Goals for this Lab Assignment:
1. Learn about and use the LADAR, IR distance sensors, the Ultra-sonic distance sensors, the rate gyro and the compass.
2. Modify a Linux application for communication with the OMAPL138’s DSP through the Console and LABVIEW.
3. Using the LADAR and possibly the distance sensors, implement a wall-following algorithm to have the robot navigate a simple course.
4. Use the rate gyro to find the bearing of the robot and keep track of the robot’s x, y position.

SYS/BIOS Objects Used: Swi, Task

Library Functions Used:

Lecture Topics: LADAR, IR and Ultrasonic distance sensors, Compass and Rate Gyro Sensors.

Prelab: If you have not done so already, update your LABVIEW GUI to be able to display both the X and Y position of where the robot is located in the course. Additionally, setting up your LABVIEW application to download gains entered into text control will be very useful when tuning parameters in your final project.

Laboratory Exercise

Big Picture: This lab has a number of tasks to get you to the final goal so a general overview is in order. The goal is to have your robot follow a wall that is on its right and when it comes to a corner turn to the left and continue right wall-following. At the same time, continuously upload coordinate data to your LABVIEW program displaying the robot’s location relative to a start position. Along the way to this final goal of wall following, you will be given a Linux programming introduction by modifying a Linux program to perform the communication with the DSP to the Linux console or LABVIEW.

I highly recommend you READ THE LAB COMPLETELY BEFORE YOU START CODING so you get the full picture of the assignment.

Exercise 1: Reading sensors

Besides the LADAR, all the sensors you will be working with in this lab are read for you by the TMS320F28335. The F28335 communicates this data to the OMAPL138’s DSP every 1ms. After receiving this data, the Swi, SW1_RobotControl, is posted causing the function RobotControl() to be run. Much of your code for this lab will be placed inside this RobotControl() function. Make sure to first perform a “SVN update” on your repository to make sure you have the latest files. Then create a new OMAPL138 project with “Lab678OMAPL138ProjCreatorFiles\Lab678OMAPL138ProjCreator.exe.” Remember to create unique project names that include you and your partner’s initials or something in order that there will be no confusion which executable file is yours. Import the new project into CCS and pull up the main C-file. Browse through this C-file. At the top of the file (around line 47) notice the long list of “extern” global variables. These global variables store the readings of the sensors communicated from the F28335. The comments of each of the ADC readings indicate to which sensor that channel of ADC is connected. When SW1_RobotControl is posted all the sensor values have been copied to these variables and ready for you to use.
Scroll down farther and you will see the ComWithLinux() Task function. As in Lab 2, this Task is used to communicate with Linux on the OMAPL138’s ARM core. We will come back to this in Exercise 2.

Scroll down further past the main() function and find RobotControl(). The given code blinks the LCD’s green, blue and red LEDs every 1s, and also every 1ms it calls the

SetRobotOutputs(vref, turn, pwm3, pwm4, servo1, servo2, servo3, servo4, dac1, dac2) function. This function transmits data back to the F28335 processor for it to use as outputs or commands. For this lab you will be using the vref and turn variables of this function. Set all the other variables to zero in your calls to this function. The given F28335 code implements the coupled PI control algorithm you developed in Lab 5. The vref variable sets the reference speed of the robot and turn sets the rate of turn of the robot. Place your code for working with the sensor data here in RobotControl().

Below is a list of the sensors you will be using. Each have different update rates and output ranges


   LADAR stands for Laser Detection And Ranging. Every 100ms the LADAR gives the robot 228 different distance readings in an angle range of 240°. The range of the reading is between 20mm and 4000mm. If a reading is less than 20 then it is an error code instead of a distance reading. The error codes are given in this document [http://coecsl.ece.illinois.edu/ge423/datasheets/ladar/hokuyo-URG-04LX-SCIP1-com-spec.pdf](http://coecsl.ece.illinois.edu/ge423/datasheets/ladar/hokuyo-URG-04LX-SCIP1-com-spec.pdf). With 228 points in the 240° operating range, this is a distance reading every 1.05° starting on the right side of the robot. The angles start on the right of the robot with the angle of -120° and end on the left side of the robot with an angle of 120°. As you would expect, 0° points straight ahead of the robot. The project creator gives you the code that reads the LADAR data and creates two arrays with the LADAR angles and the distances read at each of these angles. The angle array is called float LADARangles[228] and the distances are stored in the array float LADARdistance[228]. Each array index specifies an angle and the distance sensed at that angle. When you are deciding which of these 228 data points you would like to use in your robot navigation code, debug your code and place both of these arrays in the “Watch Expressions” window. Then you can look at the angle of each index and decide what distances to use. If you would like to read more about this LADAR its documentation is found at [http://coecsl.ece.illinois.edu/ge423/datasheets/ladar/](http://coecsl.ece.illinois.edu/ge423/datasheets/ladar/).


