Arigato the Crypto Roboto

A Robot Controlled with MQTT Messages

Generated by Updates to a Crypto-Currency Wallet

Boston University

EC 544 Networking the Physical World

Spring 2022

Final Project Report

Ian Chadwick, Sara Fagin, Santiago Gomez

https://github.com/sfagin89/Arigato_Crypto_Roboto

Introduction

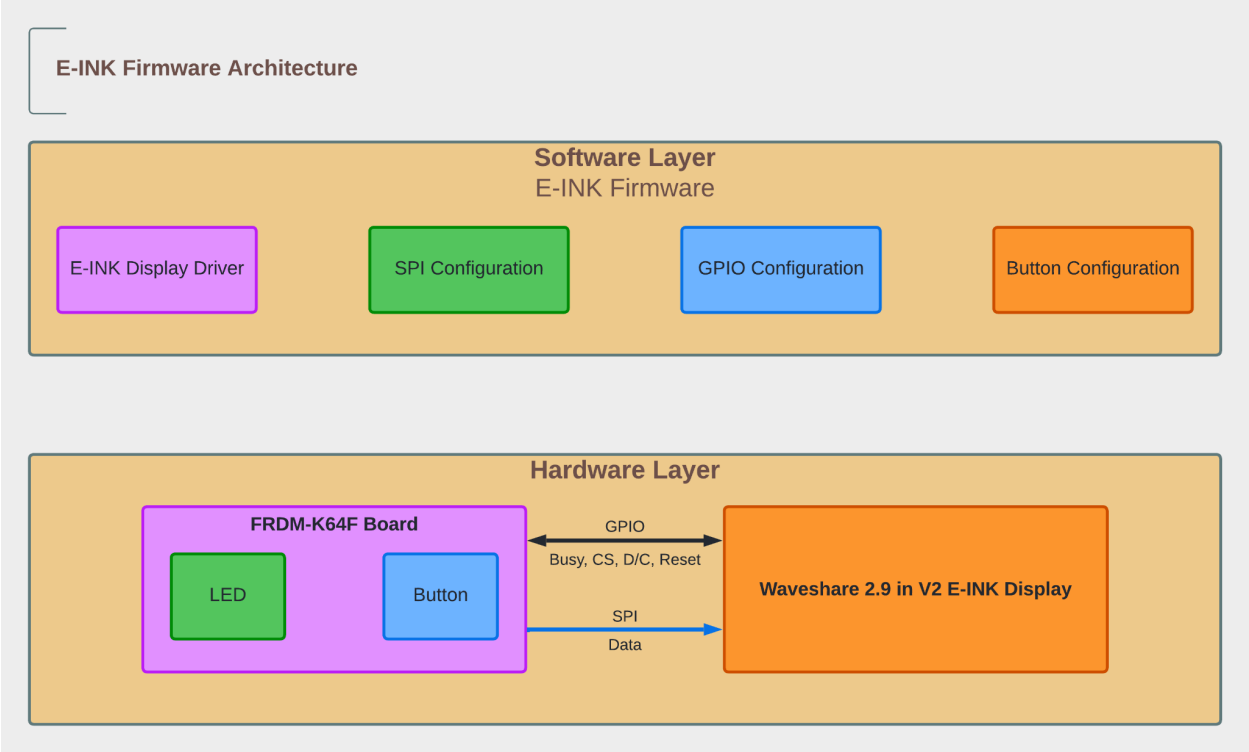
Crypto-currency's ever growing adoption by the general population presents an opportunity to connect embedded cyber-physical systems with financial transactions that do not include banks. For example, imagine a near future where an autonomous taxi pulls up to the curb to pick you up. You get in the driverless vehicle and initiate the journey by scanning a QR code that is connected to the taxi's crypto-wallet. You send some crypto-currency and complete the financial transaction without a pesky bank or credit card. Our project, "Arigato, the Crypto Roboto," is a small step toward that future. This project deploys a Raspberry Pi robotic car, whose movements are controlled by payments to a crypto-wallet. The robot is subscribed to a movement command topic on an AWS MQTT server. Messages are published to that topic when the robot's crypto-wallet receives currency, Dogecoin in this case. A user scans a QR code that appears on an e-ink display sitting on top of the robot. The display is controlled by a FRDM-64F board and it interfaces with the board via a custom PCB.

This project employs various technologies, including low-level firmware, custom PCB design, IoT communication, APIs, and crypto-currency. Ian Chadwick connected the robot with MQTT, Sara Fagin created the custom PCB for the e-ink display, and Santiago Gomez developed the e-ink firmware for the FRDM-K64F board. He is also responsible for the client application that links the robot's crypto-wallet with its MQTT Message Broker. This report delves into the implementation of each component. Furthermore, each component discussion provides insight into the development of the respective engineer.

Discussion of E-INK Firmware

The e-ink firmware integrates Waveshare's 2.9 inch V2 e-ink display with NXP's FRDM-K64F development board. This portion of the project is defined by a software layer and a hardware layer. The software layer consists of code for the e-ink driver, SPI configuration, GPIO configuration, and button configuration. The hardware layer implements an LED and a button on the FRDM-K64F board, as well as the Waveshare 2.9 inch V2 e-ink display. *Figure 1* highlights the various functional blocks of these two layers.

Figure 1: Functional Block Diagram of the E-INK Firmware



The Software Layer

E-INK Display Driver

The base code for the e-ink driver derives from open-source code provided by Waveshare. However, Waveshare's open-source code[1] is designed for an STM32 board. Therefore, modifications were made so the code utilizes the FRDM-K64F's SPI communication and delay mechanisms. Delays are implemented by configuring the FRDM-K6F's SysTick counter to generate an interrupt after a given number of milliseconds. *Figure 2* shows the configuration code.

Figure 2: Delay Configuration Code

```
1072 void Delay_Config(void)
1073 {
1074     if (SysTick_Config(SystemCoreClock / 1000U))
1075     {
1076         while (1)
1077         {
1078         }
1079     }
1080 }
```

When the SysTick interrupt fires, it calls on a SysTick_Handler function embedded within the "eink.c" file. Figure 3 is the handler code. An interesting note about this handler is it overwrites another SysTick handler that came packaged with the project SDK built by MCUXpresso. The default SysTick handler is in the FreeRTOS library included with the project. It is necessary to comment out the default handler so that the interrupt calls the custom handler instead.

[1] Waveshare e-Paper Code: https://github.com/waveshare/e-Paper/tree/master/STM32

Figure 3: SysTick Handler Code

```
1064 void SysTick_Handler(void)
1065 {
1066     if (g_systickCounter_eink != 0U)
1067     {
1068         g_systickCounter_eink--;
1069     }
1070 }
```

Besides inserting a custom delay mechanism, the firmware also overwrites Waveshare's send_data() and send_command() functions. According to Waveshare's programming model outlined in their documentation[2], these functions first require setting the Data/Command (D/C) pin high or low to indicate what type of information is transmitting over SPI. D/C high indicates command and D/C low indicates data. Next, the Chip Select pin must be set low. At this point the controller can start transmitting over SPI to the display. Once SPI transmission is complete, the controller must set the Chip Select pin high. *Figure 4* demonstrates this sequence for the send_data() function.

Figure 4: Send Data Function

```
1287 static void EINK_SendData(UBYTE byte)
1288 {
1289     GPIO_PinWrite(EINK_DC_GPIO, EINK_DC_PIN, kGPIO_Hi); // DC high tells display that data coming over SPI is data
1290     GPIO_PinWrite(EINK_CS_GPIO, EINK_CS_PIN, kGPIO_Low);
1291
1292     // spi transfer goes, the controller is the K64 FRDM board
1293     /* Start controller transfer, send data to screen */
1294     dspi_transfer_t controllerXfer;
1295     controllerXfer.txData      = &byte; // transferring the byte passed in
1296     controllerXfer.rxData      = NULL;
1297     controllerXfer.dataSize    = 1U; // only transferring one byte
1298     controllerXfer.configFlags = kDSPI_MasterCtar0 | DSPI_CONTOLLER_PCS_FOR_TRANSFER | kDSPI_MasterPcsContinuous;
1299     DSPI_MasterTransferBlocking(DSPI_CONTOLLER_BASEADDR, &controllerXfer);
1300
1301     GPIO_PinWrite(EINK_CS_GPIO, EINK_CS_PIN, kGPIO_Hi);
1302 }
```

The display() function also required a minor edit; a delay is inserted at the beginning of the function to provide a time buffer for the display from previous executions. Testing revealed a timing bug between the controller and the display. The delay is a straightforward fix to this issue.

[2] Waveshare 2.9 in V2 Specification:
https://www.waveshare.com/w/upload/e/e6/2.9inch_e-Paper_Datasheet.pdf

There are no other critical operations performed by the controller, therefore a delay is appropriate. *Figure 5* shows the delay in the first line of the display() function. The rest of the e-ink driver code from Waveshare remains unaltered.

