Lab 5: Accelerometer Integration, Slope-directed Steering

Preparation

Reading

Lab Manual

Chapter 5 - LCD & Keypad and Analog Conversion (review)

Chapter 7 - Control Algorithms

LMS, Course Materials

Accelerometer Data Sheet

LCD and Keypad Data Sheet (review)

Objectives

General

1. Keep your working code from Lab 4 someplace safe, UNALTERED. A modified version of it will be used later in Lab 6 to control a gondola on a turntable. For this lab a 2-axis Accelerometer will provide the control feedback signal instead of the Ranger and Compass.
2. A modified control strategy will use the data from the accelerometer to drive the car up or down a slope and stop when it reaches the peak or valley.
3. The LCD will be used to display relevant data from the program to assist in troubleshooting.

Hardware

1. Wire a single protoboard with the Accelerometer added to the Ranger and Compass on the SMB. Remember, only one set of pull-up resistors is needed on the SMB. **Although the Ranger and Compass will no longer be used, it is OK to leave them on your car.**
2. Keep the Liquid Crystal Display and number pad on the SMB interface.
3. Keeping your battery monitoring hardware and software from Lab 4 may prove useful.
4. A radio frequency (RF) link will be added to allow for communication between the car and your laptop. Use this to receive telemetry information sent from the autonomous car to your laptop and also to aid in debugging your code.
5. Reinstall the piezoelectric buzzer from Labs 1 and 2 with the series resistor and control it with a digital output that passes through a 74365 buffer.

Software

1. Revise the C code used in Lab 4. Write a program that first calls the **accelerometer initialization routine**, then calls a `read_accel()` function and then sets the PWM for the

steering servo based on the side-to-side tilt of the car so that is turns in the direction of the upward slope. Both the side-to-side tilt and the front-to-back tilt will be used to determine a PWM for the drive motor. The main code will also need to average 4 to 8 samples from the accelerometer each time, since there is noise on the signal.

2. Continue to use routines to output parameter and settings on both the LCD display and SecureCRT terminal to aid in troubleshooting.
3. As in Lab 4, there will be parameters to adjust affecting the behavior of the system. This time there are 3 different adjustable proportional gain feedback constants that must be optimized to give the car the best performance. Write C code to allow user entry of gain constants using the keypad or terminal.

Motivation

This lab provides a chance to interface the *Smart Car* to another sensor system. Accelerometers are being added to many electronic devices to provide safety as well as convenience features. Being able to detect a free-fall condition may allow a laptop to park a hard drive and prevent a head crash that would render the drive completely useless. Similarly, 3-axis accelerometers can detect orientation of cameras and tablets so that text and graphics are automatically rotated to allow proper viewing.

The module used here is an incredibly sophisticated device that can detect accelerations along all three axes and generate interrupts based on maximum and minimum values in any direction with durations of more than or less than a predetermined period. It also includes a 3-axis compass that measures magnetic field strength in each direction. For this lab, only a subset of all the features will be used. Here the gravitational acceleration in the x and y directions are all that is needed to determine the tilt and direction of a slope. As such, only the 12-bit values associated with the x-axis (oriented as the side-to-side direction, +x to the left) and the y-axis (oriented as the front-to-back direction, +y to the front) when the module is inserted in the protoboard in the ascribed fashion shown below in Figure 5.1.

The goal of this lab is to control the steering and speed of the car in another example of how sensor information can be incorporated in a closed-loop feedback system. It is worth noting that aspects of this test mimics altitude control of a hovering blimp. For that situation, the altitude control is then dependent on both the thrust power and thrust angle.

In addition to the accelerometer integration, this lab involves monitoring the vehicle's battery voltage. Both the *Smart Car* and the Gondola are powered by rechargeable batteries that form a part of the unit. As the charge in the battery is being consumed by running the *Smart Car* or the Gondola motors, the output voltage of the battery slowly drops. Since the ICs in the circuit require a minimum supply voltage, they fail to function properly if the output voltage drops below the required minimum voltage supply. When the voltage is low, the operation of the processor may become erratic as the motors turn on and off. A symptom of a very low battery voltage is that the microcontroller will shutdown and restart. The battery monitoring circuit developed in Lab 4 may prove to be more important here. Driving up a slope significantly increases the load on the drive motor compared to driving on a horizontal surface. A given control algorithms will behave very differently when a battery is low and unable to deliver the full amount of current the drive motor is expecting under a specific set of conditions. Keeping your previous monitoring circuit and software will assist in determining if your car's battery needs to be recharged.

