ECE2534 Fall 2012: Lab 3
Using and Calibrating the Accelerometer

This lab is an individual assignment

When you have completed this Lab, you will have experience with the following Microcontroller Programming concepts.

- Drawing graphics on the OLED Display;
- Accessing peripherals with SPI and I2C protocols;
- Accessing and interpreting accelerometer data from the PmodACL module;
- Processing Change-Notification interrupts from microcontroller ports;
- Calibrating the accelerometer on the PmodACL module.

The specification of the lab, when it is completely finished, is as follows. Upon applying power, the OLED display shows a symbol and three 16-bit values, X, Y and Z.

The X, Y and Z values on the OLED display are continuously updated with the values read from the accelerometer. If the board lies flat, the X and Y values will be close to zero (no acceleration) while the Z value will be close to 0x100 (1-G acceleration). If you hold the board on its side, you will see that the Z-acceleration reduces while the X- or Y-acceleration increases to + or - 1G.

The symbol indicates the orientation of the board, and can have one of six shapes: and up-arrow, a down-arrow, a left-arrow, a right-arrow, an empty square, and a crossed square. The symbol will always point which way is 'up'. If you lie the board flat (default position), the empty square is shown. If you flip the board over, so that the display faces down, the crossed square is shown. If you hold the board so that it rests on a side edge (4 possibilities), then either the left, right, up or down arrow will display, depending on what direction is 'up'.

The accelerometer chip contains additional features, such as tap detection and a calibration procedure. Your implementation will support accelerometer calibration as follows. Tapping (gently) in the X-direction of the Pmod accelerometer initiates calibration. After a 5-second delay, the acceleration values are read 10 times over a time period of one second. By averaging these values, offset factors for X, Y and Z can be identified, and programmed into the accelerometer chip. The offset calculation procedure is described in the datasheet of the accelerometer.

Step 1: Hardware Setup and Documentation

In order to complete this lab, you will need a Digilent Cerebot MX7cK board, a Digilent PmodOLED (Organic LED Display) peripheral board, and a Digilent PmodACL (Accelerometer) peripheral board. You need to configure the hardware for the lab as follows.

- The Cerebot board must be configured in standard debug mode, with a USB cable connected to the DEBUG port and the POWER jumper in the DEBUG position.
- The PmodOLED module must be fitted into Port JD of the Cerebot. Make sure that the jumper for port JD on the Cerebot board is configured in the 3V3 mode.
- The PmodACL module must be fitted into Port JC of the Cerebot. Make sure that the jumper for port JC on the Cerebot board is configured in the 3V3 mode.
- The I2C port on the PmodACL module must be connected, using a 6-pin Pmod cable, to header J7 on the Cerebot board. This header carries the connections for I2C Port 1 of the PIC32. Make sure to connect the pins in proper order: they are marked on the PmodACL as well as on the Cerebot board.

It will also be helpful to have the following documentation available. These documents can be found on Scholar (under Resources).

- Datasheet for the accelerometer chip on the PmodACL module (ADXL345);
- Schematics and reference manual for the PmodACL module;
- Reference manual for the Cerebot MX7 board;
- The Microchip PIC32 PLIB reference manual.

The lab includes one intermediate validation step and a final validation step. Both validations will be completed in the CEL with the help of a TA.

Step 2: Graphics Utility Routines

First, develop the following graphics utility routines to help you display the data extraced from the accelerometer. The display area will be organized as follows. The area from pixel coordinate (0,0) to (79,31) will be used for graphics (arrow display). The area from pixel coordinate (80,0) to (127,31) will be used for text (XYZ display).

To program the OLED display, you will make use of the same graphics library you've used in Lab 1. You can specifically refer to the functions declared in `OledChar.h` and `OledGrph.h` for basic text display and graphics display functions.

Use the functions of the OLED library to develop the following software functions. Write a test program that helps you to check these functions.

