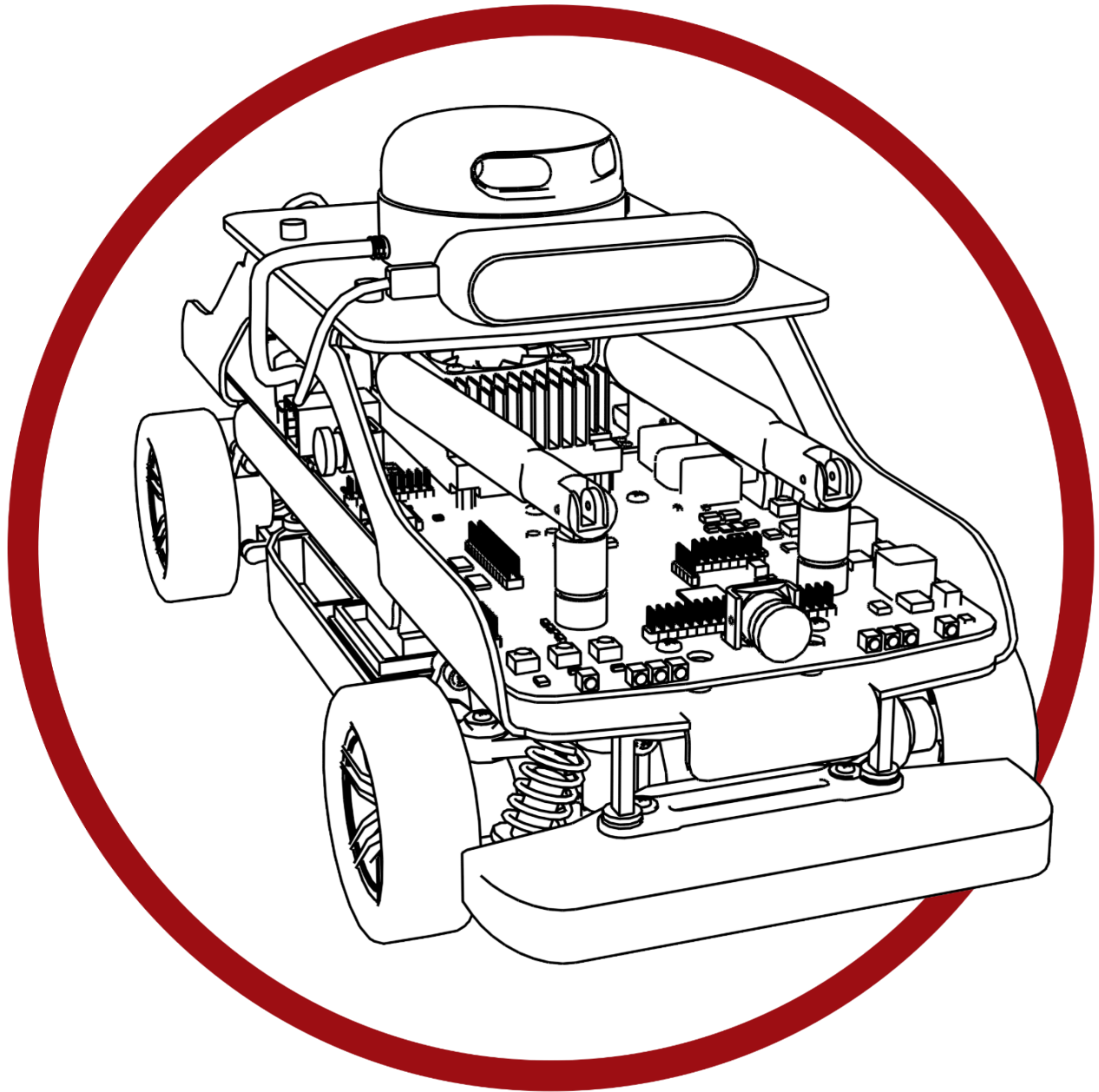


Self-Driving Car Research Studio



User Guide – Software - Python

V 1.1 (November 2020)



Caution

This equipment is designed to be used for educational and research purposes and is not intended for use by the general public. The user is responsible to ensure that the equipment will be used by technically qualified personnel only.

