RHS

USB/FPGA Interface: RhythmStim USB-7310

Version 3.2, 3 March 2023; updated 26 July 2023

Features

- ♦ Open-source Verilog hardware description language (HDL) code configures a Xilinx field-programmable gate array (FPGA) to communicate with multiple RHS digital electrophysiology stimulation/amplifier chips
- ♦ Verilog code is written for the commercially-available Opal Kelly XEM7310 module with integrated SuperSpeed USB 3.0 interface
- ♦ Up to 128 simultaneous stimulator/amplifier channels supported at sample rates up to 30 kS/s/channel
- ♦ Programmable FPGA clock for RHS interface: sample rates of 20, 25, or 30 kS/s/channel supported
- ♦ Open-source host computer application programming interface (API) in C++ for multi-platform support
- ♦ Module can interface with eight 16-bit digital-to-analog converters (DACs) and route selected amplifier channels to selected DACs for analog signal reconstruction or audio monitoring with minimal latency
- ♦ Optional control of eight 16-bit analog-to-digital converters (ADCs) for auxiliary analog inputs synchronized to all RHS amplifier channels
- ♦ Auxiliary digital I/O: 16 digital input lines and 16 digital output lines supported
- ♦ Biphasic and triphasic current pulses generated with timing resolution as fine as 33.3 μs.
- ♦ Independent or coordinated stimulation sequences on all channels triggered by digital inputs or software commands.
- ♦ Analog output ports can generate custom voltage pulses or reconstruct waveforms from selected amplifier channels in real time.
- ♦ Digital output ports can generate custom TTL pulses or act as low latency threshold-based spike detectors.

Applications

- ♦ Windows, Mac, or Linux-based electrophysiology signal acquisition systems
- ♦ Rapid prototyping of Intan Technologies RHS-based products
- ♦ Starting point for the development of custom interfaces to RHS chips

Description

To facilitate the development of electrophysiology interface systems using the RHS series of stimulation/amplifier microchips, Intan Technologies provides the following open-source USB/FPGA interface for developers. Designated **RhythmStim USB-7310**, the interface consists of Verilog HDL code written for the commercially-available Opal Kelly XEM7310 USB/FPGA interface module and a C++ API. RhythmStim USB-7310 configures the Xilinx FPGA on the Opal Kelly module to communicate with up to eight RHS chips over SPI buses and to stream data from these chips to a host computer over a SuperSpeed USB 3.0 interface.

The Opal Kelly drivers and RhythmStim USB-7310 software interfaces are designed for multi-platform development under Windows, Mac, or Linux. All API software is written in C++ to facilitate rapid development. This datasheet provides documentation on the RhythmStim USB-7310 hardware and software protocols so that developers may quickly link the RHS series chips to a host computer of their choice.

RhythmStim USB-7310 supports real-time streaming of up to 128 amplifier channels from multiple RHS chips, data from up to eight other ADCs, and signals from 16 digital inputs. Independent stimulation protocols may be set for all 128 stimulator channels. All data is synchronized and time-stamped before transmission over a standard USB 3.0 bus to the host computer.

RhythmStim USB-7310 FPGA I/O Signals

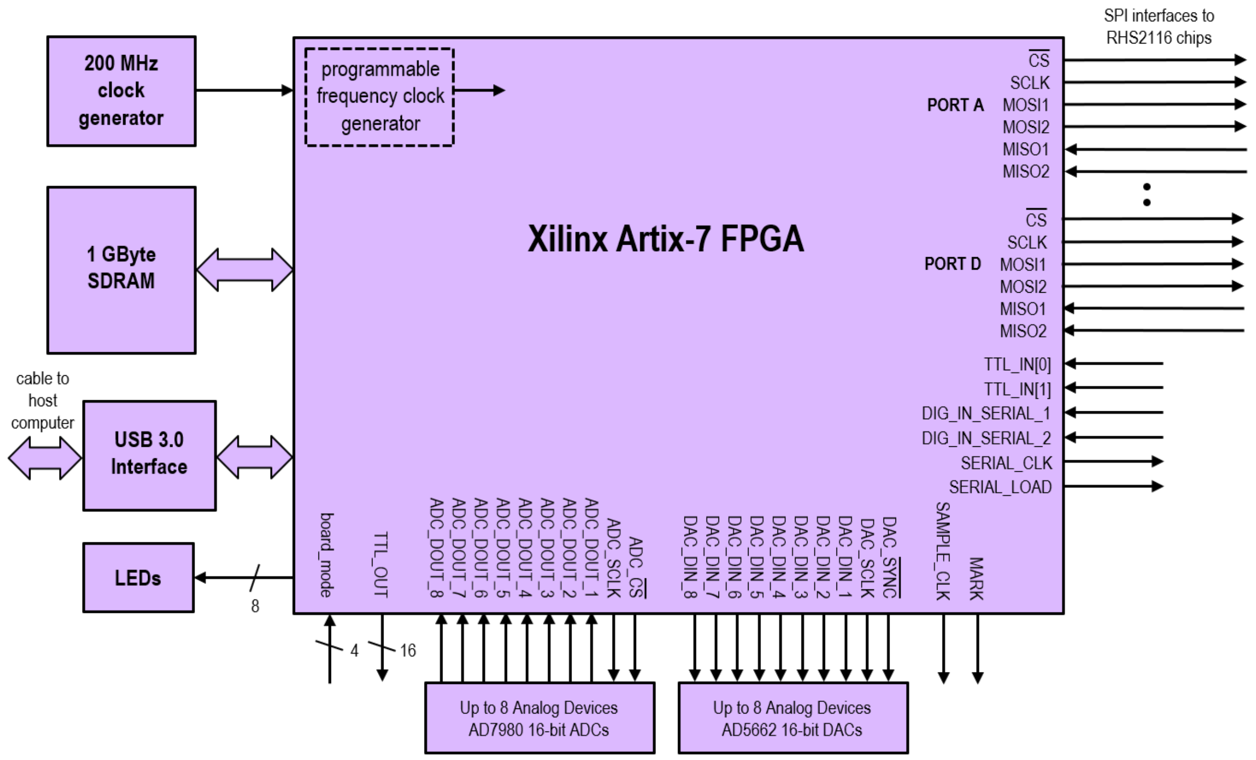
General Description

The RhythmStim USB-7310 interface code is designed for the Opal Kelly XEM7310-A75 USB/FPGA module which is a small commercially-available circuit board containing a Xilinx Artix-7 FPGA (XC7A75T-1C), a 1-GByte SDRAM chip, a 200-MHz clock source, I/O connectors, and a USB 3.0 interface chip capable of streaming data to a host computer at rates exceeding 340 MByte/s. (A photo of the board is shown on page 1 of this datasheet. See www.opalkelly.com for detailed information on this module.)

The Artix-7 FPGA is a digital chip containing hundreds of thousands of configurable logic gates, flip-flops, and memory cells with programmable connections between them. The FPGA is configured by means of a **bitfile**, which is compiled from Verilog HDL (Hardware Description Language) code using the free Xilinx Vivado software. The bitfile must be uploaded to the FPGA through the host computer USB interface every time the board is powered up; it is not stored in non-volatile RAM on the FPGA or the Opal Kelly module. Since this "booting" process takes only a fraction of a second, the flexibility it imparts becomes a useful feature: any changes made to the bitfile can be incorporated into a software release. The FPGA never needs to be programmed using a special EEPROM programmer or JTAG interface.

Opal Kelly also offers the XEM7310-A200 USB/FPGA module which is identical to the XEM7310-A75 except for the use of an XC7A200T-1C FPGA which contains approximately three times as many logic gates as the XC7A75T-1C. Developers who wish to add significant amounts of processing at the FPGA level may wish to use this module and recompile the RhythmStim USB-7310 Verilog code selecting this larger FPGA as the target.

The diagram below shows the main elements on the Opal Kelly XEM7310 board and the I/O signals defined by RhythmStim USB-7310.



RHS SPI Interfaces

RhythmStim USB-7310 sets up four SPI ports (labeled A, B, C, and D) that can send independent command streams to different sets of RHS chips. Unlike the RHD version of Rhythm, LVDS signals are not generated directly by the FPGA, requiring the user to add external CMOS-to-LVDS interfacing chips (e.g., the SN65LVDT41 from Texas Instruments) if LVDS signals will be used to interface with the RHS chips. LVDS signaling is recommended for operation over long cables.

Each SPI port on the FPGA has four output signals that coordinate communication with RHS chips and send commands: $\overline{\text{CS}}$, SCLK, and two MOSI (Master Out, Slave In) signals: MOSI1 and MOSI2. The FPGA always acts as the SPI master device, and each RHS acts as an SPI slave device. Each SPI port in RhythmStim USB-7310 has two MISO (Master In, Slave Out) inputs for receiving data from an RHS chip: MISO1 and MISO2. This means that each port can be connected to two RHS chips and send these chips independent commands. This permits the construction of compact 32-channel stimulation/recording modules using two RHS chips, for example. Each module requires a 16-conductor cable to support six LVDS SPI signals plus +3.3V power, ground, and positive and negative stimulation voltage supplies VSTIM+ and VSTIM–.

