LED Pathfinder

Andre' Walker
Druck Green

TABLE OF CONTENTS

1. Project Overview

At this very moment, the groundwork is being laid for autonomous car travel. Pioneering companies such as Google and Tesla strive to revolutionize road transportation by handing the wheel over to our most sophisticated electromechanical and software systems. This advancement enables many more opportunities for exploration. In particular, the burgeoning self-driving car industry provides fertile ground to test minimalist human interfaces in trip-planning. Our approach to this dynamic is outlined in the LED pathfinder project. In our system, a user is given the ability to dictate a vehicle's path using a "wand" outfitted with an LED. The freedom of a 3D interface exploits the familiar gesturing we make with our limbs and will simplify control of our self-driving vehicles.

Our project allows the user to intuitively determine the motion of a toy remote control tank. As mentioned earlier, this is accomplished by outfitting the user with an LED "wand" which they can then use to draw a path in the air which is picked up by an overhead mounted camera. The wand is simply a resistor and infrared LED in series with a handheld battery pack. The camera has been equipped with a filter which blocks out all light except the infrared. The camera connects to an FPGA which outputs a live feed to a VGA monitor. When the user places an infrared LED in the camera's field of view, that LED shows up as a white square on the monitor. This way, the user can immediately see where they are holding the wand, relative to the camera's field of view.

The system we built has two main modes of operation, record and playback, which are specified by the user via a switch on the FPGA. The path drawing outlined earlier takes place in record mode. Also in record mode, the path the user draws is displayed on the monitor by a series of smaller purple squares. Each purple square represents a single point in the path that has been sampled at a rate of 30 Hz. The user is limited to drawing for 15 seconds. After this time, no additional path points will be stored. Once the user is satisfied with the path they have drawn, they can switch to playback mode, and place the tank underneath the camera. The tank is outfitted with three of its own infrared LEDs, arranged to form an isosceles triangle. Our system uses these LEDs to determine the tank's direction and location. The tank will immediately begin to follow the path drawn by the user. It does this by looking at each point, in the order they we captured, and driving towards that point until it falls within a certain distance threshold. The current point on the path that the tank is driving towards appears as a different color than the rest of the purple squares in the path. This lets the user can track the tank's progress in real-time. After completion of the path, the tank will stop and wait for a new path to be drawn in record mode.

2. High Level Design

2.1. High Level Block Diagram

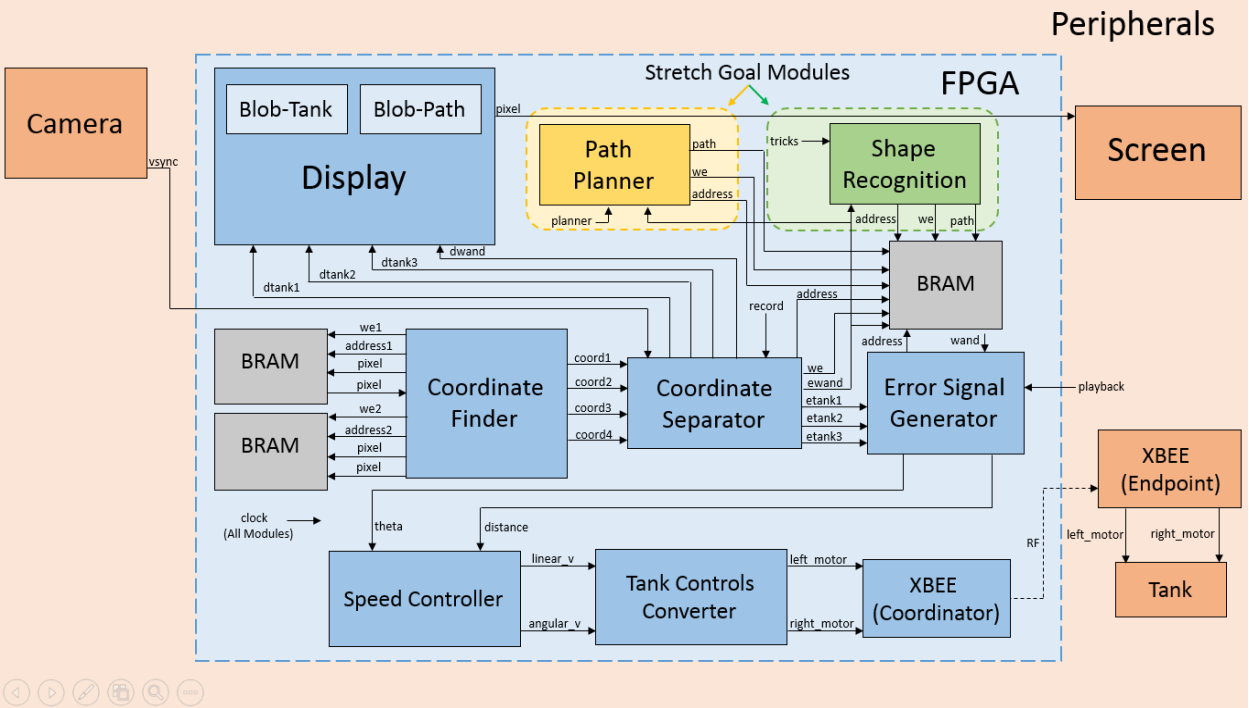


Figure 1

2.2. Signal Flow

The system begins with the input to the camera. The camera's visible light filter only permits infrared light to pass through to the lenses. This light will originate with the IR LEDs placed on the tank and wand. The camera's view is transmitted to the Nexys 4 board using Omnivision's SCCB (Serial Camera Control Bus) protocol. This protocol essentially utilizes the same methods as I2C (Inter IC) whereby a "master" device (the Nexys 4 board in this case) interfaces with a "slave" device (the camera) via a serial data line (a clock signal is also shared between the two). In this system, the Nexys 4 sends configuration data to the camera, and the camera sends its feed to be processed.

