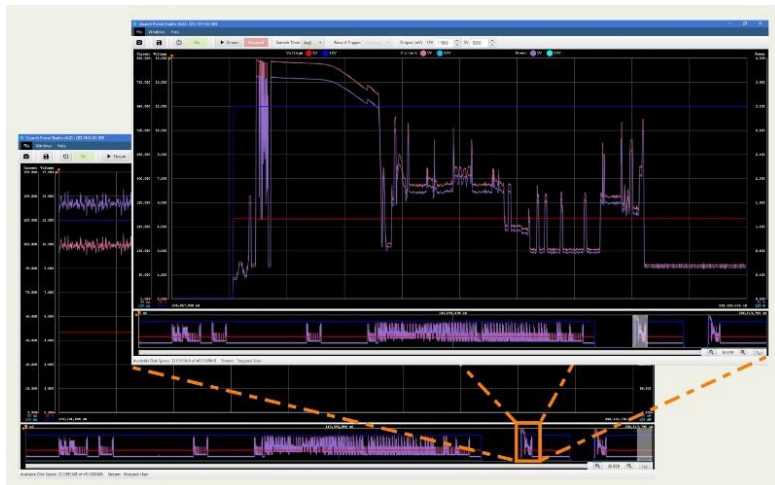


Quarch Technology Ltd

Quarch Automation Manual

Thursday, 13 June 2024



Change History

1.0	May 2019	Initial Release
1.1	August 2019	Added config_files and calibration module
1.2	August 2019	Added quarchpy run package
1.3	October 2020	Stats and snapshot for QPS added
1.4	November 2020	Save CSV command added
1.5	Feb 2024	Added new QPS commands and multi-rate

Table of Contents

Change History	2
Table of Contents	3
Introduction	5
Requirements	7
Installation	8
Python.....	8
QuarchPy Package	8
Linux USB Permissions	10
Basic Usage	11
Quarchpy Control	13
Module: device.....	13
Module: iometer	17
Module: fio	18
Module: config_files	19
Module: calibration.....	20
Module: Run.....	21
Power studio commands.....	22
QIS Control	24
Sending a command	24
Basic command set.....	24
Debug commands.....	26
Streaming commands	27
Real-time resampling	30
Power Studio Control	31
Sending a command	31
Basic command set.....	Error! Bookmark not defined.

Streaming commands**Error! Bookmark not defined.**

User data commands**Error! Bookmark not defined.**

Display commands.....**Error! Bookmark not defined.**

Introduction

Quarch provides tools and software which is simple to automate from external sources.

This manual provides an overview of the software automation options and is the main source of documentation for the APIs.

Only the software APIs are documented here, for full details of the commands that can be sent to individual hardware modules, you will still have to look at the relevant technical manual. Download these from the product page on our website:

<https://quarch.com/product-search>

Quarch modules can be easily controlled using Python through multiple interface options:

- USB
- Serial
- REST
- Telnet

QuarchPy is a python package designed to provide an easy to use API which will work seamlessly over any of the communication options.

The package contains all prerequisites needed as a single PyPI project which can be installed from the python online repository or a downloaded local copy.

This makes it easy to install and keep up to date, while also providing you with full access to the source code if you want to make changes or additions.

The package also includes several additional sections

- QIS (Quarch Instrument Server)

This is a headless Java application which handles all the data and multi-thread intensive work needed to stream high-speed data from Quarch Power Modules.

While it is possible to control power modules with pure Python code, the efficiency is much poorer and streaming speeds will be limited.

QIS is automatically started when a 'QIS' type connection is requested.

■ QPS (Quarch Power Studio)

This is a Java based application for control of the Quarch Power Modules. Also available for download from our site, a version is supplied here as QPS can be automated from Python and controlled via script.

If you request a QPS connection, the application can be automatically launched, and you can see the live power data while the automatic capture is running. This provides a great way to get an overview on the results while running a fully automated test.

Requirements

- Windows, Linux or OSX
- Python 2.x or Python 3.x

We strongly recommend using Python 3.x. Our more recent application notes require this, though we try to keep compatibility where possible

- Quarch USB driver (Windows only, if using USB comms mode)

Installation

Python

If Python is not installed on your system, go to <https://www.python.org/downloads/> and choose the relevant version for your system.

Under Windows it is sensible to ensure the Python installation directory and PythonXX\Scripts are included in the PATH environment variable. See [HERE](#) for instructions on how to do this.

QuarchPy Package

The Quarch Python package can be installed from the Python web repository (assuming you have internet access) or via the download from our site which contained this document.

Web Install

From the command line:

```
>pip install quarchpy
```

If this fails, your path to “pip” may not be set, you can instead run:

```
>python -m pip install quarchpy
```


Local Install

If you want to install from a downloaded .whl file, download the file from <https://pypi.org/manage/project/quarchpy/releases/>, navigate to the folder containing the quarchpy-x.y.z-py2.py3-none-any.whl file and run:

```
>pip install quarchpy-x.y.z-py2.py3-none-any.whl
```

If this fails, your path to 'pip' may not be set, you can instead run:

```
>python -m pip install quarchpy-x.y.z-py2.py3-none-any.whl
```

Upgrade

If you already have QuarchPy installed, you will get a failure message. If you want to upgrade to a new version, you need to add the '--upgrade' command:

```
>pip install --upgrade quarchpy
```

The --upgrade command can similarly be used in any of the other examples, to load from a local install folder.

