (Driverlib) to simplify the programming of the device. Please review these files, and make use of them in the future, as needed.

## ➢ Procedure

### Create a New Project

1. Create a new project (`File` → `New` → `CCS Project`) for this lab exercise. The top section should default to the options previously selected (setting the "Target" to "TMS320F28379D", and leaving the "Connection" box blank). Name the project **Lab5**. *Uncheck* the "Use default location" box. Using the "Browse…" button navigate to: `C:\F2837xD-DL\Labs\Lab5\cpu01` then click `Select Folder`. Set the "Linker Command File" to <none>, and be sure to set the "Project templates and examples" to "Empty Project". Then click `Finish`.

2. Right-click on `Lab5` in the Project Explorer window and add (copy) the following files to the project (`Add Files…`) from `C:\F2837xD-DL\Labs\Lab5\source`:

```
CodeStartBranch.asm                    Lab_5_6_7.cmd
DefaultIsr_5.c                         Main_5.c
device.c                               Watchdog.c
Gpio.c
```

### Project Build Options

3. Setup the build options by right-clicking on `Lab5` in the Project Explorer window and select "Properties". We need to setup the include search path to include the Driverlib files and common lab header files. Under "C2000 Compiler" select "Include Options". In the include search path box that opens ("`Add dir to #include search path`") click the `Add` icon (first icon with green plus sign). Then in the "Add directory path" window type (*one at a time*):

   **`${PROJECT_ROOT}/../../f2837xd_driverlib/driverlib`**

   **`${PROJECT_ROOT}/../../f2837xd_driverlib/driverlib/inc`**

   **`${PROJECT_ROOT}/../../Lab_common/include`**

   Click `OK` to include each search path.

4. Next, we need to setup the file search path for Driverlib. Under "C2000 Linker" select "File Search Path". The file search path box will open and in the include library file

section (“`Include library file or command file as input`”) click the `Add` icon. Then in the “Add file path” window type:

**`driverlib.lib`**

and click `OK`. Then in the library search path section (“`Add <dir> to library search path`”) click the `Add` icon. In the “Add directory path” window type:

**`${PROJECT_ROOT}/../../f2837xd_driverlib/driverlib/ccs/Debug`**

and click `OK`.

5. Next, we need to setup the predefined symbols. Under “C2000 Compiler” select “Predefined Symbols”. In the predefined name box that opens (“`Pre-define NAME`”) click the `Add` icon. Then in the “Enter Value” window type **CPU1**. This name is used in the project to conditionally include the code specific to CPU1. Click `OK` to include the name and then click the `Add` icon again. Next, in the “Enter Value” window type **_LAUNCHXL_F28379D** (note the leading underscore). This name is used in the project to conditionally include clock configuration, #defines for pin numbers, and other GPIO configuration code specific to the LaunchPad (rather than the controlCARD). This conditional code is located in the `device.h` file. Click `OK` to include the name. Finally, click `Apply and Close` to save and close the Properties window.

## Memory Configuration

6. Open and inspect the linker command file `Lab_5_6_7.cmd`. Notice that the user defined section “`codestart`” is being linked to a memory block named `BEGIN_M0`. The codestart section contains code that branches to the code entry point of the project. The bootloader must branch to the codestart section at the end of the boot process. Recall that the emulation boot mode "SARAM" branches to address 0x000000 upon bootloader completion.

   Notice that the linker command file `Lab_5_6_7.cmd` has a memory block named `BEGIN_M0: origin = 0x000000, length = 0x0002`, in program memory. The existing parts of memory blocks `BOOT_RSVD` and `RAMM0` in data memory has been modified to avoid any overlaps with this memory block.

7. In the linker command file, notice that RESET in the MEMORY section has been defined using the “(R)” qualifier. This qualifier indicates read-only memory, and is optional. It will cause the linker to flag a warning if any uninitialized sections are linked to this memory. The (R) qualifier can be used with all non-volatile memories (e.g., flash, ROM, OTP), as you will see in later lab exercises. Close the `Lab_5_6_7.cmd` linker command file.

## System Initialization

8. Open and inspect `main_5.c`. Notice the Device_init() function call to `device.c` for initializing the device.
9. Open `Watchdog.c` and notice the Driverlib function used for configuring the watchdog to generate a reset. Also, notice the Driverlib function used to disable the watchdog.
10. Open and inspect `Gpio.c`. Notice the Driverlib functions that are being used to configure the GPIO pins. Also, notice the input X-BAR configuration. This file will be used in the remaining lab exercises.

## Build and Load

11. Click the "`Build`" button and watch the tools run in the Console window. Check for errors in the Problems window.

12. Click the "`Debug`" button (green bug). A Launching Debug Session window will open. Select only CPU1 to load the program on (i.e. *uncheck* CPU2), and then click `OK`. Then the CCS Debug perspective view should open, the program will load automatically, and you should now be at the start of main().

13. After CCS loaded the program in the previous step, it set the program counter (PC) to point to _c_int00. It then ran through the C-environment initialization routine in the rts2800_fpu32.lib and stopped at the start of main(). CCS did not do a device reset, and as a result the bootloader was bypassed.

    In the remaining parts of this lab exercise, the device will be undergoing a reset due to the watchdog timer. Therefore, we must configure the device by loading values into EMU_KEY and EMU BMODE so the bootloader will jump to "RAMM0" at address 0x000000. Set the bootloader mode using the menu bar by clicking:

    `Scripts → EMU Boot Mode Select → EMU_BOOT_SARAM`

    If the device is power cycled between lab exercises, or within a lab exercise, be sure to re-configure the boot mode to `EMU_BOOT_SARAM`.

## Run the Code – Watchdog Reset Disabled

14. Place the cursor in the "main loop" section (on the `asm(" NOP");` instruction line) and right click the mouse key and select `Run To Line`. This is the same as setting a breakpoint on the selected line, running to that breakpoint, and then removing the breakpoint.

15. Place the cursor on the first line of code in main() and set a breakpoint by double clicking in the line number field to the left of the code line. Notice that line is highlighted with a blue dot indicating that the breakpoint has been set. (Alternatively, you can set a breakpoint on the line by right-clicking the mouse and selecting `Breakpoint (Code Composer Studio) → Breakpoint)`. The breakpoint is set to prove that the watchdog is disabled. If the watchdog causes a reset, code execution will stop at this breakpoint (or become trapped as explained in the watchdog hardware reset below).

16. Run your code for a few seconds by using the "`Resume`" button on the toolbar, or by using `Run → Resume` on the menu bar (or F8 key). After a few seconds halt your code by using the "`Suspend`" button on the toolbar, or by using `Run → Suspend` on the menu bar (or Alt-F8 key). Where did your code stop? Are the results as expected? If things went as expected, your code should be in the "main loop".

## Run the Code – CCS Issued CPU Reset

17. Perform a CCS CPU reset (soft reset) by clicking on the CPU Reset icon (or by selecting `Run → Reset → CPU Reset`). The program counter should now be at the entry point of the boot ROM code at 0x3FF16A. To view the boot ROM code click on `View Disassembly…`

18. Run your code. Where did your code stop? Are the results as expected? If things went as expected, your code should have stopped at the breakpoint. What happened is as follows. The ROM bootloader began execution and since the device is in emulation boot mode (i.e. the emulator is connected) the bootloader read the EMU_KEY and

EMU_BMODE values from the PIE RAM. These values were previously set for boot to RAMM0 boot mode by CCS. Since these values did not change and are not affected by reset, the bootloader transferred execution to the beginning of our code at address 0x000000 in the RAMM0, and execution continued until the breakpoint was hit in main( ).

## Run the Code – Watchdog Reset Enabled (Hardware Reset)

19. Open the Project Explorer window in the CCS Debug perspective view by selecting `View → Project Explorer`. In `Watchdog_5.c` modify the Driverlib function (using *comment out* and *uncomment*) to enable the watchdog. This will enable the watchdog to function and cause a reset. Save the file.

20. Build the project by clicking `Project → Build Project`. Select `Yes` to "Reload the program automatically".

    Alternatively, you add the "`Build`" button to the tool bar in the CCS Debug perspective (if it is not already there) so that it will available for future use. Click `Window → Perspective → Customize Perspective…` and then select the Tool Bar Visibility tab. Check the Code Composer Studio Project Build box. This will automatically select the "`Build`" button in the Tool Bar Visibility tab. Click `OK`.

21. Again, place the cursor in the "main loop" section (on the `asm(" NOP");` instruction line) and right click the mouse key and select `Run To Line`.

22. This time we will have the watchdog issue a reset that will toggle the XRSn pin (i.e. perform a hard reset). Now run your code. Where did your code stop? Why did your code stop at an assembly ESTOP0 instruction in the boot ROM at 0x3FE493 and not as we expected at the breakpoint in main( )? Here is what happened. While the code was running, the watchdog timed out and reset the processor. The reset vector was then fetched and the ROM bootloader began execution. Since the device is in emulation boot mode, it read the EMU_KEY and EMU_BMODE values from the PIE RAM which was previously set to RAMM0 boot mode. Again, note that these values do not change and are not affected by reset. When the F28x7x devices undergo a hardware reset (e.g. watchdog reset), the boot ROM code clears the RAM memory blocks. As a result, after the bootloader transferred execution to the beginning of our code at address 0x000000 in RAMM0, the memory block was cleared. The processor was then issued an illegal instruction which trapped us back in the boot ROM.

    This only happened because we are executing out of RAM. In a typical application, the Flash memory contains the program and the reset process would run as we would expect. This should explain why we did not see this behavior with the CCS CPU reset (soft reset where the RAM was not cleared). So what is the advantage of clearing memory during a hardware reset? This ensures that after the reset the original program code and data values will be in a known good state to provide a safer operation. It is important to understand that the watchdog did not behave differently depending on which type of reset was issued. It is the reset process that behaved differently from the different type of resets.

23. Since the watchdog reset in the previous step cleared the RAM blocks, we will now need to reload the program for the second part of this lab exercise. Reload the program by selecting:

    `Run → Load → Reload Program`

## Configure Watchdog Interrupt

The first part of this lab exercise used the watchdog to generate a CPU reset. This was tested using a breakpoint set at the beginning of main(). Next, we are going to use the watchdog to generate an interrupt. This part will demonstrate the interrupt concepts learned in the previous module.

24. In `Main_5.c` notice the two function calls to `interrupt.c` for initializing the PIE registers and PIE vectors:

    ```
    Interrupt_initModule();
    Interrupt_initVectorTable();
    ```

25. Also in the `main()`notice `EINT` and `ERTM` used to enable global interrupts and real-time debug at the appropriate location in the code.

26. In `Watchdog.c` modify the Driverlib function (using *comment out* and *uncomment*) to cause the watchdog to generate an interrupt rather than a reset. Save the file.

27. The watchdog interrupt "`INT_WAKE`" is located at "1.8" in the "PIE Interrupt Assignment Table". Notice the Driverlib function to re-map the watchdog interrupt signal to call the ISR function (see the #define name in `driverlib/inc/hw_ints.h` and label name in `DefaultIsr_5.c`) and the Driverlib function to enable the appropriate PIEIER and core IER.

28. Inspect `DefaultIsr_5.c`. This file contains interrupt service routines. The ISR for WAKE interrupt has been trapped by an emulation breakpoint contained in an inline assembly statement using "ESTOP0". This gives the same results as placing a breakpoint in the ISR. We will run the lab exercise as before, except this time the watchdog will generate an interrupt. If the Driverlib functions have been configured properly, the code will be trapped in the ISR.

## Build and Load

29. Build the project by clicking `Project` → `Build Project`, or by clicking on the "`Build`" button (if it has been added to the tool bar). Select `Yes` to "Reload the program automatically".

## Run the Code – Watchdog Interrupt

30. Place the cursor in the "main loop" section, right click the mouse key and select `Run To Line`.

31. Run your code. Where did your code stop? Are the results as expected? If things went as expected, your code should stop at the "ESTOP0" instruction in the `wakeISR()`.

## Terminate Debug Session and Close Project

32. Terminate the active debug session using the `Terminate` button . This will close the debugger and return Code Composer Studio to the CCS Edit perspective view.

