

Qu@sar Primer

Quanterra Data Packet (QDP) packets are used to talk to Q330 digitizers and PB14 packet balers. This QDP protocol is implemented on top of UDP, and consists of commands and responses.

Q330s and Balers respond to each QDP command packet that is sent with a QDP response packet. That response may be the appropriate response for the command, or it may be an error response, with details about the error. [Qu@sar](#) looks for the error packets and turns them into an exception that can be caught.

QDP can be used to communicate with a Q330 or PB14 in one of two ways. First, UDP/IP may be used via the Ethernet port. Second, UDP/IP may be SLIP (<http://www.faqs.org/rfc/std/std47.txt>) escaped, and sent to one of the serial (including IR on the Q330) ports.

A QDP packet is fairly simple. All QDP packets share a common header that includes the following information:

- A CRC
- The ID of the command that the packet represents
- The protocol version (Q330s up to this point use Version 2)
- The length (in bytes) of the packet's data (does not include the header)
- A sequence number (0..65535) for packet ordering
- An ACK number (representing the packet that is being responded to)

A full list of available QDP packets and their responses can be found in the document titled “Q330 Communications Protocol” available from Quanterra. The “DP Writers' Guide” also from Quanterra will give a lot of insight into the various things that you can do using QDP.

Sending a command using [Qu@sar](#)

Please note that in the example below, modules are imported as they are needed. Generally this is a bad idea, and you should do all of your module imports in one place. This was done to group all related code together, to make the explanation easier to understand.

Before being able to create a Q330 instance, we need to make sure that the Q330 class is imported. We also want to make sure that the QDP_C1_CERR exception is imported. The following line brings the class “Q330” from the module with the same name, into our namespace:

```
from Quasar.Q330 import Q330, QDP_C1_CERR
```

Now that we have access to the Q330 class, we need to create an instance of it. The following line creates an instance of a Q330 object, representing a Q330 with a serial number of 0x1234, an IP of 192.168.1.166 and a base port of 5330:

```
q330 = Q330(0x1234, '192.168.1.166', 5330)
```

We can now start to send commands to the Q330 object. Most of the time, the first thing you will want to send is a registration request. Since registration consists of several QDP commands, and it's an operation that's used frequently, the Q330 class has a register function included. Before registration can be performed, however, we need to tell the Q330 object what authentication code it should use. As with all communication with the Q330, if the communication fails for some reason, an exception is thrown. We can use a try/except pair to look for this:

```
q330.setAuthCode(0)
try:
    q330.register()
except QDP_C1_CERR:
    print 'Unable to register'
    import sys
    sys.exit(1)
```

Now that we are registered with the Q330, we can send any other commands. For example, we may want to see the Q330's physical port configuration. To do this, we would need to create an instance of a c1_rqphy command packet, and send the command to the Q330. In return, we should get a c1_phy packet. We can confirm this by checking the QDPCommand data member against the value of a constant defined in the CmdID module (Read the "PacketFields" document for a list of the QDP header data members). As with all QDP packets, we can simply print the c1_phy response and it's decoded for us. If an exception is raised, we'll print that instead, so that the user knows what went wrong.:

```
from Quasar.Commands.c1_rqphy import c1_rqphy
rqphy = c1_rqphy()

try:
    response = q330.sendCommand(rqphy)
    from CmdID import C1_PHY
    if response.getQDPCommand() != C1_PHY:
        print 'Did not get a c1_phy'
        import sys
        sys.exit(1)
    else:
        print response
except QDP_C1_CERR:
```

```
import traceback
traceback.print_exception()
```

Finally, we need to deregister from the Q330. Like registration, this is a procedure that is done commonly enough that the Q330 class has a deregister method. There are also several other methods that are built into the Q330 class. Examine the Q330 module's documentation for more.

```
q330.deregister()
```

We are now done talking to the Q330, and the application can continue. All network connections are closed, and the Q330 has been left in a state that will allow other applications to talk to it.

