TI ARM Lab 7
Accelerometers

National
Science
Foundation

Funded in part, by a grant from the
National Science Foundation
DUE 1068182

Acknowledgements

Developed by Craig Kief, Brian Zufelt, and Jacy Bitsoie at the Configurable Space Microsystems Innovations & Applications Center (COSMIAC). Co-Developers are Bassam Matar from Chandler-Gilbert and Karl Henry from Drake State. *Funded by the National Science Foundation (NSF).*

Lab Summary

This lab introduces the concepts of the I^2C and how it is implemented on the ARM processor.

Lab Goal

The goal of this lab is to continue to build upon the skills learned from previous labs. This lab helps the student to continue to gain new skills and insight on the C code syntax and how it is used in the TI implementation of the ARM processor. Each of these labs will add upon the previous labs and it is the intention of the authors that students will build with each lab a better understanding of the ARM processor and basic C code, syntax and hardware. Even though these tutorials assume the student has not entered with a knowledge of C code, it is the desire that by the time the student completes the entire series of tutorials that they will have a sufficient knowledge of C code so as to be able to accomplish useful projects on the ARM processor.

Learning Objectives

The student should begin to become familiar with the concept of the accelerometer and ways to accomplish simple projects. An accelerometer is a device that will measure acceleration forces. These forces may be static, like the constant force of gravity pulling at your feet, or they could be dynamic - caused by moving or vibrating the accelerometer. By measuring the amount of static acceleration due to gravity, you can find out the angle the device is tilted at with respect to the earth. By sensing the amount of dynamic acceleration, you can analyze the way the device is moving. An accelerometer can help the designer understand the system better. Is it driving uphill? Is it going to fall over when it takes another step? Is it flying horizontally or is it dive bombing your professor? A designer can write code to answer all of these questions using the data provided by an accelerometer. In the computing world, IBM and Apple have recently started using accelerometers in their laptops to protect hard drives from damage. If you accidentally drop the laptop, the accelerometer detects the sudden free-fall, and switches the hard drive off so the heads don't crash on the platters. In a similar fashion, high g accelerometers are the industry standard way of detecting car crashes and deploying airbags at just the right time.

The ADXL345 is a small, thin, ultralow power, 3-axis accelerometer with high resolution (13-bit) measurement at up to ±16 *g*. Digital output data is formatted as 16-bit twos complement and is accessible through either a SPI (3- or 4-wire) or I2C digital interface. The ADXL345 is well suited for mobile device applications. It measures the static acceleration of gravity in tilt-sensing applications, as well as dynamic acceleration resulting from motion or shock. Its high resolution (3.9 m*g*/LSB) enables measurement of inclination changes less than 1.0°.

Grading Criteria

N/A

Time Required
Approximately one hour

Lab Preparation
It is highly recommended that the student read through this entire procedure once before actually using it as a tutorial. It is also recommended that the tutorial software was run first to preload compiler options and source files.

Equipment and Materials
It is assumed that the student has already completed prior labs and the software is installed properly.

Software needed	**Quantity**
Install the tutorial software from the autoinstaller located at http://www.cosmiac.org/Microcontrollers.html	1
Hardware needed	Quantity
The hardware required is the TI Stellaris LaunchPad Kit and the Digilent Orbit board	1

Additional References
Current TI ARM manuals found on TI web site: www.ti.com/lit/ug/spmu289a/spmu289a.pdf.

Lab Procedure: Install/Connect board to computer

Plug in the supplied USB cable to the top of the Evaluation Kit and Digilent Orbit board as shown in Figure 1. Ensure the switch on the board is set to "DEBUG" and not "DEVICE". It is assumed that the evaluation board is properly installed and operating. Launch the Code Composer software. The easiest way to find it, is to go to Start, All programs and then look in the area identified in Figure 2.

