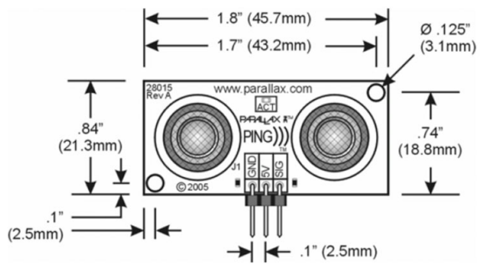
Lab 6 Network and Robot Interfaces

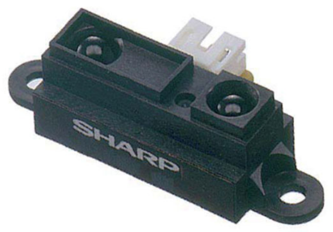
Goals	• Develop a layered communication system, • Design and implement a hardware/software interconnect protocol. • Interface two sensors and one motor needed for the robot. • Use communication skills to work effectively as team.
Review	• Textbook Chapter 8 and Sections 9.1 through 9.4 • Chapter 17 of the TM4C123 data sheet explaining basic extended CAN • ESP8266 Wifi module documentation on the class website • IR distance sensor http://www.ece.utexas.edu/~valvano/Datasheets/gp2y0a21yk.pdf • **Ping)))** sensor http://www.ece.utexas.edu/~valvano/Datasheets/PingDocs.pdf http://www.ece.utexas.edu/~valvano/Datasheets/PingAN.pdf • **HC-SR04** sensor http://www.ece.utexas.edu/~valvano/Datasheets/HCSR04b.pdf • **OPT3101** time-of-flight sensor https://www.ti.com/product/OPT3101 • CAN reference: http://www.kvaser.com/can/protocol/index.htm
Starter files	• **`RTOS_Lab6_Networking`** (starter project for Lab 6) – **`can0.c`** and **`can0.h`** containing CAN0 driver (in **RTOS_Labs_common** folder, from **`CAN_4C123`** starter project) – **`esp8266.c`** and **`esp8266.h`** containing ESP8266 Wifi module driver on UART2 (in **RTOS_Labs_common** folder). – **`WifiSettings.h`** containing Wifi access point settings for the ESP8266 module (in **RTOS_Labs_common** folder). – **`Lab6.c`** with the bare-bones OS and Testmains. • **`RTOS_SensorTestProject`** project (sensor examples) • **`RTOS_MotorTestProject`** project (motor examples) • **`OPT3101_4C123`** project (time-of-flight sensor driver) • Sensor and motor board schematics/PCB layouts (**`Robot_Sensor.*`** and **`Robot_Motor.*`**)

This lab must be performed in teams of 3 to 5 students, which will also be the Lab 7 robot competition teams. Approval of the team by the TA is required before the preparation is started. Only one version of the software need be developed for each team.

Background

Lab 6 has two or three sensors distributed across two computers. Each computer collects data from one or more sensor and shares the data with other computer using the CAN. For Lab 6 you will use the USB supply for +5V, but in Lab 7 the +5V will come from linear regulators powered from the battery. The first sensor to interface is the **HC-SR04** or **Ping)))**. This is an ultrasound transducer used to measure distance. You will use this sensor with input capture to measure the distance to a wall or to another robot. In Lab 6, you will interface one sensor, but your robot may have up to three ultrasound sensrs.





The Sharp **GP2Y0A21YK** is an IR sensor, also used to measure distance. Similar to ultrasound sensors, the goal is to measure the distance to a wall or to another robot. The sensor has an analog output inversely related to distance. The difficulty with **GP2Y0A21YK** sensors is that they are noisy. There are analog low pass filters on the sensor board. If you add digital filters it will create a more stable and reliable measurement. Your system will also require some software calibration and calculations to implement a quantitative measure of distance. The Lab 7 robot will have up to four of these sensors, but only one will be used in Lab 6.

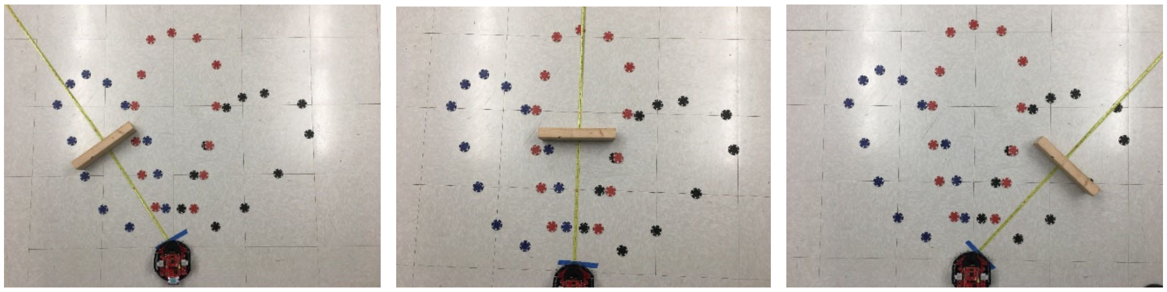
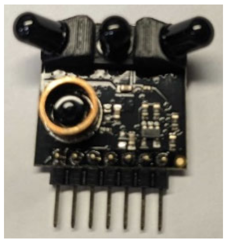


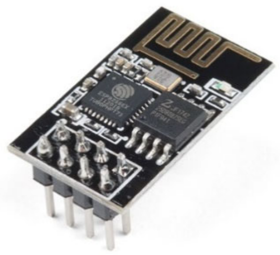
Figure 6.1. The fields of view (FoVs) for the OPT3101 time of flight sensor.