- `void ClearGraphArea()`: Clear the OLED area from pixel (0,0) to pixel (79,31).
- `void DrawRightArrow()`: Draws a right-pointing arrow in the graphics area. The exact size and shape of the arrow can be chosen by you, as long as it's constructed from functions in `OledGrph.h`.
- `void DrawLeftArrow()`: Draws a left-pointing arrow in the graphics area. The exact size and shape of the arrow can be chosen by you, as long as it's constructed from functions in `OledGrph.h`.
- `void DrawUpArrow()`: Draws an upward-pointing arrow in the graphics area. The exact size and shape of the arrow can be chosen by you, as long as it's constructed from functions in `OledGrph.h`.
- `void DrawDownArrow()`: Draws a downward-pointing arrow in the graphics area. The exact size and shape of the arrow can be chosen by you, as long as it's constructed from functions in `OledGrph.h`.
- `void DrawToArrow()`: Draws an empty square (an arrow pointing towards you) in the middle of the graphics area. The exact size of the square can be chosen by you, as long as it's constructed from functions in `OledGrph.h`.
- `void DrawFromArrow()`: Draws a square with a cross (X) (an arrow pointing away from you) in the middle of the graphics area. The exact size of the square can be chosen by you, as long as it's constructed from functions in `OledGrph.h`.
- `DrawXYZ(unsigned x, unsigned y, unsigned z)`: Draws the labels X, Y and Z on the first three lines of the display in the text area (roughly, starting at character column 10 and extending over 4 lines). The lower halfword of X, Y and Z should be displayed. For example, if X, Y, and Z have the values 0x25, 0x99 and 0xFFFE respectively, the display should show:

```
X   25
Y   99
Z FFFE
```

- `DrawArrow(x, y, z)`: Find the arrow corresponding to the largest value of X, Y and Z. To compare X, Y, Z, treat them as 16-bit two's complement values, and compare their absolute value. Find the largest, as well as the sign of the largest value. If X is the largest and X is positive, draw a right-pointing arrow. If X is the largest and X is negative, draw a left-pointing arrow. If Y is the largest and Y is positive, draw an up-pointing arrow. If Y is the largest and Y is negative, draw a down-pointing arrow. If Z is the largest and Z is positive, draw an to-pointing arrow (empty square). If Z is the largest and Z is negative, draw a from-pointing arrow (crossed square).

Test these routines in a small program; develop that program as a separate MPLabX project if needed. Don't forget to properly initialize the OLED before displaying functions. For example:

```
#include "PmodOLED.h"
#include "OledChar.h"
#include "OledGrph.h"
#include "delay.h"
#include "arrow.h"   // contains your functions

int main() {
    DelayInit();
    OledInit();

    while (1) {
        DrawXYZ(100,       0,      0); // right arrow
        DelayMS(1000);
        DrawXYZ(0xFF00,    0,      0); // left arrow
        DelayMS(1000);
        DrawXYZ(0,       100,      0); // up arrow
        DelayMS(1000);
        DrawXYZ(0,    0xFF00,      0); // down arrow
        DelayMS(1000);
        DrawXYZ(0,         0,    100); // to arrow
        DelayMS(1000);
        DrawXYZ(0,         0, 0xFF00); // from arrow
        DelayMS(1000);
    }
}
```

Step 3: I2C Access Routines

In the second section, you will develop I2C communication routines for the accelerator chip. Before you start writing software, you need to read up a little on the use of the I2C protocol in the accelerator chip, and how I2C is integrated in the PIC32 on the Cerebot.

Consult the following document sections and make notes of everything important you see[1]:

- Datasheet of ADXL345, page 18 (I2C protocol). Observe that you need to pull two pins high on the chip: the Chip Select (CSbar) and the ALT ADDRESS pin. Pay special attention to Figure 41, this figure is key to programming the chip correctly. This figure shows sequences for writing into and reading from the chip, in a single-byte sequence as well as in a multi-byte sequence.

- The schematic of the Pmod ACL module. Locate all pins that are important for this project. Besides the I2C pins on J2, there are several pins on J1 important: Chip Select (CSBar), ALT ADDRESS (ADDR), and interrupt (INT1).

- The Cerebot Schematic. The Pmod ACL is connected to port JC. Find out how the CSBar, ADDR and INT1 map onto pins of JC. Furthermore, the Pmod ACL will be controlled through I2C port 1 of the PIC32. Page 6 of the Cerebot Schematics shows the header (J7) which carries the signals of this port.

- Chapter 15 (I2C) of the PLIB reference. This reference lists all the key functions to work with I2C communications. In particular, it's possible to write the programming sequences of Figure 41 in the ADXL345 datasheet using only the following functions. Make sure you understand how to use these functions.

```
StartI2C1();
StopI2C1();
IdleI2C1();
MasterWriteI2C1(v);
v = MasterReadI2C1();
NotAckI2C1();
AckI2C1();
```

In the first step, your objective is to initialize the accelerator chip and to establish I2C communications with it. Develop the following C functions. Make use of the PLIB I2C and port programming functions to implement them.

[1]It maybe tempting to skip this upfront study and start hacking right away. Resist this urge and read the documentation carefully! Every minute of concentration invested now will earn you 10 back later.

- `void ACLInit()`: This function initializes I2C communications the ADXL345 chip. It makes sure that the chip will use I2C communications (by programming CSbar pin and ALT ADDRESS pin though a PIC32 parallel port), and it opens i2C port 1 as a master port. Set the baudrate to 50KHz or higher.