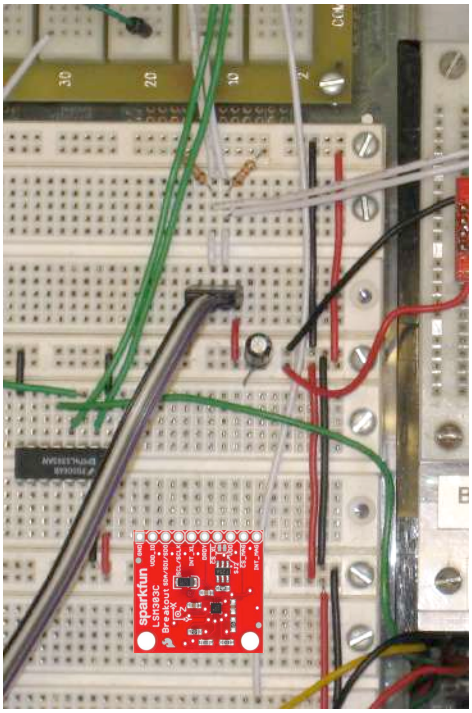

Figure 5.1 - Correct orientation of the Acceleration sensor (red module on <u>your</u> protoboard) on the car. +Y is toward the front of the car and +X is toward the left side.

Finally, this lab still requires the integration of an LCD screen and keypad. As before, the keypad may be used to input the desired heading and to set the steering gain constant. The screen is used to display current heading and the ranger reading. In labs 5 and 6 the LCD and keypad will be used as a way to display and set the gain constants for both heading and altitude. This allows these values to be set on-the-fly, rather than recompiling the program each time a gain is changed. Furthermore, the LCD can be used to display useful values such as the accelerations and gain values.

A radio frequency link will be added to the car to allow for communication between the car and your laptop. This is used to record telemetry data to be sent from the autonomous car to your laptop so that system response data can be plotted. The RF (radio frequency) link appears as another laptop serial port, but requires a new driver. The Gondola Info link on LMS provides the details as does the **Drivers for RF link** section in the **Installing_SiLabs-SDCC-Drivers** manual also on the LMS course front page.

With both the RF link and the LCD/keypad panel working, teams will have more flexibility in determining how they will enter parameters and update values. Either system can be used to provide menu-driven instructions for the user.

Lab Description and Activities

Continuing with Lab 4 and for the rest of the course, the 3-student team is expected to develop various parts of the software, each assigned to a member, in parallel. Each team must still maintain their lab notebook for final submission. **Make sure all team member names are on the notebook cover.**

The hardware and software from Lab 4 will be the starting point for this lab. Only a single protoboard from Lab 4 on should be used for final check-offs.

The integrated software and hardware will result in a car that can detect the direction of the ramp tilt and have the car drive in the direction of the maximum slope until it levels out at the bottom or top. The integrated software should poll the run/stop switch(es) connected to port pin(s) specified by you to start and stop any one or both control functions. There will be similar switches on the *Gondola*. The details are left up to the team.

Velcro has been added to the car to hold the LCD and keypad board. The board has a 4 wire cable that connects power, ground, SDA and SCL. The LCD board must be returned each class and not kept with your protoboard. NOTE: when unplugging the LCD board ribbon cable from your protoboard, **make sure you pull the ribbon cable by the header plug on the ribbon cable and NOT by the wire. The cable wires will be pulled off the connector pins.** It is extremely annoying when an LCD board can't be used because a broken wire prevents it from working or causes it to work intermittently.

Use one combined `printf()` statement that includes x & y accelerations (tilt values), the current gains, drive and steering pulse widths, (and battery voltage). This should be printed in columns (separated by commas) to allow processing and plotting using Excel or MATLAB.

