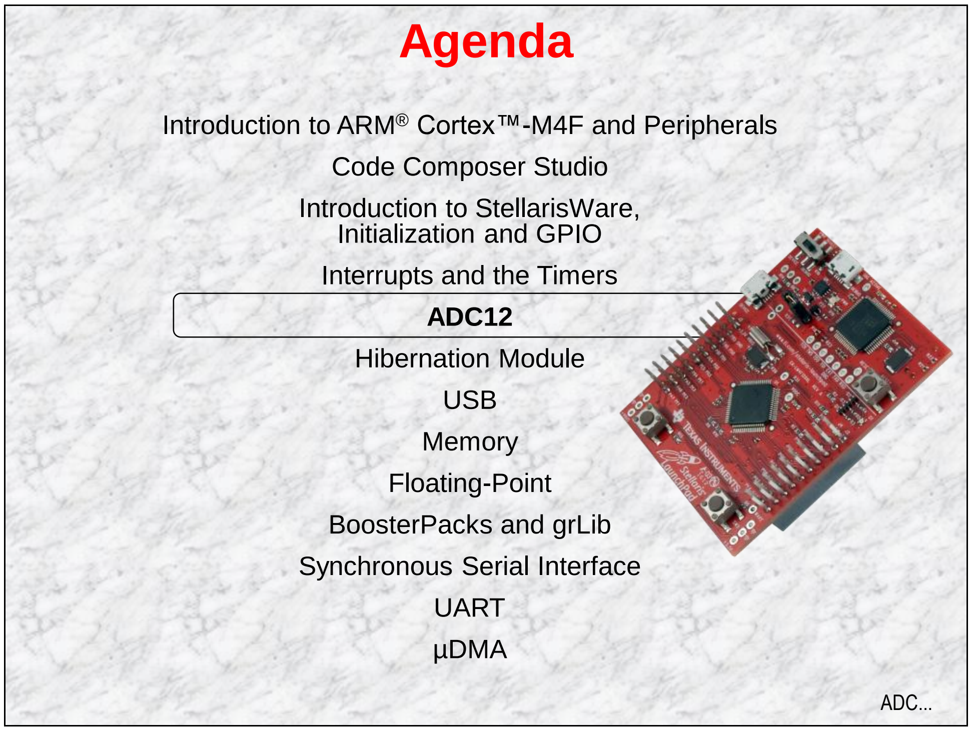
ADC12

Introduction

This chapter will introduce you to the use of the analog to digital conversion (ADC) peripheral on the Stellaris M4F. The lab will use the ADC and sequencer to sample the on-chip temperature sensor.



Agenda

Introduction to ARM® Cortex™-M4F and Peripherals

Code Composer Studio

Introduction to StellarisWare,
Initialization and GPIO

Interrupts and the Timers

ADC12

Hibernation Module

USB

Memory

Floating-Point

BoosterPacks and grLib

Synchronous Serial Interface

UART

µDMA

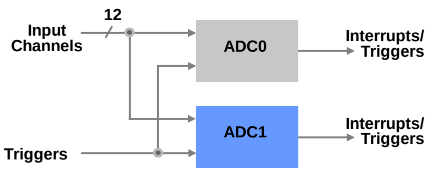
ADC...

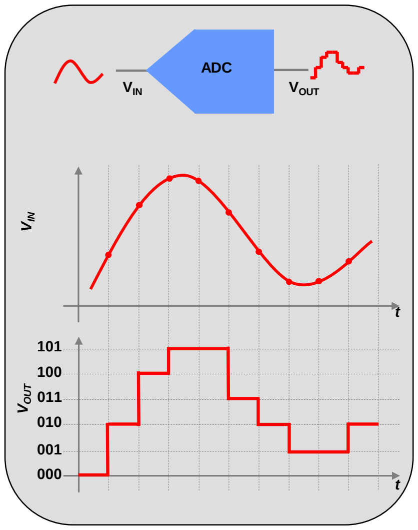
Chapter Topics

ADC12

Analog-to-Digital Converter

- Stellaris LM4F MCUs feature two ADC modules (ADC0 and ADC1) that can be used to convert continuous analog voltages to discrete digital values
- Each ADC module has 12-bit resolution
- Each ADC module operates independently and can:
  - Execute different sample sequences
  - Sample any of the shared analog input channels
  - Generate interrupts & triggers

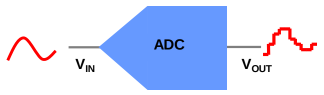




Features...

