Sync 'n Stream Documentation

James Williams

September 2020

Contents

1 Introduction

The Sync 'n Stream system implements the sender (Alice) and receiver (Bob) sides of a time-bin encoded quantum key distribution algorithm. The codebase is broken down into three main sections: FPGA firmware managed by the Vivado IDE, C firmware managed by the Vitis IDE, and Python drivers managed by the Spyder IDE. The FPGA and C firmware work in tandem to pass commands from the Python drivers over UART to the pulse generation hardware running on the FPGA. They also signal the FPGA's status back to the Python drivers for closed-loop control.

This document provides an overview of the system architecture and gives pointers on how to use the various library functions. For more detailed information pertaining to specific functions, see the code comments above each function header. The Github link to the codebase is available here.

2 FPGA Firmware

2.1 Architecture Overview

The top level design used to configure the programmable logic (PL) section of the FPGA has five main components: the Zynq processing system (PS), the AXI-GPIO module, the RF module, two general purpose FIFOs, the pulse_gen RTL design, and the gpio_to_fifo RTL design.

The PS module is automatically generated and maintained by the Vivado environment, and defines how the 6 availabe ARM processors connect to the PL. The AXI-GPIO module provides a bridge between the PS and the gpio_to_fifo module, allowing both FIFOs to be controlled from the PS. The gpio_to_fifo module is a simple breakout used to split the 32-bit GPIO bus into individual control lines for the FIFOS. The two FIFOs are responsible for passing commands and encoded photon vales from the PS to the pulse_gen module. They are set up to use two seperate clock domains as the PS and GPIO infrastructure operate in the 100 MHz clock domain while the pulse_gen and RF infrastructure operate in the 250 MHz clock domain. The RF module controls the settings of the RF digital-to-analog converters (DACs), and provides a streaming interface for the pulse_gen module to send pulses to the DACs. An internal logic analyzer is also present and connected to all ports of the pulse_gen module.

2.2 pulse_gen module

The pulse_gen module is a finite state machine which can accept and execute commands via the command FIFO. The high byte of the FIFO output defines the command while the lower 3 bytes are used to pass arguments. A second FIFO, the data FIFO, is used to store a list of encoded pulses which are transmitted via the sync_and_stream command. The pulse_gen module implements the following commands:

- reset_clock: resets the on-board clock used to encode photons.
- send_pulse: sends a single pulse with a specified delay after the next clock tick.
- set_period: sets the period of the on-board clock.
- set_phase_meas_mode: turns on phase measurement mode, generating a 4 ns pulse at each clock tick.
- reset_phase_meas_mode: turns off phase measurement mode.
- toggle_phase_meas_mode: sends a pulse at each clock tick for a user-specified number of ticks.
- sync_and_stream: sends a user-defined number of synchronization pulses, dead pulses (delay between sync and encoded pulses), and all queued encoded pulses in the data FIFO.
- clear_queue: clears and pulses in the data FIFO.
- set_amplitude: sets the amplitude of the output pulse. 0x7FFF is the most positive value, 0x8000 is the most negative value.
- set_pulse_len: sets the pulse length in steps of 250 ps.

Commands are issued from the PS using the gpio_to_fifo module. gpio_to_fifo contains two shift registers used to shift commands and encoded pulses into the FIFOs.

2.3 Setting up the external clock

The FPGA requires an external 250 MHz clock to operate the RF DACs. This clock is provided by the WindFreak RF synthesizer, a 20dB gain amplifier and a balun adapter used to turn the single-ended output of the amplifier into the balanced input required by the FPGA. The output amplitude of the WindFreak should be set to -7 dBm. The output of the balun adapter should be connected to the pair of DAC_CLKIN_3 ports.

2.4 Uploading the PL configuration to the FPGA

After PL firmware is synthesized and implemented, the bit-stream can be uploaded to the FPGA via the "Program and Debug" tab on the bottom left corner of Vivado. This requires that the appropriate Vivado project has been opened. After selecting said tab, select "open target", then "auto-connect", and "program device". If "program device" does not appear, right-click on xczu28dr and select "program device". After the device is programmed, the logic analyzer interface should appear as a black box in the top right quadrant of Vivado. If this interface does not appear, this is an indication that the FPGA is not detecting the requisite 250 MHz input clock.