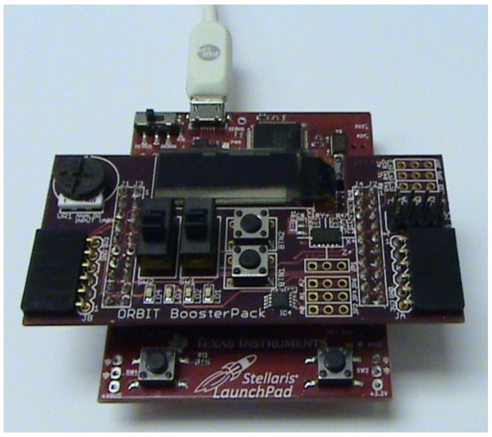


Figure 1.Stellaris and Orbit Combination

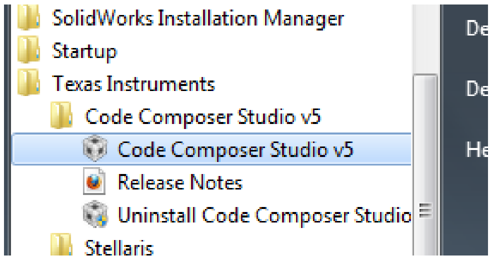
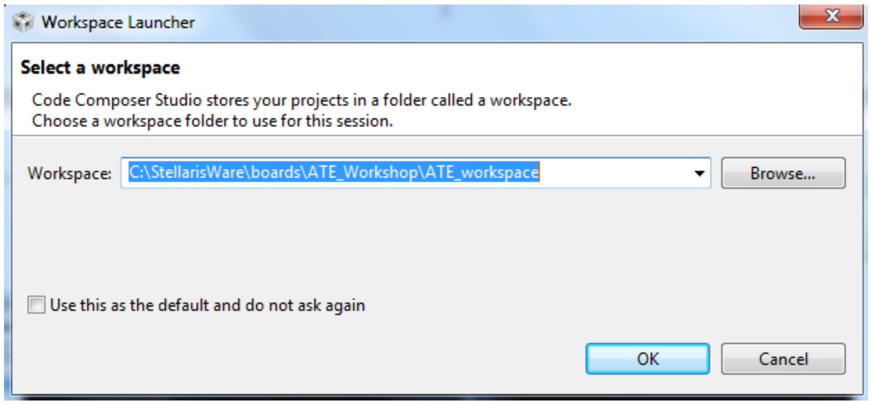


Figure 2.Code Composer



Workspace Launcher

Select a workspace

Code Composer Studio stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\StellarisWare\boards\ATE_Workshop\ATE_workspace

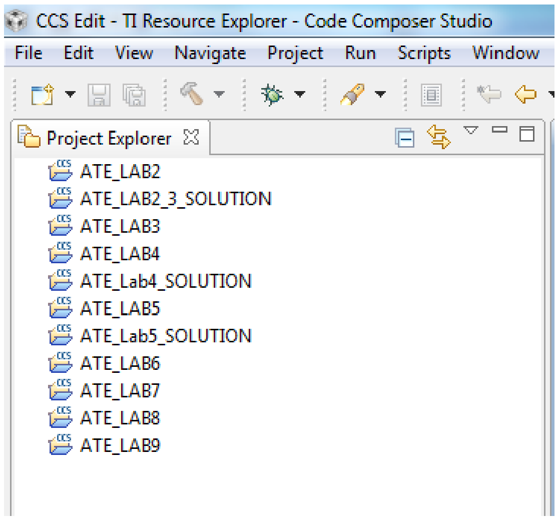
Browse...

☐ Use this as the default and do not ask again

OK Cancel

Figure 3.Workspace Launcher

The designer will be presented with the choices shown in Figure 3. The Code Composer wants to know where the workspace is located. If the auto installer was used to set up the lab files, it will automatically load up shells that were used for the workshop and these tutorials. In that case, use the path as shown in the picture in Figure 3. The student will know if they have used the auto installer program as they will be presented with a picture as shown in Figure 4 where the shells of all lab projects are installed. What is desired is to have the student step through the Lab_7 exercise. First however, read through this procedure so the student knows more about where they are going. Each tutorial/lab presents new information related to C code and to its use in programming the ARM processor.


CCS Edit - TI Resource Explorer - Code Composer Studio
File Edit View Navigate Project Run Scripts Window
Project Explorer

ATE_LAB2
ATE_LAB2_3_SOLUTION
ATE_LAB3
ATE_LAB4
ATE_Lab4_SOLUTION
ATE_LAB5
ATE_Lab5_SOLUTION
ATE_LAB6
ATE_LAB7
ATE_LAB8
ATE_LAB9

Figure 4. Shell of Projects

It is now important to spend a couple of minutes to explain a little more about the accelerometer used in this project. If you are looking on the Orbit board, there is a small chip in the upper right quadrant. It has a cross on it with a "X" and "Z" lettering. This is the ADXL345 chip.

