Maseeh College of Engineering and Computer Science

PORTLAND STATE UNIVERSITY

Winter 2023

ECE544/ECE558 - Final Project Design Report

Emily Devlin, Noah Page, Drew Seidel, Stephen Weeks
Email addresses: emdevlin@pdx.edu, nopage@pdx.edu, dseidel@pdx.edu, stweeks@pdx.edu,

Maseeh College of Engineering and Computer Science

PORTLAND STATE UNIVERSITY

Introduction

The goal of this project was to design a fully functional IoT embedded system. The group chose to create a robotic RC car that is Android controlled. It contained three major components: an android application, Raspberry Pi, and an FPGA. The android application allows user input that is processed via MqTT on the Raspberry Pi that is then sent to the FPGA for driver control. Stretch goals were added to the project such as displaying camera output to the app, returning motor data to the app, and ultrasonic sensing to the RC car direct motor control.

Design Structure

Hardware

The hardware design was set up with two major goals in mind: run the motors at an expected and consistent speed and direction as well as handle any necessary communication on the system state (driving and emergency stops). We designed the hardware such that the system was as close to the metal as possible and as simple as possible. We wanted it to be responsive and to push any processing off to the brains of the system: the Raspberry Pi.

The hardware consisted of a FIT timer, two UARTLite systems, the custom UARTLite for the MDM when debug mode is turned on, two HB3 drivers, and a Nexys4IO module to give user output on the 7-segment display and LEDs. We also included a watchdog timer (WDT) which is currently not hooked up but can be if users are worried about undefined faults (which should lead to system reset).

The UARTs are used for bidirectional communication between the Raspberry Pi, the ultrasonic sensor, and the debugger mdm interface. The latter is only in instances where the DEBUG flag is set to true. The other two allow for communication from the android app on direction to be sent and to return motor updates or to read ultrasonic pulses by sending a 0x55 on the TX line and returning the distance in mm’s.

The two HB3s are set up so they can drive the speed and direction of the motors. Combined with a firmware PID controller they can be sent correct PWM pulses to maintain the setpoint which is currently burnt into the firmware. They also allow us to read back the speed from the motors and through testing we were able to determine direction which is handled in firmware through lookup tables.

The final hardware component of the system is the is the Nexys4IO module. This is used to interact with the physical interfaces, LEDs and 7-segment displays, to give tangible information to the user. The display is used to display the heartbeat and Pi UART RX interrupt as well as displaying the motor speeds. The LEDs all turn on when the motors are being driven, giving clear visual information on system state for debugging purposes (and because LEDs are pretty).