33. Next, close the project by right-clicking on `Lab5` in the Project Explorer window and select `Close Project`.

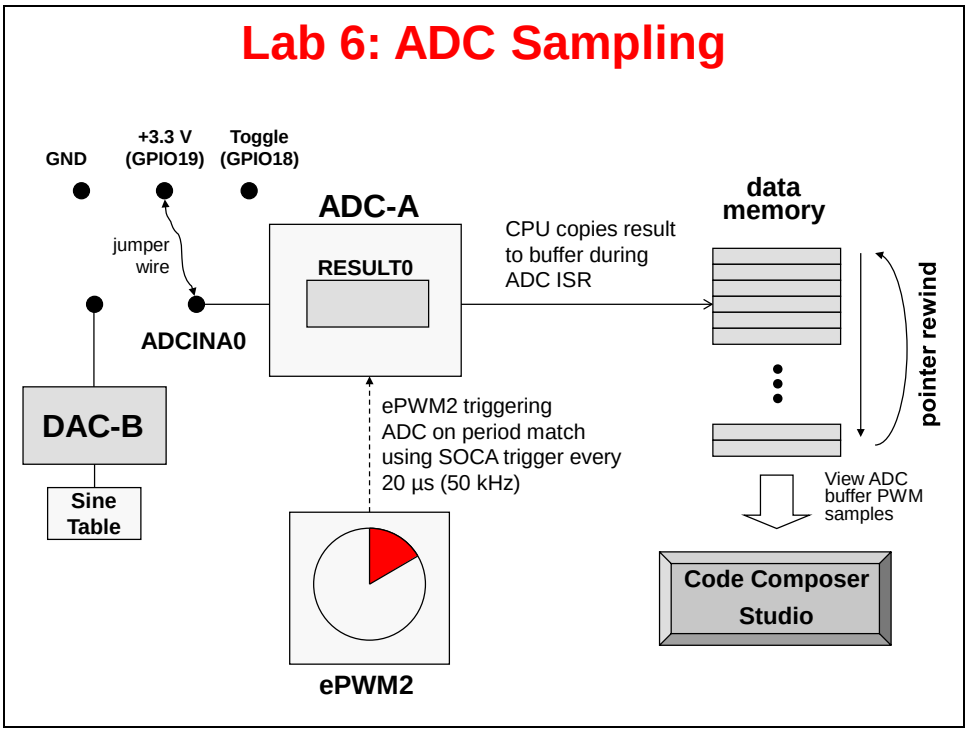
## End of Exercise

**Note:** By default, the watchdog timer is enabled out of reset. Code in the file `CodeStartBranch.asm` has been configured to disable the watchdog. This can be important for large C code projects. During this lab exercise, the watchdog was actually re-enabled (or disabled again) in the file `Watchdog.c`.

# Lab 6: Analog-to-Digital Converter

## ➢ Objective

The objective of this lab exercise is to become familiar with the programming and operation of the on-chip analog-to-digital converter (ADC). The microcontroller (MCU) will be setup to sample a single ADC input channel at a prescribed sampling rate and store the conversion result in a circular memory buffer. In the second part of this lab exercise, the digital-to-analog converter (DAC) will be explored.

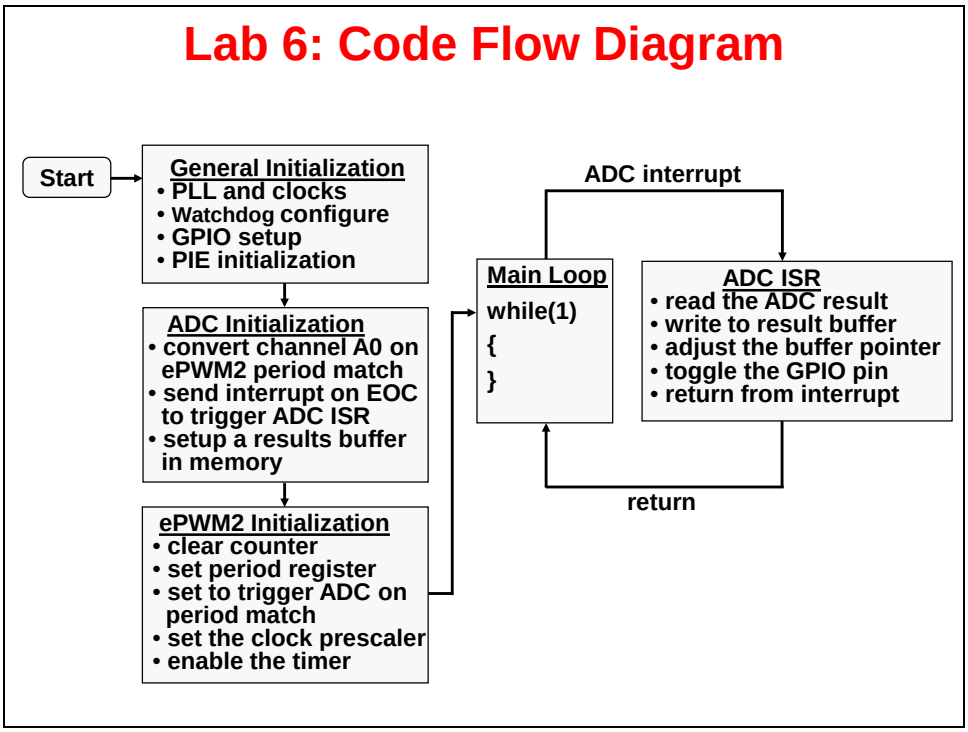



Recall that there are three basic ways to initiate an ADC start of conversion (SOC):

1. Using software
   a. SOCx (where x = 0 to 15) causes a software initiated conversion [ADC_TRIGGER_SW_ONLY]
2. Automatically triggered on user selectable conditions
   a. CPU Timer 0/1/2 interrupt [ADC_TRIGGER_CPU1_TINTx]
   b. ePWMxSOCA / ePWMxSOCB (x = 1 to 8) [ADC_TRIGGER_EPWMx_SOCA/B]
      - ePWM underflow (CTR = 0)
      - ePWM period match (CTR = PRD)
      - ePWM underflow or period match (CTR = 0 or PRD)
      - ePWM compare match (CTRU/D = CMPA/B/C/D)
   c. ADC interrupt ADCINT1 or ADCINT2
      - triggers SOCx selected by the ADC Interrupt Trigger SOC [ADC_INT_SOC_TRIGGER_NONE or ADC_INT_SOC_TRIGGER_ADCINTx]
3. Externally triggered using a pin
   a. SOCx trigger by ADCEXTSOC via INPUT5 X-BAR GPIO pin [ADC_TRIGGER_GPIO]

One or more of these methods may be applicable to a particular application. In this lab exercise, we will be using the ADC for data acquisition. Therefore, one of the ePWMs (ePWM2) will be

configured to automatically trigger the SOCA signal at the desired sampling rate (ePWM period match CTR = PRD SOC method 2b above). The ADC end-of-conversion interrupt will be used to prompt the CPU to copy the results of the ADC conversion into a results buffer in memory. This buffer pointer will be managed in a circular fashion, such that new conversion results will continuously overwrite older conversion results in the buffer. In order to generate an interesting input signal, the code also alternately toggles a GPIO pin (GPIO18) high and low in the ADC interrupt service routine. This pin will be connected to the ADC input pin, and sampled. The ADC ISR will also toggle LED D9 on the LaunchPad as a visual indication that the ISR is running. After taking some data, Code Composer Studio will be used to plot the results. A flow chart of the code is shown in the following slide.




## Notes

- Program performs conversion on ADC channel A0 (ADCINA0 pin)
- ADC conversion is set at a 50 kHz sampling rate
- ePWM2 is triggering the ADC on period match using SOCA trigger
- Data is continuously stored in a circular buffer
- GPIO18 pin is also toggled in the ADC ISR
- ADC ISR will also toggle the LaunchPad LED D9 as a visual indication that it is running

## ➢ Procedure

## Open the Project

1. A project named `Lab6` has been created for this lab exercise. Open the project by clicking on `Project` → `Import CCS Projects`. The "Import CCS Eclipse Projects" window will open. Click `Browse…` next to the "Select search-directory" box. Navigate to: `C:\F2837xD-DL\Labs\Lab6\cpu01` and click `Select Folder`. Then click `Finish` to import the project. All build options have been configured the same as the previous lab exercise. The files used in this lab exercise are:

```
Adc.c                          Gpio.c
CodeStartBranch.asm            Lab_5_6_7.cmd
Dac.c                          Main_6.c
DefaultIsr_6.c                 Sinetable.c
device.c                       Watchdog.c
EPwm_6.c
```

   *Note*: The `Dac.c` and `SineTable.c` files are used to generate a sine waveform in the second part of this lab exercise.

## ADC Initialization and Enable Core/PIE Interrupts

2. In `Main_6.c` notice the code to call the `InitAdca()`, `InitDacb()`, and `InitEPwm()` functions. The `InitEPwm()` function is used to configure ePWM2 to trigger the ADC at a 50 kHz rate. Details about the ePWM and control peripherals will be discussed in the next module. The `InitDacb()` function will be used in the second part of this lab exercise.
3. Open `Adc.c` and notice the Driverlib functions used to configure SOC0 in the ADC as follows:
   - SOC0 converts input ADCINA0 in single-sample mode
   - SOC0 has a 20 SYSCLK cycle acquisition window
   - SOC0 is triggered by the ePWM2 SOCA
   - SOC0 triggers ADCINT1 on end-of-conversion
   - All SOCs run round-robin
4. The ADC interrupt "`INT_ADCA1`" is located at "1.1" in the "PIE Interrupt Assignment Table". Notice the Driverlib function to re-map the ADC interrupt signal to call the ISR function (see the #define name in `driverlib/inc/hw_ints.h` and label name in `DefaultIsr_6.c`) and the Driverlib function to enable the appropriate PIEIER and core IER.
5. Inspect `DefaultIsr_6.c`. This file contains the ADC interrupt service routine.

## Build and Load

6. Click the "`Build`" button and watch the tools run in the Console window. Check for errors in the Problems window.
7. Click the "`Debug`" button (green bug). A Launching Debug Session window will open. Select only CPU1 to load the program on (i.e. *uncheck* CPU2), and then click `OK`. Then the CCS Debug perspective view should open, the program will load automatically, and you should now be at the start of `main()`. If the device has been power cycled since the

last lab exercise, be sure to configure the boot mode to EMU_BOOT_SARAM using the Scripts menu.

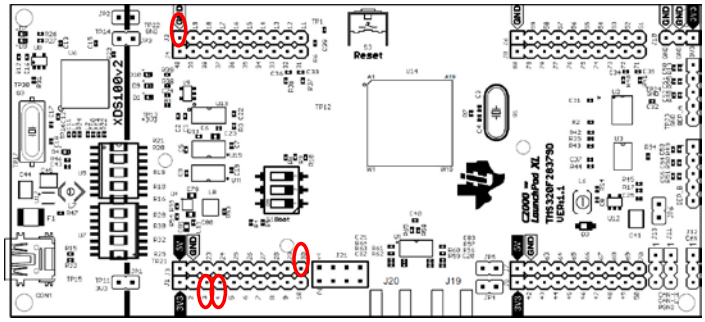
## Run the Code

8. In `Main_6.c` place the cursor in the "`main loop`" section, right click on the mouse key and select `Run To Line`.

   Open a memory browser to view some of the contents of the ADC results buffer. The address label for the ADC results buffer is *AdcBuf* (type **&AdcBuf**) in the "Data" memory page. Then <enter> to view the contents of the ADC result buffer.

**Note:** ***Exercise care when connecting any jumper wires to the LaunchPad header pins since the power to the USB connector is on!***

Refer to the following diagram for the location of the pins that will need to be connected:




9. Using a jumper wire, connect the ADCINA0 (header J3, pin #30) to "GND" (header J2, pin #20) on the LaunchPad. Then run the code again, and halt it after a few seconds. Verify that the ADC results buffer contains the expected value of ~0x0000.

10. Adjust the jumper wire to connect the ADCINA0 (header J3, pin #30) to "+3.3V" (header J1, pin #3; GPIO-19) on the LaunchPad. (Note: pin # GPIO-19 has been set to "1" in Gpio.c). Then run the code again, and halt it after a few seconds. Verify that the ADC results buffer contains the expected value of ~0x0FFF.

11. Adjust the jumper wire to connect the ADCINA0 (header J3, pin #30) to GPIO18 (header J1, pin #4) on the LaunchPad. Then run the code again, and halt it after a few seconds. Examine the contents of the ADC results buffer (the contents should be alternating ~0x0000 and ~0x0FFF values). Are the contents what you expected?

12. Open and setup a graph to plot a 50-point window of the ADC results buffer.
    Click: `Tools` → `Graph` → `Single Time` and set the following values:

| | |
|---|---|
| Acquisition Buffer Size | 50 |
| DSP Data Type | 16-bit unsigned integer |
| Sampling Rate (Hz) | 50000 |
| Start Address | AdcBuf |
| Display Data Size | 50 |
| Time Display Unit | μs |

Select `OK` to save the graph options.

13. Recall that the code toggled the GPIO18 pin alternately high and low. (Also, the ADC ISR is toggling the LED D9 on the LaunchPad as a visual indication that the ISR is running). If you had an oscilloscope available to display GPIO18, you would expect to see a square-wave. Why does Code Composer Studio plot resemble a triangle wave? What is the signal processing term for what is happening here?

14. Recall that the program toggled the GPIO18 pin at a 50 kHz rate. Therefore, a complete cycle (toggle high, then toggle low) occurs at half this rate, or 25 kHz. We therefore expect the period of the waveform to be 40 μs. Confirm this by measuring the period of the triangle wave using the "measurement marker mode" graph feature. In the graph window toolbar, left-click on the ruler icon with the red arrow . Note when you hover your mouse over the icon, it will show "`Toggle Measurement Marker Mode`". Move the mouse to the first measurement position and left-click. Again, left-click on the `Toggle Measurement Marker Mode` icon. Move the mouse to the second measurement position and left-click. The graph will automatically calculate the difference between the two values taken over a complete waveform period. When done, clear the measurement points by right-clicking on the graph and select `Remove All Measurement Marks` (or Ctrl+Alt+M).