Linux USB Permissions

Linux systems require administrative rights to run python scripts for modules connected via USB. You can do that by running your script as root (sudo command) or changing the default USB permissions. This is done by creating a text file called **Quarch-permissions-usb.rules**

For ubuntu systems, you need to enter into that file:

```
SUBSYSTEM == "usb", ATTRS{idVendor}=="16d0", MODE="0666"
```

```
SUBSYSTEM == "usb_device", ATTRS{idVendor}=="16d0", MODE="0666"
```

For Centos systems, you need:

```
SUBSYSTEM == "usb", ATTRS{idVendor}=="16d0", GROUP="users",  
MODE="0666"
```

```
SUBSYSTEM == "usb_device", ATTRS{idVendor}=="16d0", GROUP="users",  
MODE="0666"
```

This file needs to be placed in /etc/udev/rules.d

Finally, the system either needs to be restarted or run the command:

```
>sudo udevadm control -reload
```

Then reconnect the USB device.

Basic Usage

We strongly recommend you download a relevant application note to demonstrate the use of the Python API, only the basics are shown here.

Below is a simple example to import and use a Quarch module.

```
# Imports device control from the Quarchpy package
from quarchpy.device import *

# Scan for quarch devices on the system (across all interfaces)
deviceList = scanDevices ('all')
# Use the user selection function to display the list and choose one
moduleStr = userSelectDevice (deviceList)
# Create the module, using the returned connection string
myDevice = quarchDevice(moduleStr)
```

You can also specify a device directly. For USB connections, you specify a full, or partial serial number, as found on the product label.

For LAN connections (ReST and Telnet) you can specify either an IP address, or NetBIOS name (if supported by your network). The default NetBIOS name for most modules is the serial number:

```
myDevice = quarchDevice ("telnet:192.168.1.55")
```

Or

```
myDevice = quarchDevice ("telnet:QTL1079-03-137")
```

For Serial connections, specify the COM port on windows

```
myDevice = quarchDevice ("SERIAL:COM4")
```

Or the linux serial port ID

```
myDevice = quarchDevice ("SERIAL:ttyS0") or you may need
myDevice = quarchDevice ("SERIAL:/dev/ttyS0")
```

You can now send commands to the module. The full command list for each module can be found in its Technical Manual, which can be downloaded from their product pages:

<https://quarch.com/product-search>

To send the simple 'hello?' command, use:

```
myDevice.sendCommand ("hello?")
```

This function will return the command response, so you can easily see the result with:

```
print (myDevice.SendCommand("hello?"))
```

Finally you must close the connection with

```
myDevice.closeConnection()
```

Quarchpy Control

Module: device

This module provides the basic connection mechanism to send commands to devices.

> from quarchpy.device import *

Device Classes	Purpose
quarchDevice	Base device type, for control of any Quarch device
quarchArray	Extended class with for Array Controllers. Adds functions to scan for and handle sub-devices.
subDevice	Support class to control sub devices of quarchArray, as if they are directly connected as a standard quarchDevice
quarchPPM	Extended class with for Power Module, adds streaming commands

Additional Classes	Purpose
quarchQPS	Class for automation of Quarch Power Studio

Helper Functions	Purpose
scanDevices	Scan for Quarch devices, both locally and over LAN
listDevices	Display the devices found
userSelectDevice	Ask the user to select a Quarch device from a list

Controlling devices

The base device class is used for all initial connections

```
quarchDevice(connectionString, [ConType="PY"],[timeout="5"])
```

PY – (default) direct python connection to the module, this is the option to use in most cases

QIS – Quarch Instrument Server, connects to a module via QIS, this is used to gain the additional ‘streaming’ features from power modules when you want to record large amounts of power data via a script.

QPS – Quarch Power Studio, connects to a module via QPS, this is used to automate a power module with the additional features of Power Studio. This allows graphical analysis of power traces, custom user channels, annotations and more.

Connection strings

‘**connectionString**’ is in the form **INTERFACE:PATH**

The interface types available depend on the device you are using, and how you have cabled it

INTERFACE = [USB, SERIAL, REST, TCP, TELNET]

PATH is the unique path to the device, examples:

```
# Connect to the first QTL1743 on found on USB
```

```
myDevice = quarchDevice("USB:QTL1743")
```

```
# Connect to a specific QTL1743 on USB
```

```
myDevice = quarchDevice("USB:QTL1743-02-015")
```

```
# Connect to the device on COM4
```

```
myDevice = quarchDevice("SERIAL:COM4")
```

```
# Connect to an IP address, using Telnet
```

```
myDevice = quarchDevice("TELNET:192.168.1.16")
```

```
# Connect to an IP address, using TCP
```

```
myDevice = quarchDevice("TCP:192.168.1.16")
```

```
# Connect to an IP address, using TCP (or other LAN option)
```

```
# Default netBIOS names are the device serial number
```

```
myDevice = quarchDevice("TCP:QTL1079-03-010")
```

Example – Standard Device

```
# Connect to a module and send a command
myDevice = quarchDevice("USB:QTL1743")
myDevice.sendCommand("hello?")
```

