San José State University
Department of Computer Engineering
CMPE 146-01, Real-Time Embedded System Co-Design, Fall 2024

Lab Assignment 6

Due date: 11/17/2024, Sunday

1. Overview

In this assignment, we will be familiarized with the UART protocol and the energy consumption of the MCU side of the LaunchPad using TI's EnergyTrace technology.

2. Exercise 1 UART

UART is a serial communication protocol widely used in embedded systems. It is simple, reliable, and provides reasonable speeds for a lot of applications. In this exercise, we will have the LaunchPad communicate with the connected laptop/PC through a UART controller using two different methods.

Exercise 1.1 Send text string

In this exercise, we have an application running on the laptop/PC to display what the LaunchPad outputs through the UART controller and to send some data to the LaunchPad.

Install a terminal program on your laptop/PC if you have not done so before. The terminal program displays what it receives from a COM port and sends your keystrokes to the connected device. There are many such tools available from the Web. For example, we can download PuTTY from https://www.puttygen.com/download-putty. (CCS has a terminal window function, too. But it is more convenient and easier to use a standalone program. Plus, we need a program independent of CCS to verify the working of our application running on the LaunchPad. Therefore, do not use it for this exercise.)

Start up the terminal program on the laptop/PC. Use the baud rate of 9600. On a Windows machine, the COM port that is used to connect to the LaunchPad can be found on the Device Manager. For example, with the LaunchPad USB cable plugged in, if COM5 is being used, Device Manager will show such, as in Figure 1.

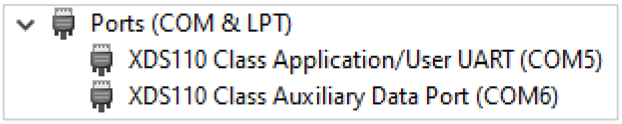

Ports (COM & LPT)
XDS110 Class Application/User UART (COM5)
XDS110 Class Auxiliary Data Port (COM6)

Figure 1. COM ports on Device Manager

If you use PuTTY, the basic setup configuration screen may look like this:

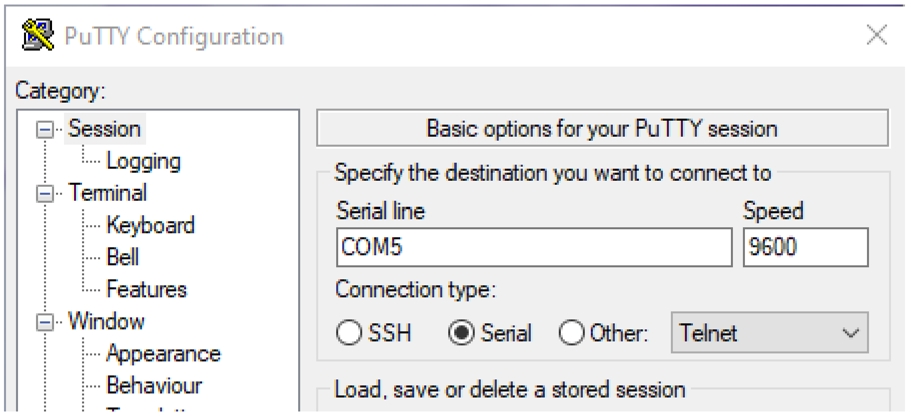

PuTTY Configuration
Category:
Session
Logging
Terminal
Keyboard
Bell
Features
Window
Appearance
Behaviour
Basic options for your PuTTY session
Specify the destination you want to connect to
Serial line: COM5
Speed: 9600
Connection type: SSH [✓ Serial] Other: Telnet
Load, save or delete a stored session

Figure 2. PuTTY session settings on Windows

On a Mac machine, the directory /dev would contain the desired port name. The configuration screen may look like this:

