Lab Guide
Manual Drive

Content Description

The following document describes a manual drive implementation in either python or MATLAB software environments.

MATLAB

In this example, we will capture commands from a Gamepad and use it to manually drive the QCar platform. The application will also display the percentage battery remaining, power consumption in Watts as well as the car's speed in m/s. The process is shown in Figure 1.

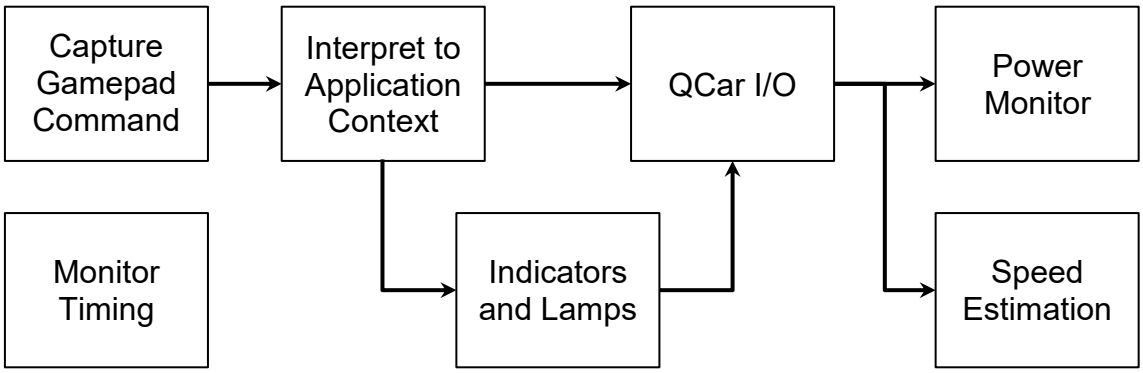

Figure 1. Component diagram

In addition, a timing module will be monitoring the entire application's performance. The Simulink implementation is displayed in Figure 2 below.

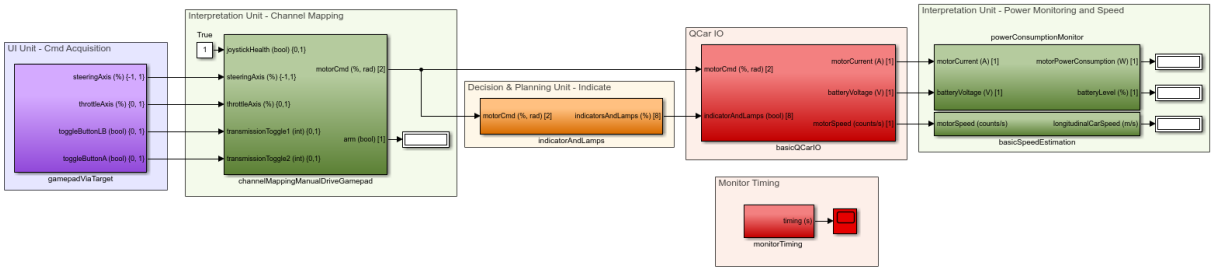

Figure 2. Simulink implementation of Manual Drive

Running the example

1. Check User Manual – Software Simulink for details on deploying Simulink models to the QCar as applications.

2. Before running this example, connect the **Logitech F710 Gamepad** (provided with the **Self-Driving Car Studio**) USB dongle to one of the USB 3.0 ports on the QCar.

Guide

1. Driving manually is mapped to the following gamepad sticks/buttons:

    a. **Left Button LB** for Arm - **QCar will be armed when this is pressed (1)**, and steering/throttle will not respond when it is released (**0**).

    b. **Left Stick** for steering - stick all the way to the left position is positive, steering the wheels left as well.

    Note: The LED light next to the **MODE** button on the gamepad must be **OFF** to use the Left Stick for control. If that LED light is on, press the MODE button again to toggle it OFF.

    c. **Right Throttle RT** for throttle - pressed all the way represents 100% command. Let the throttle go for 0% command.

    Note: Throttle is scaled by 20% for better manual control then saturated to 20% in the **basicQCarIO** subsystem for safety.

    d. **Button A** for reverse - hold this button and use the steering/throttle commands to drive backwards.