## Using Real-time Emulation

Real-time emulation is a special emulation feature that offers two valuable capabilities:

A. Windows within Code Composer Studio can be updated at up to a 10 Hz rate *while the MCU is running*. This not only allows graphs and watch windows to update, but also allows the user to change values in watch or memory windows, and have those changes affect the MCU behavior. This is very useful when tuning control law parameters on-the-fly, for example.

B. It allows the user to halt the MCU and step through foreground tasks, while specified interrupts continue to get serviced in the background. This is useful when debugging portions of a real-time system (e.g., serial port receive code) while keeping critical parts of your system operating (e.g., commutation and current loops in motor control).

We will only be utilizing capability "A" above during the workshop. Capability "B" is a particularly advanced feature, and will not be covered in the workshop.

15. The memory and graph windows displaying *AdcBuf* should still be open. The jumper wire between ADCINA0 (header J3, pin #30) and GPIO18 (header J1, pin #4) should still be connected. In real-time mode, we will have our window continuously refresh at the default rate. To view the refresh rate click:

`Window` → `Preferences…`

and in the section on the left select the "Code Composer Studio" category. Click the sign ('+' or '>') to the left of "Code Composer Studio" and select "Debug". In the section on the right notice the default setting:

- "Continuous refresh interval (milliseconds)" = 500

Click `Cancel` to close the window.

Note: Decreasing the "Continuous refresh interval" causes all enabled continuous refresh windows to refresh at a faster rate. This can be problematic when a large number of windows are enabled, as bandwidth over the emulation link is limited. Updating too many windows can cause the refresh frequency to bog down. In this case you can just selectively enable continuous refresh for the individual windows of interest.

16. Next we need to enable the graph window for continuous refresh. Select the "Single Time" graph. In the graph window toolbar, left-click on the yellow icon with the arrows rotating in a circle over a pause sign . Note when you hover your mouse over the icon, it will show "Enable Continuous Refresh". This will allow the graph to continuously refresh in real-time while the program is running.

17. Enable the Memory Browser for continuous refresh using the same procedure as the previous step.

18. To enable and run real-time emulation mode, click the "Enable Silicon Real-time Mode" toolbar button . A window may open and if prompted select `Yes` to the "Do you want to enable realtime mode?" question. This will force the debug enable mask bit (DBGM) in status register ST1 to '0', which will allow the memory and register values to be passed to the host processor for updating (i.e. debug events are enabled). Hereafter, `Resume` and `Suspend` are used to run and halt real-time debugging. In the remaining lab exercises we will run and halt the code in real-time emulation mode.

19. Run the code (real-time mode).

20. ***Carefully*** remove and replace the jumper wire from GPIO18 (header J1, pin #4). Are the values updating in the Memory Browser and Single Time graph as expected?

21. Halt the code.

22. So far, we have seen data flowing from the MCU to the debugger in realtime. In this step, we will flow data from the debugger to the MCU.
    - Open and inspect `Main_6.c`. Notice that the global variable DEBUG_TOGGLE is used to control the toggling of the GPIO18 pin. This is the pin being read with the ADC.
    - Highlight DEBUG_TOGGLE with the mouse, right click and select "`Add Watch Expression…`" and then select OK. The global variable DEBUG_TOGGLE should now be in the Expressions window with a value of "1".
    - Enable the Expressions window for continuous refresh
    - Run the code in real-time mode and change the value to "0". Are the results shown in the memory and graph window as expected? Change the value back to "1". As you can see, we are modifying data memory contents while the processor is running in real-time (i.e., we are not halting the MCU nor interfering with its operation in any way)! When done, halt the CPU.

## Using the DAC to Generate a Sine Waveform

Next, we will configure DACB to generate a fixed frequency sine wave. This signal will appear on an analog output pin of the device (ADC-A1). Then using the jumper wire we will connect the DACB output to the ADCA input (ADC-A0) and display the sine wave in a graph window.

23. Notice the following code lines in the ADCA1 ISR in `DefaultIsr_6.c`:

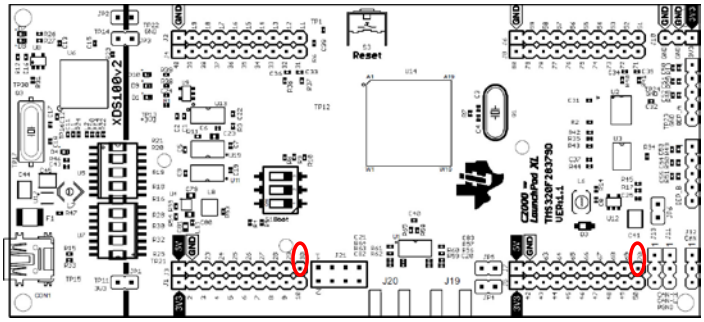
```
//--- Write to DAC-B to create input to ADC-A0
    if(SINE_ENABLE == 1)
    {
        DacOutput = DacOffset + ((QuadratureTable[iQuadratureTable++] ^ 0x8000) >> 5);
    }
    else
    {
        DacOutput = DacOffset;
    }
    if(iQuadratureTable > (SINE_PTS - 1))      // Wrap the index
    {
        iQuadratureTable = 0;
    }
    DAC_setShadowValue(DACB_BASE, DacOutput);
```

The variable `DacOffset` allows the user to adjust the DC output of DACB from the Expressions window in CCS. The variable Sine_Enable is a switch which adds a fixed frequency sine wave to the DAC offset. The sine wave is generated using a 25-point look-up table contained in the `SineTable.c` file. We will plot the sine wave in a graph window while manually adjusting the offset.

24. Open and inspect `SineTable.c`. (If needed, open the Project Explorer window in the CCS Debug perspective view by clicking `View` → `Project Explorer`). The file consists of an array of 25 signed integer points which represent four quadrants of sinusoidal data. The 25 points are a complete cycle. In the source code we need to sequentially access each of the 25 points in the array, converting each one from signed 16-bit to un-signed 12-bit format before writing it to the DACVALS register of DACB.

25. Add the following variables to the Expressions window:
    - SINE_ENABLE
    - DacOffset

26. Adjust the jumper wire to connect the ADCINA0 (header J3, pin #30) to DACB (header J7, pin #70) on the LaunchPad. Refer to the following diagram for the pins that need to be connected.

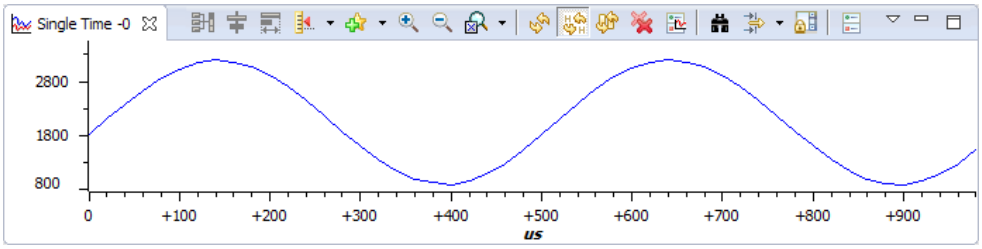


27. Run the code (real-time mode).

28. At this point, the graph should be displaying a DC signal near zero. Click on the dacOffset variable in the Expressions window and change the value to 800. This changes the DC output of the DAC which is applied to the ADC input. The level of the

graph display should be about 800 and this should be reflected in the value shown in the memory buffer (note: 800 decimal = 0x320 hex).

29. Enable the sine generator by changing the variable `SINE_ENABLE` in the Expressions window to 1.
30. You should now see sinusoidal data in the graph window.




31. Try removing and re-connecting the jumper wire to show this is real data is running in real-time emulation mode. Also, you can try changing the DC offset variable to move the input waveform to a different average value (the maximum distortion free offset is about 2000).
32. Halt the code.

## Terminate Debug Session and Close Project

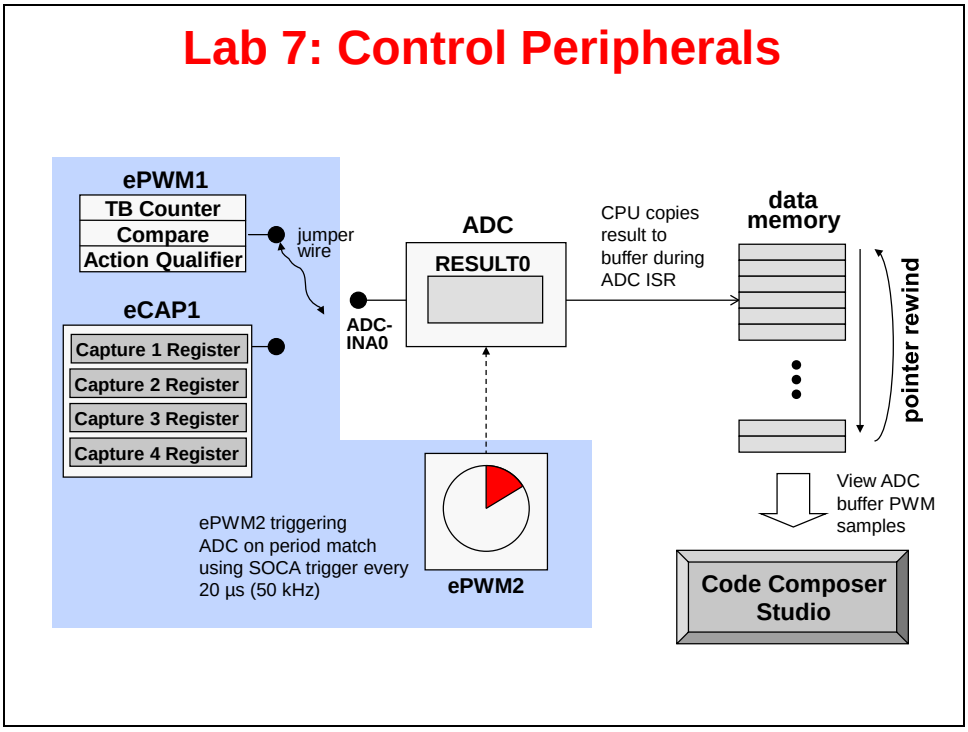
33. Terminate the active debug session using the `Terminate` button. This will close the debugger and return Code Composer Studio to the CCS Edit perspective view.
34. Next, close the project by right-clicking on `Lab6` in the Project Explorer window and select `Close Project`.

## End of Exercise

# Lab 7: Control Peripherals

## ➢ Objective

The objective of this lab exercise is to become familiar with the programming and operation of the control peripherals and their interrupts. ePWM1A will be setup to generate a 2 kHz, 25% duty cycle symmetrical PWM waveform. The waveform will then be sampled with the on-chip analog-to-digital converter and displayed using the graphing feature of Code Composer Studio. Next, eCAP1 will be setup to detect the rising and falling edges of the waveform. This information will be used to determine the width of the pulse and duty cycle of the waveform. The results of this step will be viewed numerically in a memory window.




## ➢ Procedure

### Open the Project

1. A project named `Lab7` has been created for this lab exercise. Open the project by clicking on `Project → Import CCS Projects`. The "Import CCS Eclipse Projects" window will open. Click `Browse…` next to the "Select search-directory" box. Navigate to: `C:\F2837xD-DL\Labs\Lab7\cpu01` and click `Select Folder`. Then click `Finish` to import the project. All build options have been configured the same as the previous lab exercise. The files used in this lab exercise are:

| | |
|---|---|
| `Adc.c` | `EPwm.c` |
| `CodeStartBranch.asm` | `Gpio.c` |
| `Dac.c` | `Lab_5_6_7.cmd` |
| `DefaultIsr_7.c` | `Main_7.c` |
| `device.c` | `SineTable.c` |
| `ECap.c` | `Watchdog.c` |

*Note*: The `ECap.c` file will be used with eCAP1 to detect the rising and falling edges of the waveform in the second part of this lab exercise.

## Generate PWM Waveform

2. Open and inspect `Gpio.c`. Notice the Driverlib functions used to configure ePWM1A as an output on the GPIO0 pin.
3. Open `EPwm.c` and notice the Driverlib functions used to configure ePWM1A as described in the objective for this lab exercise (i.e. generate a 2 kHz, 25% duty cycle symmetrical PWM waveform):
   - Set the timebase period and counter compare values by using the #define global variable names in the beginning of `Lab.h`, which is located in the Project Explorer window in the includes folder under `/Lab_common/include`
   - Set the action qualifier to generate the specified waveform
   - Enable the timebase count mode to generate a symmetrical PWM waveform

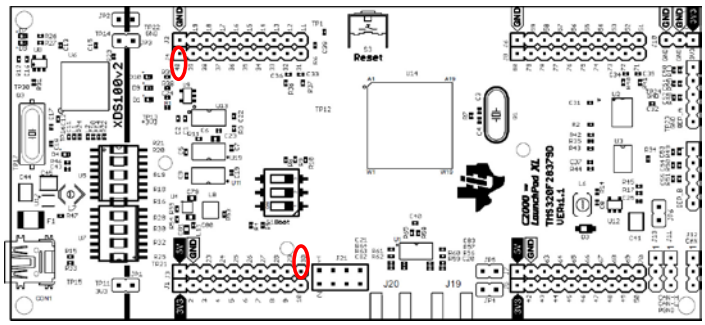
   Note that the deadband, PWM chopper, and all trip zone and DC compare actions have been disabled.