   IR one is connected to F28335’s ADCA4 and IR two is connected to ADCA5. Once powered, this IR distance sensor produces a new distance measurement approximately every 40ms. In between each measurement the sensor holds its output voltage to the value of the previous reading. The F28335 samples this output voltage every 1ms. So for 40 or so samples the F28335 will be sampling the same value. Below is a plot of the sensor’s output verses distance.
Looking at the plot you can see that the voltage to distance output is not linear and when the sensor gets too close to an object the output decreases as it does when an object is moved farther from the sensor. The reading from the sensor when too close (within 10cm) jumps around quite a bit and should not be used for measuring distance. We need to take that into account when deciding where to place this sensor on the robot. Another issue with this sensor is that a large voltage indicates when you are close to an object and a small voltage indicates a larger distance. You will reverse this direction because it is more intuitive and easier to use in a control law calculation when a small value means a short distance and a large value means a long distance. ONE LAST NOTE: Even though you could calibrate this IR sensor by taking a number of distance measurements and fitting a curve to those data points to give a conversion between volts and cm, I DO NOT recommend you perform that step. The reason is that many of these sensors are different from each other and you would have to perform this calibration on each sensor. Instead I recommend you use the raw ADC value of this sensor which will be a value from 0 to 4095 (12bit ADC).


UltraSonic one is connected to F28335’s ADCB4 and Ultrasonic two is connected to F28335’s ADCB5. The maximum update rate for these sensors is 50ms. Working with these sensors in the past I have found that the readings are quite a bit noisier updating every 50ms compared to commanding the sensor to update at a slower rate like every 150ms. There is a digital input pin on this sensor called RX. The sensor can be commanded when to take a distance reading by pulsing this pin high for 20us or longer. Currently the F28335 pulses the RX pin of the ultrasonic sensors every 150ms. As with the IR sensors, the F28335 is sampling its ADC inputs every 1ms. So the ADC channels connected to the ultrasonic sensors will read the same value for 150ms.

Telling both ultrasonic sensors to collect a distance reading at the same time is not a good idea even if they are pointing in opposite directions. This is due to how the sensor works. It outputs a high frequency sound wave and waits for that sound wave to return. If you fire two or more ultrasonic sensors at the same time they could very possibly interfere with each other. So the F28335 code alternates which ultrasonic sensor is performing a distance measurement. Every 75ms the F28335 will toggle which sensor to activate for a new reading.
The ultrasonic sensors do not have the issues discussed about the IR sensors. With the ultrasonic sensor, a small voltage reading relates to a small distance and the conversion between voltage and distance is much more linear than the IR distance sensors. Similar to the IR sensor, the ultrasonic sensor cannot measure short distances less than 15cm, but at least the measurement is constant in that range compared to the IR’s reading changing to a value that indicates the sensor is a long distance away from the object.

Again for our purposes of this lab performing wall-following, I DO NOT recommend that you calibrate this sensor by finding the conversion between volts and cm. Instead simply use the raw ADC reading as your distance measurement.


   The digital compass is the most problematic sensor on the robot. Not because of noise or nonlinearity but instead because of all the metal in the room and underneath the floor in the mechatronics lab. Metal plays havoc with this sensor. Care must be taken when using the compass reading. Depending where you are in the course, the compass reading can vary by 45 degrees or even more. I have found the only use of this compass in the lab room (and a number of other compasses I have tried) is to indicate a general direction N, NE, E, SE, S, SW, W, NW.

   The HMC6352 compass is an I2C device and the F28335 requests a reading from it every 40ms. The units of this reading are $10^{\text{o}}$s of a degree.