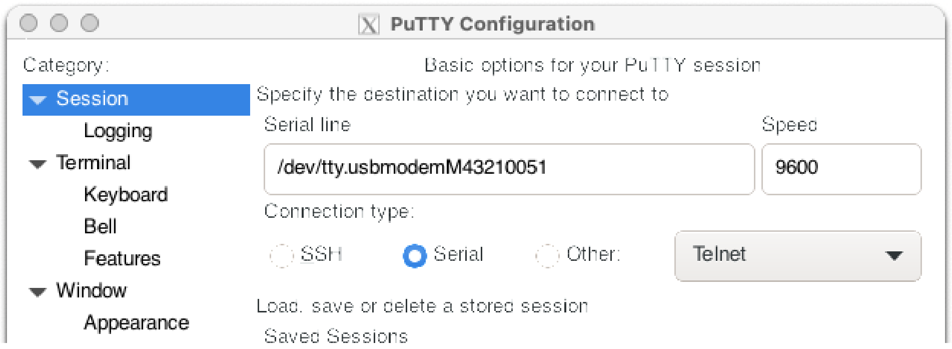
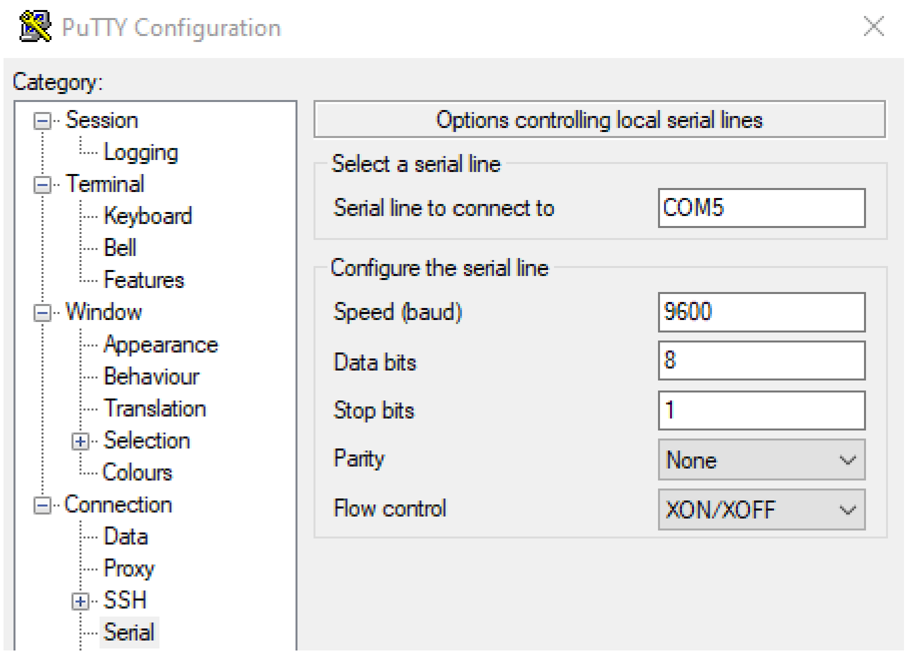

PuTTY Configuration
Category:
Session
Logging
Terminal
Keyboard
Bell
Features
Window
Appearance
Basic options for your PuTTY session
Specify the destination you want to connect to
Serial line: /dev/tty.usbmodemM43210051
Speed: 9600
Connection type: SSH [✓ Serial] Other: Telnet
Load, save or delete a stored session
Saved Sessions

Figure 3. PuTTY session settings on macOS

Make sure the number of data bits is 8, 1 stop bit and no parity. The detailed configuration screen may look like this:



Select a serial line

Serial line to connect to: COM5

Configure the serial line

Speed (baud): 9600

Data bits: 8

Stop bits: 1

Parity: None

Flow control: XON/XOFF

Figure 4. PuTTY COM port communication parameters

Import the example project, C:\ti\simplelink_msp432p4_sdk_3_40_01_02\examples\nortos\MSP_EXP432P401R\driverlib\uart_pc_echo_12mhz_brclk\ccs. This program echoes whatever it receives from the UART controller.

Build and run the program on CCS. On the terminal program, type something on the keyboard and you should see what you just type because the LaunchPad echoes what it receives. We use this example project just to make sure all the hardware and software are working properly on both sides. Note that this test does not require CCS to work; you can exit CCS and you should still see what you type on the terminal program.