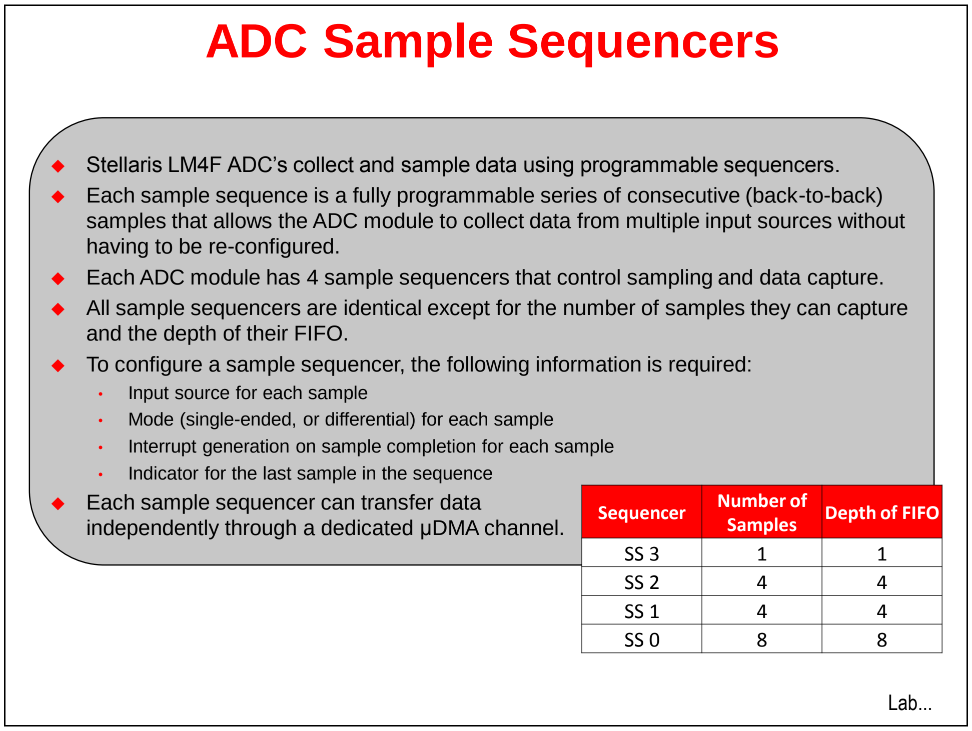
LM4F120H5QR ADC Features

- Two 12-bit 1MSPS ADCs
- 12 shared analog input channels
- Single ended & differential input configurations
- On-chip temperature sensor
- Maximum sample rate of one million samples/second (1MSPS).
- Fixed references (VDDA/GNDA) due to pin-count limitations
- 4 programmable sample conversion sequencers per ADC
- Separate analog power & ground pins
- Flexible trigger control
  - Controller/ software
  - Timers
  - Analog comparators
  - GPIO
- 2x to 64x hardware averaging
- 8 Digital comparators / per ADC
- 2 Analog comparators
- Optional phase shift in sample time, between ADC modules … programmable from 22.5 ° to 337.5°



Sequencers...

Sample Sequencers



ADC Sample Sequencers

- Stellaris LM4F ADC's collect and sample data using programmable sequencers.
- Each sample sequence is a fully programmable series of consecutive (back-to-back) samples that allows the ADC module to collect data from multiple input sources without having to be re-configured.
- Each ADC module has 4 sample sequencers that control sampling and data capture.
- All sample sequencers are identical except for the number of samples they can capture and the depth of their FIFO.
- To configure a sample sequencer, the following information is required:
  - Input source for each sample
  - Mode (single-ended, or differential) for each sample
  - Interrupt generation on sample completion for each sample
  - Indicator for the last sample in the sequence
- Each sample sequencer can transfer data independently through a dedicated μDMA channel.

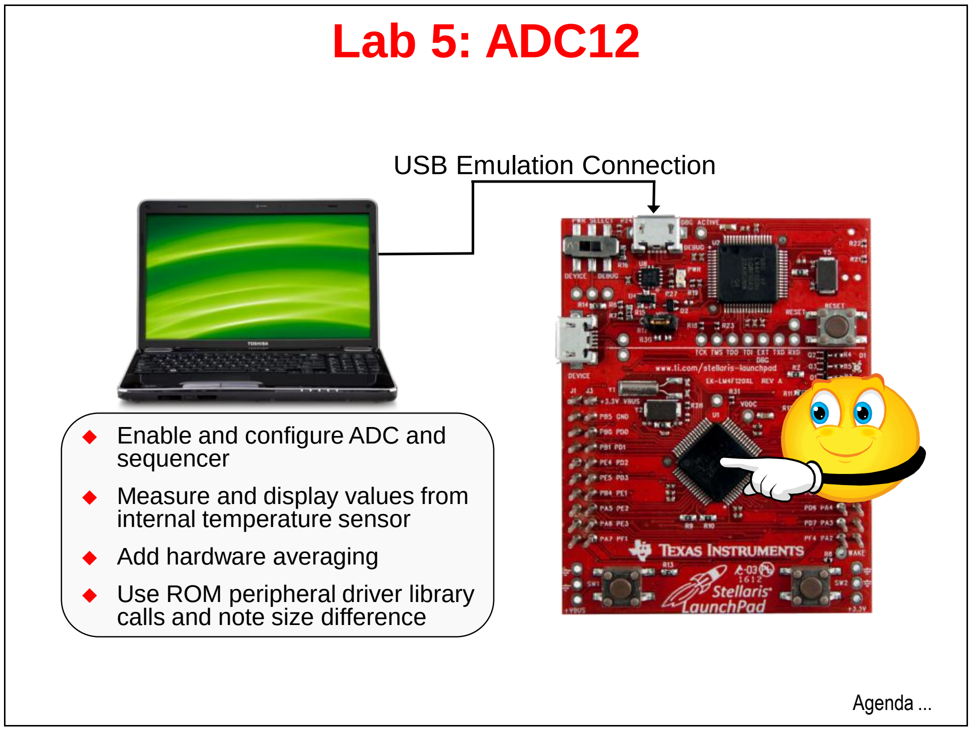
Sequencer	Number of Samples	Depth of FIFO
SS 3	1	1
SS 2	4	4
SS 1	4	4
SS 0	8	8

Lab...

