ELE 302: Building Real Systems

Professors: Andrew A. Houck and Antoine Kahn

Independent Project: Ultrasonic Positioning System

Michael Danielczuk, Andrew Kim, Monica Lu, and Victor Ying

Due Friday, May 15, 2015

Contents

1 Introduction

Our indoor positioning system works by the same principle as a satellite navigation system, with ultrasonic pings substituted for radio waves. Signals are transmitted from stations at known locations, and signals traveling different distances to a given receiver will take different amounts of time to get there. With enough information about distances to known locations, the receiver's position can be calculated. As the speed of sound at room temperature is a mere 1126 feet per second, it is easy for general-purpose microcontrollers to measure the timing of ultrasonic signals to a precision that would enable spatial resolution of better than one foot. Our 25 kHz ultrasound has a wavelength of about half an inch, so if our receive circuitry allows for timing of ultrasonic signals to within several cycles, we might expect to achieve spatial resolution of a few inches. In fact, this is exactly what we observe.

The next three sections of this report document the design of the positioning system at three levels of abstraction. First, at the highest level of abstraction, there is Section 2, which explains the principles and mathematics of positioning used, assuming the ability to measure differences in time of flight. The other two sections concern the hardware and software systems needed to obtain these measurements. Section 3 discusses the physical hardware systems we had to build, while Section 4 discusses the way the various components of the system communicate and the signals and processing that are needed to get accurate measurements of differences in times of flight.

2 Multilateration

A positioning system generally either relies upon the ability to either measure different distances to known points (multilateration), or different angles to known points (multiangulation). Multiangulation could be done by use of cameras or other angle-resolved sensors, but using cameras located on mobile robots such as our cars would be difficult as each car would need to process probably several different video feeds covering a 360 field of view and correctly identify beacons at known locations, and transform the apparent position of beacons in the video frames to the angles in which the beacons are oriented. Additionally, putting a bunch of cameras on every robot would be somewhat expensive. If doing multiangulation, it may make more sense to have a fixed number of cameras at known locations processing the angles it sees cars at. Unfortunately, this requires the cameras to actively track and communicate with each and every robot, which does not scale well to many independent robots.

Multilateration can be done by measuring the varying time of flight of signals between receivers on the robots to be located and transmitter stations at known locations. The benefit of this approach is that each receiver does not have to communicate any information to any transmitter or any other receiver. If the transmitters are transmitting a known signal, then the receiver can simply time the signals it receives to get information about the time of flight of the signals.

2.1 Time difference of arrival multilateration

It might seem easy to measure distances simply by measuring the time from when a transmitter's microcontroller starts trying to transmit and the time the receiver circuitry sends a detectable signal to a microcontroller on the car. Unfortunately, there are sources of delay that would contribute to this time measurement other than the time it takes for the ultrasound to propagate through the air. The transmitting and receiving ultrasonic transducers act as mechanical band-pass filters, as they are built to physically resonate at 25 kHz. They, together with the more significant band-pass filter in the receiver circuitry, will add some delay to the signal. We could carry out experiments to measure and correct for this delay, but with the receive circuitry in particular, this would require either being careful with tolerances to ensure each receive amplification and filter circuit is similar enough to the next, or individually characterizing and calibrating the correction term for different receivers. It would also mean every receiver would need to be synchronized with every transmitter to measure the time of transmission and time of receiving each signal on a shared clock, and such synchronization generally requires two-way communication between the transmitters and receivers, or one-way communication with known (or very small) latencies, which would impact the ability of the system to scale as more receivers are added.

In the approach used in our system, as well as in satellite navigation systems, there is a small fixed number of transmitter stations that are synchronized with each other and send out signals at fixed known times relative to each other, but the receivers do not communicate with the transmitter stations to synchronize their clocks. Knowing that the transmitters are broadcasting certain signals at certain times, the receiver can simply observe the time difference at which it sees signals from different transmitters to get a difference in the lengths of the signals flights. Since all the transmitters have identical hardware and are sending out the same sort of signal, and these signals are received through the same receiver hardware for each receiver, any difference in the signals' times of flight must be due to differences in times of propagation of the ultrasound through the air. This has the advantage that the receiver does not need to communicate with the transmitter stations in any form to synchronize its clock with their clocks. In general, calculating position using a time difference of arrival measurements requires one more measurement (i.e., one more transmitter) than using absolute time of flight measurements. However, adding one more transmitter station that is identical to the rest requires no additional hardware design, so it is a small fixed cost to pay for making the positioning system more scalable, as, without the need to synchronize receivers, now there can be any number of receivers which do not need to interact with each other or the transmitters to calculate their individual positions.

2.2 Position calculations

The problem of multilateration based on time differences of arrival was explored and solved in the context of "sound ranging", a practice used beginning in World War I to locate enemy artillery by comparing the time at which the sound of the enemy guns firing reached sound recording devices placed at known locations along the front. In sound ranging, the enemy guns at an unknown position were the transmitters of the sound signal. For our case, the transmitter stations are at known locations and the receiver's position is what we wish to know, but the ideas in the math are the same.

We place the transmitters in a rectangle of known width X and length Y. The transmitters are arbitrarily numbered zero through three going counterclockwise, starting in the third quadrant. The origin of our coordinate system is arbitrarily placed below the center of this rectangle along the z-axis, at the height of the car receiver. The receiver is at a point $(x, y, 0)$. Let Z be the height of the transmitter plane above the height of the receiver; this is known because the car travels only on the flat floor. x and y are the unknowns we wish to calculate.

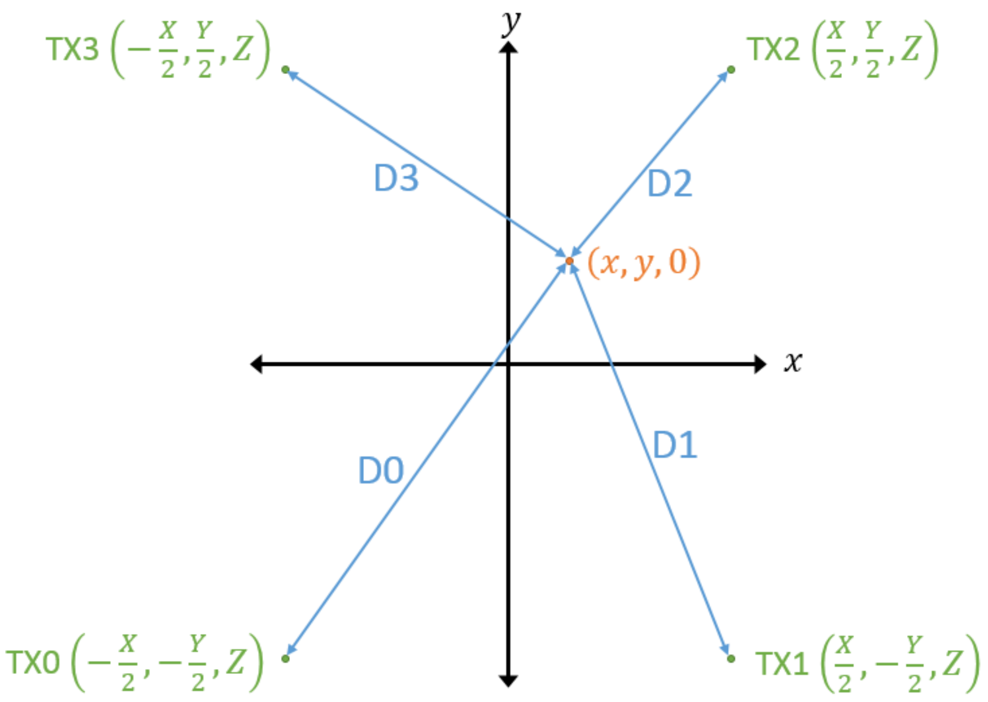
TX3 $\left(-\frac{X}{2}, \frac{Y}{2}, Z\right)$
y
TX2 $\left(\frac{X}{2}, \frac{Y}{2}, Z\right)$
D3
D2
$(x, y, 0)$
x
D0
D1
TX0 $\left(-\frac{X}{2}, -\frac{Y}{2}, Z\right)$
TX1 $\left(\frac{X}{2}, -\frac{Y}{2}, Z\right)$

Figure 1: Our coordinate system.

We can write the measurements the receiver takes as being the difference between each distance to transmitters one through three and the distance to transmitter zero:

$$\Delta_1 = D_1 - D_0 = \sqrt{\left(x - \frac{X}{2}\right)^2 + \left(y + \frac{Y}{2}\right)^2 + z^2} - \sqrt{\left(x + \frac{X}{2}\right)^2 + \left(y + \frac{Y}{2}\right)^2 + z^2}$$

$$\Delta_2 = D_2 - D_0 = \sqrt{\left(x - \frac{X}{2}\right)^2 + \left(y - \frac{Y}{2}\right)^2 + z^2} - \sqrt{\left(x + \frac{X}{2}\right)^2 + \left(y + \frac{Y}{2}\right)^2 + z^2}$$

$$\Delta_3 = D_3 - D_0 = \sqrt{\left(x + \frac{X}{2}\right)^2 + \left(y - \frac{Y}{2}\right)^2 + z^2} - \sqrt{\left(x + \frac{X}{2}\right)^2 + \left(y + \frac{Y}{2}\right)^2 + z^2}$$

Given that we can measure Δ_1, Δ_2, and Δ_3, we wish to solve these three equations for the two unknowns x and y. In the next two subsections, we will discuss two approaches to this calculation that we tried.

A Mathematica notebook file showing the derivations of all the formulas discussed below can be found at `https://github.com/YingVictor/ultrasonic-positioning/raw/master/MultilaterationDerivationMathematica.nb`. The exact implementation of these calculations in software can be found in the file position.c located in appendix section A.2.

2.2.1 A closed form solution?

The problem is overdetermined, but if we treat Z as an additional unknown, it turns out this problem has a closed-form solution that can easily be found using computer algebra software such as Mathematica:

$$\begin{aligned}
x &= \frac{\Delta_1(\Delta_2-\Delta_3)(\Delta_1-\Delta_2-\Delta_3)}{2X(\Delta_1-\Delta_2+\Delta_3)} \\
y &= \frac{\Delta_3(\Delta_2-\Delta_1)(\Delta_3-\Delta_2-\Delta_1)}{2Y(\Delta_1-\Delta_2+\Delta_3)} \\
Z &= \pm\frac{1}{2XY(\Delta_1-\Delta_2+\Delta_3)}\Big[X^2Y^2\left(\Delta_1^2+(\Delta_2-\Delta_3)^2\right)\left(\Delta_3^2+(\Delta_2-\Delta_1)^2\right) \\
&\qquad - X^2\left(\Delta_3^2(\Delta_1-\Delta_2)^2(\Delta_1+\Delta_2-\Delta_3)^2+X^2Y^2(\Delta_1-\Delta_2+\Delta_3)^2\right) \\
&\qquad - Y^2\left(\Delta_1^2(\Delta_3-\Delta_2)^2(\Delta_3+\Delta_2-\Delta_1)^2+X^2Y^2(\Delta_1-\Delta_2+\Delta_3)^2\right)\Big]^{\frac{1}{2}}
\end{aligned}$$

This geometry gives fairly simple closed form solutions for x and y, and we can throw away the calculation for Z. The solution can be calculated directly except for the locations where the denominator tends to zero. This only occurs when $\Delta_1-\Delta_2+\Delta_3 \to 0$. Substituting in the definitions of Δ_1, Δ_2, and Δ_3 and simplifying, it can be demonstrated that this only occurs when $x \to 0$ or $y \to 0$, which, intuitively, is when the receiver is equidistant from some of the transmitters. When calculating our position we can first check for these two cases and change the calculation appropriately. Specifically, if $\Delta_1 \approx 0$, then we can estimate our position as

$$\begin{aligned}
x &= 0 \\
y &= \frac{\Delta_3\Delta_2}{2Y} \\
Z &= \pm\frac{1}{2Y}\sqrt{\left(\Delta_3^2-Y^2\right)\left(Y^2-\Delta_2^2\right)-X^2Y^2}
\end{aligned}$$

And if $\Delta_3 \approx 0$, then we can estimate our position as

$$\begin{aligned}
x &= \frac{\Delta_1\Delta_2}{2X} \\
y &= 0 \\
Z &= \pm\frac{1}{2X}\sqrt{\left(\Delta_1^2-X^2\right)\left(X^2-\Delta_2^2\right)-X^2Y^2}
\end{aligned}$$

This calculation turns out to work decently. Unfortunately, by treating Z as an unknown, we are throwing away information rather than making use of all the information we have to get a better approximation. We could actually calculate the value of Z and compare it against the known value as a sort of sanity check to help throw out bad measurements, but the calculation for Z is sufficiently complex that it seemed not worth the trouble. This particular solution tends to magnify errors in measurement, particularly near the x- and y-axes where certain factors in both the numerators and denominators become very small and measurement errors can change these factors substantially. Using the approach just described where near the x- and y-axes we switch to a different approximation, the positioning calculation becomes discontinuous, so the errors one sees are not always locally consistent as one moves around in a small area. The entire problem of position calculation could be approached in other ways that do not give closed-form solutions but require iterative numerical approximation methods.

2.2.2 A least squares regression

In an overdetermined problem, a common approach to finding an approximate solution is to minimize some metric that is the sum of some squares of errors. Actual GPS receivers do this, and, with some encouragement from Prof. Houck, we ended up doing this as well. In particular, the metric we minimized was

$$f(x,y) = \Big(\Delta_1 - \hat{\Delta}_1(x,y)\Big)^2 + \Big(\Delta_2 - \hat{\Delta}_2(x,y)\Big)^2 + \Big(\Delta_3 - \hat{\Delta}_3(x,y)\Big)^2$$

and so the problem we were solving was

$$\operatorname*{arg\,min}_{x,y} \; f(x,y)$$

where Δ_n indicates the actual measured differences described earlier, and $\hat{\Delta}_n(x,y)$ indicates the expected value for Δ_n given particular values of x and y. If you review how we defined the Δ_n, you may notice that all of these deltas are with respect to D_0, the distance to transmitter zero located at $(-\frac{X}{2}, -\frac{Y}{2}, Z)$. This may seem like we have arbitrarily chosen to give more weight in the errors to this first distance. In fact this choice seems sensible in the context of how the transmitters synchronize, which is potentially a significant source of errors. As discussed in Section 4.1, TX0 is the master that determines when ultrasonic signals should begin being sent out, and the other transmitters follow its lead.

The method of minimization that we use is essentially Newton's method. Given a guess $\mathbf{r} = (x,y)$, we calculate the next guess as

$$\mathbf{r}_{\text{next}} = \mathbf{r} - \alpha \frac{f(\mathbf{r})\nabla f(\mathbf{r})}{\left\|\nabla f(\mathbf{r})\right\|^2}$$

where the factor $\alpha = 0.1$ is used to keep the guess from moving too far too quickly, which can lead to overshoot and failure to converge. The values of f and ∇f can be calculated exactly at each point (x, y), as f and its partial derivatives have perfectly cromulent closed-form expressions in terms of x and y. The expressions are quite large to write out, and so have been left out here, but many terms and factors, such as the expected distances between the four transmitter stations and the point (x, y), appear repeatedly, so they can be calculated once and reused in the calculations in software. The exact calculation can be seen in appendix section A.2.