NOTE: The switch at the back of the F710 gamepad must be in the **X** position for the above-mentioned control to work. If the switch is in the **D** position, move the switch

back to the **X** position.

2. The LEDs are in the following states

    a. **Headlamps** are always on. The reverse indicators (white) are on in Reverse.
    b. **Brake lamps** are on when the absolute speed of the vehicle is decreasing.
    c. **Left/right indicators** turn on when the corresponding steering is over a threshold.

Python

The application will also calculate the car's speed from encoder counts. The process is shown in Figure 3.

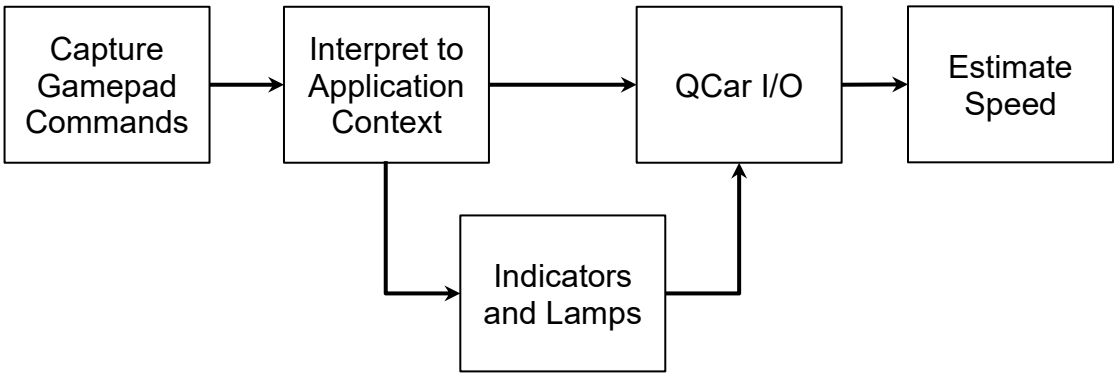

Figure 3. Component diagram

Running the example

1. Check User Manual – Software Python for details on deploying python scripts to the QCar as applications.
2. Once you have set the controller number correctly for the gamepad (see Python Hardware Test documentation for details), run the script using a **sudo** flag. There are two ways to drive the car. If users set the configuration to **3** in the script, the Gamepad's **right stick** longitudinal axis is used for the throttle, the **left stick** lateral axis is used for steering, and the **LB** button is used to arm the QCar. If users set the configuration to **4** in the script, the **right trigger RT** is used to provide positive throttle in the forward/reverse directions based on the state of the **button A**.

Note: If the steering does not appear to work, please ensure the **mode** light on the gamepad is **off**.

Guide

1. Encoder counts to linear speed

   The **q_interpretation** module contains the method **basic_speed_estimation** to convert from encoder speed (counts/s) to linear speed (m/s) based on the differential and spur parameters of the QCar 2. However, the HIL I/O provides encoder counts, which must first be differentiated. This is accomplished by using the **differentiator_variable** method within the **Calculus** class of the **q_misc** library provided. Initialized with the **sampleTime**, you can set the actual step size within the main loop when calling the differentiator, accounting for variable sample time.

   Note: Do not forget to initialize the differentiator using the **next** method!

```
# Set up a differentiator to get encoderSpeed from encoderCounts
diff = Calculus().differentiator_variable(sampleTime)
_ = next(diff)
timeStep = sampleTime

# Inside main while loop
   encoderSpeed = diff.send((encoderCounts, timeStep))
```

2. Performance considerations

   We run the example at 50 Hz. The **os** module is used here to clear the terminal screen whenever new gamepad updates are received. Although this is expensive and is not the best thing to do, we account for the slower sample rates by using the variable sample time differentiator instead of a static one. Without accounting for the variable sample times, the differentiator will underestimate sample times and thereby overestimate the speed. Comment out the system screen clear as well as the print statement (similar to the snippet below) to improve performance up to 500 Hz.

```
if new:
    os.system('clear')
    print(...)
```