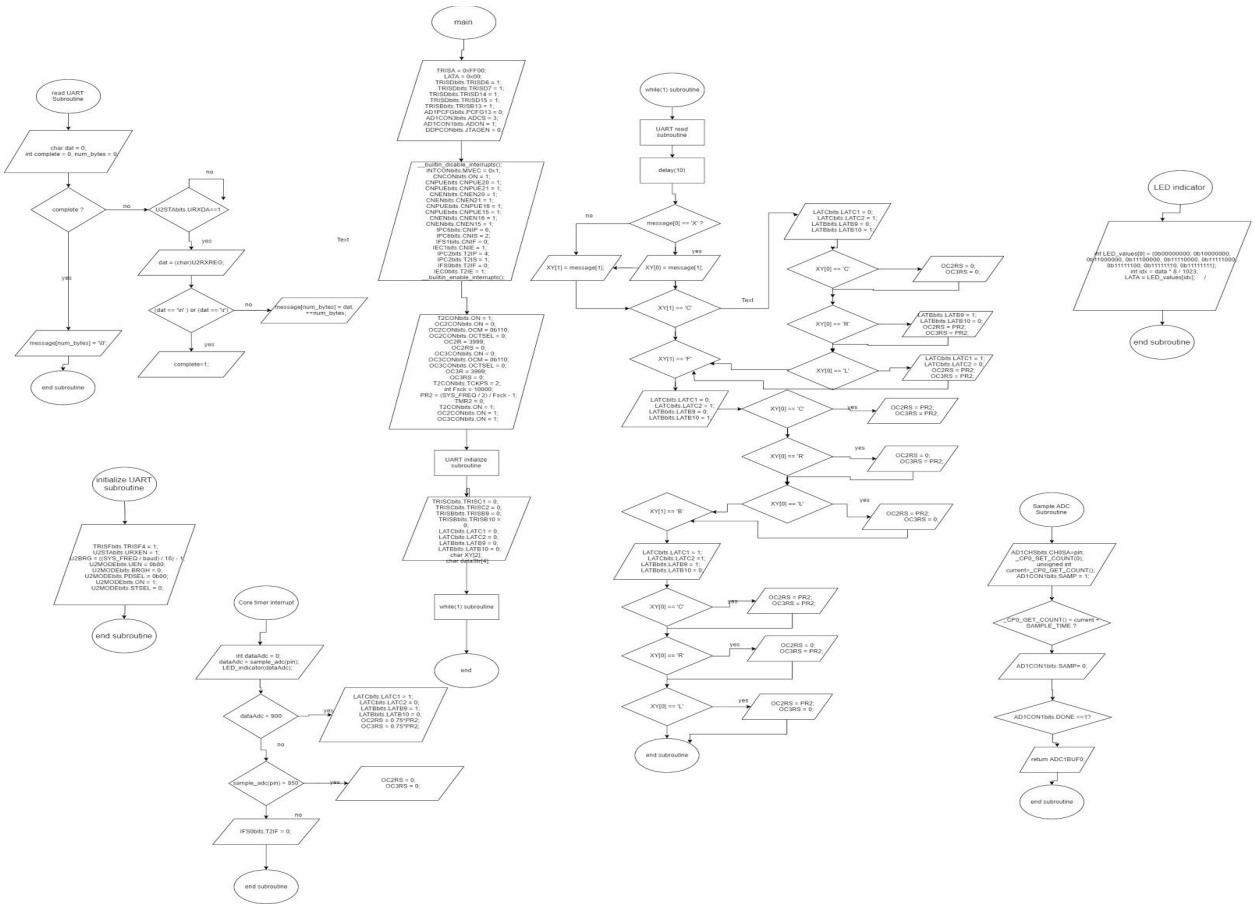
Team name: Lil Latch E
Team members: Clara Martinez Rubio
Soma Mizobuchi
Richard Akomea
Ronaldo Naveo

Abstract: Drive a remotely operated vehicle using a gamepad. The project consisted of a gamepad which sent commands to the Raspberry Pi 3. These commands were parsed in order to be sent to the Explorer 16 board over UART communication. These commands were then translated into speed and direction for our two motors. At the end, using the analog pad from the gamepad our vehicle Model-E can move forward, left, right, back and diagonal. Our Model-E also includes a distance sensor, which will shortly force the vehicle to move back in order to not bump into objects.