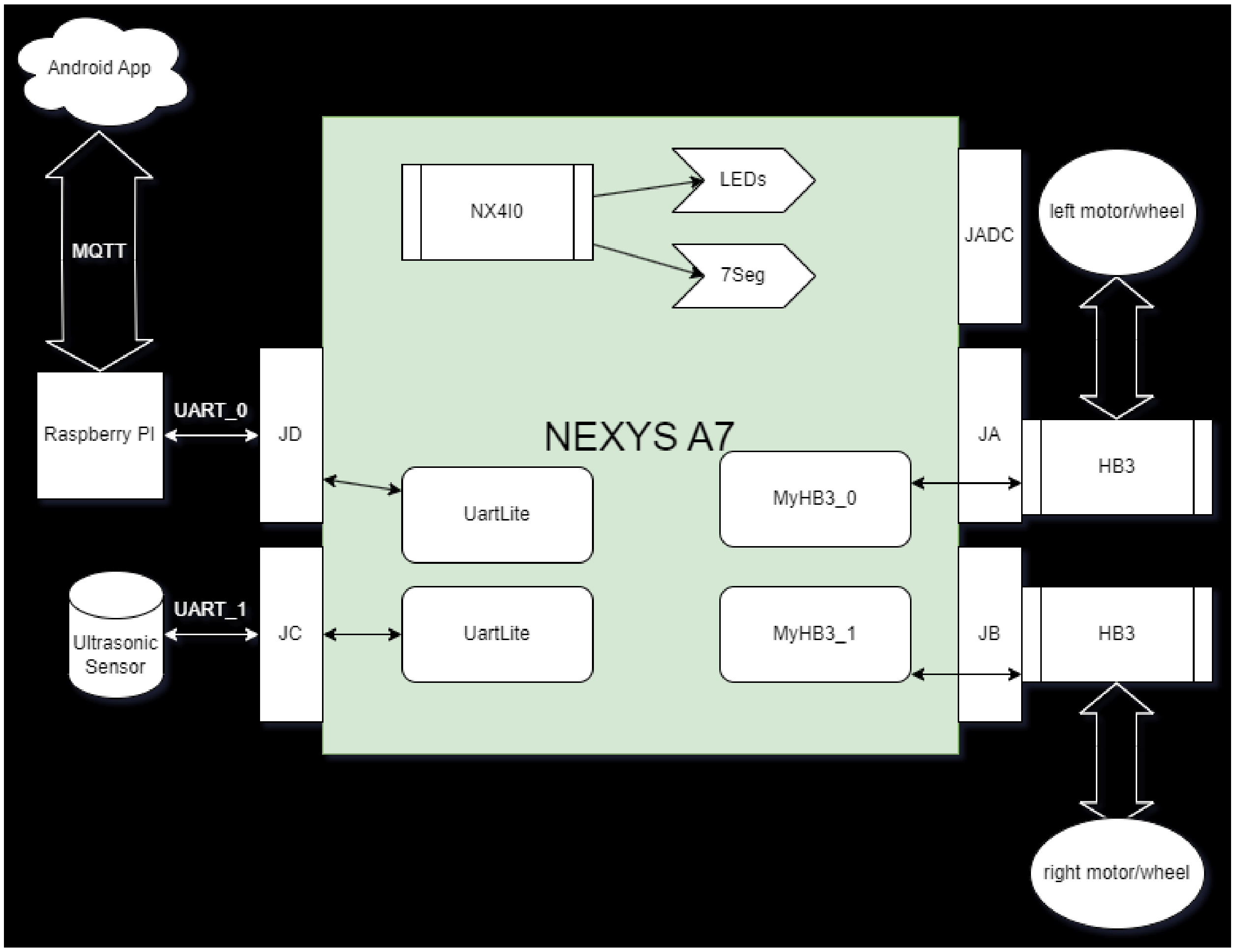


Figure 1: Overview of System

Software

Like the hardware design, the firmware design tried to follow a setup of simplicity and correct design to allow for ease of use, debug, and small code execution time (we want our system to be as responsive as possible when driving the motors). To maintain easy readability, extendability, and useability while being lightweight we opted for a hand rolled FSM (Finite State Machine) scheduler.

The FSM scheduler runs in the main while(1) loop and is a registered set of function pointers that are setup and initialization. Each task is called continuously based on the current state (I.E. run_state will be called while run is being executed). This allows for an extremely quick response time. Instead of polling through the system and executing all four tasks in series, we only execute what is necessary at that time. This also allows for easy extension because the system is interrupt (or event) driven as shown in the flow diagram below.