Hardware

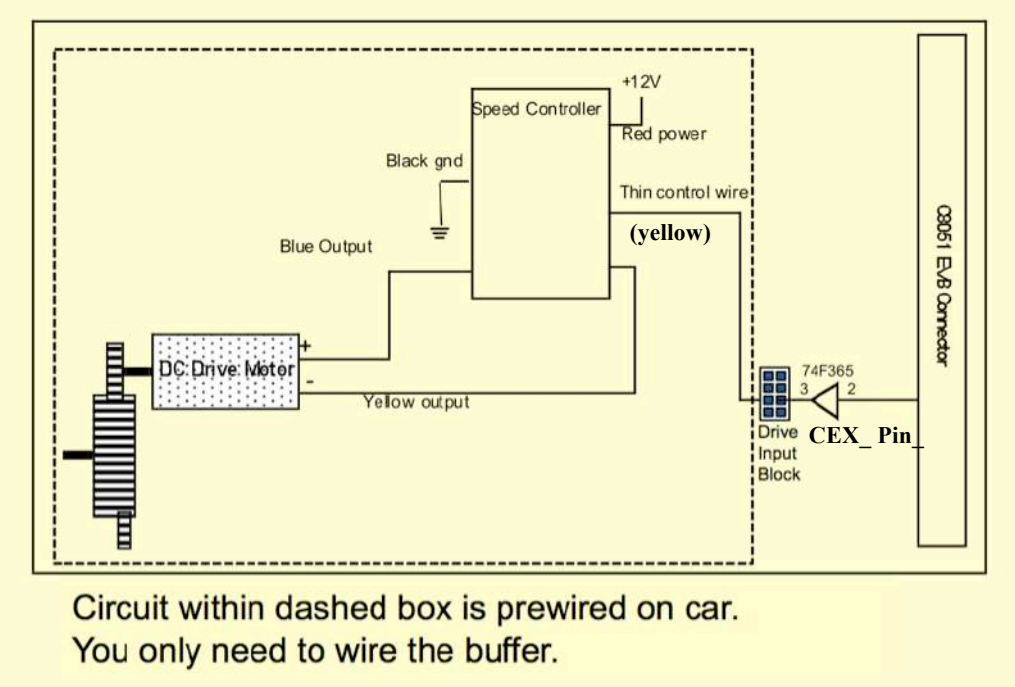

Figure 5.2 - Car drive-motor circuitry from Lab 3 (part 1).

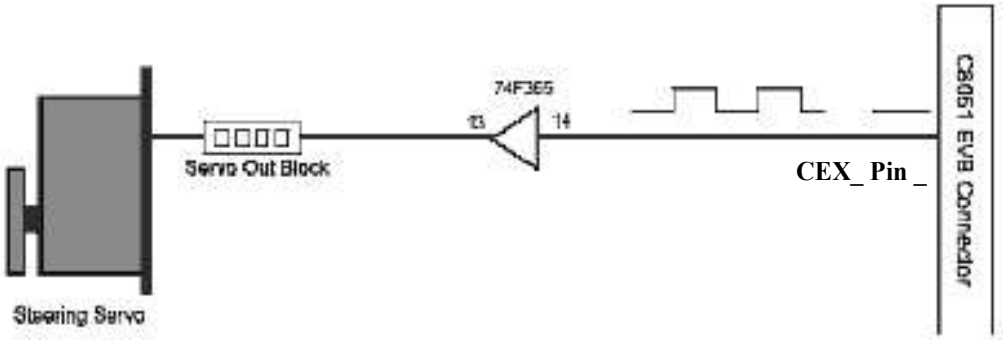

Figure 5.3 - Car steering servo motor control circuitry from Lab 3 (part 1).