Table of Contents

A. Overview	3
B. Development Details	4
Quanser Modules	4
Quanser Core	4
C. Configure Timing	6
D. Deployment and Monitoring	7
E. Troubleshooting Best Practice	8

A. Overview

The overall process is described in Figure 1 below. Design your application as you see fit for Python 3. The examples provided are tested with **Python 3.7.5** for the Ground Control Station (GCS) and **Python 3.6.9** on the QCar. You can then transfer the application to the embedded target or run it on your local development machine.

For more details, see the corresponding section below.

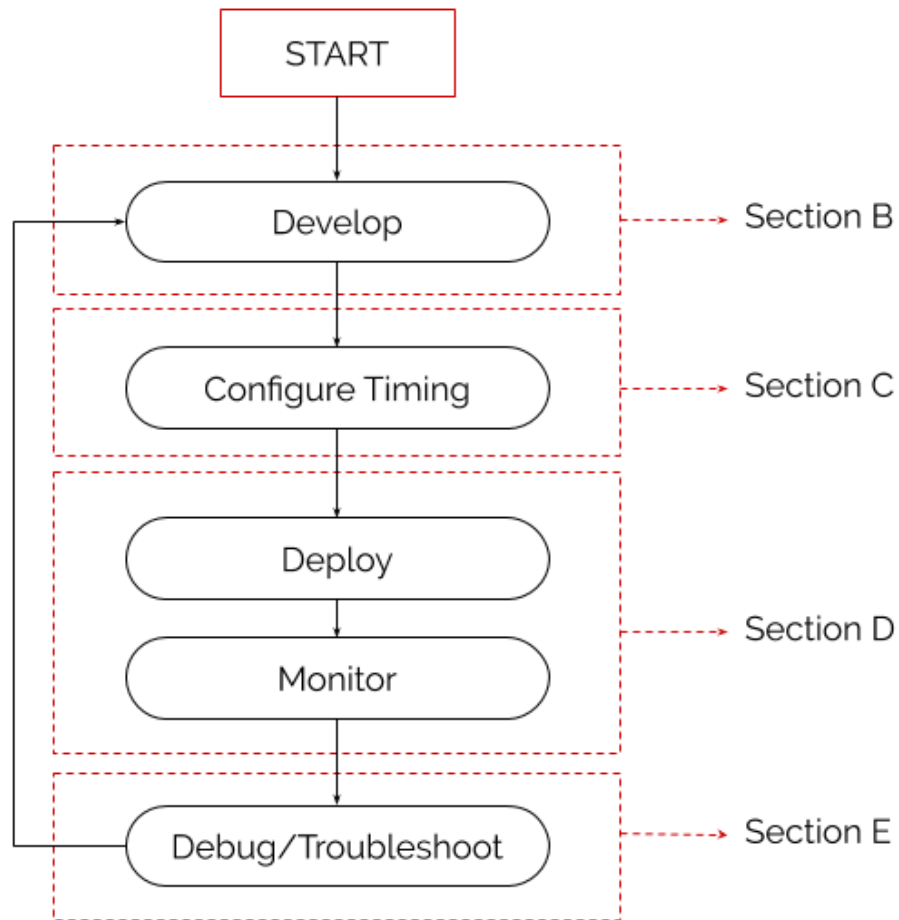


Figure 1. Process diagram for Python code deployment

B. Development Details

Ensure that all the modules required by your application are installed in the location where the script will be deployed. The GCS provided with the **Self-Driving Car Research Studio** comes equipped with numerous modules already installed, and so does the QCar platform. On the **GCS**, use the following command in a command prompt to see what packages are available.

```
C:\...\> python -m pip list
```

Note that the GCS only has **python 3.7.5** installed, and the **python** command defaults to this installation. The QCar has both python 2 and python 3 installed, and thus, **python3** must be used instead. In a terminal on the **QCar** platform (direct connection or PuTTY terminal when remote) use the following,

```
nvidia@qcar-*****:~$ python3 -m pip list
```

Quanser Modules

Both the GCS and the QCar also have Quanser modules installed that are used for additional interactions with hardware. These packages are,

quanser-common	2020.7.9
quanser-communications	2020.7.9
quanser-devices	2020.7.9
quanser-hardware	2020.7.9
quanser-multimedia	2020.7.9

The provided **Hardware Tests** and **Applications** use these packages. The package **quanser-communications** can be used for code development for the Stream API and generation communications. The package **quanser-hardware** is used for all HIL (hardware-in-the-loop) API related development. You can use **quanser-multimedia** to read most 2D and 3D cameras, and **quanser-devices** allows support for reading the LIDAR and writing to the LCD.