The use of LVDS signals permits robust data transfer over cables several meters in length. However, long cables will introduce significant delays to the SPI signals. Typical cable propagation velocities are two-thirds the speed of light – about 20 cm/ns – so a three-meter cable will have a round-trip signal delay of 30 ns. (At high sampling rates, the SCLK period may be less than 50 ns.) RhythmStim USB-7310 allows users to adjust the MISO sampling delay independently on all four SPI ports to account for cable delays.

Using dual MISO and MOSI signals on each of four ports, the FPGA can stream data from up to eight RHS chips, so a total of 128 amplifier channels may be acquired in real time. The USB 3.0 interface on the Opal Kelly XEM7310 module is capable of reliably streaming data from 128 channels to a host computer at per-channel sampling rates up to 30 kS/s, even with typical USB protocol overhead and software overhead.

RhythmStim USB-7310 sends commands to all RHS chips in a repeating 20-command sequence: every sampling period, the FPGA sends the commands CONVERT(0), CONVERT(1), CONVERT(2), and so on up to CONVERT(15) to sample from all 16 amplifiers on the chip. These 16 CONVERT commands are followed by four “auxiliary” commands that may be used to perform a variety of other tasks on an RHS chip, like controlling stimulation pulses, activating amplifier fast recovery circuits, synthesizing a waveform for electrode impedance testing, or reading and writing other registers. Under RhythmStim USB-7310, the first 16 CONVERT commands are fixed, but the remaining auxiliary commands may be programmed from the host computer.

A clock generator on the Opal Kelly circuit board provides a 200-MHz clock to the FPGA; this clock is used to run the USB interface logic. RhythmStim USB-7310 controls an on-FPGA frequency synthesizer that is used to generate a programmable-frequency clock that can be reconfigured by the host computer to produce many different SPI data rates so that the RHS amplifiers can be sampled at per-channel rates of 20, 25, or 30 kS/s. The minimum timing resolution of stimulation pulses is equal to the reciprocal of the per-channel sampling rate, so with a sampling rate of 30 kS/s, the minimum stimulation timing resolution is 33.3 μs.

Other I/O Signals

RhythmStim USB-7310 also defines I/O pins that can be connected to several commercially-available 16-bit DACs and ADCs. The Verilog code supports up to eight 16-bit DACs and eight 16-bit ADCs that communicate using a three-wire SPI interface. RhythmStim USB-7310 is designed to support the Analog Devices AD5662 DAC and the Analog Devices AD7980 ADC; other DACs and ADCs could be supported by modifying the Verilog code. RhythmStim USB-7310 permits the user to select particular RHS amplifier channels to be routed directly to selected DACs through the FPGA, eliminating any USB or host computer latency. Analog output ports can also be configured to generate custom voltage pulses. Alternatively, any of the DACs may be controlled by a dedicated data stream from the host computer if more latency can be tolerated.

The optional ADCs are sampled in synchrony with the RHS amplifiers, and their results are streamed back to the host computer over the USB interface.

RhythmStim USB-7310 also supports 16 digital inputs and 16 digital outputs. Although these are labeled as “TTL” I/O, the FPGA pins operate at 3.3V, but most input pins are tolerant of 5V signals. Two digital inputs are sampled directly; more can be implemented using 74HC165 shift register integrated circuits (contact Intan Technologies for more information). Digital and analog inputs may be used to trigger RHS stimulation pulses.

Eight of the digital outputs may serve as low-latency threshold comparators for the signals routed to the DACs. Threshold levels and polarities may be specified, and the FPGA will trigger the appropriate digital outputs if each threshold is exceeded.

The Opal Kelly board has an array of eight red LEDs (in addition to a green power LED) that may be controlled by the host computer. Additional FPGA pins are allocated for the control of eight SPI port LEDs and three general-purpose status LEDs.

A digital output signal **sample_clk** is provided on FPGA pin B35_L21P as a convenience. This signal is a clock running at the per-channel amplifier sampling rate. The duty cycle of the signal is 1/20. For example, if the board is configured to sample each RHS amplifier channel at 20 kS/s then the period of this clock will be 50 µs. It will be high for 2.5 µs, and low the rest of the cycle. (The signal goes high at the falling edge of $\overline{\text{CS}}$ that begins the CONVERT(0) command, and it goes low at the falling edge of $\overline{\text{CS}}$ that begins the CONVERT(1) command.)

Power Supply

The total current consumption of the XEM7310 board running RhythmStim USB-7310 is approximately 500 mA from 5V, not including current consumed by RHS chips and other ADCs and DACs. Current supplied from a USB port is limited to 500 mA. Since a RhythmStim USB-7310-based module will likely exceed this limit once RHS chips and other ADCs and DACs are added, it is recommended to power the Opal Kelly XEM7310 board from a 5V DC power supply capable of sourcing sufficient current and having a 2.1mm inner-diameter / 5.5mm outer-diameter connector. Digi-Key (www.digikey.com) offers a medical-rated power supply that mates with the XEM7310 board, supplies up to 2.0 A of current, and includes connectors for AC plugs in a variety of countries (part number T1233-P5P-ND).

The Opal Kelly board provides regulated 1.0V, 1.8V, and 3.3V supplies. See the Opal Kelly documentation for current limits on these supplies.

If RHS chips will be powered over long cables, it is a good idea use a linear regulator to generate a 3.5V power supply from the 5V power delivered to the board. RHS chips may be safely powered at 3.5V, and the excess 0.2 V above the nominal power supply voltage of 3.3V will make up for some of the IR losses incurred over long cables with thin power wires.

I/O Pin Locations

The I/O pin tables on the following four pages are based information in the Opal Kelly XEM7310 User's Manual. Due to reuse of some circuit board design layouts at Intan Technologies, there are naming inconsistencies with the SPI port I/O pins as noted in these tables. The pins for SPI Port A are listed as CS_A, SCLK_A, MOSI_A1, etc., as expected. However, the pins for SPI Port B are listed as CS_C, SCLK_C,… The pins for SPI Port C are listed as CS_E, SCLK_E,… The pins for SPI Port D are listed as CS_G, SCLK_G,… The pins ending in _B, _D, _F, and _H are not used in RhythmStim USB-7310. (These pins are used in Rhythm USB-7310, which is designed to operate the RHD recording-only chips. Some circuit boards used at Intan Technologies are designed to work with FPGA modules running either interface API. RhythmStim USB-7310 uses alternating ports due to the physical width of SPI interface daughterboards that are added to the motherboard.)

There are four pins labeled board_mode[0..3] which are configured as inputs. Our RHX data acquisition software requires these pins to be set to 14 (board_mode[3] = 1, board_mode[2] = 1, board_mode[1] = 1, board_mode[0] = 0) to recognize an XEM7310 as an RHS stimulation/recording controller.

Users may wish to purchase the BRK7010 breakout board which can be connected to the XEM7310 module, and which brings the I/O pins out to convenient 2mm-pitch headers.

FPGA pin names come in pairs of positive and negative signals that can be used as LVDS pairs (e.g., B35_L19P and B35_L19N are I/O pin pairs in FPGA Bank 35). An inspection of the **xem7310.xdc** file provided with RhythmStim USB-7310 shows that the IOSTANDARD LVDS_25 property is used for LVDS signals, while IOSTANDARD LVCMOS_33 is used for standard CMOS signals.

MC1 I/O Connections – Odd Pins

MC1 Pin	Connection	FPGA Pin	RhythmStim USB-7310 Interface Pin Name
1	+VDCIN		+5V
3	+VDCIN		+5V
5	+VDCIN		+5V
7	+1.8VDD		
9	+3.3VDD		VDD
11	+3.3VDD		VDD
13	+3.3VDD		VDD
15	W9	B34_L24P	ADC_CS
17	Y9	B34_L24N	DAC_SCLK
19	R6	B34_L17P	DAC_SYNC
21	T6	B34_L17N	DAC_DIN_8
23	U6	B34_L16P	unused
25	V5	B34_L16N	unused
27	T5	B34_L14P	DAC_DIN_7
29	U5	B34_L14N	DAC_DIN_6
31	AA5	B34_L10P	unused
33	D17	B34_L10N	unused
35	DGND		GND
37	AB7	B34_L20P	DAC_DIN_5
39	AB6	B34_L20N	DAC_DIN_4
41	R3	B34_L3P	unused
43	R2	B34_L3N	unused
45	Y3	B34_L9P	unused
47	AA3	B34_L9N	unused
49	U2	B34_L2P	unused
51	V2	B34_L2N	DAC_DIN_3
53	W2	B34_L4P	DAC_DIN_2
55	DGND		GND
57	Y2	B34_L4N	DAC_DIN_1
59	T1	B34_L1P	ADC_DOUT_2
61	U1	B34_L1N	ADC_DOUT_1
63	AA1	B34_L7P	DIG_IN_SERIAL_2
65	AB1	B34_L7N	DIG_IN_SERIAL_1
67	AB16	B13_L2P	EXP_ID
69	AB17	B13_L2N	EXP_DETECT
71	AA15	B13_L4P	STATUS_LED_3
73	AB15	B13_L4N	STATUS_LED_2
75	Y16	B13_L1P	STATUS_LED_1
77	V4	B34_L12P_MRCC	unused
79	W4	B34_L12N_MRCC	unused