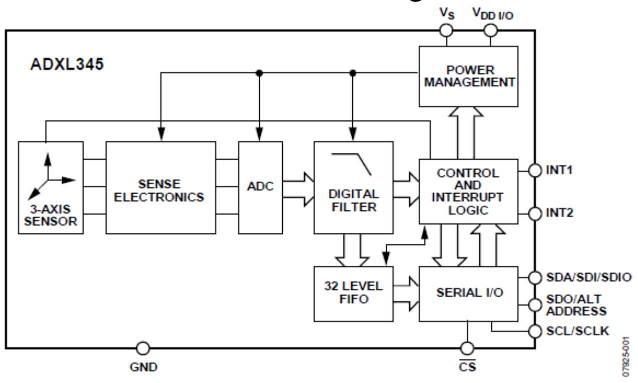


Figure 5. ADXL345

As can be seen in Figure 5, it has a 3-axis sensor that can be used to detect the orientation of the board. In this tutorial, only the X axis will be measured. This is to keep the tutorial as simple as possible. As a result, the function will have three returned values: -1 (going left), 0 (standing still), or 1 (going right). The chip is very sensitive. The system transfers data on an I^{2}C line. The ADXL345 has a I^{2}C output and the ARM processor has I^{2}C inputs as shown in Figure 6 from the datasheet.

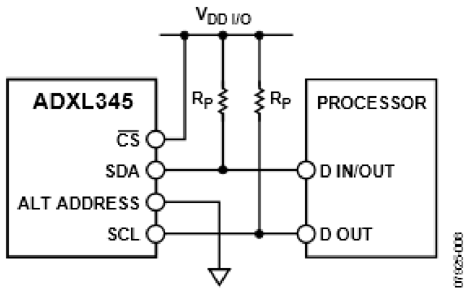


Figure 6. Connection Diagram

This protocol interface has a unique set of waveforms to allow the system to know when to start and stop the transfer of data. This waveform is shown in Figure 7 and is how the processor knows when to receive data.

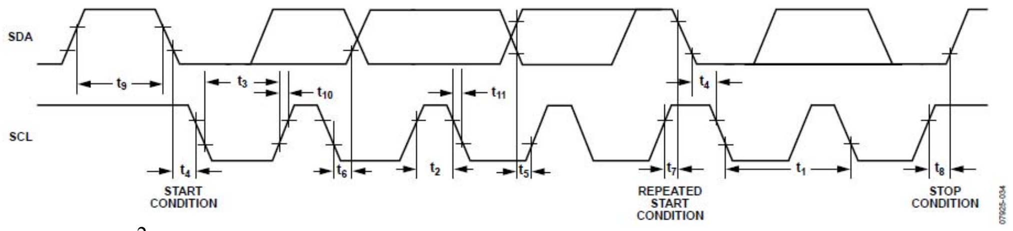
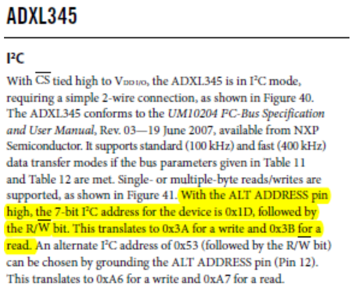


Figure 7. I^{2}C Waveform Timing Diagram

The entire code for this project is given at the end of this tutorial in attachment 1. This initially looks like a large amount of very complex code however, when you remember that much of the LED portion has been presented in previous labs it is less scary. Add to that the fact that another large portion of the code is nothing more than delays to slow the system down for human consumption and it is less scary. Some of the first code seen is shown below:

```
#define ACCEL_W 0x3A
#define ACCEL_R 0x3B
#define ACCEL_ADDR 0x1D
```

These three lines above are the definition parameters for the device. They are taken from the ADXL345 datasheet as shown below. It clearly defines the address as x1D, the read as x3B and the write as x3A.



ADXL345

I²C

With $\overline{CS}$ tied high to $V_{DD I/O}$, the ADXL345 is in I²C mode, requiring a simple 2-wire connection, as shown in Figure 40. The ADXL345 conforms to the *UM10204 I²C-Bus Specification and User Manual*, Rev. 03—19 June 2007, available from NXP Semiconductor. It supports standard (100 kHz) and fast (400 kHz) data transfer modes if the bus parameters given in Table 11 and Table 12 are met. Single- or multiple-byte reads/writes are supported, as shown in Figure 41. With the ALT ADDRESS pin high, the 7-bit I²C address for the device is 0x1D, followed by the R/$\overline{W}$ bit. This translates to 0x3A for a write and 0x3B for a read. An alternate I²C address of 0x53 (followed by the R/$\overline{W}$ bit) can be chosen by grounding the ALT ADDRESS pin (Pin 12). This translates to 0xA6 for a write and 0xA7 for a read.

