Accelerometers

Overview:

In this lab, you will apply your knowledge of interrupts to an accelerometer application. By the end of this lab, you will be able to detect a simple accelerometer gesture using analog-to-digital (ADC) converter interrupts.

Gesture Detection with an Accelerometer

In this section of the lab, you will use the accelerometer to detect when your microcontroller flips from face-down to face up. Simple gesture recognition is prevalent in many modern-day applications. In fact, if you have a recent iPhone model you will notice that the screen turns on when you pick up the phone and turn it towards your face. This is likely implemented using an onboard accelerometer or gyroscope. In order to obtain accelerometer readings on your microprocessor, you need to attach the Educational BoosterPack MKII.

Attaching the EDUMKII BoosterPack:

This section will walk you through properly attaching the BoosterPack to the MSP432P401R Launchpad.

1. Unbox the BoosterPack and remove it from its protective anti-static storage bag.
2. On the BoosterPack there are rows of connectors, two on each side of the LCD screen as shown in Figure 1.

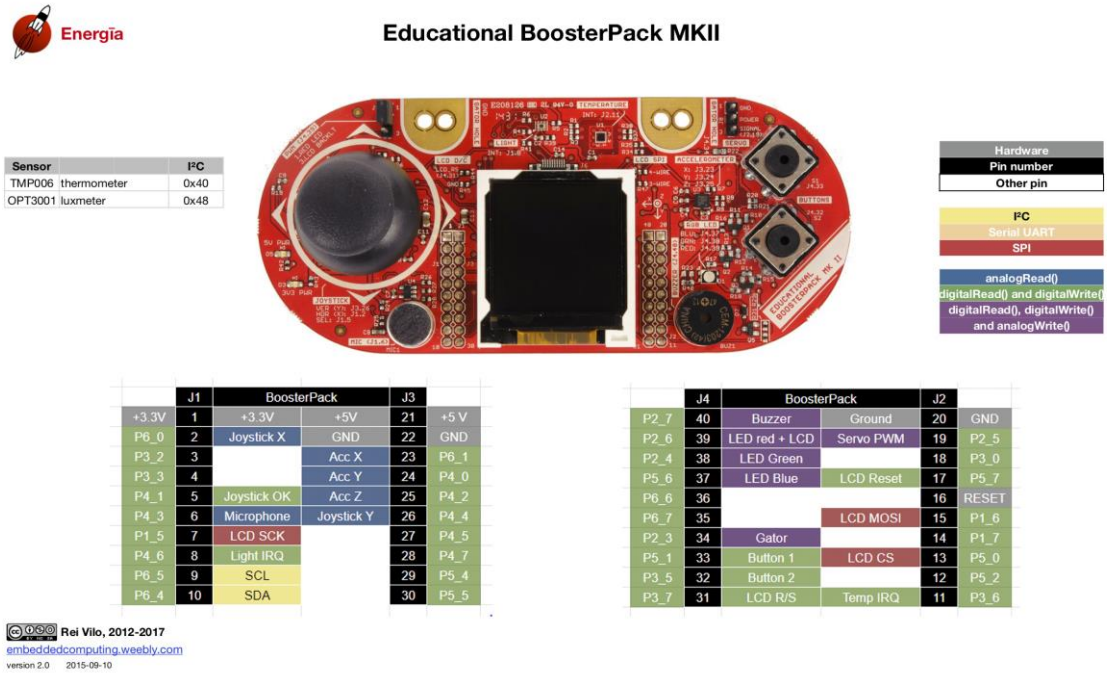

Figure 4: Pinout Diagram for the EDUMKII BoosterPack showing the jumper numbers as well as the corresponding port/pin numbers on the MSP432P401R.

3. The BoosterPack has the same number of connectors as the MSP432P401R Launchpad. The black pin numbers are the same as those on the MSP432P401R board provided in previous labs. The female connectors on the back side of the BoosterPack should align perfectly with the male pins on the Launchpad.