MC1 I/O Connections – Even Pins

MC1 Pin	Connection	FPGA Pin	RhythmStim USB-7310 Interface Pin Name
2	DGND		GND
4	+1.0VDD		
6	+1.0VDD		
8	AB11	SYS_CLK_MC1	
10	M9	XADC_VN	
12	L10	XADC_VP	
14	DGND		GND
16	V9	B34_L21P	ADC_SCLK
18	V8	B34_L21N	TTL_OUT[0]
20	V7	B34_L19P	TTL_OUT[1]
22	W7	B34_L19N	TTL_OUT[2]
24	Y8	B34_L23P	TTL_OUT[3]
26	Y7	B34_L23N	TTL_OUT[4]
28	W6	B34_L15P	TTL_OUT[5]
30	W5	B34_L15N	TTL_OUT[6]
32	R4	B34_L13P	board_mode[0]
34	T4	B34_L13N	board_mode[1]
36	VCCO_MC1		
38	Y4	B34_L11P	board_mode[2]
40	AA4	B34_L11N	board_mode[3]
42	Y6	B34_L18P	TTL_OUT[7]
44	AA6	B34_L18N	TTL_OUT[8]
46	AA8	B34_L22P	TTL_OUT[9]
48	AB8	B34_L22N	TTL_OUT[10]
50	U3	B34_L6P	TTL_OUT[11]
52	V3	B34_L6N	TTL_OUT[12]
54	W1	B34_L5P	TTL_OUT[13]
56	VCCO_MC1		
58	Y1	B34_L5N	TTL_OUT[14]
60	AB3	B34_L8P	TTL_OUT[15]
62	AB2	B34_L8N	ADC_DOUT_8
64	Y13	B13_L5P	ADC_DOUT_7
66	AA14	B13_L15N	ADC_DOUT_6
68	AA13	B13_L3P	ADC_DOUT_5
70	AB13	B13_L3N	ADC_DOUT_4
72	W15	B13_L16P	ADC_DOUT_3
74	W16	B13_L16N	TTL_IN[1]
76	AA16	B13_L1N	TTL_IN[0]
78	DGND		GND
80	DGND		GND

MC2 I/O Connections – Odd Pins

MC2 Pin	Connection	FPGA Pin	RhythmStim USB-7310 Interface Pin Name
1	DGND		GND
3	+VCCBAT		
5	FPGA_TCK		
7	FPGA_TMS		
9	FPGA_TDI		
11	SYS_CLK_MC2	AB12	
13	DGND		GND
15	P5	B35_L21P	SAMPLE_CLK
17	P4	B35_L21N	MARK
19	N4	B35_L19P	SERIAL_LOAD
21	N3	B35_L19N	SERIAL_CLK
23	L5	B35_L18P	SPI_LED_H
25	L4	B35_L18N	SPI_LED_G
27	M6	B35_L23P	SPI_LED_F
29	M5	B35_L23N	SPI_LED_E
31	M1	B35_L15P	SPI_LED_D
33	L1	B35_L15N	SPI_LED_C
35	VCCO_MC2		
37	K2	B35_L9P	SPI_LED_B
39	J2	B35_L9N	SPI_LED_A
41	K1	B35_L7P	MISO_H2 (not used)
43	J1	B35_L7N	MISO_H1 (not used)
45	H3	B35_L11P	MOSI_H2 (not used)
47	G3	B35_L11N	MOSI_H1 (not used)
49	E2	B35_L4P	SCLK_H (not used)
51	D2	B35_L4N	$\overline{\text{CS}}$_H (not used)
53	F3	B35_L6P	MISO_G2 (Port D)
55	VCCO_MC2		
57	E3	B35_L6N	MISO_G1 (Port D)
59	B1	B35_L1P	MOSI_G2 (Port D)
61	A1	B35_L1N	MOSI_G1 (Port D)
63	K4	B35_L13P	SCLK_G (Port D)
65	J4	B35_L13N	$\overline{\text{CS}}$_G (Port D)
67	T16	B13_L17P	MISO_F2 (not used)
69	U16	B13_L17N	MISO_F1 (not used)
71	V13	B13_L13P	MOSI_F2 (not used)
73	V14	B13_L13N	MOSI_F1 (not used)
75	Y11	B13_L11P	SCLK_F (not used)
77	H4	B35_L12P_MRCC	$\overline{\text{CS}}$_F (not used)
79	G4	B35_L12N_MRCC	MISO_E2 (Port C)

MC2 I/O Connections – Even Pins

MC2 Pin	Connection	FPGA Pin	RhythmStim USB-7310 Interface Pin Name
2	+3.3VDD		VDD
4	+3.3VDD		VDD
6	+3.3VDD		VDD
8	FPGA_TDO		
10	F4	B35_IO0	
12	L6	B35_IO25	
14	DGND		GND
16	P6	B35_L24P	$\overline{\text{CS}}$_A (Port A)
18	N5	B35_L24N	SCLK_A (Port A)
20	P2	B35_L22P	MOSI_A1 (Port A)
22	N2	B35_L22N	MOSI_A2 (Port A)
24	R1	B35_L20P	MISO_A1 (Port A)
26	P1	B35_L20N	MISO_A2 (Port A)
28	M3	B35_L16P	$\overline{\text{CS}}$_B (not used)
30	M2	B35_L16N	SCLK_B (not used)
32	K6	B35_L17P	MOSI_B1 (not used)
34	J6	MOSI2_B	MOSI_B2 (not used)
36	DGND		GND
38	L3	B35_L14P	MISO_B1 (not used)
40	K3	B35_L14N	MISO_B2 (not used)
42	J5	B35_L10P	$\overline{\text{CS}}$_C (Port B)
44	H5	B35_L10N	SCLK_C (Port B)
46	H2	B35_L8P	MOSI_C1 (Port B)
48	G2	B35_L8N	MOSI_C2 (Port B)
50	G1	B35_L5P	MISO_C1 (Port B)
52	F1	B35_L5N	MISO_C2 (Port B)
54	E1	B35_L3P	$\overline{\text{CS}}$_D (not used)
56	DGND		GND
58	D1	B35_L3N	SCLK_D (not used)
60	C2	B35_L2P	MOSI_D1 (not used)
62	B2	B35_L2N	MOSI_D2 (not used)
64	U15	B13_L14P	MISO_D1 (not used)
66	V15	B13_L14N	MISO_D2 (not used)
68	T14	B13_L15P	$\overline{\text{CS}}$_E (Port C)
70	T15	B13_L15N	SCLK_E (Port C)
72	W14	B13_L6P	MOSI_E1 (Port C)
74	Y14	B13_L6N	MOSI_E2 (Port C)
76	Y12	B13_L11N	MISO_E1 (Port C)
78	DGND		GND
80	DGND		GND

General Description of Interface Operation

Host Computer Interface

Most electrophysiology recording applications require that data is sampled at a steady rate for long periods of time. To interface this steady stream of data with a host computer that uses a modern, multitasking operating system requires a hardware FIFO (First In, First Out) buffer to store data during brief intervals while the computer is busy performing other tasks. Luckily, the Opal Kelly board includes a 1 GByte SDRAM chip that can be used for just such a purpose (though only 128 Mbytes are used, for consistency with earlier versions of RhythmStim). The RhythmStim USB-7310 code implements an SDRAM-based FIFO as a circular queue of 16-bit words. Three on-FPGA "mini-FIFOs" regulate the flow of data into and out of the main SDRAM FIFO. The output of the FIFO is connected to a "BTPipeOut" endpoint for transfer across the USB port to the host computer (see details below).

The FIFO is capable of reporting the number of 16-bit words it is currently holding. There is no mechanism in the FIFO to protect against underflow. That is, if the computer tries to read more data than the FIFO is currently holding, the FIFO will just repeat the last word after it runs out of data. To prevent underflow, it is essential for the host to monitor the amount of data in the FIFO and never attempt to read more words than the FIFO contains.

Neither is there a mechanism in the FIFO to protect against overflow. If the FIFO fills up, it will "lap" the unread data and begin writing over old data in the SDRAM. So the host computer must monitor the number of words in the FIFO and make sure it doesn't get too full. The RhythmStim USB-7310 FIFO can hold 2^{27} = 134,217,728 bytes, or 2^{26} = 67,108,864 16-bit words. The on-FPGA "mini-FIFOs" add a few hundred thousand more words to this total, but it is good practice never to allow the FIFO to get more than 75% full in case the computer OS hangs for a moment.