When the Nexys 4 receives the camera feed, this information arrives in the RGB 5-6-5 bit format, and is immediately translated into 4-4-4 RGB. This is further translated into a 4-4-4 YCrCb format using a linear transformation matrix, and only the luminance information ("Y") is manipulated from this point forward. We capture information this way because actual color is not maintained in the infrared range, and we make considerable memory savings storing 4 bits of information per pixel instead of 12 bits per pixel.

Before the luminance data is stored, it passes through a binary filter. If the luminance value is above the threshold, the data is stored in the frame buffer as '1111', otherwise, it is stored as '0000'.

The video module then reads from this frame buffer and prepares this pixel data for display on an LCD screen in synch with its respective hsync and vsync signals.

This video output information (essentially pixel data, hsync, and vsync) then passes through to the Coordinate (LED) Finder, which attempts to locate the LEDs on screen by placing personalized boxes around each LED. As data, the boxes consist of left, right, upper, and lower segments, and these boundaries are used to discriminate between separate LEDs, as well as to prevent mistakenly determining one LED to be multiple LEDs. This "box" data, as well as the video output data is then sent to the blob module, while the determined LED locations (encoded in 20 bits: the upper 10 for x-position and the lower 10 for y-position) are forwarded to the Coordinate Separator module.

The Blob module provides tangible realization to the "boxes" by making all of the pixels within their bounds appear white on the LCD screen. This modified pixel data and the hsync and vsync signals are ultimately what pass through to the Nexys 4 VGA module from the Blob.

The Coordinate Separator Module takes the LED locations from the Coordinate Finder, and the overall state of the system (whether it's in playback or record mode) and determines whether a specific LED is a wand LED, or a tank LED. In record mode, the wand LED coordinates are stored in the Wand BRAM. If the system is operating in playback mode, the tank LED coordinates are forwarded to the Error Signal Generator (ESG).

Finding the linear and angular distance between the tank and its next destination is the ESG's job. This "next destination" is a wand LED location pulled from the Wand BRAM. The ESG uses the tank's current coordinates and the next destination to create an angle that the tank needs to turn as well as a linear distance the tank needs to travel to reach that point. This information is forwarded to the Speed Controller.

The Speed Controller uses the error information to determine how quickly to turn to reduce the angle error to zero, and how quickly to drive forward to reduce the linear distance to zero. The angular and linear speeds are sent to the Tank controls converter, which creates a palatable pwm signal to be transmitted over an xbee radio link to the tank's motors.

2.3. Physical Setup

We originally tested our system by mounting the camera off of a tall lab desk as shown in figure 2. The major problem with this setup, and the reason we quickly abandoned it, was that the camera's field of view was so small that the tank would constantly run off screen. In order to

maximize the space in which the tank can move, we instead mounted the camera and FPGA onto the ceiling.

The monitor which displays the location of the IR LEDs, as well as the path drawn by the user, was elevated and connected to the FPGA via a standard VGA cable. On the floor, we marked out the play area by displaying the wand on the monitor so that we could position the wand at precisely the four corners of the camera's field of view and mark those locations with tape. This also provides the user with a secondary reference to use when drawing a path. The following pictures were taken of our final setup.

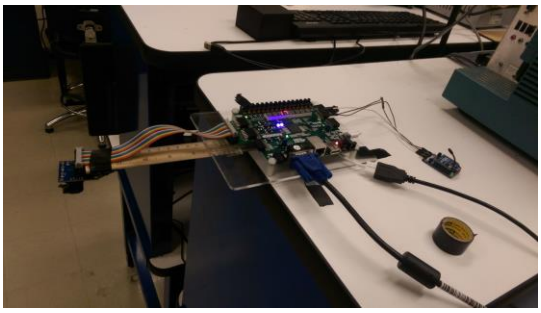


Figure 4

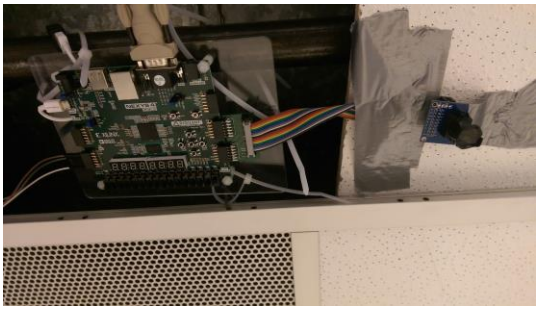


Figure 5

Figure 6

3. Hardware

<u>3. 1. Camera, Nexys 4 board, and LCD Monitor (Andre)</u>

We used an OV7670 camera for capturing images with resolution 640x480, and interfaced it with the Nexys 4 by using code provided by the 6.111 staff. The basic code was able to stream the camera input directly to the LCD monitor by default, so many of the changes made to the incoming image frames were made within this larger video capture structure.

The camera itself is quite noisy, so care had to be taken when devising methods for processing the incoming signal. The simplicity of the method used to detect the IR LEDs led to an implicit sacrifice of robustness. For example, having fixed sizes for the boxes that the Coordinate Finder establishes around the LEDs meant that detection issues would arise if any LEDs were close enough to the camera to exceed the pixel areas of those box boundaries. This would result in a single LED being attributed more than one box, on occasion. In the end, this was an acceptable circumstance because, in our application, the tank does not significantly change its distance from the camera as it moves throughout the camera's optical range. Thus, we were able to find a sufficient "sweet spot" configuration in which LED profiles remained firmly within their designated square boundaries to an appreciable degree, and the system did not confuse its decisions about specific LEDs.

3.2. Tank and XBee (Druck)

The hardware for the tank and receiving circuitry went through several iterations. Our final iteration used a 10" x 5" x 4" remote controlled toy tank with a mounted XBee and external circuitry. The upper casing of the tank, which included the turret, was removed leaving only the bottom frame and treads. The motor for the turret, as well as the tank's antenna and all of its internal circuitry were removed, leaving only the 6V power supply and the two DC motors which controlled the treads.

