Application Note- AN001

Creating a custom glitch/drive sequence

July 20, 2016

Contents

Creating a custom glitch/drive sequence

In this particular case, a customer requested to customize a sequence of events over a short time interval. This was required to simulate a fault situation that had been observed when their PCIe device was connected to a particular host.

As the shortest duration was ~200ms, and total accuracy was not required, we can do this with a simple script of command to the module.

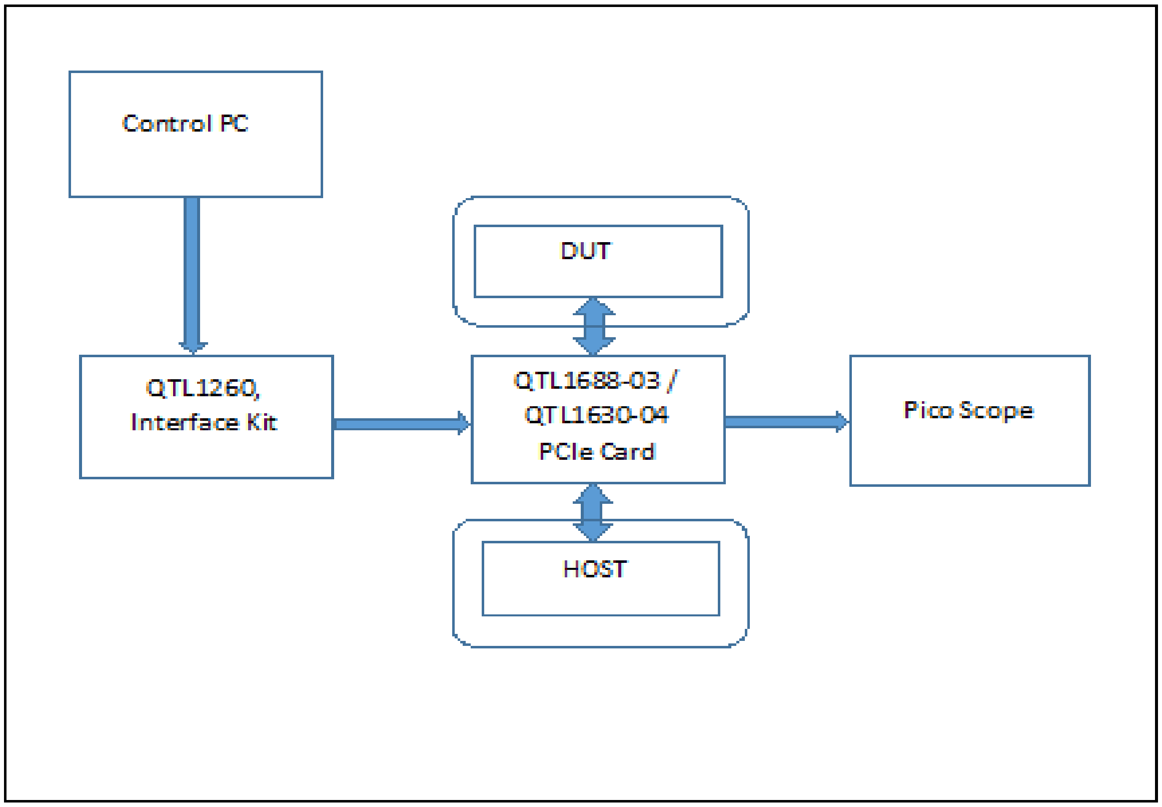
Quarch PCIe breaker modules have the ability to both make/break signals, and to actively drive some signals into specific high/low logic states.

Requirements

1) Quarch PCIe Card Module (QTL1688-03 / QTL1630-04)
   Earlier model hardware will work, but cannot 'drive' signals
2) Torridon Interface Kit (QTL1260)
   This provides us with a serial / USB connection to control the module
3) Picoscope (or similar)
   This is not essential, but is used here to display the results

Equipment Set up

The following equipment setup is used:



