FPGA Autonomus Vehicle

Juan Antonio Luera
Department of Physics
Massachusetts Institute of Technology
Cambridge, USA
jluera@mit.edu

Juan Angel Luera
Department of Physics
Massachusetts Institute of Technology
Cambridge, USA
j_luera@mit.edu

Manuel Valencia
Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, USA
manuelv@mit.edu

***Abstract*—The integration of LIDAR sensors and Inertial Measurement Units (IMUs) in robotic navigation can provide enhanced data sensing processing and energy efficiency. Our project aims to explore the robustness and responsiveness of FPGA in handling the demands of these sensors. Leveraging the parallel processing capabilities of FPGAs, our system is designed to simultaneously interpret continuous LIDAR data and calculate the car's real-time orientation and position using IMUs. This approach ensures an efficient and real-time reaction to surrounding obstacles. The minimally viable project consists of implementing the UART protocols to interface with an IMU and a LIDAR camera and use this data to navigate an RC car. The FPGA will process information from the IMU and LIDAR cameras and update the trajectory of our RC car if an object is detected. The goal is to recognize obstacles in lanes and maneuver around them. Our stretch goal is to implement an algorithm for room mapping using obstacle avoidance and position control. Our aim is to demonstrate how FPGA's timing and parallel processing can optimize real-time navigational tasks.**

***Index Terms*—FPGA, LIDAR, Autonomous Navigation, Position Control**

I. Introduction

Our project aims to integrate encoder, LIDAR, and IMU sensor data to autonomously control a small surface vehicle. The integration of these sensors data is widely studied today for various robotic and self-navigation systems. We hoped to accomplish two autonomous navigation tasks: Lane navigation and LIDAR Visualization. This paper will first go over all the components used in the project, followed by the communication protocols created to interface with the various sensors. It will then explain how we were able to visualize LIDAR and IMU data, the driver board interfaced with, and finally the motor controller made.

Please find all the code and CAD files in this GitHub: https://github.com/El-Guapo2024/6.111

II. Physical Construction

- The frame of the car itself is made from laser-cut 5 mm wood or acrylic and 50 mm hex standoffs.
- For the motors we used 18.75:1 Metal Gear 12V DC motors (37Dx68L mm) with 64 CPR Encoders mounted using aluminum L brackets.
- The wheels were 3D printed to attach to the aluminum motor shaft hub. The design and shape were those of previous wheels acquired whose rubber threads we wished to use.
- The electronics consist of a LIDAR, IMU, HC-SR04 Ultrasonic range sensor and Arduino Mega 2560, quad motor shield, voltage level shifters, 74HC14 Schmitt Trigger, 11.1 V Li-PO Battery, and 7805 Linear Voltage Regulator.
- The LIDAR sensor contains a laser that is used as a distance sensor and is rotated at 12 Hz and can retrieve up to 800 individual samples of distance and angle per rotation. We can use this to make a 2D Point Cloud Data image of the surroundings.
- The IMU is used to keep track of the heading and correct the error in movement when combined with the motor's PID controller. The IMU we will be using is the BNO055 Intelligent 9-axis absolute orientation sensor on the Adafruit breakout board.
- The HC-SR04 Ultrasonic range sensor can detect distances to obstacles and is used in place of the LIDAR sensor when not available.
- The Motor shield takes in 4 PWM signals (2 per motor) One goes to the PWM HIGH the other goes to the PWM LOW of the H bridge configuration for each motor. There is a 5v and 3.3v power in for several ICs and a 12v power in for the motor drivers. Each motor also has 2 encoder signals although we only need one for the precision we need.
- The Level shifter allows communication between the motor driver, the FPGA, and the Arduino since the Motor driver and Arduino use 5-volt Logic while the FPGA uses 3.3v.
- The Linear Voltage Regulator will be used to lower voltage from 11.1 to roughly 5 volts to power all the electronics that require 5v power.
- The LI-PO battery is an 11.1-volt battery we will use to power the motor driver, motors and 5v ICs.