When run, this script will produce the following output:

```
c1_phy
  SerialNumber: 4660
  SerialInterface1IPAddress: 10.1.1.2
  SerialInterface2IPAddress: 10.2.2.2
  SerialInterface3IPAddress: 0.0.0.0
  EthernetIPAddress: 192.168.1.166
  EthernetMACAddressHigh32: F2:3F:06:9A
  EthernetMACAddressLow16: 41:26
  BasePort: 5330
  SerialInterface1Baud: 19200
  SerialInterface1Flags:
    Enabled
    Routing disabled
    Always use programmed IP address
  SerialInterface2Baud: 38400
  SerialInterface2Flags:
    Enabled
    Routing disabled
    Always use programmed IP address
  SerialInterface3Baud: 1200
  SerialInterface3Flags:
    Disabled
    Routing disabled
    Always use programmed IP address
  Reserved: 0
  EthernetFlags:
    Enabled
    Link detect enabled
    Half duplex
    Routing disabled
```

```
SerialInterface1Throttle: 0
SerialInterface2Throttle: 0
SerialInterface3Throttle: 0
Reserved: 0
```

This is the result of the “print response” line. In this case, “response” is a c1_phy QDP packet. [Qu@sar](#)'s built in packet decoding allows us to simply print the packet, and get a human readable response. If we were to replace the “print result” line with the following:

```
print 'The base port is %d' %response.getBasePort()
```

The script would then produce the following output when run:

```
The base port is 5330
```

[Qu@sar](#)'s QDP packet classes have get/set functions for every data member, so when we call getBasePort() on the c1_phy named “response”, we are given the base port in return. Notice that in this case, what is returned is an integer. This is exactly what the Q330 returned to us.

In some cases, this may not be what you want. For example, the Q330 stores each IP address in a 32 bit integer. If you replace the “print response” line from the original example with the following:

```
print 'Ethernet IP: %s' %response.getEthernetIPAddress()
```

You will get the following result:

```
Ethernet IP: 3232235942
```

This is probably not what we would have liked to see. Another feature of all QDP command packets is that they have the ability to make a “human readable” representation of any data member. This is done using the str<DataMember>() functions. If we replace the “print” line with:

```
print 'Ethernet IP: %s' %response.strEthernetIPAddress()
```

We get the more desirable result of:

```
Ethernet IP: 192.168.1.166
```

Changing settings is very simple as well. For example, if we wanted to set Serial Port 1's baudrate to 1200, we would first need to get the physical port configuration, which we are doing already. For clarity, we will change some variable

names:

```
phyResponse = q330.sendCommand(rqphy)
```

Now that we have the physical port configuration, we need to create a QDP packet to change these settings. We can initialize this new QDP packet using the bytes from the phyResponse packet that we just received, since according to the “Q330 Communications Protocol” these two QDP packets look identical:

```
from Quasar.Commands.c1_sphy import c1_sphy
newConfig = c1_sphy( phyResponse.getPacketBytes() )
```

Now, we need to make the change in the newConfig packet. According to the “Q330 Communications Protocol” document, the value for “1200 baud” is 1, so we need to set this value in newConfig. There is no support in [HYPERLINK](#) ["mailto:Qu@sar"Qu@sar](mailto:Qu@sar) for converting values back from their string representation (the reverse of the str<DataMember> methods) for the time being, you need to do this conversion yourself. This becomes troublesome when converting IP addresses to their 32 bit value. Python provides a function named “inet_aton” to handle this, in the “socket” module (Socket module documentation can be found at <http://www.python.org/doc/current/lib/module-socket.html>):

```
newConfig.setSerialInterface1Baud(1)
```

Finally, we need to send the newConfig command packet:

```
q330.sendCommand(newConfig)
```

Once this packet is sent, the baudrate for serial interface 1 will be set. By adding the following lines, we can confirm this:

```
res = q330.sendCommand(rqphy)
print 'Baud: %s' %res.strSerialInterface1Baud()
```

The output should look like:

```
Baud: 1200
```

Again, note the use of the res.strSerialInterface1Baud() instead of res.getSerialInterface1Baud() which would have returned the coded value of “1”.