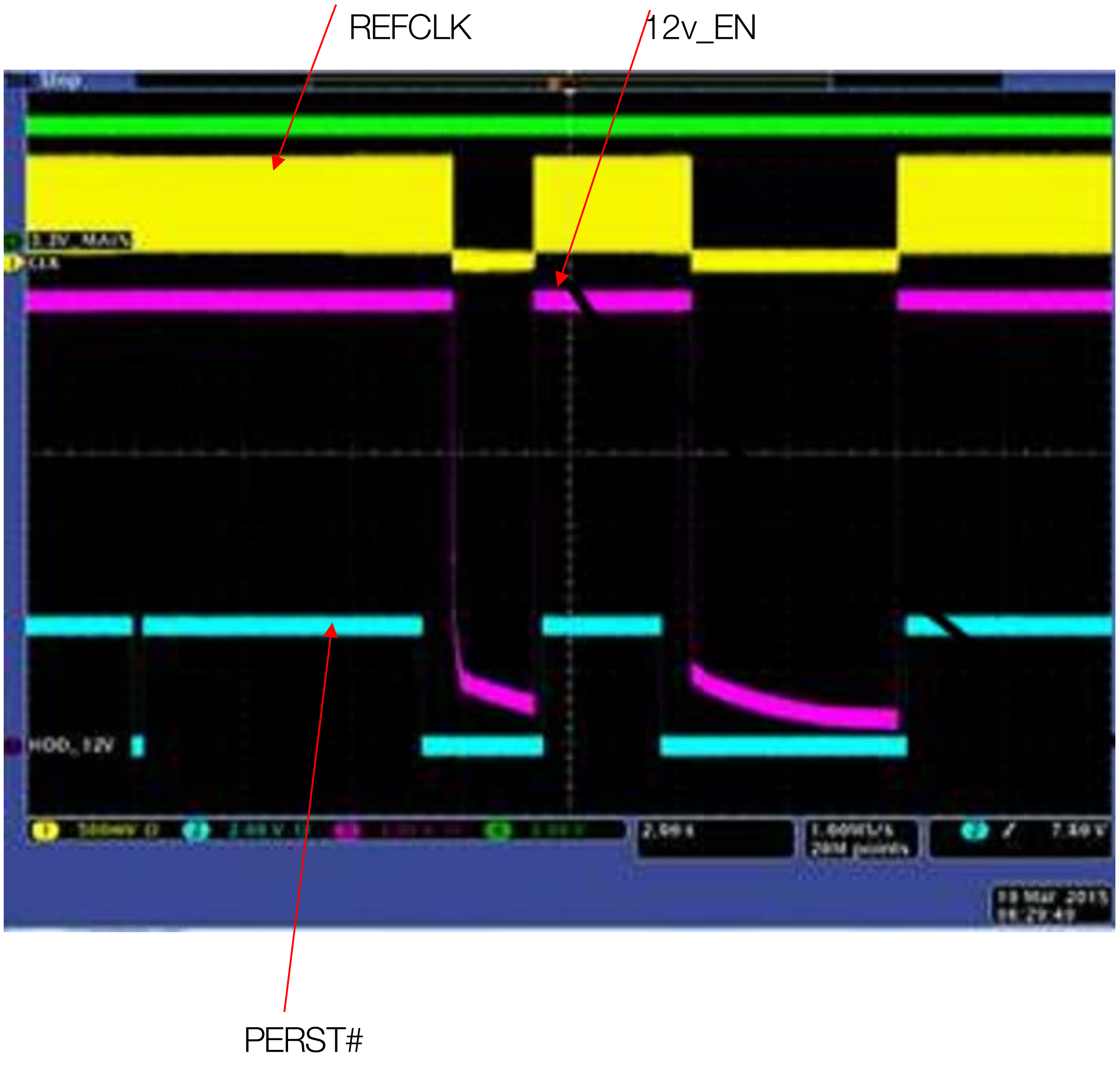
In this scenario, Control PC is connected to QTL1260, Interface Kit to control the PCIe Card, and the three signals under observation are as follows:

- PERST
- REFCLK
- 12V_EN

The three signals under observation were displayed on the Pico scope.

Customer Required Sequence

Below is the sequence of events that the customer requested.



There are three channels under observation are 12V_EN, REFCLK and PERST. Each signal needs to change state at a specific time in relation to each other, in order to recreate the failure scenario.

