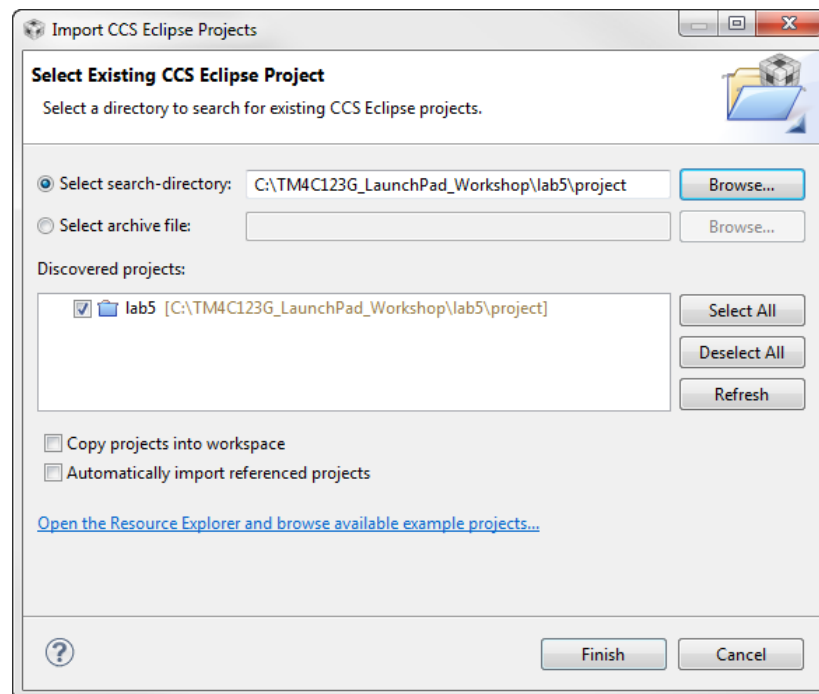


## Lab 5: ADC

### 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.**



#### *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 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

## 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. It will save the file name and line number of the 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, uint32_t ui32Line)
{
}#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()` :

```
uint32_t ui32ADC0Value[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 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:

```
volatile uint32_t ui32TempAvg;
volatile uint32_t ui32TempValueC;
volatile uint32_t ui32TempValueF;
```

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 peripheral next. ► Add a line for spacing and add this line after the last one:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
```

9. For this lab, we'll simply allow the ADC 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);
```

10. 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);
```

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"

#ifdef DEBUG
void __error__(char *pcFilename, unsigned uint32_t ui32Line)
{
}
#endif

int main(void)
{
    uint32_t ui32ADC0Value[4];
    volatile uint32_t ui32TempAvg;
    volatile uint32_t ui32TempValueC;
    volatile uint32_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);
```

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);
```

15. 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))  
{  
}
```

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

17. 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;
```

18. 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 the TM4C123GH6PM datasheet. Division is performed last to avoid truncation due to integer math rules. A later lab will cover floating point operations.

$$\text{TEMP} = 147.5 - ((75 * (\text{VREFP} - \text{VREFN}) * \text{ADCVALUE}) / 4096)$$

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.  $\text{VREFP} - \text{VREFN}$  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;
```

19. 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;
```

20. ► 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"

#ifdef DEBUG
void __error__(char *pcFilename, unsigned uint32_t u132lLine)
{
}
#endif

int main(void)
{
    uint32_t ui32ADC0Value[4];
    volatile uint32_t ui32TempAvg;
    volatile uint32_t ui32TempValueC;
    volatile uint32_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;
}
}