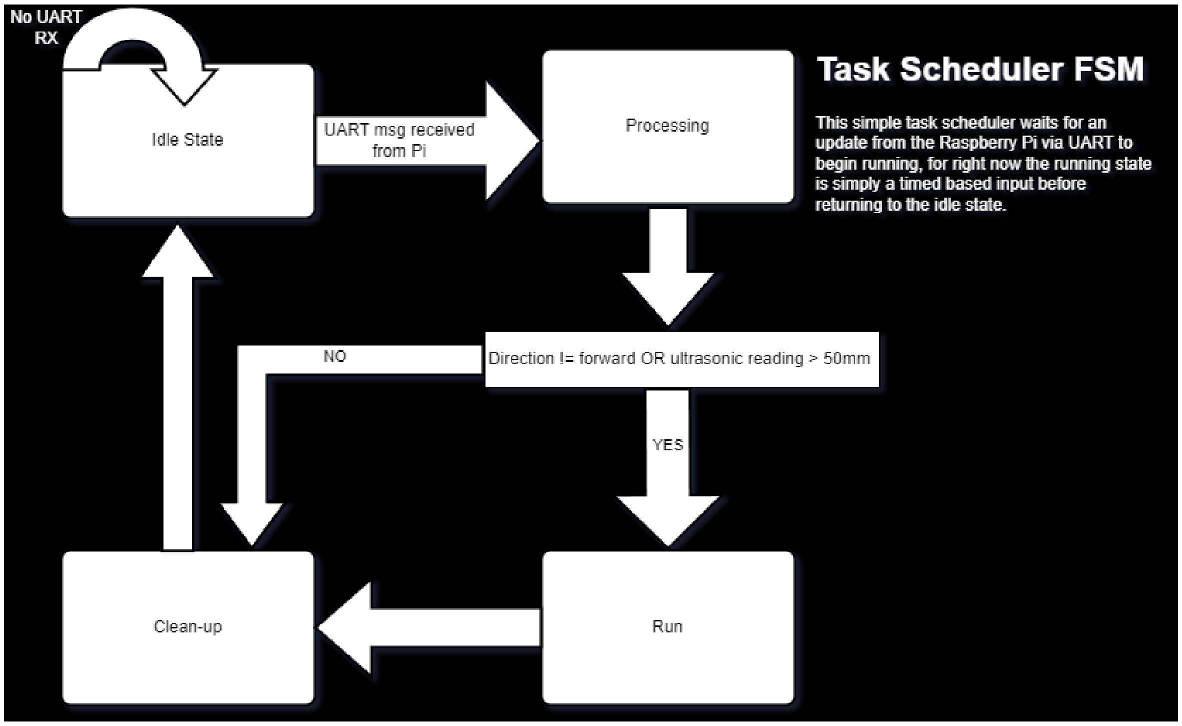

No UART RX
Idle State
UART msg received from Pi
Processing
Task Scheduler FSM
This simple task scheduler waits for an update from the Raspberry Pi via UART to begin running, for right now the running state is simply a timed based input before returning to the idle state.
NO
Direction != forward OR ultrasonic reading > 50mm
YES
Clean-up
Run

Figure 2: FPGA FSM Design

Because the hardware/firmware system needs to be as responsive as possible, it is important to keep states relatively easy to manage. The idle state, for example, merely sits and waits for a direction and a “go” message from the main system controller (the Raspberry Pi). It is interrupt triggered and sets a boolean flag to be passed to the task that a message was received. Once the event triggers, the rest of the system runs in a deterministic flow until it returns to the idle state (with the exception of the ultrasonic sensor when moving forward). This allows for easy debugging and clear boundaries of responsibilities of the system.

The most valuable state being run by the firmware is the run state. Below you can find a flow chart of the system when in the run state logic. The run operation only has one major decision responsibility to make right now, how long should it be run. Future changes would have the Raspberry Pi sending a start bit and the run state runs until a stop bit is transmitted. This loop is where criticality is of most importance. We use the FIT to count how long the state is running (since it runs at a relatively reliable 2Hz) for the motors to run. We also check if there is any obstacle 50 mm or less in front of us if running forward, if so, we will not run. Within the run state (the while loop), is where we have no blocking calls and merely run the PID control loop and push displays to the 7-segment display (the LEDs are turned on and off before/after the while run loop). If the Raspberry Pi TX UART port can send data, we quickly transmit speed and direction with a '\n' character to indicate we are done transmitting. We do not wait for the TX buffer to be clear, as this would block the motor PID controls which are vital to system operability.