The third type of distance sensor available will be an IR-based, digital time-of flight (ToF) sensor based on the **OPT3101** chip from Texas Instruments. The board using the **OPT3101** was created by Pololu and contains an effective mounting shield allowing three simultaneous distance measurements in three different directions. Figure 6.1 shows the effective fields of view (FoVs) of the sensor (the square tiles are 12 inches across). Your robot will be able to see both walls if the **OPT3101** system is mounted on the front of the robot. The sensor is interfaced to the microcontroller via a digital I^{2}C bus connection. Similar to the other sensors, it will require filtering and calibration. In Lab 6, you will interface one **OPT3101**. The robot in Lab 7 has four unoccupied I^{2}C ports (two on the sensor and two on the motor board) that support up to four sensors, but one sensor should be sufficient.

In Lab 6 you will have a third microcontroller separate from the two microcontrollers used in the CAN network of sensor boards and interfaces. This third microcontroller will spin one DC motor in both directions in an open-loop fashion. In Lab 6 the DC motor should be powered from an 11.1V bench supply. This third microcontroller will send PWM signals to the motor driver circuit. In Lab 6 this third microcontroller should be able to adjust the duty cycle of the PWM output so the software can make the motor spin fast or slow. The motor driver circuits will be provided on a separate motor PCB. In Lab 7, you will connect two motors to the robot's motor board and power it from the battery. However in Lab 6, you are just running one motor in open-loop fashion.

The motor board and corresponding microcontroller will also control a servo for the steering of your robot. Servos are a popular mechanism to implement steering in robotics. Ranging from micro servos with 15oz-in torque to powerful heavy-duty sailboat servos, they all share several common characteristics. A servo is essentially a motor for which you set the desired position or angle. The servo "knows" two things: where it is (the actual position) and where it wants to be (your desired position). When the servo receives a desired position, it attempts to move the servo horn to the desired position. The task of the servo, then, is to make the actual position equal to the desired position. For more information on servos, search the web site: http://www.brookshiresoftware.com/



As part of Lab 6, you will also have the option to connect your robot to Wifi and enable remote wireless access, e.g. from your laptop. Launchpads can be connected to a Wifi network (and the internet) using **ESP8266** Wifi modules. Specifically, we will use ESP-01S Wifi modules made by Ai-Thinker, which incorporate a UART interface, radio frontend and antenna around an ESP8266 chip by Espressif. The ESP8266 chip implements a complete TCP/IP network stack (in firmware running on an embedded microcontroller) on top of a basic Wifi hardware interface. The Sensorboard has a socket that connects an ESP-01 module to UART2 on the TM4C, including a dedicated voltage regulator to drive ESP power with higher current needs. All communication with the ESP8266 from the TM4C will be via UART2 using the ESP8266's 'AT' command syntax. The **`esp8266.c`**/**`.h`** driver implements a basic set of commands and provides a high-level API to interact with the ESP module. However, you will need to integrate the **`esp8266.c`**/**`.h`** driver with your OS. Specifically, the driver internally uses three FIFOs for UART2 communication with the ESP. Similar to UART0 in Lab 2, busy-waiting FIFOs have to be replaced with FIFOs that use semaphores for synchronization between background interrupt handlers and foreground threads.

To connect the ESP8266 to a Wifi network, you need to provide the access point SSID and password/key in **`WifiSettings.h`**. The **`esp8266.c/.h`** driver provides APIs to connect the ESP to the access point, which will assign it an IP address (via DHCP). From there on, the driver provides routines for the ESP to act as a TCP client or server: an ESP can either initiate TCP client connections to servers on the internet using their IP address or name (the ESP will internally perform DNS lookups if necessary) and port, or the ESP can be switched into server mode where it will listen for incoming connections on a specific port. The starter project includes examples for the Launchpad to fetch weather information from the internet as a TCP client sending HTTP requests to a public web server (*Testmain2*) or to act as a simple web server itself that understands basic HTTP 'GET' requests and serves an HTML page with a form that users can submit via a web browser to output on the Launchpad's display (*Testmain3*).

For Lab 6, you can optimally add capabilities to your OS to access its command interpreter remotely through Wifi and the internet. If you chose to do so, you will need to add a remote terminal server thread that will listen for incoming connections on a chosen port (e.g., the default port 23 for remote terminal/telnet servers) and will launch an instance of your interpreter that communicates with a remote terminal via the ESP whenever a new TCP connection to the server is made. This interpreter instance should run concurrently to the default interpreter on UART0. In order to do so, you must have a version of the interpreter that is able to receive inputs and send outputs via the TCP connection instead of UART0. One option is to create a copy of your interpreter in which all UART I/O is replaced with corresponding **`ESP8266_Send`**/**`ESP8266_Receive`** calls. Alternatively, you can modify your interpreter such that it allows I/O redirection to either UART0 or ESP. There are different ways of doing this. You can overload all I/O functions in the interpreter with wrappers that call corresponding UART0 or ESP routines depending on a parameter passed to the interpreter, and then create two wrapper tasks that launch interpreter instances with different parameters. If you want to achieve the same without having to pass parameters into every interpreter function, I/O redirection can be done depending on the context, i.e. by which thread ID (**`OS_Id()`**) the **`Interpreter()`** routine is called. For any I/O routines used by the interpreter, overload them with a wrapper like this:

```
int TelnetServerID = -1;
void Interpreter_OutString(char *s) {
  if(OS_Id() == TelnetServerID) {
    if(!ESP8266_Send(s)) { // Error handling, close and kill }
  } else {
    UART_OutString(s);
  }
}
```

