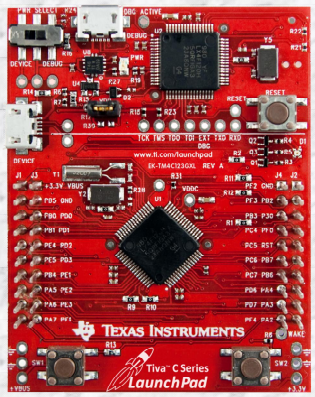
Introduction

This chapter will introduce you to the use of the analog to digital conversion (ADC) peripheral on the TM4C123GH6PM. The lab will use the ADC and the sequencer to sample the on-chip temperature sensor.

Agenda

Introduction to ARM® Cortex™-M4F and Peripherals
Code Composer Studio
Introduction to TivaWare™, Initialization and GPIO
Interrupts and the Timers
ADC12
Hibernation Module
USB
Memory and Security
Floating-Point
BoosterPacks and grLib
Synchronous Serial Interface
UART
µDMA
Sensor Hub
PWM



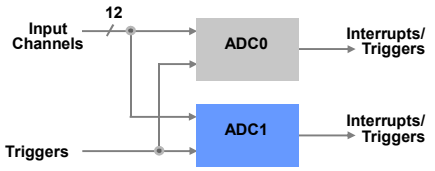
ADC...

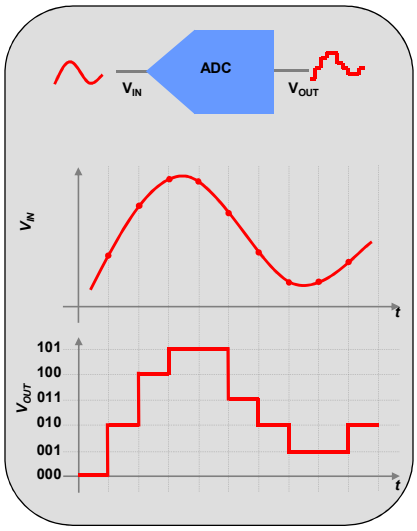
Chapter Topics

ADC12

Analog-to-Digital Converter

- Tiva TM4C 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...

TM4C123GH6PM 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

- Tiva TM4C 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.

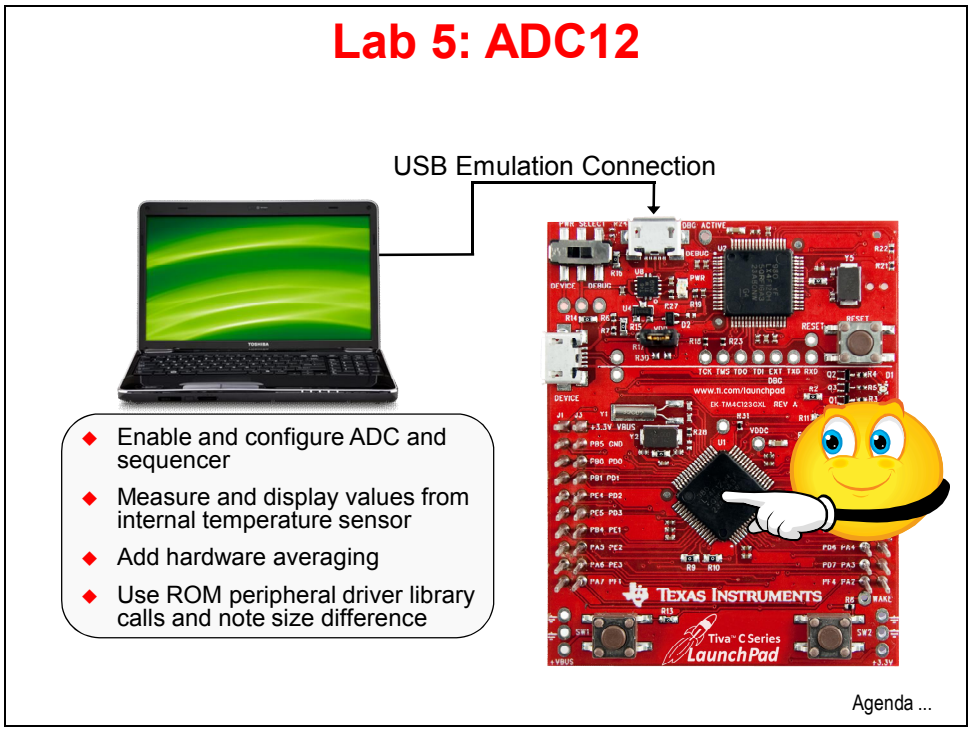
Sequencer	Number of Samples	Depth of FIFO
SS 3	1	1
SS 2	4	4
SS 1	4	4
SS 0	8	8

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.

