

San José State University
Department of Computer Engineering
CMPE 146-03, Real-Time Embedded System Co-Design, Fall 2020

Lab Assignment 6

Due date: 11/08/2020, Sunday

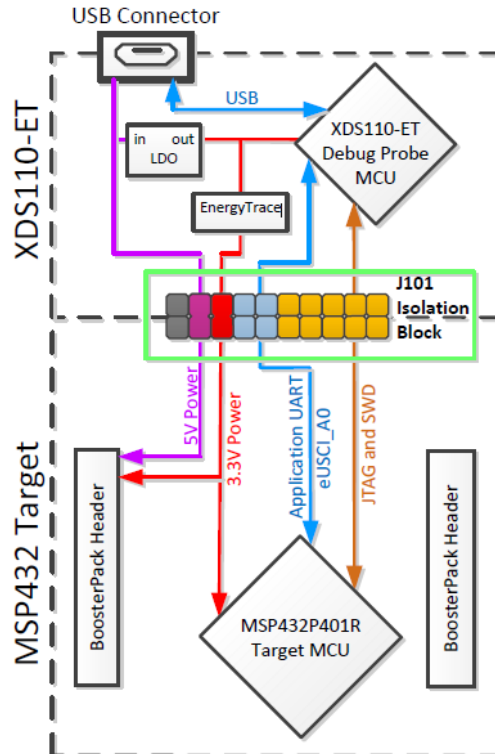
1. Description

In this assignment, we will be familiarized with the power consumption of the MCU side of the LaunchPad using TI's EnergyTrace technology, the ADC (analog-to-digital converter) and the temperature sensor of the MCU.

2. Exercise 1

EnergyTrace is a tool that measures and displays the application's energy profile. We can use it to optimize the application to reduce the power consumption of the system.

The circuitry that supports EnergyTrace resides on the debug probe (XDS110) side of the LaunchPad. It measures the current consumed by the MSP432 side in real time. It can sample, at kHz rate, the current used by the MCU and all the components on the MSP432 side through the 3.3-V supply voltage line.



For more information, consult the following websites:

http://processors.wiki.ti.com/index.php/EnergyTrace_for_MSP432

<http://www.ti.com/tool/ENERGYTRACE>

In this exercise, we are going to carry out the CRC-32 checksum computation in different ways and measure the total energy consumption required to complete the computation. There is no program to write. We just run an existing program a number of times to take energy measurements and create charts to present the results.

Create a new project (duplicate from the *empty* example project). Use the C file *Lab6_Ex1_measurement.c* on Canvas that comes with this PDF. Replace the contents of the main file of the project with this C file's contents. The program computes the CRC-32 checksum for a data block repeatedly for a number of times. Four methods are used: software, hardware, DMA and DMA in low-power mode. Let's label those methods as SW, HW, DMA and DMA-LPM, respectively.





The SW method is the same pure software method (using the *calculateCRC32()* function) to compute the CRC-32 checksum in a previous lab. The HW method uses the hardware accelerator but the processor transfers the data using a software loop. The DMA method also uses the hardware accelerator but it transfers the data with the DMA controller. So, almost the entire operation is done in hardware. The DMA-LPM method is identical to the DMA method except that once *main()* sets up all the hardware to do the work, the processor enters a low-power mode (LPM0) to sleep (to conserve energy), and wakes up only after the data transfer is done. In the DMA method, the processor does not go to sleep but busy-waits on the flag that signals the end of a data transfer.