## Build and Load

4. Click the "`Build`" button and watch the tools run in the `Console` window. Check for errors in the Problems window.
5. Click the "`Debug`" button (green bug). A Launching Debug Session window will open. Select only CPU1 to load the program on (i.e. *uncheck* CPU2), and then click `OK`. Then the CCS Debug perspective view should open, the program will load automatically, and you should now be at the start of `main()`. If the device has been power cycled since the last lab exercise, be sure to configure the boot mode to EMU_BOOT_SARAM using the Scripts menu.

## Run the Code – PWM Waveform

6. Using a jumper wire, connect the PWM1A (header J4, pin #40) to ADCINA0 (header J3, pin #30) on the LaunchPad. Refer to the following diagram for the pins that need to be connected.



7. Open a memory browser to view some of the contents of the ADC results buffer. The address label for the ADC results buffer is *AdcBuf* (type **&AdcBuf**) in the "Data" memory page. We will be running our code in real-time mode, and we will need to have the memory window continuously refresh.

8. Run the code (real-time mode). Watch the window update. Verify that the ADC result buffer contains the updated values.

9. Open and setup a graph to plot a 50-point window of the ADC results buffer. Click: `Tools` → `Graph` → `Single Time` and set the following values:

| | |
|---|---|
| Acquisition Buffer Size | 50 |
| DSP Data Type | 16-bit unsigned integer |
| Sampling Rate (Hz) | 50000 |
| Start Address | AdcBuf |
| Display Data Size | 50 |
| Time Display Unit | μs |

   Select OK to save the graph options.

10. The graphical display should show the generated 2 kHz, 25% duty cycle symmetric PWM waveform. The period of a 2 kHz signal is 500 μs. You can confirm this by measuring the period of the waveform using the "measurement marker mode" graph feature. Disable continuous refresh for the graph before taking the measurements. In the graph window toolbar, left-click on the ruler icon with the red arrow. Note when you hover your mouse over the icon, it will show "`Toggle Measurement Marker Mode`". Move the mouse to the first measurement position and left-click. Again, left-click on the `Toggle Measurement Marker Mode` icon. Move the mouse to the second measurement position and left-click. The graph will automatically calculate the difference between the two values taken over a complete waveform period. When done, clear the measurement points by right-clicking on the graph and select `Remove All Measurement Marks`. Then enable continuous refresh for the graph.

## Frequency Domain Graphing Feature of Code Composer Studio

11. Code Composer Studio also has the ability to make frequency domain plots. It does this by using the PC to perform a Fast Fourier Transform (FFT) of the DSP data. Let's make a frequency domain plot of the contents in the ADC results buffer (i.e. the PWM waveform).

    Click: `Tools` → `Graph` → `FFT Magnitude` and set the following values:

| | |
|---|---|
| Acquisition Buffer Size | 50 |
| DSP Data Type | 16-bit unsigned integer |
| Sampling Rate (Hz) | 50000 |
| Start Address | AdcBuf |
| Data Plot Style | Bar |
| FFT Order | 10 |

Select `OK` to save the graph options.

12. On the plot window, hold the mouse left-click key and move the marker line to observe the frequencies of the different magnitude peaks. Do the peaks occur at the expected frequencies?
13. Halt the code.

## Use eCAP1 to Measure Width of Pulse

The first part of this lab exercise generated a 2 kHz, 25% duty cycle symmetric PWM waveform which was sampled with the on-chip analog-to-digital converter and displayed using the graphing feature of Code Composer Studio. Next, eCAP1 will be setup to detect the rising and falling edges of the waveform. This information will be used to determine the period and duty cycle of the waveform. The results of this step will be viewed numerically in a memory window and can be compared to the results obtained using the graphing features of Code Composer Studio.

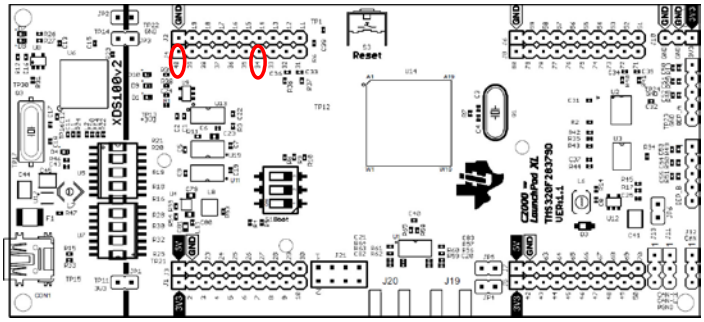
14. In `Main_7.c` notice the code used to call the `InitECap()`function. There are no passed parameters or return values, so the call code is simply:

```
InitECap();
```

15. In `Gpio.c` notice the Driverlib functions for configuring GPIO24 as the input. Next, notice the Driverlib function setting GPIO24 as the signal source for Input X-BAR INPUT7. The GPIO24 pin via INPUT7 will be routed as the input to eCAP1.
16. Open `DefaultIsr_7.c` and locate the eCAP1 interrupt service routine (ecap1ISR). Notice that PwmDuty is calculated by CAP2 – CAP1 (rising to falling edge) and that PwmPeriod is calculated by CAP3 – CAP1 (rising to rising edge).
17. Open `ECap.c` and notice the following configuration:
    - Set the event polarity to capturing the rising and falling edges of the PWM waveform in order to calculate the PWM duty and PWM period
    - Enable eCAP interrupt after three (3) capture events
18. The eCAP1 interrupt "`INT_ECAP1`" is located at "4.1" in the "PIE Interrupt Assignment Table". Notice the Driverlib function to re-map the eCAP1 interrupt signal to call the ISR function (see the #define name in `driverlib/inc/hw_ints.h` and label name in `DefaultIsr_7.c`) and the Driverlib function to enable the appropriate PIEIER and core IER.

## Run the Code – Pulse Width Measurement

19. Using a jumper wire, connect the PWM1A (header J4, pin #40) to ECAP1 (header J4, pin #34, feed from the Input X-bar using GPIO24) on the LaunchPad. Refer to the following diagram for the pins that need to be connected.



20. Open a memory browser to view the address label *PwmPeriod*. (Type ***&PwmPeriod*** in the address box). The address label *PwmDuty* (address ***&PwmDuty***) should appear in the same memory browser window. Scroll the window up, if needed.

21. Set the memory browser properties format to "32-Bit UnSigned Int". We will be running our code in real-time mode, and we will need to have the memory browser continuously refresh.

22. Run the code (real-time mode). Notice the values for *PwmDuty* and *PwmPeriod*.

23. Halt the code.

**<u>Questions:</u>**

- How do the captured values for *PwmDuty* and *PwmPeriod* relate to the compare register CMPA and time-base period TBPRD settings for ePWM1A?
- What is the value of *PwmDuty* in memory?
- What is the value of *PwmPeriod* in memory?
- How does it compare with the expected value?

## Optional Exercise – Modulate the PWM Waveform

Experiment with the code by observing the effects of changing the ePWM1 CMPA register using real-time emulation. Be sure that the jumper wire is connecting PWM1A (header J4, pin #40) to ADCINA0 (header J3, pin #30), and the Single Time graph is displayed. The graph must be enabled for continuous refresh.

a) Run the code (real-time mode).
b) Open the Registers window by clicking: `View → Registers`
c) In the Registers window scroll down and expand "EPwm1Regs". Then scroll down and expand "CMPA". In the Value field for "CMPA" right-click and set the Number Format to Decimal. The Registers window must be enabled for continuous refresh.
d) Change the "CMPA" 18750 value (within a range of 2500 and 22500). Notice the effect on the PWM waveform in the graph.

You have just modulated the PWM waveform by manually changing the CMPA value. Next, we will modulate the PWM automatically by having the ADC ISR change the CMPA value.

a) In `DefaultIsr_7.c` notice the code in the ADCA1 interrupt service routine used to modulate the PWM1A output between 10% and 90% duty cycle.
b) In `Main_7.c` add "PWM_MODULATE" to the Expressions window. Simply highlight PWM_MODULATE with the mouse, right click and select "`Add Watch Expression`…" and then select `OK`. The global variable PWM_MODULATE should now be in the Expressions window with a value of "0".
c) With the code still running in real-time mode, change the "PWM_MODULATE" from "0" to "1" and observe the PWM waveform in the graph. The value for *PwmDuty* will update continuously in the Memory Browser window. Also, in the Registers window notice the CMPA value being continuously updated.
d) Halt the code.

## Terminate Debug Session and Close Project

24. Terminate the active debug session using the `Terminate` button. This will close the debugger and return Code Composer Studio to the CCS Edit perspective view.

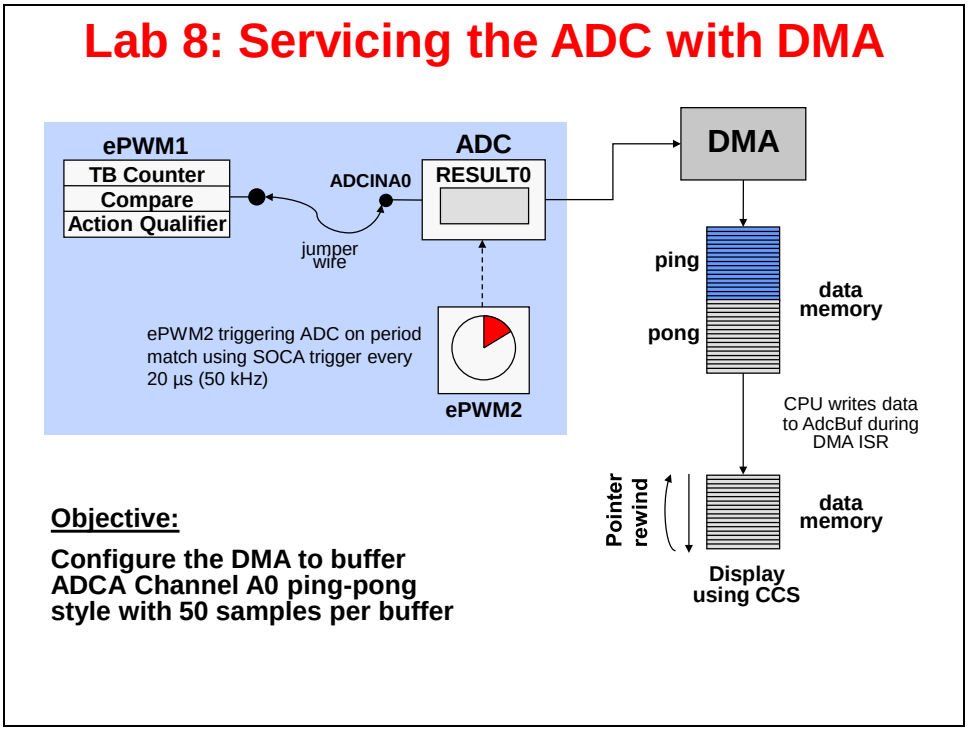
25. Next, close the project by right-clicking on `Lab7` in the Project Explorer window and select `Close Project`.

## End of Exercise

# Lab 8: Servicing the ADC with DMA

## ➢ Objective

The objective of this lab exercise is to become familiar with operation of the DMA. In the previous lab exercise, the CPU was used to store the ADC conversion result in the memory buffer during the ADC ISR. In this lab exercise the DMA will be configured to transfer the results directly from the ADC result registers to the memory buffer. ADC channel A0 will be buffered ping-pong style with 50 samples per buffer. As an operational test, the 2 kHz, 25% duty cycle symmetric PWM waveform (ePWM1A) will be displayed using the graphing feature of Code Composer Studio.




## ➢ Procedure

### Open the Project

1. A project named `Lab8` has been created for this lab exercise. Open the project by clicking on `Project → Import CCS Projects`. The "Import CCS Eclipse Projects" window will open. Click `Browse…` next to the "Select search-directory" box. Navigate to: `C:\F2837xD-DL\Labs\Lab8\cpu01` and click `Select Folder`. Then click `Finish` to import the project. All build options have been configured the same as the previous lab exercise. The files used in this lab exercise are:

```
Adc.c                          EPwm.c
CodeStartBranch.asm            Gpio.c
Dac.c                          Lab_8.cmd
DefaultIsr_8.c                 Main_8.c
device.c                       SineTable.c
Dma.c                          Watchdog.c
ECap.c
```

## Inspect Lab_8.cmd

2. Open and inspect `Lab_8.cmd`. Notice that a section called "`dmaMemBufs`" is being linked to `RAMGS4`. This section links the destination buffer for the DMA transfer to a DMA accessible memory space. Close the inspected file.

## DMA Initialization

The DMA controller needs to be configured to buffer ADC channel A0 ping-pong style with 50 samples per buffer. One conversion will be performed per trigger with the ADC operating in single sample mode.

3. Open `Dma.c` and notice the Driverlib functions used to implement the DMA operation as described in the objective for this lab exercise:
   - Enable the peripheral interrupt trigger for channel 1 DMA transfer
   - Generate an interrupt at the beginning of a new transfer
   - Enable the DMA channel CPU interrupt

   Note: the DMA has been configured for an ADC interrupt "ADCA1" to trigger the start of a DMA CH1 transfer. Additionally, the DMA is set for 16-bit data transfers with one burst per trigger and auto re-initialization at the end of the transfer. At the end of the code the channel is enabled to run.