To control the speeds of the motors we used a series 1 XB24-AWI-001 XBee module acting as a receiver, or endpoint device. XBees have multiple digital I/O pins, as well as two pwm or analog I/O pins, all of which can be remotely set or read from. Because the pins on the XBee are too small and spaced too closely together to fit on a breadboard, we used and XBee adapter kit which allowed us to mount directly onto a breadboard. Additionally, the adapter kit has a voltage regulator that allowed us to power the XBee with a 5V supply, as opposed to the usual 3.3V that XBees will accept. The digital and analog I/O pins on the xbee operate at only 3.3V. In order to power the motors of the tank, which normally operate using a 6V power supply, we connected the pwm outputs of the xbee to the bases of two BJTs, whose emitters were connected to ground. The DC motors were then connected in series with the 6V power supply and the collectors of the BJT's. Figure 5 shows a schematic of the circuit we used.

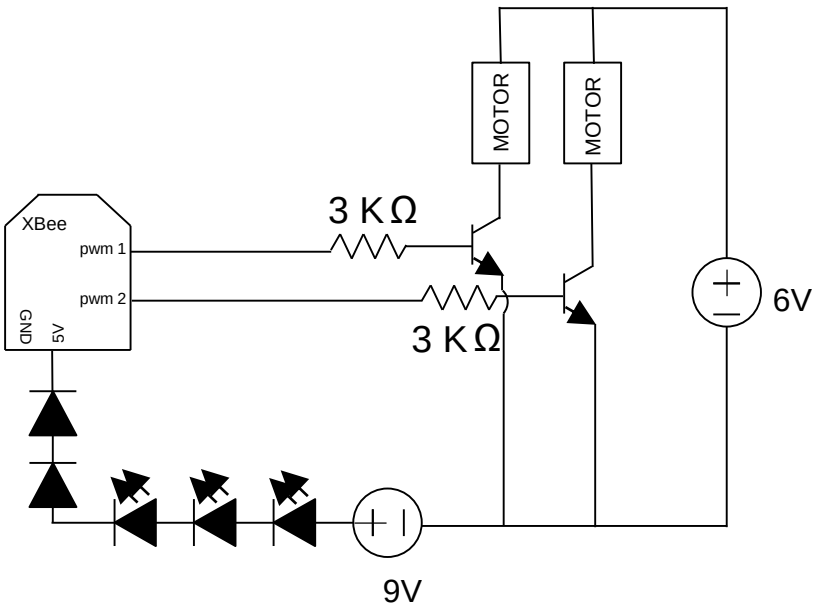


Figure 5

This final circuit was tested on a breadboard, which was then taped directly on top of the tank as shown below.

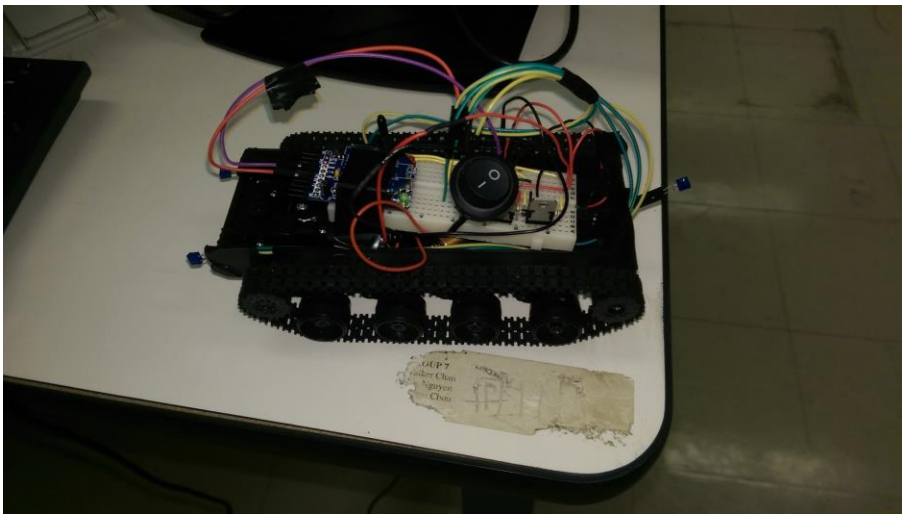


Figure 6

To send instructions to the endpoint XBee, we had another XBee acting as a coordinator connected to the FPGA. The I/O pin JC[0] on the Nexys 4 board output data in UART serial format to the Rx (receive) pin on the coordinator XBee. The connections between the XBee and FPGA are shown here in figure 7.

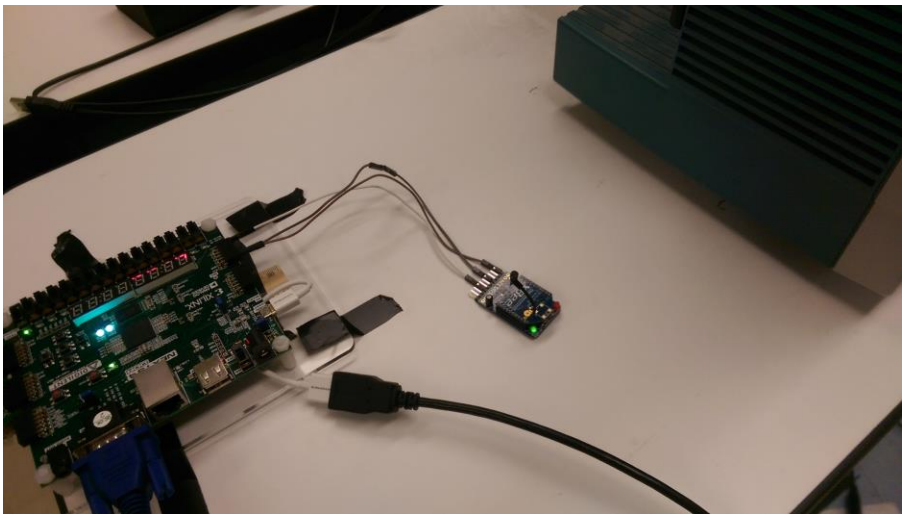


Figure 7

One of our biggest challenges when designing this setup was deciding whether to use the existing tank circuitry and remote control, or use a separate transmitter/receiver and external circuitry. We first tested the setup using the tank's own circuitry and controller. The plan was to connect MOSFETS in parallel with the switches on the tank's controller. We connected the gates of the MOSFETS to I/O ports on the FPGA to simulate physically closing the switches with the FPGA. Our initial setup is depicted below in figure 8.