If at any point $\nabla f(\mathbf{r}) = \mathbf{0}$, that is, if both partial derivatives of f are zero and we have reached a stationary point, we stop the iterative minimization method. Otherwise, we continue iterating to get better approximations for x and y until either 100 iterations have been performed, which is generally more than enough, or the value of the metric f drops below 0.01 ft^2, at which point it is sufficiently small that we don't care to keep performing the iterative minimization.

Minimizing this error metric has the useful benefit that, if we find the value of the metric is very close to zero, we can be quite confident that we are basing our position calculation off of good measurements. Conversely, if the metric is large after the minimization, then we may wish to throw this set of measurements and calculation away, as the measurements themselves may have been suspect. Sources of bad measurements from ultrasonic signals are discussed in Section 6. In our experience, when the receiver is stationary, our position calculations always give a consistent position to within a third of a foot, and the value of the error metric at that minimum is generally less than 0.1 ft^2. (The value of the metric at the minimum is, in general, likely to be smaller than the square of the actual error because the metric at the minimum is in general appreciably smaller than the metric evaluated at the correct (x, y) values.) In our code, we check that the minimized value of the error metric is less than 0.5 before recording and reporting the new calculated position. If this check fails, we simply throw the current measurement away.

3 Hardware systems

3.1 Ultrasonics

We decided to use 25 kHz ultrasonic signals for this project. This frequency is far enough from the typical 40 kHz used by the ultrasonic rangefinders commonly used by hobbyists for us to be confident that we would avoid picking up the rangefinder pings from other robots in the room using ultrasonic rangefinding, which might otherwise interfere with our positioning. We decided to use a lower ultrasonic frequency rather than a higher one because lower frequencies experience less attenuation traveling through the air. A lower frequency and hence a longer wavelength might sacrifice spatial resolution, but it was felt early on in the design process that maximizing the chances that we would be able to receive ultrasonic signals from across the room was a much greater concern. We decided to buy simple ultrasonic transducers so we could design circuits ourselves to maximize the range of our system. To this end, we purchased Prowave 250ST180 and 250SR180 transducers, which are the versions of the 18 mm diameter ultrasonic transducers offered by Prowave, optimized for transmission and reception, respectively. These ultrasonic transducers cost less than $8 each, but were still relatively expensive compared to other common ultrasonic transducers that might cost only a couple dollars. They were chosen for their relatively high advertised sensitivity and sound pressure level.

3.1.1 Ultrasonic transmitter stations

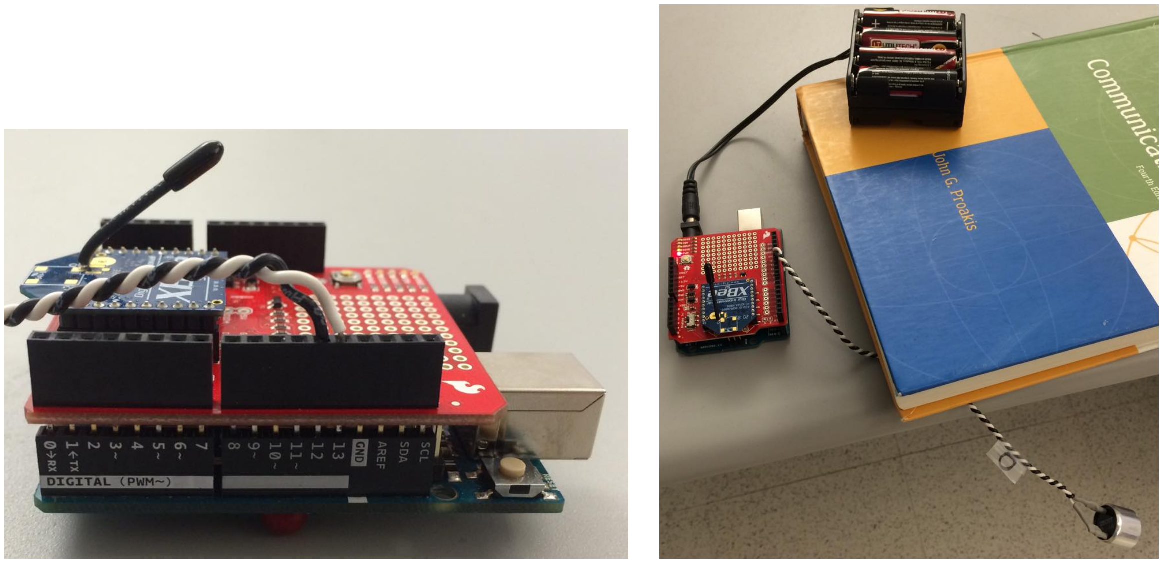

Figure 2: XBee radio module mounted to an Arduino, which connects to an ultrasonic transmitter. Note the use of the book-based precision mounting system.

We used an Arduino Uno as a low-cost microcontroller for our transmitter stations, which need to communicate with each other for synchronization purposes and control the Prowave 250ST180 ultrasonic transducers. The Prowave 250ST180 is simply connected directly to two digital output pins of the Arduino, which are capable of outputting 0 or 5 V. The reason we use two output pins rather than, for example, one output pin and ground, is that we can drive the two output pins as the inverse of each other, effectively doubling our amplitude. This is discussed further in section 4.1.2. We observed this improved our reception range a little bit in testing.

3.1.2 Ultrasonic receivers for cars

When receiving ultrasonic signals from across the room, we typically see the Prowave 250SR180 transducer produce signals on the order of a millivolt, and sometimes even lower. To be able to read this signal using a PSoC and to get rid of unwanted noise requires significant amplification and filtering. The circuit we ended up with for this purpose is shown in Figure 3.

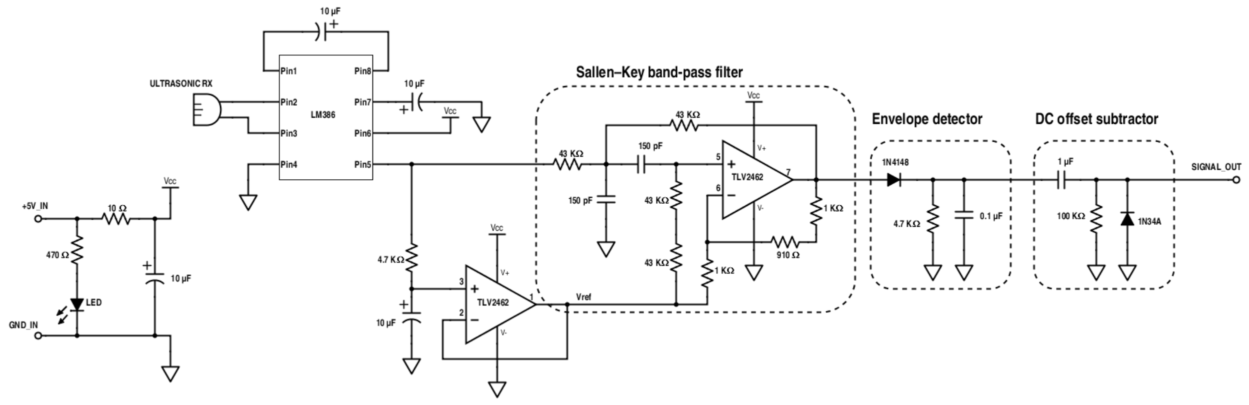

Figure 3: Schematic of ultrasonic receiver amplification and filtering circuitry.

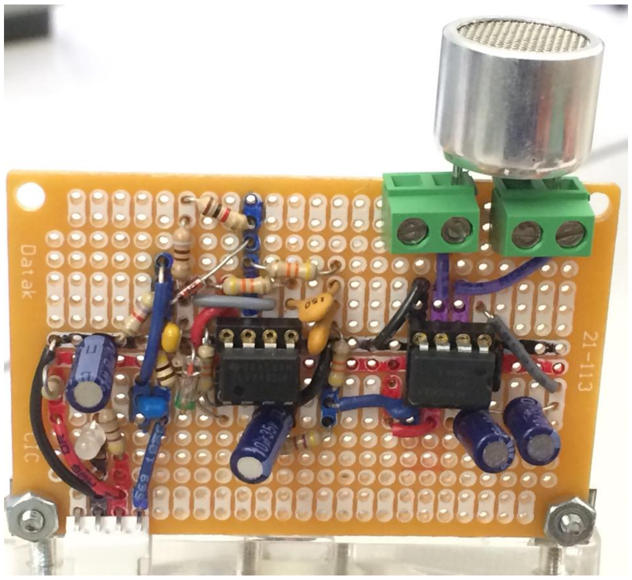

Figure 4: Photograph of ultrasonic receiver amplification and filtering circuitry.

We spent a while trying to build our own differential amplifiers with voltage gain on the order of hundreds, using general purpose op-amps. We repeatedly found that the noise was intolerably bad, it was easy to produce unstable circuits when trying to make the gain so large, and, ultimately, when we got past those problems, we rediscovered the limitations of gain-bandwidth products. After all, to give our 25 kHz signal a gain on the order of a thousand would require gain-bandwidth product on the order of 25 MHz! Finally understanding the amplifier chip specifications we needed to be looking for to provide large gain at ultrasonic frequencies, we went of in search of a solution, and found that the LM386 audio amplifier chip was readily available. This chip takes a differential input biased to ground, and produces and output referenced to half the power supply, with a gain that be set between 20 and 200. As it is designed for audio, it works just fine for our 25 kHz signal, which is just barely above the upper limit for human hearing. It works extremely well providing a large first stage of amplification with the gain set to 200.

After the initial amplification, we were seeing a decent amount of both 60 Hz noise and high frequency noise in the signal. we wished to put our signal through band-pass filtering to reduce those low and high frequency noises, as well as provide additional amplification to bring the signal up to the level of whole volts. To do this, we decided to build a Sallen–Key band-pass filter, which is a common topology for an active filter that in general provides non-unity gain, which is desirable in this case. This filter theoretically has a center frequency of 24.7 kHz, a Q factor of about 11, voltage gain of about 32 at the center frequency. We found some TLV2462 dual op-amp chips to use that have a gain-bandwidth of about 6 MHz, which is sufficient. In practice, due to poor tolerances in the components, the Q factor and the gain both tend to be lower than the theoretical value, but the gain in all the versions of this circuit that we built was above an order of magnitude at 25 kHz, and it showed a decent ability to reject of 40 kHz signals, which we might pick up from ultrasonic rangefinders used in other projects. Perhaps more importantly, it did quite a good job filtering out 60 Hz and high frequency noise, producing a fairly clean waveform. As the amplification is relative to a reference voltage of half the power supply, the maximum amplitude we can get in the output is half the power supply, or 2.5 V. The two amplification stages together have a voltage gain of several thousand which is usually enough to get clipping of the received signal. Clipping is not a problem in our context, so this is fine.

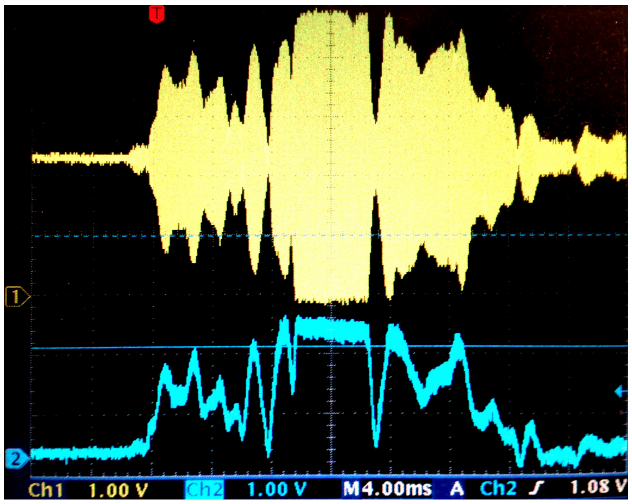

Figure 5: Ch1 (Yellow): Output of the Sallen–Key band-pass filter. Ch2 (Blue): Output of DC offset subtractor, after the envelope detector.

After the amplification and filtering, we have a fairly straightforward envelope detector to demodulate the signal that uses an ordinary silicon diode. After the envelop detector, we subtract off the DC offset in order to get a signal that is referenced to ground. This is essentially achieved using a AC coupling capacitor in the signal line and a pull-down resistor to ground on the output. However, if the DC offset subtraction consisted of only those two components, it would pull the average level of the output to ground, and the output would range from positive to negative voltages, and the voltage of the top and bottom of pulses in the signal would vary depending on the timing and durations of the pulses. To prevent this, a 1N34A germanium diode was added between the output and ground to limit the output so that it cannot drop far below ground. In particular, it cannot drop below ground by more than the turn-on voltage of the diode. A germanium diode was chosen because of its low turn-on voltage. This has the effect of keeping the parts of the signal that are low close to ground, and the pulses from when ultrasonic pings are received are all to positive voltages. A captured waveform demonstrating the function of the envelope detector and DC offset subtractor is shown in Figure 5.

3.2 Serial radio communication

To synchronize all of the transmitters, we needed a system of communication between the transmitters with either very low or very stable latency. A logical wireless communication choice, due to its relative simplicity was the XBee 802.15.4 (series 1) radio module from Digi. In contrast, we could communicate with a transport protocol such as TCP or UDP over an internet connection using WiFi, but with such a heavy protocol stack, it becomes difficult to reason about the sources of latency and to know whether latency is symmetric, and in all likelyhood the latency would be highly variable depending on contention for the WiFi bandwidth available in the room. XBee modules are essentially just modems that translate between RS-232 on a physical wire and the 802.15.4 wireless communication standard. They can be configured for point-to-point communication, but even easier is just to have all the transmitter stations XBee modules on the same channel, so that each character sent by one transmitter station is broadcast and received by all the others. While the configuration tool for XBee modules can be strangly finicky, once the XBees are configured, using them to communicate could not be simpler: they merely need to be provided with 5 V power and ground, and connected with two wires for sending and receiving data. Since we were able to obtain a exclusive channel for use in the lab, and since we are not sending much data between the transmitter stations, there is little contention in our radio communication network, and we do not expect latencies to vary due to data being held in buffers, waiting to be transmitted. As shown in Figure 2, each transmitter station has an Arduino with a shield connecting it to an XBee module. We were able to use the Arduino SoftwareSerial library to communicate between each Arduino via the XBees.

4 Signals

Measuring varying time of flights of a transmitted signal could be done by measuring the phase of a periodic signal, or calculating and finding the maximum of a cross correlation function between the received signal and the signal the transmitter is known to have transmitted, for some arbitrary transmitted signal. GPS works using the latter approach with pseudorandom signals, as this maximizes the time resolution of the time of flight measurements. In our case, we felt it would not be feasible to encode much information in the signals we send out, as there is a fair amount of distortion and echoing as the sound travels through the air, as well as ringing in the transmitter and receiver that would distort the signal. Therefore, we took a much simpler approach: we measure the time of the first rising edge for each ultrasonic ping signal received. This time should correspond to the time it took for the ultrasonic ping to travel the line-of-sight distance between the transmitter station and the receiver. The exact duration of echos and ringing after the first rising edge then does not matter.