INTERFACING BETWEEN THE PIC32 BOARD AND THE RASPBERRY PI

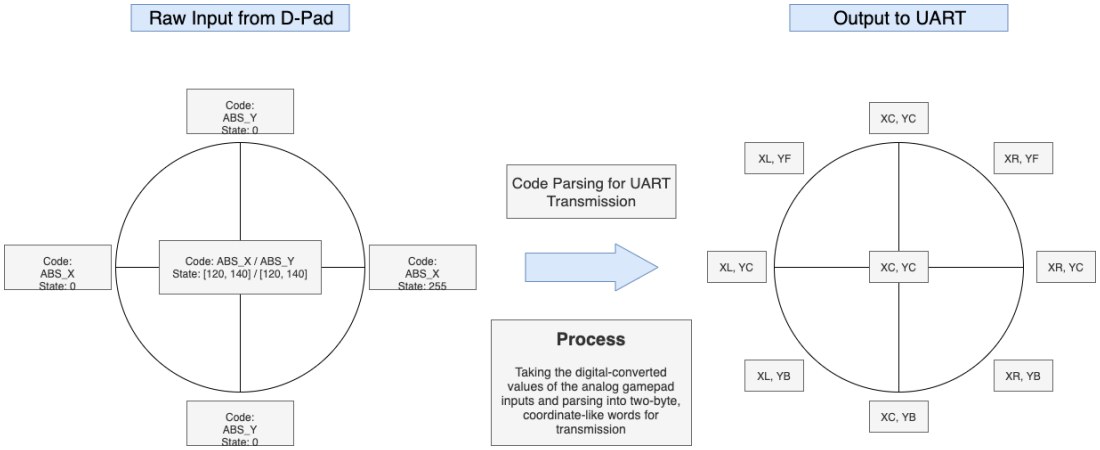
E1.

For interfacing the Logitech F710 Wireless gamepad on the PIC 32 board we used UART communication protocol. We wrote a python program to send the gamepad event codes as messages indicating direction to the PIC32 board in order to control the direction of our motors. The python script turns the input from gamepad into two concatenated 1 byte words, one per axis.

While exploring the various buttons and sticks on the controller, there were several candidates that stood out. Namely, the joystick analog inputs, as well as the d-pad seemed the most intuitive solution for controlling the motors. The resolution given by the joysticks ranged from a 8-bit digital reading to a 16-bit digital reading for each axis, X and Y. The d-pad, on the other hand, took the endpoints of the 8-bit value and mapped each to the vertical or horizontal axis. In other words, pressing up on the d-pad gave a Y value of 0, 128 in its rest state, and 255 when down was pressed.

The codes were sent in two strings: an “ABS_X” or “ABS_Y” followed by either the 8-bit or 10-bit value. After experimenting over UART, it was clear that the values given by 16-bit mode were simply too large for the purpose of this project. Not only was the communication made sluggish, the UART RX buffer on the PIC32 was being overloaded. After several iterations of testing, we decided that the d-pad input, which gave only 5 values for each axis, was sufficient.

Using Python and simple conditional logic we took the inputs from the controller and remapped them to two-byte words to send through UART. The diagram below shows the logic:



To minimize the amount of data and processing needed, we used characters as opposed to integers for the second byte. This also increased legibility and improved the debugging process immensely.