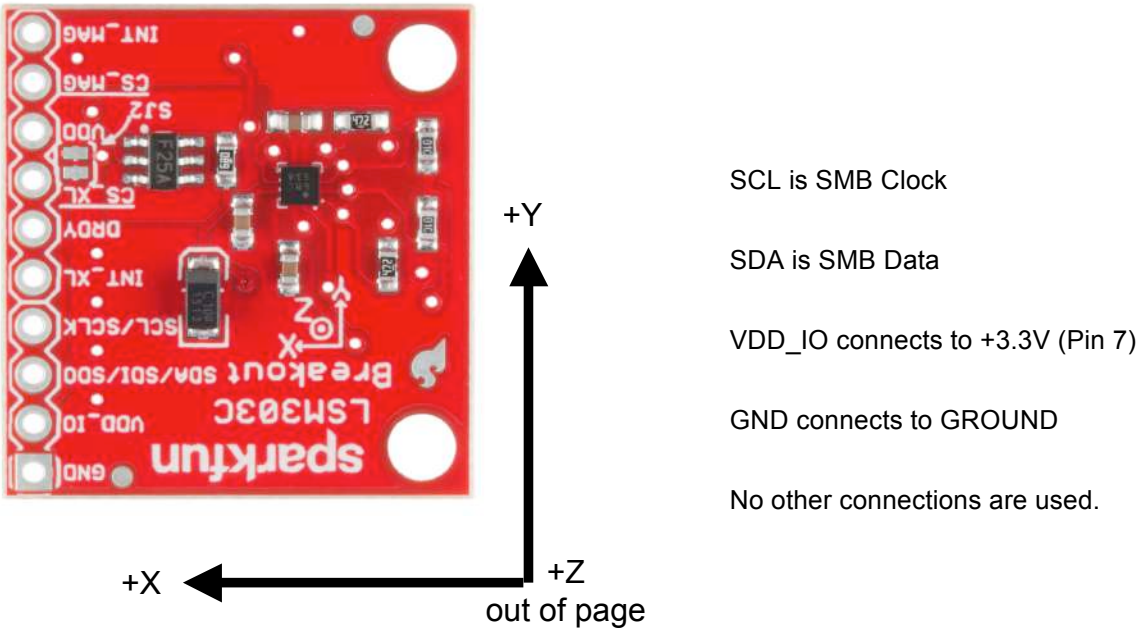

Figure 5.4 – Orientation and labeled connections for the Acceleration module.

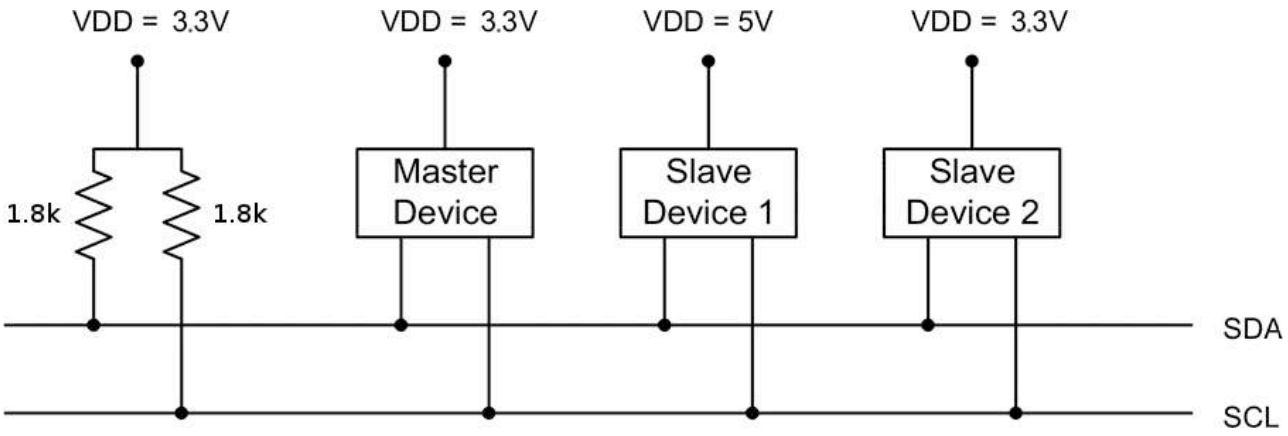

Figure 5.5 – Configuring multiple SMB slaves on a single master.

Car RF Transceiver Module

The radio frequency (RF) transceiver module on the car communicates with the matching USB RF transceiver module connected to your laptop. This is used to establish a serial connection in the same way the wired serial cable was used, which allows data to be transferred between the car and your laptop. With this commands from your laptop are sent to the car and output from the car sent to your laptop on your SecureCRT terminal.

The car RF transceiver module requires 5V power (pin 4, black wire) and ground (pin 2, orange wire) lines. Pin 5 (white wire) is also grounded To send and receive data, it also requires connections to TX (pin 1, brown wire) and RX (pin 3, green wire), which connect to P0.0 and P0.1 on the EVB respectively. Make sure the 10-pin header is attached to the module as shown in the photo, with the brown wire closest to the side with the 4 jumper pins used to set the baud rate. The wired RS-232/USB and the wireless RF **CANNOT** both be used simultaneously. There will be conflicts. If the wired connection is used the connections to P0.0 and P0.1 on the EVB bus must be temporarily pulled out. If the wireless is used the RS-232 DB9 plug must be disconnected from the EVB.