- `unsigned char ReadACLByte(unsigned char regadr)`: This function reads one byte from the ADXL345 chip. It implements the protocol shown in Figure 41 of the ADXL345 data sheet. In a nutshell, the protocol is as follows: (a) the PIC32 sends an I2C start, (b) the PIC32 sends an I2C write command to the I2C address of the ADXL345 chip, (c) the PIC32 writes the register address `regaddr` to the ADXL345, (d) the PIC32 sends an I2C restart, (e) The PIC32 sends an I2C read command from the I2C address the the ADXL345 chip, (f) the PIC32 reads the return byte value from the I2C address, (g) the PIC32 sends a I2C notack to terminate the read sequence. All of this can be implemented with the I2C PLIB commands shown above. Make sure to add an `IdleI2C()` after every I2C master activity on the PIC32. This is needed because the I2C peripheral is asynchronous, and needs time to complete I2C commands such as `MasterWriteI2C1(v)`.

- `void ReadACLMultiByte(unsigned char regadr, unsigned count, unsigned char *b)`: This function reads multiple bytes from the ADXL345 chip. It implements the protocol shown in Figure 41 of the ADXL345 data sheet. It is quite similar to `ReadACLByte`.

- `void WriteACLByte(unsigned char regadr, unsigned char value)`: This function reads multiple bytes from the ADXL345 chip. It implements the protocol shown in Figure 41 of the ADXL345 data sheet. It is quite similar to `ReadACLByte`.

- `void SetACLMode(unsigned char m)`: This function sets the resolution byte on the ADXL345 chip (ADXL345 register address 0x31). Select a resolution of +-2g at the default 10-bit resolution.

- `void ACLEnable()`: This function enables the ADXL345 chip (ADXL345 register address 0x2D). Essentially, this enables the measurement mode (measure bit).

- `unsigned char ReadACLDeviceId()`: This function returns the device ID of the ADXL345 chip. It is implemented by reading from ADXL345 register address 0x0.

- `void ReadACLData(unsigned *x, unsigned *y, unsigned *z)`: This function performs a multibyte read from ADXL345 register address 0x32. Reading 6 consecutive bytes will returns 2 bytes for X, 2 bytes for Y, 2 bytes for Z. You can paste these bytes together two-by-two to form three 16-bit values. Return the values in the arguments of this function.

The quickest way to test if you have the I2C communications correct, is the read the device ID from the ADXL345 chip. You can display the ADXL345 chip on the OLED display, or

show the lower nibble on the LEDs. The full test will come during implementation of Step 4, which integrates the work you did in Step 2 and Step 3.

Step 4: Basic Direction Finder and Validation

You will now use the functions you've developed for Step 2 and Step 3 and create the following direction finder. You will repeatedly read X, Y and Z acceleration values from the ADXL345 and display them on the OLED display. You will also use this information to find which way is 'up' (away from planet earth), and display a corresponding arrow on the OLED display.

When this program is complete, take it to a CEL TA for validation. Validate this step *at the latest* on Thursday 1 November 10:00PM. You don't have to turn in your validation sheet until the start of class on Thursday 8 November. CEL TAs will give priority to validations on Thursday 1 November.

Step 5: Tap Detection

Carefully read the section on Tap Detection in the ADXL345 datasheet, starting on page 28. A tap is a sudden, short acceleration, observable as an acceleration pulse in the X, Y, or Z direction. The ADXL 345 has dedicated hardware to do single-tap and double-tap detection.

You will implement a single-tap detection routine for taps in the X direction. These are taps that will hit the side of the Pmod ACL module toward to Cerebot board. Note that gentle taps are generally sufficient to ensure detection. Limit your force! If it doesn't work, it's more likely to be an issue with the software rather than an insensitive chip.

The ADXL345 uses a specific register for tap axis selection (register 0x2A). The register needs to be programmed as part of the initializing interrupts on the ADXL345, the routine `ConfigureACLInt()` as specified below.

Tap detection needs to be done by means of an interrupt. The ADXL345 has two interrupt pins, INT1 and INT2. Configure INT1 to issue an interrupt when a single-tap is detected. This needs to be done by programming the interrupt map (register 0x2F) and the interrupt enable (register 0x2E) on the ADXL345. Once an interrupt is generated, it needs to be cleared by reading the interrupt source (register 0x30) on the ADXL345. Thus, this operation must be included inside of the interrupt routine that will be called as a result of a single-tap detection.