The UART controller is part of the Enhanced Universal Serial Communication Interface (eUSCI) of the MCU. eUSCI supports a number of serial communication controllers on the MCU. To identify the specific controller that (eventually) connects to the external laptop/PC, we use the defined constant *EUSCI_A0_BASE* to the DriverLib function calls.

Now, add a small function to the example project to send a null-terminated character string to the UART controller. The function prototype is as follows.

```
void send_message(char* msg);
```

In the function, do not use any DriverLib functions. We are going to directly access the controller's registers to do the work. We send one character at a time to the Transmit Buffer Register (UCA0TXBUF) until the null byte is encountered (there is no need to send the null byte). Upon receiving a character, the UART controller serializes the 8-bit data and sends one bit at a time out of the MCU. Since sending out the bits is a lot slower than what the processor can send the characters to the controller, we'll need to check the Status Register (UCA0STATW) to make sure that the controller is ready to receive a character.

In the function, declare two 16-bit word pointers to access the UCA0TXBUF and UCA0STATW registers. Consult the MCU datasheet (Table 6-1) to find the base address of the controller and the offsets of those two registers. Consult the MCU technical reference (Chapter 24) for details on how they should be used. There is a “busy” bit in the status register to indicate the transmission or reception progress in the controller. Use it to determine if the controller is ready to accept another data byte for transmission in your function.

In *main()*, before it enters the while loop, add the following statements to test your function.

```
static char something[] =
{0x0d,0x0a,0x4d,0x6f,0x6f,0x6f,0x6f,0x6f,0x20,0x20,0x20,0x20,0x20,0x20,0x5e,0x
5f,0x5f,0x5e,0x0d,0x0a,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,
0x20,0x28,0x6f,0x6f,0x29,0x5c,0x5f,0x5f,0x5f,0x5f,0x5f,0x5f,0x5f,0x0d,0x0a,0x2
0,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x28,0x5f,0x5f,0x29,0
x5c,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x29,0x5c,0x2f,0x5c,0x0d,0x0a,0x20,0x20
,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x7c,0x
7c,0x2d,0x2d,0x2d,0x2d,0x77,0x20,0x7c,0x0d,0x0a,0x20,0x20,0x20,0x20,0x20,0x20,
0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x7c,0x7c,0x20,0x20,0x20,0x2
0,0x20,0x7c,0x7c,0x0d,0x0a,0x00};
send_message(something);
```

After you start the program, you should see a figure on the terminal window. Then randomly type something to make sure that the echoing function still works.

Lab Report Submission

1. List the *send_message()* function. Provide a brief explanation on how it works.
2. Provide a screenshot of the terminal window showing the test string and what you have typed.
3. Provide the program listing in the appendix.

Exercise 1.2 Emulate UART

In this exercise, to reinforce our understanding of the basic serial communication concepts, we will bypass the UART controller to send a character string to the laptop/PC. We will send the data bit by bit entirely using just the software, emulating what the UART controller would do.

The basic UART protocol involves two signals, *transmit* and *receive*, as shown in Figure 5.

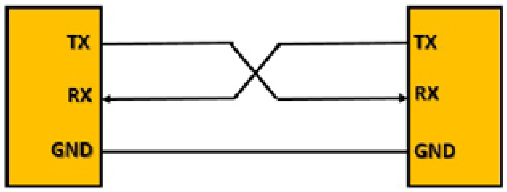


Figure 5. Basic UART wiring configuration

On the MSP432 MCU, the UART controller sends the bit stream out through the *transmit* signal, Tx, which is an output pin on the MCU. Similarly, the controller receives the incoming bit stream through the *receive* signal, Rx, which is an input pin on the MCU. Tx and Rx share the pins with the GPIO module. As shown in Figure 6, for the data channel communicating with the laptop/PC, Tx is on P1.3 and Rx is on P1.2. We can write a small program to change the state of the P1.3 GPIO pin directly to create the bit stream the same way the UART controller would do, thus bypassing the controller entirely to transmit the data.