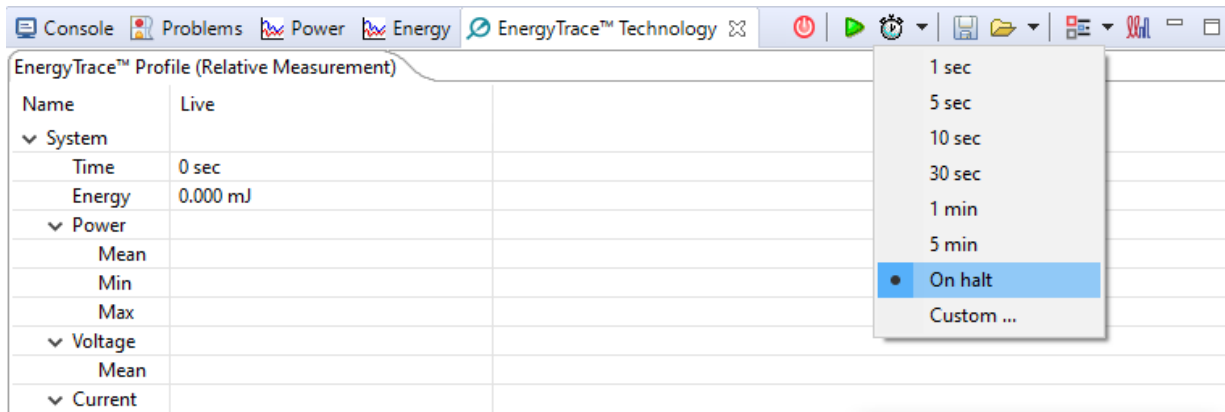
Which method to use is determined by the *#define* directives in the program:


```
#define USE_SW_METHOD
#define USE_HW_METHOD
#define USE_DMA_METHOD
#define USE_DMA_LPM_METHOD
```

To run a particular method, we will just comment out all the other *#define* statements.

Procedure to do measurement



Click on the debug icon  on the toolbar to start up the program. Before pressing the green arrow icon  to execute the program, we need to start up the EnergyTrace tool first. Click on the EnergyTrace icon  on the toolbar to bring up the EnergyTrace tab. Click on the stopwatch icon  and select "On halt" so that the current measurement on the LaunchPad will stop when the program stops execution.



Press the green arrow icon  to execute the program. During program execution, the EnergyTrace tab will update a number of values in real time. When the program finishes, the processor will halt and EnergyTrace will stop its measurement. And we would see something like this:

EnergyTrace™ Profile (Relative Measurement)	
Name	Live
▲ System	
Time	0 sec
Energy	1.341 mJ
▲ Power	
Mean	10.7727 mW
Min	10.6583 mW
Max	10.9471 mW
▲ Voltage	
Mean	3.3000 V
▲ Current	
Mean	3.2644 mA
Min	3.2298 mA
Max	3.3173 mA
Battery Life 2xAAA: 0 day (est.)	

We are interested in the two values: Energy and Mean Current. We will record those values for further processing. Energy (the value 1.341 mJ shown in the above figure) indicates how much energy was used to execute the program. Mean Current (the value 3.2644 mA in the above figure) is the average current measured during the program execution.

To have more reliable results, we will need to take multiple measurements for each method. To repeat the measurement, click on Restart icon  on the toolbar. Then click on the green arrow icon  to execute the program again.

Exercise 1.1

In this exercise, we are going to do the SW method only. Comment out the *#define* statements for the other methods.

```
#define USE_SW_METHOD
//#define USE_HW_METHOD
//#define USE_DMA_METHOD
//#define USE_DMA_LPM_METHOD
```

Take at least 3 measurements. Record the values of energy and mean current.

Lab Report Submission

1. Show the set of measured values.
2. Provide the program listing in the appendix.

Exercise 1.2

In this exercise, we are going to do the HW method only. Comment out the *#define* statements for the other methods.

```
//#define USE_SW_METHOD
#define USE_HW_METHOD
//#define USE_DMA_METHOD
//#define USE_DMA_LPM_METHOD
```

Take at least 3 measurements. Record the values of energy and mean current.

Lab Report Submission

1. Show the set of measured values.
2. Provide the program listing in the appendix.

Exercise 1.3

In this exercise, we are going to do the DMA method only. Comment out the *#define* statements for the other methods.

```
//#define USE_SW_METHOD
//#define USE_HW_METHOD
#define USE_DMA_METHOD
//#define USE_DMA_LPM_METHOD
```

Take at least 3 measurements. Record the values of energy and mean current.

Lab Report Submission

1. Show the set of measured values.
2. Provide the program listing in the appendix.

Exercise 1.4

In this exercise, we are going to do the DMA-LPM method only. Comment out the *#define* statements for the other methods.

```
//#define USE_SW_METHOD
//#define USE_HW_METHOD
//#define USE_DMA_METHOD
#define USE_DMA_LPM_METHOD
```

Take at least 3 measurements. Record the values of energy and mean current.

Lab Report Submission

1. Show the set of measured values.
2. Provide the program listing in the appendix.

Exercise 1.5

At this point, we should have four sets of results. Enter all the data in Excel (or another tool you prefer) in a table form to compute the averages for each method. Create two bar graphs, one for energy and another for mean current. The horizontal axis should be the methods. With the bar graphs, we can compare the four methods easily.