Figure 5: Display Function

```
1199 void EINK_Display(UBYTE *Image, UWORD Image_Size)
1200 {
1201     delay_ms(3000U); // delay needed to provide time buffer for the data transfer to succeed
1202     UWORD i;
1203     EINK_SendCommand(0x24);   //write RAM for black(0)/white (1)
1204     for(i=0;i<Image_Size;i++)
1205     {
1206         EINK_SendData(Image[i]);
1207     }
1208     EINK_TurnOnDisplay();
1209     PRINTF("Done Printing --> Display Turned On\r\n");
1210 }
```

SPI Configuration

The template for configuring the Serial Peripheral Interface (SPI) comes from MCUXpresso's SDK example for the SPI module. The example is called "frdmk64f_dspi_polling_b2b_transfer_master." This template provided two components for configuring the SPI module and a third component for initiating the SPI transfer. SPI configuration first necessitates programming pins on the K64F chip to act as SPI pins. Figure 6 demonstrates how to set pins for SPI operations.

Figure 6: Setting Pins for SPI

```
1484     /* PORTD0 (pin 93) is configured as SPI0_PCS0 */
1485     PORT_SetPinMux(PORTD, 0U, kPORT_MuxAlt2);
1486
1487     /* PORTD1 (pin 94) is configured as SPI0_SCK */
1488     PORT_SetPinMux(PORTD, 1U, kPORT_MuxAlt2);
1489
1490     /* PORTD2 (pin 95) is configured as SPI0_SOUT */
1491     PORT_SetPinMux(PORTD, 2U, kPORT_MuxAlt2);
1492
1493     /* PORTD3 (pin 96) is configured as SPI0_SIN */
1494     PORT_SetPinMux(PORTD, 3U, kPORT_MuxAlt2);
```

Once the pins are programmed for SPI, the next step is to initialize the SPI module. This includes several steps, but some important procedures are selecting the clock source, setting the baud rate, and indicating how many bits are in a data frame. *Figure 7* shows the SPI initialization sequence.

Figure 7: SPI Initialization Sequence

```
1511 void EINK_InitSPI(void){
1512
1513     uint32_t srcClock_Hz;
1514     dspi_master_config_t controllerConfig;
1515
1516     /* Controller config */
1517     controllerConfig.whichCtar                                = kDSPI_Ctar0;
1518     controllerConfig.ctarConfig.baudRate                      = TRANSFER_BAUDRATE;
1519     controllerConfig.ctarConfig.bitsPerFrame                  = 8U;
1520     controllerConfig.ctarConfig.cpol                          = kDSPI_ClockPolarityActiveHigh;
1521     controllerConfig.ctarConfig.cpha                          = kDSPI_ClockPhaseFirstEdge;
1522     controllerConfig.ctarConfig.direction                     = kDSPI_MsbFirst;
1523     controllerConfig.ctarConfig.pcsToSckDelayInNanoSec        = 1000000000U / TRANSFER_BAUDRATE;
1524     controllerConfig.ctarConfig.lastSckToPcsDelayInNanoSec    = 1000000000U / TRANSFER_BAUDRATE;
1525     controllerConfig.ctarConfig.betweenTransferDelayInNanoSec = 1000000000U / TRANSFER_BAUDRATE;
1526
1527     controllerConfig.whichPcs           = DSPI_CONTOLLER_PCS_FOR_INIT;
1528     controllerConfig.pcsActiveHighOrLow = kDSPI_PcsActiveLow;
1529
1530     controllerConfig.enableContinuousSCK        = false;
1531     controllerConfig.enableRxFifoOverWrite      = false;
1532     controllerConfig.enableModifiedTimingFormat = false;
1533     controllerConfig.samplePoint                = kDSPI_SckToSin0Clock;
1534
1535     srcClock_Hz = DSPI_CONTOLLER_CLK_FREQ;
1536     DSPI_MasterInit(DSPI_CONTOLLER_BASEADDR, &controllerConfig, srcClock_Hz);
1537 }
```

To start a SPI transmission, a dspi_transfer_t struct must be created. It will carry a pointer to the transmission data, a pointer to the receiving data buffer, the data size, and certain transmission flags. The pointer to this struct and the address of the SPI unit within the K64F is passed to a

function called DSPI_MasterTransferBlocking(). This function is provided by the MCUXpresso SDK. Figure 8 illustrates the creation of the dspi_transfer_t struct and subsequent calling of DSPI_MasterTransferBlocking().

Figure 8: SPI Transfer Sequence

```
1270    // spi transfer goes, the controller is the K64 FRDM board
1271    /* Start controller transfer, send data to screen */
1272    dspi_transfer_t controllerXfer;
1273    controllerXfer.txData      = &byte; // transferring the byte passed in
1274    controllerXfer.rxData      = NULL;
1275    controllerXfer.dataSize    = 1U; // only transferring one byte
1276    controllerXfer.configFlags = kDSPI_MasterCtar0 | DSPI_CONTOLLER_PCS_FOR_TRANSFER | kDSPI_MasterPcsContinuous;
1277    DSPI_MasterTransferBlocking(DSPI_CONTOLLER_BASEADDR, &controllerXfer);
```

GPIO Configuration

The Waveshare e-ink display requires additional logic pins beyond the SPI pins to function properly. These extra pins are Busy, Reset, Data/Command (D/C), and Chip Select (CS). All of these pins are configured as GPIO on the FRDM-K64F board. The display uses the Busy pin to tell the controller it is occupied and should not send additional data over SPI. In the K64F, the Busy pin direction is input. Reset, D/C, and CS pins are all set with output direction in the K64F. These pins are meant to reset the display, indicate whether the SPI transmission is data or a command, and to select the chip, respectively. One more pin on the K64F is configured as GPIO, but this pin is set to activate an LED on the development board. The LED turns blue to inform the user that the board is executing the e-ink firmware.

Programming a pin as GPIO includes providing a clock to the port on which the pin is located, setting that pin's function as GPIO, setting the direction as either input or output, and indicating the default output level for the pin. *Figure 9* outlines the code for programming the GPIO pin for the LED.

Figure 9: Programming a GPIO Pin

```
1418    /* Port E Clock Gate Control: Clock enabled */
1419    CLOCK_EnableClock(kCLOCK_PortE);
1420
1421    gpio_pin_config_t EINK_LED_BLUE_config = {
1422        .pinDirection = kGPIO_DigitalOutput,
1423        .outputLogic = 0U
1424    };
1425    /* Initialize GPIO functionality on pin PTB22 (pin 68)  */
1426    GPIO_PinInit(EINK_LED_BLUE_GPIO, EINK_LED_BLUE_PIN, &EINK_LED_BLUE_config);
```

Line 1426 of the code calls the GPIO_PinInit() function, which is provided by the MCUXpresso SDK. This is a wrapper function that does the necessary procedures for setting the registers that control a GPIO pin.

Button Configuration

The last element of the e-ink firmware is the code configuring the button that allows a user to switch between different images and to clear the screen. Button configuration has four main procedures. The first step is to provide a clock signal to the port that houses the pin. Next, a timer must be set up to handle the button interrupt and to help determine the state of the button. The button can support four states, rapid single press, normal single press, rapid double press, and long press. The third step is to set certain characteristics for the port, such as enabling the internal pullup resistor, and to indicate that the pin will be used as GPIO. The last step is to provide the button driver the callback function for when a button is pressed. *Figure 10* breaks down these four procedures.

Figure 10: Button Configuration Procedures

```
117  void SW3_configure_button(void)
118  {
119      /* Port A Clock Gate Control: Clock enabled */
120      CLOCK_EnableClock(kCLOCK_PortA);
121
122      /* Timer configuration needed for interrupts and determining the state of the button */
123      timer_config_t timerConfig;
124
125      timerConfig.instance       = 0U;
126      timerConfig.srcClock_Hz    = TIMER_SOURCE_CLOCK;
127      timerConfig.clockSrcSelect = 0U;
128      TM_Init(&timerConfig);
129
130      /* Set up port pin for the button SW3*/
131      const port_pin_config_t porta4_pin38_config = {
132              kPORT_PullUp,                                /* Internal pull-up resistor is enabled */
133              kPORT_FastSlewRate,                          /* Fast slew rate is configured */
134              kPORT_PassiveFilterDisable,                  /* Passive filter is disabled */
135              kPORT_OpenDrainDisable,                      /* Open drain is disabled */
136              kPORT_HighDriveStrength,                     /* High drive strength is configured */
137              kPORT_MuxAsGpio,                             /* Pin is configured as PTA4 */
138              kPORT_UnlockRegister                         /* Pin Control Register fields [15:0] are not locked */
139      };
140
141      PORT_SetPinConfig(PORTA, PIN4_IDX, &porta4_pin38_config);  /* PORTA4 (pin 38) is configured as PTA4 */
142
143      /* Button configuration sequence */
144      static BUTTON_HANDLE_DEFINE(buttonHandle);
145      button_config_t buttonConfig;
146      buttonConfig.gpio.direction = kHAL_GpioDirectionIn,
147      buttonConfig.gpio.port = 0;
148      buttonConfig.gpio.pin  = BOARD_SW3_GPIO_PIN; // PORTA6
149      buttonConfig.gpio.pinStateDefault = 1;
150      BUTTON_Init((button_handle_t) buttonHandle, &buttonConfig);
151      BUTTON_InstallCallback((button_handle_t) buttonHandle, button_callback, NULL);
152  }
```

The button callback function is called as a result of an interrupt caused by a button press. This callback function receives a message indicating the state of the button. A switch statement within the function performs different actions based on the type of button press. *Figure 11* provides the code for the callback function. It shows that a quick single press sets a variable to print the primary crypto-currency wallet QR code, a normal single press indicates to print the secondary crypto-curreny wallet, a double-press means to print the QR code for this project's Github, and a long press is meant to clear the clear.