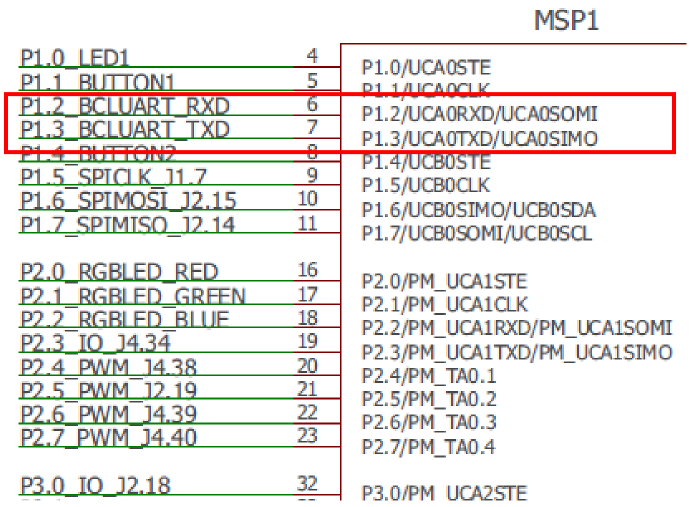

MSP1
P1.0_LED1 4 P1.0/UCA0STE
P1.1_BUTTON1 5 P1.1/UCA0CLK
P1.2_BCLUART_RXD 6 P1.2/UCA0RXD/UCA0SOMI
P1.3_BCLUART_TXD 7 P1.3/UCA0TXD/UCA0SIMO
P1.4_BUTTON2 8 P1.4/UCB0STE
P1.5_SPICLK_J1.7 9 P1.5/UCB0CLK
P1.6_SPIMOSI_J2.15 10 P1.6/UCB0SIMO/UCB0SDA
P1.7_SPIMISO_J2.14 11 P1.7/UCB0SOMI/UCB0SCL
P2.0_RGBLED_RED 16 P2.0/PM_UCA1STE
P2.1_RGBLED_GREEN 17 P2.1/PM_UCA1CLK
P2.2_RGBLED_BLUE 18 P2.2/PM_UCA1RXD/PM_UCA1SOMI
P2.3_IO_J4.34 19 P2.3/PM_UCA1TXD/PM_UCA1SIMO
P2.4_PWM_J4.38 20 P2.4/PM_TA0.1
P2.5_PWM_J2.19 21 P2.5/PM_TA0.2
P2.6_PWM_J4.39 22 P2.6/PM_TA0.3
P2.7_PWM_J4.40 23 P2.7/PM_TA0.4
P3.0_IO_J2.18 32 P3.0/PM_UCA2STE

Figure 6. LaunchPad schematic showing the UART connections

We cannot just send the data bits to the output pin. They have to be encapsulated in a frame, as shown in Figure 7. Therefore, in the software, we also have to add the header and trailer parts of the frame.

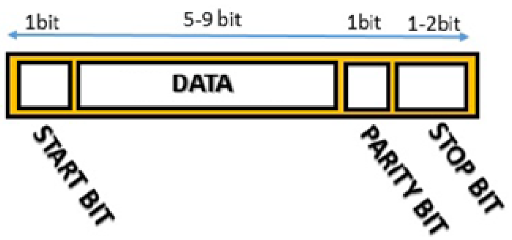


Figure 7. UART communication frame

Create a new project by importing the *empty* example project on CCS (be sure to include *stdio.h*). Write a small program to send a character string to be displayed on the terminal program running on the laptop/PC. Do not make use of anything from the previous exercise because we are going to do things very differently. The program will only deal with the transmit part of the communication; it won't process the keyboard inputs from the terminal program (as in the previous exercise).

Write a *send_message* function as before. Use exactly the same prototype. Since we are not using the UART controller, its implementation will not have anything to do with the controller's registers at all. When Tx is idling, it is at a Logic-High level. The start bit is a Logic Low. The first bit of the data payload is the LSB of the character being sent. The stop bit(s) is Logic High as the line needs to return to its idle state.

During transmission, each bit stays at the proper state on the Tx pin for a fixed duration of time, which is determined by the baud rate. For example, if the baud rate is 9600 (bps, bits per second), the duration is 104.2 µs. Use the delay function *__delay_cycles()* to implement such timing control.