The default interpreter instance on UART0 can then be launched as before. For remote terminals, create a *TelnetTask* that is launched on every incoming server connection, sets the server ID for redirection and launches an instance of the interpreter with I/O redirected:

```
void TelnetTask(void) {
  TelnetServerID = OS_Id();
  Interpreter();
  ESP8266_CloseTCPConnection();
  OS_Kill();
}
```

In all cases, you should add a command that allows exiting the interpreter, which will then close the TCP connection. For the default interpreter on UART0, exiting can be ignored or the **`Interpreter()`** can simply be respawned in an endless loop.

It is recommended to break down the implementation work for this lab among the four students in the team as outlined below. However, all students in the team must understand all aspects of the lab and be able to answer questions during checkout. If there are 5 students on the team, you can have different students design the IR and ToF sensor interfaces, or add a 4th type of sensor. The data from all sensors is displayed on the LCD. The CAN interface must be used, but Wifi interfacing is optional. Interrupts, FIFO queues, semaphores, and the OS must be used in an appropriate manner. The hardware/software system is modular, because in Lab 7 you will create up to four copies of the IR distance sensor, up to four copies of the **HC-SR04** or **Ping)))**, and two copies of the motor output driver. You will integrate these components on the two microcontrollers in Lab 7.

There are background tasks on each node: data acquisition and CAN I/O (and optionally a terminal server via Wifi). The data acquisition threads will measure parameters (Ping, IR, ToF or touch sensor) and send its measurement to a companion foreground thread (like Labs 2 and 3). A periodic task will start the **HC-SR04** or **Ping)))** measurement every 100 ms. The time delay will be measured with an input capture interrupt. A periodic task will sample the ADC measuring distance with the **GP2Y0A21YK** sensor. Another periodic task will perform regular **OPT3101** distance measurements. Make sure the execution times of the background threads are short and bounded. A foreground thread will manage the CAN I/O and the LCD display. The CAN identifier will specify the data type. The data field can be formatted however you wish. The CAN receiver thread runs in the background, accepting messages from the other nodes. Since the CAN is dedicated, you can use CAN filters however you wish. As always, you are allowed to implement the system in an alternate manner, as long as the basic educational objectives of the lab are achieved. The appropriate use of semaphores and FIFO queues are expected.

Choose a CAN channel bandwidth between 500,000 and 1,000,000 bps, and leave it as a constant. Also leave the time-quanta settings as shown in the CAN starter files. To change data bandwidth, you will adjust the ADC sampling rate for the **GP2Y0A21YK** IR sensor. Start with a data acquisition/transmission rate so slow (e.g., 500/sec) that collisions will be rare, and all the FIFOs are usually empty. Then, keeping the CAN channel bandwidth fixed, increase the sampling rate (number of ADC samples/sec and number of CAN transmissions/sec) until the maximum bandwidth is reached, without loss of data. It is possible to implement this network using just the **Data Frame**, ignoring the **Remote Frame**, the **Error Frame**, and the **Overload Frame**. Similarly, you can implement just the **Standard CAN 2.0A**, instead of the **Extended CAN 2.0B**. CAN does force a priority structure on the network, but you can implement priority however you wish.

Preparation

1A) Download, unzip, and compile the starter project, and integrate the low-level CAN drivers into your Lab 3, Lab 4 or Lab 5 OS. Add semaphore calls to the CAN routines in appropriate places. Include any Lab 4 or Lab 5 software you might which to include in your robot. The starter project includes a CAN application (*Testmain1*) that sends 4-byte messages. You may wish to increase the message length to 8 bytes.

1B) Optionally, integrate Wifi support and the **`esp8266.c/.h`** driver with your OS. In particular, replace the UART2 FIFOs with a semaphore-based implementation. Make sure to adjust **`WifiSettings.h`** and set SSID_NAME and PASSKEY to the Wifi network name and password/key of the desired access point. See Hint 1) below if you want to connect your Launchpad on campus. Run the *Testmain2* and *Testmain3* from the starter project. Study their code to make sure you understand their operation and use of the ESP8266 driver API. Debug output including and echoing of 'AT' commands exchanged with the ESP is shown in the Launchpad's UART0 terminal. In particular, write down the IP addressed assigned to your Launchpad (output of the 'AT+CIFSR' command). For *Testmain3*, you can interact with the web server on the Launchpad by opening the address http://<*Launchpad_IP*> in any web browser (tested with Firefox and Chrome).

2) Assign sensor and motor interfaces, and study the I/O pin assignments for your Lab 7 robot. Overall, you can use up to four input captures for the four **HC-SR04** or **Ping)))**, four ADC channels (for the four **GP2Y0A21YK** IR sensors), four I^2C interfaces (for **OPT3101** laser-ranging sensors), four digital outputs for the two DC motors two of which should be PWM, one PWM output for the servo, and some inputs for bumper switches or mode configurations. In addition, you should keep the UART interface for the interpreter, and the LCD to monitor robot status. The sensors, motors and servo must be interfaced using two microcontrollers and boards (one sensor and one motor board). You must be friendly and document your interfaces well. It is easier to design for the Lab 7 integrated sensor system now, than it will be to move a sensor from one interface to another later.