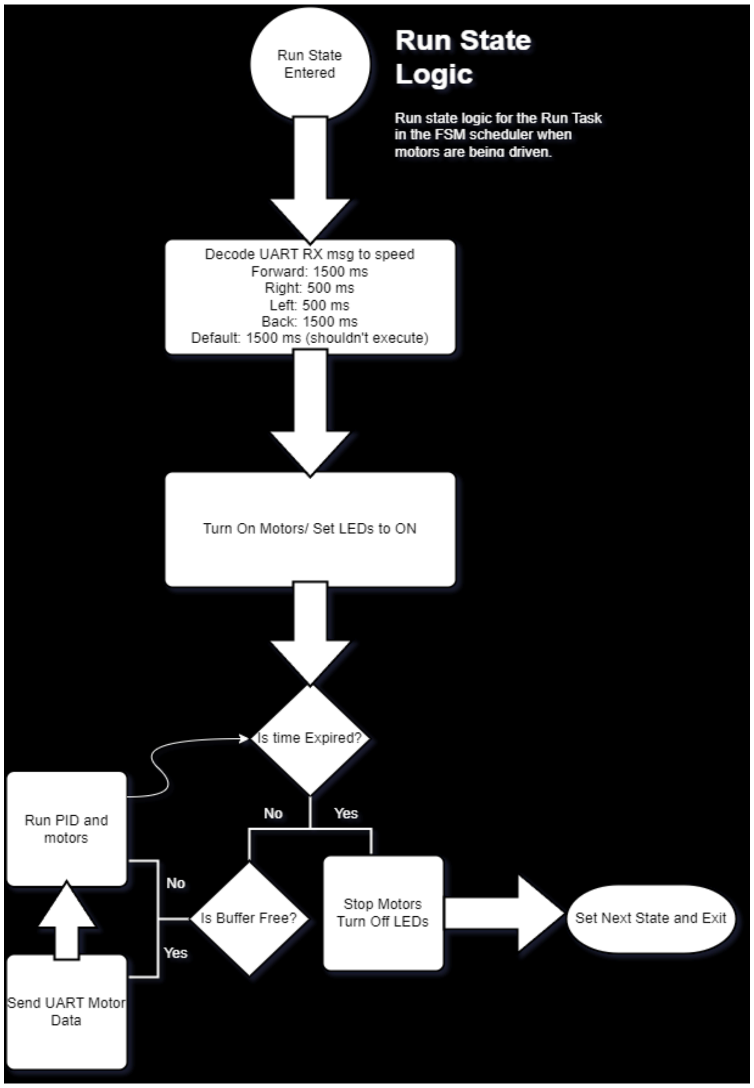
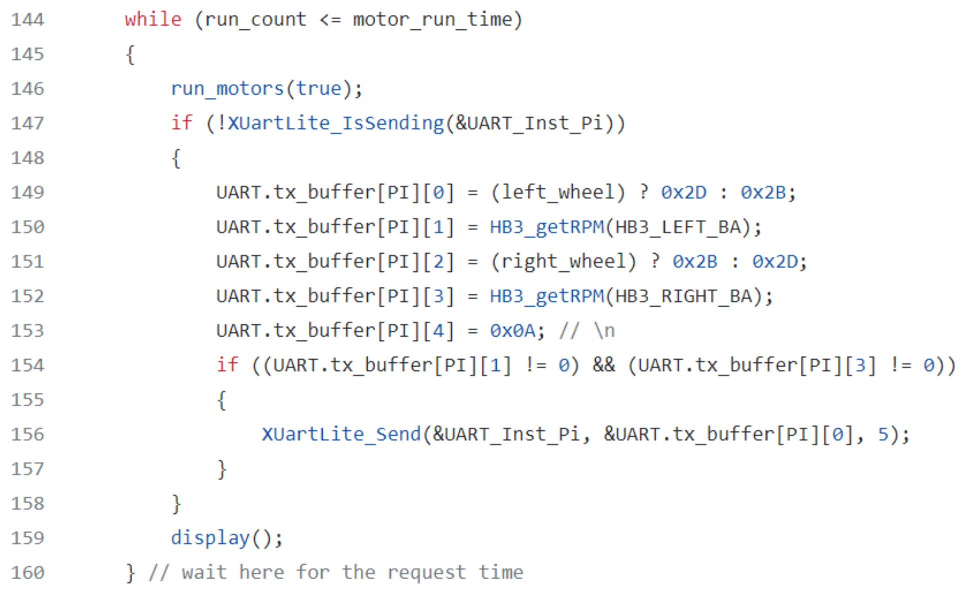

Run State Entered
Run State Logic
Run state logic for the Run Task in the FSM scheduler when motors are being driven.
Decode UART RX msg to speed
Forward: 1500 ms
Right: 500 ms
Left: 500 ms
Back: 1500 ms
Default: 1500 ms (shouldn't execute)
Turn On Motors/ Set LEDs to ON
Is time Expired?
No
Yes
Run PID and motors
No
Is Buffer Free?
Yes
Send UART Motor Data
Stop Motors Turn Off LEDs
Set Next State and Exit

Figure 3: Run State Flow Diagram

The other two tasks of the scheduler, processing and clean-up, are mostly preamble setup and clean-up of the running system loop to provide quick responsiveness and setup the states as needed (such as directions and message type). Clean-up also re-enables the Raspberry Pi RX interrupts as we shut off this messaging port when running to avoid system interruptions and increase responsiveness. For version 2.0, these interrupts would need to be removed as we will need to listen for a stop bit when in the run state.



```c
    while (run_count <= motor_run_time)
    {
        run_motors(true);
        if (!XUartLite_IsSending(&UART_Inst_Pi))
        {
            UART.tx_buffer[PI][0] = (left_wheel) ? 0x2D : 0x2B;
            UART.tx_buffer[PI][1] = HB3_getRPM(HB3_LEFT_BA);
            UART.tx_buffer[PI][2] = (right_wheel) ? 0x2B : 0x2D;
            UART.tx_buffer[PI][3] = HB3_getRPM(HB3_RIGHT_BA);
            UART.tx_buffer[PI][4] = 0x0A; // \n
            if ((UART.tx_buffer[PI][1] != 0) && (UART.tx_buffer[PI][3] != 0))
            {
                XUartLite_Send(&UART_Inst_Pi, &UART.tx_buffer[PI][0], 5);
            }
        }
        display();
    } // wait here for the request time
```