4.1 Transmitted signals and synchronization

To allow the car to know where it was in the room, we developed a system of four transmitters stations, one in each corner of the room, that send ultrasonic pulses at well-defined intervals. Each of the transmitters is pointed towards the opposite side of the room. This is because the transmitter is directional, that is, the strength of the signal depends on the angle of transmission. In this configuration, the receiver will be able to receive the pings if they are directly underneath the transmitter simply because the signal is stronger close to the transmitter, and if they are further away, the angle will help improve reception. If the last three pulses are sent at known times with respect to the first pulse during each series of pulses, then the time differences of arrival of each pulse at the receiver can be used to determine the car's position. Thus, all of the transmitters need to be synchronized such that they always send an ultrasonic pulse at the same time with respect to the first pulse, sent by the master transmitter. All of the code for the transmitter stations is given in Appendix A.4.

4.1.1 Synchronization

While radio waves are fast, XBee communication has latency much higher than the time for the propogation of radio waves. To ensure synchronization, the first step was to determine the latency in serial communication between the master transmitter and each of the slave transmitters. To find the latencies, the master transmitter sends a character (either 'a', 'b', or 'c', depending on which slave transmitter it wishes to communicate with) to the radio channel, then waits to receive the same character in return from the correct slave transmitter. Each slave transmitter is programmed to immediately respond to its own lowercase character (again 'a' for the first slave transmitter, 'b' for the second, and 'c' for the third) by transmitting the same character back to the channel. The master transmitter is then able to calculate the round-trip latency by keeping track of the time to send and receive the character. We found typical round-trip latencies to vary from about 17 ms to 20 ms. We suspect much of this time is due not necessarily to the XBees, but to the software and hardware stack on the Arduino that sends and receives RS-232. The master transmitter tests the round-trip latency for a slave transmitter five times, and divides the total of the round-trip latencies by ten to get an estimate for the one-way latency to each slave transmitter station. It then transmits another character, this time an uppercase letter corresponding to the slave transmitter it wishes to communicate with ('A' for the first slave transmitter, 'B' for the second, and 'C' for the third). Upon sending the uppercase character, the master transmitter then sends a series of four bytes encoding what it measured the one-way latency for the slave transmitter to be. The slave transmitter then sends back an acknowledgment character (the same uppercase character that it received from the master to start the transaction) to tell the master that it has received all of the bytes. We make the assumption here that the round-trip latency is symmetric — that the time taken to transmit from master to slave is the same as to transmit from slave to master. Since all the transmitter stations have identical hardware, we find this assumption to be quite reasonable, and we did a few quick tests to confirm the assumption. When repeated for each of the three transmitters (five round-trip latency tests and transmission of the average one-way latency time), this process ensures near-perfect synchronization. Although latencies can occasionally change, we found that they usually stayed relatively constant when the transmitter stations are not being disturbed, meaning the average round-trip latency is not skewed and the one-way latencies are fairly accurate.

4.1.2 Sending Pings

With the latency measurement and communication process completed, the master transmitter sends out a lowercase 'p' over its XBee to the other transmitters to tell them that it is about to send a pulse. It then sends out its ultrasonic pulse, using two pins connected to the actual ultrasonic transmitter via a twisted pair. To gain more control over the ultrasonic pulse output, we used low-level pin manipulation to turn the pins of the Arduino connected to the ultrasonic transmitter on and off. In this way, we were able to simultaneously turn on one pin while turning off another, which cannot be done using the standard digitalWrite() command. The two pins are turned on and off opposite to each other, allowing for twice the voltage range. One pin would be switched on to +5V while the other was switched to ground, and then vice versa. With this manipulation, the differential signal sent to the ultrasonic transducer is a square wave with peak to peak voltage of 10V, double what we would achieve if we tied one pin of the transmitter to ground and only switched the other one. We also have good control over timing using this method, as one very fast register write switches both pins, and can use this control to send the pings at exactly 25 kHz, the optimal rated frequency for the transmitters and receivers that we purchased.

When the master sends out the 'p' character followed by its ping, each of the slave transmitters receive the 'p' character a few milliseconds later according to their respective latencies. They can then start timing how long it has been since the master transmitter first began transmitting its ping. Once the required time has elapsed (100 ms intervals between each ping), the slave transmitter begins transmitting its ping through the same low-level pin manipulation discussed above. Thus, the first slave transmitter waits 100 ms after receiving the send pulse character from the master minus the known latency time that it has received from the master plus a small amount of processing time due to the fact that the master must first write the 'p' character and then send the ping (the actions are not simultaneous on the master end). Each consecutive slave transmitter waits an extra 100 ms more before sending its pulse, again subtracting the known latency time and adding the processing delay. 100 ms was chosen as the delay time between each ping because while the transmission of the ping itself lasts only 5 ms, the echoes created from the signal bouncing around the room cause the received signal to stretch out much more, sometimes up to about 25 ms, and if the receiver is accross the room from one transmitter while very close to another, the signal from the transmitter far away could take as much as 30 ms longer to reach the receiver. We want enough time between pings received such that there is no doubt about the reception of each ping, so 100 ms is a safe period of time between transmissions. The overall result is four 25 kHz ultrasonic pings sent out 100 ms apart.

After one cycle of four pings sent out, the master then waits 300 ms before beginning the next cycle of latency tests and pings. This delay is necessary to allow the receiver to distinguish which ping is the ping sent from the master transmitter and which are the pings sent from each of the slave transmitters: TX0, the master transmitter, always transmits first after a quiet period of at least a few hundred milliseconds. Thus, the total cycle time between when the master sends one of its pings and the next must be greater than around 500 ms assuming the 100 ms between pings so that the receiver can distinguish the master ping based on the longer delay time between the last ping and the master ping as compared to the normal 100 ms time between pings. Since the latency tests also take time, in the final version of our system, the transmitters send out a round of pings every two thirds of a second or so. However, if improvements were to be made to speed up the transmission such as to get more position calculations per second on the receiving end, both the delay between pings and the delay between sets of pings could be reduced. Also important is that the slave transmitters sit idle after transmitting their ping. Thus, the master can begin doing latency testing on the first slave transmitter immediately after it sends its ping but before the second slave transmitter sends its ping, effectively interweaving pinging and latency testing so that latency tests do not impinge on the total timing of the system. In effect, then, the overall frequency of transmission cycles is only limited by the pulse-stretching experienced by the receiver and the time needed to distinguish the master ping, not by latency test times.

4.1.3 Timeouts and Error Checking

As a note, we found that bytes are sometimes lost over the course of communicating between devices on the radio channel. This loss can be an issue if it causes the system to hang while, for example, the master looks for an acknowledgment from one of the slaves or one of the bytes containing the latency time is never received by the slave. To combat this issue, we created a timeout structure; if the master is ever waiting for a response from one of the slaves (either an acknowledgment or as part of a latency test), it will automatically timeout after 50 ms. If the timeout occurs while testing for latency, then that test is thrown out and not counted as part of the averaging. If an acknowledgment is not received, the master will simply move on without it. On the slave side, a timeout can also occur when receiving the bytes corresponding the latency from the master. If one of these bytes is lost, the slave simply uses the last known latency time and ignores the corrupted new data. This error checking and timeout structure helps to make the system more robust so that if a transmitter turns off or bytes are lost, the entire system can recover and continue operating without manual resetting.

4.2 Digital processing of received signals

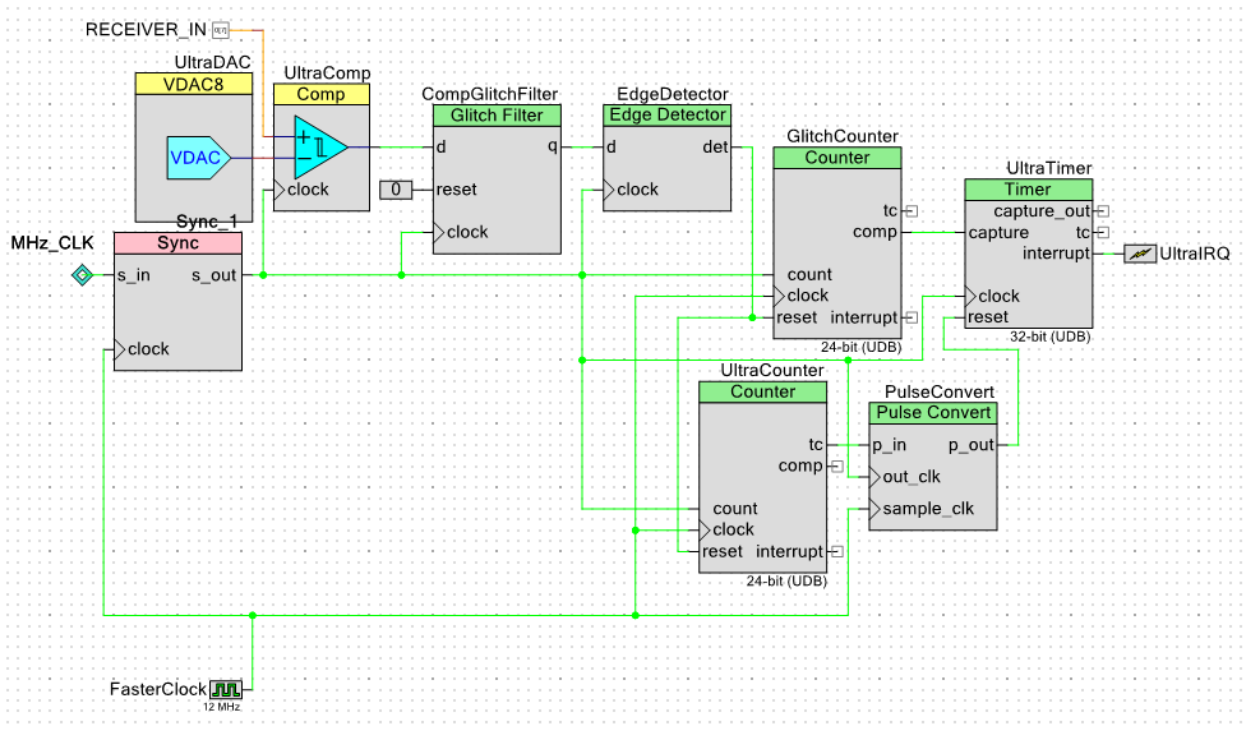

Figure 6: Programmable hardware used to capture timing information for positioning.

4.2.1 Digital filtering

Once the signal has been received, amplified, and filtered using the circuit described in section 3.1.2, we are left with the signal shown as the yellow waveform in Figure 7. To turn this waveform into a series of four clean square pulses so that timing can easily be done requires several pieces of programmable hardware. The schematic detailing the programmable hardware and connections referenced in this section can be found in Figure 6.

The first step was to use a comparator to turn the analog input into a series of digital pulses so that the edge time was rigidly defined and voltage levels could be raised even further from about the peak 2V received to 5V. We used a comparator level of 0.7V, high enough to avoid the comparator staying high for various echoes of the pings that had been sent out, but low enough to give accurate timing, since we want to record when the ping first reaches the receiver. This level of 0.7V was generated with an 8-bit voltage digital-to-analog converter. The output after the comparator step is shown as the blue waveform in Figure 7.

As is evident from the waveform representing the output of the comparator, due to the inconsistent nature of the analog waveform being received, further processing is necessary to generate a single pulse corresponding to the reception of each ping. Thus, we have a

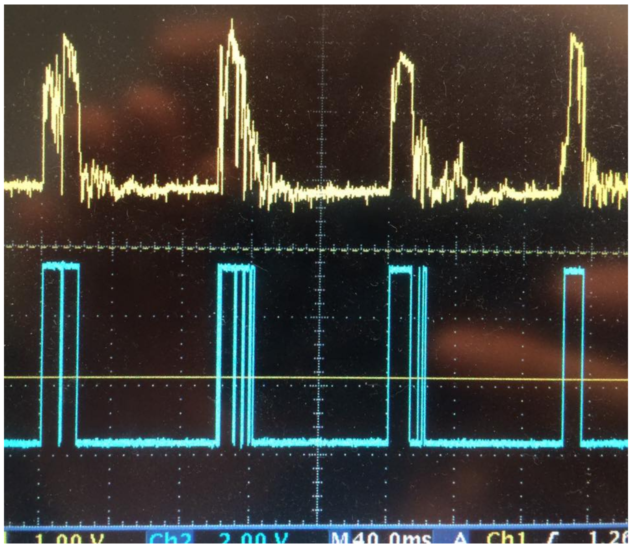

Figure 7: Ch1 (Yellow): Output from the receiver circuitry that is fed into the PSoC. Ch2 (Blue): Output of the comparator.

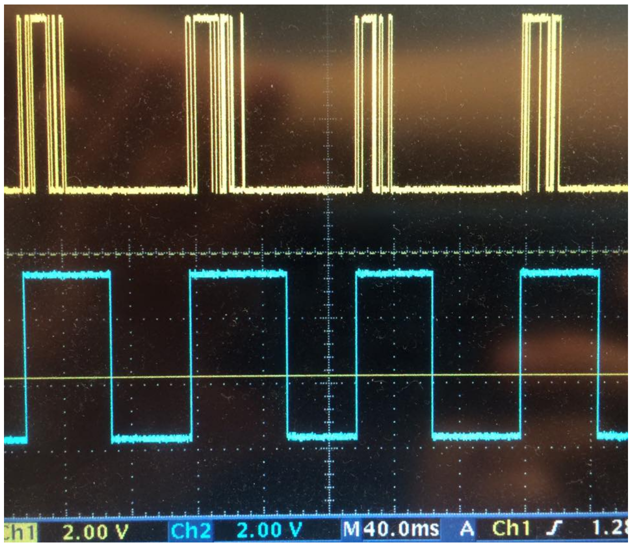

Figure 8: Ch1 (Yellow): Output of the comparator. Ch2 (Blue): Output of the filtering system that is fed into the timer.

system of two glitch filters to further refine the signal so that it is suitable for timing. A very useful, but somewhat unknown component, the glitch filter can be used either to eliminate small dips in a high signal or short spikes in the midst of a low signal. The former can be done by setting the glitch filter to bypass high mode such that its output remains high for a set period after seeing a high value while the latter can be achieved in the normal mode, equivalent to bypass low.

The first glitch filter (CompGlitchFilter in Figure 6), set to normal mode, directly takes the output of the comparator as input and has a very short period of only 40 μs, which is the period of a 25 kHz signal. This eliminates possible spikes that do not correspond to pings, possibly generated from small amounts of noise. However, it will not eliminate any significant part of the pings, since they always last longer than 40 μs for the initial spike when they are received.