3A) One student in the group designs the interface for the **HC-SR04** or **Ping)))**) and for the bumper switches. One **SIG** pin is used for both output and input of the **Ping)))**) sensor (see Figure 6.2). The fundamental approach will be to: 1) make the **SIG** pin an output; 2) issue a 5 µs output pulse (which causes a sound pulse to occur); 3) switch the **SIG** pin to back to an input; and 4) measure the time until the echo is received. The **HCSR04** sensor is a little easier to interface because it has a separate input and output pin. $\mathbf{t_{IN}}$ will be a pulse width measurement using input capture. The pulse width time is equal to twice the distance to the object divided by the speed of sound. The distance **d** is therefore $\mathbf{d}=\mathbf{c}^*\mathbf{t_{IN}}/2$, where **c** is the speed of sound, and $\mathbf{t_{IN}}$ is the time for the sound to travel to the object, reflect and travel back to

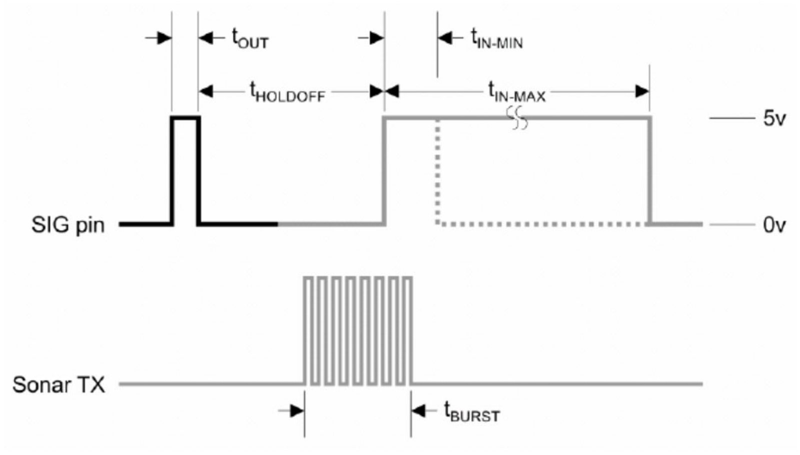


Figure 6.2. Ping))) sensor signals. See PingDocs.pdf for an explanation of this figure.

the sensor. You will perform this measurement about 10 times per second. Using the appropriate OS commands, write foreground and background software that samples distance approximately 10 times a second and sends the data to other node on the system using CAN. You might want to disable interrupts during the 5µs pulse (**t_{OUT}**) if you are using blind cycle counting for the time delay, so the OS does not suspend the thread during the pulse output.

3B) A second student in the group designs the interface for one **GP2Y0A21YK** IR and one **OPT3101** ToF distance sensor. The analog output signal of the **GP2Y0A21YK** has two types of noise. Notice the huge noise spikes in Figure 6.3 occurring about every one ms. On the spectrum analyzer you will also see both white noise and periodic EM field noise. During the procedure part of this lab you measure the noise of your system using a spectrum analyzer. The sensor board provides an analog low pass filter (LPF) with gain=1 for IR sensor inputs. The wheel diameter is about 8 cm, and the motors will spin up to 120 RPM, giving a maximum speed of about 50 cm/sec. Let x(t) be the distance to the wall as the robot travels at 50 cm/sec. Using the transfer function of the transducer, estimate the maximum slew rate, **s**, in volts/sec needed from the sensor interface. If the input were to be V=**A**sin(2πft), then the maximum slew rate dV/dt is 2πfA. Assuming **A** to be about 1 V, if the maximum slew rate is **s**, the maximum frequency response needed is s/2π. The cutoff frequency of the analog LPFs should be about 10*(s/2π)[1]. Using the appropriate OS commands, write foreground and background software that samples distance approximately 2 times the LPF cutoff frequency and sends the data to the other computer on the system using CAN. Figure 6.4 plots data collected with the **GP2Y0A21YK** IR sensor employing the 2-pole 20 Hz Bessel analog LPF front-end realized on our sensor boards. Implement an additional 3-wide median digital filter to remove the noise spikes. Another nonlinear filter could be used in place of the median if you want.

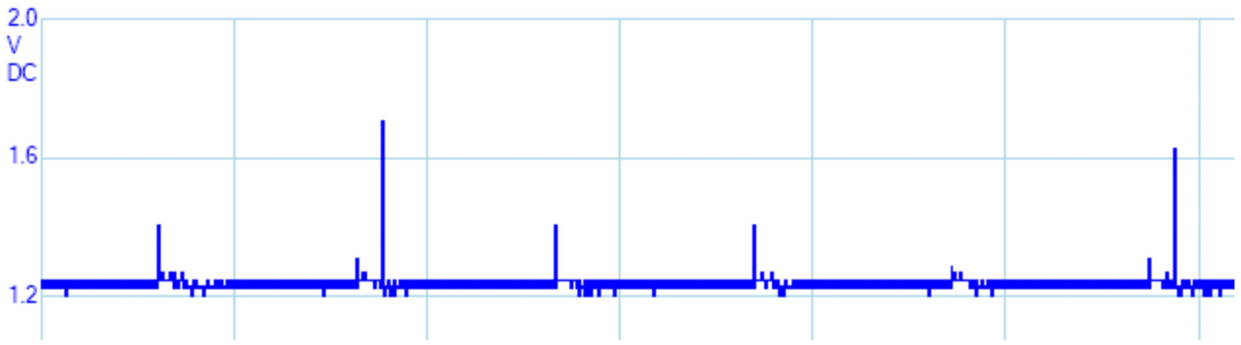


Figure 6.3. GPY0A21YK sensor output, distance=20 cm, DC mode, voltage scale is 0.4 V and time scale is 1 ms.