When the program starts, we just need to initialize two devices: P1.3 (Tx) as output and P1.2 (Rx) as input. After the device initialization, set the Tx pin to its idle state, then delay for some time (100 ms or so) to set up the initial channel condition, to make sure any pre-existing bus transaction will be terminated. Before entering a dummy forever *while* loop, send the same test string (as in the previous exercise) to the laptop/PC:

```
send_message(something);
```

Since we are using all software to control the timing, it may be difficult to achieve very high baud rate. In addition, in order to make sure that the logic of implementation is correct, on the terminal program, set the baud rate to a relatively slow 1200 baud. In the program, we can assume that it uses only a fixed baud rate. Define the baud rate like the following in your program.

```
#define BAUD_RATE 1200
```

In the program, do not hard-code the timing (the delay time) in sending out the bit stream. Derive the necessary timing parameter from the defined constant BAUD_RATE so that we can change the baud rate easily. You can assume that the system is operating at the default frequency of 3 MHz.

If the logic of our implementation is correct and the control timing is not too far off, we should see the test string correctly displayed on the terminal program. Since we have the parity bit set to *None* on the terminal program, there is no need to create the parity bit in the output bit stream.

After the program works at 1200 baud, change it to 9600 (a much higher rate). Build and test the program again. If it does not work, it is most likely a timing issue since it has worked at a lower rate. Adjust the program accordingly to make the program work again at 9600 baud.

Note that the program is not set up to handle any inputs through the UART Rx pin, typing anything on the terminal program would produce no effect, i.e., no echoing as in the previous exercise.

Lab Report Submission

1. List the *send_message()* function and related code to create the bit stream. Provide a brief explanation on how it works.
2. Report any adjustments, if any, to the program to make it work at 9600 baud.
3. Provide a screenshot of the terminal window showing the test string.
4. Provide the program listing in the appendix.

Exercise 1.3 Parity bit

In this exercise, we will add the parity generation to the bit stream. Based on what you have done in the previous exercise, add the code to generate the parity bit.

Use a macro to define the parity type used in the program, like the following:

```
#define PARITY 1            // 0: none, 1: odd parity, 2: even parity
```

For example, if odd parity is used, the total number of data bits and the parity bit that are ‘1’ must be odd. The program should generate the parity or not according to the macro. On the terminal program, set the parity setting accordingly for testing. If the parity bit is not generated correctly, the terminal program would discard the character received, so you probably will not see the complete test string.

Lab Report Submission

1. List the code to generate the parity bit. Provide a brief explanation on how it works.

2. With the parity set to even or odd in the program and on the terminal program, provide a screenshot of the terminal window showing the test string and another screenshot of the terminal program's settings, showing the parameters including the parity.
3. Provide the program listing in the appendix.

3. Exercise 2. EnergyTrace

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.

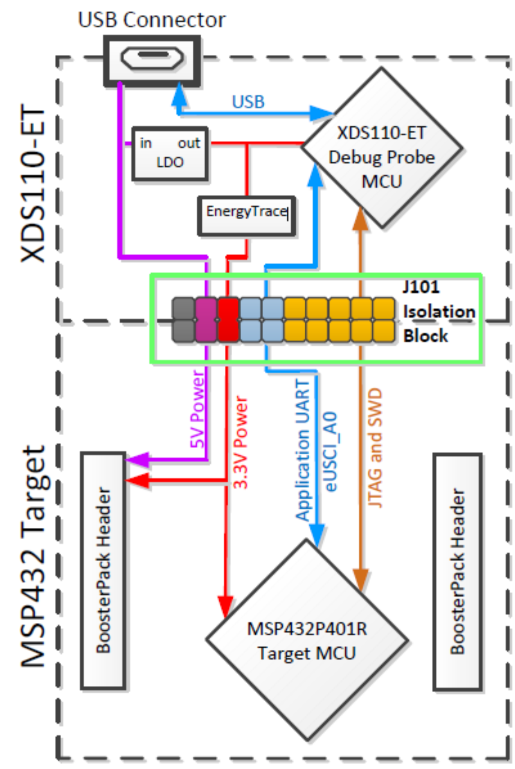


Figure 8. LaunchPad main component block diagram

For more information, consult the following websites:
http://processors.wiki.ti.com/index.php/EnergyTrace_for_MSP432
http://www.ti.com/tool/ENERGYTRACE

Exercise 2.1 Energy measurements

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_Ex2_1_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 (LPM0). 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 transfer 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.