The second glitch filter is made up of two components, an edge detector and a counter (labeled GlitchCounter in Figure 6). This glitch filter removes the occasional times when the signal goes low in the middle of the ping signal, as seen in the comparator waveform (yellow) in Figure 8. We made our own glitch filter out of two components here due to a limitation of the Glitch Filter component available in PSoC Creator: it can only maintain a moving window of length at most 256. Thus, with a clock at 1 MHz, its maximum filtering time would be only 0.256 ms, much shorter than a possible drop in the signal, which we have measured at up to 10 ms in length. To allow for a longer filtering time, the clock can be slowed, but this could cause a loss of precision in our timing measurements, since we ideally want a filtering time of about 25 ms (or about twice as long as possible drops in the signal), which would require a clock speed of about 10 kHz. With a 10 kHz clock speed, timing is only accurate to 0.1 ms, corresponding to about one tenth of a foot. That's not so bad at all, but it seems there's no reason that timing should be even close to a limiting factor in measurement precision. To get around this filtering length/measurement precision tradeoff, we created our own glitch filter using an edge detector and a counter. The edge detector outputs pulses on both edges of its input (essentially the output of the comparator). These are used to reset the counter, so the counter is always counting the number of cycles of the 1 MHz clock that have passed since the last edge. The counter outputs a high pulse if its count value is less than 25000 (corresponding to 25 ms with a 1 MHz clock). Thus, the output of the counter goes high as soon as it gets the initial edge and stays high even as the signal drops low for small periods of time. Since the high parts of the signal never last for more than 25 ms, this combination of components effectively imitates the function of a glitch filter. The output of this two-part filter can be seen in Figure 8, where the yellow waveform represents the input to the filtering (the comparator output) and the blue waveform represents the output of the filtering system: four clean pulses.

4.2.2 Timing

Now all that is left is timing the positive edges of the filtered received signal. To do this, we use a Timer component, named UltraTimer in Figure 6. Very conveniently, it can hold a queue of four values before triggering an interrupt.

In case some bad input gets past the filter, to make sure the queue is empty when the next round of pings starts, we reset the timer during the intervening quiet period between each set of four pings. To identify that quiet period, we have a counter component named UltraCounter that counts the number of cycles of a 1 MHz clock since the last time an edge was seen in the received signal, much like GlitchCounter does. Unlike GlitchCounter, UltraCounter counts up to a maximum value of 200000, or 200 ms. We use the terminal count output of UltraCounter as a reset signal for UltraTimer, to ensure UltraTimer's queue is emptied at some point during the quiet period, which is more than 200 ms long. 200 ms is longer than the delay between two received pings within a set of four pings should be, so this reset only happens during the reset period before the master transmitter sends out its next ping. Since the UltraCounter is a UDB Counter component, it counts edges of its "count" input, and must be operating at a higher speed clock. To convert the terminal count signal pulse from the faster clock domain to the 1 MHz clock domain used by the timer, a pulse converter component is used.

When UltraTimer sees four positive edges on the filtered received signal and captures those four times, it triggers an interupt handler that reads those captured values and converts the time differences into differences in distances in feet. Before doing so, some sanity checks on the captured values are performed. First, since a timer reset should occur every cycle of four pings, we know something is wrong if the captured timer value indicates it has been a long time since the last reset of the timer. If any captured timer value indicates it was captured more than a second since the previous measurement, something has gone wrong and we throw out the entire measurement. Furthermore, we throw away the entire measurement if any of the differences in distances are much larger than the dimensions of the rectangle of transmitters. Only if the measurements make it through these sanity checks do we carry out the positioning math described in section 2.2.2.

5 Testing and debugging scaffolds

Before we were doing integration testing of our positioning system, we developed various testing tools to test one part of the system by simulating inputs as if from the rest of the system. Here we will briefly discuss a few such test setups.

First, to test ultrasonic reception and the basic ability to time the difference between two pings, we had tx˙test˙2. In this test, one Arduino Uno controls two transmitters, and alternately sends out pings from them. We put the transmitters on the ends of long twisted wire pairs, so that we could move them close and further away, and see whether we could measure the time difference between when the pings are received changed. Different receiver circuits could then be tested. The code for this test is included in appendix section C.1.

To test the positioning math given relatively clean reception, we had rx˙test, whose code is included in appendix section C.3. Here, the Arduino sends out a series of four square pulses spaced unevenly, looking a lot like the blue waveform in Figure 8. A wire would be connected from the Arduino output pin directly to the PSoC, and the Arduino is essentially pretending to be the receiver circuitry, so that there were no actual ultrasonic components involved in this test setup. We could calculate what the timings of the received pulses should be at arbitrarily chosen locations, program the Arduino to send pulses with those timings, and then check whether the PSoC was correctly calculating the position we chose.

Finally, we had a version of rx˙test that actually uses ultrasonics, named tx˙test˙4. The code for tx˙test˙4, given in appendix section C.2, would run on an Arduino connected to a single transmitter. The Arduino would send out a series of four pings with certain timings. The actual ultrasonic receiver circuit would be plugged in to the PSoC, and we would see whether the PSoC was correctly calculating the position those timings should indicate. Since all the pings are being sent out a single transmitter, they all experience the same time of flight to the receiver, so the time of flight does not introduce any differences to the relative timings of the received pings. This could be tested with the transmitter on the other side of the room, and essentially the one transmitter is pretending to be four synchronized transmitters at different locations.

We did do some initial testing of radio communication by hooking Arduinos with XBees to separate laptops and communicating accross the room. The XBees always just seemed to work, and are rated to have a range of at least a few hundred feet, which is more than the length of the room. Having seen this, testing of the synchronization of transmitter stations was done simply by bringing all the transmitter stations to a single lab bench and using an oscilloscope to look at the relative timing of the pings they emitted. This did not require any special code. As practically none of the latency we are concerned with arises from the time for propagation of radio signals, we assumed that if the transmitters can correctly synchronize when they are within a few feet of each other, as assume they can still more or less correctly synchronize when they are spread around the room. The fact that positioning worked fairly well when we had the whole system running indicates this assumption was good enough.

6 Performance and sources of bad data

When the car is sitting stationary and has a clear line of sight to all four transmitter stations, the positions it will calculate from repeated measurements will generally vary by a few inches, and a new measurement occurs after each round of pings, which happens once every two thirds of a second or so. However, there are several ways the positioning system can be foiled and produce poor results.

If something is blocking the direct line of sight between the car and a transmitter station, generally the ultrasonic ping from the transmitter station will still reach the car, because ultrasound has no trouble traveling around obstacles. However, this will delay the first rising edge at the receiver for this ping, leading the car to believe it is further from that transmitter station and ruining the positioning calculation.

Sharp noises with appreciable high-frequency content, especially loud clapping, are readily observed to cause the ultrasonic transducers at the receivers to ring briefly as if receiving an ultrasonic ping. This does not appear to be a problem in our receiver circuitry, but rather has to do with the resonance of the physical transducer itself. Usually, such extra pings will fail the sanity checks or the final error metric check, and the whole measurement will be thrown out, so the problem is not that this introduces error into positioning calculations, but that it delays the next new position information. In our testing, one person clapping continuously a couple yards away from the receiver could indefinitely prevent the car from getting a position fix.

If the car is moving, because the signals from the four transmitter stations are received at different points in time, they will also be received when the car receiver is at different points in space. All of our positioning calculations described in Section 2 assume all measurements pertain to a single unknown location of the receiver, $(x, y, 0)$. When the car is moving, this assumption is false, and as shown in Figure 11, this can lead to errors that are much larger than when stationary. Intuitively, one might expect errors on the same order as the distance the car travels during a round of pings, which last a few hundred milliseconds. Thus, if the car is traveling a a couple feet per second, one can expect errors on the order of a foot or more.

7 Applications

Having built a indoor ultrasonic positioning system, we set out to demonstrate its capabilities.

7.1 Navigation

One application of our positioning system would be to create an autonomous navigation system, whereby a car could travel to defined waypoints, either making stops at desired locations or tracing out a desired path. With radio communication, cars could interact and travel routes around each other, avoiding collisions, or following a path previously traveled by another car.

7.1.1 H-Bridge

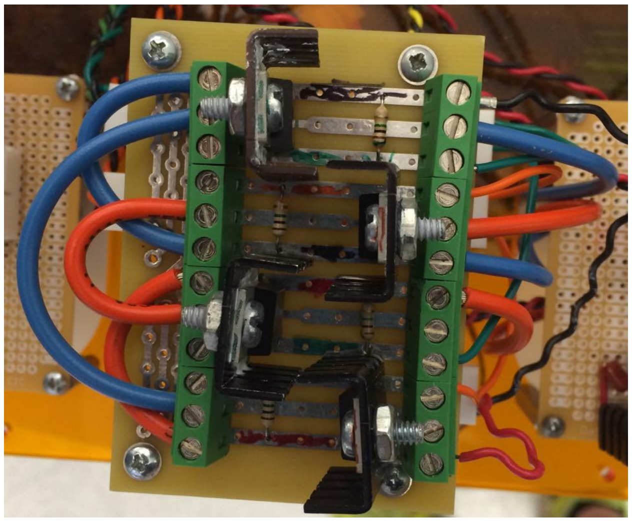

Figure 9: The board of the H-bridge circuit.

To enable the car to navigate robustly around the room, the ability to drive the car in reverse would be helpful, so we added an H-bridge circuit (as seen in Figure 9) to our motor control. The H-bridge makes it possible for the PSoC to run the motor in both directions.

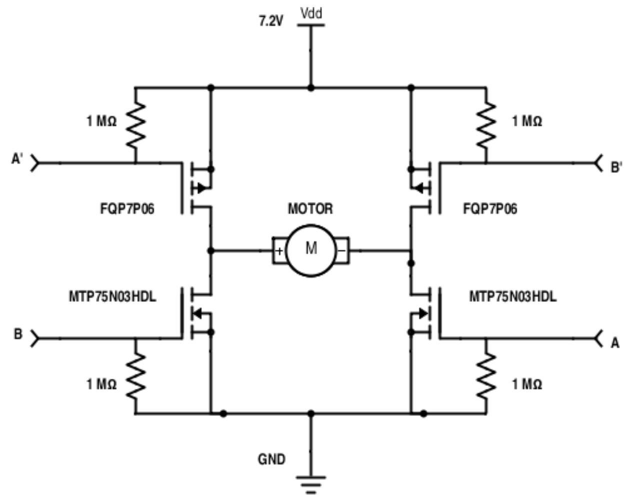

Figure 10: Schematic of the H-bridge circuit..

The gate of each transistor corresponds to a control signal from the PSoC. Inputs A and B controlling the NMOS's can come directly from the PSoC. Because the PMOS transistors had to deliver 7.2V to the motor and the PSoC only outputs a maximum of 5V, we need to put some hardware between the PSoC output and the gate of the PMOS's to raise the signal voltage to 7.2V. For one car, we chose the LM311P comparator IC to output 7.2V when the PSoC signal is above a reasonable threshold of 3.6V and output 0V when the PSoC signal is under 3.6V. For the other car, we simply ran the PSoC output to simple single-transistor inverters, constructed out of a general-purpose 2N7000 n-type pull down transistor and a 1 kΩ pull-up resistor.

With such an H-bridge circuit, driving forward can be done by, for example, turning on the PMOS controlled by A' and sending a PWM signal to A. Backwards driving can be done by turning the PMOS controlled by B' and sending a PWM signal to B. Rheostatic braking can also be acheived, for example by simultaneously turning on both NMOS's while leaving the PMOS's off. In the software that controls all this, we ensure any time we're switching between any of these three modes, (forwards, backwards, and braking), all the transistors are first turned off for a millisecond, to guarantee there is no time at which a short circuit path exists. This can be seen in appendix section A.4.

7.1.2 General autonomous navigation

Unfortunately, figuring out how to get the car to navigate to an arbitrary (x, y) position using only the positioning system proved more difficult than initially expected. If we knew the orientation of the car, getting to an arbitrary position seems fairly straightforward. At any given time, take the desired orientation to be the direction between the position of the car and the desired position. If the difference between the current orientation and desired orientation is small, we could simply perform PID or any other form of feedback control to point the car towards the desired destination. If the difference is large, we could code special turning maneuvers for the car to rotate itself around.

Unfortunately, the only way we had to figure out the orientation of the car in general is to make some movement larger than the effective spatial resolution of our positioning system, and look at the change in position. After identifying an initial orientation, in principle one could keep updating an estimate of orientation by a combination of odometry and continuing to look at the direction in which the most recent positions have moved. Unfortunately, we started investigating this relatively late, and ran out of time before coming up with a working solution. The noise in our positioning is such that merely taking our orientation to be the direction of the change between the two most recent positions is far too noisy. In general, it seems estimating orientation based on a fixed-length history of recent positions might work well for some speeds, but would do poorly if the speed of the car varied, and a special approach would have to be taken when the car comes to a stop. It seems possible in principle to come up with a good algorithm to look at a dynamically adjusting number of previous points and determining the recent trend in change direction, but we have not found this algorithm. As for odometry, we have the Hall effect sensor to tell us about our speed, and we always know something about the current steering angle as we have control over that. Unfortunately, it seems that steering is sufficiently nonlinear and/or the Hall effect sensor doesn't provide enough resolution that orientation could be tracked accurately through turns. Using inertial sensing (i.e., a gyroscope) would probably help a lot.

7.1.3 Simpler navigation demonstration

Of course, to demonstrate the ability to use the positioning system to navigate, it is not truly necessary to have the car autonomously navigate itself to an arbitrary (x, y) position from any arbitrary position and orientation. It is quite easy to start the car in a known orientation in a known area, and have it move forward or backward until it sees its x- or y-coordinate reach a certain threshold, before it takes some other action, such as changing direction. Having the car oscillate between certain y-coordinates, or stop and go at certain locations, turns out to be quite simple.

7.2 Mapping

For the mapping application of our positioning system, we again decided to use the XBee radio modules to communicate between the PSoC board on the car and our computer. The goal was to have the positioning information collected by the car sent to the computer so that it could be plotted in real time. To accomplish this goal, we used an XBee module on the car with a shield similar to those used for the Arduino transmitters to connect to the PSoC and a XBIB-U-DEV board to connect the XBee module to the computer via USB. These XBees were configured such that they were using a different channel to communicate than the XBees used in the transmitter stations to avoid possible interference that could derail both systems of communication.

On the car side, all communication was coordinated by the PSoC through the provided UART module. Using this module, we can set TX and RX pins to send and receive information via the XBee connected to those designated pins. Whenever data is generated from the receiver, it is written to the UART module and sent over the XBee to the XBee attached to the computer. The position string sent to the computer is always the same: ‘X’ then current x-coordinate, then ‘Y’ then current y-coordinate. An example string, if the car were located at (2.41, -5.34), would be “X2.41Y-5.34”.