Example – Power module via QIS

```
# Connect to a module
myDevice = quarchDevice("USB:QTL1999", ConType = "QIS")
# Convert the base device to a 'Programmable Power Module' device
myPowerDevice = quarchPPM (myDevice)
myPowerDevice.sendCommand("hello?")
# Stream power measurement data to file
myPowerDevice.sendCommand ("record:trigger:mode manual")
myPowerDevice.startStream("Stream1.txt", 2000, "My example stream")
```

Example – Power module via QPS

```
# Connect to a module
myDevice = quarchDevice("USB:QTL1999", ConType = "QPS")
# Convert the base device to a 'Power Studio' controller and open it
myQpsDevice = quarchQPS(myDevice)
myQpsDevice.openConnection()
# Send a command, as with the other connection methods
myQpsDevice.sendCommand ("record:averaging 32k")
```

Example – Array controller

```
# Connect to an Array (over USB in this example)
myDevice = quarchDevice("USB:QTL1079")

# Extend this with the quarchArray class
myArray = quarchArray(myDevice)

# Get access to the device on port number 2
myModuleTwo = myArray.getSubDevice(2)
myModuleTwo.sendCommand ("hello?")
```

More information

Relevant application notes
<p data-bbox="204 448 1088 486">https://quarch.com/file/an-006-python-control-quarch-modules</p> <p data-bbox="204 510 817 548">Basic control examples for Quarch devices</p>
<p data-bbox="204 607 1142 645">https://quarch.com/file/AN012-QIS-python-control-power-modules</p> <p data-bbox="204 669 1101 707">Automating power modules, streaming power data to CSV files</p>

Module: iometer

This module provides automation of iometer for workload generation. Iometer must be available on the host system. These functions assist the user in running one or more workloads and getting results in real-time, for display in Power Studio or similar.

These functions are used in application note AN-016, and we recommend you work from here if you wish to use these features in your own code.

```
> from quarchpy.iometer import *
```

Functions	Purpose
runIOMeter	Executes iometer as a subprocess
processIometerInstResults	Parses the real time results file created by iometer
readIcfCsvLineData	Reads in data from a CSV file describing workloads to run
generateIcfFromCsvLineData	Generates an iometer required .icf file from CSV file parameters
generateIcfFromConf	Generates an iometer required .icf file from a Quarch supplied .conf template format file

More information

Relevant application notes
https://quarch.com/file/AN-016-qps-automation-iometer Iometer automation with Power Studio, displaying IOPS Vs power performance

Module: fio

This module provides automation of FIO for workload generation. FIO must be available on the host system. These functions assist the user in running one or more workloads and getting results in real-time, for display in Power Studio or similar.

These functions are used in application note AN-017, and we recommend you work from here if you wish to use these features in your own code.

```
> from quarchpy.fio import *
```

Functions	Purpose
runFIO	Executes fio as a subprocess. Provided callback functions give access to the real-time data

More information

Relevant application notes
https://quarch.com/file/an017-gps-performance-test-fio FIO automation with Power Studio, displaying IOPS Vs power performance

Module: config_files

This module provides parsing of the Quarch configuration files which list the full capabilities of the different modules. These abilities allow you to write scripts which can work with multiple different modules, taking account of the relative abilities of each.

```
> from quarchpy.config_files import *
```

Functions	Purpose
get_config_path_for_module	Locates the correct configuration file for a given module, matching the hardware and firmware versions.
parse_config_file	Parses a configuration file into a module capabilities class

TorridonBreakerModule class functions are:

Functions	Purpose
get_signals	Returns a list of module signals
get_signal_groups	Returns a list of signal groups
get_sources	Returns list of control sources
get_general_capabilities	Returns a dictionary of general capabilities (mainly binary flags)

Module: calibration

The calibration module provides access to the calibration feature on Quarch power modules, allowing the user to calibrate and verify their devices on-site.

This module is accessed by running the calibration utility. When run with all parameters supplied, the calibration can run automatically. When required parameters are missing, the utility will run in interactive mode and request the user to choose the options at run time.

Arguments are passed in using key value pair eg.

-mUSB:QTL1999-01-002 -acalibrate - pC:\\Users\\Public\\Documents

-a, action the calibration action you would like to perform. Choices: 'calibrate', 'verify'