Lab Report Submission

1. Show the table of data.
2. Show the bar graphs.
3. Describe your observations. Provide brief explanations on the observations.

3. Exercise 2

There is a temperature sensor in the MCU. The output of the sensor can be routed to one of the ADC channels. We will convert the sensor's analog voltage to temperature in Celsius and Fahrenheit using the sensor calibration data stored in the ROM.

The temperature sensor resides in the voltage reference module of the MCU. It produces a voltage proportional to the internal temperature of the MCU. Software can monitor the temperature changes and takes necessary actions if warranted. For example, it can prevent the chip from overheating or compensate for certain sensitive components' electrical characteristics drift.

The sensor's output voltage can be digitized with one of the ADC channels on the MCU. The ADC has 14 bits of resolution, providing 16,384 levels of voltage digitization. The reference voltage is provided by the voltage reference module in the MCU. The module provides three selections of voltages: 1.2, 1.45 and 2.5 V. For example, if 2.5 V is used, the ADC provides a resolution of 0.15 mV ($2.5 / 16384$).

Details of the ADC and the temperature sensor can be found in the MCU datasheet (Sections 5.25.7 and 6.9.8) and technical reference (Section 22.2.10).

Exercise 2.1

In this exercise, we are going to digitize the temperature sensor's output voltage and display it on the debug console.

Create a new project (duplicate from the *empty* example project). Use the C file *Lab6_Ex2_template_adc.c* on Canvas that comes with this PDF. Replace the contents of the main file of the project with this C file's

contents. This template program basically digitizes the sensor voltage once and displays the converted digital value.

There are three functional units of the MCU being used in this program. The voltage reference module (REF_A) contains the temperature sensor and provides the voltage reference for analog-to-digital conversions. The ADC module (ADC14) contains a converter for digitization on a number of channels. The interrupt controller interrupts the processor when the digitization is completed.

To set up REF_A, we use the following functions:

```
REF_A_enableReferenceVoltage();  
REF_A_enableTempSensor();  
REF_A_setReferenceVoltage(REF_A_VREF2_5V);
```

REF_A_enableReferenceVoltage() enables the REF_A to be used. *REF_A_enableTempSensor()* enables the temperature sensor to be used. *REF_A_setReferenceVoltage()* selects the 2.5-V reference voltage to be used.

To enable ADC14 and route the sensor's output to one of the 32 conversion channels, we do the following.

```
ADC14_enableModule();  
ADC14_initModule(ADC_CLOCKSOURCE_MCLK, ADC_PREDIVIDER_1, ADC_DIVIDER_1, ADC_TEMPSENSEMAP);
```

ADC14_enableModule() enables the module to be used. *ADC14_initModule()* sets up the ADC clock for digitization. For this exercise, we use the main system clock (*ADC_CLOCKSOURCE_MCLK*) and do not scale it down (both clock dividers are 1: *ADC_PREDIVIDER_1*, *ADC_DIVIDER_1*) for the conversion operations. The function also routes the sensor's output to a conversion channel (which can be used for an external signal otherwise).

For this exercise, we only do one single conversion at a time with ADC14. When the conversion is done, ADC14 places the digitized value (conversion result) in one of the 32 registers in the module. These are configured with the following function call.

```
ADC14_configureSingleSampleMode(ADC_MEM0, false);
```

The second argument of *ADC14_configureSingleSampleMode()* determines if we want to repeat the conversion or not. In this case, we set it to *false* to disable the repeat action.

The digitized value is stored in register *ADC_MEM0*. The range of the reference voltage can be expressed with two end points: positive (higher) and negative (lower). We use an internal reference voltage source (2.5 V) as the positive end and ground (V_{SS}) as the negative end (*ADC_VREFPOS_INTBUF_VREFNEG_VSS*). The channel to be used is number 22 (*ADC_INPUT_A22*). These are set up using the following function.

```
ADC14_configureConversionMemory(ADC_MEM0, ADC_VREFPOS_INTBUF_VREFNEG_VSS, ADC_INPUT_A22, false);
```

The last argument determines whether to use differential mode at the input or not. In our case, the input is single-ended mode, so the argument is *false*.