Figure 4: Uart Buffer Sent to Raspberry Pi

The firmware also consists of helper functions that are separated into their own files. These files are broken into: cntrl_logic, uart, task, sys_init, fit, and debug. These allow for easy modularity and allow for minimal changes in other associated files while containing most changes within the module itself. It also helps draw clear boundaries between system operations and communication. It was very important to have clear separation of concerns and make extension as easy as possible with 4 different people working in the code base. You can see a clear difference between operations.

Android Application

The goal of the Android application is to provide a user interface for interaction with the robot car. The user should be able to control the car with the application, and ideally also receive and view information about the car's motors. We accomplished both of these goals.

The Android application has one activity which hosts three fragments and a data model. The fragments control UI-related activities and actions, while the data model controls the app's variables and data which is unrelated to the UI.

The Welcome Fragment presents the user with fields to enter their MQTT Network information and to connect. The Main Controls Fragment visually displays information about the two motors. The Main Controls Fragment also contains four arrow buttons, which the user can press to move the car in a specific direction. The Main Controls Fragment contains a button to navigate to the Video Feed Fragment. The Video Feed Fragment contains the same four arrow buttons, as well as a place to stream video directly from the robot's Raspberry Pi Camera module.

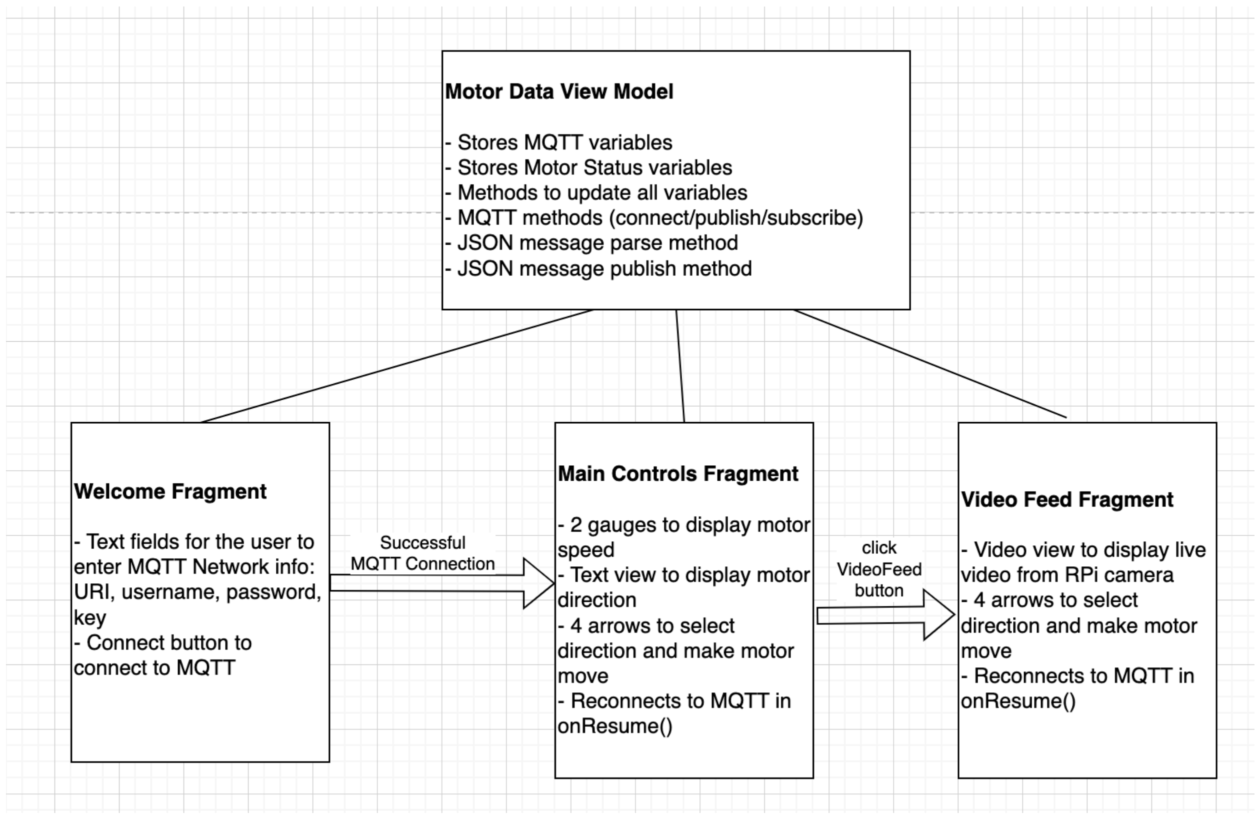

Motor Data View Model
- Stores MQTT variables
- Stores Motor Status variables
- Methods to update all variables
- MQTT methods (connect/publish/subscribe)
- JSON message parse method
- JSON message publish method
Welcome Fragment
- Text fields for the user to enter MQTT Network info: URI, username, password, key