4. Open `Main_8.c` and notice the line of code in `main()` to call the `InitDma()` function. There are no passed parameters or return values, so the call code is simply:

```
InitDma();
```

## PIE Interrupt for DMA

Recall that ePWM2 is triggering the ADC at a 50 kHz rate. In the previous lab exercise, the ADC generated an interrupt to the CPU, and the CPU read the ADC result register in the ADC ISR. For this lab exercise, the ADC is instead triggering the DMA, and the DMA will generate an interrupt to the CPU. The CPU will read the ADC result register in the DMA ISR.

5. Open `Adc.c` and notice the code used to enable the ADCA1 interrupt in PIE group 1 is *commented out*. This is no longer being used. The DMA interrupt will be used instead.

   The DMA Channel 1 interrupt "`INT_DMA_CH1`" is located at "7.1" in the "PIE Interrupt Assignment Table". Notice the Driverlib function to re-map the DMA Channel 1 interrupt signal to call the ISR function (see the #define name in `driverlib/inc/hw_ints.h` and label name in `DefaultIsr_8.c`) and the Driverlib function to enable the appropriate PIEIER and core IER.

6. Inspect `DefaultIsr_8.c` and notice that this file contains the DMA interrupt service routine which implements the ping-pong style buffer.

## Build and Load

7. Click the "`Build`" button and watch the tools run in the `Console` window. Check for errors in the `Problems` window.
8. Click the "`Debug`" button (green bug). A Launching Debug Session window will open. Select only CPU1 to load the program on (i.e. *uncheck* CPU2), and then click `OK`. Then the CCS Debug perspective view should open, the program will load automatically, and you should now be at the start of `main()`. If the device has been power cycled since the last lab exercise, be sure to configure the boot mode to EMU_BOOT_SARAM using the Scripts menu.

## Run the Code – Test the DMA Operation

**Note:** For the next step, check to be sure that the jumper wire connecting PWM1A (header J4, pin #40) to ADCINA0 (header J3, pin #30) is in place on the LaunchPad.

9. Run the code (real-time mode). Open and watch the memory browser update. Verify that the ADC result buffer contains updated values.
10. Open and setup a graph to plot a 50-point window of the ADC results buffer. Click: `Tools` → `Graph` → `Single Time` and set the following values:

| | |
|---|---|
| Acquisition Buffer Size | 50 |
| DSP Data Type | 16-bit unsigned integer |
| Sampling Rate (Hz) | 50000 |
| Start Address | AdcBuf |
| Display Data Size | 50 |
| Time Display Unit | µs |

Select OK to save the graph options.

11. The graphical display should show the generated 2 kHz, 25% duty cycle symmetric PWM waveform. Notice that the results match the previous lab exercise.
12. Halt the code.

## Terminate Debug Session and Close Project

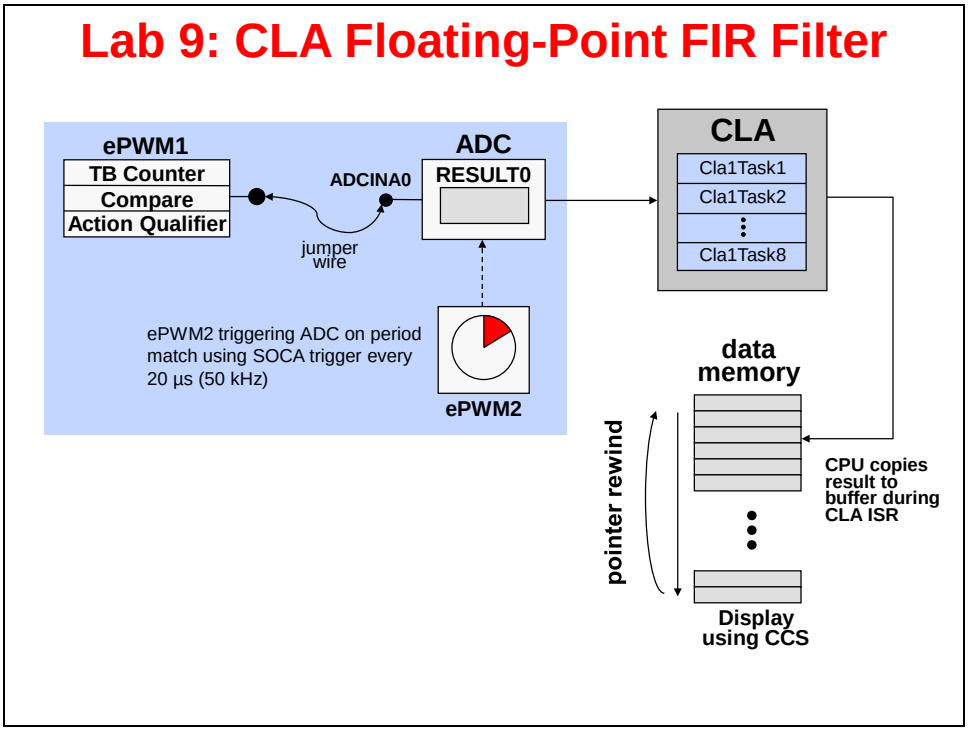
13. Terminate the active debug session using the `Terminate` button. This will close the debugger and return Code Composer Studio to the CCS Edit perspective view.
14. Next, close the project by right-clicking on `Lab8` in the Project Explorer window and select `Close Project`.

## End of Exercise

# Lab 9: CLA Floating-Point FIR Filter

## ➢ Objective

The objective of this lab exercise is to become familiar with operation and programming of the CLA. In this lab exercise, the ePWM1A generated 2 kHz, 25% duty cycle symmetric PWM waveform will be filtered using the CLA. The CLA will directly read the ADC result register and a task will run a low-pass FIR filter on the sampled waveform. The filtered result will be stored in a circular memory buffer. Note that the CLA is operating concurrently with the CPU. As an operational test, the filtered and unfiltered waveforms will be displayed using the graphing feature of Code Composer Studio.




Recall that a task is similar to an interrupt service routine. Once a task is triggered it runs to completion. In this lab exercise two tasks will be used. Task 1 contains the low-pass filter. Task 8 contains a one-time initialization routine that is used to clear (set to zero) the filter delay chain.

## ➢ Procedure

### Open the Project

1. A project named `Lab9` has been created for this lab exercise. Open the project by clicking on `Project` → `Import CCS Projects`. The "Import CCS Eclipse Projects" window will open. Click `Browse…` next to the "Select search-directory" box. Navigate to: `C:\F2837xD-DL\Labs\Lab9\cpu01` and click `Select Folder`. Then click `Finish` to import the project. All build options have been configured the same as the previous lab exercise. The files used in this lab exercise are:

```
Adc.c                          EPwm.c
Cla.c                          F2837xD_GlobalVariableDefs.c
ClaTasks_C.cla                 F2837xD_Headers_nonBIOS_cpu1.cmd
CodeStartBranch.asm            Gpio.c
Dac.c                          Lab_9.cmd
DefaultIsr_9_10.c              Main_9_10.c
device.c                       SineTable.c
Dma.c                          Watchdog.c
ECap.c
```

## Project Build Options and Enabling CLA Support in CCS

2. In this lab exercise the *Bit Field Header Files* are used for reading the ADC result register in the CLA task file (`ClaTasks_C.cla`). We need to setup the include search path to include the bit field header files. Open the build options by right-clicking on `Lab9` in the Project Explorer window and select "Properties". Under "C2000 Compiler" select "Include Options". In the include search path box that opens ("`Add dir to #include search path`") click the Add icon. Then in the "Add directory path" window type:

   **`${PROJECT_ROOT}/../../F2837xD_headers/include`**

   Click `OK` to include the search path.

   Note: from the bit field header files, `F2837xD_GlobalVariableDefs.c` and `F2837xD_Headers_nonBIOS_cpu1.cmd` have already been added to the project.

3. Next, we will confirm that CLA support has been enabled. Under "C2000 Compiler" select "`Processor Options`" and notice the "Specify CLA support" is set to `cla1`. This is needed to compile and assemble CLA code. Click `Apply and Close` to save and close the Properties window.

## Inspect Lab_9.cmd

4. Open and inspect `Lab_9.cmd`. Notice that a section called "`Cla1Prog`" is being linked to `RAMLS4`. This section links the CLA program tasks to the CPU memory space. Two other sections called "`Cla1Data1`" and "`Cla1Data2`" are being linked to `RAMLS1` and `RAMLS2`, respectively, for the CLA data. These memory spaces will be mapped to the CLA memory space during initialization. Also, notice the two message RAM sections used to pass data between the CPU and CLA.

   We are linking CLA code directly to the CLA program RAM because we are not yet using the flash memory. CCS will load the code for us into RAM, and therefore the CPU will not need to copy the CLA code into the CLA program RAM. In the flash programming lab exercise later in this workshop, we will modify the linking so that the CLA code is loaded into flash, and the CPU will do the copy.

5. The CLA C compiler uses a section called `.scratchpad` for storing local and compiler generated temporary variables. This scratchpad memory area is allocated using the linker command file. Notice `.scratchpad` is being linked to `RAMLS0`. Close the `Lab_9.cmd` linker command file.

## CLA Initialization

During the CLA initialization, the CPU memory block `RAMLS4` needs to be configured as CLA program memory. This memory space contains the CLA Task routines. A one-time force of the CLA Task 8 will be executed to clear the delay buffer. The CLA Task 1 has been configured to

run an FIR filter. The CLA needs to be configured to start Task 1 on the ADCAINT1 interrupt trigger. The next section will setup the PIE interrupt for the CLA.

6. Open `ClaTasks_C.cla` and notice Task 1 has been configured to run an FIR filter. Within this code the ADC result integer (i.e. the filter input) is being first converted to floating-point, and then at the end the floating-point filter output is being converted back to integer. Also, notice Task 8 is being used to initialize the filter delay line. The .cla extension is recognized by the compiler as a CLA C file, and the compiler will generate CLA specific code.

7. Open `Cla.c` and notice the Driverlib functions used to implement the CLA operation as described in the objective for this lab exercise:
   - Set Task 1 peripheral interrupt trigger source to ADCA1
   - Set Task 8 peripheral interrupt trigger source to SOFTWARE
   - Enable the use of the IACK instruction to trigger a task
   - Enable CLA Task 8 interrupt for one-time initialization routine (clear delay buffer)
   - Enable CLA Task 1 interrupt

   Note: the CLA has been configured for `RAMLS0`, `RAMLS1`, `RAMLS2`, and `RAMLS4` memory blocks to be shared between the CPU and CLA. The `RAMLS4` memory block is mapped to CLA program memory space, and the `RAMLS0`, `RAMLS1` and `RAMLS2` memory blocks are mapped to CLA data memory space. Also, the `RAMLS0` memory block is used for the CLA C compiler scratchpad. Notice that CLA Task 8 interrupt is disabled after the one-time initialization routine (clear delay buffer) is completed.

8. Open `Main_9_10.c` and notice the line of code in `main()` to call the `InitCla()` function. There are no passed parameters or return values, so the call code is simply:

```
InitCla();
```

9. In `Main_9_10.c` notice that the line of code in `main()` which calls the `InitDma()` function is *commented out*. The DMA is no longer being used. The CLA will directly access the ADC RESULT0 register.

## PIE Interrupt for CLA

Recall that ePWM2 is triggering the ADC at a 50 kHz rate. In the Control Peripherals lab exercise (i.e. ePWM lab), the ADC generated an interrupt to the CPU, and the CPU read the ADC result register in the ADC ISR. Then in the DMA lab exercise, the ADC instead triggered the DMA, and the DMA generated an interrupt to the CPU, where the CPU read the ADC result register in the DMA ISR. For this lab exercise, the ADC is instead triggering the CLA, and the CLA will directly read the ADC result register and run a task implementing an FIR filter. The CLA will generate an interrupt to the CPU, which will store the filtered results to a circular buffer implemented in the CLA ISR.

10. Remember that in `Adc.c` we *commented out* the code used to enable the ADCA1 interrupt in PIE group 1. This is no longer being used. The CLA interrupt will be used instead.

11. The CLA Task 1 interrupt "`INT_CLA1_1`" is located at "11.1" in the "PIE Interrupt Assignment Table". Notice the Driverlib function to re-map the CLA Task 1 interrupt signal to call the ISR function (see the #define name in `driverlib/inc/hw_ints.h` and label name in `DefaultIsr_9_10.c`) and the Driverlib function to enable the appropriate PIEIER and core IER.

12. Open and inspect `DefaultIsr_9_10.c`. Notice that this file contains the CLA interrupt service routine.

## Build and Load

13. Click the "`Build`" button and watch the tools run in the Console window. Check for errors in the Problems window.
14. Click the "`Debug`" button (green bug). A Launching Debug Session window will open. Select only CPU1 to load the program on (i.e. *uncheck* CPU2), and then click `OK`. Then the CCS Debug perspective view should open, the program will load automatically, and you should now be at the start of `main()`. If the device has been power cycled since the last lab exercise, be sure to configure the boot mode to EMU_BOOT_SARAM using the Scripts menu.

## Run the Code – Test the CLA Operation

**Note:** For the next step, check to be sure that the jumper wire connecting PWM1A (header J4, pin #40) to ADCINA0 (header J3, pin #30) is in place on the LaunchPad.