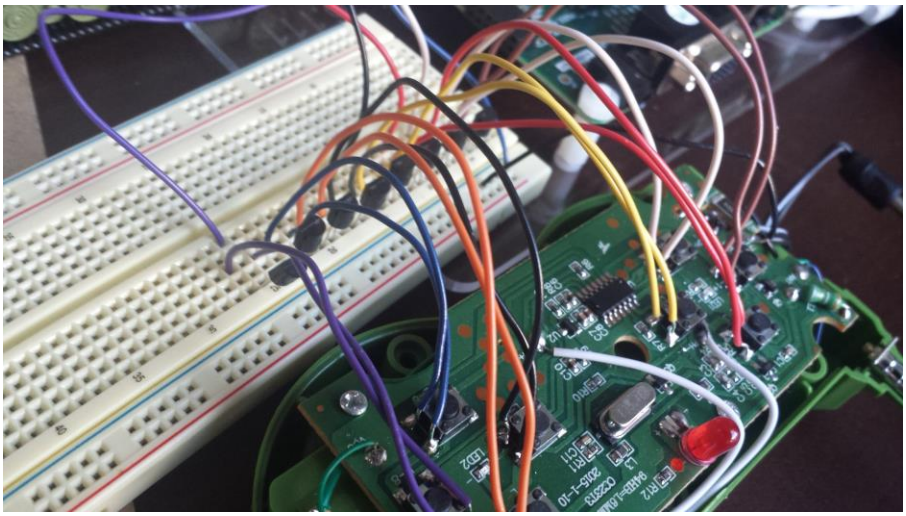
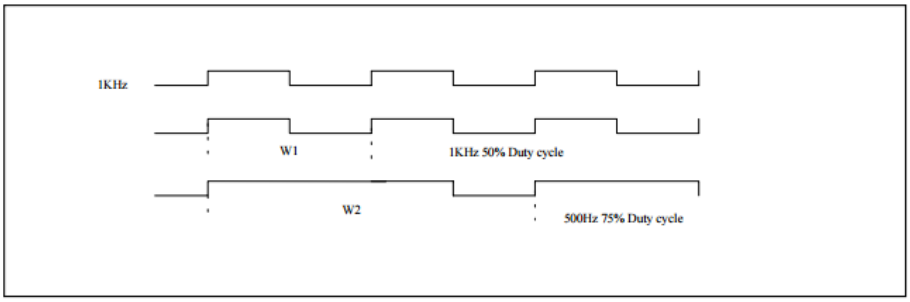


Figure 8

Although we were able to successfully move the tank around using the FPGA with this simple setup, we had very little control over the speed. In order to decrease the tank's speed, we needed to pulse width modulate the voltage being sent to the motors at a reasonable frequency - at least a few kilohertz. However, the receiving circuitry did not allow for this.

The receiver expected 2 types of signals. The first was a square wave with a 1 KHz frequency and a 50% duty cycle, called W1, and another with a 500 Hz frequency and a 75% duty cycle, called W2. Instructions sent to the receiver were formatted as 4 periods of the W2 signal followed by n periods of the W1 signal, where n determines the type of instruction. The square waves W1 and W2, as well the lookup table mapping n to each instruction, are shown below in figure 9.

Bit Format



Data Format

W2 W2 W2 W2 (n) x W1 W2 W2 W2 W2 (n) x W1 W2 W2 W2 W2

Number of Function Codes (n) W1	Function Key	Decode Result
4		End Code
10	Forward	Forward
16	Forward & Turbo	Forward
22	Turbo	Turbo
28	Turbo & Forward & Left	Forward & Left
34	Turbo & Forward & Right	Forward & Right
40	Backward	Backward
46	Backward & Right	Backward & Right

Figure 9

Using this scheme, the best frequency we could achieve for a pwm signal barely exceeded 10 Hz. Because of this we decided to use our own receiver/transmitter pair, and directly supply power to the tanks motors. We chose XBee for their simple interfacing and ability to easily set the duty cycles for its analog pins.

Our original XBee circuit included an L293D H-bridge. The enable pins of the chip would be controlled using the XBee's pwm pins, and the motor driver pins were driven using the XBee's digital pins. We came across an issue with the voltage swing of the motor outputs of the H-bridge. The voltage would only drop down to about 5V and would hit a maximum of 6V. The

problem seemed to originate from trying to use a pwm signal on the enable pins of the H-bridge. When the enable pins were supplied with a DC voltage and we instead connected the pwm signals to the motor driver pins of the H-bridge, we didn't observe any voltage swing issues.

We decided to exclude the H-bridge which meant the final circuit design only allowed for forward movement of each motor. Additionally, the extra circuitry over loaded the motors of our original 90 x 50 x 40mm size tank. The original tank had to be switched out for a larger one.