Lab 5: ADC12

Objective

In this lab we'll use the ADC12 and sample sequencers to measure the data from the on-chip temperature sensor. We'll use Code Composer to display the changing values.



Lab 5: ADC12

USB Emulation Connection

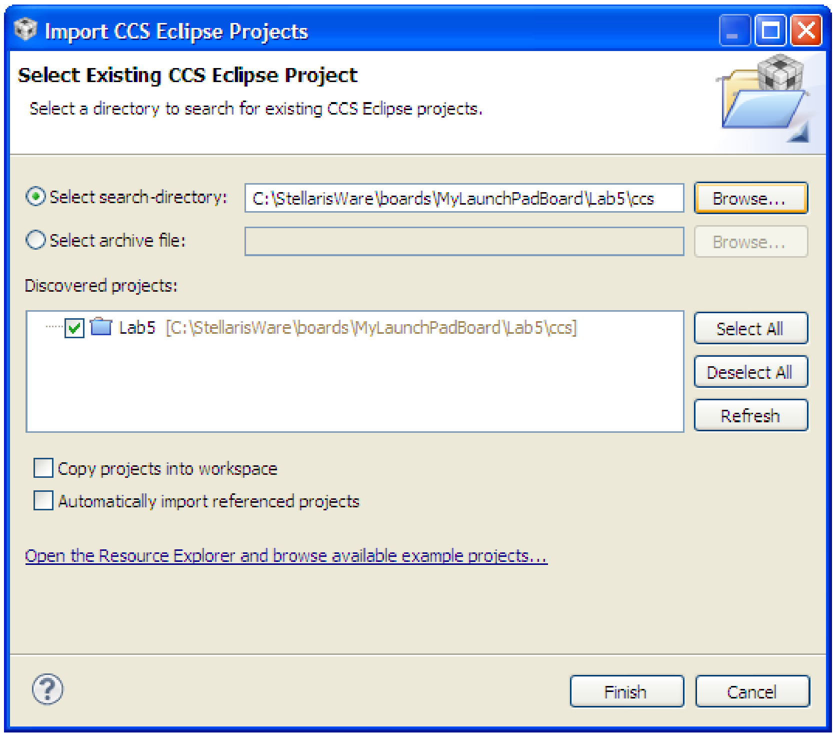
- Enable and configure ADC and sequencer
- Measure and display values from internal temperature sensor
- Add hardware averaging
- Use ROM peripheral driver library calls and note size difference

Agenda ...

Procedure

Import Lab5 Project

1. We have already created the Lab5 project for you with an empty `main.c`, a startup file and all necessary project and build options set. Maximize Code Composer and click Project → Import Existing CCS Eclipse Project. Make the settings shown below and click Finish. **Make sure that the "Copy projects into workspace" checkbox is unchecked**.



Select Existing CCS Eclipse Project

Select a directory to search for existing CCS Eclipse projects.

(●) Select search-directory: C:\StellarisWare\boards\MyLaunchPadBoard\Lab5\ccs

( ) Select archive file:

Discovered projects:

☑ Lab5 [C:\StellarisWare\boards\MyLaunchPadBoard\Lab5\ccs]

☐ Copy projects into workspace

☐ Automatically import referenced projects

Open the Resource Explorer and browse available example projects…

Header Files

2. Delete the current contents of `main.c`. Add the following lines into `main.c` to include the header files needed to access the StellarisWare APIs:

```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "driverlib/adc.h"
```

`adc.h:` definitions for using the ADC driver

Driver Library Error Routine

3. Run-time parameter checking by the Peripheral Driver Library is fairly cursory since excessive checking would have a negative effect on cycle count. But, during the debug process, you may find that you have called a driver library API with incorrect parameters or a library function generates an error for some other reason. The following code will be called if the driver library encounters such an error. In order for the code to run, DEBUG needs to be added to the pre-defined symbols for the project … we'll do that later.

   Leave a blank line for spacing and add these lines of code after the lines above:

   ```
   #ifdef DEBUG
   void__error__(char *pcFilename, unsigned long ulLine)
   {
   }
   #endif
   ```

Main()

4. Set up the main() routine by adding the three lines below:

   ```
   int main(void)
   {
   }
   ```

5. The following definition will create an array that will be used for storing the data read from the ADC FIFO. It must be as large as the FIFO for the sequencer in use. We will be using sequencer 1 which has a FIFO depth of 4. If another sequencer was used with a smaller or deeper FIFO, then the array size would have to be changed. For instance, sequencer 0 has a depth of 8.

   Add the following line of code as your first line of code inside **`main()`** :

   ```
   unsigned long ulADC0Value[4];
   ```

6. We'll need some variables for calculating the temperature from the sensor data. The first variable is for storing the average of the temperature. The remaining variables are used to store the temperature values for Celsius and Fahrenheit. All are declared as 'volatile' so that each variable will not be optimized out by the compiler and will be available to the 'Expression' or 'Local' window(s) at run-time. Add these lines after that last line:

   ```
   volatile unsigned long ulTempAvg;
   volatile unsigned long ulTempValueC;
   volatile unsigned long ulTempValueF;
   ```

7. Set up the system clock again to run at 40MHz. Add a line for spacing and add this line after the last ones:

   ```
   SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);
   ```

8. Let's enable the ADC0 module next. Add a line for spacing and add this line after the last one:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
```

9. As an example, let's set the ADC sample rate to 250 kilo-samples per second (since we're measuring temperature, a slower speed would be fine, but let's go with this). The `SysCtlADCSpeedSet()` API can also set the sample rate to additional device specific speeds (125KSPS, 500KSPS and 1MSPS)..

   Add the following line directly after the last one:

```
SysCtlADCSpeedSet(SYSCTL_ADCSPEED_250KSPS);
```

10. Before we configure the ADC sequencer settings, we should disable ADC sequencer 1. Add this line after the last one:

```
ADCSequenceDisable(ADC0_BASE, 1);
```

11. Now we can configure the ADC sequencer. We want to use ADC0, sample sequencer 1, we want the processor to trigger the sequence and we want to use the highest priority. Add a line for spacing and add this line of code:

```
ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);
```

12. Next we need to configure all four steps in the ADC sequencer. Configure steps 0 - 2 on sequencer 1 to sample the temperature sensor (ADC_CTL_TS). In this example, our code will average all four samples of temperature sensor data on sequencer 1 to calculate the temperature, so all four sequencer steps will measure the temperature sensor. For more information on the ADC sequencers and steps, reference the device specific datasheet. Add the following three lines after the last:

```
ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_TS);
ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_TS);
ADCSequenceStepConfigure(ADC0_BASE, 1, 2, ADC_CTL_TS);
```

13. The final sequencer step requires a couple of extra settings. Sample the temperature sensor (ADC_CTL_TS) and configure the interrupt flag (ADC_CTL_IE) to be set when the sample is done. Tell the ADC logic that this is the last conversion on sequencer 1 (ADC_CTL_END). Add this line directly after the last ones:

```
ADCSequenceStepConfigure(ADC0_BASE, 1, 3, ADC_CTL_TS | ADC_CTL_IE | ADC_CTL_END);
```

14. Now we can enable the ADC sequencer 1. Add this line directly after the last one:

```
ADCSequenceEnable(ADC0_BASE, 1);
```

15. Still within `main(),` add a while loop to your code. Add a line for spacing and enter these three lines of code:

    ```
    while(1)
    {

    }
    ```

16. Save your work. As a sanity-check, right-click on `main.c` in the Project pane and select Build Selected File(s). If you are having issues, check the code below:

```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "driverlib/adc.h"

#ifdef DEBUG
void__error__(char *pcFilename, unsigned long ulLine)
{
}
#endif

int main(void)
{
	unsigned long ulADC0Value[4];
	volatile unsigned long ulTempAvg;
	volatile unsigned long ulTempValueC;
	volatile unsigned long ulTempValueF;

	SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);

	SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
	SysCtlADCSpeedSet(SYSCTL_ADCSPEED_250KSPS);
	ADCSequenceDisable(ADC0_BASE, 1);

	ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);
	ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_TS);
	ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_TS);
	ADCSequenceStepConfigure(ADC0_BASE, 1, 2, ADC_CTL_TS);
	ADCSequenceStepConfigure(ADC0_BASE, 1, 3, ADC_CTL_TS | ADC_CTL_IE | ADC_CTL_END);
	ADCSequenceEnable(ADC0_BASE, 1);