[1] The cutoff of the analog 2-pole Butterworth LPFs on the sensor board is 50 Hz.

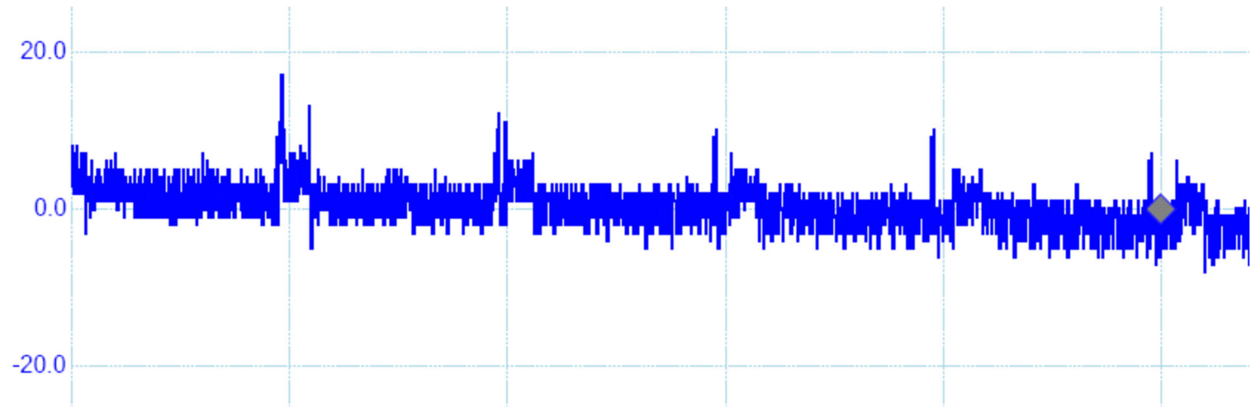


Figure 6.4 GPY0A21YK filtered output, distance=20 cm, AC mode, voltage scale is mV and time scale is 1 ms.

For the **OPT3101**, interface the sensor to any of the available and not otherwise occupied I^{2}C ports using the provided driver. The driver uses the I2C0 I^{2}C port by default, but can be configured to use a different port if needed. The **OPT3101** driver supports polling- or interrupt-based measurements, but you need to integrate the driver with your OS and use appropriate OS features (background periodic or foreground with sleeping). Write software that samples distance approximately 10 times a second and sends the data to the other CAN node.

3C) A third student in the group designs the interface for one **DC motor** and the servo. The motor interface must allow for the software to control direction and power. Controlling direction will be handled via the H-bridges on the motor board, and controlling power will require PWM output from the microcontroller. This motor interface need not be integrated into the other components in Lab 6 (integration will occur in Lab 7).

http://youtu.be/p4QevDMCuKo

Figure 6.5. Measurements of motor voltage. The DC motor is controlled via PWM.

For the servos, the first step to understanding how servos work is to understand how to control them. All timing and electrical characteristics described here have been experimentally determined from a "HS-303 HiTec" servo (Figure 6.6). The servo is controlled by three wires: ground (black), power (red), and command (yellow). Power is usually between 4V and 6V and should be separate from system power (as servos are electrically noisy). Even small servos can draw over an amp under heavy load so the power supply should be appropriately rated. Though not recommended, servos may be driven to higher voltages to improve torque and speed characteristics. Servos are commanded through PWM signals sent through the command wire. Essentially, the width of a pulse defines the

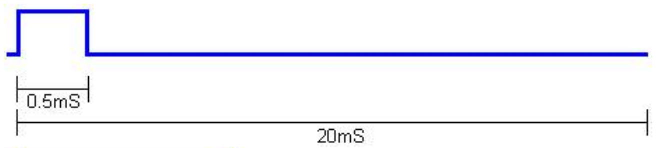
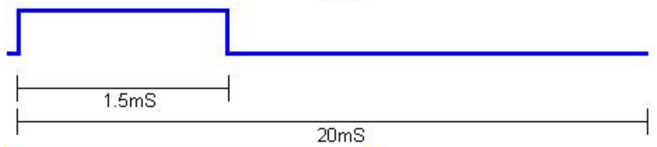
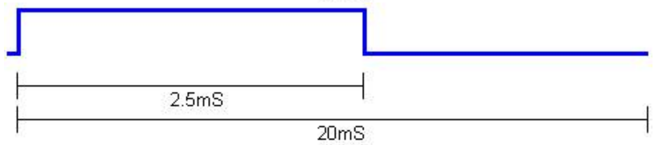
Position	**Pulse Width**	**Example Pulse**
Minimum	0.5ms	
Center	1.5ms	
Maximum	2.5ms	

*Figure 6.6. The timing constraints of the **HS-303** servo (http://www.brookshiresoftware.com/) not drawn to scale.*

position. For example, sending a 1.5ms pulse to the servo, tells the servo that the desired position is 90 degrees. In order for the servo to hold this position, the command must be sent at about 50 Hz, or every 20 ms.