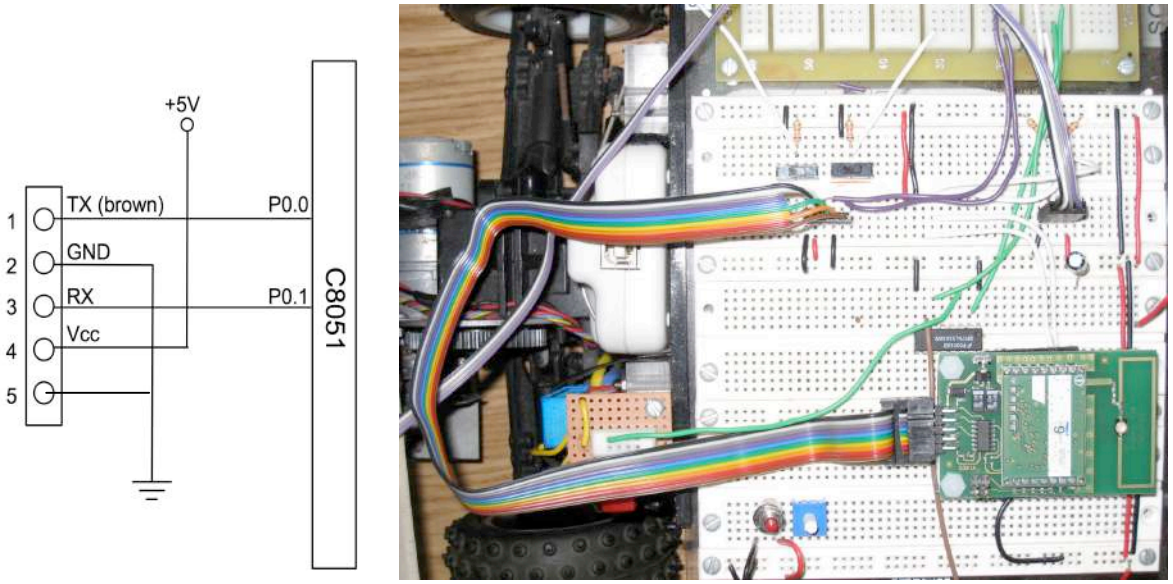

Figure 5.6 – Connections for RF transceiver and photo of module.

Note: It is suggested that the previously used additional hardware – a run/stop slide switch connected to any unused I/O pin of the team choice, still be used to disable the drive motor on the car so that adjustments can be made with the program running without requiring that the car be placed on a foam block. Use whatever was done in Lab 4.

Software

Write code to use the LCD and keypad and/or SecureCRT to print the results.

Continue developing the code from Lab 4 with the following constraints:

1. Include a run/stop slide switch.
2. Of the 2 drive feedback gains, the drive gain for the front-back pitch may be selectable via the pot setting used in Lab 2 or 4. Adjusting the pot from min to max should create a gain that varies from 1 to 50 (a pure integer value is fine) that can be updated on the fly as the car

drives. The other gain is to be set by pressing keys on the keypad or the SecureCRT terminal and is fixed for a run. The system should allow the operator to enter a specific gain, or pick from a predefined list of values. If the user selects a specific desired gain, the screen should prompt for a specific value. Monitoring the battery voltage is again optional, but when using the pot value on a separate analog input from the battery voltage monitor, it is necessary to wait ~2 ms after changing the analog MUX channel selector before starting a conversion, otherwise the converted value will be wrong.

3. The steering gain must also be selectable. As with the heading, the entry is done using the keypad. Entry options include: 1) select from a menu, 2) increment or decrement using key strokes, or 3) key in a value. (Please only pick one option, not all.)
4. Once everything has been specified, the LCD and/or terminal should display the current x & y accelerations, the current gains and the motor pulsewidths. Updating the display every 400 ms is reasonable (but not more frequently).

There are several other considerations:

1. The steering servo and the speed controller must be updated every 20 ms. The PCA hardware does this as long as it is properly configured.
2. The Accelerometer updates every 20 ms (50 times a second). The controller will work best if each reading is a new value, so continue to update the heading after a PCA ISR.
3. The keypad can be queried for input when appropriate. This shouldn't be done faster than once every few ms. Alternatively the laptop keyboard may be queried using the function `getchar_nw(void)`, the version or `getchar()` that immediately returns a value of 0xFF if no key has been pressed, but otherwise returns the normal ASCII code for the character.
4. It is necessary to try several different gain constants for the gains. The car should attempt to get to the bottom or top of the ramp quickly in a short distance but the noisy acceleration values may cause the steering adjustments to be jerky.
5. For this semester the car must drive up the ramp in reverse. While driving in reverse, the buzzer from Labs 1 & 2 (mounted as described in those labs to any available digital output) should be beeping with a regular on-off pattern (on for 0.5 s, off for 1.0 s).
6. The condition where maximum error in the x or y tilt corresponds to a maximum pulsewidth should be reevaluated to produce a faster response in the system. This can be accomplished by increasing the gain coefficient, however, care should be taken so that the maximum and minimum allowed pulsewidths are not exceeded.
7. After an initial estimate of an appropriate gain, code can be implemented to allow the user to change the gain term upon execution rather than recompiling the software. Before entering the infinite loop, the program asks to manipulate the proportional gain of the steering. Use the following function and call it from the main function. The function below uses SecureCRT, but your code may use the keypad instead.