- Connect button to connect to MQTT
Successful MQTT Connection
Main Controls Fragment
- 2 gauges to display motor speed
- Text view to display motor direction
- 4 arrows to select direction and make motor move
- Reconnects to MQTT in onResume()
click VideoFeed button
Video Feed Fragment
- Video view to display live video from RPi camera
- 4 arrows to select direction and make motor move
- Reconnects to MQTT in onResume()

Figure 5: Android App Overview

We used Navigation and a Nav Graph to control the interaction between these fragments. The Welcome Fragment is our home fragment, where the app opens. There is an action to move from this fragment to the Main Controls Fragment, and another action to move from the Main Controls Fragment to the Video Feed Fragment. Our app has a top toolbar with a Back button, so this is how the user can navigate back to a previous fragment. A user of our app might wish to alternate between the Main Controls Fragment and the Video Feed Fragment.

The Motor Data View Model extends the ViewModel class. It contains all the variables and methods that are not UI-related. These variables are of type Live Data, and are either related to MQTT or related to the motor info. For example, we store the user-entered MQTT URI, Username, Password, etc. in the data model, as well as the MQTT Connect, Subscribe, Publish functions. This way, if the MQTT connection is lost, any of the fragments can use the data model to handle reconnecting to MQTT, so the user doesn't have to restart the app or navigate back to the Welcome screen.

The motor-related variables stored in the Motor Data View Model are the speed of each motor and commands to move the motor in each direction (set by the arrow buttons). If a user presses one of the arrow buttons, a movement command is encoded into a JSON message and sent via MQTT. When the app receives an MQTT Motor Status message, this JSON message is parsed and the results stored in the motor speed variables. All of this is handled by the Motor Data View Model.