4. Make sure the Launchpad has the micro-usb connector facing forwards and the BoosterPack has the joystick on the left as shown in Figure 2. While the Launchpad **is not connected** to power, gently fit the Launchpad's pins into the BoosterPack's rear connectors so that the BoosterPack sits atop the Launchpad with all their connector pins perfectly aligned. None of the in-line connector pins should be left free on the Launchpad.

5. The Launchpad and BoosterPack should now be properly connected.

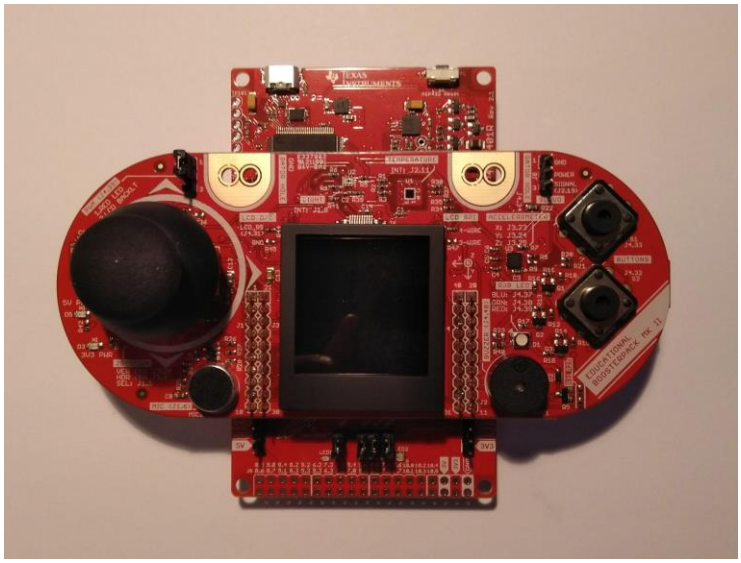

Figure 5: EDUMKII BoosterPack sitting atop the MSP432P401R Launchpad.

Project Initialization

Create a new sketch in Energia and save it to your Cosmos Workspace.

As before include the "stdio.h" and "driverlib.h" header files at the top of your sketch. Additionally, include the other libraries in figure 4. These contain helpful functions for working with the LCD screen on the EDUMKII.

```
#include <stdio.h>
#include <driverlib.h>

/* LCD Screen libraries*/
#include <LCD_screen.h>
#include <LCD_screen_font.h>
#include <LCD_utilities.h>
#include <Screen_HX8353E.h>
#include <Terminal12e.h>
#include <Terminal6e.h>
#include <Terminal8e.h>
Screen_HX8353E myScreen;
```

Figure 4: Library Includes for Part 2

*Note: The LCD screen libraries are written in C++ which is object oriented. Consequently, the purpose of the last line of code in figure 4 is to declare a new object, "myScreen." The source code of the libraries above specify the attributes of "myScreen." In this way, we can do useful things to the LCD through the following syntax, "myScreen.<Attribute>(parameters)."
In this part of the lab, we will modify the LCD screen's text through an attribute.

Next, we will need to create a few functions to get us started with both the accelerometer and the LCD interface. In the following steps, make sure to carefully copy the codes as they are in the figure.

1. Create the LCD initialization function.

```
void LCD_init(){

  myScreen.begin();

  // Find the LCD Screen Size
  Serial.print("X Screen Size: ");
  Serial.println(myScreen.screenSizeX());
  Serial.print("Y Screen Size: ");
  Serial.println(myScreen.screenSizeY());

  /* Let's make a title*/
  myScreen.gText(8, 20, "Accelerometer Data");

}
```

Figure 6: LCD Initialization Function

The LCD_init function initializes the LCD screen with the "myScreen.begin()" function. The following lines of code are used to print the LCD screen size (in pixels) to the serial monitor and create a title that will never change in our control loop.