Specifically, here are some things to remember:

1. The i2c.h header function that should already be downloaded from the LMS course page and put in the correct folder so that SDCC has the required accelerometer initialization function, `void Accel_Init_C(void).` **Make sure your code calls this** with all the other initializations.

2. The Accelerometer values are read from registers in the module. The SMB ID address of the Accelerometer is **0x3A**. The status register is 0x27. The 2 least significant bits 0 and 1 in the status register go high when the x-axis and y-axis acceleration values are ready to be read. At that point registers 0x28 and 0x29 contain the 12-bit x-axis acceleration and registers 0x2A and 0x2B the y-axis acceleration, with the low byte in the lower register number. Since the acceleration values are extremely noisy, the low byte will be discarded (effectively set to zero, leaving only the 8 most significant bits from the high byte) for both the x and y values. After sign-extending the high byte to get a 16-bit signed integer (logical shift with << 8), the equivalent 12-bit value (8 bits of real data and 4 bits of zeros) is in the 12 high order bits. The value needs to be shifted down into the 12 low order bits (logical shift with >> 4). Since the values are still noisy, 4 reading should be made consecutively and all averaged together to give the program something that is practical to use. This is a common problem with many sensors where digital filtering (averaging) is used to remove noise from signals. Once the averages for both the x and y data have been calculated, the accelerometer read routine should set global variable for the control calculation to use. Make sure the array (Data[4]) is still declared as **unsigned char**. The pseudocode for the read routine is given below.

```
Clear the averages
    avg_gx = 0;     //declared as signed int
    avg_gy = 0;     //declared as signed int
For 4 iterations (or maybe 8)
    (A delay is no longer required: Wait one 20ms cycle to avoid hitting the SMB too frequently and locking it up)
    Read status_reg_a into Data[0] (register 0x27, status_reg_a, indicates when data is ready)
    If the 2 LSbits are high: (Data[0] & 0x03) == 0x03, then continue, otherwise reread the status
    Read 4 registers starting with 0x28. NOTE: this SMB device follows a modified protocol. To
    read multiple registers the MSbit of the first register value must be high:
        i2c_read_data(addr_accel, 0x28|0x80, Data, 4);     //assert MSB to read mult. Bytes
        Discard the low byte, and extend the high byte sign to form a 16-bit acceleration
          value and then shift value to the low 12 bits of the 16-bit integer.
          Accumulate sum for averaging.
                avg_gx += ((Data[1] << 8) >> 4);     //a simple ">> 4" WILL NOT WORK;
                avg_gy += ((Data[3] << 8) >> 4);     //it will not set the sign bit correctly
          Or to attempt to acquire 12 bits of accuracy with more required averaging, use:
                avg_gx += ((Data[1] << 8 | Data[0]) >> 4);
                avg_gy += ((Data[3] << 8 | Data[2]) >> 4);
Done with 4 iterations (or maybe 8)
Finish calculating averages.
    avg_gx = avg_gx/4 (or use >> 2 for faster execution)
    avg_gy = avg_gy/4 (or use >> 2 for faster execution)
Set global variables and remove constant offset, if known.
    gx = avg_gx (or gx = avg_gx - x0 if nominal gx offset is known)
    gy = avg_gy (or gy = avg_gy - y0 if nominal gy offset is known)
```

3. The control statement for the steering servomotor and drive motor and will be:

```
steering_pw = steering_pw_center - ks * gx        (ks is the steering feedback gain)
drive_pw = drive_pw_neutral + kdy * gy            (kdy is the y-axis drive feedback gain)
Add correction for side-to-side tilt, forcing a forward movement to turn the car.
drive_pw += kdx * abs(gx)                         (kdx is the x-axis drive feedback gain)
```