Creating the Sequence

We used the customers supplied image to work out the timing sequence required. We then assigned each of the signals to one of the timed sources on the module:

```
SIGNAL:PERST:SOURCE 1
SIGNAL:REFCLK:SOURCE 2
SIGNAL:12V_EN:SOURCE 2
```

REFCLK and 12V_EN are assigned to the same source, as they move at exactly the same time in the customer's image.

We start in a 'power up' state with all signals connected, as if our module was not present. The DUT is operating normally.

We can now change the state of the signals, by changing the state of the source that the signal is assigned to:

Source is ON:	The signal will be connected as normal
Source is OFF:	The signal will be disconnected

This is fine for REFCLK and 12V_EN, however the PERST signal needs to be actively pulled down. This could be done with an extra pull-down resistor on the DUT side, or using the DRIVE function of the latest Quarch PCIe Card Module:

```
SIGNAL:PERST:DRIVE:OPEN LOW
SIGNAL:PERST:DRIVE:CLOSED HIGH
```

These commands order the Module to drive the signal LOW when the signal would normally be disconnected, and to drive it high when it would normally be connected. These settings completely disconnect the PERST signal from the host and allow us to take full control of it.

We can now control the source enables flags to transition the signals. When the source is 'ON' signals will be connected. When 'OFF', signals will be disconnected.

Following the rules we set above, the PERST signal will be driven high when source 1 is 'ON' and low when it is 'OFF'

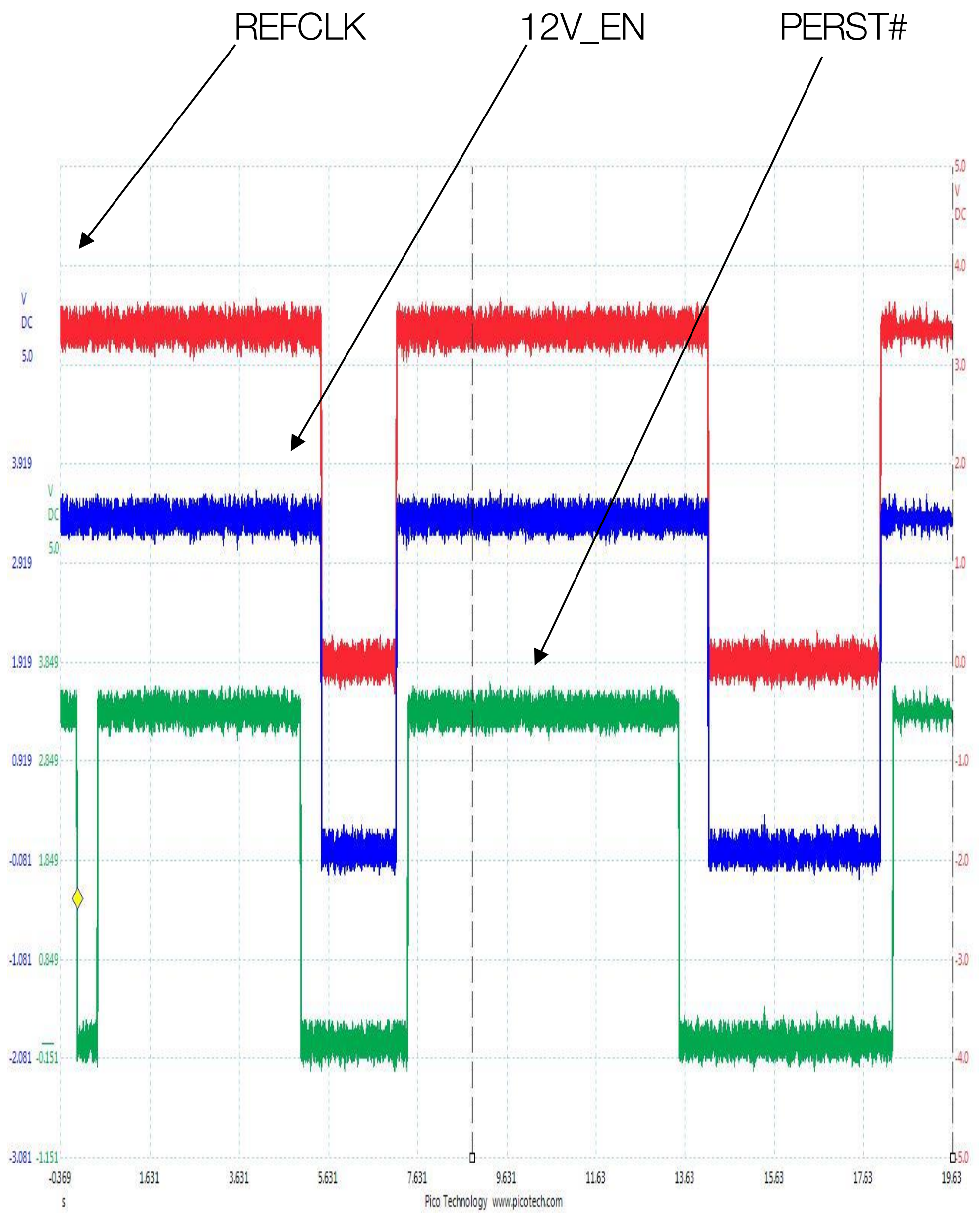
Placing delays in between the source on/off commands allows us to create the required delays. As the fastest transition is ~0.2 seconds, this is a reasonably accurate solution.

```
SOURCE:1:STATE OFF
sleep(0.4)
SOURCE:1:STATE ON
sleep(4.5)
SOURCE:1:STATE OFF
sleep(0.4)
…
```