-m, module IP address or net bios name of the module you would like to use

-i, instrument IP address or net BIOS name of calibration instrument

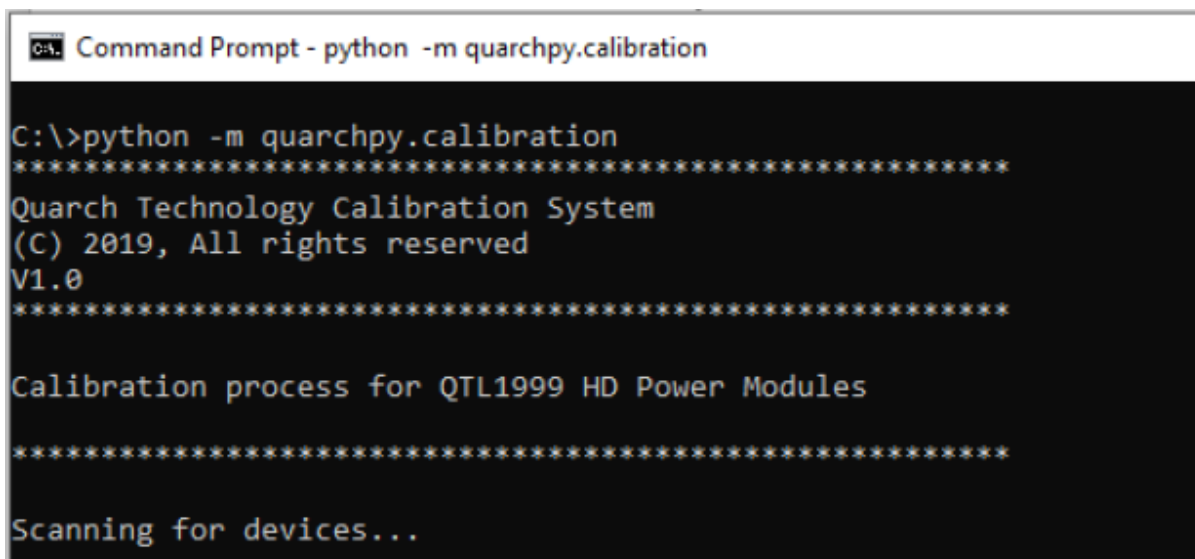
-p path path to store calibration logs

-l logging Lever of logging to report choices: 'warning', 'error', 'debug'

-u userMode Internal use only, should always be set to 'console'(and defaults to console)

-t temperature Encloser temperature that the unit is calibrated at.

> **python -m quarchpy.calibration [args]**



```
C:\>python -m quarchpy.calibration
*****
Quarch Technology Calibration System
(C) 2019, All rights reserved
V1.0
*****

Calibration process for QTL1999 HD Power Modules

*****

Scanning for devices...
```

Module: Run

The run package within quarchpy allows the user to run selected scripts and applications from the command line with one line. Quarchpy run can be utilized with the following command:

```
> python -m quarchpy.run [application] [args]
```

Applications/Scripts	Description
debug_info	Takes no arguments. Gives information system information about the host computer and scans for available Quarch devices on the system to test the basic functionality of quarchpy and allow a debug starting point for possible issues.
qcs	Takes no arguments. Launches Quarch Compliance suite back end.
calibration_tool	Takes in: action, module, instrument, path, logging, userMode, temperature. More information can be found in the calibration section. Launches the calibration tool.
qis	Launches Quarch Instrument Server.
qps	Takes in: keepQisRunning:True/False Launches Quarch Power Studio.
upgrade_quarchpy	Launches Upgrade Quarchpy Script.

Power studio commands

Power studio does not have its own module, but is included as part of the quarchDevice module. The quarchQPS class has the following functions:

Functions	Purpose
getCanStream	Returns true if the module supports streaming
startStream	Starts streaming to the given path, returns a quarchStream object for interacting with the stream data

quarchStream has a number of useful functions. These will affect the GUI trace view:

Functions	Purpose
addAnnotation	Add an annotation to the trace in real time (or at a specified time). Annotations allow the trace statistics to be split between test regions.
addComment	Add an annotation comment to the trace in real time (or at a specified time)
createChannel	Creates a custom channel to add additional data, such as drive temperature or IOPS
hideChannel	Hides the given channel
showChannel	Shows the given channel
channels	Lists the channels
stopStream	Ends the current stream
addDataPoint	Adds a data point to a custom channel

takeSnapshot	Triggers QPS take snapshot function and saves it in the streams directory.
get_stats	Returns the QPS annotation statistics grid information as a pandas dataframe object
stats_to_CSV	Saves the statistics grid to a specified csv file.
get_custom_stats_range	Returns the QPS statistics information over a specific time ignoring any set annotations.
saveCSV	This writes the whole stream data to csv file. This included all power data, custom channel data, annotations, and comments.

More information

Relevant application notes
https://quarch.com/file/AN-015-automating-qps Basic example showing scripted automation with QPS
https://quarch.com/file/an017-qps-performance-test-fio FIO automation with Power Studio, displaying IOPS Vs power performance
https://quarch.com/file/AN-016-qps-automation-iometer Iometer automation with Power Studio, displaying IOPS Vs power performance

QIS Control

QIS is a java based server app. It runs in the background when a 'QIS' type connection is requested. It also runs as part of Power Studio.

QIS handles streaming of power data from Quarch Power Modules. It deals with buffering, resampling and processing of the raw data.

In most cases, you will not need to use QIS directly, you will access it via the quarchpy API. However, QIS can run on its own and be controlled from any TCP compatible automation system.

A version of QIS is installed automatically with quarchpy, or can be downloaded from the Quarch website.

When run, QIS uses TCP and/or USB to connect to one or more Power Modules. You can control QIS via a simple network port

Port	Purpose
9722	Default TCP port for QIS control. This can be edited in the QIS properties file.

Sending a command

Commands are in simple text form. Commands beginning '\$' are parsed by QIS. All other commands are assumed to be intended for a specific Power Module (and so should follow the command spec as set out in the module's technical manual).

TCP commands are sent as a simple UTF-8 text string with \r\n termination:
"start stream\r\n"

Basic command set

\$help

Displays copyright, license and help information.

\$scan

Perform a scan for connected network attached devices via UDP port 30303. This command returns immediately but the actual scan takes about 3 seconds to complete. It is recommended to wait for a period before trying to \$list the devices or select a device.