4. Modules

4.1 Image Processing

4.1.1. Preliminary Processing (Andre)

Before pixel data is stored in the frame buffer BRAM, it undergoes some minor changes. Firstly, the data received from the camera is changed from RGB 5-6-5 bit to RGB 4-4-4 bit because the VGA module of the Nexys 4 board can only handle 4 bits assigned to its red, green, and blue inputs. To tailor the system more to capturing the signatures of colorless IR LEDs, the RGB 4-4-4 data is converted into a single, 4-bit luminance ("Y") value -- referring to the YCrCb, or YUV formats. This step is useful because color has no meaning for this application outside of the visible light spectrum, and we therefore only needed to identify an LED by its brightness (against a black background, thanks to the visible light filter placed across the OV7670 camera lens). The official matrix conversion to create this Y-value is:

$$Y = 0.299*R + 0.587*G + 0.114*B$$

However, these decimal coefficients are somewhat costly to create, so, in our implementation, values that could be created by bit-shifting were favored for this conversion instead:

$$Y = 0.25*R + 0.625*G + 0.125*B$$

From here, the set of Y-values associated with the pixels seen by the camera constituted a grayscale image. For our system, we only needed to observe the LEDs within the camera's view, so a threshold of decimal value 10 was applied such that if the Y value was above this threshold, the pixel data saved to the frame buffer BRAM would be '1111', and if the Y value was below 10, the data would be '0000'.

The thresholding and "Y" conversions were tested by viewing their outputs on an LCD. As soon as the desired grayscale and black and white images were observed, the effort was deemed a success.

We first approached this step thinking that the noisy camera would be unmanageable without mean or median averaging, but we soon realized that the camera noise seemed to be localized near edges of colors. For example, the centers of the IR LEDs were almost entirely consistent

in their appearance on screen. It was only near the edges of the LEDs, where the brightness was much less consistent, that noise became an issue. This was rectified with some tweaking of the thresholding value. The alternative would have been to implement a 3x3 sliding window with row buffers so that the averaging could be applied. In that case, the latency before the data from a single pixel reached the LCD would have been over a thousand clock cycles at worst. The solution we settled on made the latency only a handful of clock cycles.

4.1.2. Video Display Module (Andre)

In the code provided by the 6.111 staff, there is a video display module that transfers the pixel data from the frame buffer BRAM to the Nexys 4 VGA. The video module receives the hsync and vsync from the camera so that it knows when to apply blanking to the LCD screen, the module also keeps track of the hcount and vcount of each piece of pixel data it reads from the frame buffer BRAM.

To simplify timing aspects of the image processing, the decision was made to tack on the subsequent modules (Coordinate Finder and Blob) in series with the Video Display Module (VDM) such that the Blob module would now interface with the Nexys 4 VGA instead of the VDM. The vcount, hcount, hsync, vsync, and blanking (which indicates whether the LCD should be blanking or not) signals, and the pixel data from the frame buffer all pass from the VDM to the Coordinate Finder to the Blob. All video signals leaving the VDM are made synchronous, and enter the Coordinate finder; all video signals leaving the Coordinate Finder are synchronous and enter the Blob to be displayed via interface with the VGA. This approach makes pixel data latency a handful of clock cycles.

4.1.3. Coordinate Finder (Andre)

After the video signals are output by the VDM, the Coordinate Finder uses this data to make estimates about the locations of LEDs in the camera's view.

First, some assumptions are made about the profile of each LED. Namely: the LED will appear roughly circular to the camera, and the edges of the LED profile will remain relatively well-behaved (they won't flicker too erratically due to the thresholding in the preliminary processing phase). The LEDs' adherence to these two criteria permitted the replication of an identification approach for a similar project from Fall 2015 (the "Autonomous RC Car" by Chan, Gomez, and Myanganbayar).

The approach is as follows: as hcount and vcount scan across the camera image from left to right and top to bottom, the Coordinate Finder waits until it receives a single white pixel (with data '1111'). Because each LED's profile is essentially circular, this first pixel will be the very top of an LED. The module stores the hcount and vcount of this pixel in the first 20-bit LED position data register.

That single pixel is attached to a single LED that is made up of tens or hundreds of pixels (and they are all white thanks to the thresholding completed earlier). We don't want to pick any of the other white pixels associated with this LED and mistake them for a new, separate LED, so the Coordinate Finder needs to know to group all of these pixels together.

The module achieves this by crafting a square around that first white pixel's position. This square has four values: two horizontal values calculated using hcount (left bound and right bound), and two vertical values using vcount (upper bound and lower bound). These will serve as boundaries such that if another white pixel is within this square, it will not be counted as a new LED. Likewise, the first white pixel found outside of this square will be seen as the start of a separate LED.

The Coordinate Finder expects there to be at most four LEDs on the screen at once, so there will be four sets of boundaries that will each define a square containing a single LED, as well as four 20-bit LED coordinate registers. The square boundaries, alongside the video signals from the VDM (hcount, vcount, hsync, vsync, and the pixel data) are all sent to the Blob module. The 20-bit LED coordinates are forwarded to the Coordinate Separator.

Lots of testing went into this module. We first created a test bench that we fed theoretical white pixels to ensure that the correct boundaries were being created as well as the correct 20-bit LED coordinates. Once this was working, we tried different instances where we added LEDs within the formulated boundaries and outside of the boundaries until we got the maximum four LEDs. When this functionality was working, the module was finished.

The first design of the Coordinate Finder would have used more complex image processing techniques such as skeletonization or center of mass calculations to determine the centroids of the LEDs. We determined that these would have been too taxing a series of operations to complete in real-time while streaming video to the LCD. More delays would have been necessary as well (hundreds of clock cycles at the least) to have all of the requisite pixels for either operation. Although the approach we settled on may have been less capable of incorporating more complicated processing steps, the choice worked splendidly in our application.

4.1.4. Coordinate Separator (Andre)

Taking in the LED locations discovered by the Coordinate Finder, the Coordinate Separator determines which LED corresponds to the wand, and which corresponds to the tank. The separator also takes in the overall state of the system (record or playback). If the coordinate separator is in record mode, it counts the number of LEDs present and waits until there is just one. Then it starts writing the location of this LED (which it assumes to be the wand) to the Wand BRAM once per frame. This continues until the LED turns off, or if there is more than one LED on screen, or if 15 seconds have passed (which is when the buffer fills up).