2. Create the ADC initialization function.

```
void ADC_init(){
  /* Configures Pin 4.0, 4.2, and 6.1 ad ADC inputs */
  // ACC Z = P4.2
  // ACC Y = P4.0
  // ACC X = P6.1
  GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P4, GPIO_PIN0 | GPIO_PIN2, GPIO_TERTIARY_MODULE_FUNCTION);
  GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P6, GPIO_PIN1, GPIO_TERTIARY_MODULE_FUNCTION);

  ADC14_registerInterrupt(ADC14_IRQHandler);

  /* Initializing ADC (ADCOSC/64/8) */
  ADC14_enableModule();
  ADC14_initModule(ADC_CLOCKSOURCE_ADCOSC, ADC_PREDIVIDER_64, ADC_DIVIDER_8, 0);

  /* Configuring ADC Memory (ADC_MEM0 - ADC_MEM2 (A11, A13, A14)  with no repeat)
   * with internal 2.5v reference */
  ADC14_configureMultiSequenceMode(ADC_MEM0, ADC_MEM2, true);
  ADC14_configureConversionMemory(ADC_MEM0, ADC_VREFPOS_AVCC_VREFNEG_VSS, ADC_INPUT_A14, ADC_NONDIFFERENTIAL_INPUTS);

  ADC14_configureConversionMemory(ADC_MEM1, ADC_VREFPOS_AVCC_VREFNEG_VSS, ADC_INPUT_A13, ADC_NONDIFFERENTIAL_INPUTS);

  ADC14_configureConversionMemory(ADC_MEM2, ADC_VREFPOS_AVCC_VREFNEG_VSS, ADC_INPUT_A11, ADC_NONDIFFERENTIAL_INPUTS);

  /* Enabling the interrupt when a conversion on channel 2 (end of sequence)
   *  is complete and enabling conversions */
  ADC14_enableInterrupt(ADC_INT2);

  /* Enabling Interrupts */
  Interrupt_enableInterrupt(INT_ADC14);
  Interrupt_enableMaster();

  /* Setting up the sample timer to automatically step through the sequence
   * convert.*/
  ADC14_enableSampleTimer(ADC_AUTOMATIC_ITERATION);

  /* Triggering the start of the sample */
  ADC14_enableConversion();
  ADC14_toggleConversionTrigger();
}
```

Figure 7: ADC Initialization Function

An analog-to-digital converter (ADC) is a special type of hardware in microprocessors that convert a “real world” signal into binary. In short, the ADCs used here take data from a 3-axis accelerometer and convert the data to some unsigned binary representation. As an embedded system engineer, we need to take this binary data and so something meaningful with it.

Take a second to look through the ADC_init function and get a feel for how it works.

- GPIO_setAsPeripheralModuleFunctionInputPin converts the GPIOs that correspond to the accelerometer x, y, and z pins to ADC inputs.
- ADC14_configureConversionMemory sets aside three memory registers to store the inputs from the ADC.
- AD14_enableInterrupt and Interrupt_enableInterrupt enable two interrupts, one to trigger at the end of data sampling and the other to trigger the start of data sampling, respectively.

- AD14_toggleConversionTrigger triggers the ADC to collect our first accelerometer data.

3. Create ADC Interrupt Function.

```
void drawAccelData(void){
  myScreen.gText(40, 50, "X: " + String(resultsBuffer[0]));
  myScreen.gText(40, 70, "Y: " + String(resultsBuffer[1]));
  myScreen.gText(40, 90, "Z: " + String(resultsBuffer[2]));
}

void ADC14_IRQHandler(void)
{
    uint64_t status;

    status = MAP_ADC14_getEnabledInterruptStatus();
    MAP_ADC14_clearInterruptFlag(status);

    /* ADC_MEM2 conversion completed */
    if(status & ADC_INT2)
    {
        /* Store ADC14 conversion results */
        resultsBuffer[0] = ADC14_getResult(ADC_MEM0);
        resultsBuffer[1] = ADC14_getResult(ADC_MEM1);
        resultsBuffer[2] = ADC14_getResult(ADC_MEM2);

        /*
         * Draw accelerometer data on display
         */

        drawAccelData();
    }
}
```

Figure 8: ADC Interrupt and Data Display Functions

These last two functions collect ADC data using the ADC interrupt handler and redraw the data onto the LCD screen. Like the SysTick interrupt, the ADC14_IRQHandler does not need to be called because it is triggered by the hardware itself. Note that you must create the volatile unsigned 16-bit integer array “resultsBuffer[3]” outside of your main function. Like the variables modified in SysTick, this array must be **volatile**.