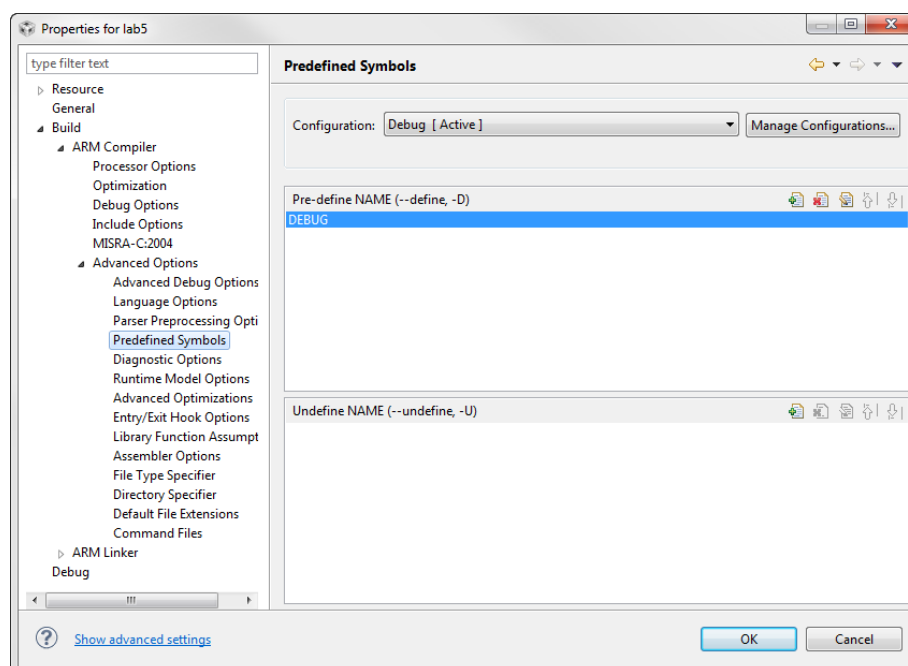
```

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

### Add Pre-defined Symbol

21. ► Right-click on `lab5` in the Project Explorer pane and select *Properties*. Under *Build* → *ARM Compiler*, expand *Advanced Options*. 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.



## 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 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 `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.

Expression	Type	Value	Address
ui32ADC0Value	unsigned int[4]	0x20000200	0x20000200
ui32TempAvg	unsigned int	0	0x20000210
ui32TempValueC	unsigned int	1453713120	0x20000214
ui32TempValueF	unsigned int	3084513938	0x20000218
Add new expression			

## Breakpoint

Let's set up the debugger so that it will update our watch 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)` loop by 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);

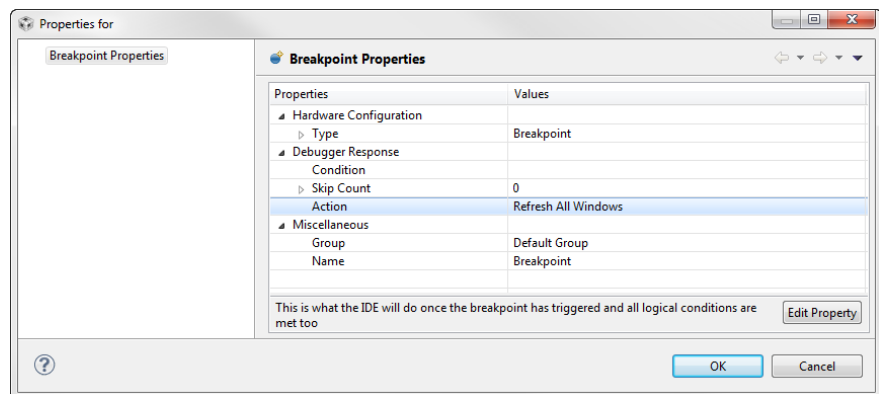
```

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.



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.



Expression	Type	Value	Address
ui32ADC0Value	unsigned int[4]	0x200001E0	0x200001E0
ui32TempAvg	unsigned int	1958	0x200001F0
ui32TempValueC	unsigned int	29	0x200001F4
ui32TempValueF	unsigned int	84	0x200001F8

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

- Click the Terminate button to return to the CCS Edit perspective.
- 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 `SysCtlClockSet()` 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);
26

```

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.

- Build, download and run the code on your LaunchPad board. Observe the `ui32TempAvg` variable in the Expressions window. You should notice that the range over which it is changing is much smaller than before.

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

## Calling APIs from ROM

4. 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.

5. 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	000005ec
		0000026c	00000104
		00000370	000000dc

The length of our .text section is 5ech. ► Check yours and write it here: \_\_\_\_\_

6. 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.

7. ► 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"

#ifdef DEBUG
void __error__(char *pcFilename, unsigned uint32_t u132lLine)
{
}
#endif

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

8. ► Click the Debug button to build and download your code to the TM4C123GH6PM 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.
9. Check the SECTION ALLOCATION MAP in lab5.map. Our results are shown below:

section	page	origin	length
.intvecs	0	00000000	0000026c
		00000000	0000026c
.init_array	*	00000000	00000000
.text	0	0000026c	000003c0
		0000026c	00000118
		00000384	0000009c

The original length of our `.text` section was `5ech`. The new size is `3c0h`. That's 36% smaller than before.

Write your results here: \_\_\_\_\_

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

### Follow the submission guideline to be awarded points for this Lab.

Task00: Execute the supplied code, no submission required.

Task 01: Change the ADC Sequencer to SS1 (4 sequence). Turn on the LED at PF2 if the temperature is greater than 72 degF. Use internal temperature sensor for all SS1 sequence. Display the temperature in the built-in graph tool.

Task 02: Introduce hardware averaging to 32. Using the timer TIMER1A conduct an ADC conversion for every 0.5 sec. Use the Timer1A interrupt. Display the temperature in the built-in graph tool.

### Follow the submission guideline to be awarded points for this Lab.

Follow the submission guideline to be awarded points for this Lab.

Submit the following for all Labs:

1. In the document, for each task submit the modified or included code (only) with highlights and justifications of the modifications. Also include the comments.
2. Create a Github repository with a random name (no CPE/403, Lastname, Firstname). Place all labs under the root folder TIVAC, sub-folder named LABXX, with one document and one video link file for each lab, place modified c files named as LabXX-TYY.c.

3. If multiple c files or other libraries are used, create a folder LabXX-TYY and place these files inside the folder.
4. The folder should have a) Word document (see template), b) source code file(s) with startup\_ccs.c and other include files, c) text file with youtube video links (see template).