   This gyro sensor is actually a dual axis sensor but since the robot can only rotate about z or yaw axis we will only be interested in measuring that one axis. The gyro sensor also outputs two ranges, -400°/s to 400°/s (0.23Volts to 2.23Volts) and -100°/s to 100°/s (0.23Volts to 2.23Volts). The 400°/s range is sampled by ADCA3 and the 100°/s range is sampled by ADCA2. When we get to wall following at the end of this lab you will try both of these ranges and see which one works best for calculating the bearing of our robot.

   The gyro, compared to all the sensors described above, has a very fast response. Each 1ms sample, will be a new reading of this sensor. To find the bearing angle of the robot, you will integrate this signal.

   **Add the following code:** Inside RobotControl(), add code to print the sensor readings to the text LCD screen every 100ms. As you did in Lab 5, use the global variable `switchstate` to determine the state of the four switches on the shield board and print different sensors to the text LCD screen depending on the position of the switches. **NOTE:** the global variable `switchstate` is defined as a float in the OMAPL138’s code. So when using this variable in a switch statement you will need to type cast it to an `int`. Print some of the 228 distance readings of the LADAR: `LADARdistance[1] ≃ -118°, LADARdistance[225] ≃ 118°, LADARdistance[172] ≃ 62°, LADARdistance[54] ≃ -62°, LADARdistance[113] ≃ 0°`. Next print the IR distance readings and the ultrasonic distance readings in raw ADC units. Also print the compass value which is units of a $10^{\text{o}}$ of a degree. For the gyro, convert both the 100°/s range ADCA2 reading and the 400°/s range ADCA3 reading to radians per second and print these scaled values to the LCD. (Recall that all 12-bit ADCs map a 0-3 V signal to the range 0-4095.) Don’t forget to subtract off the zero °/s voltage of 1.23. **(NOTE: 1.23V is not exactly the zero voltage every time you turn on the robot. This value changes slightly, so later in this lab you will be asked to have the robot sit still for 3**
seconds before it starts wall following in order to find this gyro zero value.) If you perform this scaling correctly, both gyro readings should produce the same angular rate measurement up to 100°/s.

Add one additional print “dip switch” state to get your program ready for exercise 2, print the values of \texttt{vref} and \texttt{turn}. You will also need to delete the two lines of code in the \texttt{RobotControl()} function that set \texttt{vref} and \texttt{turn} always to zero. You are going to be asked to load your programs in a different fashion this assignment so DON’T USE the “bug” icon to build and load this project. Instead just build your project by right clicking on the project and selecting “Build.”

Instead of using the JTAG to load code to the OMAPL138’s DSP, you are going to practice using Linux (running on the OMAPL138’s ARM core) to load your compiled programs. You will find this a very useful tool, because you will be able to copy, load and run your new program using Ethernet instead of the JTAG. This means your robot can be on the floor without any physical connection to your workbench and you still will be able load your new program on the OMAPL138’s DSP. Of course the JTAG will still play an important role when you need to debug the source code you have just written, but when making small changes to your code you will find this remote program loading a life saver.

Before you can load code to the OMAPL138’s DSP, you need to be inside a terminal window. You have already used COM1 at the work stations to bring up Linux inside Tera Term. At the bench this is the easiest terminal to use. Simply plug the serial cable into the robot and hit the enter key and you should see a log in prompt. Use the user “root” with no password to enter Linux. When your robot is off the bench too far for the serial cable to reach, you can use the SSH (Secure Shell) program, Putty, to connect a terminal over WiFi. At a command prompt simply type “putty root@192.168.1.???” where ?? is your robot’s ending IP number. This will launch an SSH terminal for you, connecting to your robot.

Load your program. In order to Load/Run your compiled program, you need to copy the compiled file to Linux. Using a Windows “CMD prompt”, change directory to the location of your current compiled project. Use the “cd” command to change directories and the “dir” command to list the directory’s contents. Inside this project directory you should find a file call <yourprojectname>.hex. This is the compiled file that needs to be copied to Linux. At the Windows command prompt copy the .bin file to your robot’s Linux /home/root directory by typing

```
pscp <yourprojectname>.hex root@192.168.1.??:
```

where ?? is your robot’s last IP address number and don’t forget the “:”. You will be asked for a password, so just press enter since we have not set up a password. Now go back to your Linux console and enter “ls”. You should see your .bin file among the other files stored in your robot’s /home/root directory.