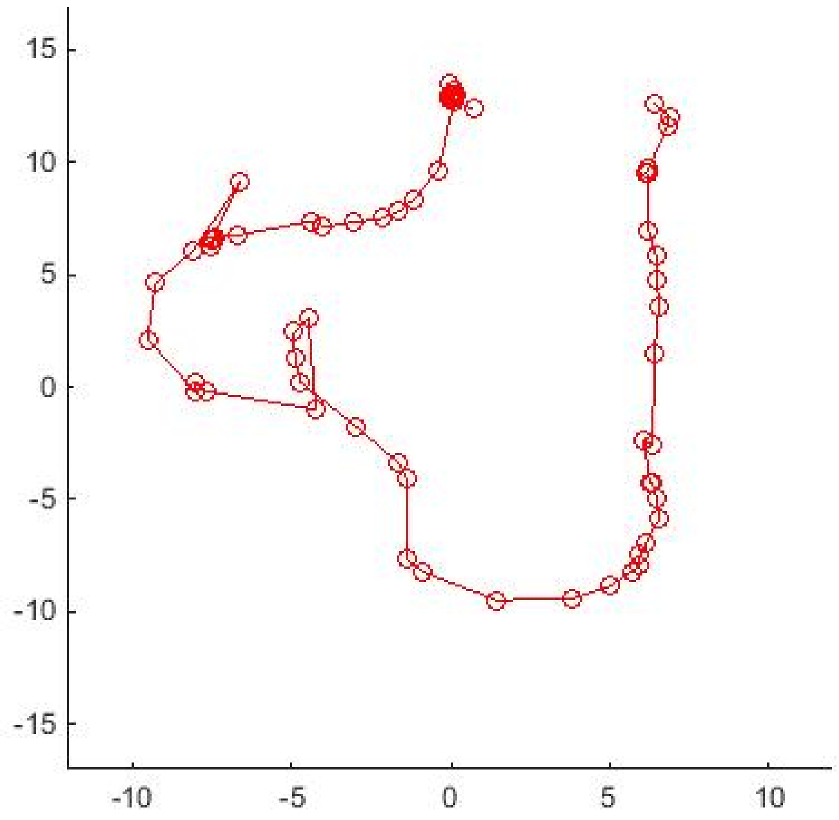

Figure 11: A partial map plotted in Matlab using coordinates received from the car as it drove around the racetrack. At about $(-5, 0)$ you can see where the car veered off the track, and was manually returned to the track, before it continued along the track. On the other side of the curve, at around $(-7, 7)$, there is one noticeable bad data point, which appeared to be caused by a human blocking line of sight between the car and one of the transmitter stations.

On the computer side, we used Matlab to communicate via serial with the XBee due to its relatively simple serial port I/O commands and ability to easily turn points received into a plot. To create a real-time connection between the car, we first connect to the correct COM port and open it using the correct Baud Rate. Then we set up the plot according to the parameters of our grid such that the plot will be to the same scale of the room. We used the animatedline object in Matlab to integrate new information into the plot as we received it from the car. The animatedline is very efficient for this application as it does not require the user to store each individual point in an array and update it each time a new point is received. Instead, the animatedline object has an addPoint() function that allows the user to add new points to the line that are then stored in the object dynamically, requiring only a refresh of the plot and no vector separately storing points. Then, as each string containing the data points is received via the attached XBee, we can parse the string according to the known formatting and extract the x and y coordinates and feed them into the animatedline for a dynamic plot of real-time position. The Matlab script used to generate these plots can be found in Appendix D.

Just as the Arduino code for the transmitters required timeouts and other checks to ensure a robust system, due to occasional data loss, this system also needs similar checks to be sure that it does not quit or produce undesirable results when it receives corrupted data. Thus, we have implemented various checks on the input, such as whether the string contains all four elements in the correct order (character, float, character, float), whether the string contains any information, and a timeout to determine if the connection to the car has been lost. Using the fgetl() function, we can take one line of input at a time to process, since we look for a newline character to terminate each string sent from the car. However, one problem that arose with this strategy was that sometime either in the transmission or receiving process, extra characters could be appended onto the front of the string that were either random or whitespace, throwing off our parsing. To avoid these characters being included in the parsing process, we do an initial paring of the string down to just the necessary information before the string format check; specifically, we iterate through the string until the 'X' character is found, signifying the beginning of the relevant coordinate string. With these error checks, the script runs smoothly, plotting the points received effectively and exiting gracefully by closing the serial port if an error occurs.

A C code compiled in PSoC Creator for car

Cypress's APIs use capitalized letters at the start of words. Therefore, whenever we define our own functions, we use only lowercase letters, which helps distinguish functions we write from the provided APIs, and reduces the chance of a name collision.

A.1 main.c

```c
/* ========================================
 * main.c
 * Victor A. Ying and Monica Lu
 *
 * This is the main program. It contains very little code.
 * ========================================
 */

#include <project.h>
#include <stdio.h>

#include "usb_uart.h"
#include "shell.h"
#include "drive.h"
#include "speed.h"
#include "steer.h"
#include "position.h"

/*
 * MAIN PROGRAM
 */
int main(void) {
    // Turn on interrupts
    CyGlobalIntEnable;

    // Initialize USB serial connection for I/O
//    usb_uart_init(USBUART_5V_OPERATION); //!!NOTE!! Make sure this matches your
        board voltage!

    // Initialize LCD display
    LCD_Start();

    // Initialize radio communincation
    UART_Start();

    // Begin positioning
    position_init();

    // Initialize navigation stuff
    drive_init();

    // Main loop performs user interface/communication actions.
    // Other actions are performed in interrupt handlers.
    for (;;) {
//        shell_handle_received_chars();
//        speed_display_info();

        // Display position to LCD
        if (position_data_available()) {
            char buf[32], radiobuf[32];
            float x, y;
            uint16 counter = 0u;
            uint8 status = CyEnterCriticalSection();
            x = position_x();
            y = position_y();
            sprintf(radiobuf, "X%.2fY%.2f\n", x, y);
            UART_PutString(radiobuf);
            /*
            LCD_Position(0,0);
            LCD_PrintNumber(counter++);
            sprintf(buf, "Error:%.2f ", error());
            LCD_Position(0,4);
            LCD_PrintString(buf);
            sprintf(buf, "X:%.2f Y:%.2f ", x, y);
            LCD_Position(1,0);
            LCD_PrintString(buf);
            */
            CyExitCriticalSection(status);
        }
    }
}

/* [] END OF FILE */
```

A.2 position.c

```
/* ============================================================
 *
 * position.c
 * Michael Danielczuk, Andrew Kim, Monica Lu, and Victor Ying
 *
 * Provides positioning using time difference of arrival
 * multilateration with four transmitters arranged in a
 * rectangle. Assumes the transmitters send out pings in turn
 * with a certain amount of spacing time between when pings are
 * sent, and that these pings are sent in a counterclockwise
 * order.
 *
 * ============================================================
 */

#include <project.h>
#include <math.h>
#include <limits.h>
#include <stdio.h>

#include "position.h"


/*
 * CONSTANTS
 */

#define X 23.5 // distance between first and second transmitters in feet
#define Y 33.75 // distance between second and third transmitters in feet
#define Z 7.583
#define CLOCK_FREQ 1000000 // Hz
#define WAVE_SPEED 1135.0 // ft/s
#define TX_SPACING 100 // ms
#define EPSILON 0.5 // ft
#define DEL_FACTOR 0.1 // ft
#define MAX_ERROR 0.5 // ft^2
#define ERROR_THRESHOLD 0.01 // ft^2
#define MAX_ITERATIONS 100

//#define SHOW_GARBAGE // Uncomment this to check if sanity checks are failing
#define PRINT_CONVERGENCE // Comment this out to make positioning silent

/*
```

```
 * STATIC FUNCTION PROTOTYPES
 */

static CY_ISR_PROTO(positioningHandler) ;


/*
 * GLOBAL VARIABLES
 */

static float x = 0.0, y = 0.0; // the current position
static float fxy = 0.0; // the current error
static uint8 new_data = 0u; // Boolean indicating whether new data available


/*
 * FUNCTIONS
 */

/*
 * position_init:
 * Start positioning.
 */
void position_init(void) {
    UltraCounter_Start();
    GlitchCounter_Start();
    UltraTimer_Start();
    UltraComp_Start();
    UltraDAC_Start();
    UltraIRQ_Start();
    UltraIRQ_SetVector(positioningHandler);
}

/*
 * position_data_available:
 * returns nonzero if new data since the last time this function was called.
 */
uint8 position_data_available(void) {
    uint8 status = CyEnterCriticalSection();
    uint8 ret = new_data;
    new_data = 0u;
    CyExitCriticalSection(status);
    return ret;
}

```

```
/*
 * position_?:
 * Getter functions for position in units of feet, with the origin at the
 * center of the rectangle formed by the transmitters.
 */
float position_x(void) {
    return x;
}
float position_y(void) {
    return y;
}

float error(void) {
    return fxy;
}

float fabsf(float num) {
    if (num >= 0.0)
        return num;
    else
        return -num;
}

/*
 * positioningHandler:
 * Interrupt handler run after sequence of four pings, calculating position.
 */
static CY_ISR(positioningHandler) {
    uint32 time[4];
    float diff[4];
    int i, iters;
    float new_x, new_y, new_fxy;

    // Get the times of arrival
    for (i = 0; i < 4; i++) {
        time[i] = UltraTimer_ReadCapture();

        // If more than a second since the last reset, then throw away this
        // set of measurements
        if (time[i] == 0u || time[i] < ULONG_MAX - CLOCK_FREQ) {
#ifdef SHOW_GARBAGE
            x = (float)i;
            y = (float)time[i];
            new_data = 1u;
#endif
```

```
            return;
        }
    }

    // Calculate differences in distances in feet
    for (i = 1; i < 4; i++) {
        diff[i] = (float)((int32)(time[0] - time[i])
                        - i*(CLOCK_FREQ/1000*TX_SPACING))
                  * (WAVE_SPEED/CLOCK_FREQ);

        // If difference is much larger than the size of the rectangle of
        // transmitter stations, the data is probably bad, so throw it away
        if (fabsf(diff[i]) > X + Y) {

#ifdef SHOW_GARBAGE
            x = (float)i;
            y = diff[i];
            new_data = 1u;
#endif
            return;
        }
    }

    // Positioning using Newton's method
    new_x = x;
    new_y = y;
    iters = 0;
    do {
        float dfx, dfy, gradient_magnitude_squared;
        float dist[4], error[4];
        char buf[32];
        uint8 status;

        // Calculate what the distances should be based on our most recent (x,y)
        dist[0] = sqrt((new_x+X/2)*(new_x+X/2) + (new_y+Y/2)*(new_y+Y/2) + Z*Z);
        dist[1] = sqrt((new_x-X/2)*(new_x-X/2) + (new_y+Y/2)*(new_y+Y/2) + Z*Z);
        dist[2] = sqrt((new_x-X/2)*(new_x-X/2) + (new_y-Y/2)*(new_y-Y/2) + Z*Z);
        dist[3] = sqrt((new_x+X/2)*(new_x+X/2) + (new_y-Y/2)*(new_y-Y/2) + Z*Z);

        // Calculate disagreement between hypothetical distances and measurements
        for (i = 1; i < 4; i++)
            error[i] = (dist[i]-dist[0]) - diff[i];

        // Calculate our metric as the sum of the squares of the errors
        new_fxy = 0.0;
        for (i = 1; i < 4; i++)
            new_fxy += error[i]*error[i];

        // Calculate the partial derivatives of the metric
        dfx = 2*error[2] * (((new_x - X/2)/dist[2]) - (new_x + X/2)/dist[0]);
        dfx += 2*error[3] * (((new_x + X/2)/dist[3]) - (new_x + X/2)/dist[0]);
        dfx += 2*error[1] * (((new_x - X/2)/dist[1]) - (new_x + X/2)/dist[0]);

        dfy = 2*error[2] * (((new_y - Y/2)/dist[2]) - (new_y + Y/2)/dist[0]);
        dfy += 2*error[3] * (((new_y - Y/2)/dist[3]) - (new_y + Y/2)/dist[0]);
        dfy += 2*error[1] * (((new_y + Y/2)/dist[1]) - (new_y + Y/2)/dist[0]);

        // Quit now if we're already at a stationary point
        gradient_magnitude_squared = dfx*dfx + dfy*dfy;
        if (gradient_magnitude_squared == 0.0)
            break;

        // Otherwise, update according to a version of Newton's method
        new_x -= DEL_FACTOR * new_fxy * dfx / gradient_magnitude_squared;
        new_y -= DEL_FACTOR * new_fxy * dfy / gradient_magnitude_squared;

#ifdef PRINT_CONVERGENCE
        // Show convergence happening on the LCD
        status = CyEnterCriticalSection();
        sprintf(buf, "dX:%.1f dY:%.1f %d ", dfx, dfy, iters);
        LCD_Position(1,0);
        LCD_PrintString(buf);
        sprintf(buf, "X:%.1f Y:%.1f ", new_x, new_y);
        LCD_Position(0,0);
        LCD_PrintString(buf);
        sprintf(buf, " %.1f   ", new_fxy);
        LCD_Position(0,13);
        LCD_PrintString(buf);
        CyExitCriticalSection(status);
#endif

        iters++;
    } while ((fabsf(new_fxy) > ERROR_THRESHOLD) && (iters < MAX_ITERATIONS));

    if (fabsf(new_fxy) < MAX_ERROR) {
        x = new_x;
        y = new_y;
        fxy = new_fxy;
        new_data = 1u;
    }

    // Clear interrupt
    UltraTimer_ReadStatusRegister();
}


/* [] END OF FILE */
```

position.h

```
/* ============================================================
 *
 * position.h
 * Michael Danielczuk, Andrew Kim, Monica Lu, and Victor Ying
 *
 * Provides positioning using time difference of arrival
 * multilateration with four transmitters arranged in a
 * rectangle.
 *
 * ============================================================
 */

#ifndef POSITION_H
#define POSITION_H

#include <project.h>

/*
 * position_init:
 * Start positioning.
 */
void position_init(void) ;

/*
 * position_data_available:
 * returns nonzero if new data since the last time this function was called.
 */
uint8 position_data_available(void) ;

/*
 * position_?:
 * Getter functions for position in units of feet, with the origin at the
 * center of the rectangle formed by the transmitters.
 */
```

```c
float position_x(void) ;
float position_y(void) ;

/*
 * error:
 * Getter function for error in units of feet squared.
 */
float error(void) ;

#endif

/* [] END OF FILE */
```

A.3 drive.c

```c
/* ========================================
 * drive.c
 * Monica Lu and Victor Ying
 *
 * High-level movement program.
 * ========================================
 */

#include <project.h>

#include "drive.h"
#include "speed.h"
#include "steer.h"


/*
 * drive_init:
 * Begins driving the car.
 */
void drive_init(void) {
    steer_init();
    speed_init();

    steer_pid_start();
    speed_forward();
    speed_pid_start("0.5");
}

/*
 * magnet_callback:
 * Called on every hall sensor tick with a running total distance traveled.
 */
void magnet_callback(void) {
    // Do about half a lap
    if (distance_traveled >= 50.0)
        speed_brake();

    /*
    // Do two laps and maximum speed
    if (distance_traveled < 1.0)
        ;
    else if ((distance_traveled > 3.0 && distance_traveled < 20.0) ||
             (distance_traveled > 100.0 && distance_traveled < 118.0))
        speed_set(9.0);
    else if (distance_traveled < 194.0)
        speed_set(5.7);
    else
        speed_brake(); */
}


//[] END OF FILE
```

drive.h

```
/* ========================================
 * drive.h
 * Monica Lu and Victor Ying
 *
 * High-level movement program.
 * ========================================
 */

#ifndef DRIVE_H
#define DRIVE_H


/*
 * drive_init:
 * Begins driving the car.
 */
void drive_init(void) ;

/*
 * magnet_callback:
 * Called on every hall sensor tick with a running total distance traveled.
 */
void magnet_callback(void) ;

#endif

//[] END OF FILE
```