III. UART Communication and IC Protocols

A. UART Communication with FPGA

The FPGA will need to communicate over UART with the LIDAR, IMU, and Arduino for testing. The modules for rx and tx communication are based on the 8N1 standard (8-bit data, no parity bit, 1 stop bit). Each IC uses different protocols for communication and different lengths of responses. Our module for rx and tx will read or write a byte of data at a time at a

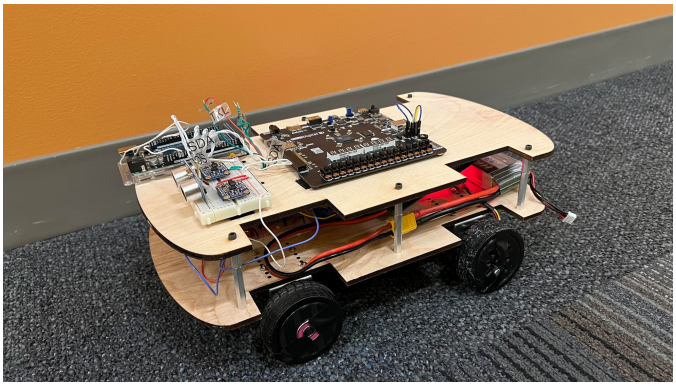

Fig. 1. Assembled FPGA Vehicle

baud rate set through a parameter on initialization. All of our sensors used 115200 baud rates.

B. LiDAR UART Protocol (Antonio)

For the LIDAR, we made use of UART RX and TX Protocol. The Lidar uses 8N1 and a baud rate of 128000bps. This LIDAR uses 3.3 volts, so we are able to directly connect from the FPGA to the Lidar for communication.

The Protocol to communicate is broken into two main modules the Start Protocol and the Scan Protocol. The start protocol will start the LIDAR and set it to scan mode. The scan protocol will scan the dots and store the data in a BRAM.

1) StartScan Protocol: The Scan Protocol has two main states: send command and wait response. The send command state will send a sync byte followed by the startScan Command. It will then wait for the response. It should receive two commands to confirm is scanning. If it receives the correct commands, it will move into the StartScan modules, if it fails it will repeat this module.

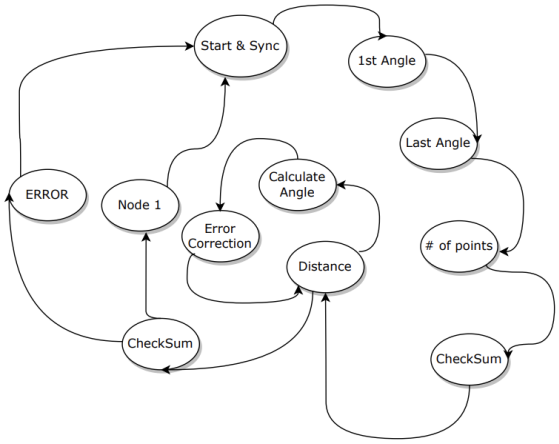

Fig. 2. Lidar UART PROTOCOL

The Scan protocol will listen periodically for the information from the LIDAR sensor. It will then go through a roughly 12-step process to produce 2 16-bit signed logic values, one for the angle in degrees where 23040 corresponds to an angle of 2π. The diagram above shows the flow of the code where each bubble corresponds to two bytes being sent, the first, the most significant Byte, followed by the Least significant byte. The first step is to wait for a specific command to start and make sure we are receiving the correct data. This step is important since the LIDAR can also send other kinds of data such as health. From here is fairly straightforward we receive the First Angle, Second Angle, number of points, and checksum. If the checksum fails, then we fail we go to Error state and repeat the protocol. If we are successful then we proceed with the next phase. We will receive N number of distance byte based on the 2 bytes received in the previous phase. From here, we run the move to calculate the angle of that specific point; for this, I made a module that will make use of the divider to find the interval angle. The internal angle is just the abs(LastSampleAngle-FirstSampleAngle). We wait until the end of the previous step, and we proceed to perform error correction. The error correction is based on the formula provided by the manufacturer and its purpose is to help with the inconsistent data since the LIDAR is not very accurate. We need to implement the formulas shown in the image. We start by calculation the angle of the point and find the angle for the current point. The next step is to find the AngleCorrectForDistance. At first, my attempt was to use Arctan cordic; however, it was quite difficult to get all the correct transformations. In the end, I implemented this by making a Python script that would map all the possible input to the output of the equation. Since I knew that the only input that varies is the distance and we can make use of the fact that is bounded, we can easily run through all the possible inputs and outputs. I then made a Bram using an input map to a palette Bram similar to Pop Cat. From here, we can calculate the AngleCorrectForDistance. Lastly, I run the second equation or variation of the second equation. From here, we can calculate the corrected angle and distance. This is then stored in the Bram. If we will still haven't done the N iterations, we repeat. After we reach the N iterations we move into CheckSum. If it works out, we repeat the whole protocol, and if it fails, we move into ErrorState and set the error flag to 1. Then repeat the protocol

```
AngleCorrectForDistance = (int32_t)((atan(((21.8
        155.3) / (node.distance))) * 3666.93);

    node.angle = (((uint16_t)
    (FirstSampleAngle + sampleAngle
    + AngleCorrectForDistance +23040))
    << LIDAR_RESP_MEASUREMENT_ANGLE_SHIFT)
    + LIDAR_RESP_MEASUREMENT_CHECKBIT;
```