If the user has re-synthesized the FPGA design, the hardware description files will need to be exported again to the Vitis development environment used for the C firmware. This is done through Vivado by selecting File → Export → Export Hardware and selecting both the "Fixed" and "Include Bitstream" options. After exporting the hardware, the associated hardware project in Vitis must be updated by right-clicking on test_proj_plat in the explorer window, selecting "Update Hardware Specification", and clicking OK. Then right-click on test_project_plat again and select build. If the build fails for any reason, delete test_proj_plat and make an identical project platform using File → New → Platform Project with the same name and try building the project again.

3 C Firmware

3.1 Structure Overview

The purpose of the C firmware is to provide a bridge between the FPGA's pulse_gen module and the Python driver running on the PC. It accepts commands via a UART connection and forwards them to the pulse_gen module. The firmware is also capable of detecting the absence of the 250 MHz clock and alerting the user. All functions and variables in the code base are fully commented. The C firmware is developed, compiled, and uploaded to the FPGA using the Vitis IDE which is launched through the Tools menu in Vivado.

3.1.1 UART Driver

The UART driver is responsible for accepting UART bytes from the Python driver and passing them to the command handler. This driver uses interrupts to receive and store bytes in a small ring buffer as they are received. This buffer is then read out and interpreted by the command handler. Bytes are transmitted via a polled process, so transmissions longer than 1 byte will consume a significant amount of CPU time.

3.1.2 GPIO Driver

The GPIO driver provides a bridge between the C firmware and the pulse_gen module. It is used to control the 32-bit GPIO bus which connects to the gpio_to_fifo module described earlier. The GPIO driver can also reset the pulse_gen module and check if pulse_gen is busy executing a command.

3.1.3 Command Handler

The command handler is a finite state Maine responsible for reading out bytes from the UART ring buffer and executing them as commands. It supports all of the same commands as the pulse_gen module, acting as a software-hardware bridge between the Python driver and pulse_gen. The finite state machine design allows additional service functions to be called within the firmware's main loop by bypassing the need for the CPU to wait on incoming bytes from the Python driver.

3.2 Uploading C Firmware to the FPGA

Before uploading the C firmware, the user must ensure that the firmware itself has been built up-to-date. This can be done by right-clicking test_proj_pulse_ctrl_system in the explorer tab near the right of the screen and selecting "Build Project".

To upload compiled C firmware, right-click on test_proj_pulse_ctrl_system and select Debug As → Debug Configurations and select the configuration ending in no_fpga_flash. This will flash the C firmware to the FPGA without resetting the PL configuration uploaded through Vivado.

4 Python Drivers

4.1 FPGA Wrapper

The FPGA wrapper is defined as the pulse_gen object in the file pulse_gen.py It implements all of the pulse_gen module commands as Python functions, and provides feedback over the terminal to indicate problems with the FPGA configuration or clock. Input parameters are checked by each function. Specific information pertaining to arguments and return values can be found above each function header.

If you plan on using this library as a standalone module in your own custom program, you must call close_board when you are finished sending commands to the FPGA so as to clear the lock on the PC's serial port resource used to communicate with the FPGA. Alternatively you can let your program exit when finished and close_board will be called automatically in the destructor of pulse_gen. If Python gives you any errors pertaining to the serial port, restarting the current Python console should solve the issue.

4.2 TDC Wrapper

The TDC wrapper provides high level functions for the Qutag TDC, allowing users to start, stop, and recover timestamps from the TDC. The TDC driver has three modes: normal, client, and server. In normal mode, a single user is allowed to access, configure, and recover timestamps from the TDC.

Client and server mode work together to allow two users to access the same TDC. The TDC itself will be managed by a TDC wrapper instance running in server mode, with multiple clients connecting over an unsecured socket to issue various commands. Examples on how to use the TDC library can be found in any of the Python files beginning with "tdc". The list of TDC commands is as follows:

- ping: check connection between TDC server and client.
- close_connection: gracefully shutdown socket between server and client.
- clear_all: clears all recorded timestamps from server's timestamp list.
- get_busy: checks if the TDC server is busy offloading timestamps from the TDC.
- get_and_clear: gets the latest timestamp from a specified channel and clears it from the server's timestamp list.
- dump_all: gets all recorded timestamps from a specified channel.
- record_pulses: starts the server's timestamp recording loop and saves any timestamps recorded after the command has been issued.
- stop_record: stops the server from saving timestamps from the TDC. Timestamps will continue to be offloaded and deleted.