	while(1)
	{

	}

}
```

When you build this code, you may get a warning "`ulADC0Value was declared but never referenced`". Ignore this warning for now, we'll add code to use this array later.

Inside the while(1) Loop

Inside the `while(1)` we're going to read the value of the temperature sensor and calculate the temperature endlessly.

17. The indication that the ADC conversion is complete will be the ADC interrupt status flag. It's always good programming practice to make sure that the flag is cleared before writing code that depends on it. Add the following line as your first line of code inside the `while(1)` loop:

    ```
    ADCIntClear(ADC0_BASE, 1);
    ```

18. Then we can trigger the ADC conversion with software. ADC conversions can also be triggered by many other sources. Add the following line after the last:

    ```
    ADCProcessorTrigger(ADC0_BASE, 1);
    ```

19. Then we need to wait for the conversion to complete. Obviously, a better way to do this would be to use an interrupt, rather than burn CPU cycles waiting, but that exercise is left for the student. Add a line for spacing and add the following three lines of code:

    ```
    while(!ADCIntStatus(ADC0_BASE, 1, false))
    {
    }
    ```

20. When code execution exits the loop in the previous step, we know that conversion is complete and we can read the ADC value from the ADC Sample Sequencer 1 FIFO. The function we'll be using copies data from the specified sample sequencer output FIFO to a buffer in memory. The number of samples available in the hardware FIFO are copied into the buffer, which must be large enough to hold that many samples. This will only return the samples that are presently available, which might not be the entire sample sequence if you attempt to access the FIFO before the conversion is complete. Add a line for spacing and add the following line after the last:

    ```
    ADCSequenceDataGet(ADC0_BASE, 1, ulADC0Value);
    ```

21. Calculate the average of the temperature sensor data. We're going to cover floating point operations later, so this math will be fixed-point.

    The addition of 2 is for rounding. Since 2/4 = 1/2 = 0.5, 1.5 will be rounded to 2.0 with the addition of 0.5. In the case of 1.0, when 0.5 is added to yield 1.5, this will be rounded back down to 1.0 due to the rules of integer math.

    Add this line after the last on a single line:

    ```
    ulTempAvg = (ulADC0Value[0] + ulADC0Value[1] + ulADC0Value[2] +
    ulADC0Value[3] + 2)/4;
    ```

22. Now that we have the averaged reading from the temperature sensor, we can calculate the Celsius value of the temperature. The equation below is shown in section 13.3.6 of the LM4F120H5QR datasheet. Division is performed last to avoid truncation due to integer math rules. A later lab will cover floating point operations.

    TEMP = 147.5 – ((75 * (VREFP – VREFN) * ADCVALUE) / 4096)

    We need to multiply everything by 10 to stay within the precision needed. The divide by 10 at the end is needed to get the right answer. VREFP – VREFN is Vdd or 3.3 volts. We'll multiply it by 10, and then 75 to get 2475.

    Enter the following line of code directly after the last:

    ```
    ulTempValueC = (1475 - ((2475 * ulTempAvg)) / 4096)/10;
    ```

23. Once you have the Celsius temperature, calculating the Fahrenheit temperature is easy. Hold the division until the end to avoid truncation.

    The conversion from Celsius to Fahrenheit is F = (C * 9)/5 +32. Adjusting that a little gives: F = ((C * 9) + 160) / 5

    Enter the following line of code directly after the last:

    ```
    ulTempValueF = ((ulTempValueC * 9) + 160) / 5;
    ```

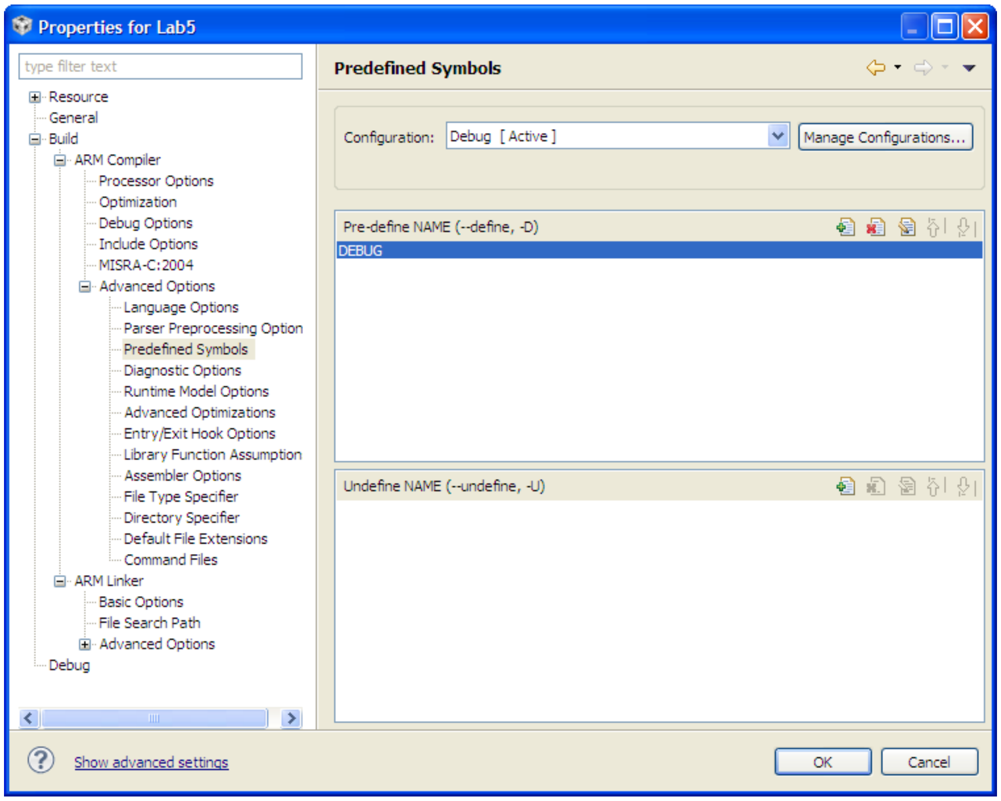
24. Save your work and compare it with our code below:

```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "driverlib/adc.h"

#ifdef DEBUG
void__error__(char *pcFilename, unsigned long ulLine)
{
}
#endif

int main(void)
{
	unsigned long ulADC0Value[4];
	volatile unsigned long ulTempAvg;
	volatile unsigned long ulTempValueC;
	volatile unsigned long ulTempValueF;

	SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);

	SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
	SysCtlADCSpeedSet(SYSCTL_ADCSPEED_250KSPS);
	ADCSequenceDisable(ADC0_BASE, 1);

	ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);
	ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_TS);
	ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_TS);
	ADCSequenceStepConfigure(ADC0_BASE, 1, 2, ADC_CTL_TS);
	ADCSequenceStepConfigure(ADC0_BASE, 1, 3, ADC_CTL_TS | ADC_CTL_IE | ADC_CTL_END);
	ADCSequenceEnable(ADC0_BASE, 1);

	while(1)
	{
		ADCIntClear(ADC0_BASE, 1);
		ADCProcessorTrigger(ADC0_BASE, 1);