In this part, I used 3 IPs: signed integer division, arctan, and ILA(This is the part I am currently testing). The next

step is to convert from polar coordinates to cartesian. I plan to do this using the Cordix sin and cos IP. All the code can be found in the GitHub under UARTPROTOCOl.

C. IMU UART Protocol (Manuel)

The BNO055 allows for both I2C communication and UART communication. For UART communication, the command and response structure is based on reading and writing to the registers of the microprocessor on board that handles collecting sensor data and performing the data fusion to get reliable readings on the IMU's orientation. One first sends write commands to set up the IMU before sending read commands to read the information in the registers containing the orientation data.

On boot up the BNO055 needs 80ms before being able to receive the first command. You then needs to set the OPR MODE register to NDOF mode for data fusion to be turned on. This is done by sending the command:

0xAA 0x00 0x3D 0x01 0x0C

0xAA signifies the start of a command. Followed by 0x00 for writing or 0x01 for reading. 0x3D is the register for OPR MODE, 0x01 is the number of bytes to be sent, and 0x0C is the value for NDOF mode. The response structure for writing to a register in the IMU is:

0xEE 0xXX

If the message was received it will respond with the first byte being 0xEE followed by 0x01 if successful. The second byte will be a different value for a different type of error encountered.

After setting OPR MODE, I set the UNIT SEL register (0x3B) for the units I want the information to be returned in. For degrees and the range to be from +/- 180 degrees following right-hand rule the command written was as follows:

0xAA 0x00 0x3B 0x01 0x10

Once done the IMU needs to calibrate before sending accurate orientation headings. This can be done by reading the CALI STAT register (0x35) to see if the first two bits which represent system fusion on have value 2'b11. The command sent is:

0xAA 0x01 0x35 0x01

For read commands, no data is sent but one specifies the length of data one wants back. The response will either start with 0xBB followed by the data in the register or 0xEE followed by an error type byte.

Once the calibration status is confirmed then one can read the registers for the MSB(0x1A) and LSB(0x1B) of the current Euler x orientation value which signifies the heading of the IMU. The full data sheet for the BNO055 can be found using the link in the references.

D. Arduino UART Protocol (Manuel)

For testing, we also set up all our sensors to interface with an Arduino MEGA 2560 and send the data from the

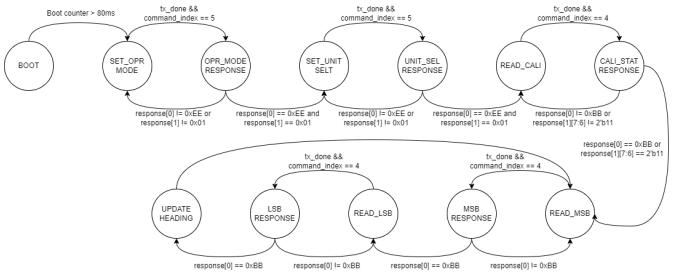

Fig. 3. IMU UART PROTOCOL

various sensors to our FPGA using UART communication. We chose the MEGA since it allows for multiple serial communications as well as I2C for the IMU. The protocol for the Arduino communication is a simplified version of the UART communication with the IMU and follows the same 115200 baud rate 8N1 standard. The Arduino is the only sender that periodically sends the latest sensor values. The FPGA checks the message which is 6 bytes wide and makes first is in a state where the motors won't be enabled until the calibration status on the IMU is set. Then it will send an enable signal to the motor controller module and will update the heading and distance values on each new response that is valid. The first byte of the response is a validating 0xAA signal. The next is either 0xFF if the IMU or other sensors are not calibrated or 0xBB if they are. The next two bytes are the MSB and LSB values of the IMU reading and the last two bytes are the MSB and LSB of our Lidar or Ultrasonic distance sensor. The code for the Arduino sensor communication and FPGA communication can be found in the git under the sketch file named FPGA COMMS.