Quanser Core

In addition to the **Hardware Tests** and **Application** examples, the **Self-Driving Car Research Studio** comes with higher-level python libraries, equipped with a list of python functions commonly used throughout the provided examples. These are broken down into 7 functional groups - Essential, Control, Interpretation, User Interface, Decision & Planning, Miscellaneous and product_QCar, which are summarized in Table 1.

Usage: On the QCar, set up a working directory. Drop the **Quanser** folder under the **Core** section of the provided package in your working directory, in addition to any examples you need to run. An example folder structure is shown in Figure 2.

Functional Units	Description
User Interface (UI)	This consists of methods to read the joystick or gamepad on the host (GCS) or QCar platform.
Essential	This module contains higher-level classes to simplify 2D and 3D camera acquisition, as well as LIDAR data acquisition. These classes are generic and can be used for any 2D webcam. The 3D webcam classes have been tested with the Intel RealSense camera set on both the host (GCS) as well as the QCar platform. The LIDAR class has been tested with the RP-LIDAR-A2 device.
Interpretation	This includes classes, methods or functions that interpret sensor data according to the application at hand. Analyzing images to detect obstacles or road signs, detecting lanes and their location, etc. all belong here.
Control	This serves as the classic control group of methods. Cruise control, steering-based speed reduction, etc. get included here.
Decision & Planning	These modules include the higher-level decision making that is involved with autonomous or intelligent systems.
Miscellaneous	This includes other support methods that act as a glue for the other units. You can find classes for signal generators, filters, simplified stream, variable and fixed step differentiator/integrators and other utilities here.
Product_QCar	This includes numerous I/O methods commonly used for the QCar, Motor commands and LED values written to the device, and measurements such as IMU, motor current, and battery voltage are included here.

Table 1. Quanser higher-level modules











/home/nvidia/Documents/Python/				
Name	Size	Changed	Rights	Owner
		2020-03-27 12:00:40 PM	rw-r-xr-x	nvidia
 Quanser		2020-03-27 12:00:44 PM	rw-rwxr-x	nvidia
 Hardware_Test_Basic_IO.py	3 KB	2020-03-25 9:55:18 AM	rw-rw-r--	nvidia
 Hardware_Test_CSI_Camera_Single.py	3 KB	2020-04-20 3:35:35 PM	rw-rw-r--	nvidia
 Hardware_Test_CSI_Cameras.py	3 KB	2020-05-06 2:18:39 PM	rw-rw-r--	nvidia
 Hardware_Test_Gamepad.py	2 KB	2020-05-14 1:09:07 PM	rw-rw-r--	nvidia
 Hardware_Test_IntelRealsense.py	3 KB	2020-04-20 3:33:16 PM	rw-rw-r--	nvidia
 Hardware_Test_RP_LIDAR_A2.py	3 KB	2020-04-22 10:33:48 AM	rw-rw-r--	nvidia
 Task_Manual_Drive.py	4 KB	2020-05-15 10:52:43 AM	rw-rw-r--	nvidia
 Task_Point_Cloud.py	3 KB	2020-05-11 11:18:33 AM	rw-rw-r--	nvidia

Figure 2. Example folder structure on the QCar

C. Configure Timing

It is important to maintain a consistent sample rate for real-time applications. Given a sample time, all code in a single iteration must be executed in a time window that is less than the required sample time. In cases where the execution of an iteration is completed in less than the sample time, it is also essential that the next iteration not begin until a full unit of the sample time has elapsed.

