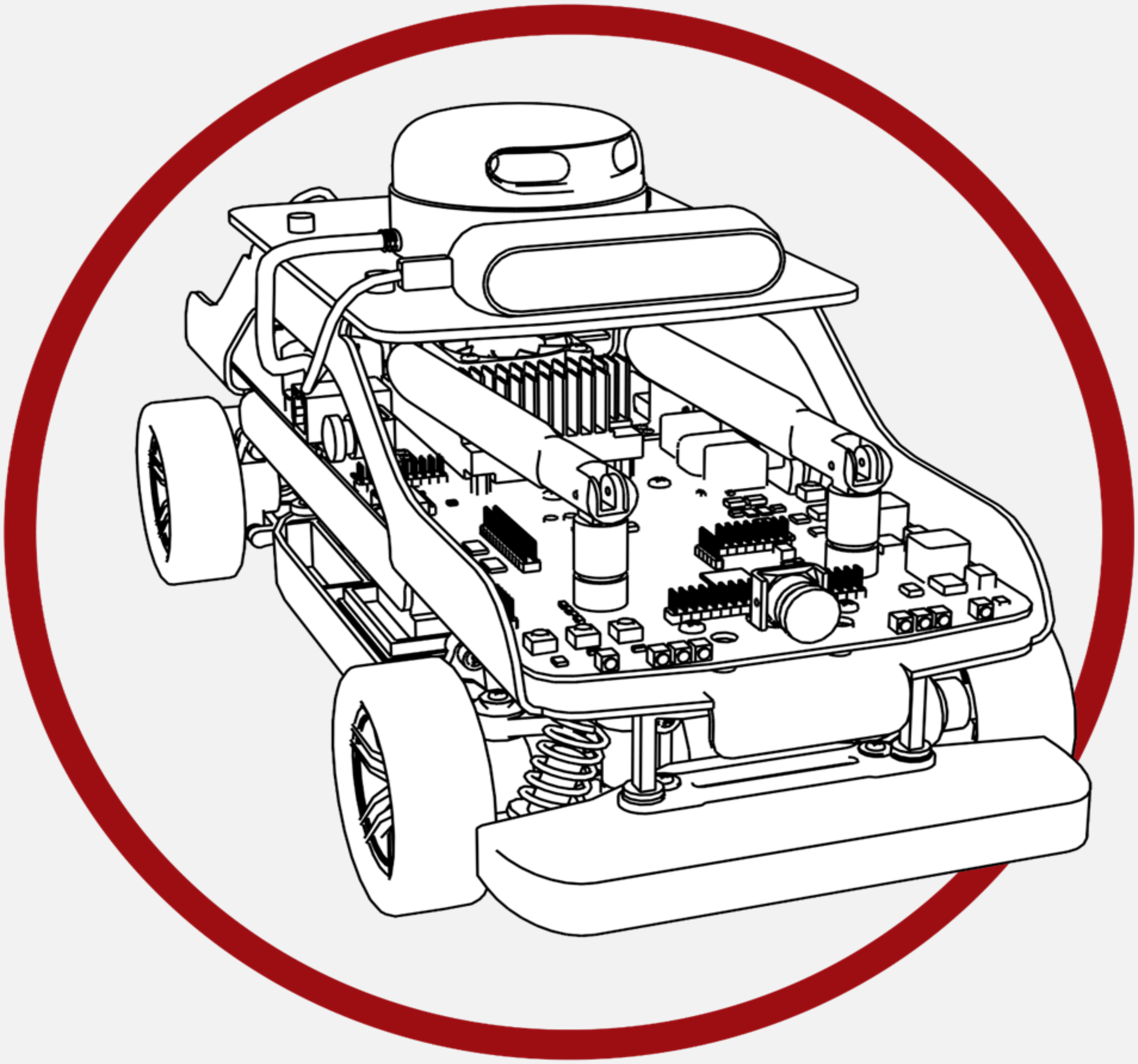

QCar

Software Python User Manual

v 1.0 — 9*th Jan* 2023

For more information on the solutions Quanser Inc. offers, please visit the web site at: http://www.quanser.com

Quanser Inc.
119 Spy Court
Markham, Ontario
L3R 5H6, Canada

info@quanser.com
Phone : 19059403575
Fax : 19059403576
printed in Markham, Ontario.

FCC Notice This device complies with Part 15 of the FCC rules. Operation is subject to the following two conditions: (1) this device may not cause harmful interference, and (2) this device must accept any interference received, including interference that may cause undesired operation.

Industry Canada Notice This Class A digital apparatus complies with Canadian ICES-003. Cet appareil numérique de la classe A est conforme à la norme NMB-003 du Canada.