Selecting the single-tap sensitivity requires you to choose a duration and a threshold parameter. Consult the ADXL345 datasheet; they contain suggested values that work fine for the PmodACL module as attached to the Cerebot board. Program the tap duration and tap threshold values in the TRESH_TAP (register address 0x1D) and DUR (register address 0x21) registers on the ADXL345.

The INT1 pin of the ADXL345 is connected to port JC on the Cerebot board. You can verify in the Cerebot board schematics that the corresponding pin on port JC can function as a Change-Notification (CN) pin, namely CN14. A change-notification pin is used by the PIC32 to generate external interrupts. This requires proper configuration of the PIC32. First, you need to configure the port pin that will be used by CN14 as a digital input (`PORTSetPinsDigitalIn()`. Next, you need to enable interrupts from Change Notification pins. In PLIB, there's a function `mCNOpen` to do for this. You don't need to make use of pull-up resistors; the ADXL345 is properly driving INT1.

Finally, you need to write the interrupt service routine. The interrupt service routine needs to implement the following. First, it checks if the interrupt source is a Change Notification (`mCNIntGetFlag()`). Next, it clears the interrupt condition on the ADXL345. Next, it clears the interrupt condition of the Change Notification (`mCNClearIntFlag()`). Finally, it initiates actual interrupt processing. While developing the interrupt service routine, you can implement a trivial interrupt service activity, such as toggling the status of a LED. This will help you to verify that the interrupt service routine works before developing further functionality on top of it.

In summary, in Step 5, you need to implement an interrupt service routine for single-tap detection. Implement the following functions.

- The interrupt service routine. You can implement a simple Single-vector interrupt routine mapped to priority level 1:

```
#pragma interrupt InterruptHandler ipl1 vector 0
void InterruptHandler( void) {
   ...
}

int main() {
   ...
   INTEnableSystemSingleVectoredInt();
   ...
}
```

- `void ConfigureACLTap(unsigned char treshold, unsigned char duration)`: This function configures the threshold and duration parameters for a single-tap detection.
- `void ConfigureACLInt()`: This function configures the interrupt map and interrupt source on the ADXL345. It also sets up the ADXL345 to detect X-axis taps, while ignoring those in the Y-axis or Z-axis direction.
- `unsigned char ReadACLIntSource()`: This function reads the interrupt source from the ADXL345.
- `void ConfigureCNInt()`: This function enables interrupt from change-notification pin 14, and configures them at priority level 1 (so that the proper interrupt handler will be called).

Step 6: Calibration

In the final step, you will use the tap detection interrupt to calibrate the ADXL345 sensor. First, read the section on offset calibration in the ADXL345 data sheet. The idea of calibration is as follows. Ideally, when the board is laying flat on a desk, we'd expect to see the values 0, 0, and 0x100 for the acceleration values in the X, Y, and Z direction respectively. However, due to manufacturing imperfections, you will notice this is not the case. The 'default-zero' value for X and Y will be different from 0, and the 'default-1G' value for Z will be different from 0x100.

The ADXL345 contains three 8-bit registers, OFSX, OFSY and OFSZ (at register address 0x1E, 0x1F and 0x20 respectively) that can be programmed with correction values. The datasheet clarifies how to define the correction values.

The objective of step 6 is to implement a calibration procedure as follows.

- After detecting a single tap, the application goes into a calibration mode.
- The first step in calibration is a delay of 5 seconds. This delay is required to let the ADXL345 settle after a single-tap event. During these 5 seconds, the LEDs on the Cerebot should turn on and off with a period of half a second.
- After the 5 second delay, the board will capture 10 X,Y,Z values from the ADXL345 with a 100ms delay. Thus, the board will take 10 readings from the ADXL345 in one second.
- Next, the PIC32 will average the 10 X, Y, Z values in order to find an average X, and average Y and an average Z.
- The correction factors for OFSX, OFSY and OFSZ can now be computed as follows. The correction factor for OFSX is -AX/4, with AX the averaged X reading. The correction factor for OFSY is -AY/4, with AY the averaged Y reading. The correction factor for OFSZ is -(AZ - 0x100)/4, with AZ the averaged Z reading.
- After the OFSX, OFSY, and OFSZ registers have been programmed, the board returns to normal operation mode (as described in Step 4)

When this program is complete, take it to a CEL TA for validation. Validate this step *at the latest* on Thursday 8 November 10:00PM (Sunday 11 November 10:00PM, 25% penalty). Turn in your validation sheet (for Step 4 and Step6) at the next class meeting.

Step 7: Submit your code on Scholar

Create a ZIP archive of all the source code developed for Lab 3. Include all .c and .h files. Submit it on Scholar before 10:00PM on Friday 9 November. The ZIP archive should be named

`Lastname_Firstname_Lab_3.zip`