For example, consider an image analysis task that must be executed at 60 Hz, corresponding to a '**sample time**' of 16.7 ms ($1/60$). If the time taken to execute the analysis code, also referred to as the '**computation time**', is less than the sample time, say 10 ms, then it is important to wait an additional 6.7ms at each time step before proceeding to the next iteration. On the other hand, if the computation time is greater than the sample time, say 20ms, then the sample time cannot be met. In such cases it may be essential to lower the sample rate or increase the sample time, to say 40Hz or 25 ms. Note that the **time** module's **time()** method returns the current hardware clock's timestamp in seconds.

In Python, the code is executed as fast as possible, and a wait can be inserted using the **time** module's **sleep()** method or the **opencv** module's **waitkey()** method for imaging applications. The following snippet provides a detailed example on how to accomplish this.

```
import time

# Define the timestamp of the hardware clock at this instant
startTime = time.time()

# Define a method that returns the time elapsed since startTime was defined
def elapsed_time():
    return time.time() - startTime

# Define sample time starting from the rate
sampleRate = 100 # Hertz
sampleTime = 1/sampleRate # Seconds

# Total time to execute this application in seconds.
simulationTime = 5.0

# Refresh the startTime to ensure you start counting just before the main loop
startTime = time.time()

# Execute main loop until the elapsed_time has crossed simulationTime
while elapsed_time < simulationTime:
    # Measured the current timestamp
    start = elapsed_time()

    # All your code goes here ...

    # Measure the last timestamp
    end = elapsed_time()

    # Calculate the computation time of your code per iteration
    computationTime = end - start

    # If the computationTime is greater than or equal
    # to sampleTime, proceed onto next step
    if computationTime < sampleTime:
        # sleep for the remaining time left in this iteration
        time.sleep(sampleTime - computationTime)
```

D. Deployment and Monitoring

1. When done with python code development, and deploying the application on the development platform (GCS or direct connection with QCar), proceed to Step 3.

Note: If your code requires access to onboard hardware (such as LIDAR or camera), you must deploy the code to the QCar for execution (see Step 2)..

2. To deploy applications to the QCar while developing on a different machine, you will need to use **WinSCP** or **USB** to transfer your files and may additionally need **VcXsrv** to view the output. See the **III - Connectivity User Guide** for more information on this.

3. Run your application using the terminal command:

```
>> python test.py
```

Note: If using any Hardware-in-the-loop (HIL) methods or using the LIDAR, you will also need the sudo flag :

```
>> sudo python test.py
```

Note: If running on the QCar, you must specify **python3** for python 3.x.x

E. Troubleshooting Best Practice

In order to ease debugging during application development, we use the **try/except/finally** structure to catch exceptions that otherwise terminate the application unexpectedly. Most of our methods in the Quanser library have this structure built in. After configuration and initialization, scripts begin with **try**. If an unexpected error arises, it will be captured by the **except** section instead. This can ensure that code in the **finally** section still gets executed and the application ends gracefully. For example, if you specify an incorrect channel number for HIL I/O, a **HILError** will be raised. However, you still want to call the **terminate()** method to close access to the HIL board, without which, opening it on the next script call may fail.

```
## Main Loop
try:
    while elapsed_time() < simulationTime:
        # Start timing this iteration
        start = time.time()

        # Basic IO - write motor commands
        mtr_cmd = np.array([ 0.2*np.sin(elapsed_time()*2*np.pi/5),
                             -0.5*np.sin(elapsed_time()*2*np.pi/5)])
        LEDs = np.array([0, 1, 0, 1, 0, 1, 0, 1])

        current, batteryVoltage, encoderCounts = myCar.read_write_std(mtr_cmd, LEDs)

        # End timing this iteration
        end = time.time()

        # Calculate computation time, and the time that the thread should pause/sleep for
        computation_time = end - start
        sleep_time = sampleTime - computation_time%sampleTime

        # Pause/sleep and print out the current timestamp
        time.sleep(sleep_time)
        print('Simulation Timestamp :', elapsed_time(), ' s, battery is at :', 100 - (12.6 -
                                                                                   batteryVoltage)*100/(2.1), ' %.')

        counter += 1

except KeyboardInterrupt:
    print("User interrupted!")

finally:
    myCar.terminate()
```

© Quanser Inc., All rights reserved.



Solutions for teaching and research. Made in Canada.