To load the program you just copied, first run the command “./DSP_Reset”. This holds the OMAPL138’s DSP in the reset state. Then type “./DSP_Load <yourprojectname>.hex”. This will load and run your program. You should see your code working and the sensor readings printed to the text LCD screen. Change the switches and see all the sensor readings. When you have an OMAPL138 DSP program that you would like to store in flash so that it boots to that program on power up or reset, perform these steps. Enter the command “./DSP_Reset”, then enter “./DSP_Flash <yourprojectname>.hex”. Wait for all five red LEDs to continuously blink on and off. This indicates that the flash procedure has completed successfully. Then reset or power off and on the robot to see your newly flashed program run on reboot. Practice loading and flashing your DSP program a couple of times.

Change the switches in order that the IR distance readings are being displayed. Putting an object in front of one of the sensors, you will notice that when the object is moved closer to the sensor (but not closer than 10cm its invalid range)
the ADC reading becomes larger. This is not desirable for a signal to be used in a feedback control law. Modify your code to switch the direction of these two IR signals by subtracting them from their largest possible reading of 4095. For example, IR1 = 4095 – IR1. Compile, copy to Linux and reload your code to verify that the IR sample value now increases as an object is moved away from the sensor.

As a last item for this first exercise, drive around the robot using optical encoder #3 as you did at the end of Lab 5. The given code on the F28335 is already implementing the coupled PI control you implemented in Lab 5. All you need to do on the OMAPL138’s DSP end is send the value of \(vref\) and \(turn\) to the F28335. So modify your code so that \(vref\) is set to 1.0 tile/second and \(turn\) is set to optical encoder #3’s value divided by 100. \(vref\) and \(turn\) are sent to the F28335 by passing their value to the

\[\text{SetRobotOutputs}(vref, \text{turn}, \text{pwm3, pwm4, servo1, servo2, servo3, servo4, dac1, dac2})\]

function. Add one more feature to this drive about code. Stop the robot if the minimum of the five front LADAR distance readings detect a close object. (Remember index 113 is approximately 0º.) If the object is removed, return the robot to its driving around mode. Show this working to your instructor.

To get ready for exercise 2, comment out the line in your C code that assigns \(\text{turn}\) to encoder three’s reading divided by 100 and also make sure to only assign \(vref=1.0\) once when you declare it as a global. Do leave in your code the functionality that stops the robot (setting \(vref=0\) and \(\text{turn}=0\)) if the front LADAR readings detect a close object. In exercise 2 you will be changing the turn value from a Linux program.

**Exercise 2: Modify and Compile a Linux program to run on the OMAPL138’s ARM core.** For this exercise we will be compiling a Linux program from the Windows 7 command line using an open source “GCC” compiler.

Open a command window “cmd”. There should be an icon on your desktop. Using the “cd” command, change the directory to your repository folder and then to the “linux” directory. Use “dir” to list the contents of the directory. List the contents of this directory and you will find a number of different directories containing Linux applications and source code.

Create your own directory here to store your new application using the “md” command. A unique name for your application would be smart here because you will be copying this application to the robot’s root directory. Change into this new directory and type the following to copy files of a starter program into your directory: “copy ..\DSPCommShell\* .”

You should now have the following files in your directory:

```
dspcomms.h
DSPcommshell.c
DSPcommshell.h
Makefile
netapi.c
netapi.h
omap1138_gpiofuncs.c
omap1138_gpiofuncs.h
```

Rename DSPcommshell.c and DSPcommshell.h to the unique name you chose for this directory by typing: “ren DSPcommshell.c <youruniqueiname>.c” and “ren DSPcommshell.h <youruniqueiname>.h.”

Edit “Makefile” (type “notepad++ Makefile”) and perform a find and replace replacing DSPcommshell with <youruniqueiname>.

Edit <youruniqueiname>.c and perform a find and replace replacing DSPcommshell with <youruniqueiname>.
Edit <youruniqname>.h and perform a find and replace replacing DSPCOMMSHELL, all Caps, with <YOURUNIQUENAME>, all Caps.

Now you should be able to build this application by typing “make.” After the build is complete, list the files in your directory and you should see a number of .o files and your application file which has no file extension.