\$scan LanAddress

Scan for a device at a specific tcp/ip address, used where the network does not allow UDP broadcast identification.

```
$scan tcp::192.168.1.90  
>Located Device: qt11944-01-020
```

```
$scan tcp::192.168.1.91  
>No Quarch Device Found at: 192.168.1.91
```

Once a device tcp/ip address has been scanned successfully the device will appear in the \$list and is available for use.

\$list

\$list details

Lists the devices seen by QIS (optionally with additional details for debug). The list is numbered 1-n for easy selection.

\$default listNumber

\$default deviceName

As QIS can control more than one Power Module at a time, you can specify a 'default' device to simplify control

```
# Specify the default module by position in the last $list  
$default 2
```

```
# Specify the default module by connection path  
$default tcp::192.168.1.90
```

```
# After this, any non '$' command will go to the default  
hello?
```

```
>HD Programmable Power Module +Triggering
```

\$shutdown

Closes QIS.

Addressed commands

If you do not use the `$default` command, perhaps because you are controlling more than one device at a time, you **MUST** specify the device for every command:

```
tcp::192.168.1.90 hello?  
>HD Programmable Power Module +Triggering
```

Here we chose to send the 'hello?' command to the specific LAN connected module.

Debug commands

These commands help to debug issues

```
$debug [on|off]  
$debug?
```

Sets/returns the debug more flag. Debug mode allows you to monitor historical events.

```
$debug history
```

Returns the most recent debug messages

```
$sysinfo
```

Displays memory usage and task information

Streaming commands

These commands are directed to a specific module (using `$default` or the module path) BUT are executed by QIS, as they are complex actions requiring large amounts of data to be transferred. QIS can buffer around 8 million stripes, allowing you to pull down the data when you are ready for it.

The commands allow you to easily handle the large amounts of high resolution data than can be returned by a power module.

Command examples below all assume that `$default` has been used to set the default module.

stream mode header [v1|v2|v3]

Selects the header version to return with the '**stream text header**' command. Setting this to **v3** is recommended for all new development

stream text header

Returns the stream header information from the most recent stream to be run on the selected module

Assuming you are using a recent version of QIS, this will be an XML encoded header, containing data on the module and each channel

Header Format (v3)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<header>
  # Header version
  <version>V3</version>
  # Future function, tracks mainPeriod for now
  <devicePeriod>4096us</devicePeriod>
  # Sampling rate of module
  <mainPeriod>4096uS</mainPeriod>
  # Legacy information, can be ignored
  <legacyVersion>5</legacyVersion>
  <legacyFormat>15</legacyFormat>
  <legacyAverage>10</legacyAverage>
  # List of all measured channels and their units
  <channels>
    <channel>
      # Name of channel, and its group name
      <name>Status</name>
      <group>status</group>
      # Channel units and max value range
      <units>NA</units>
      <maxTValue>0</maxTValue>
      # Position in return data
      <dataPosition>0</dataPosition>
    </channel>
    <channel>
      <name>5V</name>
      <group>voltage</group>
      <units>mV</units>
      <maxTValue>16384</maxTValue>
      <dataPosition>1</dataPosition>
    </channel>
    ...more channels...
  </channels>
</header>
```

stream?

Returns running status and buffer information for the current stream operation from the power module, eg:

```
Stopped: User  
Stripes Buffered: 94 of 8388608
```

or:

```
Running  
Stripes Buffered: 1167 of 8388608
```

stream text [all|number of stripes]

Returns a number of data stripes in formatted space delimited ASCII text from the stream buffer, example:

stream text 3