Waste Electrical and Electronic Equipment (WEEE)

This symbol indicates that waste products must be disposed of separately from municipal household waste, according to Directive 2002/96/EC of the European Parliament and the Council on waste electrical and electronic equipment (WEEE). All products at the end of their life cycle must be sent to a WEEE collection and recycling center. Proper WEEE disposal reduces the environmental impact and the risk to human health due to potentially hazardous substances used in such equipment. Your cooperation in proper WEEE disposal will contribute to the effective usage of natural resources.

CE Compliance CE

This product meets the essential requirements of applicable European Directives as follows:

- 2006/95/EC; Low-Voltage Directive (safety)
- 2004/108/EC; Electromagnetic Compatibility Directive (EMC)

Warning: This is a Class A product. In a domestic environment this product may cause radio interference, in which case the user may be required to take adequate measures.

Caution

This equipment is designed to be used for educational and research purposes and is not intended for use by the public. The user is responsible to ensure that the equipment will be used by technically qualified personnel only. While the end-effector board provides connections for external user devices, users are responsible for certifying any modifications or additions they make to the default configuration.

Table of Contents

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.

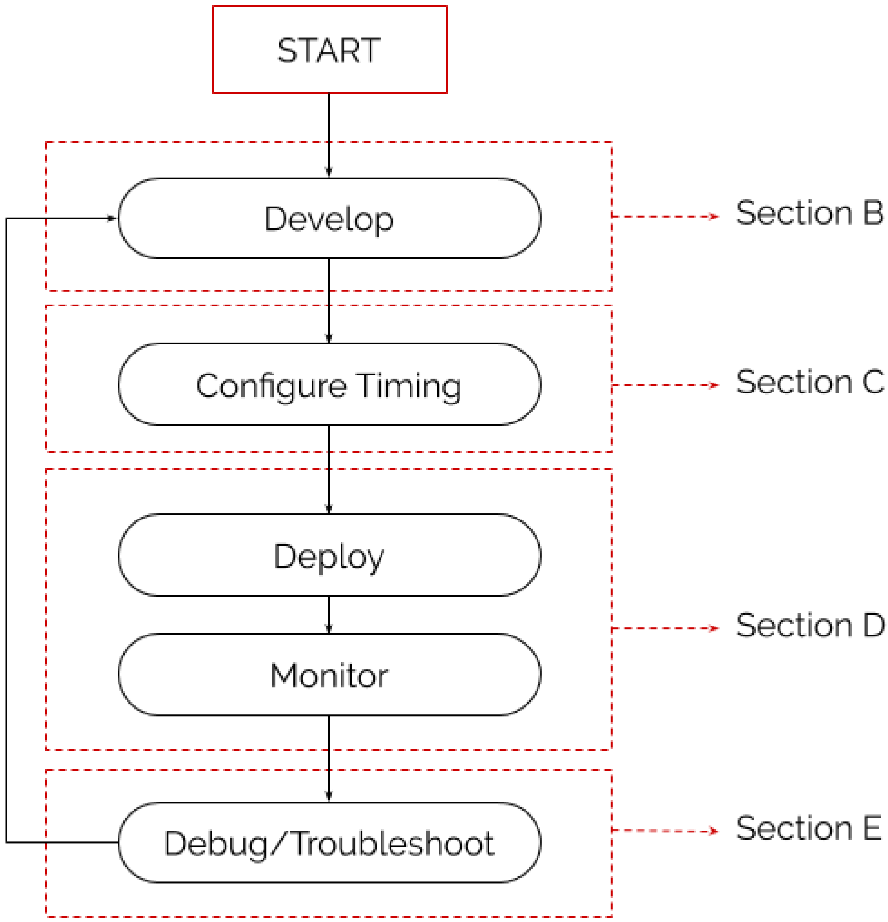

Figure 1. Process diagram for Python code deployment

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 Studio (SDCS)** comes equipped with numerous modules already installed, and so does the QCar platform. On the Ground Control Station (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-api	2021.3.29
quanser-common	2021.3.29
quanser-communications	2021.3.29
quanser-devices	2021.3.29
quanser-hardware	2021.3.29
quanser-multimedia	2021.3.29

To view how to update the packages preinstalled on the QCar please review User Manual – Customizing the QCar section D for software packages. The package **quanser-communications** can be used for code development which requires the use of a stream communication. 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. These are low level python libraries which are the building blocks of the Python Application Libraries covered later in this document.

If the GCS has an updated version of QUARC run the following command to update the python packages on the GCS:

```
cd “C:\Program files/Quanser\QUARC\python”
dir
```

Typing `dir` will indicate the date needed for the next command:

```
python -m pip install --upgrade --find-links . quanser_api-<date>-py2.py3-none-any.whl
```

where `<date>` is the date for the API being installed. For example:

```
sudo python3 -m pip3 install --upgrade --find-links . quanser_api-2021.4.1-py2.py3-none-any.whl
```

The terminal window should indicate that all existing packages were successfully uninstalled, then the new packages were installed.