The Site Configuration System:

When there are multiple Q330s on a network, things start to get complicated.

You need a way to keep track of which devices are available, and how to communicate with them without confusing it or the other Q330s on the network.

The `Quasar.Site.Config` module addresses this issue in a few different ways. First, it has a `discover()` method that attempts to discover all Q330s on the network. This discovery process is highly configurable, and includes registration with each Q330 it finds in order to determine the IP address of that device, and deregistration, leaving the Q330 in the same state it was found in. The documentation for `discover()` is as follows:

```
discover all Q330s reachable, and build a site configuration

defaultAuthCode can be passed in to define a default authentication
code for all registration attempts. This defaults to zero.

authCodeMap is a Python dict type, that maps either a Q330's tag or it's
serial number to an auth code. This can be used for exceptions to
the default. This defaults to an empty dict.
```

The result of this function call is a `Quasar.Site.Config.Config` instance, which defines several methods for searching through the configuration for a particular device's information, as well as the ability to write itself to a file. This file can be used later to re-create the configuration object. When queried for information about a Q330, the `Config` class returns a dict type with various key/value pairs explaining the device.

The `discoverNetwork` example demonstrates the most basic way of polling the network for Q330s (all of which are assumed by the example, to have an `AuthCode` of zero) and generating a site config file.

An instance of `Quasar.Site.Config.Config` may be queried for the information about a particular Q330 by providing the proper information (IP, the `SerialNumber`, or the `Tag` number) to one of the following functions:

```
getDeviceInfoByTag()
getDeviceInfoByIP()
getDeviceInfoBySerial()
```

All of these functions will return a dict with at least the following keys: `IP`, `Serial`, `BasePort`. Other key/value pairs may also be included (see below).

An example entry that the config file may contain is:

```
Q330.3.IP = 192.168.1.165
Q330.3.Serial = 10000069a37e501
Q330.3.BasePort = 5330
```

In this case, there is a Q330 with a KMI Tag number 3. It has an IP address of “192.168.1.165”, a serial number “10000069a37e501” and a base port of “5330”. These values may be edited by hand at any time, and the next time the file is read, the new values will take effect.

In addition to these three values, the user may add new values to a Q330's entry in the config file. For example, if the user adds the following line:

```
Q330.3.AuthCode = 12
```

Whenever the information for this Q330 is returned from the Config object, it will have a new key "AuthCode" that has a value of "12". If the Config object ever overwrites the config file, all custom key/value pairs will be lost. [Qu@sar](#) does not know about any other key/value pairs, but code that you write can take advantage of this convenient location to store information about a Q330.

It is suggested that the discovery process be used to generate an initial config file, and future editing be done by hand, to preserve custom key/value pairs.

How to read the "Q330 Communications Protocol" document:

When reading through the Quanterra "Q330 Communications Protocol" document, there are a few conventions to be aware of.

The following table describes the datatype names, and their meanings:

<i>Name</i>	<i>Explanation</i>
Byte	Unsigned 8 bit value
Shortint	Signed 8 bit value
Word	Unsigned 16 bit value
Integer	Signed 16 bit value
Longword	Unsigned 32 bit value
Longint	Signed 32 bit value
Double	Double precision floating point value
Float	Single precision floating point value

Packets will be described using tables similar to the table below representing the QDP Header:

CRC		
Command	Version	Length
Sequence Number		Acknowledge Number

In tables such as these, each row represents 32 bits, and may be divided up into as many as four 8 bit values. In the above example, CRC is a 32 bit value, Command and Version are 8 bit values, and Length, Sequence Number and Acknowledge Number are all 16 bit values.