The entire sequence is created in this way. It does not matter which version of hardware you are using (which may support driving). The command sequence is identical in either case. Any driving of signals is handled by the module internally, using the settings we made previously.

Results

Using the PICOSCOPE, we place probes onto the control signals for each switch (We cannot monitor the live data, as we did not have the customers host available for the testing). We observe the signals behaviour and captured the image of the sequence of the events:



Controlling the Module

In our case, we used Perl to send commands to the module, via the USB Virtual COM port created by the QTL1260 Interface Kit. The commands could be sent over any other Serial, USB or LAN connection (though network latency could make the test less accurate via LAN).

Quarch have examples for Perl, Python, .NET Java. Many other languages and automation tools can also be used to control the modules.

Appendix 1 – Perl Example Script

```
#!/usr/bin/perl
#
# Author: Mike Dearman
# Date:   20-03-2015
# Requirements: Win32::SerialPort
# (Uses same interface as Device::SerialPort linux module so can be ported)
#
# Version: 1.0
#
# Description:  This perl script is for customizing a sequence of events
over a short interval of time manually.
# PCIe Card Module has been connected via a serial connection.In this case
the customer has been using complex
# server backplane design which has an FPGA which is controlling PERST#, SSD
+12V and REFCLK.

use strict;
use warnings;
use Time::HiRes qw (sleep);        # Perl Sleep command for short intervals
use Win32::SerialPort;             # use Win32 serial port module

# Import torridon common functions, See Appendix - 2
require 'TorridonCommon.pl';

#
# Serial Settings
#
# Create SerialPort object called $Connection and configure

my $PORT = "COM4";                # **** Serial port to use ****


my $Connection = Win32::SerialPort->new ($PORT) || die "Can't Open $PORT:
$!";
$Connection->baudrate(19200)    || die "failed setting baudrate";
$Connection->databits(8)        || die "failed setting databits";
$Connection->stopbits(1)        || die "failed setting stopbits";
$Connection->handshake("none")  || die "failed setting handshake";
$Connection->parity("none")     || die "failed setting parity";



#SerialPort defers change to serial settings until write_settings()
#method which validates settings and then implements changes
$Connection->write_settings     || die "no settings";
```

```
print "\n# Running Script\n\n>";

#reset module to defaults
print"Print RETURN to reset module";
<STDIN>;
&SendTorridonCommand ($Connection, "conf:def state");

# assign all signals to source 8 so they are always on
&SendTorridonCommand ($Connection, "sig:all:source 8");

#reassign the signals we want to control
&SendTorridonCommand ($Connection, "sig:PERST:source 1");
&SendTorridonCommand ($Connection, "sig:12V_POWER:source 2");
&SendTorridonCommand ($Connection, "sig:REFCLK_MN:source 2");
&SendTorridonCommand ($Connection, "sig:REFCLK_PL:source 2");

#set signal driving logic on PERST signal (supported on QTL1688-04 &
QTL1630-04 modules and above)
#&SendTorridonCommand ($Connection, "SIGnal:PERST:DRIve:OPEn LOW");
#&SendTorridonCommand ($Connection, "SIGnal:PERST:DRIve:CLOsed HIGH");

print"Print RETURN to start script";
<STDIN>;

# Timing sequence begins here
&SendTorridonCommand ($Connection, "source:1:STATE off");
sleep(0.4);
&SendTorridonCommand ($Connection, "source:1:STATE on");
sleep(4.5);
&SendTorridonCommand ($Connection, "source:1:STATE off");
sleep(0.4);
&SendTorridonCommand ($Connection, "source:2:STATE off");
sleep(1.6);
&SendTorridonCommand ($Connection, "source:2:STATE on");
sleep(0.2);
&SendTorridonCommand ($Connection, "source:1:STATE on");
sleep(6);
&SendTorridonCommand ($Connection, "source:1:STATE off");
sleep(0.6);
&SendTorridonCommand ($Connection, "source:2:STATE off");
sleep(3.8);
&SendTorridonCommand ($Connection, "source:2:STATE on");
sleep(0.2);
&SendTorridonCommand ($Connection, "source:1:STATE on");

print "\n\n# Script Completed!";

# Close the COM port
undef $Connection;
```