We initially had a Python Dictionary type with key value pairs to parse the input, but the analog nature of the d-pad sometimes gave unexpected values. For example, we would get index errors for expecting a 128 in the rest position when the gamepad sent 130 or values close, but not exact. The function implementation, though not the most elegant, did the trick:

```
def gp2dirX(gp):
    if gp < 50:
        return 'L'
    elif gp >= 50 and gp <= 205:
        return 'C'
    elif gp > 205:
        return 'R'
```

For data transmission, we used a standard 8-N-1 configuration: 8 data-bits, no parity, and 1 stop bit. The baud rate was set to 9600 bits/second, as this yielded the most consistent and fast performance without overflow errors. The code for the UART transmission setup is listed below:

```
import serial
from inputs import devices
from inputs import get_gamepad
gamepad = devices.gamepads[0]
serial_interface = serial.Serial('/dev/ttyAMA0',9600)
print(serial_interface.name)
```

CONTROLLING 2 DC GEAR MOTORS ON THE PIC32 BOARD

E2.

The motors are able to be controlled individually, since for each direction the motors behave independently. This is done using 4 GPIO.
Our program reads from the two 48 counts encoders. These trigger the change notifications when the motor rotates and a counter is incremented or decremented according to the direction of rotation.
By setting or clearing each of the GPIO pairs and modifying the value of the 2 PWM, our vehicle can drive forward, back, left and right, plus it can drive diagonal, according to the messages received from the gamepad.

CONTROLLING DC GEAR MOTORS ON THE PIC32 BOARD WITH GAMEPAD MESSAGES

E3.

We modified our main C script to change the PWMs and GPIO pin directions based on our gamepad messages in order to control the speed and direction of our remote control vehicle. The PIC 32 board will receive X and Y coordinates from the raspberry pi through UART communication. The messages that the PIC32 board can receive is “C”, “L”, and “R” based on the X position on the D-pad and “C”, “B”, “F” for the Y position of the D-pad. The messages are parsed into a char array with two indexes; the first index for the X position and the second index for the Y Position. After the messages are parsed, the direction of the Y coordinate is checked first followed by the X position

which will then determine which GPIO pin Latches to flip and the speed of the PWM signals to control the DC gear motors.

To turn right we reverted the speed of the right motor, to simulate rotating about the center of the vehicle. Opposite to turn left.
To drive diagonally forward to the right, the right motor slows down to half the PWM, and vice versa to go diagonally left.
To go backwards, all the GPIO pins invert, changing the motor's polarity. We can also drive back diagonally, in which case, to drive diagonally left, the left motor PWM reduces by half, and vice versa to turn diagonally right. When going forward and backwards, both motors duty cycle are 100%.

ASSEMBLING HARDWARE AND DESIGNING OUR VEHICLE
E4.

Minimal design changes were made to the provided chassis, but design decisions were made based on a case-by-case basis. The chassis, constructed from an aluminium frame and a plexiglass panels, resembled a L-shape with a panel on the bottom and a panel tangent to it on the rear. On the bottom, a swivel wheel (passive) was added as the third support wheel in addition to those connected to the motor (active), while on the rear, the touchscreen was mounted. Cutouts on the bottom panel allowed the wheels to be recessed into the inner fold of the L-Shape, lowering the center of gravity, thereby increasing stability.

While there are several key decisions that we made that distinguished our model, the outward appearance was quite common compared to other groups. Most groups opted to use a cube-like structure by adding another L, but we decided to cut down on weight by omitting this. To compensate for the lack of friction provided by the uneven weight distribution, we mounted the battery packs parallel to the wheel axle length, as far out as possible to increase the downward torque applied to the wheels. The imbalance was caused by the touchscreen being rear-mounted and upright to the with the panel, causing torque focused on the rear end of the vehicle.

Despite compensating for the weight imbalance, the tires did tend to lose traction on dusty or slightly uneven surfaces, causing the vehicle to turn in directions that were not intended. However, this did not pose a threat to the vehicle getting stuck in place and could easily be circumvented with the gamepad controls.