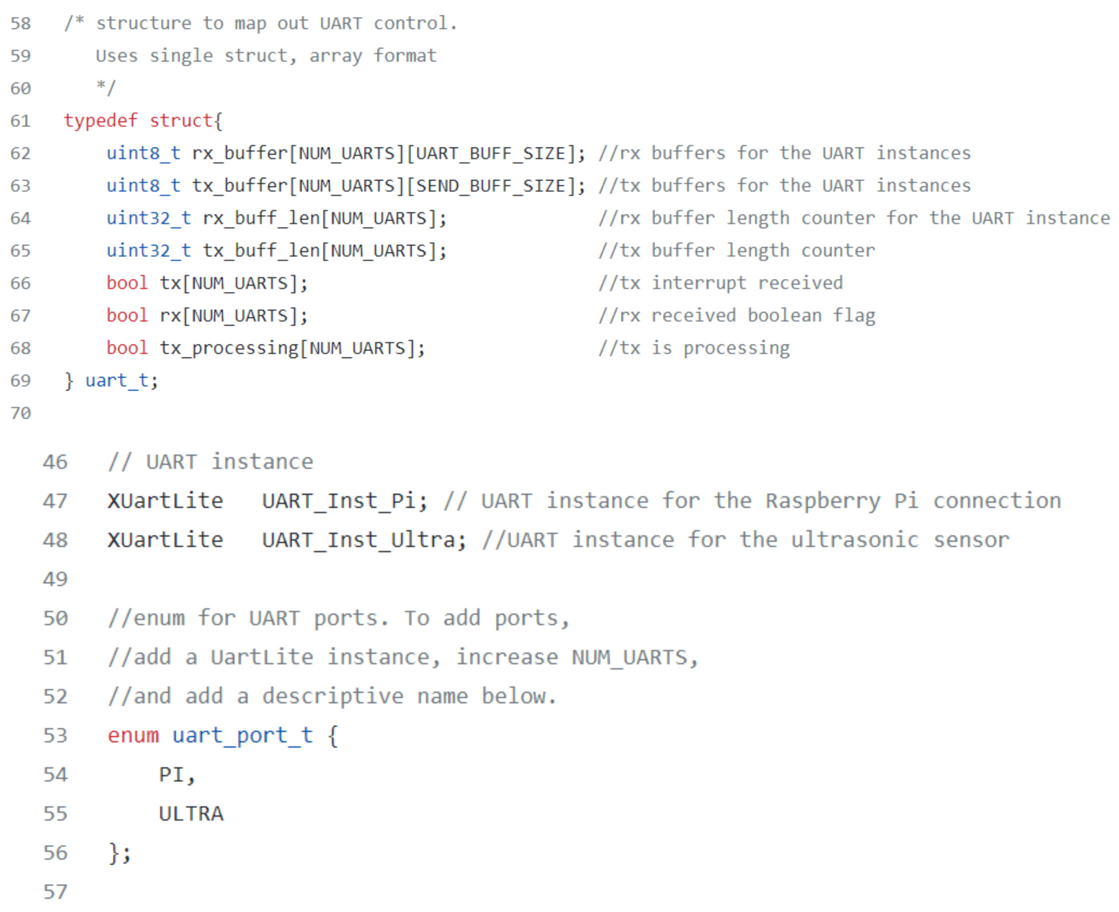
Each Fragment interacts with the Motor Data View Model through data binding and observers. If a Fragment needs to change the value of a Motor Data View Model variable, it calls a custom function to do so. If a Fragment needs to access the value of a Motor Data View Model variable, it must call the observe() function to watch for changes to this variable. The XML fragment layout files can also access the Motor Data View Model directly, for example, to set a text field.

Communication

There are five major communication components to the system:

- FPGA to Raspberry Pi (UART). FPGA receives direction command, FPGA sends the two motor RPM and direction.
    - Raspberry Pi Tx to FPGA Rx sends singular byte upon Android App button press:
        - 0x00 is forward for 1500ms
        - 0x01 is rotate right for 500ms
        - 0x02 is rotate left for 500ms
        - 0x03 is reverse for 1500ms
    - FPGA Tx to Raspberry Pi Rx sends a 5 byte buffer that is composed of:
        - {left motor direction sign, left wheel RPM, right motor direction sign, right wheel rpm, newline = 0x0A}. The newline character is checked to make sure the package was received successfully.
- FPGA to Y401 ultrasonic sensor (UART). FPGA sends control bytes for millimeter measurement reception. Y401 ultrasonic sensor sends back a two byte millimeter number.
    - FPGA Tx to Y401 Tx (by device labeling convention) sends singular byte 0x55 used to receive millimeters back from device
    - Y401 Rx to FPGA Rx sends back two byte wide millimeters measurement
- Raspberry Pi publisher to Android App Subscriber (MQTT Topic RobotCar/Motors)
    - Raspberry Pi publishes a singular JSON object to the topic denoting the rotational state and RPM of each message. An example of the formatting showing the robot rotating is the following: {“Left_Motor”, “-43”, “Right_Motor”, “+43”}
- Android App publisher to Raspberry Pi Subscriber (MQTT Topic RobotCar/Move)
    - Android publishes upon a button press to the topic denoting which direction the motors should move. This message is translated by the Raspberry Pi firmware and formatted for the described Raspberry Pi to FPGA UART protocol. A sample JSON message is: {“UP”:1, “DOWN”:0, “RIGHT”:0, “LEFT”:0}