Figure 11: Button Callback Function

```
button_status_t button_callback(void *buttonHandle, button_callback_message_t *message, void *callbackParam)
{
    button_status_t status = kStatus_BUTTON_Success;

    /* Set the display_state based on the type of button press */
    switch (message->event)
    {
        case kBUTTON_EventOneClick:
            PRINTF("kBUTTON_EventOneClick\r\n");
            display_state = kDisplay_PrintPrimaryImage;
            break;
        case kBUTTON_EventShortPress:
            PRINTF("kBUTTON_EventShortPress\r\n");
            display_state = kDisplay_PrintSecondImage;
            break;
        case kBUTTON_EventDoubleClick:
            PRINTF("kBUTTON_EventDoubleClick\r\n");
            display_state = kDisplay_PrintThirdImage;
            break;
        case kBUTTON_EventLongPress:
            PRINTF("kBUTTON_EventLongPress\r\n");
            display_state = kDisplay_Clear;
            break;
        case kBUTTON_EventError:
            PRINTF("kBUTTON_EventError\r\n");
            break;
        default:
            status = kStatus_BUTTON_Error;
            break;
    }

    return status;
}
```

All four functional blocks of the software layer are woven together in the main function of "eink_spi_firmware.c". *Figure 12* shows the main function first calling a wrapper method that initializes the button, and then a second method makes calls to initialize the e-ink display, SPI, and GPIO. Lastly, *Figure 12* also highlights that the first image printed to the e-ink display is the primary crypto-currency wallet.

Figure 12: Main Function Initializing the Functional Code Blocks

```
209     PRINTF("Welcome to the E-INK Display\r\n");
210
211     /* Initialize SW3 button on FRDM board*/
212     SW3_configure_button();
213
214     /*SPI and EINK display initialization*/
215     EINK_InitSequenceSPI();
216
217     display_state = kDisplay_PrintPrimaryImage; // start by printing the primary image
```

Once all functional blocks are initialized, the main function enters a while-loop that checks for updates from the button. This project utilizes a super-loop architecture, as opposed to FreeRTOS, since the main object for the controller is to print images to an e-ink display. The controller does not perform other processes, and it does not make real-time guarantees. Therefore, the super-loop suffices for this application. Figure 13 shows the super-loop.

Figure 13: Main Super-Loop

```
220     /* Super-loop checking for changes resulting from button press. */
221     while(1) {
222
223         if (display_state == kDisplay_Clear)
224         {
225             EINK_Clear();
226             display_state = kDisplay_DoNothing;
227         }
228         else if (display_state == kDisplay_PrintPrimaryImage)
229         {
230             displayPrimaryImage();
231             display_state = kDisplay_DoNothing;
232         }
233         else if (display_state == kDisplay_PrintSecondImage)
234         {
235             displaySecondaryImage();
236             display_state = kDisplay_DoNothing;
237         }
238         else if (display_state == kDisplay_PrintThirdImage)
239         {
240             displayThirdImage();
241             display_state = kDisplay_DoNothing;
242         }
243     }
```

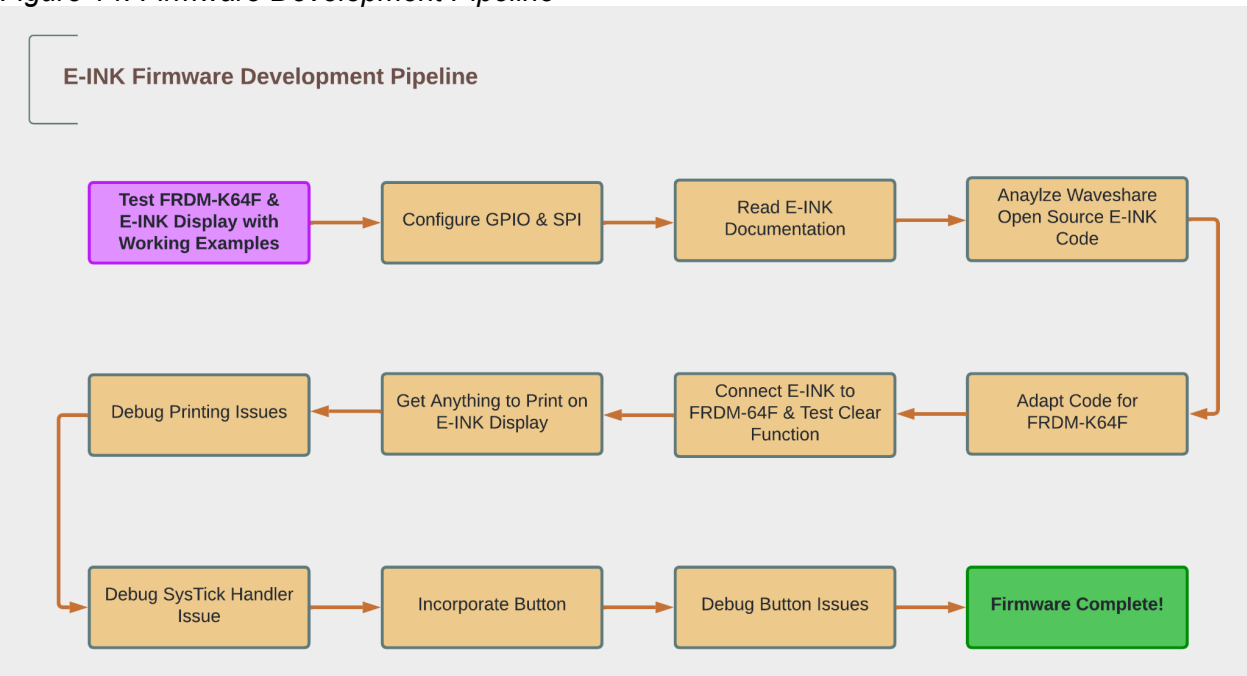
The Hardware Layer

The two main hardware components are the FRDM-K64F development board and the Waveshare 2.9 inch V2 e-ink display. As mentioned before, this project utilizes the boards RGB LED and button SW3. The LED is meant to provide some indication that the firmware started executing. Button SW3 allows a user to interface with the e-ink display. The button's firmware enables several functions, rather than just turning on and off the display. SPI data transmission between the board and the display is one-way. In this application, only the controller needs to send data over SPI to the display. The display communicates back to the controller with binary logic signals. In particular, the Busy pin informs the controller when it can send more data to the display. Though the firmware implements only a few hardware components, the development process to support those elements was not simple.

Firmware Development Pipeline

The firmware development process had its high and lows. The lows were mainly caused by stubborn bugs. However, those bugs were promptly resolved by stepping through every line of code and understanding what the processor was doing. Often, the hard faults and infinite loops were attributed to a module not having its clock source enabled or a timer not being initialized. This development journey is captured by *Figure 14*.

Figure 14: Firmware Development Pipeline



The first stage in the pipeline was ensuring that the hardware functioned properly. SPI capabilities were tested by connecting two FRDM-K64F boards and running the example SPI demo provided by MCUXpresso. This demo confirmed that the SPI module on the board is in working order. The e-ink display was tested by connecting it to a RaspberryPi and executing the test code available from Waveshare. The example code for the display demonstrated its full range of capabilities.

Configuring the SPI was straightforward since the code would be the same as the demo. The GPIO setup needed more probing. Test code was written to check that each pin was configured correctly. After each GPIO pin-write, a multimeter was attached to each pin to test that it was outputting 3.3V. Probe results were also validated against the GPIO register values. This methodically testing established a strong foundation for the rest of the firmware development.

Next on the docket was to understand how the programming model for the e-ink display worked. Waveshare's specifications outlined each step needed to eventually print an image on the screen. However, it proved to be cryptic and difficult to follow. Luckily, their source code followed every step outlined in the specification. The trick now was to adapt Waveshare's code for an STM32 board so that it could run on the FRDM-K64F. The key was identifying which functions triggered the SPI data transmission. In case, it was the `send_data()` and `send_command()` functions. Inserting the K64F's SPI transmission procedure into these two functions allowed Waveshare's code to be ported into the K64F.