```
0 0 7 2 8 49  
1 0 7 2 8 50  
2 0 7 2 8 50
```

Data values are defined by the stream header and in the form:
[Stripe number] [Status] [Data channel 0 value] [Data channel 1 value] [...]

The max number of stripes returned will never exceed 4096 per call. If fewer stripes are available than is requested, the smaller number will be immediately returned.

stream bin [all|number of stripes]

Returns a number of data stripes in compressed binary format for faster transfer. This is used by Power Studio and not generally required by the user. Please contact support@quarch.com for details if needed.

Stream mode power [enable|disable]

Enable/disable automatic power calculation. When enabled, this creates new 'power' channels for each rail. Only valid when both voltage and current values are available for a given channel. Default is disabled

Stream mode power total [enable|disable]

Enable/disable internal power calculation **and** total power calculation. Only valid when two or more power calculations are available. Default is disabled and once enabled a stream mode power [enable|disable] command will disable this option.

Real-time resampling

stream mode resample?

stream mode resample off

stream mode resample 1..n[us|ms|s]

Force stream data to specified resample period. This allows arbitrary resampling to any required time-base. This involves averaging of every sample into the new scale, so no information is lost. (average power will always remain the same for example)

Resampling can only reduce the number of samples. If a resample time is selected which is smaller than the current instrument rate, the instrument setting will be used.

As an example, using hardware averaging (a device command), the closest we can get to 100mS is use 16k averaging of the 4uS base rate ($2^{16} = 16,384$) = 64uS

record:averaging 16k

Instead, using the QIS software resample command you can request the exact 100mS rate. This is more flexible and easier to understand than worrying about 1uS / 4uS base sample rates and averaging factors.

stream mode resample 100mS

For devices that support multiple groups (multi-rate feature of PAMs), an extended version of the command exists to set the resample time for each group.

stream mode resample [group_number] 1..n[us|ms|s]

In this example, we set group 0 (analogs) to 100mS and group 1 (digitals) to a faster 10uS rate.

stream mode resample 0 100mS
stream mode resample 1 10uS

QPS Control

Quarch Power Studio is a java based analysis application. It allows you to easily view and analyse huge power data recordings.

In most cases, you will not need to control QPS directly, you will access it via the quarchpy API. However, QPS can be controlled from any TCP compatible automation system.

A version of QPS is installed automatically as part of quarchpy, or can be downloaded directly from the Quarch website.

You can control QPS via a simple network port

Port	Purpose
9822	Default TCP port for QPS control

Sending a command

Commands are in simple text form. Commands beginning '\$' are parsed by QPS. All other commands are passed down the communications line to QIS and then finally to the connected Power Module (and so should follow the command spec as set out in the module's technical manual).

TCP commands are sent as a simple UTF-8 text string with \r\n termination:

“\$stream record\r\n”

All commands QPS V2 Commands Overview

All commands

Command	Brief Description
\$chart channel hide	Hides a channel on the QPS chart main view
\$chart channel show	Shows a channel on the QPS chart main view
\$chart snapshot	Takes a snapshot of the chart, timeline and channels windows (or one of the before mentioned) and saves it to the file path provided
\$debug list commands	Returns a list of commands
\$module connect	Connects QPS to the provided Quarch Module and opens the Chart main view read to use QPS with that module.
\$module disconnect	Disconnects the current Quarch module connection
\$module list	Lists available quarch modules
\$module list details	Lists available quarch modules in detail
\$module name	Returns the connected Quarch modules name
\$module scan	Starts a scan for new Quarch modules (Use \$module list to display found modules)
\$qis launch	Launches new QIS instance (The backend to QPS)
\$qis shutdown	Shuts down current QIS instance (The backend of QPS)
\$qps hide	Hides the QPS application
\$qps show	Shows the QPS application
\$stream annotation add	Adds an annotation to a stream. Annotations are used to mark points in time like the start of a test and stats between annotations are automatically calculated in the stats grid.
\$stream annotations list	Returns a list of all annotations with their titles in order of occurrence
\$stream annotations table	Returns a table of all annotations with additional information such as title, time, additional text, y position on chart.
\$stream channel add	Creates a new channel
\$stream channel add synthetic	Defines a new synthetic channel
\$stream channel clear synthetic	Clears all created functions
\$stream channel list	Returns a list of all channels in the stream
\$stream channel remove synthetic	Deletes a previously created synthetic channel
\$stream channel table	Returns details on all channels in the stream
\$stream channels	Returns just the names of the channels in the stream
\$stream comment add	Adds a comment to the stream (Comments are the same as annotations but don't interact with stats calculations)

\$stream data add	Adds a datapoint of a given value at a certain time to the specified channel
\$stream export	Saves the stream data to file in the specified format
\$stream read	Reads stream data between from a given start point and end point
\$stream record	Starts the QPS stream
\$stream record duration	Returns the total stream recording time
\$stream start time	Returns the time that stream started
\$stream stats export	Saves the stream statistics to a file
\$stream stats table	Returns the stream statistics
\$stream stop	Stops the current running stream
\$stream stop time	Returns the time that streaming stopped

All commands

Below is a list of all commands with more information on how to use them and the arguments that can be passed with each.

\$chart channel hide

Hides a channel

Arguments

channel : String : The name of the channel you would like to hide : arg index 0 : default none

group : String : The name of the group the channel you would like to hide belongs to: arg index 1 : default none

Use case

This can be used to hide some channels from the main chart window, so you can focus on only the channels you are interested in. If you are streaming from a device with many channels or you have added your own custom channels or synthetic channels to the main chart it can get cluttered. Hiding a channel simply stops the channel from being drawn but the data for that channel is still recorded so it can be added back into the chart later.

\$chart channel show

Shows a channel

Arguments

channel : arg index 0 : String : The name of the channel you would like to show : default none

group : arg index 1 : String : The name of the group the channel you would like to show belongs to : default none

Use case

If you have previously used “\$chart channel hide” you can use “\$chart channel show” to show the channel again. This is particularly useful when you are automating taking screenshots and would like multiple shots of the same area of the graph but showing different channels in each shot.

\$chart reposition

Sets the chart main view to the specified start and end times

Arguments

startTime : The start time that the chart window will show from : arg index 0 : String : default None

endTime : The end time the chat window will show time until : arg index 1 : String : default None

Use case

If you know the time in which a test took place in your stream then you could use this command to fit the chart window to exactly that position. This can be very powerful when coupled with the snapshot and channel show/hide commands to automate screenshots of test sections for report generation.

Both the start and end times are specified as a time-unit pair; for example: 100mS

Both start time and end time must be larger than 0 and start time must be larger than end time.

\$chart snapshot

Takes a snapshot of the QPS Window

Arguments

type : Type of snapshot to take consisting of a comma separated list of options: [all, chart, timeline, key]: arg index 0 : String : default "all" : Acceptable values: [all, chart, timeline, key]

filePath : Optional file path instead of using default archive path : arg index 1 : String : default None

Use case

This can be used to screenshot qps and saved to file for later use in reports or presentations. The command can be used with chart reposition and channel show/hide to automate many different displays of test data on the chart view.

\$debug list commands

Returns a list of commands

Arguments

None

Use case

This is an internal command used to list the commands in a specific format for internal testing. If you have come across this... these are not the droids you are looking for. Move along.

\$module connect

Connects QPS to the provided Quarch Module and opens the Chart main view read to use QPS with that module.

Arguments

device : The device you would like to connect to : arg index 0 : String : Default none

Use case

You must connect to a module before getting stream data from it.