Before we copy and run this program on your robot’s Linux, let’s take a quick walk through the given code. Keep in mind that there are two sections/purposes of this program. 1. To interface with you the user at the command prompt. 2. After exiting the command prompt user interface, connect to a TCPIP socket and receive and send data to your LABVIEW application. Open your Linux program’s main C file by typing at the command prompt “notepad++ <youruniqname>.c”. Scroll down through the file. First you will see the global variables used in this application. Then into the main() function, the beginning code is mapping memory space in order to allow this Linux application read/write privileges to the DSP/ARM shared memory and to control the processor’s GPIO pins.

The next section of code in main() implements a simple user interface requesting information from the user. Study this code section in more detail. You will be adding additional options to this user interface. Inside this while loop the menu options for the interface are displayed. Then the mygetch() function is called which waits for any key on the keyboard to be pressed. Once a key is pressed, mygetch() returns with the character corresponding to that key. Next the returned character is checked using a switch case statement. Depending what key was pressed a different operation is performed. If ‘q’ is pressed the floating point variable turn is decremented by 0.2 and the new turn value is sent to the OMAPL138’s DSP. If ‘s’ is pressed the program asks the user for a new velocity set point (vref), and once entered, sends that new value to the DSP core. If you press ‘e’ the application is exited. If you press ‘v’ the while loop is exited and the TCPIP server is started which waits for a connection from your LABVIEW application. Scrolling past the user interface while loop, you will see the TCPIP initialization. Once inside run_server() it does not return until the TCPIP connection is terminated.

Scrolling down out of main() you will see the run_server() function. First inside run_server() the function connaccept() is called. This sits and waits for a single connection from your LABVIEW application. Once LABVIEW connects to this TCPIP socket, run_server() sets up a call back function sd_signal_handler_IO() that will be called automatically when data is sent from LABVIEW to this application. Then at the bottom of run_server() notice the while loop. Here the application is continuously checking to see if any data has been sent from the OMAPL138’s DSP to Linux using shared memory. When a flag indicates that data is waiting for Linux in shared memory, this code copies the data out of the shared memory DSPSend_buf[] character array into a local array. Finally the character array is sent over TCPIP to LABVIEW. To see what data is actually being sent to LABVIEW look at the ComWithLinux() Task function in your OMAPL138 DSP code.

The next function in this C file is the call back function sd_signal_handler_IO() mentioned earlier. This function is called when data has been sent over TCPIP from LABVIEW to this Linux application. The first part of this function reads the sent data from the TCPIP socket and then searches the data for the ‘253’ start character. Once the start character is found the rest of the data is stored into a character array until the stop character is found. After finding the stop character the data is processed by simply copying the array of characters to the DSP/ARM shared memory and setting a flag indicating to the DSP to read the information copied. You will notice that we actually copy this data five times. This is done because I have not yet figured out how to correctly invalidate and write back the cache of the ARM processor like I do on the DSP processor. For the time being we will use this hack and hope that it works. (So far I have not noticed a
failure using this method). To see what is being done with the data sent from LABVIEW look at the `ComWithLinux()` Task function in your OMAPL138 DSP code.

In this DSP program, when you receive new values from Linux inside the `ComWithLinux()` Task function, make sure to communicate these values to other DSP/BIOS processes correctly. We will be covering this in lecture, but here is a quick explanation: You have to keep in mind that especially at the Task level, code is being interrupted by `Hwis` and `Swis`. Therefore when you are assigning variables at the Task level to be used at a higher priority, say a `Swi`, you need to use a “flag” variable (an `int`) to tell the `Swi` that all the data is ready for reading. So for example, we are receiving new values in the `ComWithLinux()` Task function for `vref` and `turn`. In the Task function you should use “new” global variables to transfer the data to the `Swi` function. Do the following:

```c
// Inside ComWithLinux() Task function (this is already by default in your code)
if (newnavdata == 0) { // Check if previous transfer to Swi complete
    newvref = ptrshrdmem->Floats_to_DSP[0]; //assign “new” variables
    newturn = ptrshrdmem->Floats_to_DSP[1];
    newnavdata = 1;  // set flag new data ready
}
```

```c
// Inside Swi function (this below code you will need to add)
if (newnavdata){ //check if there is new data. This is check each time into the Swi
    vref = newvref;  // copy data from “new” variables to variables used in Swi
    turn = newturn;
    newnavdata = 0; //set flag back to zero telling Task that Swi ready for next
}
```