With the code ported into the K64F, it was time to connect the e-ink display to the board and test out the clear display function. This was the first function to test since there was no method for creating an image at this stage in development. There was much elation when the clear function executed and the e-ink display flickered several times, ultimately returning to a blank state. This provided high hopes that the code transferred correctly to the K64F board. During this stage, it was discovered that Waveshare also provides a utility that allows a user to

convert a JPEG image into a byte array.[3] The goal now was to get anything to print on the e-ink display. Random pictures were converted into byte arrays. These arrays were inserted into the code, and the process was successful! Well, partially successful since what appeared on the screen was a garbled mess. But, this proved the code was working. Several tweaks to the utility settings helped transform the project's Github QR code into a byte array that properly printed on the display. Scanning the QR code on the display with a phone brought up the appropriate website.

At this point, it was time to transfer the code from the example SPI demo into a new project structure created with the help of MCUXpresso. In the new project structure, all of the code to operate the e-ink display was put into "eink.h" and "eink.c" files. Testing the code in the new project structure revealed a bug where the processor did not know where the handler for the SysTick counter interrupt was located. The source of the issue is that the project structure includes a directory for FreeRTOS. Consequently, this directory provides a handler for the SysTick interrupt. Invoking a delay caused the processor to call the wrong handler and hard fault. The solution was to comment out the FreeRTOS handler. This enabled the processor to call the custom handler within "eink.c".

The last phase of the firmware was incorporating support for the FRDM-K64F board's SW3 button. Once again the SDK's demos were incredibly helpful. There is a demo called "frdmk64_led_control_bm," and this demo provided the code necessary to implement the SW3 button. Integrating the button code allows a user to switch between various images and clear the screen. Finishing the tests for the button also concluded firmware development. The only thing left was to remove debugging code and provide helpful comments. The development process was long and frustrating. But it revealed the thrill and excitement of working with hardware.

---

[3] Waveshare Image2LCD.exe: https://www.waveshare.com/wiki/File:Image2Lcd.7z

Discussion of Custom PCB

The custom PCB shield provides a physical interface between the e-ink display and the FRDM-K64F board. This portion of the project focuses exclusively on the hardware layer, consisting of circuitry to support power requirements of the e-ink display through an FPC connector, as well as providing the necessary I/O connections between the display and the FRDM-K64F board GPIO.

E-Ink Shield Circuit

Waveshare's 2.9 inch V2 e-ink display uses a 24 pin ribbon connection for its I/O, which connects to the custom PCB through an FPC connector. Only a handful of these pins are used for controlling what is displayed, the majority of pins intended to power the display and control the voltage and current through it.

Pins 1 and 4 are specifically Non-Connecting pins, meant to be left open. For the purposes of the project, where SPI was used instead of I2C, pins 6 and 7 were also left open, as these are the I2C Clock and Data line respectively.

Pin 2 acts as the N-Channel MOSFET Gate Drive Control and Pin 3 acts as the Current Sense Input for the Control Loop.

Pins 5 & 20 are Positive Sources driving voltage. They need to connect to ground, with a 1μF capacitor to act as a low-pass filter. The filter helps remove high-frequency signals from the circuit by providing a low-impedance path to GND.

Pin 8 is the Bus Selection Pin. It allows for the selection between 4-line and 3-line SPI, depending on whether the pin is set 'Low' or 'High' respectively. From the specifications provided by Santiago, 4-line SPI is used for the firmware, requiring pin 8 to be set to 'Low' or ground. Rather than hardwire the pin 8 to ground on the PCB however, a 3-Pin jumper was implemented to allow a user to switch between the two modes. Pin 1 of the jumper is connected

to ground, while Pin 3 is connected to power. Pin 8 of the E-Ink Ribbon connects to Jumper Pin 2, the center pin. To set the board to 4-line SPI mode, a jumper cap can be used to connect Jumper Pin 2 to Jumper Pin 1, thus setting Pin 8 to ‘Low’.

Pins 9 through 14 connect directly to the FRDM-K64F GPIO, providing the interface for the FRDM board to control the e-ink display. Pin 9 provides the Busy state output for the E-Ink display, while Pin 10 is the Reset signal. Pin 11 is the D/C, or Data/Command control pin. Pin 12 is the Chip Select input, and per the data sheet is required to be ‘pulled low’ or connected to ground. Pins 13 and 14 are the SPI serial clock and data pin respectively.

Pins 15, 16 and 19 provide power input to the display, Pin 15 to the interface logic pins, and Pin 16 to the chip itself, and 19 to allow for OTP programming of the display. Pin 17 connects straight to ground, and pins 20-23 provide the Positive/Negative Source and Gates driving voltage for the circuit. And finally, Pin 24 is the VCOM driving voltage for the circuit.

Based on the functions of the pins, and the requirements for GPIO connections based on the developed firmware, the below Schematic and Board layouts were designed.

Figure 15: Custom E-Ink Shield PCB Schematic

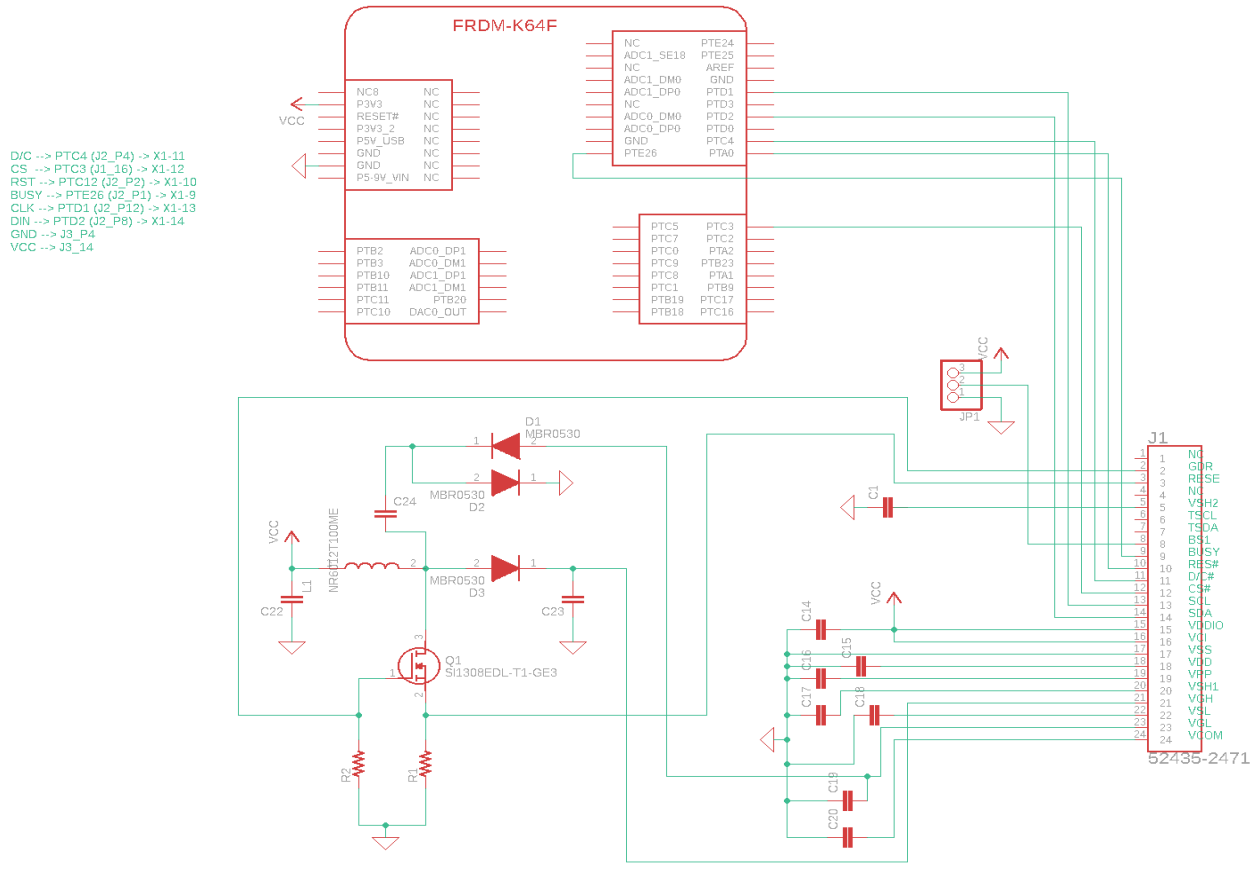
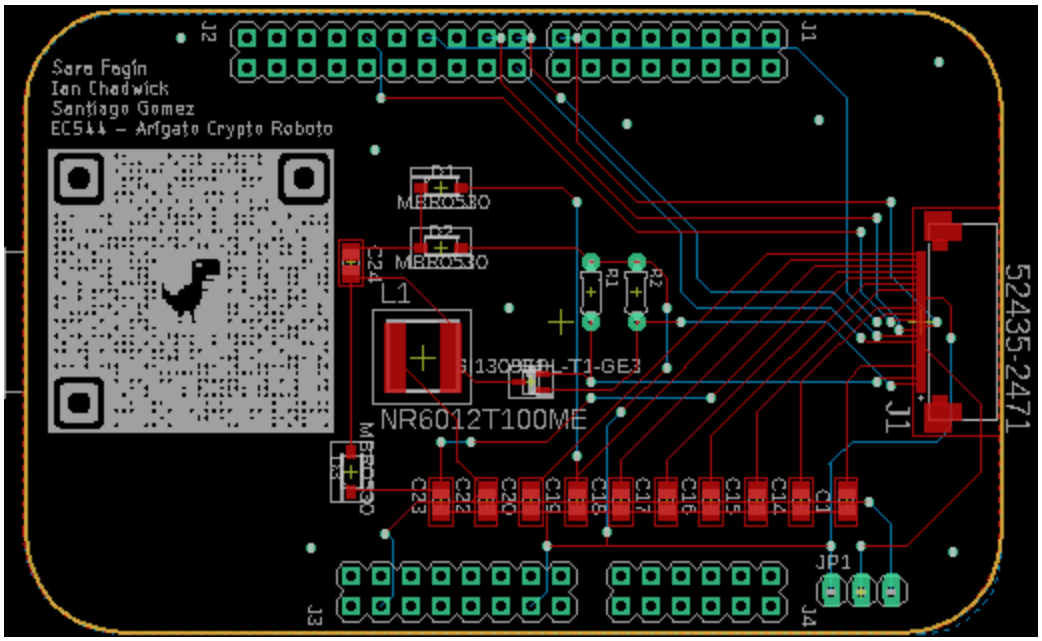
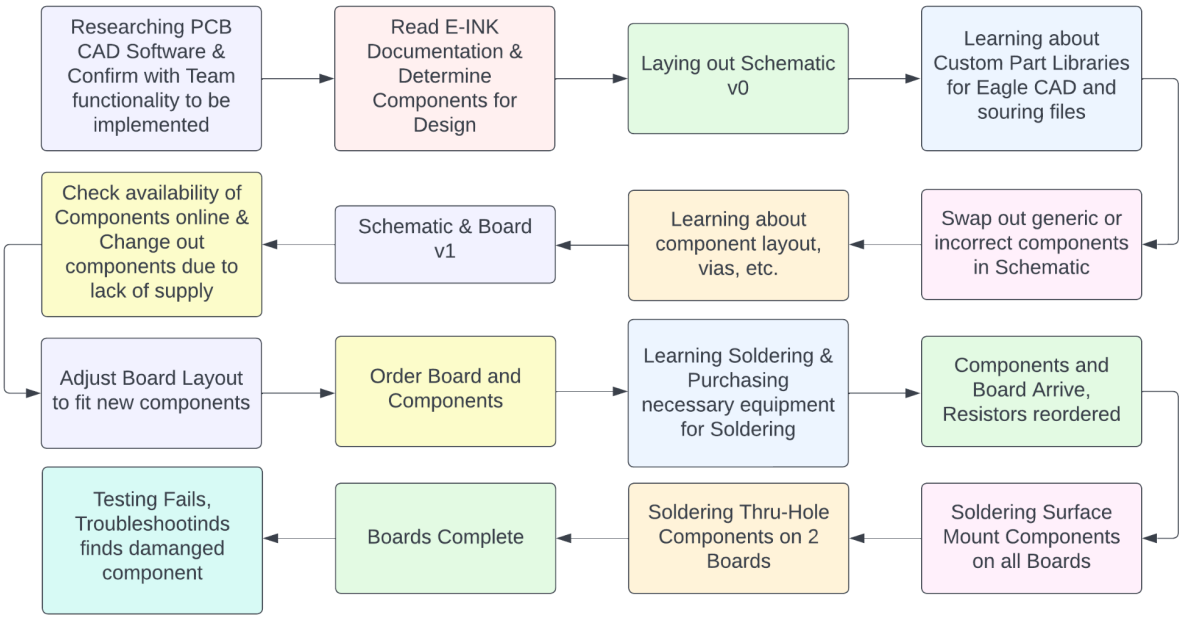