The digitization process consists of two stages. The first stage is sample-and-hold and the second is conversion. The sample-and-hold time allows the input signal to be captured and held stable before the actual conversion to digital form can be done. The sample-and-hold time is programmable and the actual length depends on the nature of the input signal. The conversion time is fixed. For the 14-bit conversion that we are using, it takes 16 ADC clock cycles. The digitization timing is determined by the following function calls.

```
ADC14_setSampleHoldTime(ADC_PULSE_WIDTH_192, ADC_PULSE_WIDTH_192);
ADC14_enableSampleTimer(ADC_MANUAL_ITERATION);
ADC14_enableConversion();
```

ADC14_setSampleHoldTime() sets the sample-and-hold time for all channels be 192 clock cycles. *ADC14_enableSampleTimer()* sets the digitization process to be under software (or manual) control; it can be all done automatically and repeatedly with additional internal circuitry. *ADC14_enableConversion()* enables the conversion to occur.

The end of digitization will be signaled by an interrupt. The following function calls set up ADC14 to generate such interrupt and allow it to be received by the processor.

```
ADC14_enableInterrupt(ADC_INT0);
Interrupt_enableInterrupt(INT_ADC14);
Interrupt_enableMaster();
```

Finally, we trigger the entire digitization process by writing to a control register with the following function call.

```
ADC14_toggleConversionTrigger();
```

When the conversion completes, the interrupt service routine *ADC14_IRQHandler()* will be invoked. It makes sure the interrupt indeed comes from our intended ADC source and then set the global flag *adc_done* to notify *main()* that the conversion is done.

```
void ADC14_IRQHandler(void)
{
    uint64_t status;
    status = ADC14_getEnabledInterruptStatus();
    ADC14_clearInterruptFlag(status);

    if (status & ADC_INT0)
    {
        adc_done = true;
    }
}
```

For this exercise, add a loop in *main()* to continually digitize the sensor output voltage, display the digitized value and the corresponding voltage in mV. You may want to add some delay (using the delay function that you had written before or was provided to you), like 500 ms, in the loop to slow down the display a bit.

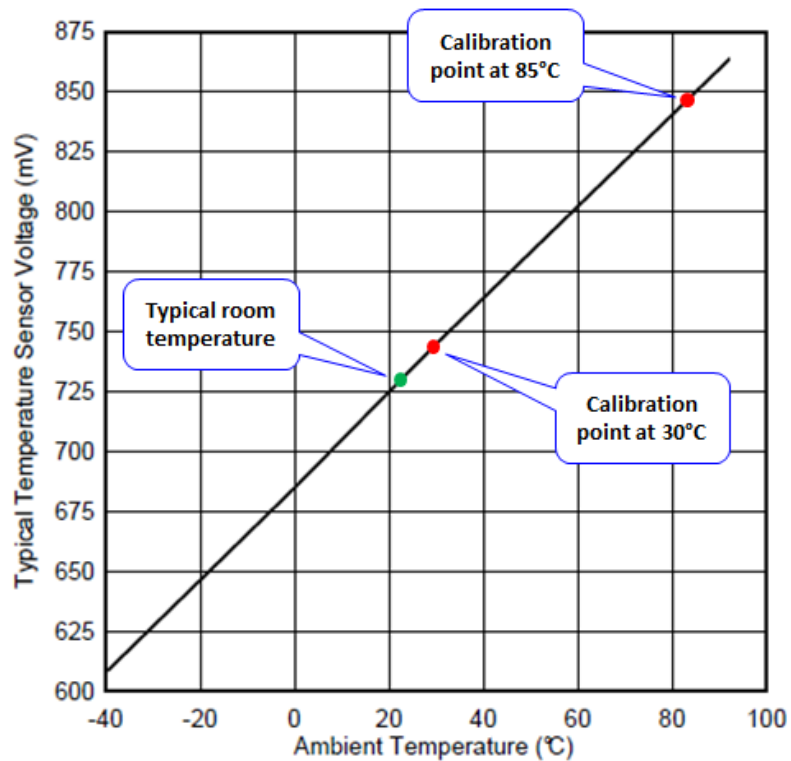
Lab Report Submission

1. List the code that does the repeated actions mentioned above.
2. Show the debug console outputs.
3. Provide the program listing in the appendix.