15. Run the code (real-time mode). Open and watch the memory browser window update. Verify that the ADC result buffer contains updated values.
16. Setup a dual-time graph of the filtered and unfiltered ADC results buffer. Click: `Tools` → `Graph` → `Dual Time` and set the following values:

| | |
|---|---|
| Acquisition Buffer Size | 50 |
| DSP Data Type | 16-bit unsigned integer |
| Sampling Rate (Hz) | 50000 |
| Start Address A | AdcBufFiltered |
| Start Address B | AdcBuf |
| Display Data Size | 50 |
| Time Display Unit | µs |

17. The graphical display should show the filtered PWM waveform in the Dual Time A display and the unfiltered waveform in the Dual Time B display. You should see that the results match the previous lab exercise.
18. Halt the code.

## Terminate Debug Session and Close Project

19. Terminate the active debug session using the `Terminate` button. This will close the debugger and return Code Composer Studio to the CCS Edit perspective view.
20. Next, close the project by right-clicking on `Lab9` in the Project Explorer window and select `Close Project`.

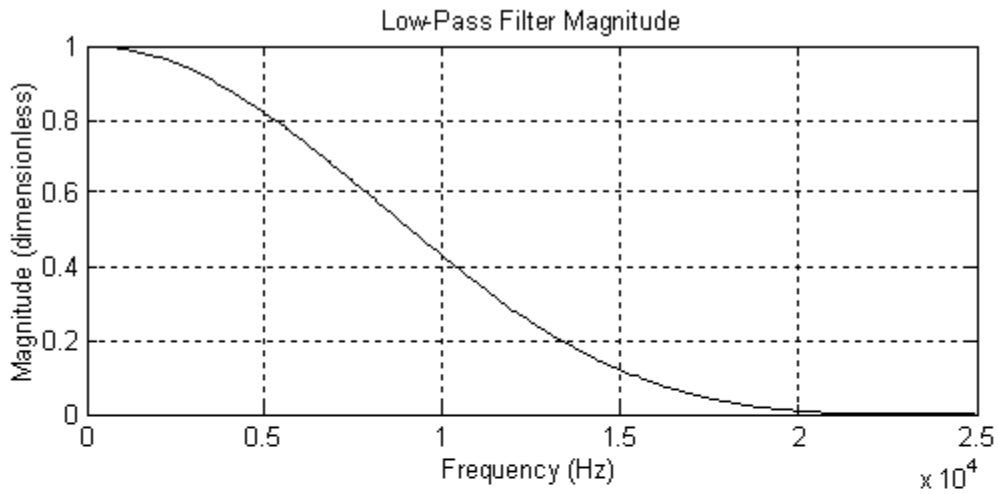
## End of Exercise

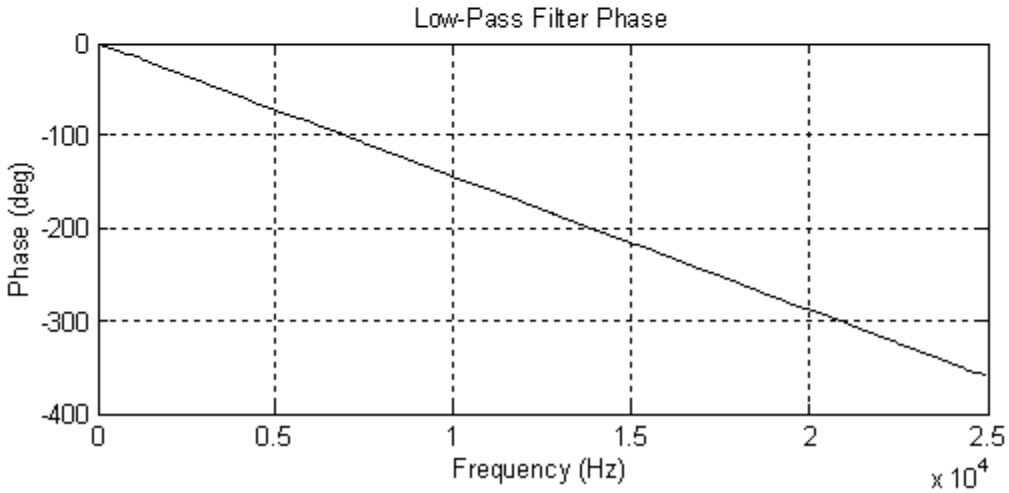
# Lab 9 Reference: Low-Pass FIR Filter

Bode Plot of Digital Low Pass Filter

Coefficients: [1/16, 4/16, 6/16, 4/16, 1/16]

Sample Rate: 50 kHz

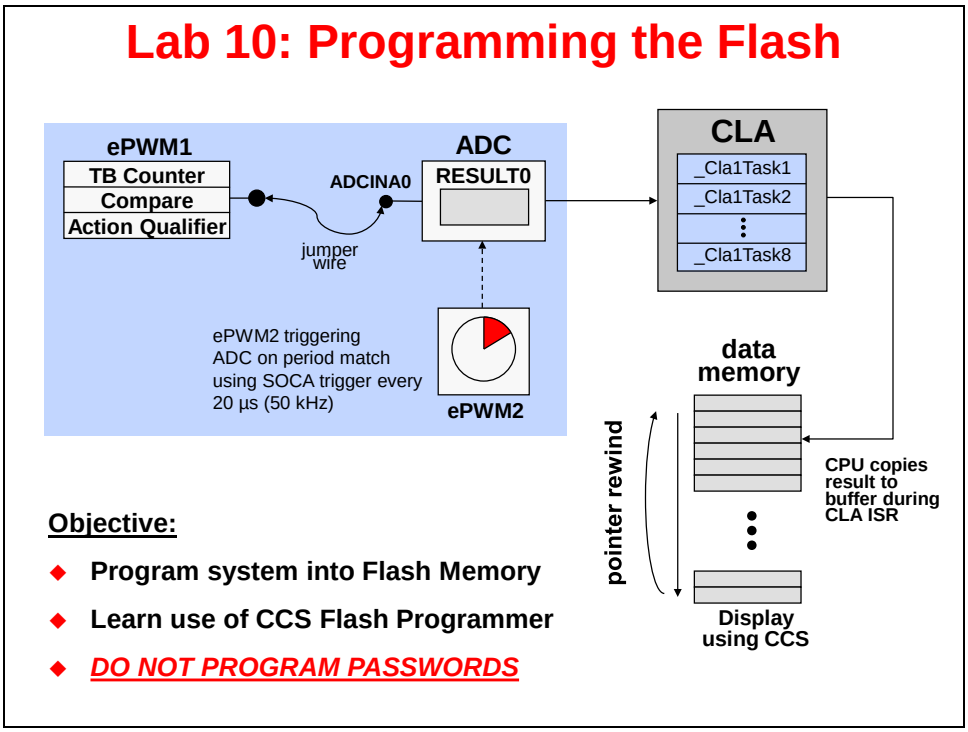


# Lab 10: Programming the Flash

## ➢ Objective

The objective of this lab exercise is to program and execute code from the on-chip flash memory. The TMS320F28379D device has been designed for standalone operation in an embedded system. Using the on-chip flash eliminates the need for external non-volatile memory or a host processor from which to bootload. In this lab exercise, the steps required to properly configure the software for execution from internal flash memory will be covered.




## ➢ Procedure

### Open the Project

1. A project named `Lab10` has been created for this lab exercise. Open the project by clicking on `Project` → `Import CCS Projects`. The "Import CCS Eclipse Projects" window will open. Click `Browse…` next to the "Select search-directory" box. Navigate to: `C:\F2837xD-DL\Labs\Lab10\cpu01` and click `Select Folder`. Then click `Finish` to import the project. All build options have been configured the same as the previous lab exercise. The files used in this lab exercise are:

```
Adc.c                          EPwm.c
Cla.c                          F2837xD_GlobalVariableDefs.c
ClaTasks_C.cla                 F2837xD_Headers_nonBIOS_cpu1.cmd
CodeStartBranch.asm            Gpio.c
Dac.c                          Lab_10.cmd
DefaultIsr_9_10.c              Main_9_10.c
device.c                       SineTable.c
Dma.c                          Watchdog.c
ECap.c
```

## Project Build Options

2. We need to setup the predefined symbols for programming the flash. Open the build options by right-clicking on `Lab10` in the Project Explorer window and select "Properties". Under "C2000 Compiler" select "Predefined Symbols". In the predefined name box that opens ("`Pre-define NAME`") click the `Add` icon. Then in the "Enter Value" window type **_FLASH**. This name is used in the project to conditionally include code specific to initializing the flash module. This conditional code is located in the `device.c` file. Click `OK` to include the name. Then click `Apply and Close` to save and close the Properties window.

## Link Initialized Sections to Flash

Initialized sections, such as code and constants, must contain valid values at device power-up. Stand-alone operation of an embedded system means that no debug probe (emulator) is available to initialize the device RAM. Therefore, all initialized sections must be linked to the on-chip flash memory.

Each initialized section actually has two addresses associated with it. First, it has a LOAD address which is the address to which it gets loaded at load time (or at flash programming time). Second, it has a RUN address which is the address from which the section is accessed at runtime. The linker assigns both addresses to the section. Most initialized sections can have the same LOAD and RUN address in the flash. However, some initialized sections need to be loaded to flash, but then run from RAM. This is required, for example, if the contents of the section needs to be modified at runtime by the code.

3. Open and inspect the linker command file `Lab_10.cmd`. Notice that the first flash sector has been divided into two blocks named BEGIN_FLASH and FLASH_A. The FLASH_A flash sector origin and length has been modified to avoid conflicts with the other flash sector spaces. The remaining flash sectors have been combined into a single block named FLASH_BCDEFGHIJKLMN. See the reference slide at the end of this lab exercise for further details showing the address origins and lengths of the various flash sectors used.

4. Notice in `Lab_10.cmd` that the following compiler sections are linked to on-chip flash memory block FLASH_BCDEFGHIJKLMN:

   **Compiler Sections:**

| | | | | | |
|---|---|---|---|---|---|
| .text | .cinit | .const | .econst | .pinit | .switch |

## Initializing the Interrupt Vectors from Flash to RAM

The interrupt vectors must be located in on-chip flash memory and at power-up needs to be loaded to the PIE RAM as part of the device initialization procedure. The code that performs this process is part of Driverlib. In main(), the call to Interrupt_initModule() enables vector fetching

from PIE block, and the call to Interrupt_initVectorTable() loads the vector table. In Driverlib, both functions are part of interrupt.c.

## Initializing the Flash Control Registers

The initialization code for the flash control registers cannot execute from the flash memory (since it is changing the flash configuration!). Therefore, the initialization function for the flash control registers must be copied from flash (load address) to RAM (run address) at runtime. The code that performs this process is part of Driverlib. In main(), the call to Device_init() copies time critical and flash setup code to RAM and then calls Flash_initModule() to initialize the flash. In Driverlib, these functions are part of device.c and flash.c.

5. In the Driverlib, flash.c uses the C compiler CODE_SECTION pragma to place the Flash_initModule() function into a linkable section named ".TI.ramfunc".

6. The ".TI.ramfunc" section will be linked using the user linker command file `Lab_10.cmd`. In `Lab_10.cmd` the ".TI.ramfunc" will load to flash (load address) but will run from RAMLS5 (run address). Also notice that the linker has been asked to generate symbols for the load start, load size, and run start addresses.

   While not a requirement from a MCU hardware or development tools perspective (since the C28x MCU has a unified memory architecture), historical convention is to link code to program memory space and data to data memory space. Therefore, notice that for the RAMLS5 memory we are linking ".TI.ramfunc" to, we are specifiying "PAGE = 0" (which is program memory).

## Dual Code Security Module and Passwords

The DCSM module provides protection against unwanted copying (i.e. pirating!) of your code from flash, OTP, LS0-5 RAM blocks, D0-1 RAM blocks, and CLA memory blocks. The DCSM uses a 128-bit password made up of 4 individual 32-bit words. They are located in the OTP. During this lab exercise, dummy passwords of 0xFFFFFFFF will be used – therefore only dummy reads of the password locations are needed to unsecure the DCSM. ***DO NOT PROGRAM ANY REAL PASSWORDS INTO THE DEVICE***. After development, real passwords are typically placed in the password locations to protect your code. We will not be using real passwords in the workshop. Again, ***DO NOT CHANGE THE VALUES FROM 0xFFFFFFFF***.

## Executing from Flash after Reset

The F28379D device contains a ROM bootloader that will transfer code execution to the flash after reset. When the boot mode selection is set for "Jump to Flash" mode, the bootloader will branch to the instruction located at address 0x080000 in the flash. An instruction that branches to the beginning of your program needs to be placed at this address. Note that BEGIN_FLASH begins at address 0x080000. There are exactly two words available to hold this branch instruction, and not coincidentally, a long branch instruction "LB" in assembly code occupies exactly two words. Generally, the branch instruction will branch to the start of the C-environment initialization routine located in the C-compiler runtime support library. The entry symbol for this routine is *_c_int00*. Recall that C code cannot be executed until this setup routine is run. Therefore, assembly code must be used for the branch. We are using the assembly code file named CodeStartBranch.asm.