High-Level Application Libraries (hal)

In addition to the **Hardware Tests** and **Application** examples, the **SDCS Studio** comes with higher-level python libraries, equipped with a list of python functions commonly used throughout the provided examples. Depending on the implementation there are two core folders inside of hal:

- products
- utilities

Products folder contains a class for QCar called **qcar.py** which is a specific implementation of the generic classes found in the **utilities** folder. The high-level applications for the QCar include:

- QCarEKF
- QCarGeometry

Utilities folder contains generic high-level application which can be used in any application, specific classes include:

- Control
- Estimation
- Geometry
- Image Processing
- Mapping

Python Application Libraries (pal)

The **SDCS Studio** also comes with a series of python application libraries which make use of the Quanser Modules. These are intended to give users the ability to interface with the hardware on the QCar. Just like in hal, pal also contains two folders:

- products
- utilities

Products folder contains a class for QCar called **qcar.py** which is a specific implementation of the generic classes found in the **utilities** folder. **qcar.py** was designed to give users the ability to interact with the standard set of sensors in the qcar from a single location. The classes found in **qcar.py** include:

- QCar
- QCarCameras
- QCarLidar
- QCarRealSense
- QCarGPS

If the sensor/peripheral is not defined in **qcar.py** the standard utilities can be used for added flexibility, these include:

- gamepad
- lidar
- math
- scope
- stream
- vision

Application Modules Setup

To make use of **hal** and **pal** (or include future updates) on the QCar:

1. Make a common directory on the QCar for Quanser resources where files will be transferred to and from. We recommend creating a folder called Quanser under Documents as the common directory (Figure 3). You may use a keyboard/mouse to connect to the QCar or see the User Manual - Connectivity for how to remotely connect to the QCar.
2. Copy over the resources inside the *src* in the Quanser Application Libraries over to your new Quanser folder in the QCar.

Name	Date modified	Type	Size
src	10/12/2023 1:32 PM	File folder	
installer.py	10/12/2023 1:32 PM	Python Source File	12 KB
python-3.11.4-amd64.exe	9/14/2023 8:56 AM	Application	24,831 KB
README.txt	10/12/2023 1:32 PM	Text Document	4 KB
setup.bat	9/14/2023 8:56 AM	Windows Batch File	1 KB

Figure 2. Example folder structure on QCar

3. Include the following lines at the end of the ~/.bashrc file on the QCar.

```
export PYTHONPATH=‘‘<PATH TO hal and pal>’’
export QAL_DIR=‘‘<PATH TO Quanser Resources>’’
```

As an example, if the contents of the *src* folder were copied over to the Quanser folder, the environment variable on your qcar should look like:

```
export PYTHONPATH=‘‘/home/nvidia/Documents/Quanser/libraries/python”
export QAL_DIR=‘‘/home/nvidia/Documents/Quanser”
```

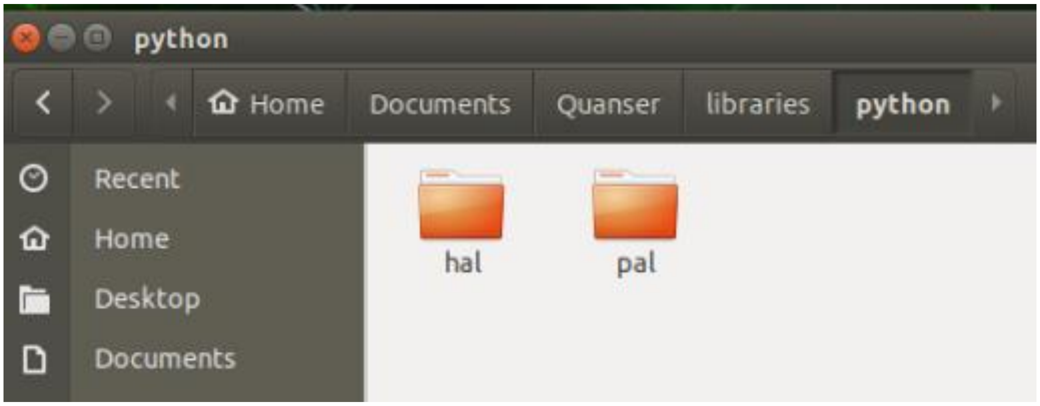

Figure 3. Example folder structure on QCar

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 does 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
```

```python
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

When **developing** code for the QCar there are two main methods:

- Using a Ground Control Station (GCS) and downloading code
- Writing code directly on the QCar.

To **run** python code on the QCar the two methods stated above are also valid.

When ready to run python code:

1. The QCar has both python2 and python3 installed. All the examples available will require the use of **python3**.
2. To run an application, use the following syntax:

```
sudo PYTHONPATH=$PYTHONPATH python3 <application name>.py
```

   As an example, to run a QCar hardware test:

```
sudo PYTHONPATH=$PYTHONPATH python3 hardware_test_basic_io.py
```

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()
```

Solutions for teaching and research. Made in Canada.