Now that you know a little more about this Linux application, copy it to your robot’s root directory and run it. To copy from your PC to the robot type the command, “pscp <youruniqueuname> root@192.168.1.?? :”. When asked for a password, just press enter. Then go to your robot’s terminal and make sure your application is executable by typing “chmod +x <youruniqueuname>”. Also from your robot’s terminal, run and experiment with your Linux application. Press the different keys to activate the options. On your robot, change the dip switches so that the robot is printing its `vref` and `turn` value to the LCD screen. When you change `vref` and `turn` in the Linux application, you should see `vref` and `turn` changing values at the text LCD screen of the robot. Enable your robot’s motors and you should be able to control the speed and rate of turn of the wheels. If you would like, place the robot on the floor and steer it around using the application. In this case you will have to use a SSH terminal to run the application.

Take some time and add a few additional options to this Linux application to get you ready to tune your robot’s wall following algorithm in exercise 3. You will also have to study and modify the `ComWithLinux()` Task function to complete the options. Look at the file `sharedmem.h` and see that there is an array of 20 floats that you can use to communicate these new values to the DSP. Index 0 and 1 are already used for `vref` and `turn`. You will not use these below variables until you implement wall following but create these variables and display them to the robot’s text LCD screen to verify your new options are working.

Add the following variables and create a menu option that will allow you to type in a new value for each and send it to the OMAPL138 DSP:

- `ref_right_wall` Distance robot should stay to the left of a right wall when right wall following. Place the robot on the floor next to a right wall and figure out a good value for this variable using LADAR readings.
b. **front_error_threshold**  front error distance from a front wall before you stop wall-following and switch to a front wall-avoidance mode. Again place the robot on the floor about a tile (250mm) in front of a wall and figure out a good front error value for this variable in LADAR units (millimeters). You will see in Exercise 3 that the front error is defined as (3000mm - front wall distance).

c. **Kp_right_wall**  proportional gain for controlling distance of robot to wall. Start with 0.002

d. **Kp_front_wall**  proportional gain for turning robot when front wall error is high. Start with 0.001.
e. **front_turn_velocity**  velocity when the robot starts to turn to avoid a front wall. Start slow, say 0.4 tiles/second.

f. **turn_command_saturation**  maximum turn command to prevent robot from spinning quickly if error jumps too high, start with 1.0

g. Add a final menu option that will print all the values of the above variables. This option should just print the current values and not send the values to the DSP core.

**NOTE!** By default in your Linux code, the initial values of ptrshrdmem->Floats_to_DSP[0] index 0 to index 20 are zero. You will want to change these initial values to first the initial value given above. Then once you have tuned your wall-following (or as you tune your wall-following) you should update these initial values in order that they are saved for the next time the code is run. So to do this first of all set the global variables that you create for these wall-following parameters to their initial values both in your Linux program and also in your DSP code. Then also inside you Linux program scroll down to around line 120 in your source file. You should find 20 assignment statements assigning DSPFloats index 0 through 19. Above you have made the decision which DSPFloats index relates to which variable you would like to be able to adjust (or tune). Here initialize all the DSPFloats elements that you are using to the global variable’s value and leave all the unused elements at zero.

Test that your Linux program is working by printing the communicated variables to the LCD screen in your DSP program. Demonstrate it to your instructor.

When you are done modifying and testing your Linux and DSP applications ask your instructor for an Ipod Touch or use a laptop. Use the “Prompt” App as an SSH terminal and control your robot from the Ipod.

**Exercise 3: Right Wall-following**

Our next task is to use the LADAR’s distance reading (IR sensors and Ultrasonic sensors if you wish) to control the robot in such a fashion that it follows a wall on its right. By the end of lab 5 we had implemented a coupled closed-loop PI control for the speed of the robot’s wheels. That control algorithm had two input variables that you could change to make the robot speed up and turn. We defined those variables **vref** and **turn**. Our wall-following algorithm is going to control these two variables to make the robot perform as we desire.