Both UART connections utilize a separate Xilinx UartLite instance that uses interrupt handling for receiving.



```c
/* structure to map out UART control.
   Uses single struct, array format
   */
typedef struct{
    uint8_t rx_buffer[NUM_UARTS][UART_BUFF_SIZE]; //rx buffers for the UART instances
    uint8_t tx_buffer[NUM_UARTS][SEND_BUFF_SIZE]; //tx buffers for the UART instances
    uint32_t rx_buff_len[NUM_UARTS];              //rx buffer length counter for the UART instance
    uint32_t tx_buff_len[NUM_UARTS];              //tx buffer length counter
    bool tx[NUM_UARTS];                           //tx interrupt received
    bool rx[NUM_UARTS];                           //rx received boolean flag
    bool tx_processing[NUM_UARTS];                //tx is processing
} uart_t;

```

```c
// UART instance
XUartLite   UART_Inst_Pi; // UART instance for the Raspberry Pi connection
XUartLite   UART_Inst_Ultra; //UART instance for the ultrasonic sensor

//enum for UART ports. To add ports,
//add a UartLite instance, increase NUM_UARTS,
//and add a descriptive name below.
enum uart_port_t {
    PI,
    ULTRA
};

```

Figure 6: Extendibility of Uart Instances Using Struct with Array Methodology

Challenges

One challenge we faced with the Android app was figuring out the Live Data View Model and its interaction with the rest of the application. None of us successfully used this in Project 2, but we knew it would be important to implement in this project since we needed several Fragments. We followed an Android Codelab tutorial (the Cupcake app) to get started using a Live Data View Model. However, this Codelab only had the XML Layout files access the Live Data, rather than the Fragment accessing the Live Data directly.

We found out that we still needed a Live Data observer for a Fragment to access a Live Data variable and have any change to the variable also happen in the Fragment. This was confusing since Observer has been deprecated. However, the observe() function is not deprecated and is now what Android recommends using. Because different tutorials and Android documentation are from different eras, it was hard to find all of this information in one place.

The biggest challenges faced in the hardware were in integration and testing. We did not originally have the HB3 module setup for sending direction data or the ability to instantiate more than one IP block. That had to be configured and updated to allow for hardware testing and validation (a simple button script that was maintained for debug in the cntrl_logic.c file). We also had major issues with UART testing. Because we charge the two boards with their own separate battery packs, ground noise is very important. We found that there was a lot of noise when sending from the FPGA to the Raspberry Pi. Current working theory is we had a bad ground plane (wire) which was causing large noise spikes and junk to be sent to the controller hub, making it difficult to impossible to send reliable motor data to the application.

For firmware, the biggest problem child also came in the form of the UART. It turns out Xilinx documentation on how to use the UART IRQ is completely incorrect. The HAL libraries seem to barely perform as intended. We ended up having to search through the interwebs to Digilent message boards to find a good framework using the low-level libraries to get the RX IRQ working (we never managed to get TX to work properly). This caused huge pains as the UART is the major critical path between the app to the FPGA. We also had to come up with a messaging schema with the UART. We chose a simple schema, but for future improvement work timing and schema analysis would be needed, which was not an easy area to work through.

Overall, one of the biggest challenges was testing and integration. The project is set up so that the MQTT broker is on the Raspberry Pi and so its static IP is hard coded into the Android App along with the MJPEG video stream from the camera. So the project only works when all parts of it are on the same Wi-Fi network. This made it difficult for team members to test various modules on their own. Furthering that problem was that there was only one robot. So as updates were made, bugs were only discovered after being implemented.

Conclusion

This project brought together concepts we learned this quarter in both ECE 558 and ECE 544, as well as our skills and interests from hobbies and other classes. We had to work together closely in order to make sure that each piece of the project would integrate well with everything else. We also had to work incrementally and carefully, with frequent testing to make sure that changing something didn’t break something else.

As a result, our communication skills and teamwork were strengthened. We frequently solved problems together, or one person took over debugging a problem and was able to look at it from a different perspective and find a solution.

There is a lot of room for future expansion of this project, such as adding buttons to control speed (this was a stretch goal which we did not implement), or adding a way to make the robot keep moving. We could also, of course, add more sensors and collect and display more information about the motors, such as voltage, current, etc.

Work Division

This project required us to work together very closely and carefully, so almost every piece of it was a team effort. However, the main division of work is as follows:

- Hardware: Stephen, Drew, Noah
- FPGA Software: Drew, Stephen, Noah
- Raspberry: Noah
- Android App:
    - LiveData / ViewModel: Emily, Drew
    - Video Feed: Noah
    - Gauges: Emily
    - Debugging: Emily, Drew, Noah
- Documentation: Everyone
- Testing: Noah
- Designing and building robot: Noah