If you were to send a pulse longer than 2.5 ms or shorter than 0.5 ms, the servo would attempt to overdrive and damage itself. Once the servo has received the desired position (via the PWM signal) the servo must attempt to match the desired and actual positions. It does this by turning a small, geared motor left or right. If, for example, the desired position is less than the actual position, the servo will turn to the left. On the other hand, if the desired position is greater than the actual position, the servo will turn to the right. In this manner, the servo "zeros-in" on the correct position. Should a load force the servo horn to the right or left, the servo will attempt to compensate. Note that there is no control mechanism for the speed of movement and, for most servos, the speed is specified in degrees/second. Indeed, one of the primary tasks of electronics is to synthesize servo speed control by stepping through a series of positions. The servo outputs on the motor board are powered by an additional 7805 voltage regulator, creating a separate +5V source just for the servo. If you connect the servo to the +5V used by the microcontroller, then the servo will cause transient power losses to the microcontroller, causing software resets.

3D) Finally, a fourth student in the group designs the CAN and optional Wifi network interfaces. For the CAN, write background/foreground threads that accept CAN messages from the network. The foreground thread displays the most recent data from all nodes, including itself. Count the number of times a packet is lost (i.e., the CAN receiving ISR calls **`CanFifo_Put`** and the FIFO is full. For Wifi networking and remote robot access, write a server thread that accepts incoming telnet connections over Wifi and launches a dedicated remote Interpreter thread for each connection, where the Interpreter thread acts as a remote terminal communicating over Wifi.

Procedure

1) Modify your Lab 3, Lab 4 or Lab 5 OS to run on the two microcontrollers.

2A) Implement and test the CAN physical layer of the network. There are terminating resistors at each end of the cable. These resistors eliminate reflections and limit the rise/fall times of the voltages on the network. To test the network hardware, you can run one of the StellarisWare/Valvano starter examples. Next you will debug your integration of the CAN programs into the two computers.

2B) Optionally implement, test and debug the remote terminal (telnet) server for your Launchpad. Test the server by opening a TCP connection to the Launchpad's IP on the chosen port from a laptop or PC on the same local network, and then issuing commands to the remote interpreter:

`telnet` *<Launchpad_IP> <port>* or **`nc`** *<Launchpad_IP> <port>* (or using Putty)

If client and server are on different (home) networks, make sure that the server is reachable from the client's network, if necessary using port forwarding on the server's router or via SSH tunnels (as described under Hints below). Test the remote connection from your laptop/PC by opening a TCP connection to the remote Launchpad and issuing remote interpreter commands.

3) One by one, implement, test and debug the sensor and motor interfaces. Please verify proper operation of the circuit before testing using software on the microcontroller. Also remember to turn off both 3.3 and 5 V power when moving parts and wires on your circuit. Double check the power connections every time before applying power.

4A) Using known distances, calibrate the **HC-SR04** or **Ping)))** so it measures distance accurately. The **maximum deviation** (or **span**) is the difference between maximum and minimum data points given a fixed distance. The **range** is the minimum and maximum distances at which the measurements can be accurately obtained. Write software that measures the average, standard deviation and maximum deviation of 10 consecutive measurements (one second). The walls of the race track will be pieces of 3.5 by 3.5 inch cedar wood. The location will be the outdoor ECJ plaza (with the ETC T-Room/Student Lounge as indoor backup location in case of rain). The 19 cm wide robot will travel down a race track about 80 cm wide. Envision trying to speed your robot down the race track keeping a constant distance to the right wall. At what heights from the floor would it work? At what angles to the wall would it work? In order to determine the range, accuracy and noise level, take measurements (average, standard deviation and maximum deviation) at 5 known distances while detecting the race wall at outdoor and indoor locations. Calculate accuracy as the average absolute value of the differences between truth and measured. Standard deviation and maximum deviation are measures of noise.

Truth d_T	Measured d_M	Standard Deviation	Span

$$\text{Average accuracy (in cm)} = \frac{1}{5}\sum_{i=0}^{4}\left|d_{Ti} - d_{Mi}\right|$$

4B) Using known distances, calibrate the **GP2Y0A21YK** and **OPT3101** sensors so they measure distance accurately. Write software that measures the average, standard deviation and maximum deviation of consecutive measurements over a one second time period. Look at the data sheet to choose the optimal orientation of the sensor to detect the wall while the robot is moving. Using the same scenario described in procedure 4A, at what heights from the floor would it work? At what angles to the wall would it work? In order to determine the range, accuracy and noise level, take measurements (average, standard deviation and maximum deviation) at 5 known distances while detecting the race wall at its possible outdoor and indoor locations. Similar to Ping))), determine accuracy, range and noise for this sensor. Given a constant distance, measure the **GP2Y0A21YK** noise at the output of the LPF with a spectrum analyzer.

4C) Implement, test and debug the motor interfaces. Measure the current and voltage delivered to the motor under a no-load condition. Once the wheel is placed on the motor shaft and the robot placed on the ground, the current will increase 4 to 10 times this amount. You will not be able to increase the voltage above the 11.1V from the battery, so the design of the H-bridge will significantly affect the speed of your robot.

4D) Implement, test and debug the servo operation for setting and holding different horn positions via software.

5) Put all the pieces together so data measured in each node is available in both computers. The file system is not needed for Lab 6. Measure a typical CAN frame with a dual trace scope showing both the output of the microcontroller and CANH on the actual bus.

6) Measure the maximum sustained bandwidth of the system, under the conditions that virtually no data is lost. Keep the CAN channel speed fixed at 500,000 (or 1,000,000) bps and increase the sampling rate of the **GP2Y0A21YK** sensor, while measuring the number of lost packets. "Sustained" means the measurement should occur over many seconds, so that the steady state behavior is measured, and not a start-up behavior measured when all the FIFOs are initially empty. Which factor limits the bandwidth? Consider factors such as the speed of the CAN channel, the speed of the ADC converter, execution speeds of various background threads, execution speed of foreground program, or the LCD display speed.