7. Open and inspect `CodeStartBranch.asm`. This file creates an initialized section named "codestart" that contains a long branch to the C-environment setup routine. This section needs to be linked to a block of memory named BEGIN_FLASH.

8. In the earlier lab exercises, the section “codestart” was directed to the memory named BEGIN_M0. Now in `Lab_10.cmd` the section “codestart” is being directed to BEGIN_FLASH.

On power up the reset vector will be fetched and the ROM bootloader will begin execution. If the emulator is connected, the device will be in emulation boot mode and will use the EMU_KEY and EMU_BMODE values in the PIE RAM to determine the boot mode. This mode was utilized in the previous lab exercises. In this lab exercise, we will be disconnecting the emulator and running in stand-alone boot mode (but do not disconnect the emulator yet!). The bootloader will read the OTP_KEY and OTP_BMODE values from their locations in the OTP. The behavior when these values have not been programmed (i.e., both 0xFF) or have been set to invalid values is boot to flash boot mode.

## Initializing the CLA

Previously, the named section “Cla1Prog” containing the CLA program tasks was linked directly to the CPU memory block RAMLS4 for both load and run purposes. At runtime, all the code did was map the RAMLS4 block to the CLA program memory space during CLA initialization. For an embedded application, the CLA program tasks are linked to load to flash and run from RAM. At runtime, the CLA program tasks must be copied from flash to RAMLS4. The C-compiler runtime support library contains a memory copy function called *memcpy()* which will be used to perform the copy. After the copy is performed, the RAMLS4 block will then be mapped to CLA program memory space as was done in the earlier lab.

9. In `Lab_10.cmd` notice that the named section “Cla1Prog” will now load to flash (load address) but will run from RAMLS4 (run address). The linker will also be used to generate symbols for the load start, load size, and run start addresses.

10. Open `Cla.c` and notice that the memory copy function memcpy() is being used to copy the CLA program code from flash to RAMLS4 using the symbols generated by the linker. Just after the copy the Driverlib MemCfg_setCLAMemType() function is used to configure the RAMLS4 block as CLA program memory space. Close the opened files.

## Build – Lab.out

11. Click the “Build” button to generate the Lab.out file to be used with the CCS Flash Programmer. Check for errors in the Problems window.

## Programming the On-Chip Flash Memory

In CCS the on-chip flash programmer is integrated into the debugger. When the program is loaded CCS will automatically determine which sections reside in flash memory based on the linker command file. CCS will then program these sections into the on-chip flash memory. Additionally, in order to effectively debug with CCS, the symbolic debug information (e.g., symbol and label addresses, source file links, etc.) will automatically load so that CCS knows where everything is in your code. Clicking the “Debug” button in the CCS Edit perspective will automatically launch the debugger, connect to the target, and program the flash memory in a single step.

12. Program the flash memory by clicking the “Debug” button (green bug). A Launching Debug Session window will open. Select only CPU1 to load the program on (i.e. <u>*uncheck*</u> CPU2), and then click `OK`. The CCS Debug perspective view will open and the flash memory will be programmed. *(If needed, when the “Progress Information” box opens select* “Details >>” *in order to watch the programming operation and status).* After successfully programming the flash memory the “Progress Information” box will close. Then the program will load automatically, and you should now be at the start of main().

## Running the Code – Using CCS

13. Reset the CPU using the "CPU Reset" button or click:

    `Run → Reset → CPU Reset`

    The program counter should now be at address 0x3FF16A in the "Disassembly" window, which is the start of the bootloader in the Boot ROM. If needed, click on the "View Disassembly…" button in the window that opens, or click `View → Disassembly`.

14. Under Scripts on the menu bar click:

    `EMU Boot Mode Select → EMU_BOOT_FLASH`

    This has the debugger load values into EMU_KEY and EMU_BMODE so that the bootloader will jump to "Flash" at address 0x080000.

15. Next click:

    `Run → Go Main`

    The code should stop at the beginning of your main() routine. If you got to that point succesfully, it confirms that the flash has been programmed properly, that the bootloader is properly configured for jump to flash mode, and that the codestart section has been linked to the proper address.

16. You can now run the CPU, and you should observe the LED D9 on the LaunchPad blinking. Try resetting the CPU, select the EMU_BOOT_FLASH boot mode, and then hitting run (without doing the Go Main procedure). The LED should be blinking again.

17. Halt the CPU.

## Terminate Debug Session and Close Project

18. Terminate the active debug session using the `Terminate` button. This will close the debugger and return Code Composer Studio to the CCS Edit perspective view.

19. Next, close the project by right-clicking on `Lab10` in the Project Explorer window and select `Close Project`.

## Running the Code – Stand-alone Operation (No Emulator)

Recall that if the device is in stand-alone boot mode, the state of GPIO72 and GPIO84 pins are used to determine the boot mode. On the LaunchPad switch SW1 controls the boot options for the F28379D device. Check that switch SW1 positions 1 and 2 are set to the default "1 – on" position (both switches up). This will configure the device (in stand-alone boot mode) to GetMode. Since the OTP_KEY has not been programmed, the default GetMode will be boot from flash. Details of the switch positions can be found in the LaunchPad User's Guide.

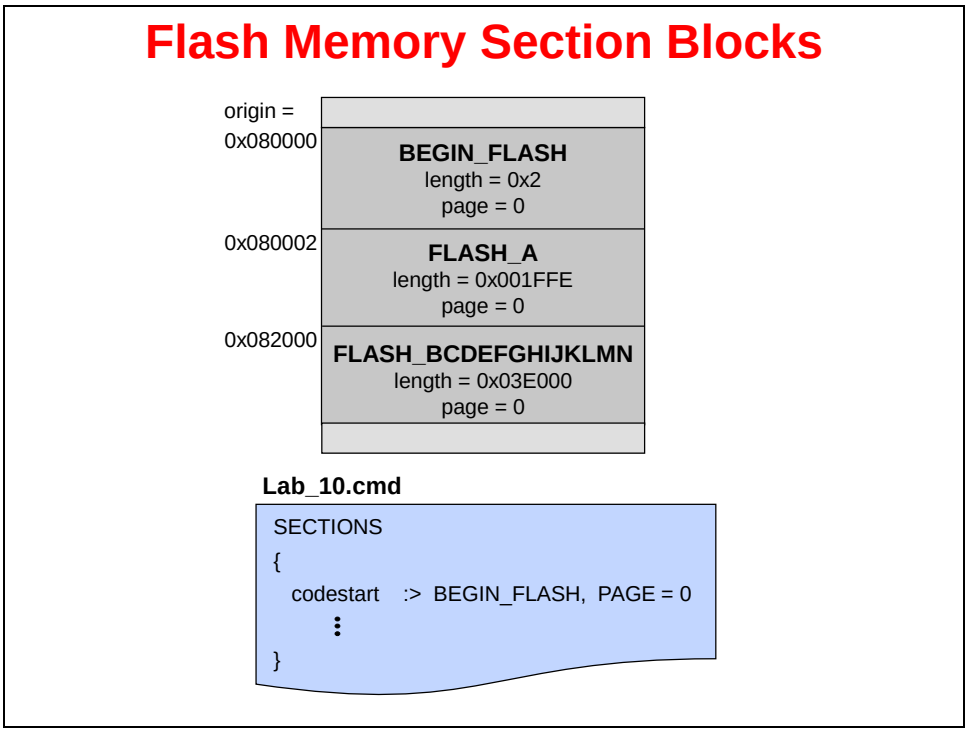
20. Close Code Composer Studio.

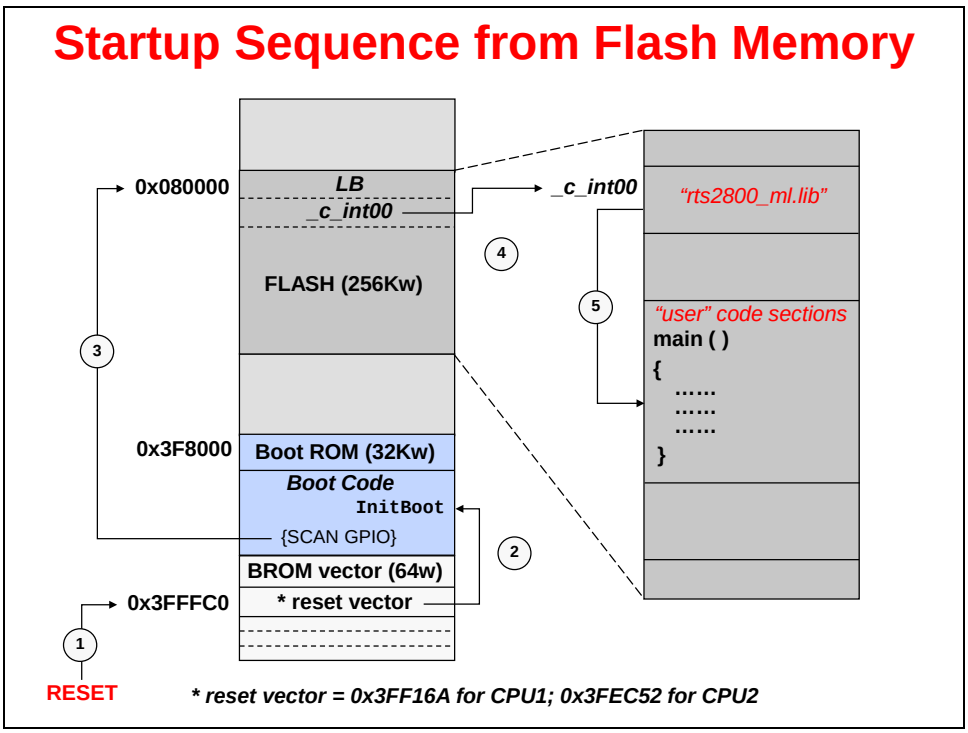
21. Disconnect the USB cable from the LaunchPad (i.e. remove power from the LaunchPad).

22. Re-connect the USB cable to the LaunchPad (i.e. power the LaunchPad). The LED should be blinking, showing that the code is now running from flash memory.

### End of Exercise

## Lab 10 Reference: Programming the Flash

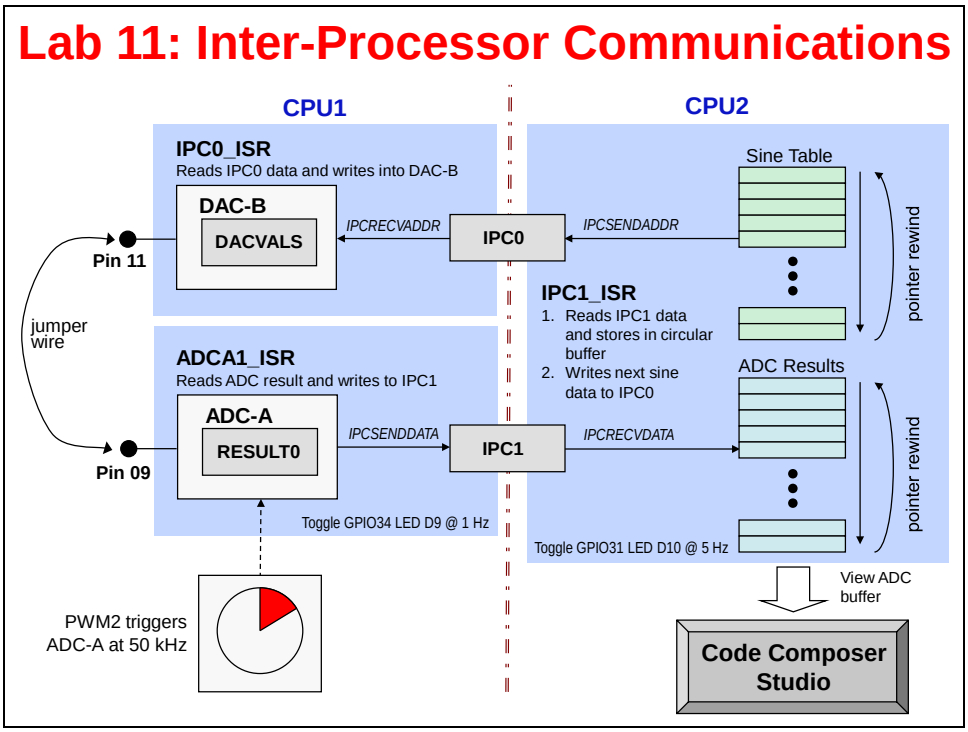


# Lab 11: Inter-Processor Communications

## ➢ Objective

The objective of this lab exercise is to demonstrate and become familiar with the operation of the IPC module. We will be using the basic IPC features to send data in both directions between CPU1 and CPU2. A typical dual-core F2837xD application consists of two separate and completely independent CCS projects. One project is for CPU1, and the other project is for CPU2. As in the previous lab exercises, PWM2 will be configured to provide a 50 kHz SOC signal to ADC-A. An End-of-Conversion ISR on CPU1 will read each result and write it into a data register in the IPC. An IPC interrupt will then be triggered on CPU2 which fetches this data and stores it in a circular buffer. The same ISR grabs a data point from a sine table and loads it into a different IPC register for transmission to CPU1. This triggers an interrupt on CPU1 to fetch the sine data and write it into DAC-B. The DAC-B output is connected by a jumper wire to the ADCINA0 pin. If the program runs as expected, the sine table and ADC results buffer on CPU2 should contain very similar data.