At the same instant the wand coordinate is saved in the Wand BRAM, the Coordinate separator works to fill up another BRAM called the Full-path BRAM. The Full-path BRAM is used to display the entire wand path as it is drawn using the wand. Each address in the BRAM corresponds to a unique pixel hcount and vcount where address = hcount + 640*vcount. Each frame (when a new wand location is written to memory), a series of twenty five pixels in the Full-path BRAM (and around that wand location) are given a value of '1'. This is to indicate to the Blob module which pixels to color magenta.

If the Coordinate Separator is in playback mode, it waits until it counts three LEDs, and then assumes that these constitute the tank. The module then exploits the strict ordering of the LEDs (the first LED has the lowest vcount, the second LED has the next highest, and the third LED has the highest) and their arrangement in an isoceles triangle to find the tank's front LED. Mathematically, the square of each segment of the triangle is determined, and the LED at the vertex of the longest segments is chosen as the front LED. Once the front LED is found, it, alongside the two back LEDs and a "ready" signal, are sent to the Error Signal Generator.

Testing for this module was done using an LCD screen and the hex display on the Nexys 4. The main goal was to ensure correct functionality in the playback mode, so we displayed which LED the module believed was in front.

4.1.5. Blob Module (Andre)

The Blob module's purpose is to change the color of any pixel found to be within a given set of boundaries. The incoming LED boundaries are all colored white by this module for display on the LCD. The Blob also interfaces with the Full-path BRAM populated by the Coordinate Separator Module. Each location in that BRAM corresponds to a pixel location. If the Blob reads a location written with a '1', the Blob makes the pixel with the corresponding hcount and vcount magenta.

Finally, the Blob module is where the VDM data is finally output to the VGA. Hsync and vsync remain the same, but the pixel output is modified by the Blob module if it meets either of the above criteria.
This was a difficult module to test because it is at the end of a series of modules, and the main form of testing was whether anything showed up on screen. This module required a lot of backtracking to catch issues with previous modules that only arise in the integrated state (when there was a continuous stream of data that needed to be sifted through). It turned out that the main issue was a misunderstanding about the parity of a single signal. The lesson here was to check your input signals more carefully early on.

4.2 Tank Control

4.2.1. Error Signal Generator (Druck)

This module takes in the coordinates of the three tank LEDs as well as a ready signal from the coordinate separator. The purpose of the ready signal is to let the module know that the tank coordinates have been updated and will not change until the next camera frame has been processed. This allows us to stop any glitches from propagating through our system and will also be used in later modules.

The tank coordinates are each 20 bit unsigned integers, with the 10 most significant bits representing the x-coordinate and the 10 least significant bits representing the y-coordinate. The module also takes in a playback signal. This signal goes high when the user puts the system in “playback” mode, and tells the error signal generator to begin calculations. Finally, the module also has a 20 bit input for the wand coordinate. When the module enters playback mode, it specifies the index in the stored path that it wants to pull a wand coordinate from using an output called address. Address is wired directly to the address input of the BRAM and the output of that memory is wired to the wand input of the error signal generator.

The purpose of this module it to take the desired angle and coordinate of the tank, and subtract from that the actual angle and coordinate of the tank. The two error signals generated in this process, theta and distance, are used in the next modules to set the speeds of the tanks motors. The distance is an unsigned 10 bit integer ranging from 0 - 800, while theta is a signed 10 bit integer ranging from -180 - 180.

Calculating the distance is as straightforward as applying the Pythagorean distance formula. The distance is calculated between the front LED of the tank and the wand, using the following formula.

$$\text{Distance} = \text{sqrt}((\text{wand\_x} - \text{tank\_x})^2 + (\text{wand\_y} - \text{tank\_y})^2)$$

To perform the square root operation, we used Vivado’s CORDIC IP square root function with default presets.

Our approach to calculating the angle was to find define the midpoint of the back two LEDs of the tank as an origin. Next we found the angles of the vectors formed by the front tank LED with the origin and the wand with the origin, and took the difference of these angles. To go from the x and y values of a vector to the angle of that vector, we used Vivado’s CORDIC IP arctan function.

Finding the midpoint of the back two LEDs and the x and y components of the two vectors was simple, but the main challenge came in formatting the integers for Vivado’s CORDIC IP. This module required that the x and y inputs take on values between -1 and 1 using two’s complement fixed point representation, with a 2-bit integer width.

To do this, we left bit shifted the original values, then divided both values by the larger of the two. This gave us one component with a value of 1 or -1, and the other with an absolute value less than or equal to 1.

The process of finding distance and theta takes many clock cycles, but because we only need to update these values at a rate of 30 HZ, the frame rate of the camera, we didn't encounter any timing issues.

Finally, the module is structured as an FSM. The first two states consist of an idle state, when the system is not in playback mode, followed by another waiting state, where the module waits for the coordinate separator to update the tank coordinates. Then there are a few states which transition after only one clock cycle and perform the calculations, and another waiting state that waits for the arctan and square root modules to finish their calculations. Finally, after all of the calculations are finished, the module updates distance and theta and asserts a ready signal for the next module.

To test this module, we output distance and theta to the on-board hex display, and fed in values for the tank and wand coordinates to make sure the outputs matched hand calculations. Because many of the modules used for tank control were tested using the hex display, we created an extra module, used only for testing purposes, to display values on the hex display in decimal. This module, called hex_to_dec, allowed us to quickly verify outputs by eye.

4.2.2. Speed Controller (Druck)

The distance and theta outputs of the error signal generator are wired to this module which uses them to calculate the desired linear and angular velocity of the tank. These velocities are represented as values between 0 and 255 and between -255 and 255 respectively. Similarly to the last module, the actual time it takes to perform the calculations is not a concern because we updating the tank's speed at such a low frequency.