Figure 16: Custom E-Ink Shield PCB Board LayoutInk Shield PCB Board Layout



Custom PCB Development Timeline

Figure 17: Custom PCB Development Timeline



The development process for the custom PCB was extremely non-linear. For the initial schematic, the Eagle CAD Parts Library was missing some of the parts listed in the e-ink data sheets sample circuit. This required learning about and tracking down custom part libraries to import into Eagle for those missing components. Generating the board file from that schematic revealed the next issue. Generic components in the schematic do not always translate well to the generated board file. Again, new part files to match the specific components were needed. At this point, it appeared the schematic and board files were ready for production. However, before putting in an order for boards, a search for the components revealed multiple issues, including extremely delayed shipping times for backordered stock, and some components no longer in production. New components now needed to be chosen to replace those that could not be sourced within the time restrictions for the project. Choosing new components also required at least a basic understanding of what those original components needed to provide, so that the replacements could fill the same role. As an example, the initially planned 4.7uF resistors with 50v voltage rating on backorder were a concern at first. However, realizing that the voltage rating for the circuit could be considerably lower for the purposes of this project, which only requires a 3.3v source, allowed for several replacement options. The FPC connector in particular was difficult, as it needed to meet multiple requirements, including physical restrictions such as supporting 24 pins with contacts on the top and a .5mm pitch, among others.

With available components chosen and their part files obtained, a final schematic and board file, shown in the above figure, could be created and sent to 4pcb.com to be manufactured, and the components ordered. During the next two weeks waiting for everything to arrive, equipment for soldering needed to be purchased, and a space set up that provided good ventilation. Much of this time was spent watching soldering tutorial videos and practicing with inexpensive breadboard style pcbs. There was still some concern over soldering the FPC connector, as the contacts were extremely fine and close together relative to the other board components, and there was no good way to practice soldering them.

Once the board and components had all arrived, the soldering process was mostly straightforward. The wrong resistors had been ordered, meant for far greater power tolerances, and thus much larger than would fit on the board with the rest of the components. An Important takeaway here is to pay attention to dimensions and specs, even for commonly ordered products. Luckily, the correct resistors could be ordered from Amazon and arrive 2 days later. As the resistors were thru-hole components, they weren't needed for the initial round of soldering either.

Having ordered the new resistors, soldering the boards could begin. First the top layer surface mounted components were soldered on each of the three boards, with the exception of the inductor and FPC connector. The professor's help was requested for both as they proved difficult to solder with the current level of experience. For each component, a handy trick involving applying solder to one of the pads before placing the component, to reduce the number of things needing to be done at a time, helped immensely. The capacitors and MOSFET were all applied without issue. The Rectifier Diodes required some double checking of the data sheet, the schematic, and images online to confirm the correct direction they needed to be soldered. Further soldering was postponed until the FPC connector and Inductor could be looked at by the professor. Once that was done, the thru-hole components could now be soldered. Soldering the thru-hole components was completed without issue, and all that remained was testing the now constructed e-ink shields with the FRDM-K64F board and its new firmware.

Unfortunately, testing the boards did not end in success. Connecting the e-ink display to the shield, and the shield to the FRDM board GPIO produced no results. A Multimeter was used to test various components, checking if power was making it across the components. At each spot, the appropriate voltage was detected up to and including the FPC connector. At this point it appeared power was not the issue. A closer visual inspection of the FPC connectors showed what looked like instances of bridging across some of the pins, which could explain the failed

connection. A closer inspection with a microscope revealed bridging across important pins and disconnects of others. At this point it was clear the FPC connector was one of, if not the main, source of the problem.