***Note:*** The F2837xD Driverlib does not include IPC driver APIs. A user can create their own custom API driver functions (using HWREG), however this requires detailed knowledge of the operation of each register and bit field along with the interactions and sequencing needed for proper peripheral operation. For simplicity, this lab exercise will be a hybrid approach using the bit-field header files for IPC and Driverlib for everything else. Each project in this lab exercise has been created to support both programming models.

## ➢ Procedure

## Open the Projects – CPU1 & CPU2

1. Two projects named `Lab11_cpu01` and `Lab11_cpu02` have been created for this lab exercise. Open ***both*** projects by clicking on `Project` → `Import CCS Projects`. The "Import CCS Eclipse Projects" window will open. Click `Browse…` next to the "Select search-directory" box. Navigate to: `C:\F2837xD-DL\Labs\Lab11` and click `OK`.

   Both projects will appear in the "Discovered projects" window. Click `Select All` and click `Finish` to import the projects. All build options for each project have been configured the same as the previous lab exercise.

   The files used in the CPU1 project are:

   ```
   Adc.c                                F2837xD_GlobalVariableDefs.c
   CodeStartBranch.asm                  F2837xD_Headers_nonBIOS_cpu1.cmd
   Dac.c                                Gpio.c
   DefaultIsr_11_cpu1.c                 Lab_11_cpu1.cmd
   device.c                             Main_11_cpu1.c
   EPwm_11.c                            Watchdog.c
   ```

   The files used in the CPU2 project are:

   ```
   CodeStartBranch.asm                  Lab_11_cpu2.cmd
   DefaultIsr_11_cpu2.c                 Main_11_cpu2.c
   F2837xD_GlobalVariableDefs.c         SineTable.c
   F2837xD_Headers_nonBIOS_cpu1.cmd     Watchdog.c
   ```

## Inspect the Project – CPU1

2. Click on the project name `Lab11_cpu01` in the Project Explorer window to set the project active. Then click on the plus sign (+) to the left of Lab11_cpu01 to expand the file list.

3. Open and inspect `Main_11_cpu1.c`. Notice the synchronization handshake code using IPC17 during initialization:

   ```
   //--- Wait here until CPU02 is ready
       while (IpcRegs.IPCSTS.bit.IPC17 == 0) ;  // Wait for CPU02 to set IPC17
       IpcRegs.IPCACK.bit.IPC17 = 1;            // Acknowledge and clear IPC17
   ```

   CPU1 will start first and then wait until CPU2 releases it from the while() loop. This only needs to be done once. In effect, CPU1 is waiting until CPU2 is ready to accept IPC interrupts, thereby making sure that the CPUs are ready for messaging through the IPC.

4. Open and inspect `DefaultIsr_11_cpu1.c`. This file contains two interrupt service routines – one (`adcA1ISR`) at PIE1.1 reads the ADC results which is sent over IPC1 to CPU2, and the other (`ipc0ISR`) at PIE1.13 reads the incoming sine table point for the DAC which is sent over IPC0 from CPU2. Additionally, adcA1ISR toggles the LaunchPad LED D9 at 1 Hz as a visual indication that it is running.

   In `adcA1ISR()` the ADC result value being sent to CPU2 is written via the `IPCSENDDATA` register. In `ipc0ISR()` the incoming data from CPU2 for the DAC is read via the `IPCRECVADDR` register. These registers are part of the IPC module and provide an easy way to transmit single data words between CPUs without using memory.

## Inspect the Project – CPU2

5. Click on the project name `Lab11_cpu02` in the Project Explorer window to set the project active. Then click on the plus sign (+) to the left of Lab11_cpu02 to expand the file list.

6. Open and inspect `Main_11_cpu2.c`. Notice the synchronization handshake code used to release CPU1 from its while() loop:

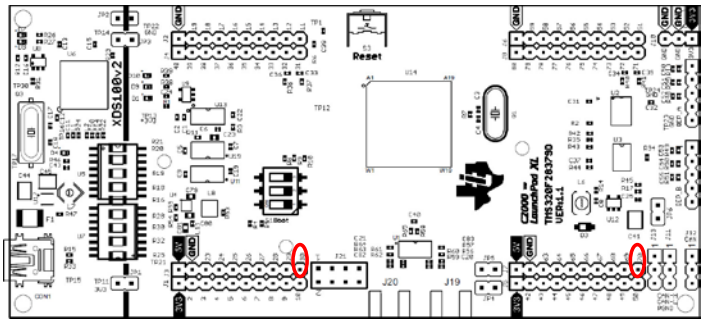
```
//--- Let CPU1 know that CPU2 is ready
    IpcRegs.IPCSET.bit.IPC17 = 1;            // Set IPC17 to release CPU1
```

7. Open and inspect `DefaultIsr_11_cpu2.c`. This file contains a single interrupt service routine – (`ipc1ISR`) at PIE1.14 reads the incoming ADC results which is sent over IPC1 from CPU1, and writes the next sine table point for the DAC which is sent over IPC0 to CPU1. Additionally, ipc1ISR toggles the LaunchPad LED D10 at 5 Hz as a visual indication that it is running.

   In `ipc1ISR()` the incoming ADC result value from CPU1 is read via the `IPCRECVDATA` register, and the sine data to CPU1 is written via the `IPCSENDADDR` register. The `IPCSENDDATA` and `IPCRECVDATA` registers are mapped to the same address on each CPU, as are the `IPCSENDADDR` and `IPCRECVADDR` registers.

## Jumper Wire Connection

8. Using a jumper wire, connect the ADCINA0 (header J3, pin #30) to DACB (header J7, pin #70) on the LaunchPad. Refer to the following diagram for the pins that need to be connected.

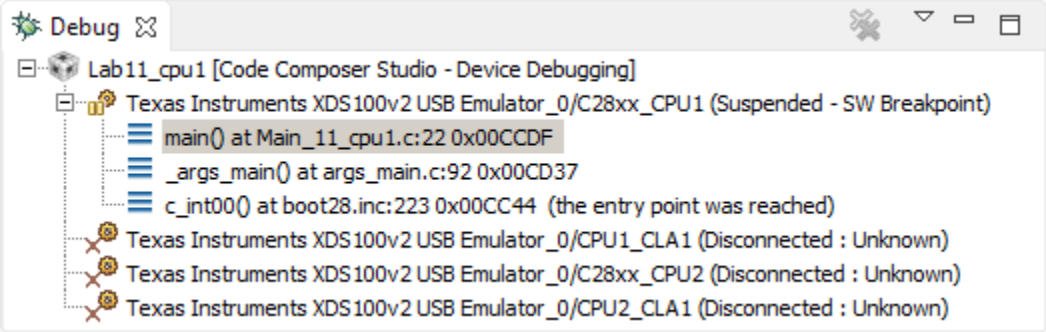


## Build and Load the Project

9. In the Project Explorer window click on the `Lab11_cpu01` project to set it active. Then click the Build button and watch the tools run in the Console window. Check for any errors in the Problems window. Repeat this step for the `Lab11_cpu02` project.

10. Again, in the Project Explorer window click on the `Lab11_cpu01` project to set it active. Click on the `Debug` button (green bug). A Launching Debug Session window will open. Select only CPU1 to load the program on (i.e. *uncheck* CPU2), and then click `OK`. The CCS Debug perspective view should open, then CPU1 will connect to the target and the program will load automatically.

11. The Debug window reflects the current status of CPU1 and CPU2.




Notice that CPU1 is currently connected and CPU2 is "Disconnected". This means that CCS has no control over CPU2 thus far; it is freely running from the view of CCS. Of course CPU2 is under control of CPU1 and since we have not executed an IPC command yet, CPU2 is stopped by an "Idle" mode instruction in the Boot ROM.

12. Next, we need to connect to and load the program on CPU2. Right-click at the line "Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU2" and select `Connect Target`.

13. With the line "Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU2" still highlighted, load the program:

    `Run` → `Load` → `Load Program…`

    Browse to the file: `C:\F2837xD-DL\Labs\Lab11\cpu02\Debug\Lab11_cpu02.out` and select `OK` to load the program.

14. Again, with the line "Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU1" highlighted, set the bootloader mode using the menu bar by clicking:

    `Scripts` → `EMU Boot Mode Select` → `EMU_BOOT_SARAM`

    Use the same procedure above to set the bootloader mode for CPU2. If the device has been power cycled between lab exercises, or within this lab exercise, be sure to configure the boot mode to `EMU_BOOT_SARAM` using the Scripts menu for both CPU1 and CPU2.

## Run the Code

15. In the Debug window, click on the line "Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU1". Run the code on CPU1 by clicking the green `Resume` button. At this point CPU1 is waiting for CPU2 to be ready.

16. In the Debug window, click on the line "Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU2". As before, run the code on CPU2 by clicking the `Resume` button. Using the IPC17, CPU2 communicates to CPU1 that it is now ready. On the LaunchPad, LED D9 connected to CPU1 should be blinking at approximately 1 Hz and LED D10 connected to CPU2 should be blinking at approximately 5 Hz.

17. In the Debug window select CPU1. Halt the CPU1 code after a few seconds by clicking on the `Suspend` button.

18. Then in the Debug window select CPU2. Halt the CPU2 code by using the same procedure.

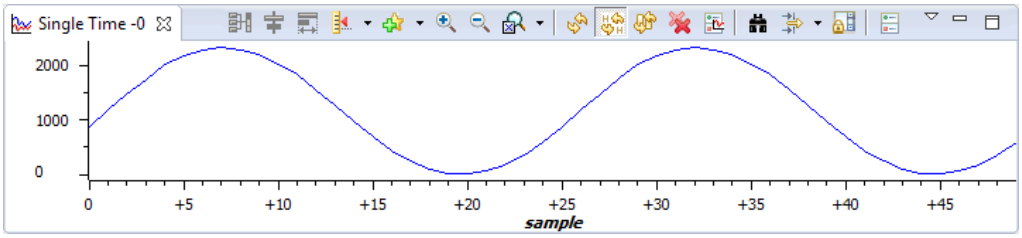
## View the ADC Results

19. Open and setup a graph to plot a 50-point window of the ADC results buffer.
    Click: `Tools` → `Graph` → `Single Time` and set the following values:

| | |
|---|---|
| Acquisition Buffer Size | 50 |
| DSP Data Type | 16-bit unsigned integer |
| Sampling Rate (Hz) | 50000 |
| Start Address | AdcBuf |
| Display Data Size | 50 |
| Time Display Unit | sample |

Select `OK` to save the graph options.

20. If the IPC communications is working, the ADC results buffer on CPU2 should contain the sine data transmitted from the look-up table. The graph view should look like:




## Run the Code - Real-Time Emulation Mode

21. We will now run the code in real-time emulation mode. Enable the graph window for continuous refresh. On the graph window toolbar, left-click on "`Enable Continuous Refresh`" (the yellow icon with the arrows rotating in a circle over a pause sign). This will allow the graph to continuously refresh in real-time while the program is running.

    In the Debug window highlight the line "Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU1". Run the code on CPU1 in real-time mode. Be sure to *"Enable Silicon Real-time Mode"* .

22. Next, in the Debug window highlight the line "Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU2". Run the code on CPU2 in real-time mode by using the same procedure above.

    The graph should now be updating in real-time. Be sure to enable the graph for *"Continuous Refresh"* .

23. *Carefully* remove and replace the jumper wire from the DACB output (header J7, pin #70) to the ADCINA0 input (header J3, pin #30). The ADC results graph should disappear and be replaced by a flat line when the jumper wire is removed. This shows that the sine data is being transmitted over IPC0 to CPU1, and (after being sent from DAC to ADC) received from CPU1 over IPC1.

24. Now we will view the IPC registers while the code is running in real-time emulation mode on CPU1 and CPU2. Open `Main_11_cpu1.c` (or `Main_11_cpu2.c`), highlight the "IpcRegs" structure and right click, then select `Add Watch Expression…` and click `OK`. Enable the Expressions window for continuous refresh.

25. In the Debug window highlight the line “Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU1”. Then in the Expressions window open “IpcRegs”, scroll down and notice the `IPCSENDDATA` and `IPCRECVADDR` registers is updating, as expected for CPU1. Also, notice that `IPCSENDADDR` and `IPCRECVDATA` registers, as well as the graph (ADC buffer) are not updated on CPU1.

26. In the Debug window highlight the line “Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU2”. Then in the Expressions window open “IpcRegs”, scroll down and notice the `IPCRECVDATA` and `IPCSENDADDR` registers, and the graph is updating, as expected for CPU2. Likewise, notice that `IPCRECVADDR` and `IPCSENDDATA` registers are not updated on CPU2.

27. Again, in the Debug window highlight the line “Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU1”. Halt the code on CPU1.

28. Next, halt the code on CPU2 in by using the same procedure.

## Terminate Debug Session and Close Project

29. Terminate the active debug session using the `Terminate` button. This will close the debugger and return Code Composer Studio to the CCS Edit perspective view.

30. Next, close the `Lab11_cpu01` and `Lab11_cpu02` projects by right-clicking on each project in the Project Explorer window and select `Close Project`.

## End of Exercise