Figure 8. Datasheet Excerpt

The next set of code shown below is a new topic. It is called a "function prototype." Most experienced designers like to have the main starting portion of their code appear at the top of the main.c file. The beginning of the program is normally identified with "**void main**(**void**) {". To make things cleaner, most designers will use the function prototypes to declare the functions at the top of the file and then further down in the program after the main function, will do the full explanation and declaration of what the functions actually do. In this case, there are two function prototypes. The first is to initialize the accelerometer and the second is to read the chip.

```
void Accel_int();       // Function to initialize the Accelerometer

signed int Accel_read();// Function to read the Accelerometer
```

The next portion of the code initializes the LEDs. This is very similar to what has been done in previous labs. All peripherals must be initialized before they can be used. The way this code is designed to operate is to have the LEDs fall or flow downhill based on the way the board is tilted. If the board is tilted 90 degrees to the left, the LEDs will fall down to the bottom. If the board is tilted the other direction, the LEDs will still attempt to fall to the bottom. After this section of the code, the two functions that were prototyped earlier are expanded upon.

Open the main.c file that is in the lab installer for Lab 7. When this is compared to the code in attachment 1, it is clear to see that the code for the LED waterfall is missing. The task of this tutorial is to type on this code. When done, debug/compile and download. Run the program and make sure the LEDs flow appropriately.

As a challenge exercise, reduce the time to flow the lights from one end to the other or light up the Stellaris board with different color LEDs when extremes are hit.

The start of the main function outlines how to setup a peripheral to be used. The I^{2}C controller is housed within a GPIO port. Thus setting up the Chip to use this peripheral is a three-step process.

1. Setup the core to enable the I^{2}C hardware. This is part of the SysCtl set of functions outline in the Driver users guide. The Driver users guide provides an outline of all the available functions found in the Stellaris API

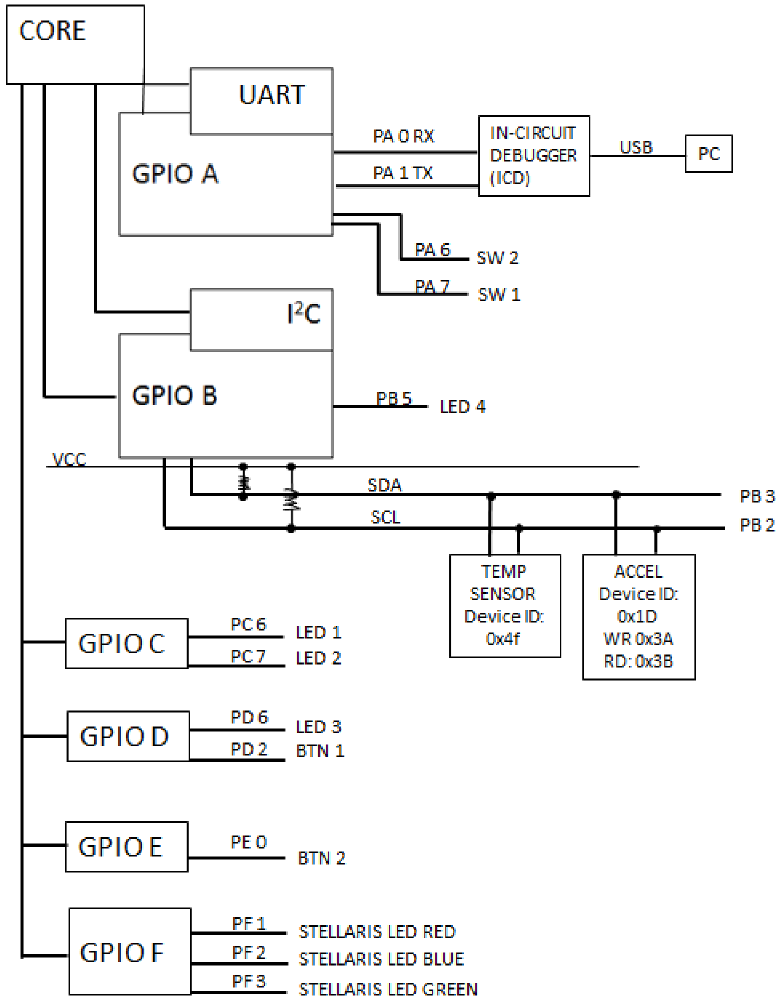
2. Setup the GPIO port to use the I^{2}C controller. This is part of the GPIO set of functions found within the Driver users guide.