So of the 228 distance readings the LADAR measures, which ones should you use for wall-following? You can use any of the measurements for wall following but the angles that I have tried are [111, 112, 113, 114, 115] \(\approx [-2.1^\circ, -1.05^\circ, 0^\circ, 1.05^\circ, 2.1^\circ]\). Take the minimum of these five values to get a front distance. Index [54] \(\approx [-62^\circ]\) is a good distance reading for right wall distance but again you may find different angles to work better for you and taking the minimum of a few around 54 is a good idea. You will also want to use LADAR readings that look behind the robot. To find other indexes, halt your code in CCS and put the global array **LADARangles** into a Watch Expression. Look at all of the angles to determine other indexes you may want to use. You will need to look slightly behind your robot in order to determine when to turn to the right when the wall edge has no front wall but just open space.
Initially implement a wall following algorithm that only has two situations the robot will encounter. The highest priority (i.e. the situation that should be checked for first) case is when the front distance reading detects an object within a certain distance. In this case we need to tell the robot to turn to the left until the front reading no longer detects the object. To do this we are going to define an error state \( \text{front\_wall\_error} \). \( \text{front\_wall\_error} \) will be defined as the error between the maximum distance the LADAR can detect in the course (3000mm) and its current distance reading. Then a simple proportional control law can be defined that is \( \text{turn} = K_p \text{front\_wall\_error} \). Also, set the reference speed of the robot slower, \( \text{front\_turn\_velocity} \), to allow time for the turn. These two adjustments will cause the robot to continue turning to the left until \( \text{front\_wall\_error} \) is again outside of the turning threshold.

The second situation is when no obstacles are in front of the robot and there is a wall close on its right. In this case we want to tell the robot to follow the object (or wall) seen by the right LADAR distance with a desired gap between the robot and the right object. For this situation we define an error term \( \text{right\_wall\_error} \) that is the error between the desired gap distance and the actual measured distance. Use here the proportional control law \( \text{turn} = K_p \text{right\_wall\_error} \) and set the robot’s speed to the desired forward speed \( v_{ref} \). This will servo the robot close to the right wall or obstacle. Set the desired speed, \( v_{ref} \), to 1.0 tile/second initially. Try to see how smooth of a control you can create.

Now we are ready to code the wall-following algorithm. Below is a code outline you should use in the RobotControl() function. You have already started this code in the above sections.

```c
// Add declarations for tunable global variables to include:
// ref_right_wall – desired distance from right wall, approx. 200 mm. You will have
// to figure out what distance is read by the LADAR (54) reading when the robot is
// placed approximately 200 mm from the wall.
// front_error_threshold – with front_error defined as a large front distance (3 meters is
// a large distance reading inside the square course) minus the actual front
// distance, front_error_threshold is the front_error at which the robot should
// stop wall-following and switch to a front wall left turn mode. Start with a
// front_wall_error that occurs when the robot is 1 tile (250mm) away for the front wall.
// Kp_right_wall – proportional gain for controlling distance of robot to wall,
// start with 0.002
// Kp_front_wall – proportional gain for turning robot when front wall error is high,
// start with 0.001
// front_turn_velocity – velocity when the robot starts to turn to avoid
// a front wall, use 0.4 to start
// forward_velocity – velocity when robot is right wall following, use 1.0 to start
// turn_command_saturation – maximum turn command to prevent robot from spinning quickly if
// error jumps too high, start with 1.0
// These are all ‘knobs’ to tune in lab!

// declare other globals that you will need

// inside RobotControl()

    // calculate front wall error (3000.0 - front wall distance)

    // calculate error between ref_right_wall and right wall measurement
```
if (fabs(front_wall_error) > front_error_threshold) {
    // Change turn command according to proportional feedback control on front error
    // use Kp_front_wall here...

    vref = front_turn_velocity;
} else {
    // Change turn command according to proportional feedback control on right error
    // use Kp_right_wall here

    // vref = forward_velocity
}

// Add code here to saturate the turn command so that it is not larger
// than turn_command_saturation or less than -turn_command_saturation

SetRobotOutputs(vref, turn, 0, 0, 0, 0, 0, 0, 0, 0);

Now tune your wall following. Try to make the robot follow smoothly around the course. Take advantage of your Linux program to tune the different parts of your control algorithm.

Add one more option to your wall following robot. Be able to handle the situation where the robot is following along a right wall that ends without a front wall. In this situation the robot should turn to the right until it finds a right wall to follow. This could happen at a corner that goes to the right or a wall that is just sticking out in the course. Here it will be useful to look at LADAR measurements that look behind the robot to determine when to turn to the right so the robot does not hit the wall on the right when turning.