One additional issue that may be addressed is the asymmetrical strength of the drive motors between forward and reverse. This results in inaccuracies with the car stopping at the peak or in a valley. You may choose to modify your code to adjust for this by incorporating different feedback gains, depending on the direction of travel. The solution to this is left open-ended. Introduction of an integral term in the feedback control alongside the proportional term will provide a mechanism that will increase the value on the drive motor over time until it is large enough to actually move the car. The drive motor forcing function modifications would be something like:

```
drive_pw += kdx * abs(gx) + ki * error_sum        //ki is the integral gain
error_sum += gy + abs(gx)
```

NOTE: Each accelerometer must be calibrated for a correct zero point so that the x & y readings return a ~0 when the car is on flat ground. This should be done EVERY TIME at the start whenever your program runs. Tilting the car 90° in any direction will change the zero point significantly so that the car may try to move even while on a flat surface. New values must be found if the car has been jostled or the accelerometer has been changed. It is suggest that 64 reading be averaged for both the x & y values and those numbers be saved and subtracted from every reading. Make sure your car is on a flat surface and unperturbed when the program begins its calibration to insure accurate offset values are found.

The following function can be used to adjust the value of the feedback gains on the car without having to recompile. If the keypad is used to enter characters rather than the terminal keyboard, the gains may be adjusted without having to carry a laptop around. This function allows the user to bump a value up or down with key presses. Alternatively, a function could be written to enter a gain from scratch.

```
void Update_Value(int Constant, unsigned char incr, int maxval, int minval)
{
    int deflt;
    char input;

    // 'c' - default, 'i' - increment, 'd' - decrement, 'u' - update and return
    // This can easily be changed to use the keypad instead of the terminal
    deflt = Constant;
    while(1)
    {
        input = getchar();
        if (input == 'c') Constant = deflt;
        if (input == 'i')
        {
            Constant += incr;
            if (Constant > maxval) Constant = maxval;
        }
        if (input == 'd')
        {
            Constant -= incr;
            if (Constant < minval) Constant = minval;
        }
        if (input == 'u') return;
    }
}
```

Data Acquisition

When your code is functioning correctly, gather data to plot response curves for your steering control. You should plot x & y accelerations and both motor pulsewidths (y-coordinate) vs. time (x-coordinate). In order to save the data, you will need to print the x & y accelerations to the SecureCRT screen and then copy the output to a plotting utility, such as Excel or MATLAB. Read the **Terminal Emulator Program** section in the **Installing_SiLabs-SDCC-Drivers** manual on LMS for more details. You need to obtain a few curves for different gain combinations. You must find an 'optimal' setting, but you should also look at other combinations to verify trends. Plot the data as a scatter plot or straight-line scatter plot (in Excel). Make sure the axes show units: seconds or ms on the x-axis, and mg on the y-axis (where $1g = 9.8m/s^2$ so $1mg = 9.8mm/s^2$).

Lab Check-Off: Demonstration and Verification

1. Demonstrate how the heading and drive direction & speed are determined by the tilt orientation. Show by lifting the car and changing the pitch and roll (look up these terms if you aren't sure of their definition) that the steering and drive will try to compensate for tilt error.
2. Set the desired feedback gains using the keypad.
3. Place car at the top or bottom of the ramp (see current semester specs), pointing perpendicular to the slope. Move slide switch to 'run'. Car direction is controlled by the accelerometer tilt value. Car will turn to the head down or up the slope and maintain a heading that maximizes the slope. The steering may be jerky but should attempt to achieve desired heading in a short distance of travel. Try starting from both the left and right corners. The car must drive up the ramp in reverse for the Fall 2017 check-off while sounding the buzzer and stop the buzzer once parked at the top.
4. Car will stop at the valley at the bottom or peak at the top of the slope when it levels off and starts to point uphill or downhill. Car must be parallel to the direction of maximum slope when it stops. (If no valley is present, just stop at the bottom.)
5. While driving, the program should find the maximum slope of the ramp and display that value (in degrees) when it stops at the top.
6. Your TA may ask you to explain how sections of the C code or circuitry you developed for this exercise works. To do this, you will need to understand the entire system.
7. Display the gains, motor pulsewidths, and x & y acceleration values on the LCD screen.
8. Print the output to the terminal screen in the following format:

```
X accel. - Y accel. - Drive PW - Steering PW
  xxxx,      xxxx,      xxxx,      xxxx
  xxxx,      xxxx,      xxxx,      xxxx
  ....,      ....,      ....,      ....
  xxxx,      xxxx,      xxxx,      xxxx
```

   Capture the screen output and save it to a file on your laptop PC.