A.4 speed.c

```c
/* ========================================
 * speed.c
 * Monica Lu and Victor Ying
 *
 * Speed measurement and control.
 * ========================================
 */

#include <project.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#include "speed.h"
#include "steer.h"
#include "drive.h"
#include "usb_uart.h"

#define DISTANCE_PER_TICK 0.1285 // in feet
#define DERIV_CONTROL_AVERAGING 3
#define TIMER_COUNTS_PER_SECOND 1000000.0 // timers are fed by 1 MHz clock
#define PID_INTERVALS_PER_SECOND 100.0 // PID control is reevaluated every 10 ms

enum state {
    COAST = 0u,
    FORWARD = 1u,
    BACKWARD = 2u,
    BRAKE = 3u,
};

// Because the speed sensor is on the right side of the car, it underestimates
// speed during right turns and overestimates during left turns. This is for a
// compensating correction term.
#define SPEED_ADJUSTMENT_COEFFICIENT (-0.12)


static CY_ISR_PROTO(hall_handler) ;
static CY_ISR_PROTO(speed_pid_handler) ;
static void speed_pid_control(void) ;

static enum state current_state;

float speed = 0.0;
float distance_traveled = 0.0;
float power_output = 0.0;

static uint16 count_since_magnet = 0u;
static uint8 speed_pid_enabled = 0u; // Boolean value indicating whether or not
        to do PID speed control.
static float speed_setpoint = 0.0; // in feet per second
static float kp = 0.1; // in normalized control output adjustment per (feet per
        second) error.
static float ki = 0.2; // in normalized control output adjustment per (feet per
        second) error per second.
static float kd = 0.0; // in normalized control output adjustment per (change in
        feet per second per second).


static void set_state(enum state desired) {
    current_state = desired;
    Drive_Control_Reg_Wakeup();
    Drive_Control_Reg_Write(desired);
}


/*
 * speed_init:
 * Initializes timers, pwm, etc. for controlling speed. Initializes
 * to coasting state.
 */
void speed_init(void) {
    uint8 status = CyEnterCriticalSection();

    speed_coast();

    Hall_Timer_Start();
    Hall_IRQ_Start();
    Hall_IRQ_SetVector(hall_handler);

    Speed_PID_Timer_Start();
    Speed_PID_IRQ_Start();
    Speed_PID_IRQ_SetVector(speed_pid_handler);

    Drive_PWM_Start();

    CyExitCriticalSection(status);
}

```

```c
/*
 * speed_display_info:
 * Print current speed and normalized controller output to the LCD.
 */
void speed_display_info(void) {
    char strbuf[17];

    uint8 status = CyEnterCriticalSection();

    // Print out info
    sprintf(strbuf, "Speed: %.6f", speed);
    LCD_Position(0,0);
    LCD_PrintString(strbuf);
    if (speed_pid_enabled) {
        sprintf(strbuf, "Control:%.6f", power_output);
        LCD_Position(1,0);
        LCD_PrintString(strbuf);
    }

    CyExitCriticalSection(status);

    CyDelay(50u);
}

/*
 * speed_pid_start:
 * Sets the speed to the floating point value represented by desired_speed,
 * and enables PID speed control
 */
void speed_pid_start(const char* desired_speed) {
    uint8 status = CyEnterCriticalSection();

    if (*desired_speed != '\0')
        speed_setpoint = atof(desired_speed);
    speed_pid_enabled = 1;

    CyExitCriticalSection(status);
}

/*
 * speed_set:
 * Sets the speed_setpoint.
 */
void speed_set(float desired) {
    uint8 status = CyEnterCriticalSection();

    speed_setpoint = desired;

    CyExitCriticalSection(status);
}

/*
 * speed_coast:
 * Disables PID speed control, and sets main drive motor PWM compare value
 * to 0, and sets state to coasting.
 */
void speed_coast(void) {
    uint8 status = CyEnterCriticalSection();

    speed_pid_enabled = 0;
    Drive_PWM_WriteCompare(0);
    set_state(COAST);

    CyExitCriticalSection(status);
}

/*
 * speed_brake:
 * Disables PID speed control, and sets main drive motor PWM compare value
 * to 0, and sets state to braking.
 */
void speed_brake(void) {
    uint8 status = CyEnterCriticalSection();

    speed_pid_enabled = 0;
    Drive_PWM_WriteCompare(0);
    if (current_state == FORWARD || current_state == BACKWARD) {
        set_state(COAST);
        CyDelay(1);
    }
    set_state(BRAKE);

    CyExitCriticalSection(status);
}

/*
 * speed_forward:
 * Changes the state to forward motion.
 */
void speed_forward(void) {
```

```c
    uint8 status = CyEnterCriticalSection();

    if (current_state == BRAKE || current_state == BACKWARD) {
        set_state(COAST);
        CyDelay(1);
    }
    set_state(FORWARD);

    CyExitCriticalSection(status);
}

/*
 * speed_backward:
 * Changes the state to forward motion.
 */
void speed_backward(void) {
    uint8 status = CyEnterCriticalSection();

    if (current_state == BRAKE || current_state == FORWARD) {
        set_state(COAST);
        CyDelay(1);
    }
    set_state(BACKWARD);

    CyExitCriticalSection(status);
}

/*
 * speed_set_power:
 * Sets the main drive motor PWM compare value.
 */
void speed_set_power(const char *valstr) {
    uint8 status = CyEnterCriticalSection();

    if (speed_pid_enabled) {
        usb_uart_putline(
            "Cannot manually set power while PID control is running!");
    }
    else {
        uint16 cmp;

        power_output = atof(valstr);
        if (power_output > 1.0)
            power_output = 1.0;
        else if (power_output < 0.0)
```

```
            power_output = 0.0;
        cmp = (uint16)(power_output * USHRT_MAX);
        Drive_PWM_WriteCompare(cmp);
    }

    CyExitCriticalSection(status);
}

/*
 * speed_set_k*:
 * Set PID control constants.
 */
void speed_set_kp(const char *val) {
    uint8 status = CyEnterCriticalSection();
    kp = atof(val);
    CyExitCriticalSection(status);
}
void speed_set_ki(const char *val) {
    uint8 status = CyEnterCriticalSection();
    ki = atof(val);
    CyExitCriticalSection(status);
}
void speed_set_kd(const char *val) {
    uint8 status = CyEnterCriticalSection();
    kd = atof(val);
    CyExitCriticalSection(status);
}


/*
 * hall_handler:
 * Should run every time a magnet is seen.
 * Recalculates the current speed estimate.
 */
static CY_ISR(hall_handler) {
    static uint32 prev;
    uint32 val = Hall_Timer_ReadCapture();
    uint32 diff = prev - val;
    float time;
    uint8 status;

    prev = val;

    time = (float)diff / TIMER_COUNTS_PER_SECOND;
    speed = DISTANCE_PER_TICK / time; // in units of feet/second
```

```c
    speed /= 1.0 + steer_output * SPEED_ADJUSTMENT_COEFFICIENT;

    count_since_magnet = 0;

    distance_traveled += DISTANCE_PER_TICK /
            (1.0 + steer_output * SPEED_ADJUSTMENT_COEFFICIENT);

    status = CyEnterCriticalSection();
    magnet_callback();
    CyExitCriticalSection(status);

    // Clear interrupt
    Hall_Timer_ReadStatusRegister();
}

/*
 * speed_pid_handler:
 * Should run once every 10 ms interval. Does speed control.
 */
static CY_ISR(speed_pid_handler) {
    uint8 saved_interrupt_status;

    // If a magnet is not seen for a long time, decrease the speed estimate
    if (count_since_magnet > 0) {
        float time_since_magnet = count_since_magnet / PID_INTERVALS_PER_SECOND;
        float speed_estimate = DISTANCE_PER_TICK / time_since_magnet;
        speed_estimate /= 1.0 + steer_output * SPEED_ADJUSTMENT_COEFFICIENT;
        if (speed_estimate < speed)
            speed = speed_estimate;
    }
    count_since_magnet++;

    saved_interrupt_status = CyEnterCriticalSection();
    if (speed_pid_enabled &&
            (current_state == FORWARD || current_state == BACKWARD))
        speed_pid_control();
    CyExitCriticalSection(saved_interrupt_status);
}

static void speed_pid_control(void) {
    float error, deriv, next_riemann_sum,
          prop_control, integ_control, deriv_control;
    static float riemann_sum = 0.0;
    static float prev_errors[DERIV_CONTROL_AVERAGING];
    static uint8 prev_errors_index = 0;
    float change;
    uint16 pwm_cmp;
    uint8 status;

    // Begin control calculations
    error = speed_setpoint - speed;

    // Proportional control
    prop_control = kp * error;

    // Integral control
    next_riemann_sum = riemann_sum + error / PID_INTERVALS_PER_SECOND;
    integ_control = ki * next_riemann_sum;

    // Derivative control
    change = error - prev_errors[prev_errors_index];
    deriv = change * PID_INTERVALS_PER_SECOND / (float)DERIV_CONTROL_AVERAGING;
    prev_errors[prev_errors_index] = error;
    prev_errors_index = (prev_errors_index + 1) % DERIV_CONTROL_AVERAGING;
    deriv_control = kd * deriv;

    // Add it all together, with limiting to valid duty cycle values
    power_output = 0.15 + prop_control + integ_control + deriv_control;
    if (power_output < 0.0) {
        power_output = 0.0;
    }
    else if (power_output > 1.0) {
        power_output = 1.0;
    }
    else {
        // Anti-windup: only allow integrator to build up if not saturating
        riemann_sum = next_riemann_sum;
    }

    // Scale control value to a PWM compare value
    pwm_cmp = (uint16)(USHRT_MAX * power_output);
    status = CyEnterCriticalSection();
    Drive_PWM_WriteCompare(pwm_cmp);
    CyExitCriticalSection(status);
}


//[] END OF FILE
```

speed.h

```c
/* ========================================
 * speed.h
 * Monica Lu and Victor Ying
 *
 * Speed measurement and control.
 * ========================================
 */

#ifndef SPEED_H
#define SPEED_H


extern float speed;
extern float distance_traveled;
extern float power_output;


/*
 * speed_init:
 * Initializes timers, interrupts, pwm, etc. for speed control. Initializes
 * to coasting state.
 */
void speed_init(void) ;

/*
 * speed_display_info:
 * Print current speed and normalized controller output to the LCD.
 */
void speed_display_info(void) ;

/*
 * speed_pid_start:
 * Sets the speed to the floating point value represented by desired_speed,
 * and enables PID speed control
 */
void speed_pid_start(const char* desired_speed) ;

/*
 * speed_set:
 * Sets the speed_setpoint.
 */
void speed_set(float speed_setpoint) ;

```

```
/*
 * speed_brake:
 * Disables PID speed control, and sets main drive motor PWM compare value
 * to 0, and sets state to braking.
 */
void speed_brake(void) ;

/*
 * speed_coast:
 * Disables PID speed control, and sets main drive motor PWM compare value
 * to 0, and sets state to coasting.
 */
void speed_coast(void) ;

/*
 * speed_forward:
 * Changes the state to forward motion.
 */
void speed_forward(void) ;

/*
 * speed_backward:
 * Changes the state to forward motion.
 */
void speed_backward(void) ;

/*
 * speed_set_power:
 * Sets the main drive motor PWM compare value.
 */
void speed_set_power(const char *cmpstr) ;

/*
 * speed_set_k*:
 * Set PID control constants.
 */
void speed_set_kp(const char *val) ;
void speed_set_ki(const char *val) ;
void speed_set_kd(const char *val) ;


#endif

//[] END OF FILE
```

A.5 steer.c

```c
/* ========================================
 * steer.c
 * Monica Lu and Victor Ying
 *
 * Line detection and steering control.
 * ========================================
 */

#include <project.h>
#include <stdio.h>
#include <stdlib.h>

#include "steer.h"
#include "usb_uart.h"


#define FASTEST_CLK_FREQ 48000000.0 // 48 MHz
#define EXPECTED_ROW_LEN (0.000045 * FASTEST_CLK_FREQ) // in clock cycles
#define ACCEPTABLE_ROW_MARGIN 0.2
#define MIN_ACCEPTABLE_ROW_LEN ((uint16)(EXPECTED_ROW_LEN * (1.0 -
        ACCEPTABLE_ROW_MARGIN)))
#define MAX_ACCEPTABLE_ROW_LEN ((uint16)(EXPECTED_ROW_LEN * (1.0 +
        ACCEPTABLE_ROW_MARGIN)))
#define PID_INTERVALS_PER_SECOND 60.0 // Camera is 30 fps interlaced
#define DERIV_CONTROL_AVERAGING 4
#define STEERING_CENTER 1500 // 1.5 ms pulse = steer straight ahead


static CY_ISR_PROTO(camera_handler) ;
static void steer_pid_control(void) ;


float steer_output = 0.0;

static float measurement = 0.0;
static uint8 steer_pid_enabled = 0; // Boolean value indicating whether or not
        to do PID steering control.
static float kp = 0.75;
static float ki = 0.0;
static float kd = 0.03;


/*
```

```
 * steer_init:
 * Initializes timers, interrupts, pwm, etc. for steering the car.
 */
void steer_init(void) {
    steer_stop();

    Camera_Comp_Start();
    Camera_Counter_Start();
    Camera_Timer_Start();
    Camera_IRQ_Start();
    Camera_IRQ_SetVector(camera_handler);

    Steering_PWM_Start();
}

/*
 * steer_display_info:
 * Print current measurements and normalized controller output to the LCD.
 */
void steer_display_info(void) {
    char strbuf[17];

    // Print out info
    sprintf(strbuf, "Meas.: %f ", measurement);
    LCD_Position(0,0);
    LCD_PrintString(strbuf);
    sprintf(strbuf, "Steer: %f ", steer_output);
    LCD_Position(1,0);
    LCD_PrintString(strbuf);
    CyDelay(50u);
}

/*
 * steer_pid_start:
 * Enables PID steering control
 */
void steer_pid_start(void) {
    steer_pid_enabled = 1u;
}

/*
 * steer_stop:
 * Disables PID steering control.
 */
void steer_stop(void) {
```

```c
    steer_pid_enabled = 0u;
}

/*
 * steer_set:
 * Sets the steering angle.
 */
void steer_set(const char *valstr) {
    uint8 status = CyEnterCriticalSection();

    if (steer_pid_enabled) {
        usb_uart_putline(
            "Cannot manually set steering while PID control is running!");
    }
    else {
        uint16 cmp;
        steer_output = atof(valstr);
        if (steer_output > 1.0)
            steer_output = 1.0;
        else if (steer_output < -1.0)
            steer_output = -1.0;
        cmp = (int16)(500.0 * steer_output) + STEERING_CENTER;
        Steering_PWM_WriteCompare(cmp);
    }

    CyExitCriticalSection(status);
}

/*
 * steer_set_k*:
 * Set PID control constants.
 */
void steer_set_kp(const char *val) {
    kp = atof(val);
}
void steer_set_ki(const char *val) {
    ki = atof(val);
}
void steer_set_kd(const char *val) {
    kd = atof(val);
}

static CY_ISR(camera_handler) {
    uint16 row_start_time = Camera_Timer_ReadCapture();
    uint16 black_start_time = Camera_Timer_ReadCapture();
    uint16 black_end_time = Camera_Timer_ReadCapture();
    uint16 row_end_time = Camera_Timer_ReadCapture();
    uint16 black_mid_time, row_mid_time;
    uint8 saved_interrupt_status;

    // Only update our measurement of where the black strip is if the row looks
    // like a full row containing a single black strip. (This effectively tosses
    // out data at intersections.)
    uint16 row_length = row_start_time - row_end_time;
    if (row_length >= MIN_ACCEPTABLE_ROW_LEN &&
            row_length <= MAX_ACCEPTABLE_ROW_LEN) {
        black_mid_time = black_end_time + (black_start_time - black_end_time)/2u;
        row_mid_time = row_end_time + row_length/2u;

        measurement = (float)(int16)(black_mid_time - row_mid_time) * 2 /
                  (float)row_length;
    }

    // Do PID steering control
    saved_interrupt_status = CyEnterCriticalSection();
    if (steer_pid_enabled)
        steer_pid_control();
    CyExitCriticalSection(saved_interrupt_status);

    // Clear interrupt
    Camera_Timer_ReadStatusRegister();
}

/*
 * steer_pid_control:
 * Should run once after every new video frame. Does steering control.
 */
static void steer_pid_control(void) {
    float error, deriv, next_riemann_sum,
          prop_control, integ_control, deriv_control;
    static float riemann_sum = 0.0;
    static float prev_errors[DERIV_CONTROL_AVERAGING] = {0};
    static uint8 prev_errors_index = 0;
    float change;
    uint16 pwm_cmp;
    uint8 status;

    // Begin control calculations
    error = - measurement;

    // Proportional control
    prop_control = kp * error;

    // Integral control
    next_riemann_sum = riemann_sum + error/PID_INTERVALS_PER_SECOND;
    integ_control = ki * next_riemann_sum;

    // Derivative control
    change = error - prev_errors[prev_errors_index];
    deriv = change * PID_INTERVALS_PER_SECOND / (float)DERIV_CONTROL_AVERAGING;
    prev_errors[prev_errors_index] = error;
    prev_errors_index = (prev_errors_index + 1) % DERIV_CONTROL_AVERAGING;
    deriv_control = kd * deriv;

    // Add it all together, with limiting to valid duty cycle values
    steer_output = prop_control + integ_control + deriv_control;
    if (steer_output < -1.0) {
        steer_output = -1.0;
    }
    else if (steer_output > 1.0) {
        steer_output = 1.0;
    }
    else {
        // Anti-windup: only allow integrator to build up if not saturating
        riemann_sum = next_riemann_sum;
    }

    // Scale control value to a PWM compare value
    pwm_cmp = (int16)(500.0 * steer_output) + STEERING_CENTER;
    status = CyEnterCriticalSection();
    Steering_PWM_WriteCompare(pwm_cmp);
    CyExitCriticalSection(status);
}

//[] END OF FILE
```