Exercise 4: Rate Gyro

As our final task for this lab we will add the rate gyro to our mix of sensors. The rate gyro, part number LPY510AL, produces a voltage proportional to its angular velocity. This sensor has a range of +/- 400°/s with an analog output of ~0.23Volts for -400°/s and ~2.23Volts for 400°/s. The rate gyro’s signal is brought into the F28335 processor through ADCA3. An amplified version of this rate gyro signal is also brought into ADCA2. This signal has the range~0.23Volts for -100°/s and ~2.23Volts for 100°/s. We will experiment to see which of these signals work best for calculating our robot’s bearing. To scale this voltage reading to °/s or rad/s you will first subtract the sensor’s zero value and then multiply by the gain close to 400 for degrees and (π/180)*400 for radians. I say “close to” here because each sensor has a slightly different gain. You will tune this gain value later in the lab.

With our robot application we are not too interested in how fast or slow the robot is spinning. Instead, what we are more interested in is our angle or heading. With that information and the average of our wheel’s optical encoder velocity readings we can approximate the XY coordinates of the robot. To find our heading with the rate gyro we will need to integrate the angular velocity value each sample period. Below you are supplied with a shell of source code to get you started integrating this signal to find the robot’s angle.

Before you implement this code, there is a large problem with this method of finding the heading of your robot. The integrator naturally drifts due to an inexact signal produced by the rate gyro. Even a single bit (.7mV) of error from the
ADC can cause, in a short amount of time, a large error in the angle measurement. The analog signal from the rate gyro does have some noise in it causing the angle measurement to drift. You will see this drift when you implement your code. Unfortunately there is no way to totally fix this drift problem. Even the super expensive orientation sensors have noticeable drift. Some way of resetting the angle calculation every so often needs to be implemented in the system when this type of angle sensing is used. One example would be to use other sensors to find a home position for the robot that would have a known angle of orientation. This home position would have to be found every so often to reset the angle measurement. Another method that we have tried is comparing the compass reading to the rate gyro angle measurement. This works somewhat but, as we discussed earlier, the compass also has its problems. Implement the code below and then see if you can think of some ways to reduce the drift.

```c
// global variables
float gyro_zero = 0.0;  // not set here. Found below by averaging 3s of samples
// add here all other floating point variables needed.
// inside your RobotControl Swi function add
// for the first 3 seconds find the zero offset voltage for the rate gyro.
// During these first three seconds the robot should not be allowed to move.

// Then after 3 seconds have expired start calculating the angle measurement
// 1. Find the angular velocity value by first subtracting the zero offset voltage from ADCA3’s reading. (or ADCA2 if trying the amplified reading) Then multiply this value by the sensor gain given above. Use rad/seconds
// 2. Calculate the integral of this signal by using the trapezoidal method of integration. This value is your angle measurement in units of Radians.
// 3. Calculate the X, Y position of your robot using the average of the left and right wheel velocity and your bearing
// 3. Display this angle and X Y coordinates to the LCD every 100ms or so.
```

Now with your robot wall following and calculating its X, Y coordinates our last step is to tune the rate gyro’s gain. The easiest way to do this is get your LABVIEW application working plotting the position of your robot. If you programmed your LABVIEW application to leave a trail where the robot has been you can tune the gyro gain until the trail on the screen shows pretty close to a square as the robot travels around the 12 X 12 tile enclosure. With what you have learned in this lab and previous lab you should be able to figure out how to do this. Your LABVIEW application should
show the robot moving X, Y around the enclosure with a line indicating the robot’s speed and direction. The LABVIEW application should also have a text box that will allow you to enter in a new value for the gyro’s gain and download it to your robot. To send data to and from LABVIEW, you have all the code needed in your Linux program. What you will need to change is the ComWithLinux() Task function some to upload and download the correct items. You need to change the code to the items you would like to communicate.

**Lab Check Off:** Show your TA the final LABVIEW program as your robot follows the wall. Use the rate gyro measurement to determine the orientation of your robot. Upload this data to the LABVIEW program and update the robot’s position on a XY grid. Your robot should run smoothly as it follows the wall inside the course. You will need to adjust your wall-following gains if your robot is swaying or jerking a lot.