In order to completely "clean out" the FIFO after pausing or stopping the flow of data into it, it is necessary to always write an integer multiple of **four** 16-bit words to the FIFO. If there are one, two, or three 16-bit words of data remaining in the input mini-FIFO, they will not be read into the SDRAM (and passed to the output mini-FIFO and thence the USB bus) after the flow of data from the source has stopped. The RhythmStim USB-7310 Verilog code is structured to ensure that data frames are always integer multiples of four 16-bit words.

Communication with the SDRAM chip must be done in 256-bit chunks, so if only 15 16-bit words are written to the FIFO, they will not pass through the SDRAM until a final 16-bit word is written. In order to fully flush the FIFO, it necessary to always write an integer multiple of 16 16-bit words to the FIFO. The RhythmStim USB-7310 Verilog code includes a 4-bit counter that increments with every word written into the FIFO to track how many remainder words are present and writes the necessary number of dummy words when pausing or stopping the flow of data to ensure the last words are included in a full 256-bit chunk.

The RhythmStim USB-7310 interface is capable of transmitting up to eight simultaneous **data streams**. In RhythmStim USB-7310, a data stream is defined as the SPI output of one MISO line from an RHS chip. In most cases, the eight data streams correspond to the eight MISO inputs to the FPGA in Ports A, B, C, and D. Each data stream conveys 16 amplifier channels. Data streams may be disabled if they are not used; this will reduce the amount of data flowing through the FIFO and across the USB interface to the computer.

Data Frame Format

A data frame of identical format is transmitted to the FIFO once per amplifier sampling period. The size of the data frame depends on the number of data streams that are enabled. The data frame has the following structure:

64-bit header: a "magic number" always equal to 0x8D542C8A49712F0B that can be used to check for data synchrony.

32-bit timestamp: a 32-bit counter that starts at zero and increments by one every data frame.

32-bit MISO result 1 from data stream 1 (if data stream 1 is enabled)
32-bit MISO result 1 from data stream 2 (if data stream 2 is enabled)
32-bit MISO result 1 from data stream 3 (if data stream 3 is enabled)
32-bit MISO result 1 from data stream 4 (if data stream 4 is enabled)
32-bit MISO result 1 from data stream 5 (if data stream 5 is enabled)
32-bit MISO result 1 from data stream 6 (if data stream 6 is enabled)
32-bit MISO result 1 from data stream 7 (if data stream 7 is enabled)
32-bit MISO result 1 from data stream 8 (if data stream 8 is enabled)

32-bit MISO result 2 from data stream 1 (if data stream 1 is enabled)

32-bit MISO result 2 from data stream 2 (if data stream 2 is enabled)
32-bit MISO result 2 from data stream 3 (if data stream 3 is enabled)
32-bit MISO result 2 from data stream 4 (if data stream 4 is enabled)
32-bit MISO result 2 from data stream 5 (if data stream 5 is enabled)
32-bit MISO result 2 from data stream 6 (if data stream 6 is enabled)
32-bit MISO result 2 from data stream 7 (if data stream 7 is enabled)
32-bit MISO result 2 from data stream 8 (if data stream 8 is enabled)

…

32-bit MISO result 20 from data stream 1 (if data stream 1 is enabled)
32-bit MISO result 20 from data stream 2 (if data stream 2 is enabled)
32-bit MISO result 20 from data stream 3 (if data stream 3 is enabled)
32-bit MISO result 20 from data stream 4 (if data stream 4 is enabled)
32-bit MISO result 20 from data stream 5 (if data stream 5 is enabled)
32-bit MISO result 20 from data stream 6 (if data stream 6 is enabled)
32-bit MISO result 20 from data stream 7 (if data stream 7 is enabled)
32-bit MISO result 20 from data stream 8 (if data stream 8 is enabled)

16-bit stimulation on/off present value from data stream 1 (if data stream 1 is enabled)
16-bit stimulation on/off present value from data stream 2 (if data stream 2 is enabled)
16-bit stimulation on/off present value from data stream 3 (if data stream 3 is enabled)
16-bit stimulation on/off present value from data stream 4 (if data stream 4 is enabled)
16-bit stimulation on/off present value from data stream 5 (if data stream 5 is enabled)
16-bit stimulation on/off present value from data stream 6 (if data stream 6 is enabled)
16-bit stimulation on/off present value from data stream 7 (if data stream 7 is enabled)
16-bit stimulation on/off present value from data stream 8 (if data stream 8 is enabled)

16-bit stimulation polarity present value from data stream 1 (if data stream 1 is enabled)
16-bit stimulation polarity present value from data stream 2 (if data stream 2 is enabled)
16-bit stimulation polarity present value from data stream 3 (if data stream 3 is enabled)
16-bit stimulation polarity present value from data stream 4 (if data stream 4 is enabled)
16-bit stimulation polarity present value from data stream 5 (if data stream 5 is enabled)
16-bit stimulation polarity present value from data stream 6 (if data stream 6 is enabled)
16-bit stimulation polarity present value from data stream 7 (if data stream 7 is enabled)
16-bit stimulation polarity present value from data stream 8 (if data stream 8 is enabled)

16-bit amplifier settle on/off present value from data stream 1 (if data stream 1 is enabled)
16-bit amplifier settle on/off present value from data stream 2 (if data stream 2 is enabled)
16-bit amplifier settle on/off present value from data stream 3 (if data stream 3 is enabled)
16-bit amplifier settle on/off present value from data stream 4 (if data stream 4 is enabled)
16-bit amplifier settle on/off present value from data stream 5 (if data stream 5 is enabled)
16-bit amplifier settle on/off present value from data stream 6 (if data stream 6 is enabled)
16-bit amplifier settle on/off present value from data stream 7 (if data stream 7 is enabled)
16-bit amplifier settle on/off present value from data stream 8 (if data stream 8 is enabled)

16-bit charge recovery on/off present value from data stream 1 (if data stream 1 is enabled)
16-bit charge recovery on/off present value from data stream 2 (if data stream 2 is enabled)
16-bit charge recovery on/off present value from data stream 3 (if data stream 3 is enabled)
16-bit charge recovery on/off present value from data stream 4 (if data stream 4 is enabled)
16-bit charge recovery on/off present value from data stream 5 (if data stream 5 is enabled)
16-bit charge recovery on/off present value from data stream 6 (if data stream 6 is enabled)
16-bit charge recovery on/off present value from data stream 7 (if data stream 7 is enabled)
16-bit charge recovery on/off present value from data stream 8 (if data stream 8 is enabled)

16-bit DAC 1 present value
16-bit DAC 2 present value
16-bit DAC 3 present value
16-bit DAC 4 present value

16-bit DAC 5 present value
16-bit DAC 6 present value
16-bit DAC 7 present value
16-bit DAC 8 present value

16-bit ADC 1 result
16-bit ADC 2 result
16-bit ADC 3 result
16-bit ADC 4 result
16-bit ADC 5 result
16-bit ADC 6 result
16-bit ADC 7 result
16-bit ADC 8 result

16-bit TTL input result

16-bit TTL output present value

This format ensures that each data frame contains an integer multiple of four 16-bit words, as required by the FIFO (see previous section). Each data frame contains (44·N + 24) words or 2·(44·N + 24) bytes, where N is the number of data streams that are enabled. The Opal Kelly USB interface sends data in bytes. In RhythmStim USB-7310, the least-significant byte of each multi-byte word is always sent first.

If we know the size of the data frame and the sampling rate, we can calculate the capacity of the FIFO. For example, a 32-channel system (N = 2) will have a data frame 224 bytes in length. Running at the maximum rate of 30 kS/s, the data rate will be 6.72 MByte/s. At this rate, the FIFO can hold up to 20 seconds of data. A 128-channel system (N = 8) will have a data frame 752 bytes in length. Running at the maximum rate of 30 kS/s, the data rate will be 22.56 MByte/s. At this rate, the FIFO can hold up to 5.9 seconds of data. These calculations demonstrate that the FIFO has plenty of capacity to handle typical brief operating system pauses during USB data transfers.

The Opal Kelly USB interface can easily handle 22.56 MByte/s data rates, provided a reasonably fast host computer is used. A Dell XPS 8930 Windows 10 PC purchased in 2020 with a 3.0 GHz Intel i7-9700 processor routinely achieved maximum USB data rates exceeding 29 MByte/s with the Opal Kelly XEM7310 board and C++ software. Data rates of 35-38 MByte/s for USB 3.0 are not reached unless very large packet transfers (1 MByte or greater) are executed, which would lead to significant latency in most applications.