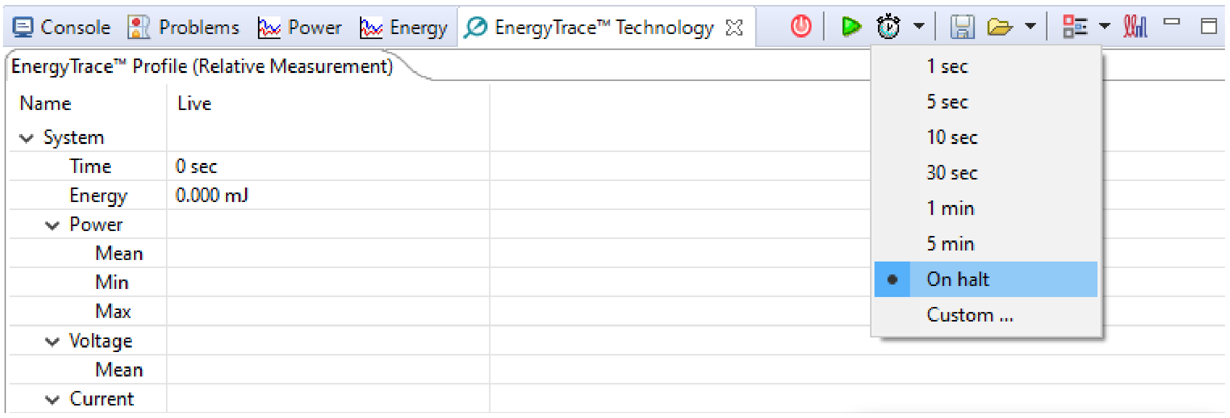
Which method to use is determined by the *#define* directives in the program:

```c
#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.



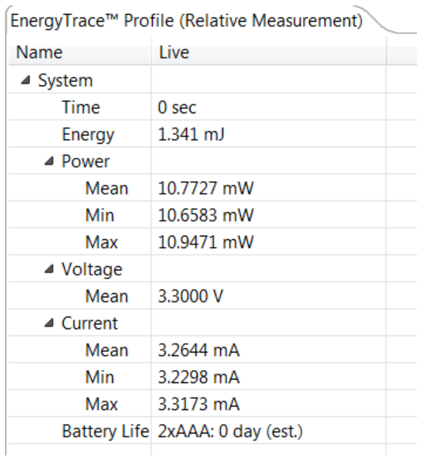
EnergyTrace™ Profile (Relative Measurement)

Name	Live
System	
Time	0 sec
Energy	0.000 mJ
Power	
Mean	
Min	
Max	
Voltage	
Mean	
Current	

● On halt

Figure 9. EnergyTrace setup

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.)

Figure 10. EnergyTrace measurement results

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.

We are going to collect data with the four different methods. For example, 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.

With the four sets of results, enter all the data in Excel (or a different tool you prefer) in a table form to compute the averages for each method. Create two separate 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.

Exercise 2.2 LPM4.5

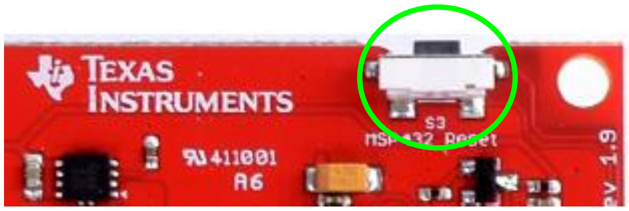
In this exercise, we will compare the energy consumptions between two implementations doing exactly the same thing. The task here is to blink the red LED (LED1) once when the push button S1 is pressed. One implementation uses polling and the other uses interrupt with low-power mode LPM4.5, which essentially shuts down the MCU when it is idling.

Create a new project (duplicate from the *empty* example project). Use the C file *Lab6_Ex2_2_LPM45.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 C file illustrates the mechanism of how to build an application using LPM4.5 to keep power consumption low. Since returning from the LPM4.5 (to Active Mode) involves a power-on reset (POR), some special handling is needed during the reset process. The program implements only the interrupt method; you will need to implement the polling method. Study the interrupt method and implement the polling method to produce the same program behavior of the interrupt method.

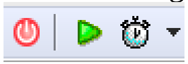
Note that if the MCU is in LPM4.5, where almost all components are shut down, we cannot run the program under debug mode on CCS because certain components must remain powered on. LPM4.5 won't be entered when the program is being debugged. Therefore, do not press the green icon to run the program.

We will collect the same types of data as in the previous lab, i.e., energy and mean current, using EnergyTrace. We will collect the data in a 10-second window. Here is the procedure:

1. Build the program. After CCS downloads the program to the LaunchPad, do not click on the green icon to run the program. Instead, click on the red square icon.
2. Click on the icon to start up EnergyTrace.
3. Click on the stopwatch icon and select "10 sec."
4. Do a hardware reset to the LaunchPad by pressing the reset button (shown below).



5. Click on the green triangle to start data collection.



6. After the data collection has started and "Time" is incrementing, press the push button S1 once to blink the red LED.
7. When time is up, record the data.

Collect three sets of data. To repeat the data collection, we can start from Step 4. (During the measurement, you should not see any other LEDs (LED1 and LED2) being turned on. If that does happen, redo the measurement.)

The program provided sets up the GPIO port so that it generates an interrupt when S1 is pressed. After the setup is done, it will enter LPM4.5 to wait for the event to occur, and to save energy consumption. When the interrupt happens, the MCU basically goes through a hardware reset process to return back to Active Mode (AM). That is the only way the MCU transitions from LPM4.5 to AM. At the beginning of the program, it will check if the program starts up due to returning from LPM4.5 or not, then set things up accordingly. Before LPM4.5 is re-entered, the ISR is invoked and the LED is blinked once.

There is a macro to determine which method the program is using. So, to use the polling method, make sure the following line is not commented, and then rebuild the program.