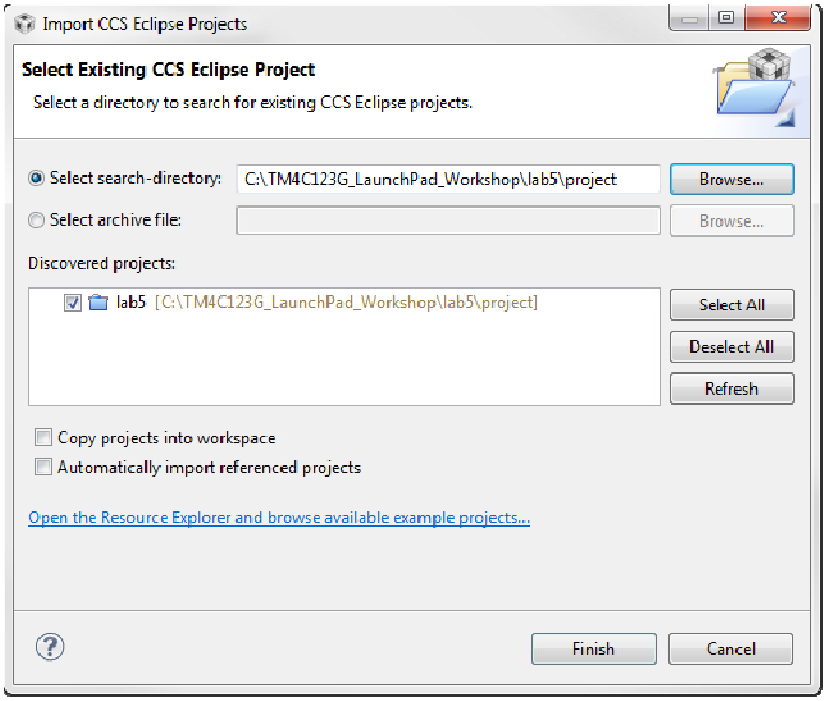


Procedure

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 isunchecked**.



Header Files

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

```
#include<stdint.h>
#include<stdbool.h>
#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

main()

3. ► Set up the `main()` routine by adding the three lines below:

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

4. 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()` :

```
uint32_t ui32ADC0Value[4];
```

5. 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 cannot 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:

```
volatileuint32_t ui32TempAvg;
volatileuint32_t ui32TempValueC;
volatileuint32_t ui32TempValueF;
```

6. 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);
```

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

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
```

8. For this lab, we'll simply allow the ADC12 to run at its default rate of 1Msps. Reprogramming the sampling rate is left as an exercise for the student.

   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);
```

9. 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);
```

10. 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);
```

11. Now we can enable ADC sequencer 1.

    ► Add this line directly after the last one:

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

12. Still within main(), add a while loop to the bottom of your code.

    ► Add a line for spacing and enter these three lines of code:

```
while(1)
{
}
```

13. ► Save your work.

    As a sanity-check, click on the Build button. If you are having issues, check the code on the next page:

```
#include<stdint.h>
#include<stdbool.h>
#include"inc/hw_memmap.h"
#include"inc/hw_types.h"
#include"driverlib/debug.h"
#include"driverlib/sysctl.h"
#include"driverlib/adc.h"

intmain(void)
{
        uint32_t ui32ADC0Value[4];
        volatileuint32_t ui32TempAvg;
        volatileuint32_t ui32TempValueC;
        volatileuint32_t ui32TempValueF;

        SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);

        SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);

        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 will get a warning "`ui32ADC0Value was declared but never referenced`". Ignore this warning for now, we'll add the 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.

14. The indication that the ADC conversion process 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);`

15. Now we can trigger the ADC conversion with software. ADC conversions can be triggered by many other sources.

    ► Add the following line directly after the last:

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

16. We need to wait for the conversion to complete. Obviously, a better way to do this would be to use an interrupt, rather than waste 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))
{
}
```

17. When code execution exits the loop in the previous step, we know that the conversion is complete andthat 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 sequenceif 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, ui32ADC0Value);
```

18. 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 directly after the last:

```
ui32TempAvg = (ui32ADC0Value[0] + ui32ADC0Value[1] + ui32ADC0Value[2] + ui32ADC0Value[3] + 2)/4;
```

19. 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 theTM4C123GH6PM 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:

```
ui32TempValueC = (1475 - ((2475 * ui32TempAvg)) / 4096)/10;
```

20. Once you have the Celsius temperature, calculating the Fahrenheit temperature is easy. Wait to perform the division operation 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:

```
ui32TempValueF = ((ui32TempValueC * 9) + 160) / 5;
```

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

```
#include<stdint.h>
#include<stdbool.h>
#include"inc/hw_memmap.h"
#include"inc/hw_types.h"
#include"driverlib/debug.h"
#include"driverlib/sysctl.h"
#include"driverlib/adc.h"

intmain(void)
{
     uint32_t ui32ADC0Value[4];
     volatileuint32_t ui32TempAvg;
     volatileuint32_t ui32TempValueC;
     volatileuint32_t ui32TempValueF;

     SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);

     SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);

     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, ui32ADC0Value);
     ui32TempAvg = (ui32ADC0Value[0] + ui32ADC0Value[1] + ui32ADC0Value[2] + ui32ADC0Value[3] + 2)/4;
     ui32TempValueC = (1475 - ((2475 * ui32TempAvg)) / 4096)/10;
     ui32TempValueF = ((ui32TempValueC * 9) + 160) / 5;
     }
}
```

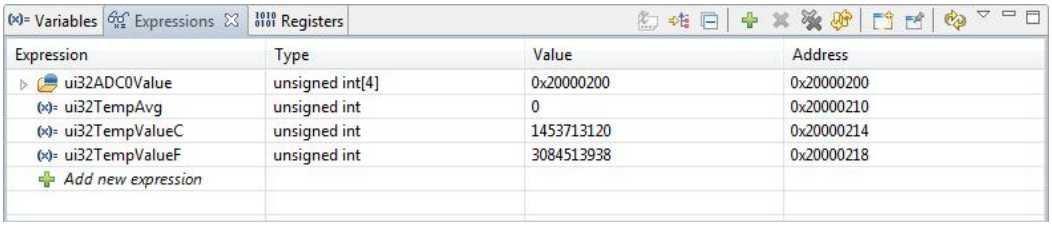
You can also find this code in `main1.txt` in your project folder.

Build and Run the Code

22. ► 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.

23. ► Click on theExpressionstab (upper right).Remove all expressions (if there are any) from the Expressions pane by right-clickinginside the pane and selecting *Remove All*.

    ► Find the **`ui32ADC0Value, ui32TempAvg, ui32TempValueC`**and**`ui32TempValueF`**variables in the last four lines of code. Double-click on each variable to highlight it, then right-click on it, select *Add Watch Expression* and then click *OK*. Do this for all four variables, one at the time.



Breakpoint

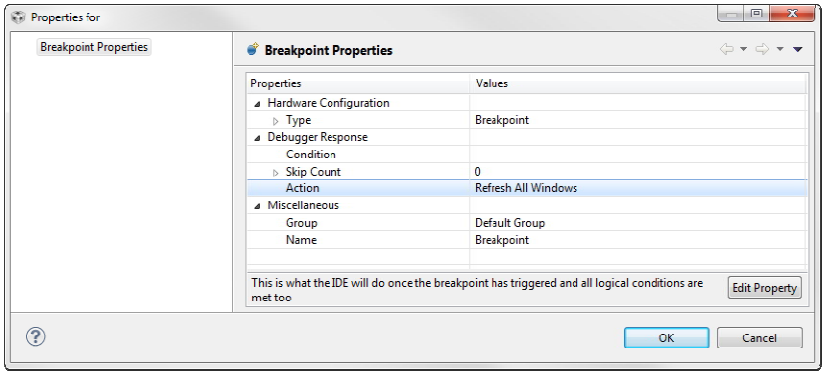
Let's set up the debugger so that it will update ourwatch windows each time the code runs. Since there's no line of code after the calculations are completed, we'll choose the one right before them and display the result of the last calculation.

24. ► Set a breakpoint on the first line of code in the `while(1)` loopby double-clicking in the blue area left of the line number.