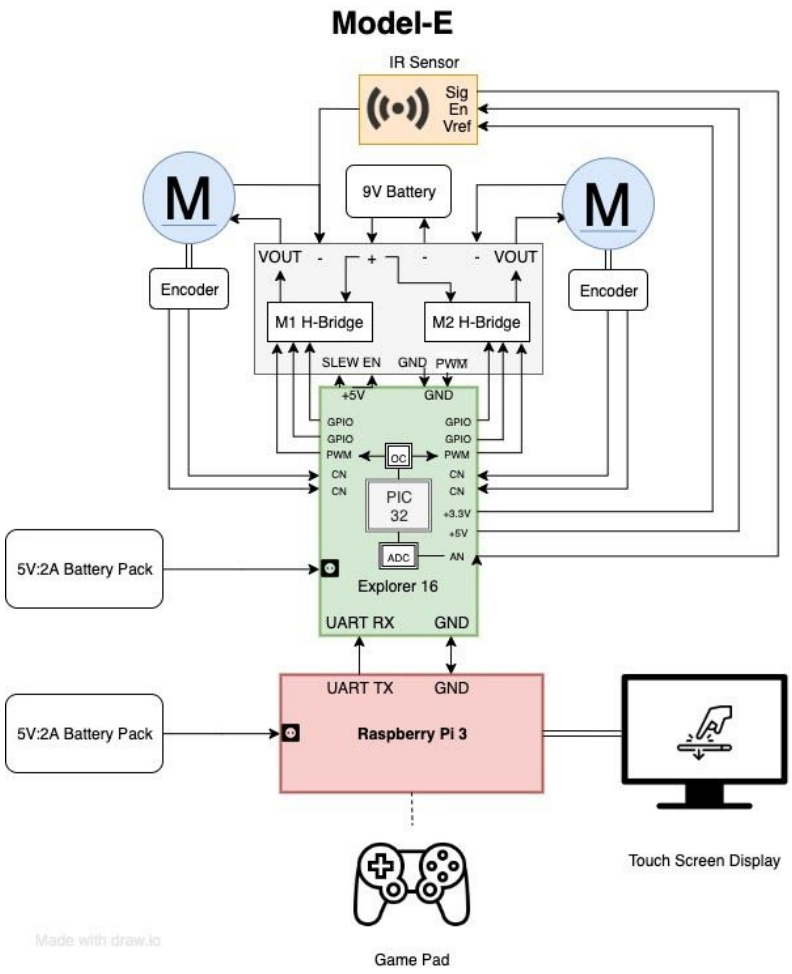
Mounting the PIC32 with Explorer 16 board onto the chassis was the center of focus. On top of being the component with the largest area, it also was the central unit of control for the motors and sensors. By design and intuition, it was placed face up at the center of the bottom panel where pins were easily accessible from all directions. The schematic diagram shown in the **Connections** section illustrates just how central the PIC32 was to the motor vehicle. The Raspberry Pi, was conveniently mounted back-to-back on the screen where the display ribbon cable was easily accessible.

No further designs were added besides those mentioned above. As velocity was not a focal point, neither was drag, traction control, and breaking. For this reason, the cables were left exposed, yet organized in such a way that allowed easy manipulation while still remaining somewhat robust. Peripherals not mounted by screws - motor driver, 9V battery, and lithium ion battery packs - were taped down with electrical tape.

CONNECTIONS

Model E: PINOUT			
EXPLORER 16	**PIN**	**I / O**	**PERIPHERAL DEVICE**
OC2	76	Output	MD_PWM1
OC3	77	Output	MD_PWM2
GPIO	6	Output	MD_M1IN1
GPIO	7	Output	MD_M1IN2
GPIO	32	Output	MD_M2IN1
GPIO	33	Output	MD_M2IN2
CN15	83	Input	M1ENC_A
CN16	84	Input	M1ENC_B
CN20	47	Input	M2ENC_A
CN21	48	Input	M2ENC2_B
UART2_RX	49	Input	RPI3_UART_RX
AN13	42	Input	IR_SENS_SIG
+5V	5V	Output	MD_EN, MD_SLEW, M1ENC_PWR, M2ENC_PWR, IR_SENS_EN
+3.3V	3.3 V	Output	IR_SENS_VREF

* MD - Motor driver
* ENC - Encoder
* M1 - Motor 1, M2 - Motor



EXTRA CREDIT: Distance Sensor

The implementation of a bumper sensor seemed as an intuitive addition to our vehicle. Namely its widespread use in cars today gave us the inspiration. Using an infrared emitter-detector pair as our source of input, we measured the distance between the Model-E and obstructions by first converting the analog signal using an ADC. The 3.3V reference voltage was approximated to a 10-bit value using successive approximation done in the multiplexer within the Explorer 16 board.