Writing Assignment – Lab Notebook

Enter full schematics of the combined circuit. Print and attach the full code. Both pairs of students are responsible for both the schematics and the code. Both pairs must keep copies of the code (for hard drive crash recovery!). In the lab notebook describe what had to be changed to allow functions to be merged. Initialization functions are of particular note. Include a discussion of how the code meets the required timing of the sensors.

Several steering gains must be tried. Make comments about the car performance with both high and low steering and drive gains. Determine a usable set of gains for your car.

Justify the algorithm provided to adjust the steering servomotor and drive motor based on the x-axis and y-axis acceleration values. Explain how the pitch and roll tilts affect these measurements and how the measurements are used to change the steering angle and drive speed & direction.

Sample C Code for Lab 5

The following code provides an example of the main function for combining control of both the steering and the drive motors. Your debugging skills developed through experience in deciding which values to observe on the terminal and LCD display while troubleshooting will speed up the development process:

- Initialization routines are declared, but the routines should be taken from your previous code
- Pulse width variables are declared globally for both the steering servo and the drive motor
- Flags to read the accelerometer and the keypad are declared and set in the PCA Interrupt Service Routine
- Functions to set the PCA output pulses are called, but not defined. They should be taken from your previous code.
- Functions to read the sensors are called, but not defined. They should be modified versions of similar functions in your previous code.

```
/* Sample code for main function to read the accelerometer */
#include <c8051_SDCC.h>
#include <stdlib.h>// needed for abs function
#include <stdio.h>
#include <c8051_SDCC.h>
#include <i2c.h>

//-----------------------------------------------------------------------------
// 8051 Initialization Functions
//-----------------------------------------------------------------------------
void Port_Init(void);
void PCA_Init (void);
void SMB_Init (void);
void Interrupt_Init(void);
void PCA_ISR ( void ) __interrupt 9;
void read_accel (void);    //Sets global variables gx & gy
void set_servo_PWM (void);
void set_drive_PWM(void);
void updateLCD(void);
void set_gains(void);  // function which allow operator to set feedback gains

//define global variables
unsigned int PW_CENTER = ____;
unsigned int PW_RIGHT = ____;
unsigned int PW_LEFT = ____;
unsigned int SERVO_PW = ____;
unsigned int SERVO_MAX= _____;
unsigned int SERVO_MIN= _____;

unsigned char new_accel = 0;  // flag for count of accel timing
unsigned char new_lcd = 0;    // flag for count of LCD timing
unsigned int  range;
unsigned char a_count;        // overflow count for acceleration
unsigned char lcd_count;      // overflow count for LCD updates

//-----------------------------------------------------------------------------
// Main Function
//-----------------------------------------------------------------------------
void main(void)
{
   unsigned char run_stop;    // define local variables
   Sys_Init();                // initialize board
   putchar(' ');
   Port_Init();
   PCA_Init();
   SMB_Init();
   Interrupt_Init();
   Accel_Init_C();

   a_count = 0;
   lcd_count = 0;

   while (1)
   {
      run_stop = 0;
      while (!run)            // make run an sbit for the run/stop switch
      {                       // stay in loop until switch is in run position
         if (run_stop == 0)
         {
            set_gains();      // function adjusting feedback gains
            run_stop = 1:     // only try to update once
         }
      }
```

```
        if (new_accels)         // enough overflows for a new reading
        {
            read_accels();
            set_servo_PWM();    // set the servo PWM
            set_drive_PWM();    // set drive PWM
            new_accels = 0;
            a_count = 0;
        }

        if (new_lcd)     // enough overflow to write to LCD
        {
            updateLCD(); // display values
            new_lcd = 0;
            lcd_count = 0;
        }
    }
}

//-----------------------------------------------------------------------------
// PCA_ISR
//-----------------------------------------------------------------------------
//
// Interrupt Service Routine for Programmable Counter Array Overflow Interrupt
//
void PCA_ISR ( void ) __interrupt 9
{
    if (CF)
    {
        CF = 0; // clear overflow indicator
        a_count++;
        if(a_count>=____)
        {
            new_accel=1;
            a_count = 0;
        }
        lcd_count++;
        if (lcd_count>=____)
        {
            new_lcd = 1;
            lcd_count = 0;
        }
        PCA0 = PCA_start;
    }
    // handle other PCA interrupt sources
    PCA0CN &= 0xC0;
}
```