3. Finally, Setup the I^{2}C controller with the settings for the bus you are plugging it into. Like the UART, I^{2}C has many settings and options. This is part of the I2C set of functions found within the Driver users guide.

Attachment 1: main.c file

```
/*****************************************************

Project : Orbit Lab 7 ATE (ACCEL)
Version : 1.0
Date    : 2/20/2013
Author  :BrianZufelt / CraigKief
Company : COSMIAC/UNM
Comments:
This source provides an introduction to the sensing capabilities
for embedded systems. The student will read accelerometer data
and toggle the proper LED for the application selected.


*****************************************************
Chip type           : ARM LM44F120H5QR
Program type        : Firmware
Core Clock frequency : 80.000000 MHz
*****************************************************/

#define ACCEL_W 0x3A
#define ACCEL_R 0x3B
#define ACCEL_ADDR 0x1D

// Define needed for pin_map.h
#define PART_LM4F120H5QR

#include"inc/hw_memmap.h"
#include"inc/hw_types.h"
#include"inc/hw_i2c.h"
#include"driverlib/gpio.h"
#include"driverlib/pin_map.h"
#include"driverlib/sysctl.h"
#include"driverlib/i2c.h"

voidAccel_int();                    // Function to initialize the Accelerometer

signedintAccel_read();// Function to read the Accelerometer

voidmain(void) {

        signedshortintLED_value = 1;

        SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);  //setup clock

        SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);                // Enable I2C hardware
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);    // Enable Pin hardware

        GPIOPinConfigure(GPIO_PB3_I2C0SDA);                        // Configure GPIO pin for I2C Data line
        GPIOPinConfigure(GPIO_PB2_I2C0SCL);                        // Configure GPIO Pin for I2C clock line

        GPIOPinTypeI2C(GPIO_PORTB_BASE, GPIO_PIN_2 | GPIO_PIN_3);  // Set Pin Type

        // Enable Peripheral ports for output
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);                           //PORTC
        GPIOPinTypeGPIOOutput(GPIO_PORTC_BASE, GPIO_PIN_6|GPIO_PIN_7);         // LED 1 LED 2

        GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_5);                    // LED 4

        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);                           //PORT D
        GPIOPinTypeGPIOOutput(GPIO_PORTD_BASE, GPIO_PIN_6);                    // LED 3

        //setup the I2C
        GPIOPadConfigSet(GPIO_PORTB_BASE, GPIO_PIN_2, GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD);        // SCL MUST BE STD
        GPIOPadConfigSet(GPIO_PORTB_BASE, GPIO_PIN_3, GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_OD_WPU);     // SDA IS OPEN DRAIN
        I2CMasterInitExpClk(I2C0_MASTER_BASE, SysCtlClockGet(), false);   // The False sets the controller to 100kHz communication
        Accel_int();                            // Function to initialize the Accelerometer

        while(1){

                // Fill in this section to read data from the Accelerometer and move the LEDs according to the X axis

                LED_value = LED_value + Accel_read();  //Accel_read only returns 1,0 or -1
                //if decrementing, going to the right on leds

                if(LED_value<= 1){
                        // Cycle through the LEDs on the Orbit board
                        GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6|GPIO_PIN_7, 0x40); // LED 1 on LED 2 Off
                        GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_6, 0x00);                    // LED 3 off, Note different PORT
                        GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, 0x00);                    // LED 4 off
                        LED_value = 1;         // reset value to maintain range
                }

                elseif(LED_value == 2){
                        // Cycle through the LEDs on the Orbit board
                        GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6|GPIO_PIN_7, 0x80); // LED 1 off LED 2 on
                        GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_6, 0x00); // LED 3 off, Note different PORT
                        GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, 0x00); // LED 4 on
                }