steer.h

```
/* ========================================
 * steer.h
 * Monica Lu and Victor Ying
 *
 * Line detection and steering control.
 * ========================================
```

```c
 */

#ifndef STEER_H
#define STEER_H


extern float steer_output; // Ranges from -1.0 to 1.0.


/*
 * steer_init:
 * Initializes timers, interrupts, pwm, etc. for steering the car.
 */
void steer_init(void) ;

/*
 * steer_display_info:
 * Print current steering info to the LCD.
 */
void steer_display_info(void) ;

/*
 * steer_pid_start:
 * Enables PID steering control
 */
void steer_pid_start(void) ;

/*
 * steer_stop:
 * Disables PID steering control.
 */
void steer_stop(void) ;

/*
 * steer_set:
 * Sets the steering angle.
 */
void steer_set(const char *valstr) ;

/*
 * steer_set_steer_k*:
 * Set PID control constants.
 */
void steer_set_kp(const char *val) ;
void steer_set_ki(const char *val) ;
void steer_set_kd(const char *val) ;


#endif

//[] END OF FILE
```

A.6 shell.c

```c
/* ========================================
 * shell.c
 * Victor A. Ying and Monica Lu
 *
 * a very minimal shell.
 * ========================================
 */

#include <project.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "speed.h"
#include "steer.h"
#include "usb_uart.h"
#include "drive.h"

/*
 * vshell_do_command()
 * A function for parsing and dispatching command lines.
 * Must be passed a null-terminated string!
 */
void shell_do_command(const char* line) CYREENTRANT {
    char cmd[SHELL_MAX_COMMAND_LENGTH + 1];
    uint8 i = 0u;

    // Ignore leading whitespace
    while (isspace(*line))
        line++;

    // Identify command as the longest prefix not containing a space.
    while (*line != '\0' && !isspace(*line)) {
        cmd[i] = *line;
        i++;
        line++;
    }
    cmd[i] = '\0';

    // Throw away any whitespace between the command and the arguments
    while (isspace(*line))
        line++;

    // Now we can execute the identified command

    // Do nothing in case of empty command
    if (strcmp(cmd, "") == 0)
        ;

    // Shell built-in commands
    else if (strcmp(cmd, "repeat") == 0) {
        char numloops[SHELL_MAX_COMMAND_LENGTH];
        i = 0u;

        while (*line != '\0' && !isspace(*line)) {
            numloops[i] = *line;
            i++;
            line++;
        }
        numloops[i] = '\0';
        while (isspace(*line))
            line++;

        for (i = 0; i < atoi(numloops); i++) {
            shell_do_command(line);
        }
    }

    // Other commands
    else if (strcmp(cmd, "serial") ==0) {
        usb_uart_print_info();
    }
    else if (strcmp(cmd, "lcd") == 0) {
        LCD_ClearDisplay();
        LCD_PrintString(line);
    }
    else if (strcmp(cmd, "pid") == 0) {
        speed_pid_start(line);
    }
    else if (strcmp(cmd, "brake") == 0) {
        speed_brake();
    }
    else if (strcmp(cmd, "coast") == 0) {
        speed_coast();
    }
    else if (strcmp(cmd, "fw") == 0) {
        speed_forward();
    }
    else if (strcmp(cmd, "bw") == 0) {
        speed_backward();
    }
    else if (strcmp(cmd, "power") == 0) {
        speed_set_power(line);
    }
    else if (strcmp(cmd, "speedkp") == 0) {
        speed_set_kp(line);
    }
    else if (strcmp(cmd, "speedki") == 0) {
        speed_set_ki(line);
    }
    else if (strcmp(cmd, "speedkd") == 0) {
        speed_set_kd(line);
    }
    else if (strcmp(cmd, "steerpid") == 0) {
        steer_pid_start();
    }
    else if (strcmp(cmd, "steerstop") == 0) {
        steer_stop();
    }
    else if (strcmp(cmd, "steerset") == 0) {
        steer_set(line);
    }
    else if (strcmp(cmd, "steerkp") == 0) {
        steer_set_kp(line);
    }
    else if (strcmp(cmd, "steerki") == 0) {
        steer_set_ki(line);
    }
    else if (strcmp(cmd, "steerkd") == 0) {
        steer_set_kd(line);
    }
    // If command was not any of the above...
    else {
        char8 strbuf[128];

        sprintf(strbuf, "Command \"%s\" not recognized", cmd);
        usb_uart_putline(strbuf);
    }
}

/*
 * shell_handle_char()
 * A function for handling keypresses from a
 * terminal, and building up command lines.
 */
void shell_handle_char(char c) {
   static char linebuf[SHELL_MAX_COMMAND_LENGTH + 1];
   static uint8 line_index = 0;

   if (c < 127 && c >= 32) { // If the user pressed a character
      #ifdef SHELL_LCD_DEBUG
      // For debugging purposes, display the character to the LCD
      LCD_ClearDisplay();
      LCD_PutChar(c);
      #endif

      if (line_index < SHELL_MAX_COMMAND_LENGTH) {
         // Add character to current line
         linebuf[line_index] = c;
         line_index++;

         // Echo character to terminal.
         usb_uart_putchar(c);
      }
   }
   else if (c == '\b' || c == '\177') { // If the user pressed backspace
      #ifdef SHELL_LCD_DEBUG
      // For debugging purposes, display to the LCD
      LCD_ClearDisplay();
      LCD_PrintString("Backspace");
      #endif

      if (line_index > 0) {
         // Clear one character on the terminal
         usb_uart_putstring("\b \b");

         // Discard one character from the command buffer
         line_index--;
      }
   }
   else if (c == '\r') { // If the user has hit the Return/Enter key
      #ifdef VSHELL_LCD_DEBUG
      // For debugging purposes, display to the LCD
      LCD_ClearDisplay();
      LCD_PrintString("Return");
      #endif

      // Echo newline to terminal
```

```
        usb_uart_putCRLF();

        // Do command
        linebuf[line_index] = '\0';
        shell_do_command(linebuf);

        // Print a prompt to indicate ready for next command
        usb_uart_putstring("$ ");
        line_index = 0;
    }
    else { // For any other characters/keypress
        #ifdef SHELL_LCD_DEBUG
        char8 strbuf[128];

        // For aid in debugging, print out ASCII codes in hex
        sprintf(strbuf, " %X", (int)c);
        LCD_PrintString(strbuf);
        #endif
    }
}

/*
 * shell_handle_recieved_chars()
 * For every character that has been received, calls handler with
 * that character. Does nothing if no data has been received since
 * the previous time this function was called.
 */
void shell_handle_received_chars(void) {
    while(USBUART_DataIsReady()) { // Check for input data from PC
        uint8 i, count, buffer[128];

        count = USBUART_GetData(buffer, sizeof(buffer)); // Get any data from PC
        for (i = 0; i < count; i++) { // For each character recieved
            shell_handle_char(buffer[i]); // Pass the character to the handler
        }
    }
}

/* [] END OF FILE */
```

shell.h

```
/* ========================================
 * shell.h
 * Victor A. Ying and Monica Lu
 *
 * a very minimal shell.
 * ========================================
 */

#ifndef SHELL_H
#define SHELL_H

#define SHELL_MAX_COMMAND_LENGTH 63

/*
#define SHELL_LCD_DEBUG // Uncomment this line to print keypresses to the LCD
*/

/*
 * shell_do_command()
 * A function for parsing and dispatching command lines.
 * Must be passed a null-terminated string!
 */
void shell_do_command(const char* line) CYREENTRANT ;

/*
 * shell_handle_char()
 * For every character that has been received, calls handler with
 * that character. Does nothing if no data has been received since
 * the previous time this function was called.
 */
void shell_handle_char(char c) ;

/*
 * shell_handle_recieved_chars()
 * For every character that has been received, calls handler with that character.
 * Does nothing if no data has been received since the previous time this
          function
 * was called.
 */
void shell_handle_received_chars(void) ;

#endif

//[] END OF FILE
```

A.7 usb_uart.c

```c
/* ========================================
 * usb_uart.c
 * Victor A. Ying and Monica Lu
 *
 * Manage the serial connection over USB
 * ========================================
 */

#include <project.h>
#include <stdio.h>

#include "usb_uart.h"


/*
 * usb_uart_init()
 * Initialize USBUART for I/O. Should be called after CYGlobalIntEnable;.
 * NOTE: Make sure mode matches your board voltage!
 */
void usb_uart_init(uint8 mode) {
    USBUART_Start(0, mode);
    while(!USBUART_GetConfiguration())
        ;
    USBUART_CDC_Init();
}

/*
 * usb_uart_print_info()
 * Sends information about the USBUART connection to the USBUART connection.
 */
void usb_uart_print_info(void) {
    char8 strbuf[128];

    // USBUART info
    uint32 rate = USBUART_GetDTERate();
    uint8 stop_bits = USBUART_GetCharFormat();
    uint8 parity = USBUART_GetParityType();
    uint8 data_bits = USBUART_GetDataBits();
    sprintf(strbuf,
            "UART over USB data rate is %lu bits per second", rate);
    usb_uart_putline(strbuf);
    sprintf(strbuf,
            "UART over USB characters consist of %i data bits",
            (int)data_bits);
    usb_uart_putline(strbuf);
    sprintf(strbuf, "UART over USB parity mode; %s",
            (parity == USBUART_PARITY_NONE) ? "none" :
            (parity == USBUART_PARITY_ODD) ? "odd" :
            (parity == USBUART_PARITY_EVEN) ? "even" :
            (parity == USBUART_PARITY_MARK) ? "mark" :
            (parity == USBUART_PARITY_SPACE) ? "space" :
            "?ERROR?");
    usb_uart_putline(strbuf);
    sprintf(strbuf, "UART over USB stop bits: %s",
            (stop_bits == USBUART_1_STOPBIT) ? "1" :
            (stop_bits == USBUART_1_5_STOPBITS) ? "1.5" :
            (stop_bits == USBUART_2_STOPBITS) ? "2" :
            "?ERROR?");
    usb_uart_putline(strbuf);
}

/*
 * usb_uart_put*()
 * Wrappers around Cypress's provided USBUART_Put*() functions
 * that wait for previous Tx to finish before continuing.
 */
void usb_uart_putdata(const uint8* pData, uint16 length) {
    while(!USBUART_CDCIsReady())
        ;
    USBUART_PutData((uint8 *)pData, length);
}
void usb_uart_putstring(const char8* str) {
    while(!USBUART_CDCIsReady())
        ;
    USBUART_PutString((char8 *)str);
}
void usb_uart_putchar(char8 txDataByte) {
    while(!USBUART_CDCIsReady())
        ;
    USBUART_PutChar(txDataByte);
}
void usb_uart_putCRLF(void) {
    while(!USBUART_CDCIsReady())
        ;
    USBUART_PutCRLF();
}

/*
```

```
 * usb_uart_putline()
 * An additional helpful function for printing a string followed by CRLF.
 */
void usb_uart_putline(const char8* str) {
    usb_uart_putstring(str);
    usb_uart_putCRLF();
}

/* [] END OF FILE */
```

usb_uart.h

```
/* ========================================
 * usbuart_helpers.h
 * Victor A. Ying and Monica Lu
 *
 * Functions to manage the serial
 * connection over USB.
 * ========================================
 */

#ifndef USB_UART_H
#define USB_UART_H

#include "USBUART.h"
#include "shell.h"

/*
 * usb_uart_init()
 * Initialize USBUART (and possibly LCD) for I/O
 * NOTE: Make sure mode matches your board voltage!
 */
void usb_uart_init(uint8 mode);

/*
 * usb_uart_print_info()
 * Sends information about the USBUART connection to the USBUART connection.
 */
void usb_uart_print_info(void);

/*
 * usb_uart_put*()
 * Wrappers around Cypress's provided USBUART_Put*() functions
 * that wait for previous Tx to finish before continuing.
```

```c
 */
void usb_uart_putdata(const uint8* pData, uint16 length);
void usb_uart_putstring(const char8* string);
void usb_uart_putchar(char8 txDataByte);
void usb_uart_putCRLF(void);

/*
 * usb_uart_putline()
 * An additional helpful function for printing a string followed by CRLF.
 */
void usb_uart_putline(const char8* string);


#endif

//[] END OF FILE
```