The first step in the calculation is to map the values of distance and theta to the ranges output by the speed controller. For example, a theta of 180 will be mapped to 255 and -180 to -255, etc. This process was fairly straightforward, and the only difficulties surfaced when trying to perform arithmetic and comparisons using signed integers. At times, verilog interpreted integers in unexpected ways. For example, we found that the following if statement would get executed even though the argument is clearly always false.

if(-'d45 > 'd180) begin

This meant we had to be extra careful with signed numbers, and the general strategy we took was to used unsigned temp registers for all of the calculations and use if statements to assign the proper sign after all of the calculations were finished. Furthermore, we specifically used bit

operations to convert from positive to negative and vice versa, as opposed to using multiplication.

Our final implementation of the speed controller acted as a proportional-integral controller. We based the PI control off of the following difference equation.

$$y[n] = y[n-1] + Ki*e[n] + Kp(e[n] - e[n-1])$$

Where y[n] is the current output, e[n] is the current error measurement, and Ki and Kp are the integral and proportional constants. In our case, we had two sets of difference equations, one for the linear speed output and another for the angular speed output.

In order to simplify the conversion from angular and linear speed to actual duty cycle values for the motors, we separated movement into purely rotational and purely linear modes of operation. We did this by setting a threshold for theta, greater than which the tank would only turn, and less than which the tank would only move forward. To facilitate this we added two more outputs, turn and forward, which let the tank controls converter know which mode of operation we were in. Here we traded off performance for simplicity, but if we were to continue this project further, one major improvement to this module would be to eliminate the need for these separate states.

Another improvement to performance would be simply choosing better integral and proportional constants. Currently, we still notice some oscillations when the tank is turning towards the next point in the path due to not optimizing the constants we used.

Once again, just like the error signal generator, this module consists of a main FSM to satisfy timing constraints for calculations and for other modules. In the first state, the module waits for the error signal generator to assert its data_valid signal, letting us know that theta and distance won't change mid-way through our calculations. The next few states simply carry out the calculations similarly to the error signal generator. Finally, in the last state, we check to see if tank_ready has been asserted by the tank controls converter module. If not, then we jump back to the idle state.

For our first pass of this module, we only implemented a simple proportional controller. This helped us to get our overall system functioning faster and to debug any issues with the fsm and handshake signals before trying to implement more complex control. Just like with the last module, we originally tested the module using the on board hex display and the hex_to_dec module to verify that various inputs gave the correct outputs. As expected, when we first tested our overall system with the proportional controller, we noticed significant oscillations when setting the angle of the tank.

Another possible improvement would be to incorporate derivative control. However, for systems like ours, where we don't need incredibly precise control, PID controllers generally don't add any noticeable performance when compared with a well made PI controller.

4.2.3. Tank Controls Converter (Druck)

In order to get any meaningful behavior out of the tank, we need to convert a desired speed into the appropriate duty cycles for the pwm signals which are applied to the motors on the tank. This module takes in the 9-bit values of the linear and angular speeds from the speed controller and determines those duty cycles for each motor. Additionally, it satisfies the timing constraints imposed by the baud rate of the serial connection to the XBee.

Because we have separated the tank's movement into two modes of operation, the conversion from linear or angular speed to duty cycle becomes simple. First, if we are in turn mode, one motor will receive no power, while the power the other one receives is directly proportional to the angular speed. We first look at the sign of the angular velocity and determine which motor's duty cycle should be set to zero using an if statement. If the angular velocity if positive, set the left motor duty cycle to zero, otherwise the right one to zero. In that same statement, we map the magnitude of the angular velocity from a value between 0 and 255 to a value between 0 and 1023. If, instead, we are in forward mode, we simply perform the same mapping for the linear velocity and set both motors' duty cycles to that value.

To satisfy the timing constraints of the serial communication to the XBee, this module is configured as an FSM with three states; idle, left motor enable, and right motor enable. In the idle state, we assert tank_ready to let the speed controller know we are ready for new speed values, and wait for speed controller to assert its own data_valid signal. We then transition to each motor enable state, at a frequency of 44Hz. 44Hz is the maximum rate at which we can send an AT command request to the XBee. In each of these states we choose which pin on the endpoint XBee we want to set, and the duty cycle. These outputs are used by the xbee_write_pin module to construct and send the AT command request.

Our original pass of this module had more states. These extra states determined the values we wanted to set for the digital pins on the endpoint XBee. However, in our final design we removed the H-bridge from the on board tank circuitry, and thus removed the need to specify any digital pin values.

This module was tested in conjunction with the xbee_write_pin module. The linear and angular speeds were set using the switches on the FPGA, and the up button was used as the data_valid signal to make the module send.

Currently, because we are only sending AT command requests at 44 Hz, this module can only update the state of the tank's motors at a rate of 22 Hz. A possible improvement would be to increase the baud rate on the xbee to allow for faster response. The tank controls converter and xbee write pin modules have parametrized the baud rate so that theoretically this should be as simple as changing those constants. However, we have not tested adjusting the baud rate. In addition to changing the verilog, the coordinator and endpoint XBees need to be reconfigured for a higher baud rate.

4.2.4. XBee Write Pin and UART Send Byte (Druck)

These are the modules that directly interface with the coordinator XBee. XBees have two modes of operation, AT mode and API mode. In AT mode, whatever inputs one XBee sees on its Rx pin are automatically transmitted and output to the Tx pin of any other XBee that it has been configured to talk to. In order for two XBees to communicate in this mode, they need to be configured to be on the same Physical Area Network (PAN) and the destination address of each must match the source address of the other. If, for example, we only needed to control one motor, using the XBees in AT mode would greatly simplify the control scheme. We could simply apply the desired pwm to the Rx pin of the coordinator. This would also be useful because we could adjust the frequency to suit whatever motor we were using, as opposed to using the analog XBee pins which are automatically set to a frequency of 15 KHz.