		while(!ADCIntStatus(ADC0_BASE, 1, false))
		{
		}

		ADCSequenceDataGet(ADC0_BASE, 1, ulADC0Value);
		ulTempAvg = (ulADC0Value[0] + ulADC0Value[1] + ulADC0Value[2] + ulADC0Value[3] + 2)/4;
		ulTempValueC = (1475 - ((2475 * ulTempAvg)) / 4096)/10;
		ulTempValueF = ((ulTempValueC * 9) + 160) / 5;
	}
}
```

You can also find this code in main1.txt in your project folder.

Add Pre-defined Symbol

25. Right-click on Lab5 in the Project Explorer pane and select Properties. Under Build → ARM Compiler, click the + next to Advanced Options. Then click on Predefined Symbols. In the top Pre-define NAME window, add the symbol DEBUG as shown below and click OK. In future labs, the project will already have this symbol defined.



Configuration: Debug [Active]

Pre-define NAME (--define, -D)

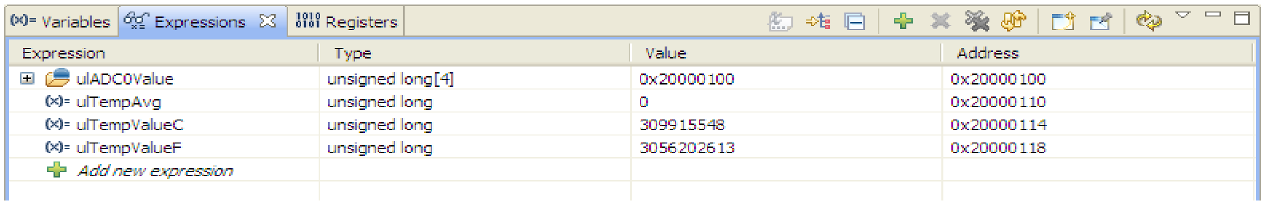
DEBUG

Build and Run the Code

26. Compile and download your application by clicking the Debug button on the menu bar. If you have any issues, correct them, and then click the Debug button again. After a successful build, the CCS Debug perspective will appear.

27. Click on the Expressions tab (upper right). Remove all expressions (if there are any) from the Expressions pane by right-clicking inside the pane and selecting Remove All.

    Find the **`ulADC0Value, ulTempAvg, ulTempValueC and ulTempValueF`** variables in the last four lines of code. Double-click on a variable to highlight it, then right-click on it, select Add Watch Expression and then click OK. Do this for all four variables.



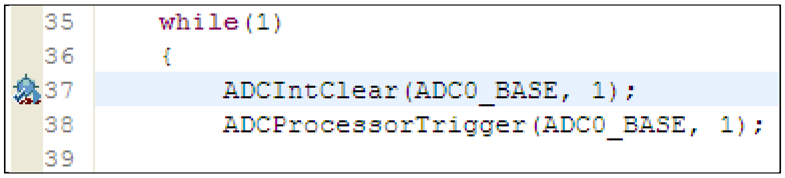
Expression	Type	Value	Address
ulADC0Value	unsigned long[4]	0x20000100	0x20000100
ulTempAvg	unsigned long	0	0x20000110
ulTempValueC	unsigned long	309915548	0x20000114
ulTempValueF	unsigned long	3056202613	0x20000118
Add new expression			

28. We'd like to set the debugger up so that it will update the windows each time the code runs. Since there is no line of code after the calculations, we'll choose one right before them and display the result of the last calculation.

    Click on the first line of code in the while(1) loop;

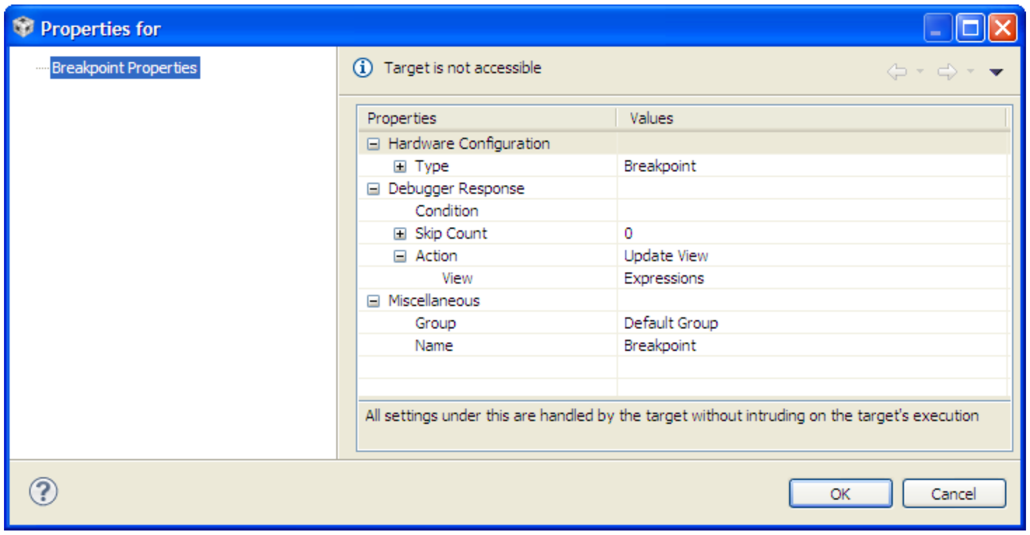
    `ADCIntClear(ADC0_BASE, 1);`

    and then right-click on it. Select Breakpoint (Code Composer Studio) then Breakpoint to set a breakpoint on this line.