B Arduino code for transmitter stations

B.1 Master transmitter station

```
/*****************************************************************
master_transmitter.ino

Communicate with slaves to establish radio communication latency,
then start off the sequence of ultrasonic pings.

Hardware Hookup:
  The XBee Shield makes all of the connections you'll need
  between Arduino and XBee. If you have the shield make
  sure the SWITCH IS IN THE "DLINE" POSITION. That will connect
  the XBee's DOUT and DIN pins to Arduino pins 2 and 3.
*****************************************************************/
// We'll use SoftwareSerial to communicate with the XBee:
#include <SoftwareSerial.h>

#define NUM_TRANSMITTERS 3
#define NUM_TESTS 5
#define TX_PORT PORTB

enum {
  TX_PIN_1 = 12,
  TX_PIN_2 = 13,
  TX_PIN_1_MASK = 1u << (TX_PIN_1 % 8u),
  TX_PIN_2_MASK = 1u << (TX_PIN_2 % 8u),
  FREQ = 25000u, // Hz
  PERIOD = 1000000u / FREQ, // s
  HALF_PERIOD = PERIOD / 2u, // s
  TIMEOUT = 50000u, // s
  DURATION = 5000u, // s
  BAUD_RATE = 9600u, // bps
};


// XBee's DOUT (TX) is connected to pin 2 (Arduino's Software RX)
// XBee's DIN (RX) is connected to pin 3 (Arduino's Software TX)
SoftwareSerial XBee(2, 3); // RX, TX

void setup()
{
  pinMode(TX_PIN_1, OUTPUT);
  pinMode(TX_PIN_2, OUTPUT);

  // Set up both ports at 9600 baud. This value is most important
  // for the XBee. Make sure the baud rate matches the config
  // setting of your XBee.
  XBee.begin(BAUD_RATE);
  Serial.begin(BAUD_RATE);
}

void loop()
{
  int i, j;
  unsigned long startTime, totalLatTime, averageLatTime, beginning;
  byte b[4];
  unsigned long startTotTime = micros();

  for (i = 0; i < NUM_TRANSMITTERS; i++) {
    totalLatTime = 0u;
    int successCount = 0;
    for (j = 0; j < NUM_TESTS; j++) {
      startTime = micros();
      XBee.write((char)('a' + i));
      while (!XBee.available() && (micros() - startTime < TIMEOUT));
      if (XBee.available()) {
        XBee.read();
        totalLatTime += (micros()-startTime);
        successCount++;
      }
      //Serial.println(totalLatTime);
    }
    if (successCount > 0) {
      averageLatTime = totalLatTime/(2*successCount);
      Serial.print("Average Time: ");
      Serial.println(averageLatTime);
      XBee.write((char)('A' + i));
      LongToBytes(averageLatTime, b);
      for (j = 0; j < 4; j++) {
        XBee.write(b[j]);
      }
      beginning = micros();
      while (!XBee.available() && (micros()-beginning < TIMEOUT));
      Serial.print("Acknowledgement: ");
      Serial.write(XBee.read());
      Serial.println();
    }
  }
  XBee.write('p');
  sendPing();
  Serial.print("Total Time: ");
  Serial.println(micros()-startTotTime);
  delay(300);
}

void LongToBytes(unsigned long val, byte b[4]) {
  b[0] = (byte )((val >> 24) & 0xff);
  b[1] = (byte )((val >> 16) & 0xff);
  b[2] = (byte )((val >> 8) & 0xff);
  b[3] = (byte )(val & 0xff);
}

/*
unsigned long BytesToLong(byte b[4]) {
  return ((unsigned long)(b[0] & 0xFF) << 24) + ((unsigned long)(b[1] & 0xFF) <<
           16)
         + ((unsigned long)(b[2] & 0xFF) << 8) + (unsigned long)(b[3] & 0xFF);
}
*/

void sendPing() {

  //Serial.println("Sending ping");

  // Send out ping from transmitter
  unsigned long beginning = micros();
  unsigned long time = micros() - beginning;
  unsigned long next = time;
  while (time < DURATION) {
    // Turn off TX_PIN_2 and turn on TX_PIN_1
    TX_PORT = TX_PORT & (~TX_PIN_2_MASK) | TX_PIN_1_MASK;
    next += HALF_PERIOD;
    while (micros() - beginning < next)
      ;

    // Turn off TX_PIN_1 and turn on TX_PIN_2
    TX_PORT = TX_PORT & (~TX_PIN_1_MASK) | TX_PIN_2_MASK;
    next += HALF_PERIOD;
    while ((time = micros() - beginning) < next)
      ;
  }
  Serial.println("Ping sent");
}
```

```
/*
// ASCIItoInt
// Helper function to turn an ASCII hex value into a 0-15 byte val
int ASCIItoInt(char c)
{
  if ((c >= '0') && (c <= '9'))
    return c - 0x30; // Minus 0x30
  else if ((c >= 'A') && (c <= 'F'))
    return c - 0x37; // Minus 0x41 plus 0x0A
  else if ((c >= 'a') && (c <= 'f'))
    return c - 0x57; // Minus 0x61 plus 0x0A
  else
    return -1;
}
*/
```

B.2 Slave transmitter stations

```
/*****************************************************************
slave_transmitter.ino

Responds to messages from the master and sends out ultrasonic pings
at the appropriate time.

Hardware Hookup:
  The XBee Shield makes all of the connections you'll need
  between Arduino and XBee. If you have the shield make
  sure the SWITCH IS IN THE "DLINE" POSITION. That will connect
  the XBee's DOUT and DIN pins to Arduino pins 2 and 3.
*****************************************************************/
// We'll use SoftwareSerial to communicate with the XBee:
#include <SoftwareSerial.h>
#define TX_PORT PORTB
#define TRANSMITTER_NUMBER 1

enum {
  TX_PIN_1 = 12,
  TX_PIN_2 = 13,
  TX_PIN_1_MASK = 1u << (TX_PIN_1 % 8u),
  TX_PIN_2_MASK = 1u << (TX_PIN_2 % 8u),
  FREQ = 25000u, // Hz
  PERIOD = 1000000u / FREQ, // s
  HALF_PERIOD = PERIOD / 2u, // s
  DURATION = 5000u, // s
  TIMEOUT = 50000u, // s
  BAUD_RATE = 9600u, // bps
  MAX_LAT_TIME = 20000u, // s
  SOFTWARE_SERIAL_DELAY = 2000u, // s
};

// XBee's DOUT (TX) is connected to pin 2 (Arduino's Software RX)
// XBee's DIN (RX) is connected to pin 3 (Arduino's Software TX)
SoftwareSerial XBee(2, 3); // RX, TX
unsigned long latTime;

void setup()
{
  // Set up both ports at 9600 baud. This value is most important
  // for the XBee. Make sure the baud rate matches the config
  // setting of your XBee.
  XBee.begin(BAUD_RATE);
  //Serial.begin(BAUD_RATE); // Serial for debugging

  pinMode(TX_PIN_1, OUTPUT);
  pinMode(TX_PIN_2, OUTPUT);
}

void loop()
{
  int i;
  char c;
  byte b[4];

  if (XBee.available())
  { // If data comes in from XBee, send it out to serial monitor
    c = XBee.read();
    if (c == (char)('a' + TRANSMITTER_NUMBER - 1)) {
      XBee.write(c);
    }
    else if (c == (char)('A' + TRANSMITTER_NUMBER - 1)) {
      for (i = 0; i < 4; i++) {
        unsigned long beginning = micros();
        while (!XBee.available() && (micros()-beginning < TIMEOUT));
        if (XBee.available()) {
          b[i] = XBee.read();
        }
        else {
          break;
        }
      }
      if (i == 4) {
       unsigned long temp = BytesToLong(b);
       if (temp < MAX_LAT_TIME) {
         latTime = temp;
       }
      }
      XBee.write(c);
    }
    else if (c == 'p') {
      delayMicroseconds(SOFTWARE_SERIAL_DELAY);
      sendPing();
    }
  }
}

unsigned long BytesToLong(byte b[4]) {
  unsigned long ret = 0u;
  for (int i = 0; i < 4; i++) {
    ret += (unsigned long)(b[i] & 0xFF) << 8*(3-i);
  }
  return ret;
}

void sendPing() {
  unsigned long beginning = micros();
  //Serial.println("Sending ping");
  while(micros() - beginning < (100000u*TRANSMITTER_NUMBER)-latTime)
    ;

  // Send out ping from transmitter
  beginning = micros();
  unsigned long time = micros() - beginning;
  unsigned long next = time;
  while (time < DURATION) {
    // Turn off TX_PIN_2 and turn on TX_PIN_1
    TX_PORT = TX_PORT & (~TX_PIN_2_MASK) | TX_PIN_1_MASK;
    next += HALF_PERIOD;
    while (micros() - beginning < next)
      ;

    // Turn off TX_PIN_1 and turn on TX_PIN_2
    TX_PORT = TX_PORT & (~TX_PIN_1_MASK) | TX_PIN_2_MASK;
    next += HALF_PERIOD;
    while ((time = micros() - beginning) < next)
      ;
  }

  //Serial.println("Ping sent");
}
```

C Arduino code for testing tools

C.1 tx_test_2

```
// One Arduino controls two transmitters, alternately sending
// out pings from the two of them, simulating two synchronized
// transmitter stations sending out pings in turn.

#define TX_1_PORT PORTD
#define TX_2_PORT PORTB

enum {
  TX_1_PIN_1 = 2,
  TX_1_PIN_2 = 3,
  TX_2_PIN_1 = 10,
  TX_2_PIN_2 = 11,
  TX_1_PIN_1_MASK = 1u << (TX_1_PIN_1 % 8u),
  TX_1_PIN_2_MASK = 1u << (TX_1_PIN_2 % 8u),
  TX_2_PIN_1_MASK = 1u << (TX_2_PIN_1 % 8u),
  TX_2_PIN_2_MASK = 1u << (TX_2_PIN_2 % 8u),
  FREQ = 25000u, // Hz
  PERIOD = 1000000u / FREQ, // s
  HALF_PERIOD = PERIOD / 2u, // s
  DURATION = 5000u, // s
  SPACING = 100000u, // s
  CYCLE = 1000000u, // s
};

void setup() {
  pinMode(TX_1_PIN_1, OUTPUT);
  pinMode(TX_1_PIN_2, OUTPUT);
  pinMode(TX_2_PIN_1, OUTPUT);
  pinMode(TX_2_PIN_2, OUTPUT);
}

void loop() {
  unsigned long beginning = micros();

  // Send out ping from transmitter 1
  unsigned long time = micros() - beginning;
  unsigned long next = time;
  while (time < DURATION) {
    // Turn off TX_1_PIN_2 and turn on TX_1_PIN_1
    TX_1_PORT = TX_1_PORT & (~TX_1_PIN_2_MASK) | TX_1_PIN_1_MASK;
    next += HALF_PERIOD;
    while (micros() - beginning < next)
      ;

    // Turn off TX_1_PIN_1 and turn on TX_1_PIN_2
    TX_1_PORT = TX_1_PORT & (~TX_1_PIN_1_MASK) | TX_1_PIN_2_MASK;
    next += HALF_PERIOD;
    while ((time = micros() - beginning) < next)
      ;
  }

  // Send out a ping from transmitter 2
  while ((time = micros() - beginning) < SPACING)
    ;
  next = time;
  while (time < SPACING + DURATION) {
    // Turn off TX_2_PIN_2 and turn on TX_2_PIN_1
    TX_2_PORT = TX_2_PORT & (~TX_2_PIN_2_MASK) | TX_2_PIN_1_MASK;
    next += HALF_PERIOD;
    while (micros() - beginning < next)
      ;

    // Turn off TX_2_PIN_1 and turn on TX_2_PIN_2
    TX_2_PORT = TX_2_PORT & (~TX_2_PIN_1_MASK) | TX_2_PIN_2_MASK;
    next += HALF_PERIOD;
    while ((time = micros() - beginning) < next)
      ;
  }

  // Wait for end of cycle
  while (micros() - beginning < CYCLE)
    ;
}
```

C.2 tx_test_4

```
// One Arduino controling one transmitters, simulates four transmitters

#define TX_PORT PORTD

enum {
  TX_PIN_1 = 2,
  TX_PIN_2 = 3,
  TX_PIN_1_MASK = 1u << (TX_PIN_1 % 8u),
  TX_PIN_2_MASK = 1u << (TX_PIN_2 % 8u),
  FREQ = 25000u, // Hz
  PERIOD = 1000000u / FREQ, // s
  HALF_PERIOD = PERIOD / 2u, // s
  NUM_TRANSMITTERS = 4u,
  DURATION = 5000u, // s
};

unsigned const long SPACINGS[NUM_TRANSMITTERS] =
    {117766u, 95420u, 77814u, 709001u};

void setup() {
  pinMode(TX_PIN_1, OUTPUT);
  pinMode(TX_PIN_2, OUTPUT);
}

void loop() {
  static unsigned i = 0;
  unsigned long beginning = micros();

  // Send out ping from transmitter
  unsigned long time = micros() - beginning;
  unsigned long next = time;
  while (time < DURATION) {
    // Turn off TX_PIN_2 and turn on TX_PIN_1
    TX_PORT = TX_PORT & (~TX_PIN_2_MASK) | TX_PIN_1_MASK;
    next += HALF_PERIOD;
    while (micros() - beginning < next)
      ;

    // Turn off TX_PIN_1 and turn on TX_PIN_2
    TX_PORT = TX_PORT & (~TX_PIN_1_MASK) | TX_PIN_2_MASK;
    next += HALF_PERIOD;
    while ((time = micros() - beginning) < next)
      ;
  }

  // wait until time for next ping
  unsigned long spacing = SPACINGS[i];
  i = (i + 1) % NUM_TRANSMITTERS;
  while (micros() - beginning < spacing)
    ;
}
```

C.3 rx_test

```
// Mimics the receiver board, sending a repeating series of four pings
// as if from four transmitter stations to the PSoC, for the PSoC to
// attempt position calculations.

enum {
  RX_PIN = 13,
  NUM_TRANSMITTERS = 4u,
  DURATION = 5000u, // s
};

unsigned const long SPACINGS[NUM_TRANSMITTERS] =
    {117766u, 95420u, 77814u, 709001u};

void setup() {
  pinMode(RX_PIN, OUTPUT);
}

void loop() {
  static unsigned i = 0;
  digitalWrite(RX_PIN, HIGH);
  unsigned long beginning = micros();
  while (micros() - beginning < DURATION)
    ;
  digitalWrite(RX_PIN, LOW);
  unsigned long spacing = SPACINGS[i];
  i = (i + 1) % NUM_TRANSMITTERS;
  while (micros() - beginning < spacing)
    ;
}
```

D MATLAB code for mapping

```
if (~isempty(instrfind))
    fclose(instrfind);
    delete(instrfind);
    clear s;
end

s = serial('COM14');
s.BaudRate = 9600;
s.Timeout = 20;
s.Terminator = 'LF';
fopen(s);

figure(1);
xlim([-12 12]);
ylim([-17 17]);
pbaspect([1 1 1]);
clear h;
h = animatedline('Color','red','Marker','o');

while 1
    try
        [str, count, msg] = fgetl(s);
        if(~isempty(msg))
            error(msg);
        end
    catch err
        err
        disp('A timeout occurred or the user quit the program!')
        break;
    end

    if str < 0
        break;
    end

    for i=1:size(str)
        if str(i) ~= 'X'
        else
            str = str(i:end);
            break;
        end
    end
    if strcmp(str, '')
```

```
        continue;
    end
    [c, num] = sscanf(str, '%c%f%c%f', 4);
    if num == 4
        x = c(2);
        y = c(4);
        if (x ~= 0 || y ~= 0)
            addpoints(h, x, y);
            drawnow
        end
    end
end

fclose(s);
delete(s)
clear s
```