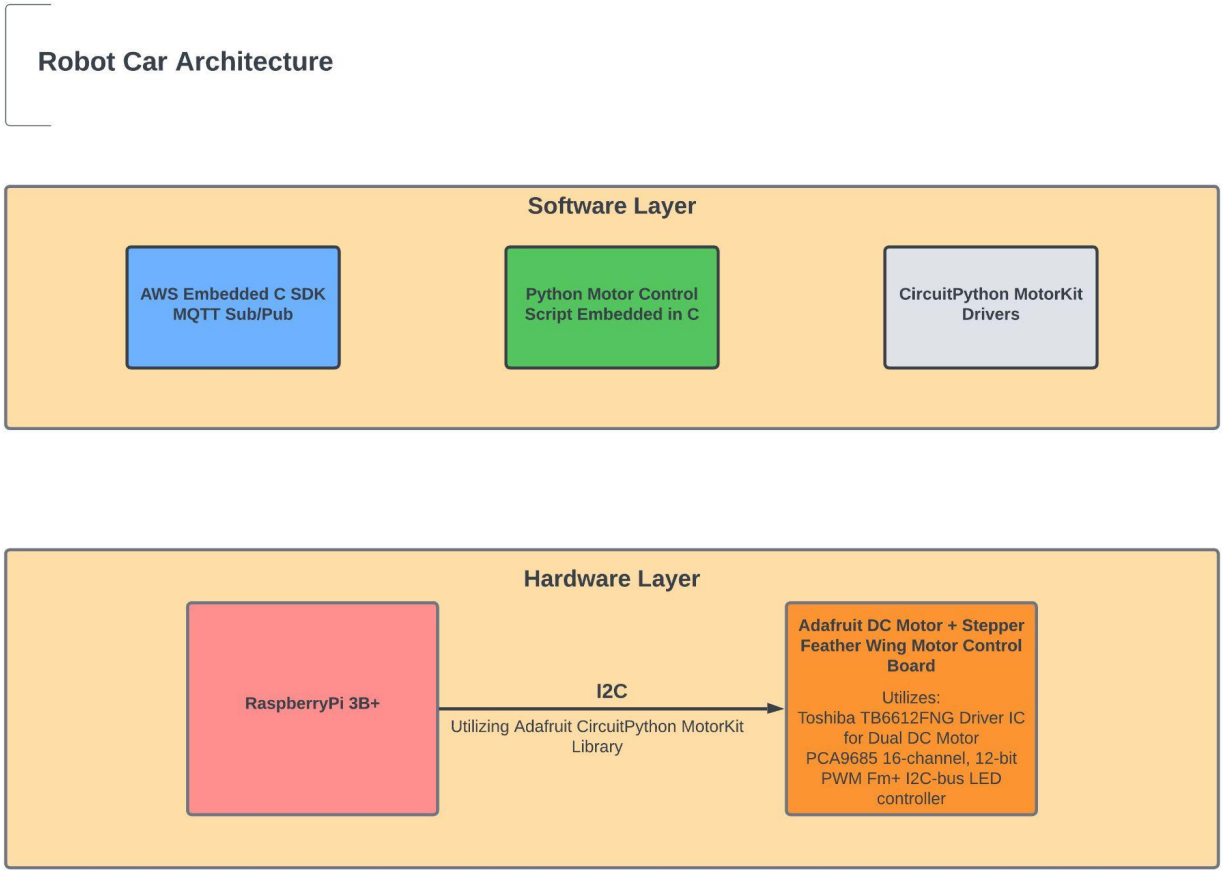
Discussion of Robot

The high level description of the robot car is that it subscribes to AWS MQTT broker topic in order to receive a published move command and drive forward a specified distance based on how much Dogecoin was put into a wallet (determined by the Crypto App). This section is divided into the two components of the robot, a software layer and a hardware layer.

The software layer of the robot car architecture incorporates the Amazon Web Services (AWS) Embedded C SDK, an embedded Python motor control script and the CircuitPython Motor Drivers to control the hardware.

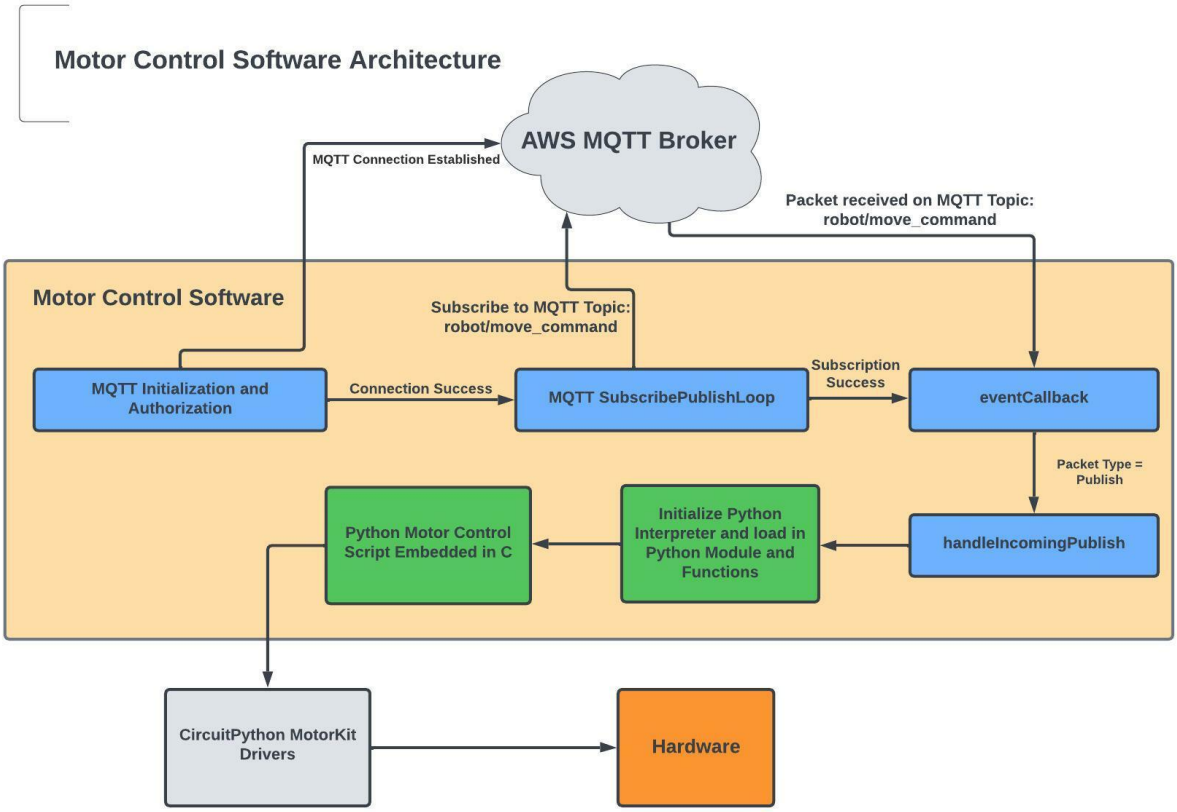
The hardware layer of the robot car integrates a RaspberryPi 3 B+ (RPi3) with an Adafruit Stepper + DC Motor FeatherWing motor control board (FeatherWing) which were mounted onto a 3D printed chassis in order to control the 2 DC motors, and the attached wheels.

Figure 18 Functional Block Diagram of Robot Car Architecture



The Software Layer

Figure 19 Motor Control Software Architecture



The software layer is composed of three main components, the CircuitPython motor drivers, the AWS C SDK based MQTT pub/sub executable (robot.c) and the motor control script written in Python and embedded into robot.c.

CircuitPython Motor Drivers

The motor driver will not be discussed in detail, since it is a part of CircuitPython, a pre-built library that was used because of the compatibility with the motor control board. CircuitPython was installed on the RPi3, along with all of the necessary dependencies to use the specific hardware that is integrated into the FeatherWing motor control board (see the hardware discussion for more detail).

AWS MQTT Pub/Sub

The file robot.c was based on the "mqtt_demo_mutual_auth" sample demo from the AWS Embedded C SDK and expanded upon that framework to develop what was needed for the project. Once the executable is run, it first initializes the MQTT libraries and then it enters a super loop. In the super loop it attempts to connect to the AWS MQTT Broker.

Figure 20 - Connecting to MQTT in main()

```
1497            returnStatus = connectToServerWithBackoffRetries( &networkContext,
1498                                                              &mqttContext,
1499                                                              &clientSessionPresent,
1500                                                              &brokerSessionPresent );
```

If a connection is not successful it will call a function which will wait a randomly generated amount of milliseconds up to a maximum limit and then attempts to retry a connection 5 times before terminating the program if it cannot connect.

Once the connection is successful, subscribePublishLoop is run, where it subscribes to a topic called robot/move_command.

Figure 21 Subscribing to topic inside subscribePublishLoop

```
1403        LogInfo( ( "Subscribing to the MQTT topic %.*s.",
1404                   MQTT_EXAMPLE_TOPIC_LENGTH,
1405                   MQTT_EXAMPLE_TOPIC ) );
1406        returnStatus = subscribeToTopic( pMqttContext );
1407    }
```

After it is subscribed to the topic, when the MQTT Broker receives a publish to that topic and sends it to the subscribers, this will trigger the callback function eventCallback. The eventCallback function will process the published message, identify the message as a publish and call handleIncomingPublish.

Figure 22 Process incoming payload in eventCallback

```
1035    if( ( pPacketInfo->type & 0xF0U ) == MQTT_PACKET_TYPE_PUBLISH )
1036    {
1037        assert( pDeserializedInfo->pPublishInfo != NULL );
1038        /* Handle incoming publish. */
1039        handleIncomingPublish( pDeserializedInfo->pPublishInfo, packetIdentifier );
1040    }
```

The function handleIncomingPublish will parse the message payload to get the integer value which corresponds to how many seconds to move the robot forward. The Python

interpreter is then initialized and the module motor.py and the function motor are instantiated as PyObjects in the C code. The integer from the payload is then passed to the motor function.