```
$module connect TCP::QTL1999-01-001
```

```
$module connect USB::QTL1999-01-001
```

\$module disconnect

Disconnects the current Quarch module connection

Arguments

None

Use case

To free up the current quarch module to be used by another part of your script.

\$module list

Lists available quarch modules

Arguments

None

Use case

You may want to check your devices is visible to QPS before continuing with your script. This command allows you to receive a list of all visible devices QPS can see.

\$module list details

Lists available quarch modules in detail

Arguments

None

Use case

You may wish to see all available devices and parse details from them before deciding what to connect to.

\$module name

Returns the connected Quarch modules name

Arguments

None

Use case

This is very useful for confirmation of the connected device and is useful to add to reports generated to keep track of what modules were used during tests.

\$module scan

Starts a scan for new Quarch modules (Use \$module list to display found modules)

Arguments

deviceSpecifier : TCP/IP address to scan for devices : arg index 0 : String : default none

Use case

Used to scan for devices on a specific IP address. Usually needed when devices are on different subnets.

\$open recording

Opens a QPS recording file into the chart view

Arguments

qpsFile : The file to be opened : arg index 0 : String : default None

Use case

To automate opening a recording. This could be used to open old recordings gather stats and then move onto the next saved recording for mass data processing.

```
$open recording qpsFile="C:\quarch\recordings\AC\AC_Gen_test\AC_Gen_Test.qps"
```

\$qis launch

Launches new QIS instance

Arguments

None

Use case

Only to be used If QIS has been shutdown. QPS Launches QIS automatically on start up so you would rarely need to use this command.

\$qis shutdown

Shuts down current QIS instance

Arguments

None

Use case

In the rare event you wish to close a crashed QIS or you are comparing different versions of QIS with the same QPS. Mainly an internal command.

\$qps hide

Hides the QPS application

Arguments

None

Use case

If you wanted to run fully automated then you could hide QPS so it runs in background not showing the main chart view.

\$qps show

Shows the QPS application

Arguments

None

Use case

To show a QPS application had had been hidden, potentially at the end of the entire test.

\$stream annotation add

Adds an annotation to a stream. Annotations are used to mark points in time like the start of a test and stats between annotations are automatically calculated in the stats grid.

Arguments

time : Annotation time : arg index 0 : string default none

text : annotation title : arg index 1 : String : default ""

extraText : Additional annotation text : arg index 2 : String : default ""

ypos : y position percentage on screen : arg index 3 Int Range [0:100] : default 100

type : type string used when assigning colour and text colour : arg index 4 : String : default Type1

colour : colour of annotation icon : arg index 5 : Hex : default #FF0000

textColor : colour of annotation text : arg index 6 : Hex : default #FF0000

timeFormat : format for arg time : arg index 7 : String [elapsed, unix] : default elapsed

Use case

Annotations are very useful and are usually added at the start and end of tests where more than one test is ran per stream. Stats are auto generated between annotations and can be exported at any time.

```
$stream annotation add 7S Annotation1
```

```
$stream annotation add 7500mS Annotation2
```

```
$stream annotation add 00:00:08 Annotation3
```

\$stream annotations list

Returns a list of all annotations

Arguments

type : Specifies the type of annotations : arg index 0 : String [all, annotations, comments] : default all

Use case

If you wanted to gather the list of all annotations and/or comments comments added to the stream, this command returns that information which can be parsed to check what tests have ran or whatever information you have added to your annotations and comments.

\$stream annotations table

Returns a table of all annotations

Arguments

type : Specifies the type of annotations : arg index 0 : String [all, annotations, comments] : default "all"

timeFormat : Specifies the time format : arg index 1: String [nS, uS, mS, S, HMS] : default "S"

Use case

This returns a table containing annotation and or comment information for the current stream. There is much more detail in this than \$stream annotation list such as extra text

\$stream channel add

Creates a new channel

Arguments

channelName : name of the channel to be added :Index: 0 : Type: STRING :Default value: NONE

channelGroup : group of the channel to be added : Index: 1 : STRING Default value: NONE

baseUnits : units of the channel to be added : index 2 : String : default value None

usePrefix: Indicates whether scientific notation prefix : index 3 : default value None

Use case

Create a custom channel such which you populate with data yourself from script. This could be drive temperature and could be added live to the stream.

`$stream channel add synthetic`

Defines a new channel function

Arguments

channel : Defines the synthetic channel : arg index 0 : String : default none

function : Specifies the synthetic function arg index 1 : String : default none

Use case

To add a channel that is based on another channel. You might want to plot the Root Square Mean of a channel for example.