The bulk of each data frame consists of MISO results received from the SPI interface from each RHS chip. These MISO results are numbered 1 through 20, and correspond to the 20 repeated commands sent over the MOSI lines to each device during each sampling period. The repeating MOSI command structure is as follows:

CONVERT(0)
CONVERT(1)
CONVERT(2)

…

CONVERT(15)
auxiliary command 1
auxiliary command 2
auxiliary command 3
auxiliary command 4

It is important to remember that the RHS uses a pipelined communication protocol; each command sent over the MOSI line generates a 32-bit result that is transmitted over the MISO line two commands later (see the "SPI Command Words" section of the RHS2116 datasheet for details). The FPGA introduces another one-command pipeline delay in the received MISO results, so **every MISO result received through the RhythmStim USB-7310 FPGA corresponds to the MOSI command sent three steps earlier**. Thus "MISO result 1" is the result of the "auxiliary command 2" from the previous sampling period; "MISO result 4" corresponds to the "CONVERT(0)" command from the current sampling period; and "MISO result 20" corresponds to the "auxiliary command 1" command from the current sampling period. To see the result of auxiliary commands 2-4 at the end of a sampling period, the next data frame must be read.

Programming Auxiliary Command Sequences

Each RHS SPI port (A, B, C, and D) can send a different set of commands in the four auxiliary command slots described above, which we will abbreviate as **auxcmd1**, **auxcmd2**, **auxcmd3**, and **auxcmd4**. Sequences of auxiliary commands can be uploaded into on-FPGA RAM banks that each hold up to 8192 commands for each of the four slots. These command sequences will be transmitted over both MOSI lines from each port, but **WireInAuxEnable** (0x0C; see below) can be used to mask off particular MOSI lines so that individual RHS chips can be sent unique command sequences (e.g., for setting stimulation current amplitudes on different headstages).

A concrete example will make things clearer: Let's assume that we always use the **auxcmd1** slot for updating the RHS on-chip DAC used to generate waveforms for electrode impedance testing (i.e., **Zcheck DAC** in Register 3). We can construct a list of commands of the form WRITE(6, x) that generate a sine wave at a particular frequency and amplitude. We can upload this sequence of commands to a RAM bank on the FPGA that is associated with the **auxcmd1** slot, as long as the sequence has 8192 or fewer commands.

For each auxiliary command slot (**auxcmd1**, **auxcmd2**, **auxcmd3**, and **auxcmd4**), the user must specify the length of the command sequence. This **command sequence length** is the same for Ports A, B, C, and D. When data acquisition is started, all auxiliary command sequences start at the beginning of their selected RAM banks and increment to the next command every sampling period. When the command index reaches the specified command sequence length, the command index resets to a **command loop index**. In many cases, the command loop index will be set to zero so that the entire command sequence repeats in its entirety (e.g., the DAC waveform generator commands). However, the command loop index may be set to a number greater than zero so that the first part of the command sequence is executed only once, and a later sequence is executed in a loop. This may be used, for example, to initialize the registers on the chip only once at the beginning of data acquisition.

Any changes made in the command sequence length or the command loop index only take effect when the current command sequence reaches its end, or when the SPI interface is stopped and restarted.

Automatic Stimulation Command Mode

Once the registers on all RHS chips have been initialized, the FPGA can be put into **automatic stimulation command mode** (see **WireInStimCmdMode** below). In this mode, auxiliary command slots are controlled by the Stimulation Sequencer state machines described below. These state machines generate independent stimulation commands for each channel on every chip, and can be triggered by a variety of hardware and software events. The duration and shape of each stimulation pulse can be defined by programming registers in these Sequencer state machines.

If automatic stimulation command mode is enabled, the first auxiliary command slot is always a WRITE command to Register 42, which turns individual current stimulators on or off for all 16 channels on each chip. The second auxiliary command is always a WRITE command to Register 44, which sets the stimulation polarity for all 16 channels.

The third auxiliary command can be a WRITE command to Register 10 or 12 to activate or deactivate amplifier settling for fast artifact recovery. However, if the FPGA detects that no channels on a particular chip need amplifier settling parameters to be changed at a particular time then this command slot is used to read from the compliance monitor (Register 40) to detect voltage compliance problems.

The fourth auxiliary command slot is used to WRITE to Register 46 or 48 to activate or deactivate charge recovery circuitry for each channel. This command always has its U flag set to trigger a simultaneous update of the triggered registers that were updated in the last four commands. If the previous command was reading from the compliance monitor, then the M flag is also set in order to reset the compliance monitor.

Detailed Description of Interface Operation

Opal Kelly provides platform-specific driver files (e.g., a DLL file for Windows) and a C++ API (application programming interface) that defines software **endpoints** used to communicate with XEM boards. **WireIn** and **TriggerIn** endpoints transfer information from the host computer to the FPGA, while **WireOut** and **BTPipeOut** endpoints transfer information from the FPGA to the host computer. Opal Kelly makes all drivers and API software available for download with the purchase of an XEM board. In the sections below, we list all the Opal Kelly endpoints used by RhythmStim USB-7310, and we provide sample C++ code for communicating over these endpoints to a RhythmStim USB-7310-powered XEM7310 board.

Opening and Initializing the Opal Kelly Board

The following C++ code shows how an Opal Kelly XEM7310 board is opened and initialized. Additional code should be added to check for errors and other exceptions (e.g., board not connected, multiple boards connected). The **ConfigRHSController_7310.bit** bitfile contains the compiled RhythmStim USB-7310 Verilog code provided by Intan Technologies. After this bitfile is uploaded to the XEM7310 board, it behaves as the RhythmStim USB-7310 interface described in this document.

```
okCFrontPanel* dev;

dev = new okCFrontPanel;

// If only one Opal Kelly board is plugged in to the host computer, we can use this.
dev->OpenBySerial();

// Set XEM7310 PLL to default configuration to produce 100 MHz FPGA clock.
dev->LoadDefaultPLLConfiguration();

// Upload RhythmStim USB-7310 bitfile which is compiled from RhythmStim USB-7310
 // Verilog code.
dev->ConfigureFPGA("ConfigRHSController_7310.bit");
```

USB Software Endpoints: WireIn and TriggerIn Ports

USB-7310 WireIn ports are virtual 32-bit wires that transfer data asynchronously from the host computer to the FPGA. Up to 32 WireIn ports (with addresses from 0x00 to 0x1F) are available for use with each Opal Kelly module. The following C++ code shows how data is sent over WireIn ports.

```
okCFrontPanel* dev;
// Code to open Opal Kelly board not shown here.

// Set WireIn port 0x00 to 0x12345678.
dev->SetWireInValue(0x00, 0x12345678);

// Optional third term is a bit mask; this command sets bits 0 and 1 of
// WireIn port 0x01 to one, sets bits 2 and 3 to zero, and leaves bits 4-31
// unchanged.
dev->SetWireInValue(0x01, 0x00000003, 0x0000000F);

// WireIn ports are only updated on the FPGA when UpdateWireIns() is called.
// Here, WireIn ports 0x00 and 0x01 are updated simultaneously.
dev->UpdateWireIns();
```

TriggerIn ports are virtual 32-bit wires that transfer digital one-shot pulses from the host computer to the FPGA. Up to 32 TriggerIn ports (with addresses from 0x40 to 0x5F) are available for use with each module. The following C++ code shows how pulses are sent over TriggerIn ports.

```
okCFrontPanel* dev;
// Code to open Opal Kelly board not shown here.

// Send one-shot pulse on bit 0 of TriggerIn port 0x40.
dev->ActivateTriggerIn(0x40, 0);

// Now send one-shot pulse on bit 7 of TriggerIn port 0x40.
dev->ActivateTriggerIn(0x40, 7);
```

WireIn 0x00: WireInResetRun
TriggerIn 0x41: TrigInSpiStart

WireInResetRun[0]: **reset**

When the Opal Kelly board is first powered up and RhythmStim USB-7310 is uploaded, this **reset** signal should be brought high momentarily to initialize many internal registers and finite state machines. This **reset** signal should then remain low for the remainder of RhythmStim USB-7310 operation. Pulling this signal high resets the sampling rate to its default value of 30 kS/s/channel and clears all command RAM banks.

WireInResetRun[1]: **SPI_run_continuous**

When this bit is set high, data acquisition will run continuously once it is started by pulsing **TrigInSpiStart[0]**. To halt data acquisition immediately, set this bit to zero and set **WireInMaxTimeStep** (WireIns 0x01 and 0x02) to zero. If this bit is set low, data acquisition will only run for a finite number of samples set by **MaxTimeStep**.

WireInResetRun[2]: **DSP_settle**

When this bit is set high, the LSB of all CONVERT commands sent to on all MOSI lines is set to one, settling the RHS digital offset removal filters. (See the RHS2116 datasheet for more information on this function.)

WireInResetRun[3]: **amp_settle_mode**

When this bit is set low, fast artifact recovery is performed using low frequency cutoff shifting (Register 12 on the RHS). This is the recommended method. When this bit is set high, fast artifact recovery is performed using the “fast settle” function (Register 10 on the RHS).