Checkout (show this to the TA)

Show the sensors, motors and servo in operation creating scope tracings similar to Figures 6.2 to 6.6. Be prepared to discuss factors such as accuracy, noise, and reproducibility. Connect a logic analyzer to CAN signals PE4 and PE5. Explain these two signals as data are communicated across the network.

Deliverables (exact components of the lab report and lab submission)

A) Objectives (1/2 page maximum)
B) Hardware Design
  1) List of interface and pin assignments of the sensors and motors used in your system (Preparation 3)
C) Software Design (documentation and code)
  1) CAN module including its FIFO
  2) Software to implement the sensor measurements
  3) Software to implement the distributed data acquisition
D) Measurement Data
  1) Table of **HC-SR04** or **Ping)))** data at 5 known distances and average accuracy (Procedure 4A)
  2) Table of **GP2Y0A21YK** and **OPT3101** data at 5 known distances and average accuracy (Procedure 4B)
  3) Spectrum of **GP2Y0A21YK** noise (Procedure 4B)

4) Motor current and voltage under no-load conditions (Procedure 4C)
5) Scope traces of CAN signals measured on both sides of the MCP2551 (Procedure 5)
6) Network bandwidth (Procedure 6)

E) Analysis and Discussion (2 page maximum). In particular, answer these questions
1) What is one advantage of the **HC-SR04** or **Ping)))** sensor over the **GP2Y0A21YK** sensor?
2) What is one advantage of the **GP2Y0A21YK** sensor over the **HC-SR04** or **Ping)))** sensor?
3) What is one advantage of the **OPT3101** sensor over the others, and vice versa?
4) Describe the noise of the **GP2Y0A21YK** when measured with a spectrum analyzer.
5) Why did you choose the digital filters for your sensors? What is the time constant for this filter? I.e., if there is a step change in input, how long until your output changes to at least 1/e of the final value?
6) Present an alternative design for an H-bridge and describe how your H-bridge is better or worse?
7) Give the single-most important factor in determining the maximum bandwidth on this distributed system. Give the second-most important factor. Justify your answers.

F) Post-mortem concerning team member interactions (attached to the report)
1) Each team member evaluates each other team member including oneself
    - Simply list one or two weaknesses.
    - Simply list two or three strength characteristics.
2) Major failures in the way the team interacted (if any)
3) Major successes in the way the team interacted

G) Peer Review (each student submits independently and confidentially directly to the TA)
1) Classify each team member including oneself as:
    - worked harder than average (explain), worked an average amount, worked less than average (explain)

Hints

1) You will need to desolder R9 and R10 on your Launchpad to use all the sensor and motor board functionality. The resistors connect pins PD0/PD1 and PB6/PB7 to support TM4C self-test. They tie up those ports, which are needed for ultrasound J9 and IR sensors J7/J8 on the sensor board, and for one of the motors and the servo on the motor board.

2) You will not be able to use the time-of-flight sensor (in its default configuration) at the same time as an ultrasound sensor on connector J11. They both use pins PB2 and PB3 in either I2C0 or Timer 3 alternate functions.

3) The ESP8266 driver currently only supports one active client (outgoing) or server (incoming) connection at a time, i.e. no two simultaneous outgoing connections as a client, no outgoing client connection while some incoming server connection is active (in practice, one should be a client or server at any time, but can switch between at runtime), and no two incoming server connections at the same time (if a client connects to the server while another connection is already active, the second connection will simply be ignored and not served).

4) If you want to connect your Launchpad to Wifi on campus, you need to use the "utexas-iot" network. To get a passkey for this network, you will need to register your Launchpad's MAC address with UT's Information Technology Services (ITS) at https://network.utexas.edu. Don't forget to unregister your Launchpad if you ever sell or get rid of it, or else you will be on the hook for anybody abusing the network with your old device.

5) To connect a laptop/PC and a Launchpad that are on different local networks and are not directly reachable from each other, you will need to setup SSH tunneling using the following steps. This works by setting up two tunnels, one each on the client and server side that route and connect everything through an ECE LRC machine as follows:

```
Laptop1 -- ssh -L --> LRC <-- ssh -R -- Laptop2
Laptop1 == tunnel ==> LRC == tunnel ==> Laptop2 --> Launchpad
(Client)                                           (Server)
```

First, setup the server side forwarding:
1. On the server Launchpad:
    a) Start the server code and note the Launchpad's local <*server_IP*> and <*server_port*>
2. On a local laptop/PC that is on the same local network as the server Launchpad:
    a) Test the local connection to the server Launchpad:
        `telnet` <*server_IP*> <*server_port*> or **`nc`** <*server_IP*> <*server_port*> (or using Putty)

b) If you are outside of the UT campus, start the UT VPN to be able to SSH into the ECE LRC machines.
c) SSH into an LRC machine (e.g.' kamek') using remote port forwarding, i.e. setting up an *<LRC_port>* on the LRC machine that will listen for and forward any incoming connections to the Launchpad:

   `ssh -R` *<LRC_port>*`:`*<server_IP>*`:`<server_port> **`kamek.ece.utexas.edu`**

   Or, in Putty, setup an SSH connection to the LRC machine with SSH remote tunnel options as in shown in Figure 6.7(a) below. Pick a random *<LRC_port>* above 1024. If you get an error that the port is already in use (already "bound"), use a different port.
d) From within the SSH session/terminal on the LRC machine, test the forwarding to the server Launchpad:

   `nc localhost` *<LRC_port>*

   This should open a session/connection to the server on your Launchpad.