```
33      while(1)
34      {
35          ADCIntClear(ADC0_BASE, 1);
36          ADCProcessorTrigger(ADC0_BASE, 1);
```

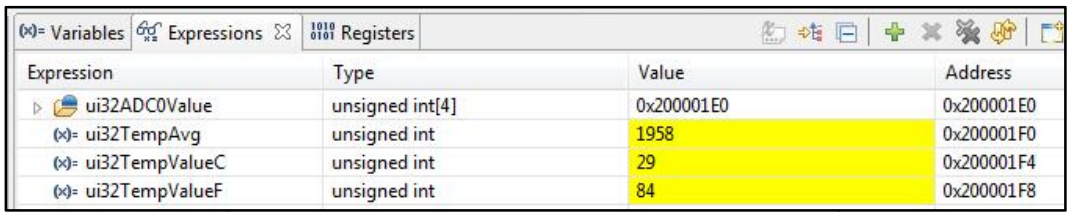
25. ► Right-click on the breakpoint symbol and select Breakpoint Properties … Find the Action line and click on the *Remain Halted* value.

► Click on the down-arrow that appears on the right and select *Refresh All Windows* from the list. ► Click *OK*.



26. ► Click the Resume button to run the program.If the Watch window does not immediately start updating, click the Suspend button and then the Resume button.

You should see the measured value of `ui32TempAvg`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 TM4C123GH6PM 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.



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

► Note the range over which`ui32TempAvg` is changing (not the rate of change, the amount). We can reduce the amount by using hardware averaging in the ADC.

Hardware averaging

27. ► Click the Terminate button to return to the CCS Edit perspective.

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

```
21
22      SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);
23
24      SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
25
26
```

Right after the `SysCtlPeripheralEnable()` API, ► add the following line:

`ADCHardwareOversampleConfigure(ADC0_BASE, 64);`

Your code will look like this:

```
21
22      SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);
23
24      SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
25      ADCHardwareOversampleConfigure(ADC0_BASE, 64);
```

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 being averaged together. We will then average four of those samples together in our code for a total of 256.

29. ► Build and download the code to your LaunchPad board. You may need to replace the breakpoint as shown in step 24 if you cheated and loaded the solution. Run the program and observe the `ui32TempAvg` variable in the Expressions window. You should notice that the range over which it is changingis much smaller than before.

This code is saved in `main2.txt` in your project folder.

Calling APIs from ROM

30. 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`.

31. When you click the build button, CCS compiles and assembles your source files into relocatable object files (`.obj`). Then, in a multi-pass process, the linker creates an output file (`.out`)using the device's memory map as defined in the linker command (`.cmd`) file along with any library (`.lib`) files.. 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

.init_array
*           0   00000000    00000000

.text       0   0000026c    000005e4
                  0000026c    00000104
                  00000370    000000d0
```

The length of our `.text` section is `5e4h`. ► Check yours and write it here: _____

32. Remember that the Tiva C Series device 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 ones in your list of includes at the top of your code:

```
#define TARGET_IS_BLIZZARD_RB1
#include "driverlib/rom.h"
```

Blizzard is the internal TI product name for the device family on your LaunchPad. This symbol will give the libraries access to the proper API's in ROM.

► Save your work.

33. ► Now add `ROM_` to the beginning of every DriverLib call as shown below in **`main.c`**:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "driverlib/adc.h"
#define TARGET_IS_BLIZZARD_RB1
#include "driverlib/rom.h"

int main(void)
{
         uint32_t ui32ADC0Value[4];
         volatile uint32_t ui32TempAvg;
         volatile uint32_t ui32TempValueC;
         volatile uint32_t ui32TempValueF;

         ROM_SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);

         ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
         ROM_ADCHardwareOversampleConfigure(ADC0_BASE, 64);

         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, ui32ADC0Value);
            ui32TempAvg = (ui32ADC0Value[0] + ui32ADC0Value[1] + ui32ADC0Value[2] + ui32ADC0Value[3] + 2)/4;
            ui32TempValueC = (1475 - ((2475 * ui32TempAvg)) / 4096)/10;
            ui32TempValueF = ((ui32TempValueC * 9) + 160) / 5;
         }
}
```

If you're having issues, this code is saved in your lab folder as `main3.txt`.

Build, Download and Run Your Code

34. ► Since you changed the instruction that the breakpoint was set on, the breakpoint has likely disappeared. Remove the indicated "breakpoint" if there is one by double-clicking on it. Add it back using the steps shown earlier.

35. ► Click the Debug buttonto build and download your code to the TM4C123GH6PM flash memory. When the process is complete, click the Resume buttonto run your code. When you're sure that everything is working correctly, click the Terminate buttonto return to the CCS Edit perspective.

36. 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

.init_array
*           0    00000000    00000000

.text       0    0000026c    000003c0
                   0000026c    00000118
                   00000384    0000009c
```

The original length of our `.text` section was `5e4h`. The new size is `3d4h`. That's 35% smaller than before.

Write your results here: _____

37. When you're finished, close the lab5 project and minimize Code Composer Studio.

You're done.