IV. LIDAR and IMU Visualizer(Antonio)

Fig. 4. Compact PNG

Visualization made use of the previously existing code for HDMI signal handling but removed all the camera functionality. I then made a similar structure to that of the camera. The LIDAR data came from an Arduino, and instead of Popcat, I used the compact.png. My first step was creating a compact binary map of the letters, numbers, and other characters. I found a PNG of characters and then made some Python code to remove all the unnecessary spaces. I did other optimizations, such as using a gray scale and setting all the bytes to either black or white. Then, my last step was to sample down to

two colors and make image.mem file with only zeros or ones. Where black is 0 and white is 1. This step is essential since we are able to sample down to only roughly 1 kilobyte of BRAM. Then I made a module that will allow me to select a display a single letter by only displaying specific sections of the Bram. After that, I made a module that will allow me to display up to ten letters and any specified location by single letter module. This was mainly used for debugging, but it can be seen on the demo displayed "Visualizer". The letters are still slightly rough since I did not fully pipeline this step since it was mainly used to display values.

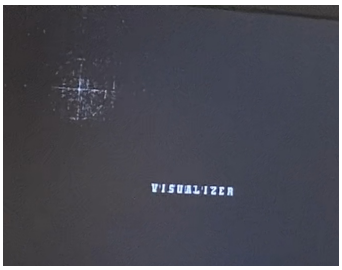

Fig. 5. Visualizer Display

Since my LIDAR stopped working the week the project was due, I was unable to get direct readings from FPGA to the visualizer. As a backup I received a similar LIDAR from Joe, I did not have enough time to write the protocol in FPGA or write error-correcting code. However, I wrote a UART protocol between the Arduino and the FPGA. I send the Sync byte AA, then I send BB for new data followed by 4 bytes [xpos, xpos,ypos,ypos]. Otherwise, I send EE if a new frame is made. I then interpret this on the FPGA if new data is given I will map it to the frame buffer, and if new frame is received i clear the screen. I have a similar mechanism to the camera where I sample from the frame buffer at my own rate. In the end, I can see a live plotting of all the points and a reset every time we have a new frame.

V. Interfacing with motor driver board(Angel)

To interface the motors with the FPGA we used a motor driver shield from 2.74. First we reversed engineered the board to decipher which port produced each signal and needed what inputs. We discovered that the driver needed 3 different voltage sources 12V (for motor power/totem pole + some logic), 5V(encoder power + some logic), and 3v (logic). To produce these supplies we had to use an external power supply, 7805 chips, and Buck converters. Once we figured out the power electronics for the driver we needed to engineer a circuit for control purposes. Initially we had plan to use the following configuration:

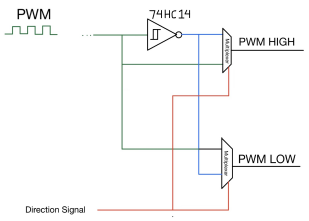

Fig. 6. Old Wire Diagram for Motor control

As it would've allowed us full bidirectional control with a minimal use of ports. However, after running into issues with assembly and controlling multiple wheels, we pivoted to 2 wheel drive and using only the Servo signals. We connected one servo signal to the PWM HIGH of the driver and the other to the PWM LOW. We would ground one of the servo pins depending on the direction of the rotation we desired and used the same duty cycle for both of the PWM's. This had the added benefit of making the control more robust as we had less wires and sources of noise. Conversely, this did mean we had to design a new way to hold our front wheels, but we solved this by CADing wheels compatible with ball bearings for skateboards with mounts.