Then setup the client side forwarding:

3. On the client laptop/PC:
   a) If you are outside of the UT campus, start the UT VPN to be able to SSH into the ECE LRC machines.
   b) SSH into the same LRC machine as above using local port forwarding, i.e. setting up a *<local_port>* on the laptop/PC that will be forwarding any incoming connection to the *<LRC_port>* on the LRC machine, which in turn will forward the connection to the server Launchpad through the other tunnel:

      `ssh -L` `*:`*<local_port>***`:localhost:`***<LRC_port>* **`kamek.ece.utexas.edu`**

      Or, in Putty, setup an SSH connection to the LRC machine with SSH local tunnel options as shown in Figure 6.7(b). *<local_port>* can be any port available on the laptop/PC, e.g. the same as *<server_port>*
   c) Make sure that the firewall on the laptop/PC allows incoming connections from the local network to *<local_port>*. For Windows, approve any such requests that pop up when starting the SSH session.
   d) Open a command prompt/window and get the laptops/PC's *<local_IP>* address on the local network:

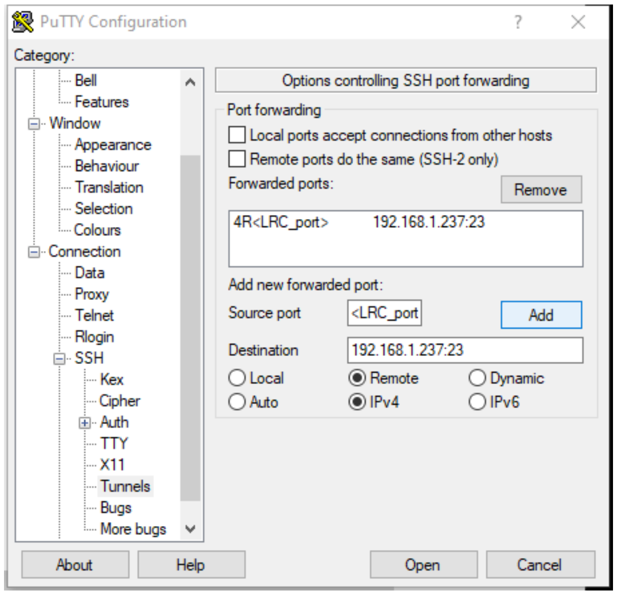
      `ipconfig`
   e) Test the connection to the remote server Launchpad via the two tunnels:

      `telnet localhost` *<local_port>* or **`nc localhost`** *<local_port>* (or using Putty)

      This should open a session/connection to the server on the remote Launchpad.
4. If you want to connect from a second local computer that is on the same network as the client laptop/PC:
   a) Test the connection through the forwarding laptop/PC:

      `telnet` *<local_IP> <local_port>* or **`nc`** *<local_IP> <local_port>* (or using Putty)

      If this fails but the equivalent test 3e) worked, then the firewall on the forwarding laptop/PC is interfering.



Options controlling SSH port forwarding

Port forwarding

☐ Local ports accept connections from other hosts
☐ Remote ports do the same (SSH-2 only)

Forwarded ports:

4R<LRC_port> 192.168.1.237:23

Add new forwarded port:

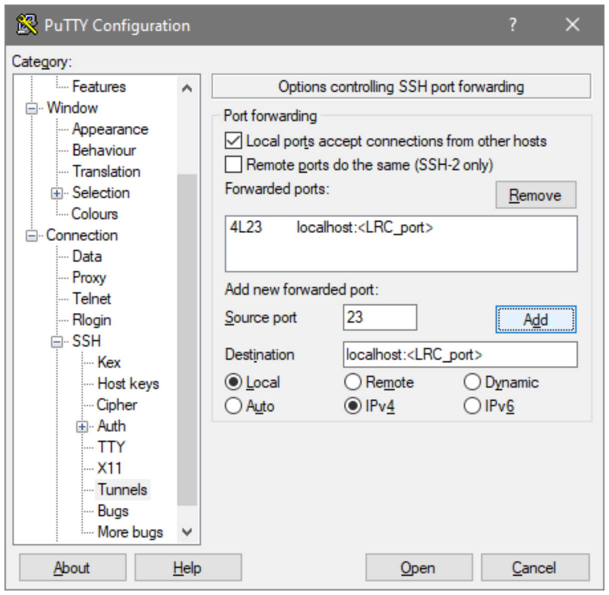
Source port: <LRC_port

Destination: 192.168.1.237:23

Local [✓ Remote] Dynamic

Auto [✓ IPv4] IPv6

(a) Server side



Options controlling SSH port forwarding

Port forwarding

☑ Local ports accept connections from other hosts
☐ Remote ports do the same (SSH-2 only)

Forwarded ports:

4L23 localhost:<LRC_port>

Add new forwarded port:

Source port: 23

Destination: localhost:<LRC_port>

[✓ Local] Remote Dynamic

Auto [✓ IPv4] IPv6

(b) Client side

Figure 6.7: Putty SSH tunnel configurations for port forwarding (using 192.168.1.237 as <server_IP> and port 23 as both <server_port> and <local_port> in this example).