WireInResetRun[4]: **charge_recov_mode**

When this bit is set low, charge recovery is performed using charge-limited charge recovery drivers (Register 48 on the RHS). When this bit is set high, charge recovery is performed using the charge recovery switches (Register 46 on the RHS).

WireInResetRun[5]: unused

WireInResetRun[12:6]: **DAC_noise_slice**

The first two DACs are designed to be connected to audio left and right channels. This variable slices out the central +/-16 x **DAC_noise_slice** LSBs of the signals in these two DACs and shifts the remaining signal up or down to zero, improving audibility of neural spikes.

WireInResetRun[15:13]: **DAC_gain**

The signals in all eight DACs are scaled by a factor of two raised to the power of **DAC_gain**. A gain of 1, 2, 4, 8, 16, 32, 64, or 128 may be specified in this way.

WireIn 0x01: WireInMaxTimeStepLsb
WireIn 0x02: WireInMaxTimeStepMsb

These two 16-bit ports are used to convey a 32-bit unsigned integer **MaxTimeStep** that indicates the maximum number of time steps (samples) that will run once data acquisition is started by pulsing **TrigInSpiStart**. If **SPI_run_continuous** (in WireIn address 0x00) is set high, the value in this register is ignored. Setting this register to zero will halt data acquisition, provided **SPI_run_continuous** is set to zero.

WireIn 0x03: WireInDataFreqPll
TriggerIn 0x40: TrigInDcmProg

WireInDataFreqPll[7:0]: "M" multiply parameter for clock frequency synthesizer

WireInDataFreqPll[15:8]: "D" divide parameter for clock frequency synthesizer

These parameters are used to set the RHS amplifier sampling frequency. A 200 MHz reference clock is provided to the FPGA from an off-chip clock generator. The output frequency of a programmable FPGA clock generator (a multi-mode clock manager, or MMCM) is given by:

$$\text{FPGA internal clock frequency} = 200 \text{ MHz} \times (M / D) / 4$$

M and D are "multiply" and "divide" integers used in the FPGA's MMCM phase-locked loop (PLL) frequency synthesizer, and are subject to the following restrictions:

- M must have a value in the range of 2 to 256.
- D must have a value in the range of 1 to 256.
- The ratio M / D must fall in the range of 0.05 to 3.33

(See pages 85-86 of Xilinx document UG382 "Spartan-6 FPGA Clocking Resources" for more details.)

For compatibility with software that also communicates with the original version of RhythmStim and its digital clock manager (DCM), the above M,D calculation is used. However, the Artix-7 FPGA that is used for the 7310 interface actually controls the output frequency with a different equation, detailed on page 72 of Xilinx document UG472 "7 series FPGAs Clocking Resources User Guide". RhythmStim USB-7310 expects the original RhythmStim M,D values and translates them into the M,D,O values used by the 7-series FPGAs.

This variable-frequency clock drives the state machine that controls all SPI communication with the RHS chips. A complete SPI cycle (consisting of one $\overline{\text{CS}}$ pulse and 32 SCLK pulses) takes 140 clock cycles. The SCLK period is 4 clock cycles; the $\overline{\text{CS}}$ pulse is high for 10 clock cycles between commands.

RhythmStim USB-7310 samples all 16 channels and then executes 4 auxiliary commands that can be used to read and write from other registers on the chip. Therefore, a complete cycle that samples from each amplifier channel takes 140 × (16 + 4) = 140 × 20 = 2800 clock cycles. So the per-channel sampling rate of each amplifier is 2800 times slower than the internal FPGA clock frequency.

Based on these design choices, we can use the following values of M and D to generate the following useful amplifier sampling rates for electrophysiological applications.

M	D	FPGA internal clock frequency	per-channel sample rate	per-channel sample period
7	125	2.80 MHz	1.00 kS/s	1000.0 μs
7	100	3.50 MHz	1.25 kS/s	800.0 μs
21	250	4.20 MHz	1.50 kS/s	666.7 μs
14	125	5.60 MHz	2.00 kS/s	500.0 μs
35	250	7.00 MHz	2.50 kS/s	400.0 μs
21	125	8.40 MHz	3.00 kS/s	333.3 μs
14	75	9.33 MHz	3.33 kS/s	300.0 μs
28	125	11.20 MHz	4.00 kS/s	250.0 μs
7	25	14.00 MHz	5.00 kS/s	200.0 μs
7	20	17.50 MHz	6.25 kS/s	160.0 μs
112	250	22.40 MHz	8.00 kS/s	125.0 μs
14	25	28.00 MHz	10.00 kS/s	100.0 μs

7	10	35.00 MHz	12.50 kS/s	80.0 μs
21	25	42.00 MHz	15.00 kS/s	66.7 μs
28	25	56.00 MHz	20.00 kS/s	50.0 μs
35	25	70.00 MHz	25.00 kS/s	40.0 μs
42	25	84.00 MHz	30.00 kS/s	33.3 μs

To set a new clock frequency, assert new values for M and D and pulse **TrigInDcmProg[0]**:

```
okCFrontPanel* dev;
// Code to open Opal Kelly board not shown here.
dev->SetWireInValue(0x03, (M << 8) + D);  // 0x03 = WireInDataFreqPll
dev->UpdateWireIns();
dev->ActivateTriggerIn(0x40, 0);  // 0x40 = TrigInDcmProg
```

If the board is reset, the sample rate will revert to 30 kS/s/channel.

WireIn 0x04: WireInMisoDelay

WireInMisoDelay[3:0]: MISO delay for Port A
WireInMisoDelay[7:4]: MISO delay for Port B
WireInMisoDelay[11:8]: MISO delay for Port C
WireInMisoDelay[15:12]: MISO delay for Port D

These four 4-bit registers set the sampling delay applied to MISO1 and MISO2 inputs on each of the four RHS SPI ports to account for cable propagation delays. Each register takes a value between 0-15. If the register is set to zero, MISO1 and MISO2 on that port are sampled on the rising edge of each SCLK pulse. Increasing the register value by one delays MISO sampling by one-quarter of an SCLK period. Each SCLK period is equal to 1/700 of the per-channel sampling period.

For example, if the per-channel amplifier sampling rate is set to 20 kS/s then the per-channel sampling period is 50 μs. The SCLK period will then be 71.4 ns, so every unit increase in the MISO delay register will delay MISO sampling by 17.9 ns.

In addition to the expected round-trip cable delays, there will also be delays due to the FPGA I/O and the RHS I/O. The Xilinx Artix-7 FPGA has input and output pin delays of several nanoseconds The RHS has a typical I/O delay of 9.0 ns. This total I/O delay, plus any delays added by additional circuitry (e.g., isolation buffers) should be added to any expected cable delays when calculating the optimum values for these registers.

WireIn 0x05: WireInStimCmdMode

WireInStimCmdMode[0]: **stim_cmd_mode**

Setting this bit to one enables automatic stimulation command mode, where the four auxiliary command slots are controlled by the FPGA in response to stimulation parameters and stimulation trigger sources that have been configured (see below for details). Setting this bit to zero allows the user to specific arbitrary SPI commands for the four auxiliary command slots.

WireIn 0x06: WireInStimRegAddr
WireIn 0x07: WireInStimRegWord
TriggerIn 0x41: TrigInSpiStart
TriggerIn 0x42: TrigInRamAddrReset

WireInStimRegAddr[3:0]: **StimRegAddress**
WireInStimRegAddr[7:4]: **StimRegChannel**
WireInStimRegAddr[12:8]: **StimRegModule**

WireInStimRegWord[15:0]: **StimProgWord**

TrigInSpiStart[1]: **ResetSequencers**

TrigInRamAddrReset[1]: **ProgramStimReg**

The RhythmStim USB-7310 FPGA code creates 128 independent **Stimulation Sequencer** state machines that automatically control stimulation sequences across all channels on eight RHS chips when **stim_cmd_mode** is set to one. Each Stimulation Sequencer has 14 internal registers that can be programmed using these WireIns and TriggerIn.