```
    while(1)
    {
        ADCIntClear(ADC0_BASE, 1);
        ADCProcessorTrigger(ADC0_BASE, 1);
```

Right-click on the breakpoint symbol and select Breakpoint Properties … Find the line that contains Action and click on the Remain Halted value. That's the normal way a breakpoint should act, but let's change it to Update View (look up and down in the list). In the dialog below, note that only the Expressions window will be updated. Now the variables in the Expression window will be updated and the code will continue to execute. Click OK.



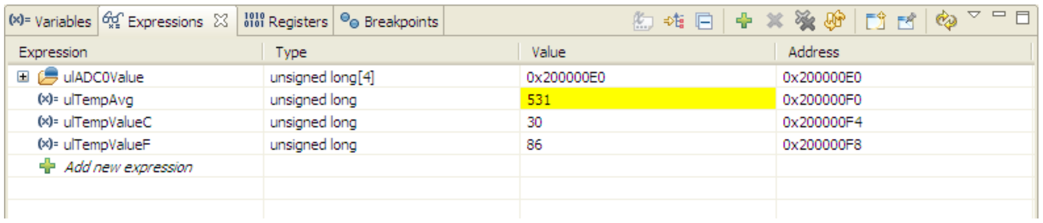
Target is not accessible

Properties	Values
Hardware Configuration	
Type	Breakpoint
Debugger Response	
Condition	
Skip Count	0
Action	Update View
View	Expressions
Miscellaneous	
Group	Default Group
Name	Breakpoint

All settings under this are handled by the target without intruding on the target's execution

29. Click the Resume button to run the program.

    You should see the measured value of `ulTempAvg` changing up and down slightly. Changed values from the previous measurement are highlighted in yellow. Use your finger (rub it briskly on your pants), then touch the
    LM4F120 device on the LaunchPad board to warm it. Press your fingers against a cold drink, then touch the device to cool it. You should quickly see the results on the display.



Expression	Type	Value	Address
ulADC0Value	unsigned long[4]	0x200000E0	0x200000E0
ulTempAvg	unsigned long	531	0x200000F0
ulTempValueC	unsigned long	30	0x200000F4
ulTempValueF	unsigned long	86	0x200000F8
Add new expression			

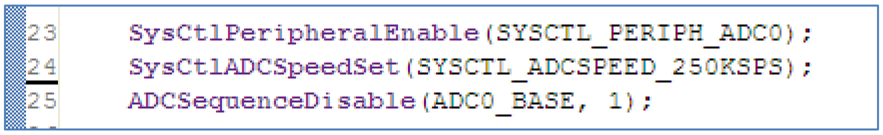
    Bear in mind that the temperature sensor is not calibrated, so the values displayed are not exact. That's okay in this experiment, since we're only looking for changes in the measurements.

    Note how much `ulTempAvg` is changing (not the rate, the amount). We can reduce the amount by using hardware averaging in the ADC.

Hardware averaging

30. Click the Terminate button to return to the CCS Edit perspective.

    Find the ADC initialization section of your code as shown below:

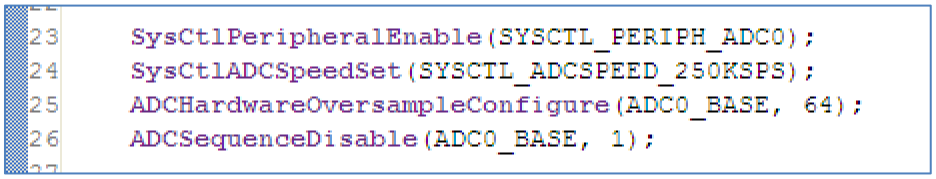


```
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
    SysCtlADCSpeedSet(SYSCTL_ADCSPEED_250KSPS);
    ADCSequenceDisable(ADC0_BASE, 1);
```

Right after the SysCtlADCSpeedSet() call, add the following line:

`ADCHardwareOversampleConfigure(ADC0_BASE, 64);`

Your code will look like this:



```
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
    SysCtlADCSpeedSet(SYSCTL_ADCSPEED_250KSPS);
    ADCHardwareOversampleConfigure(ADC0_BASE, 64);
    ADCSequenceDisable(ADC0_BASE, 1);
```

The last parameter in the API call is the number of samples to be averaged. This number can be 2, 4, 8, 16, 32 or 64. Our selection means that each sample in the ADC FIFO will be the result of 64 measurements averaged together.

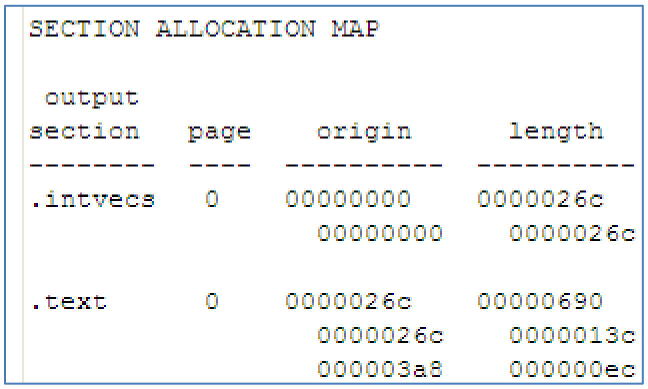
31. Build, download and run the code on your LaunchPad board. Observe the `ulTempAvg` variable in the Expressions window. You should notice that it is changing at a much slower rate than before.

Calling APIs from ROM

32. Before we make any changes, let's see how large the code section is for our existing project. Click the Terminate button to return to the CCS Edit perspective. In the Project Explorer, expand the Debug folder under the Lab5 project. Double-click on `Lab5.map`.

33. Code Composer keeps a list of files that have changed since the last build. When you click the build button, CCS compiles and assembles those files into relocatable object files. (You can force CCS to completely rebuild the project by either cleaning the project or rebuilding all). Then, in a multi-pass process, the linker creates the output file (.out) using the device's memory map as defined in the linker command (.cmd) file. The build process also creates a map file (.map) that explains how large the sections of the program are (.text = code) and where they were placed in the memory map.

    In the `Lab5.map` file, find the `SECTION ALLOCATION MAP` and look for `.text` like shown below:



```
SECTION ALLOCATION MAP

 output
section   page    origin      length
--------  ----  ----------  ----------
.intvecs   0    00000000    0000026c
                  00000000    0000026c

.text      0    0000026c    00000690
                  0000026c    0000013c
                  000003a8    000000ec
```

    The length of our `.text` section is `690h`. Check yours and write it here: ___________

34. Remember that the M4F on-board ROM contains the Peripheral Driver Library. Rather than adding those library calls to our flash memory, we can call them from ROM. This will reduce the code size of our program in flash memory. In order to do so, we need to add support for the ROM in our code.

    In main.c, add the following include statement as the last one in your list of includes at the top of your code:

    `#include "driverlib/rom.h"`

35. Open your properties for `Lab5` by right-clicking on Lab5 in the Project Explorer pane and clicking Properties. Under Build → ARM Compiler → Advanced Options, click on Predefined Symbols. Add the following symbol to the top window:

    `TARGET_IS_BLIZZARD_RA1`

    Blizzard is the internal TI product name for the LM4F series. This symbol will give the libraries access to the API's in ROM. Click OK.

36. Back in main.c, add `ROM_` to the beginning of every DriverLib call as shown below:

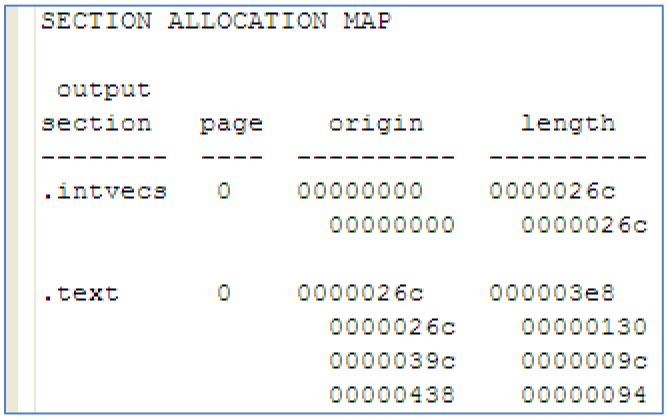
```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "driverlib/adc.h"
#include "driverlib/rom.h"

#ifdef DEBUG
void__error__(char *pcFilename, unsigned long ulLine)
{
}
#endif

int main(void)
{
 unsigned long ulADC0Value[4];
 volatile unsigned long ulTempAvg;
 volatile unsigned long ulTempValueC;
 volatile unsigned long ulTempValueF;

 ROM_SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);

 ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
 ROM_SysCtlADCSpeedSet(SYSCTL_ADCSPEED_250KSPS);
 ROM_ADCHardwareOversampleConfigure(ADC0_BASE, 64);
 ROM_ADCSequenceDisable(ADC0_BASE, 1);

 ROM_ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);
 ROM_ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_TS);
 ROM_ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_TS);
 ROM_ADCSequenceStepConfigure(ADC0_BASE, 1, 2, ADC_CTL_TS);
 ROM_ADCSequenceStepConfigure(ADC0_BASE, 1, 3, ADC_CTL_TS | ADC_CTL_IE | ADC_CTL_END);
 ROM_ADCSequenceEnable(ADC0_BASE, 1);

   while(1)
   {
    ROM_ADCIntClear(ADC0_BASE, 1);
    ROM_ADCProcessorTrigger(ADC0_BASE, 1);

    while(!ROM_ADCIntStatus(ADC0_BASE, 1, false))
    {
    }

    ROM_ADCSequenceDataGet(ADC0_BASE, 1, ulADC0Value);
    ulTempAvg = (ulADC0Value[0] + ulADC0Value[1] + ulADC0Value[2] + ulADC0Value[3] + 2)/4;
    ulTempValueC = (1475 - ((2475 * ulTempAvg)) / 4096)/10;
    ulTempValueF = ((ulTempValueC * 9) + 160) / 5;
    }
}
```

If you're having issues, this code is saved in your lab folder as `main2.txt`.

Build, Download and Run Your Code

37. Click the Debug button to build and download your code to the LM4F120H5QR flash memory. When the process is complete, click the Resume button to run your code. When you're sure that everything is working correctly, click the Terminate button to return to the CCS Edit perspective.

38. Check the SECTION ALLOCATION MAP in Lab5.map. Our results are shown below:



```
SECTION ALLOCATION MAP

 output
section    page    origin      length
--------   ----  ----------  ----------
.intvecs    0     00000000    0000026c
                    00000000    0000026c

.text       0     0000026c    000003e8
                    0000026c    00000130
                    0000039c    0000009c
                    00000438    00000094
```

The new size for our `.text` section is `3e8h`. That's 40% smaller than before. Write your results here: ________

39. The method shown in these steps is called "direct ROM calls". It is also possible to make mapped ROM calls when you are using devices (like the TI ARM Cortex-M3) that may or may not have a ROM. Check out section 32.3 in the Peripheral Driver Library User's Guide for more information on this technique.

40. When you're finished, , close the Lab5 project and minimize Code Composer Studio.

You're done.