```
#define USE_POLLING
```

If *USE_POLLING* is defined, the program will use the polling method, not the LPM4.5 method. The program behaviors from these two methods must be the same. You will need to implement the polling method, similar to something that you have done in a previous lab. Basically, the program keeps monitoring the state of S1 and blinks the LED the same way that the LPM4.5 method does. Place the necessary code only in the forever control loop in *main()*, between "*#ifdef USE_POLLING*" and "*#else*" in the code snippet shown below. Do not change or add any code in other parts of the program.

```
    while(1)
    {
#ifdef USE_POLLING
        // To be implemented
#else
        MAP_PCM_shutdownDevice(PCM_LPM45);
#endif
    }
```

Compute the averages of the measurements. You would have two sets of results, one for each implementation. Compute how much energy saving in terms of percentage you get from using LPM4.5.

Lab Report Submission

1. Show the sets of data.
2. Report the energy saving in percentage.
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 program source code in your report. Do not use screenshots to show your codes. In the report body, if requested, 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 for the report.

In your report, show the debug console outputs (when requested) in text format; copy and paste them to your report. In general, do not use screenshot to show any text outputs or data; otherwise, points will be deducted.

When presenting a screenshot in the report, only present the relevant information. You may need to crop them from the initial screenshot. Please use proper scale so that the contents can be read easily.

Your report should have several sections. The first section is the information header that contains at least your name, course name and lab assignment number. Each exercise should have its own section where you discuss the results of the exercise. At the end, if requested, there should be the appendix that contains the complete source code for each exercise. The source code may be used to reproduce your reported results during grading. So, make sure they do produce the results you reported.

5. Grading Policy

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

If your program (listed in the appendix) fails to compile, you will get zero point for the exercise.

If your program's outputs do not match what you reported, you will get zero point for the exercise.