To program a state machine register, assert new values for **StimRegAddress**, **StimRegChannel**, **StimRegModule**, and **StimProgWord**, and pulse **TrigInRamAddrReset[1]**:

```
okCFrontPanel* dev;
// Code to open Opal Kelly board not shown here.
dev->SetWireInValue(0x06, (module << 8) + (channel << 4) + address);
dev->SetWireInValue(0x07, register_value);
dev->UpdateWireIns();
dev->ActivateTriggerIn(0x42, 1);
```

The value of StimRegChannel should be between zero and 15; this corresponds to the channel on an RHS chip. The value of StimRegModule is used to select between RHS chips on the following ports:

StimRegModule	Port
0	Port A, MISO1
1	Port A, MISO2
2	Port B, MISO1
3	Port B, MISO2
4	Port C, MISO1
5	Port C, MISO2
6	Port D, MISO1
7	Port D, MISO2

The 14 registers in each of the 128 Stimulation Sequencer state machines are listed and described below:

Stimulation Sequencer Register Address	Stimulation Sequencer Register Name
0	TriggerParams
1	StimParams
2	EventAmpSettleOn
3	EventAmpSettleOff
4	EventStartStim
5	EventStimPhase2
6	EventStimPhase3
7	EventEndStim
8	EventRepeatStim
9	EventChargeRecovOn
10	EventChargeRecovOff
11	EventAmpSettleOnRepeat
12	EventAmpSettleOffRepeat
13	EventEnd

The functions of each Stimulation Sequencer register are described below:

TriggerParams[4:0]: TriggerSource. This 5-bit number selects the signal to be used as the trigger for stimulation pulses. TriggerSource may have the following values:

0-15: Digital In 1-16 ports
16-23: Analog In (ADC) 1-8 ports (logic threshold voltage set by WireIn 0x0F)

24-31: Software triggers (see WireIn 0x12)

TriggerParams[5]: TriggerOnEdge. If this bit is set to one, a single stimulation pulse will trigger on the rising or falling edge of the trigger source. If this bit is set to zero, stimulation pulses will be level triggered, and will repeat as long as the trigger signal remains at the level specified by TriggerPolarity (see next item).

TriggerParams[6]: TriggerPolarity. If this bit is set to zero, stimulation will trigger on a low or falling-edge signal. If this bit is set to one, stimulation will trigger on a high or rising-edge signal.

TriggerParams[7]: TriggerEnabled. This bit must be set to one to enable stimulation on this channel. If this bit is set to zero, all trigger events are ignored and stimulation is disabled on this channel.

StimParams[7:0]: NumberOfStimPulses. This 8-bit number should be set to the number of stimulation pulses that a single trigger should elicit **minus one**. So if a single stimulation pulse is desired, this number should be set to zero. If 256 pulses are desired, this number should be set to 255.

StimParams[9:8]: StimShape. This 2-bit number determines the shape of the stimulation current pulse. There are three valid values for the Stimulation Sequencer:

0: Biphasic. Current of one polarity followed immediately by current of the opposite polarity.
1: Biphasic with interphase delay. Current of one polarity followed by a period of zero current, followed by current of the opposite polarity.
2: Triphasic. Current of one polarity followed immediately by current of the opposite polarity, then followed by current of the first polarity.

StimParams[10]: NegStimFirst. If this bit is set to one, the first phase of stimulation current will be negative (i.e., cathodic). If this bit is set to zero, the first phase of stimulation current will be positive (i.e., anodic). Most neural stimulation applications use cathodic-first stimulation.

The remaining “event” registers take 16-bit numbers that specify times relative to the trigger time (t = 0), in multiples of amplifier sampling period. For example, if the amplifiers are sampled at 20 kS/s, the sampling period will be 50 μs, so an event register value of 40 would correspond to a time of 2.0 ms after the trigger.

EventAmpSettleOn: At this time after a trigger event, the amplifier settle circuitry is activated for this channel. If you do not wish to enable the amplifier settle circuitry during a stimulation event, set this event time to be greater than EventEnd.

EventAmpSettleOff: At this time, the amplifier settle circuitry is turned off.

EventStartStim: At this time, the stimulation current is turned on with a polarity controlled by NegStimFirst. The amplitude of this stimulation current should have been previously selected by manually programming Registers 34-35, 64-79, and 96-111 on each RHS chip.

EventStimPhase2: At this time, the stimulation current switches polarity. The amplitude can be different depending on how Registers 64-79 and 96-111 are programmed.

EventStimPhase3: This register is only used if StimShape is set to Triphasic. At this time, the stimulation current switches back to the original polarity.

EventEndStim: At this time, the stimulation current is turned off.

EventRepeatStim: If more than one stimulation pulse was selected (i.e., NumberOfStimPulses > 0), then at this time the sequencer will jump back to t = 0 until the desired number of pulses have been delivered.

EventChargeRecovOn: At this time after a trigger event, the charge recovery circuitry is activated for this channel. If you do not wish to enable the charge recovery circuitry during a stimulation event, set this event time to be greater than EventEnd.

EventChargeRecovOff: At this time, the charge recovery circuitry is turned off.

EventAmpSettleOnRepeat and EventAmpSettleOffRepeat: If multiple stimulation pulses have been selected, these registers can be used to turn the amplifier settle circuitry on and off between repeated pulses. These times should always be less than EventRepeatStim. If you wish for the amplifier settle circuitry to remain on continuously during a multiple-pulse train or you wish to avoid using amplifier settle entirely, set these times greater than EventEnd.

EventEnd: This time marks the end of a stimulation event. The Stimulation Sequencer is blind to new trigger events until EventEnd is reached, so this register can be used to set a “refractory period” where re-triggering is disabled for a time after a stimulation pulse has completed. If level-based triggered is selected, this parameter will determine the repetition rate of multiple pulses caused by an extended trigger signal.

The RhythmStim USB-7310 FPGA code also contains eight **Analog Output Sequencers** that can be used to create triggered voltage pulses on any of the Analog Out (DAC) ports. The registers in the Analog Output Sequencers are addressed with values of StimRegModule ranging from 8-15, with StimRegChannel equal to zero. Each state machine has 11 registers that are listed and described below:

Analog Output Register Address	Analog Output Register Name
0	TriggerParams
1	StimParams
4	EventStartStim
5	EventStimPhase2
6	EventStimPhase3
7	EventEndStim
8	EventRepeatStim
9	DACBaseline
10	DACPositive
11	DACNegative
13	EventEnd

TriggerParams and StimParams have the same functions as in the Stimulation Sequencer described above, with the exception that the StimShape variable can take one of four values:

0: Biphasic. Voltage of one level followed immediately by voltage of a different level.
1: Biphasic with interphase delay. Voltage of one level followed by a period of baseline voltage, followed by voltage of a different polarity.
2: Triphasic. Voltage of one level followed immediately by voltage of a different polarity, then followed by voltage of the first level.
3: Monophasic: Voltage of one level.

EventStartStim, EventStimPhase2, EventStimPhase3, EventEndStim, EventRepeatStim, and EventEnd have analogous functions to the registers of the same name in the Stimulation Sequencer.

DACBaseline is a 16-bit number that sets the default baseline level for the DAC. Similarly, DACPositive sets the positive voltage level and DACNegative sets the negative voltage level. All of these values are unsigned numbers with offsets: a value of 32768 sets the voltage to the middle of the DAC range. A value of zero sets the DAC to its minimum voltage; a value of 65535 sets the DAC to its maximum voltage.

The RhythmStim USB-7310 FPGA code also contains sixteen **Digital Output Sequencers** that can be used to create triggered logic pulses on any of the Digital Out ports. The registers in the Digital Output Sequencers are addressed with StimRegModule equal to 16 and with StimRegChannel set to a value between 0-15 to select Digital Out ports 1-16. Each state machine has 6 registers that are listed and described below:

Digital Output Register Address	Digital Output Register Name
0	TriggerParams
1	StimParams
4	EventStartStim
7	EventEndStim

8	EventRepeatStim
13	EventEnd

TriggerParams, EventStartStim, EventEndStim, EventRepeatStim, and EventEnd have analogous functions to the registers of the same name in the Stimulation Sequencer.

The StimParams register contains only the NumberOfStimPulses variable; there is no StimShape or NegStimFirst variable. All logic pulses are monophasic.

The FPGA code includes provisions that ensure when data acquisition is stopped, all units under Sequencer control are turned off: RHS current stimulators are turned off, DACs are set to their baseline values, and Digital Out channels are set to zero. However, when data acquisition is stopped it is possible that some of the Sequencers could be in the middle of producing multiple output pulses, and when data acquisition is restart (possibly a long time later) the pulses will continue. To avoid this preservation of state, the user can pulse the **ResetSequencers** bit (bit 1 on **TrigInSpiStart**) to reset all multiple pulse counters to zero. It is good practice to pulse this trigger after data acquisition is stopped.

WireIn 0x08: WireInDcAmpConvert

WireInDcAmpConvert[0]: **DcAmpConvertEnable**

This bit controls the D flag in the CONVERT commands issued to the RHS chips. Setting it to one causes all attached chips to perform ADC conversions on the DC low-gain amplifiers as well as the AC high-gain amplifiers. See the RHS datasheet for more details.

WireIn 0x09: WireInExtraStates

Setting WireInExtraStates to a value greater than zero extends the number of internal clock cycles that CS is high from ten to ten plus the value of this variable. Under normal operation, this WireIn should be set to zero.

WireIn 0x0A: WireInDacReref

WireInDacReref[4:0]: **DAC_reref_channel_sel**
WireInDacReref[9:5]: **DAC_reref_stream_sel**
WireInDacReref[10]: **DAC_reref_mode**

If **DAC_reref_mode** is set to one, the amplifier channel from stream **DAC_reref_stream_sel** and channel **DAC_reref_channel_sel** is subtracted from all eight amplifier signals that are routed to the DACs. If digital re-referencing is implemented in software, these registers may be used to add real time re-referencing to the DAC outputs.