4. Lastly, create your setup function.

```
void setup() {

  WDT_A_hold(WDT_A_BASE);

  Serial.begin(115200);

  LCD_init();
  ADC_init();
}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Figure 9: Main Function

This function simply initializes the serial terminal and calls the functions we just created in the previous steps.

Project Description

You will now implement a simple gesture recognition algorithm using a simple state machine. First run the accelerometer code developed above. You should see raw unsigned integers displayed on the LCD screen. Twist your microprocessor so that you can start to understand which degree of rotation corresponds to which axis. Additionally, find the range of data displayed for each axis so that you can find a reasonable threshold. In other words, find the largest/smallest unsigned integer displayed on the LCD screen for each axis.

The gesture we want to detect is designed as follows.

1. Place the sensor such that,
   The z-axis value is below a specific threshold for at least time **tau** milliseconds. In terms of physical orientation, this corresponds to the LaunchPad being “upside down” for at least “tau” milliseconds.
2. Within the next time, **tau** milliseconds, the sensor must be reoriented such that,
   The z-axis value is above a specific threshold for at least **tau** milliseconds. In terms of physical orientation, this corresponds to the LaunchPad being “right side up” for at least “tau” milliseconds.

The state machine for this simple gesture is illustrated below.

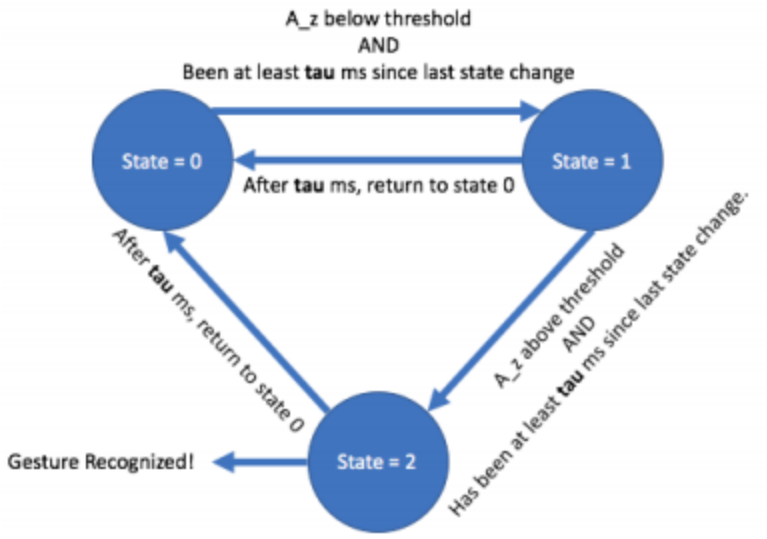

Figure 10: Gesture Recognition State Machine

The entirety of the gesture recognition logic should be implemented within the ADC14_IRQHandler function. With that said, the logic should be quick and concise (only using if-else statements). The time threshold tau can be implemented by incrementing a counter within the ADC interrupt (each interrupt iteration is ~ 1ms).

Since you cannot print within an interrupt, it might be helpful to encode your states with LaunchPad's LED. This is useful for creating a visual of "location" in your code at various circumstances.

Finally, increment a counter that is displayed on your LCD every time a gesture is recognized. Simply, this can be done by adding an additional gText function to the drawAccelData function.

```
void ADC14_IRQHandler(void)
{
    uint64_t status;

    status = MAP_ADC14_getEnabledInterruptStatus();
    MAP_ADC14_clearInterruptFlag(status);

    /* ADC_MEM2 conversion completed */
    if(status & ADC_INT2)
    {
        /* Store ADC14 conversion results */
        resultsBuffer[0] = ADC14_getResult(ADC_MEM0);
        resultsBuffer[1] = ADC14_getResult(ADC_MEM1);
        resultsBuffer[2] = ADC14_getResult(ADC_MEM2);

        /*
         * Draw accelerometer data on display
         */

        drawAccelData();


        /*
         * Implement Gesture Detection State Machine Here!
         */


    }
}
```

Figure 11: Implementing State Machine Logic