Lab Guide
Manual Drive

Content Description

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

MATLAB

In this example, we will capture commands from the keyboard 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.
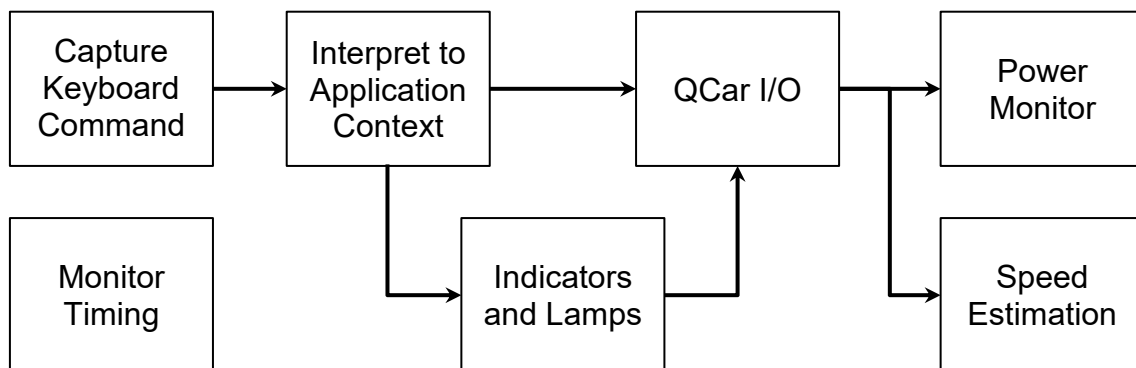

Figure 1. Component diagram

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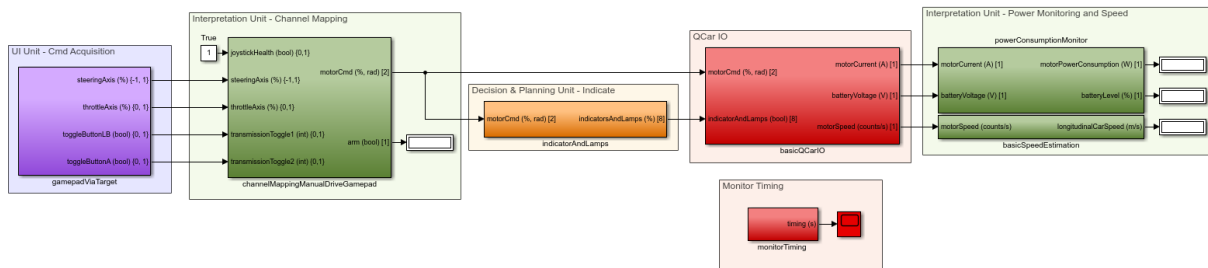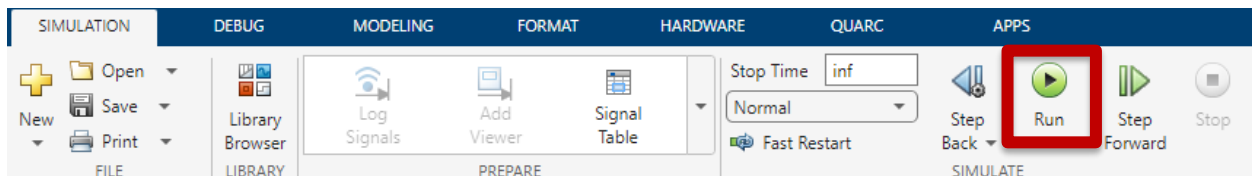
Figure 2. Simulink implementation of Manual Drive

Running the example

To run examples for virtual QCar please go to the **SIMULATION** tab in the ribbon interface and click on the Run icon.

Guide

1. Driving manually is mapped to the following keys:

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

   b. **Left/Right Arrow Key** for steering.

   c. **Up/Down Arrow Key** for throttle. Throttle is scaled by 20% for better manual control then saturated to 20% in the **basicQCarIO** subsystem for safety.

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.
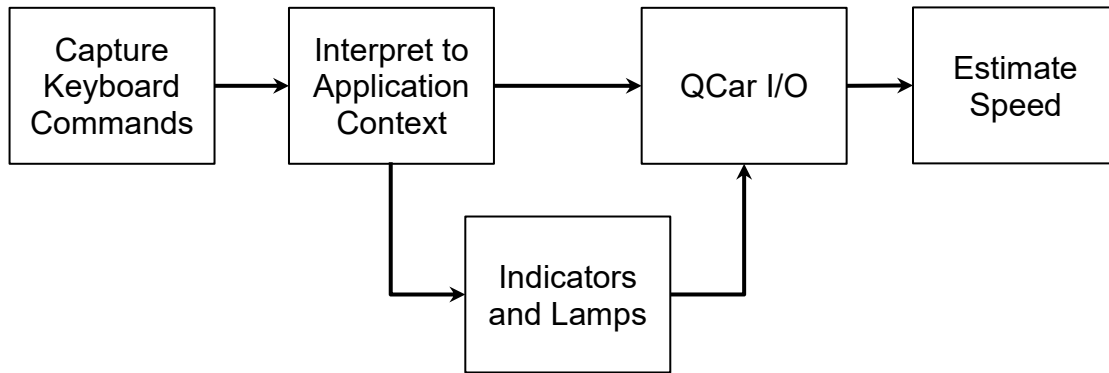

Figure 3. Component diagram

Running the example

There are two ways to drive the car. In both configurations, the QCar is armed with the **Space Bar**. If users set the configuration to **0** in the script, the QCar can be driven with "**WASD**" **keys**. If users set the configuration to **1** in the script, the QCar can be driven with **Arrow keys**. After configuration, execute the script.

Details

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!

    ```python
    # 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 each loop. 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.

```
os.system('clear')
print(...)
```