Unfortunately, we of course need to control more than one motor, which makes using API mode a better option. In API mode, the user can change the settings of the endpoint XBee, such as setting a digital or analog pin to output a specific value or act as an input which can then be sampled. It's only through API mode that we can set the duty cycles of the pwm pins of the endpoint XBee. The coordinator in API mode expects a data packet which contains, among other things, the type of request, the length of the packet, the pin number to change, the value to set it to, and a checksum. The format of the specific command we are using for this project looks like this.

API format for Remote AT Command Request	Byte	Example	Description
	0	0x7e	Start byte – Indicates beginning of data frame
	1	0x00	Length – Number of bytes (ChecksumByte# – 1 – 2)
	2	0x10	
	3	0x17	Frame type - 0x17 means this is a AT command Request
	4	0x52	Frame ID – Command sequence number
	5	0x00	64-bit Destination Address (Serial Number)
	6	0x13	MSB is byte 5, LSB is byte 12
	7	0xA2	
	8	0x00	0x0000000000000000 = Coordinator
	9	0x40	0x000000000000FFFF = Broadcast
	10	0x77	
	11	0x9C	
	12	0x49	
	13	0xFF	Destination Network Address
	14	0xFE	(Set to 0xFFFE to send a broadcast)
	15	0x02	Remote command options (set to 0x02 to apply changes)
	16	0x44 (D)	AT Command Name (Two ASCII characters)
	17	0x02 (2)	
	18	0x04	Command Parameter (queries if not present)
	19	0XF5	Checksum

Figure 10 (Rhysider, 2012)

The UART send byte module is initialized within the xbee write pin module. It takes a single byte as input and outputs that byte is serial UART at a specified bit rate from LSB to MSB. UART uses an idle high and surrounds each byte by a low start bit and a high stop bit.

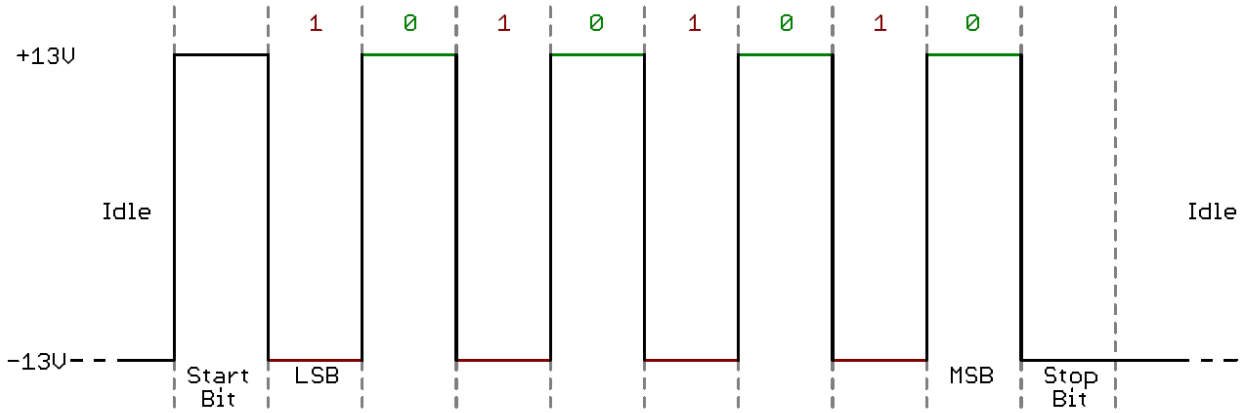


Figure 11 (SparkFun)

The UART module consists of an FSM with an idle state, in which it sets the output to 1'b1 and waits for the signal to begin transmission. It has a counter which is used to generate a pulse signal at the same frequency as the baud rate. After the module exits its idle state, it transitions from state to state on the rising edge of this pulse signal, outputting the corresponding bit at each state.

The XBee works in exactly the same way except that every state outside of the idle state is sending the appropriate byte for the AT command request. Because each byte is 8 bits plus a start and stop bit, the frequency of the pulse signal used for state transitions here is 10 times slower than the one used for UART send byte.

This module was simply tested by hardcoding in a pin number and value and using one of the buttons on the FPGA to act as the send signal. Once we were able to set a pin on the endpoint XBee to a specific value, we knew the modules worked. As mentioned with the previous module, one improvement would be to increase the baud rate.

5. Conclusion

The LED Pathfinder has been an exciting project for our team. We chose to complete this project because of our interest in learning how to interface with different types of hardware and love of autonomous systems. Hacking an RC tank and video camera has provided ample opportunity to work around real-world non-idealities in technology. We faced many challenges and had to work through multiple iterations of the design process for many of our modules and hardware.

In the end, we were successful in meeting our expected goals. We created a system which allows the user to freely determine a toy tank's path in a fun and creative way. Improvements we could make to our project would be increasing the bit transfer rate of the XBees to allow for faster feedback control, and fine tuning our speed controller for additional performance.

Additionally, there are improvements to be made in the video processing side of the project. We could quarter the amount of memory being used to store frames from the camera by storing only one bit for each pixel. Furthermore, we could improve our coordinate separator module by adding the ability to distinguish between the tank and wand when both objects are within view of the camera.

Even beyond that, there is still room to expand on the fundamental capabilities of our system, such as adding a module to recognize specific shapes drawn by the user and perform predetermined behaviors based on those shapes. This project still has so much more potential, and has been an amazing learning experience for the both of us.

Works Cited

Rhysider, Jack. (2012, November 30). *XBee S2 Quick Reference Guide/Cheat Sheet and Video Tutorials to Get Started* [Blog post]. Retrieved from https://www.tunnelsup.com/xbee-guide/

Jimbo. *Serial Communication.* Retrieved from https://learn.sparkfun.com/tutorials/serial-communication