ECE 2534 Fall 2015 – Homework 6 (40 points) **Plug-n-play accelerometer**

The objectives of this homework are:

1. Modify vendor-supplied driver code to use a peripheral in an application. This is a realistic scenario.
2. Avoid hard-coding the ports and pins connected to a peripheral.
3. Structure driver and application code into layers, with each layer's code contained in separate files.

Write a routine that uses SPI to take measurements with the PmodACL accelerometer. These measurements need not be interrupt-driven. Your code should display readings from the accelerometer on the OLED as follows:

```
   MIN  CUR  MAX
X sddd sddd sddd
Y sddd sddd sddd
Z sddd sddd sddd
```

Current values should be read every 10 ms so that the minimum and maximums can be recomputed. However the display should only be updated every 500 ms. For each reading, `s` represents a sign (+ or -) and `d` represents a digit. The values should be displayed with leading zeroes and signs so that columns stay aligned. The accelerometer's data registers contain 10-bit signed values by default, which corresponds to a range of -512 to +511. The `sprintf()` function will be useful.

In order to move the PmodACL while the ChipKIT board is stationary, use the twelve-pin extension cable and 12-pin pin header. Be sure that the cable's pin marking (it looks like a three-toed footprint) is connected to pin 1 on `JE` or `JF` and pin 1 on the PmodACL.

The accelerometer should function correctly when plugged into either `JE` or the `JF` *without* changing your code. In other words, your code must auto-sense whether the PmodACL is connected to `JE` or `JF`. If the PmodACL is not connected, the message "`ACL not found`" should be displayed on the OLED. Be sure to power off your chipKit board before connecting or disconnecting any peripheral such as the PmodACL.

Like most chip vendors, Analog Devices provides microcontroller driver code for the ADXL345. A version of this code targets the PIC32:

https://wiki.analog.com/resources/tools-software/uc-drivers/microchip/adxl345

The code resides in GitHub repositories:

https://github.com/analogdevicesinc/no-OS/tree/master/drivers/ADXL345
https://github.com/analogdevicesinc/no-OS/tree/master/Microchip/PIC32MX320F128H/Common

`ADXL345.c` and `ADXL345.h` provide an API to configure and use the ADXL345. `Communication.c` and `Communication.h` include SPI and I2C drivers for the ADL345 and a UART driver for a console. You might think everything has been done for you, but the code does not meet the plug-n-play objective since the SPI channel and pin assignments are fixed. Review of the SPI code also reveals deficiencies such as:

1. Portability is reduced by the use of hardcoded registers rather than PLIB functions such as `SpiChnGetC()` and `SpiChnPutC()`.
2. `SPI_Read()` will overflow a temporary array and corrupt the stack if `bytesNumber` > 4.
3. The SPI API does not reflect the fact that SPI performs simultaneous reads and writes.

You should rewrite `Communication.c` and `Communication.h` so that:

1. The I2C and UART code is removed.
2. The SPI functions are replaced with:

```
int SpiMasterInit(int channel)
int SpiMasterIO(char bytes[], int numWriteBytes, int numReadBytes)
```

There is nothing to be gained from retaining any of the old `Spi_Init()`, `Spi_Write()`, or `Spi_Read()` code. New function interfaces are required, and you need to use PLIB function calls such as `SpiChnOpen()`, `mPORTxSetPinsDigitalOut()`, `mPORTxClearBits()`, `mPORTxSetBits()`, `SpiChnPutC()`, and `SpiChnGetC()`. Do not retain busy wait loops or direct writes and reads of PIC32 registers.

If `SpiMasterInit()` is called with the SPI channel number for connectors `JE` or `JF`, the function:

1. Configures the slave select pin on the connector as a digital output.
2. Calls the PLIB function `SpiChnOpen()` with the appropriate arguments to configure the channel as a SPI master, with clock polarity and edge options matching the ADXL345, and using byte transfers with enhanced buffering.
3. Returns a value of 0 to indicate success.

The slave select pin will be manipulated by `SpiMasterIO()` code since automatic assertion and deassertion of the slave select pin by the SPI controller can conflict with the ADXL345's multibyte transfer option. If the `SpiMasterInit()` function is not called with the SPI channel number for connectors `JE` and `JF`, an error value (-1) is returned.

The `SpiMasterIO()` function accepts a `bytes` array whose first `numWriteBytes` entries are bytes to be written from the `bytes` argument to the slave. The next `numReadBytes` entries are values to be read from the slave to the `bytes` argument. This single API suits handles Figure 37 (SPI 4-Wire Write) and Figure 38 (SPI 4-Wire Read) on page 16 of the ADXL345 data sheet, and the `ADXL345.c` code. SPI is inherently full duplex, although for each transfer either the byte read from the slave or the value written to the slave may be a don't care (`X`). A single I/O function better suits the SPI protocol and avoids code duplication.

The `SpiMasterIO()` function should return 0 if the SPI channel number is for connector `JE` or `JF`; otherwise an error code (-1) is returned. `SpiMasterIO()` needs to call PLIB I/O port macros (such as `mPORTxClearBits()`) to assert the slave select pin before starting the byte transfer, and deassert the slave select pin after finishing the transfer. The PLIB blocking functions `SpiChnPutC()` and `SpiChnGetC()` should be used to write and read bytes. If you can correctly implement the `SpiMasterInit()` and `SpiMasterIO()` functions, then you understand the SPI protocol.

After revising the SPI layer, the ADXL345 layer should replace the old SPI layer API with the new SPI layer API. This should be straightforward except for the `ADXL345_Init()` function, which needs to try communicating with the ADXL345 over the SPI channels. After calling `SpiMasterInit()` on a channel, it should attempt to read the ADXL345's Device ID register and see if it matches the expected value. This check should be done with the `ADXL345_GetRegisterValue()` function and using the constants defined in `ADXL345.h`. If an ADXL345 is not connected to the channel, the `SDO` (slave data out) signal will be floating (undefined). The `ADXL345_Init()` function should return the channel number which has the ADXL345 connected, or -1 if an ADXL345 is not detected.

The ADXL345 does not power up in an active state. The ADXL345 datasheet and the `main.c` code in the following GitHub repository should help you configure and calibrate the ADXL345:

https://github.com/analogdevicesinc/no-OS/tree/master/Microchip/PIC32MX320F128H/PmodACL

Your requirements may differ from the configuration used by this sample code. Look the `#include` directives used in the Analog Devices code to see what is needed when your code is split over several files.

Submit a zipped archive containing your complete source code. A grader should be able to unzip the archive, open the project, and run it as is. Use this naming convention for the archive that you submit:

<Last name>_<First name>_HW6.zip

Replace <Last name> and <First name> by your family name and given name respectively.