Figure 23 Motor Control Software Architecture

```
( const char * ) pPublishInfo->pPayload ) );
/* This section handles the payload if it is an integer and passes the value to the motor control
 * function*/
long int fuel;
// Extract the integer value from the payload
fuel = strtol(( const char * ) pPublishInfo->pPayload, NULL, 10 );
printf("The fuel value is %ld \n\r", fuel);
/* Begining of code for calling motor control python
 * module.
 * First create all required PyObjects*/
PyObject *pName, *pModule, *pFunc;
PyObject *pArgs, *pValue;
int i;
/* Initialize Python interpreter and establish path
 * to the script*/
Py_Initialize();
PyRun_SimpleString(
        "import sys\n"
        "sys.path.append('/home/pi/Documents/arigato/Arigato_Crypto_Roboto/RPi/motor_control')\n"
 );
 /* Find and create a Python module object for
  * motor.py and then free the memory*/
 pName = PyUnicode_DecodeFSDefault("motor");
 pModule = PyImport_Import(pName);
 Py_DECREF(pName);
 /* Get motor function from motor.py and pass arguments
  * from MQTT into the funciton.
  * */
 if (pModule != NULL)
 {
     pFunc = PyObject_GetAttrString(pModule, "motor");
     pArgs = PyTuple_New(1);
     pValue = PyLong_FromLong(fuel);
     PyTuple_SetItem(pArgs, 0, pValue);
     PyObject_CallObject(pFunc, pArgs);

     LogInfo(("Success!\n"));
 }
 // Free the memory and the close the interpreter.
 Py_XDECREF(pFunc);
 Py_DECREF(pModule);
 Py_Finalize();
```

Motor.py is a very straightforward module which instantiates the Adafruit MotorKit object, which in turn, controls the motors through I2C. The motor function takes in an integer argument and turns on the motors at a predefined speed, then waits for a number of seconds equal to the input integer.

Figure 24 *motor.py*

motor.py

```
import time
import sys
import board
from adafruit_motorkit import MotorKit

def motor(runTime):

    kit = MotorKit(i2c=board.I2C())

    kit.motor1.throttle = .5
    kit.motor2.throttle = -.5
    time.sleep(runTime)
    kit.motor1.throttle = 0
    kit.motor2.throttle = 0

```

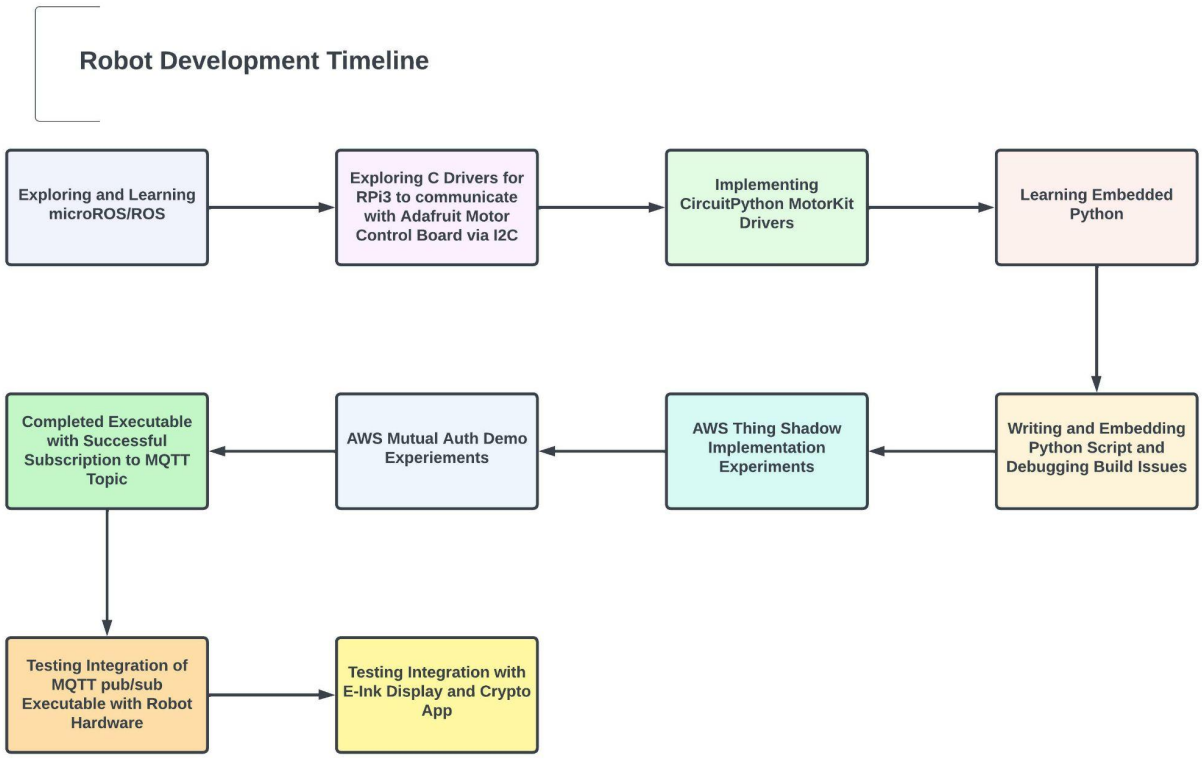
Once the motor finishes, the Python interpreter and all of the objects are freed from memory and the super loop continues and restarts the process.

The Hardware Layer

With the exception of the RaspberryPi, these components for the car were adapted from a JetBot kit, to be used as the basis to build and develop the remaining software components of the robot car. It was decided to use the existing chassis, motor and wheel setup from the Jetbot to save development time that would be expended by building everything from scratch. The FeatherWing motor control board is actually composed of two integrated chips for controlling the motors via I2C, a Toshiba TB6612FNG Driver IC for Dual DC Motor and a PCA9685 16-channel 12-bit PWM Fm+ I2C-bus LED controller.

Robot Development Timeline and Initial Directions

Figure 25 Robot Development Timeline



The initial project proposal outlined software architecture for the robot motion control which utilized the FRDM-K64 board as the central control for the robot which would then communicate with the motor control board via I2C. The FRDM-K64 board would then utilize MicroROS middleware on top of FreeRTOS to communicate via wifi connection and a ROS1 bridge to a linux laptop running ROS2. This would allow a ROS sub/pub connection to be established for sending commands to the robot's motor controller board. Then the laptop would communicate with a Websocket server that was also connected to Decentralland, where users could control the speed and a virtual representation of the robot would mirror the movement.

However after several weeks of deeper exploration of microROS/ROS2, it was found that microROS/ROS2 might not be the best solution for this particular use case and that the FRDM-K64 board may not be the best for controlling the robot, considering the hardware in use. After some discussion, it was decided that the best course of action would be to pivot away from

using the FRDM-K64 board to control the robot and instead use a RaspberryPi B+ with the Amazon Web Services Embedded C SDK to establish an MQTT connection for the pub/sub component of the robot control.

This raised the issue of how to control the motor driver, because the available drivers from Adafruit for the FeatherWing utilized CircuitPython instead of C. A week was spent exploring C libraries for writing new C drivers for the FeatherWing board utilizing I2C and the RaspberryPi's GIPIO pins. Some of the libraries explored include, WiringPi (which was not used because it was deprecated), PiGPIO, and the BCM2835 C library, which was chosen for its better documentation and compatibility with the RaspberryPi 3's Broadcom BCM2837 CPU. However, after some discussion it was again decided that developing new drivers for the FeatherWing board was too far outside of the scope of the initial project and that the CircuitPython motor drivers would be sufficient given the scope of this part of the project was to focus on the connectivity aspect, rather than developing new drivers.

The next challenge was determining how to call the Python function that controlled the motor from within a C program. Fortunately, there is a relatively straightforward way of accomplishing this task with a method called embedding Python, that can be done by using the Python.h library. A little over a week was spent researching and understanding how to embed Python in C, quickly writing a script in Python and connecting the RPi3 to the FeatherWing to test the motor control script and functionality. One challenge was to determine how to incorporate the necessary parameters into the build process that came with the C SDK for each of the demos. This was necessary because the authentication with AWS with the certificates, and keys that are required for access to the service were abstracted away as a part of the build process. In order to build the executable correctly, so that it would function with AWS without having to create a new process for ensuring proper authentication from scratch, it was necessary to use the provided Cmake files. The organization of the SDK's build process required several Cmake files at different levels of the directories to be modified. This took some

additional time to debug and to ensure that the paths to all of the new libraries were able to be found correctly and that the new build commands could be incorporated correctly.