VI. Motor Control Module on FPGA (Manuel)

The motor control module written on the FPGA is a basis for further implementation of a full heading and position controller. Currently, the module takes in a clk, rst, encoderL, encoderR, IMUHeading, distance, and enable signal. The encoderL and encoderR signals are the direct signals fed through from the encoders on the motors and are used along with local parameters to keep track of how many revolutions the wheel has spun. There is also a local parameter DUTY which holds the standard duty cycle we want the motors to run at. An inner module PWM creates the two PWM signals needed for each motor, along with a duty cycle value and a direction signal. If the direction signal is 0 the motors will spin forward, if the direction signal is 1 the motors will spin backwards. Direction is controlled through two PWMs since the driver board provides power to the motors using an H-bridge. Which signal of the two is sent to the driver board will determine the direction the motor spins. The IMUHeading and distance signals are one-bit signals used to drive transitions between the state the motor is currently in.

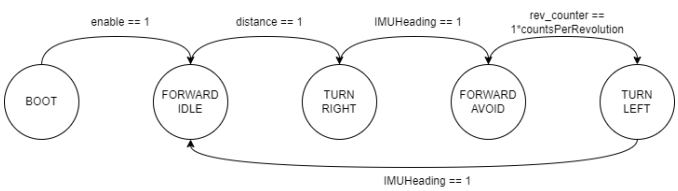

Fig. 7. Motor control state machine

First, motors are off until we get the enable signal signifying we have sensor data coming in where it enters a FORWARD IDLE state. If the sensors data returned a value too low in our top level module we set distance high which transitions our motor controller into a TURN RIGHT state. Motors are then commanded to turn the vehicle right until IMUHeading goes high. In the top module this is done by saving the current heading value before transitioning to the TURN RIGHT or TURN LEFT state and then calculating the difference between current and saved heading value until it is greater than 90 degrees. From TURN RIGHT it transitions to FORWARD AVOID where it moves forward an amount of revolutions specified by a local parameter before transitioning to a TURN LEFT state where it should go back to the original heading and loop back to FORWARD IDLE. This state machine allows the vehicle to detect and object and move around it. All the sensor signals are used and implemented and can be built off of to create a more complete position controller and lane navigation system.

VII. Future Work and Discussion

While the project was successful in achieving our MVP there is still further work to be done and some hurdles to overcome.

- The IMU communication with the FPGA is fully established. However, readings from the IMU directly onto the FPGA reveal stale values that don't update how they should. This seems to be an issue with the BNO055 protocol and most possibly a step that might be missing in communicating with it. The commands sent are the same sent from the Arduino using I2C communication yet the values aren't updating like it should. From the data sheet it doesn't seem obvious what the missing step is but there is a possibility that it is one of the following: (1) The Arduino does activate an external clock signals to power the clock on the BNO055 which could be used to improve readings. (2) The trigger pin on the BNO055 can be set up so that when it goes high it creates an interrupt where the BNO055 will make a reading and respond immediately with the value in a series of registers. This wasn't ever implemented on the Arduino to get values but could be implemented on the FPGA to ensure sensor readings are taking place. (3) There may be some sensor offsetting values handled by the Arduino library that aren't implemented on the FPGA.
- The motor controller can be expended on to provide greater position and heading control following the initial plan for our PID control system (Fig 7). It would also allow for trajectories to be planned and 2D mapping using the LIDAR to be done. We also ran into issues trying to run the wheels at slower pwms since the torque produced was to small to overcome static friction. Implementing the PID controller would fix this issue.

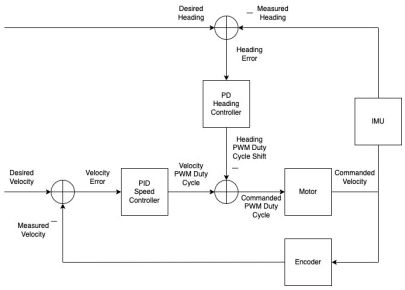

Fig. 8. Diagram for PID control

References

- https://cdn-shop.adafruit.com/datasheets/BST_BNO055_DS000_12.pd
- https://www.amazon.com/YDLIDAR-X4-Degree-Communication-Interface/dp/B07DBYHJVQ
- https://www.pololu.com/product/4752
- https://www.adafruit.com/product/2472
- https://www.sparkfun.com/products/15569

Acknowledgment

Thanks to Joseph Feld, Darrem Lim, Adrianna Wojtyna, Pleng Chomphoochan, and Ivy Liu for all the guidance and feedback.