Exercise 2.2

Once we learn how to convert an analog signal to a digital form, we can transform it to something that has a physical meaning. In this exercise, we are going to convert the digitized value to temperature in Celsius and Fahrenheit using some calibration data stored in the ROM.

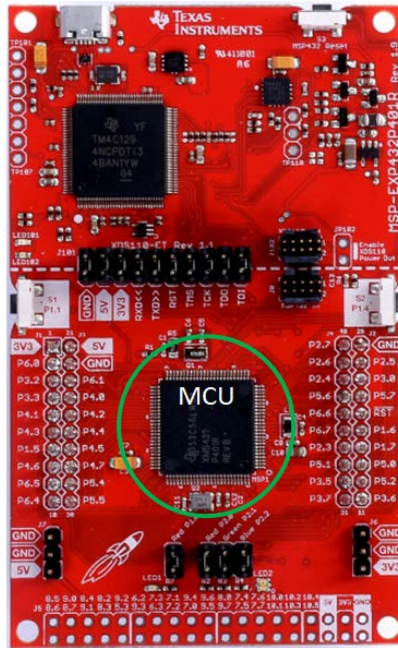
The picture below shows a typical linear relationship between the temperature and the temperature sensor's output voltage.



Such linear relationship is defined by two calibration points on the curve. Data of these two calibration points are stored in the Device Descriptor Table. The calibration data are stored in the “ADC14 Calibration” section of the table. Each calibration point is the temperature sensor output voltage’s digitized value at a specific temperature with a specific reference voltage applied to the ADC. In the final stage of the MCU manufacturing process, the factory would place the chip in a constant temperature environment and record the digitized value at various different reference voltages. The values are then burned to the ROM. As shown on the graph, two temperature points are used, one at 30°C and another one at 85°C.

For this exercise, duplicate the previous project and expand it to do more work. Since the voltage reference being used is 2.5 V, pick out the calibration data for such voltage from the Device Descriptor Table. Use the calibration data and the newly digitized value to compute the temperature of chip, which should be close to the room temperature because the MCU is not very busy. Display the temperature measured in Celsius and Fahrenheit on one line repeatedly. Show the temperature values with two decimal places of precision. As in the previous exercise, you may want to add some delay to slow down the display.

The sensor is sensitive to its environment. If you touch the surface of the MCU, you should get a higher reading because the finger’s temperature is higher than the bare surrounding. If you rub your finger for a few seconds before touching the chip, you should get an even higher reading.






Lab Report Submission

1. List the code that does the repeated actions mentioned above.
2. Show the debug console outputs. Highlight the readings between touch and no-touch conditions.
3. Provide the program listing in the appendix.

4. Submission Requirements

Write a formal report that contains at least what are required to be submitted in the exercises. Submit the report in PDF to Canvas by the deadline. Include source code in your report. As for the report contents, do not use screenshots to show your codes. In the report body, list the relevant codes or screenshots only for the exercise you are discussing. Put the entire program listing in the appendix. Use single-column format.

5. Grading

Lab 6 Rubric   				
Criteria	Ratings			Pts
Exercise 1.1	0.5 pts 0.5 Reported reasonable results. Program listing in appendix.	0 pts 0 Not attempted or reported.		0.5 pts
Exercise 1.2	0.5 pts 0.5 Reported reasonable results. Program listing in appendix.	0 pts 0 Not attempted or reported.		0.5 pts
Exercise 1.3	0.5 pts 0.5 Reported reasonable results. Program listing in appendix.	0 pts 0 Not attempted or reported.		0.5 pts
Exercise 1.4	0.5 pts 0.5 Reported reasonable results. Program listing in appendix.	0 pts 0 Not attempted or reported.		0.5 pts
Exercise 1.5	3 pts 3 Described observations and provided reasonable explanations.	2 pts 2 Provided table of data and bar graphs.	0 pts 0 Not attempted or reported.	3 pts
Exercise 2.1	2 pts 2 Correct program implementation. Program listing in appendix.	1 pts 1 Showed relevant code and reasonable console outputs.	0 pts 0 Not attempted or reported.	2 pts
Exercise 2.2	3 pts 3 Correct program implementation. Program listing in appendix.	2 pts 2 Showed relevant code and reasonable console outputs.	0 pts 0 Not attempted or reported.	3 pts
Total Points: 10				

Grading Policy

The lab's grade is determined by the report. If you miss the submission deadline for the report, you will get zero point.