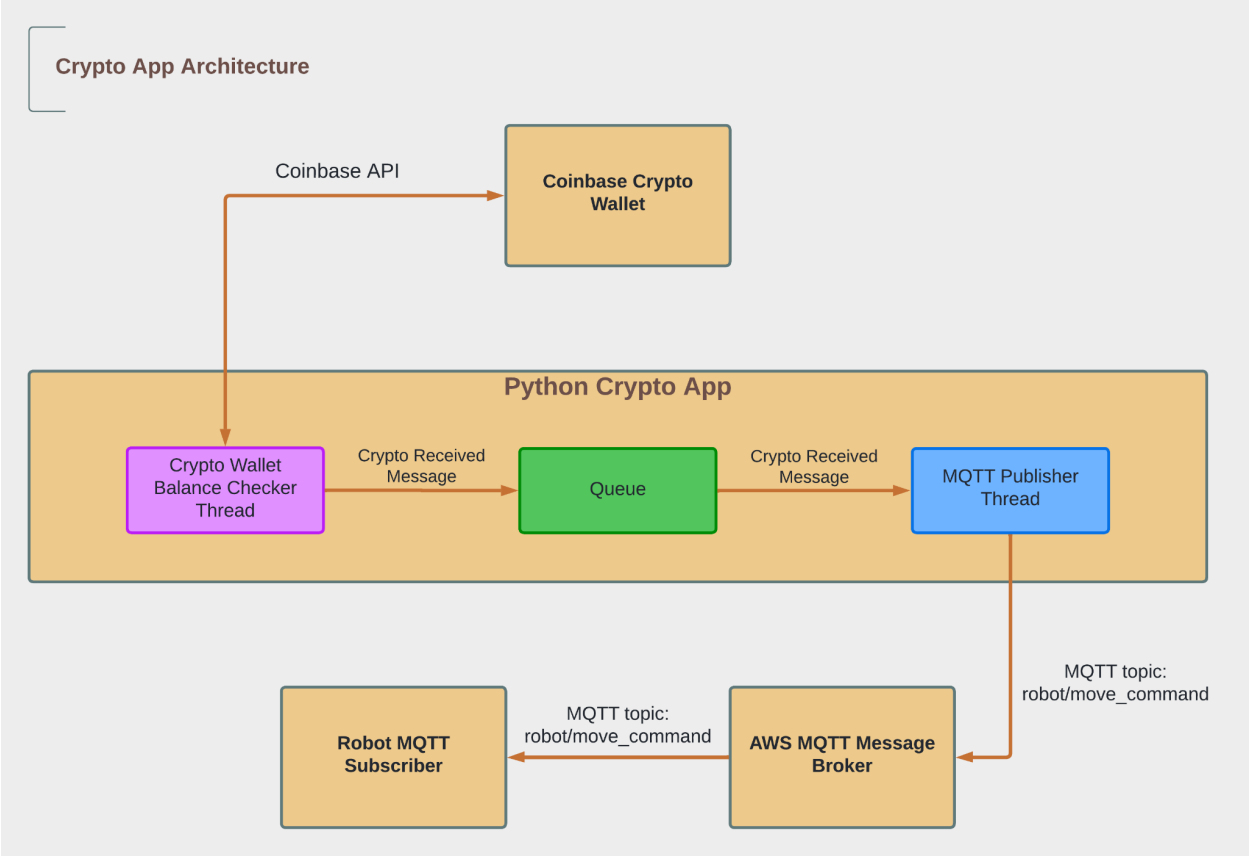
Once the final software architecture was decided, the work on implementing the MQTT sub/pub using the AWS C SDK began. It took a few days to read through the documentation and understand how the functions were working and related to one another and to determine which of the demo files would be the best way to implement the functionality.

Initially, the plan was to utilize the Thing Shadow as a way to update the robot's movement status. A week was spent trying different implementations based on the device_shadow_demo. However, after several iterations, it was decided that the device shadow was not the best way to implement the overall goal of the project. Instead, the mqtt_demo_mutual_auth was used as a basis for developing the final product. The mutual auth demo was tested using the AWS MQTT test console and it was successful in receiving the published message. Then the motor.py script was incorporated into the c code, and tested with the robot. To this point all testing was done with a breadboard while connected to a power outlet. So, the robot was then tested with a direct connection between the RPi3 and the motor control board while operating on battery power and it successfully received messages over the simulated AWS MQTT test console and drove forward. Unfortunately however, during late stage testing with the full crypto app, it was found that the MQTT subscriber program was not performing quite as expected (see failures section for more details).

The Crypto App

The Crypto App is a multi-threaded Python application that scans for updates on a Dogecoin wallet hosted by Coinbase. Updates to the wallet trigger messages to be published to an AWS MQTT Message Broker. In particular, the balance checker thread makes calls to the Coinbase API, and then communicates with the MQTT publisher thread via a queue. When there is a message in the queue, the MQTT thread grabs the message and forwards it on to AWS. The messages represent the distance the robot should move. Specifically, 1 Dogecoin equals a short distance, 2 Dogecoins is a medium distance, and 3 Dogecoins or greater drives the robot a long distance. The datatype of the message is an integer. Therefore, the message is either a 1, 3, or 5. The integers represent the number of seconds the motors will run. Figure XX outlines the dataflow from the Coinbase wallet, through Crypto App, and into AWS.

Figure 26: Crypto App Architecture



Discussion of Failures

The original project proposal included connecting the robot with the metaverse, using FreeRTOS on the FRDM-K64F board, using microROS/ROS and putting LEDs on the custom PCB to indicate the speed of the robot. All four of these elements were not implemented.

The metaverse connection was an exciting proposition, given that this is a trending topic. However, several obstacles distracted from the core technologies of this project, MQTT, firmware, and PCB design. Those obstacles included learning TypeScript and game design principles. These two items are the prerequisites for developing an immersive world with Decentraland. The original idea was that a user updates a virtual car in Decentraland by paying some crypto-currency. That update would then propagate to the robot car in the physical world. Given that the project veered from the metaverse, it was still important to receive a signal from a crypto API. Therefore, the final implementation makes a call to the Coinbase API to get the balance of a crypto wallet.

In addition, as was discussed in the Robot Development Timeline section, the original proposal called for ROS to be used with a Websocket server. This server would then communicate with Decentraland. But, it was ultimately decided that the MQTT pub/sub technology would be better suited for the project. As a result, plans to use ROS and microROS on the FRDM board were scrapped in favor of using the RPi3 and the AWS C SDK.

One of the failures that we noticed during testing was that there was an initial misunderstanding of how the MQTT pub/sub was supposed to function. The demo that was used as a basis, connected, subscribed to a topic, published to the same topic and then disconnected in loop. That reconnect loop was initially thought to be a prerequisite for keeping the MQTT connection alive, but during testing it was discovered that sometimes some published messages were either missed, or delayed. This was likely because the robot continuously reconnected to the MQTT server and if a publish was pushed down while it was reconnecting

then the message would be lost or delayed. An attempt to correct the issue by moving the MQTT functions to before the superloop was made. However, when the program was run, the callback was never called. In this case the robot would never receive published messages from the MQTT broker. Due to time constraints it was decided that since the robot received the published message from the MQTT server most of the time, the issue would be fixed in a later iteration of the project.

Finally, the last feature that was not added were the LEDs on the PCB. Once it was determined that Decentraland was going to be omitted from the final product, it also made sense to exclude the LEDs from the PCB, since their primary function was to provide the physical user an indication of the speed set in Decentraland. There was some discussion about whether to include them anyway, but given the difficulty of ordering parts due to the supply chain issues caused by COVID, it was eventually decided that while it was a nice to have, ultimately it would be extraneous.

Discussion of Success

The final project differed slightly from the initial project proposed and some of those deviations have been discussed in the respective sections of the report and in the previous failures section. However, four goals that formed the main pillars of the project: developing a custom PCB shield for the FRDM-K64 board with an e-ink display, writing new firmware for the e-Ink display and button functionality on the FRDM-K64, an autonomous robotic car controlled by commands from MQTT on AWS and a CryptoApp which monitors a Coinbase wallet to send commands to the robot via MQTT. All of these components were essential pieces of the project design. Regardless of the challenges and setbacks in various areas, overall each of the goals were achieved and the project was ultimately successful.

The PCB was successfully completed despite many supply chain issues that occurred due to COVID and other global supply chain disruptions that have been affecting the price and general availability of electronics and components.

The firmware development was also successful in integrating the Waveshare 2.9 inch V2 e-ink display with the FRDM-K64 board. This was accomplished through rigorous testing and configuration of SPI, GPIO, LED and the Button components to control the display.

The implementation of the robot car subscribed to MQTT was successful, baring some inconsistency with receiving packets. However, the robot motor control board and associated hardware seamlessly integrated with the RaspberryPi. The RaspberryPi effectively translated the messages from the MQTT broker to the motors. As mentioned previously, the motor control functionality is implemented by embedding Python in the callback function for MQTT.

The goal of the final pillar of the project, the Crypto App is a Python program that monitors the balance of a Coinbase crypto wallet and publishes movement commands MQTT Broker on AWS. It does this by making calls to the Coinbase API and then publishing to the

move command topic based on changes in the wallet balance via two threads. The crypto app serves as the perfect bridge between the cloud layers and the low level hardware.

Conclusion

The overall goal of this project was to demonstrate and implement the integration of cutting-edge and innovative technology, such as cloud computing, cryptocurrency, and low-level embedded IoT devices. The aim was also to show how embedded systems can be used to create novel interactions between the virtual and physical world. On the surface, the Arigato project provides the entertaining experience of powering a robotic car with crypto-coins. However, it is a proof-of-concept idea of how technologies like crypto-currencies, which have been traditionally considered the domain of desktop computers and server farms, can be used in conjunction with resource constrained devices. These new connections provide the opportunity to create machine-to-machine communication systems. As more devices are connected to the internet, machine-to-machine communication will be vital for autonomous devices to address pressing societal needs.

Next steps for the project would be to fix some of the issues with the MQTT communication to ensure that packets are not missed and to add in the additional interactivity with Decetraland that was cut from the project's original scope. These additional features would further bridge the gap between the metaverse and the physical world, laying the foundation for a new decentralized cyber-physical economy. Arigato the Crypto Roboto provided a vital experience for designing the technology that will drive this new economy.