                elseif(LED_value == 3){
                        // Cycle through the LEDs on the Orbit board
                        GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6|GPIO_PIN_7, 0x00); // LED 1 off LED 2 off
                        GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_6, 0x40); // LED 3 on, Note different PORT
                        GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, 0x00); // LED 4 0ff
                }
                elseif(LED_value>= 4){
                        // Cycle through the LEDs on the Orbit board
                        GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6|GPIO_PIN_7, 0x00); // LED 1 off LED 2 Off
                        GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_6, 0x00); // LED 3 off, Note different PORT
                        GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, 0x20); // LED 4 on
                        LED_value = 4;         // reset value to maintain range
                }

        }
}

voidAccel_int(){                        // Function to initialize the Accelerometer

        I2CMasterSlaveAddrSet(I2C0_MASTER_BASE, ACCEL_ADDR, false);  // false means transmit
        I2CMasterControl(I2C0_MASTER_BASE, I2C_MASTER_CMD_BURST_SEND_START);          // Send Start condition
        I2CMasterDataPut(I2C0_MASTER_BASE, 0x2D);       // Writing to the Accel control reg
        SysCtlDelay(20000);                             // Delay for first transmission
        I2CMasterDataPut(I2C0_MASTER_BASE, 0x08);       // Send Value to control Register
        I2CMasterControl(I2C0_MASTER_BASE, I2C_MASTER_CMD_BURST_SEND_FINISH);         // Send Stop condition
        while(I2CMasterBusBusy(I2C0_MASTER_BASE)){};                // Wait for I2C controller to finish operations

}

signedintAccel_read() {// Function to read the Accelerometer

        signedint data;
        signedshort value = 0;                  // value of x

        unsignedchar MSB;
        unsignedchar LSB;

        I2CMasterSlaveAddrSet(I2C0_MASTER_BASE, ACCEL_ADDR, false);                         // false means transmit
        I2CMasterDataPut(I2C0_MASTER_BASE, 0x32);
        SysCtlDelay(20000);
        I2CMasterControl(I2C0_MASTER_BASE, I2C_MASTER_CMD_SINGLE_SEND);                     //Request LSB of X Axis
        SysCtlDelay(2000000);                                   // Delay for first transmission
        I2CMasterSlaveAddrSet(I2C0_MASTER_BASE, ACCEL_ADDR, true);                          // false means transmit
        I2CMasterControl(I2C0_MASTER_BASE, I2C_MASTER_CMD_SINGLE_RECEIVE);          //Request LSB of X Axis, read from the bus
        SysCtlDelay(20000);

        LSB = I2CMasterDataGet(I2C0_MASTER_BASE);
        SysCtlDelay(20000);

        I2CMasterSlaveAddrSet(I2C0_MASTER_BASE, ACCEL_ADDR, false);                         // false means transmit
        I2CMasterDataPut(I2C0_MASTER_BASE, 0x33);

        I2CMasterControl(I2C0_MASTER_BASE, I2C_MASTER_CMD_SINGLE_SEND);                      //Request LSB of X Axis
        SysCtlDelay(2000000);           // Delay for first transmission

        I2CMasterSlaveAddrSet(I2C0_MASTER_BASE, ACCEL_ADDR, true);                          // false means transmit

        I2CMasterControl(I2C0_MASTER_BASE, I2C_MASTER_CMD_SINGLE_RECEIVE);                   //Request LSB of X Axis
        SysCtlDelay(20000);

        MSB = I2CMasterDataGet(I2C0_MASTER_BASE);

        value = (MSB << 8 | LSB);  //bit shift 8 bits and or with LSB.  Value is 16 bits.

        if(value < -250 ){                                      // testing axis for value
                data = -1;
        }
        elseif (value > 250){
                data = 1;
        }

        else{
                data = 0;
        }

        SysCtlDelay(20000);

        return data;                                                // return value
}
```

Attachment 2: Block Diagram of the Pins Used in Projects


CORE
UART
GPIO A
PA 0 RX
PA 1 TX
IN-CIRCUIT DEBUGGER (ICD)
USB
PC
PA 6
SW 2
PA 7
SW 1
I²C
GPIO B
PB 5
LED 4
VCC
SDA
SCL
PB 3
PB 2
TEMP SENSOR Device ID: 0x4f
ACCEL Device ID: 0x1D WR 0x3A RD: 0x3B
GPIO C
PC 6
LED 1
PC 7
LED 2
GPIO D
PD 6
LED 3
PD 2
BTN 1
GPIO E
PE 0
BTN 2
GPIO F
PF 1
STELLARIS LED RED
PF 2
STELLARIS LED BLUE
PF 3
STELLARIS LED GREEN