Taking guidance from information we found about the specific model, we mapped the 10-bit digital value to distances in comprehendable units (I.e centimeters). After confirming the turn radius, we settled on 15cm as the threshold in which the vehicle would encounter an obstruction. The vehicle would move backwards and override any further user input until it reached this threshold. We did this by setting an interrupt for the ADC with a higher priority than the UART receiver subroutine in the main loop. In the Interrupt Service Routine we added the control statement. After shortly moving backwards, the control will be returned to the user.

TESTING

1. Testing Gamepad connecting to Raspberry Pi: To test the functionality of the gamepad as well as see the gamepad event codes(messages) we were going to send over to the the PIC32 we adopted and modified the code used in class exercise 13 to print the gamepad event codes onto the console.
2. Raspberry Pi to PIC32 through UART; To test if signals were being transferred over from the Pi to the PIC32 or vice versa we connected sed input pins to the oscilloscope test see if signals were being transferred when the buttons were triggered. We also used the driver display code from early labs to help us debug our programs. The LCD display screen is used to display values for encoder count, messages received from Pi and other characters/register values which enabled us debug communication and motor control errors amongst others.We also used the LEDs to help debug our code by putting them in specific points in our code to help us see if an ISR has been entered,buffer was full,message has been read etc.
3. H-Bridge and motors: To test out motors we first connected them directly to power with change notification pins to see if they were functioning properly. We made sure to have a working battery for every test run as sometimes a defective battery can affect the speed of the motor
4. Test fully assembled Model-E: Finally the touchscreen was mounted and tested for functionality. The battery packs were placed on the sides close to the sides to help with balance and add friction in order to help the tires turn on their axes.

CHALLENGES & LESSONS

There were multiple challenges that our group faced when implementing our ideas for our Model-E. One challenge we had was having a clear outline of the project timeline and project parts. At first we started off doing SPI communication and using the analog stick to send information for the PIC32 board. However, we did not know how to deal with buffer with SPI communication and how to parse the information from the analog stick. With this miscommunication, we were setback by a few days and had to redo our design of how we wanted to implement our project by changing SPI to UART and using the D-pad and the wireless controller. The lesson learned was that we have to create very in depth outline and schematic our how our project should function and split up the work accordingly. Having the pins and outline already written would've made our group days ahead.

Another challenge we struggled an abundance with was debugging. There were many times where we did not know where something was failing at whether it was the software or the hardware. This sometimes took us a day to debug a simple coding issue of what was failing. We implemented using the LCD display and LED's later on through the project for debugging and it helped us out tremendously and increased the production time in finishing our project. Our group knows now to test different parts of the hardware separately and not try to debug many different codes of the project at the same time as too many issues will arise. The debugger is also a great friend when you

know the LCD display is working but you are not seeing anything showing.

DEMONSTRATION

The initial demonstration was halted due to technical difficulties with the gamepad. The cause, as we figured out soon after, was that the AA batteries were dead. Although this was an honest and unforeseen error, it is still a disappointment in any case. Knowing that our robot had performed seamlessly in many testings, the failure to perform in front of the big stage did not feel great. In spite of this setback, we were able to debug and solve the issue in a matter of minutes.

On our second try, the Model-E performed as we intended. The first 50cm of travel was not straight due to the lack of traction in the tires, but we were able to make accurate adjustments using the gamepad. Furthermore, the fore-mounted IR sensor would not allow any forward collisions in the case that the vehicle went out of control. We forced the Bumper Sensor to activate in the demo by simply going full throttle into the box ahead of the vehicle. As we expected the vehicle maintained a 15cm cushion and took over control from the human.