Appendix – 2 Torridoncommon.pl

```
#!/usr/bin/perl
#
# Author: Andy Norrie
# Date:   27-04-2011
# Requirements: Win32::SerialPort
# (Uses same interface as Device::SerialPort linux module so can be ported)
#
# Version: 1.0
#
# Description:  This perl script is for demonstrating issuing commands to
# multiple torridon modules via a Torridon array controller using a serial
#  connection. Data from the serial port is logged to the specified file
(Quarch.log).
# The logfile can then be parsed with other programs for reporting purposes.
#
use strict;
use Win32::SerialPort;            # use Win32 serial port module

# 1 = Log commands and responses, 0 = No logging
my $LogText = 1;
# 1 = Config:Terminal set to 'SCRIPT', 0 = Config:Terminal set to 'USER'
mode
my $ScriptMode = 1;

# Subroutine to send a command to the attached device.  The serial port must
be open
# and the device ready to receive commands.  This function will not block,
even if the
# module does not respond
# string SendTorridonCommand ($SerialPort, $CommandString)
sub SendTorridonCommand {
        my($Connection, $Command) = @_;
        my $Response = "";
        my $Section = "";
        my $Timeout = 0;
        my $TimeoutLimit = 50;          # 1 = approx 10ms timout delay

        if ($LogText) {
        # Print out the command for logging
                print $Command . "\n";
        }

        # Write the command to the serial port
        $Connection->write($Command . "\r");

        # Discard any echoed characters of the command
        # By waiting for the <CR><LF> return at the end of the command echo
        if ($ScriptMode) {
                $Connection->are_match ("\n");
        }
        else {
                $Connection->are_match ("\r\n");
        }
        until ("" ne $Section || $Timeout > $TimeoutLimit) {
                # Look for the line feed section
                $Section = $Connection->lookfor;

                # Track a 2 second timeout
                select(undef, undef, undef, 0.01);
                $Timeout++;
        }

        # If the line feed was seen, as expected
        if ($Timeout <= $TimeoutLimit) {
                $Connection->are_match (">");
                $Timeout = 0;
                until ("" ne $Response || $Timeout > $TimeoutLimit) {
        # Look for the cursor
                        $Response = $Connection->lookfor;

        # Track a 2 second timeout
                        select(undef, undef, undef, 0.01);
                        $Timeout++;
                }

                if ($LogText) {
        # Print out the full response for logging
        # Add the cursor back as the loop above will have removed it
                        print $Response . ">";
                }

        # If the cursor was NOT seen (response did not end correctly)
                if ($Timeout > $TimeoutLimit) {
                        if ($LogText) {
                                # Log the error
                                print "FAIL: (1)Module timed out without
releasing the bus\n";
                        }

                        return $Response . "\nFAIL: (2)Module timed out
without releasing the bus";
                }
        # Else the command completed fully
                else {
        # Return the response from the module
                        return $Response;
                }
        }
        # Command line feed was not returned
        else {
                if ($LogText) {
        # Log the error
                        print "FAIL: (3)No response from the module\n";
                }
                return "FAIL: (4)No response from the module";
        }
}

1;
```