```
$stream channel add synthetic channel="chan(L1_RMS,V)" function="rms(100ms,chan(L1,V))"
```

`$stream channel clear synthetic`

Clears all created functions

Arguments

None

Use case

Clears all synthetic channels from stream

`$stream channel list`

Returns a list of all channels in the stream

Arguments

None

Use case

List the Channels in the stream

`$stream channel table`

Returns details on all channels in the stream

Arguments

channelType : Defines the user or synthetic channel: arg index 0 : String : default “all”

Use case

To get detailed information about the channels in the stream.

`$stream channel remove synthetic`

Deletes a previously created channel function

Arguments

channel : Specifies the synthetic channel : arg index 0 : String : default None

Use case

Specifies the synthetic channel to be removed.

```
$stream channel remove synthetic channel="chan(L1_RMS,V)"
```

`$stream channel table`

Returns details on all channels in the stream in detail

Arguments

channelType : Defines the user or synthetic channel : arg index 0 : String : default “all”

Use case

If you wanted to know what channels were in the stream, the sources of their data, and how they are calculated if synthetic.

\$stream channels

Returns just the names of the channels in the stream

Arguments

None

Use case

If you simply want a list of all channels in the current stream.

\$stream comment add

Adds a comment to the stream (Comments are the same as annotations but don't interact with stats calculations)

Arguments

time : Comment time : arg index 0 : string default none

text : Comment title : arg index 1 : String : default ""

extraText : Additional comment text : arg index 2 : String : default ""

ypos : y position percentage on screen : arg index 3 Int Range [0:100] : default 100

type : type string used when assigning colour and text colour : arg index 4 : String : default Type2

colour : colour of comment icon : arg index 5 : Hex : default #FFFF00

textColor : colour of comment text : arg index 6 : Hex : default #FFFF00

timeFormat : format for arg time : arg index 7 : String [elapsed, unix] : default elapsed

Use case

You may wish to annotate a stream visually by adding a marker to the chart but not have it effect the stats calculation, such as at the end of a stream you could add a comment at the peak of one of your channels.

```
$stream comment add 10.25S Comment1
```

```
$stream comment add 11000mS Comment2  
$stream comment add 00:00:12 Comment3
```

\$stream data add

Adds a datapoint of a given value at a certain time to the specified channel

Arguments

channel: channel name : arg index 0 : String : default None

group : group the channel belongs to : arg index 1 : String : default None

time : The time the data is to be added at: arg index 2: String : default None

value : the value of the data to be added: arg index 3 : String : default None

TimeFormat : format for value time : arg index 4 : String ["elapsed", "unix"] default "elapsed"

Use case

If you have a custom channel such as temperature you can plot temperature at a given time y for value x.

All data is linearly interpolated but if you would like the interpolation to stop between two points then use an end marker.

To indicate the end marker of an interval then set the value to 'endSeq'; eg

```
$stream data add 'Fet1Temp', 'Temperature', '1100ms' 'endSeq', 'elapsed'
```

\$stream export

Saves the stream data to file in the specified format

Arguments

file : The file path to save the export : arg index 0 : String : default None

maxLines : max number of lines per file : arg index 1 : String : default all

lineTerminator : Indicates whether to use a lineTerminator: arg index 2 : String [yes/no] : default "yes"

delimiter : Value separator : arg index 3 : String ["\t", ";", ",", " "] : default ","

fileType : file format : arg index 4 : String ["csv"]: default "csv"

Use case

Export the stream to a csv file for post processing.

\$stream read

Reads stream data between from a given start point and end point

Arguments

startTime : read start point : arg index 0 : Int : default None

endTime : read end point : arg index 1 : Int : default None

timeFormat : timeFormat : arg index 2: String [elapsed, unix] : default “elapsed

Use case

You may want to read the stream data back for a specified time for processing in script.

\$stream record

Starts the QPS stream

Arguments

file : the path to file where you would like to stream data to: arg index 0 : String : default None

Use case

Used to start a stream from Quarch module to QPS.

\$stream record duration

Returns the total stream recording time

Arguments

timeFormat : timeFormat : arg index 0 : String [nS, uS, mS, S, HMS] : default “S”

Use case

If you want to find out the QPS stream running time total for use in your automation script.

Notes:

A value of -1 is returned if streaming has not commenced

`$stream start time`

Returns the time that streaming started

Arguments

`timeFormat : timeFormat : arg index 0 : String [nS, uS, mS, S, HMS] : default “S”`

Use case

To find out the time the stream started.

Notes:

A value of -1 is returned if streaming has not commenced

`$stream stats export`

Saves the statistics to a given format

Arguments

`file : file for stats to be save to : arg index 0 : String : Default None`

`fileType : format of data in file: arg index 1 : String [“csv”] : Default None`

Use case

Used to save the stats to file for processing later.

`$stream stats table`

Returns the stream statistics

Arguments

`startTime : start point for stats : arg index 0 : int : default None`

`endTime : end point for stats : arg index 1 : int : default None`

`timeFormat : timeFormat : arg index 2: String [elapsed, unix] : default “elapsed`

Use case

Use this command to get back all the stats information based on the annotations set, or pass a start and end time to calculate stats between those two points.

`$stream stop`

Stop the current stream

Arguments

None

Use case

To stop the current running stream.

`$stream stop time`

Returns the time that streaming stopped

Arguments

timeFormat : timeFormat : arg index 0 : String [nS, uS, mS, S, HMS] : default "S"

Use case

Find the time when the stream stopped for processing in your scripts.
A value of -1 is returned if streaming has not commenced