4.2.1 Installing TDC Drivers

In order for the TDC Python drivers to work, the 64-bit DLLs found on the QuTag website under the "lib" directory of the zip folder downloaded from here must be in the System32 directory of Windows.

4.2.2 Changing TDC Initialization Settings

The initialization settings for the TDC, including threshold voltage and channels to be enabled, can be found at the top of the james_utils.py file. When using SNSPDs, the threshold voltage should be 0.1 (100mV).

4.3 Time_Sync Object

The time_sync object implements the Python routines required for absolute time synchronization and secure key transmission for both Alice (client) and Bob (server). In server mode, the time_sync object works similar to the TDC server in that it accepts connections from a client and allows the client to execute various commands. The list of commands supported by Bob is as follows:

- send_pulse: sends a single pulse from Bob to Alice, and returns Bob's timestamp to Alice.
- receive_pulse: receives a single pulse from Alice and returns the timestamp.
- close_connection: gracefully closes socket between Alice and Bob.
- exit: shuts down Bob's instance.
- ping: checks the connection between Alice and Bob
- set_bin_size: sets the bin size of the encoding protocol.
- set_bin_number: sets the bin number of the encoding protocol.
- receive_stream: receives a stream of synchronization and encoded pulses from Alice.

4.4 Running the TDC

In most cases, the TDC should be run in server mode to facilitate easy access for both the Bob and Alice sides of the experiment. This can be done by running the script "tdc_server_run.py" in a dedicated Anaconda console. The TDC server can be shut down by pressing enter while in the console.

4.5 Running Bob

Bob can be run in much the same fassion as the TDC. The Bob server should be run by executing the script "bob_test_run.py" in a dedicated Anaconda console. Bob can also be shut down by pressing enter while in the console.

4.6 Running A Key Transmission Experiment

- Step 0: Start Windfreak. Start the Windfreak software "SynthNV Pro" and set to output to 250 MHz at -7 dBm. Ensure that the 12V amplifier is connected between the Windfreak and FPGA clock inputs. The amplifier should connect to the single-ended input of the balun, and the two outputs of the balun should be connected to the "DAC_CLKIN_3" ports.
- Step 1: Opening the Vivado project. Open the Vivado software and then open the project located at C:/James/test_project.
- Step 2: Uploading FPGA firmware. With the Vivado project open, click on "Open Hardware Manager" at the bottom right of the Vivado window. Then go to "Open Target" at the top of the window, "auto-connect", and "Program Device". If the clock has been successfully detected, the "Waveform" window will automatically open after programming is complete. Otherwise a yellow message will appear in the console at the bottom of the screen.
- Step 3: Opening the Vitis project. In the Vivado window, select "Tools" from the top left and go to "Launch Vitis IDE".

- Step 4: Uploading the C firmware. From Vitis, right click on "test_proj_pulse_ctrl_system" at the bottom right of the window. Vitis will hang for several seconds the first time you do this. Then select "Debug as" and "Debug Configurations". In the debug configurations window, select "SystemDebugger_test_proj_pulse_ctrl_system _no_fpga_flash" and select "Debug". Once the debugger is finished launching, navigate to the debug tab at the right of the window and click on the Cortex processor that says "Breakpoint: main". Then click the resume button (yellow bar and green arrow) at the top left of the screen twice. To see the debug printout, select the small monitor icon in the console window near the bottom of the Vitis window and select "CortexA53 #0". If the debugger does not launch correctly, restart the board and try again after uploading the FPGA firmware (Step 2).
- Step 5: Running the Python Script. Launch the script "alice_pulse_stream_test.py" in the Spyder IDE, and launch the scripts tdc_server_run.py and "bob_test_run.py" in seperate Anaconda consoles. The Alice script script will use the Bob and TDC scripts to repeatedly transmit random keys between Bob and Alice, and log all results to a logfile defined near the top of the script.

Parameters such as bin number, bin size, number of synchronization pulses, number of dead pulses, pulse amplitude, Bob's IP address, and the TDC's IP address can be chanced at the top of Alice's file. The TDC's threshold voltage can be changed in the james_utils.py file. This will need to be done when switching to using the SNSPDs. If Bob and the TDC server are running on the same machine as Alice, then these values should be set to your LAN IP found by running the command "ipconfig" in command prompt and reading the field "IPV4 Address".

4.6.1 Running A Time Synchronization Experiment

Due to the presence of dark counts, absolute time synchronization experiments will need to be re-designed and should not be attempted using entangled pairs or the SNSPD.