WireIn 0x0B: unused

WireIn 0x0C: WireInAuxEnable

WireInAuxEnable[0]: AuxEnablePortA1
WireInAuxEnable[1]: AuxEnablePortA2
WireInAuxEnable[2]: AuxEnablePortB1
WireInAuxEnable[3]: AuxEnablePortB2
WireInAuxEnable[4]: AuxEnablePortC1
WireInAuxEnable[5]: AuxEnablePortC2
WireInAuxEnable[6]: AuxEnablePortD1
WireInAuxEnable[7]: AuxEnablePortD2

These eight bits control whether auxiliary command sequences uploaded to the FPGA are sent to chips connected to SPI Ports A-D, MOSI 1 and 2, when **stim_cmd_mode** is set to zero. When any of these bits are set to zero, the selected SPI Ports are sent dummy commands (reading from ROM register 255) instead of commands in the auxiliary command list. This "masking" capability allows users to program the registers of one RHS chip without overwriting the registers of other connected chips.

WireIn 0x0D: WireInGlobalSettleSelect

WireInGlobalSettleSelect[0]: SettleWholeHeadstageA
WireInGlobalSettleSelect[1]: SettleWholeHeadstageB
WireInGlobalSettleSelect[2]: SettleWholeHeadstageC
WireInGlobalSettleSelect[3]: SettleWholeHeadstageD
WireInGlobalSettleSelect[4]: SettleAllHeadstages

Since stimulation on one channel can create recording artifacts on nearby channels due to capacitive coupling in headstages, connectors, and/or electrode arrays, it is often useful to perform fast artifact recovery (i.e., amplifier settling) across an entire headstage. If any of the first four bits of this WireIn are set to one, activation of amplifier settling on any channel on a port (A-D) will activate amplifier settling on **all** channels on that port. The fifth bit performs the same function globally for all ports.

WireIn 0x0E: unused

WireIn 0x0F: WireInAdcThreshold

This 16-bit number sets the voltage level used as a logic threshold for Analog In (ADC) channels that are used as triggers for the Stimulation Sequencers, Analog Output Sequencers, or Digital Output Sequencers (see above). This variable uses an unsigned offset representation, so that a value of 32768 represents the midpoint of the ADC's conversion range.

WireIn 0x10: WireInSerialDigitalInCntl

WireInSerialDigitalInCntl[0]: **serial_CLK_manual**

WireInSerialDigitalInCntl[1]: **serial_LOAD_manual**

These bits control two 16-to-1 digital multiplexer shift registers that read digital inputs and configuration switch settings. The output of these shift registers are conveyed via WireOut 0x26. These signals are controlled in the readDigitalInManual() and readDigitalInExp() methods in rhs2000evalboard.cpp.

WireIn 0x11: WireInLedDisplay

WireInLedDisplay[7:0]: These bits control the eight red LEDs on the Opal Kelly board. Setting bits to one turns LEDs on.

WireInLedDisplay[15:8]: These bits control the FPGA pins SPI_LED_A through SPI_LED_F, and can be used to implement SPI port status indicator LEDs.

WireIn 0x12: WireInManualTriggers

WireInManualTriggers[7:0]: These bits provide eight software-controlled trigger sources that can be selected by any of the Stimulation Sequencers, Analog Output Sequencers, or Digital Output Sequencers (see above).

WireIn 0x13: WireInTtlOutMode

WireInTtlOutMode[7:0]: Setting these bits to one causes Digital Out ports 1-8 to be controlled by the FPGA threshold comparators. Setting these bits to zero causes the corresponding Digital Out ports to be controlled by Digital Output Sequencers, if they are enabled. (Digital Out ports 9-16 are always controlled by Digital Output Sequencers.)

DataStreamSel	Data Stream
0	Port A, MISO1
1	Port A, MISO2
2	Port B, MISO1
3	Port B, MISO2
4	Port C, MISO1
5	Port C, MISO2
6	Port D, MISO1
7	Port D, MISO2

WireIn 0x14: WireInDataStreamEn

WireInDataStreamEn[0]: DataStreamEnA1
WireInDataStreamEn[1]: DataStreamEnA2
WireInDataStreamEn[2]: DataStreamEnB1
WireInDataStreamEn[3]: DataStreamEnB2
WireInDataStreamEn[4]: DataStreamEnC1
WireInDataStreamEn[5]: DataStreamEnC2
WireInDataStreamEn[6]: DataStreamEnD1
WireInDataStreamEn[7]: DataStreamEnD2

Setting one of these bits to one enables at particular data stream from SPI port A-D, MISO 1 or 2, so that its data is sent over the USB interface to the host computer. Setting the bit to zero disables the stream, reducing USB bandwidth and FIFO usage.

Any changes made to these bits do not take effect while data acquisition is running. Data acquisition must be stopped and restarted to enable or disable data streams, so that the size of a data frame never changes during active acquisition.

WireIn 0x15: unused

WireIn 0x16: WireInDacSource1
WireIn 0x17: WireInDacSource2
WireIn 0x18: WireInDacSource3
WireIn 0x19: WireInDacSource4
WireIn 0x1A: WireInDacSource5
WireIn 0x1B: WireInDacSource6
WireIn 0x1C: WireInDacSource7
WireIn 0x1D: WireInDacSource8

WireInDacSourceX[4:0]: DacSourceChannelX
WireInDacSourceX[8:5]: DacSourceStreamX
WireInDacSourceX[9]: DacSourceEnableX

These registers route selected amplifier signals to the eight DACs that RhythmStim USB-7310 supports. For each DAC, the user may select an amplifier channel (0-15) and a data stream (see table below). To enable the DAC, the **DacSourceEnable** bit must be set high. If **DacSourceStream** is set to 8, the DAC will be controlled directly by the host computer via **WireInDacManual**; the **DacSourceChannel** parameter is ignored in this case.

DacSourceStream	Data Stream Routed to DAC
0	Port A, MISO1
1	Port A, MISO2

2	Port B, MISO1
3	Port B, MISO2
4	Port C, MISO1
5	Port C, MISO2
6	Port D, MISO1
7	Port D, MISO2
8	DAC Manual Input
10-15	all zeros

WireIn 0x1E: WireInDacManual

This WireIn is used to control DACs directly when their DacSourceStream parameter is set to 8 (see table above). Typically, this WireIn can only be updated around 1,000 times per second in compiled C++ software, so it cannot be used to synthesize high-frequency waveforms.

WireIn 0x1F: WireInMultiUse

WireInMultiUse is used in concert with TriggerIn signals to program other registers in the FPGA. (See below.)

TrigIn 0x43: TrigInDacThresh

TrigInDacThresh[0]: SetDac1ThresholdLevel
TrigInDacThresh[1]: SetDac2ThresholdLevel
TrigInDacThresh[2]: SetDac3ThresholdLevel
TrigInDacThresh[3]: SetDac4ThresholdLevel
TrigInDacThresh[4]: SetDac5ThresholdLevel
TrigInDacThresh[5]: SetDac6ThresholdLevel
TrigInDacThresh[6]: SetDac7ThresholdLevel
TrigInDacThresh[7]: SetDac8ThresholdLevel

TrigInDacThresh[8]: SetDac1ThresholdPolarity
TrigInDacThresh[9]: SetDac2ThresholdPolarity
TrigInDacThresh[10]: SetDac3ThresholdPolarity
TrigInDacThresh[11]: SetDac4ThresholdPolarity
TrigInDacThresh[12]: SetDac5ThresholdPolarity
TrigInDacThresh[13]: SetDac6ThresholdPolarity
TrigInDacThresh[14]: SetDac7ThresholdPolarity
TrigInDacThresh[15]: SetDac8ThresholdPolarity

RhythmStim USB-7310 includes on-FPGA threshold comparators that produce low-latency digital signals indicating if each waveform routed to one of the eight DACs exceeded user-programmed levels. The bits in this TrigIn are used to program DAC threshold levels and polarities. To program a DAC threshold level, apply the desired value to **WireInMultiUse** and pulse one of the bottom eight bits in **TrigInDacThresh**.

To program a DAC threshold polarity, apply either a zero or one to **WireInMultiUse** and pulse one of the top eight bits in **TrigInDacThresh**. A polarity value of zero will cause the corresponding digital output to go high when the signal routed to that DAC equals or falls below the DAC threshold level. A polarity value of one will cause the digital output to go high when the signal routed to that DAC equals or rises above the DAC threshold level.

TrigIn 0x44: TrigInDacHpf

TrigInDacHpf[0]: EnableDacHighpassFilter
TrigInDacHpf[1]: SetDacHighpassFilterCutoff

RhythmStim USB-7310 includes on-FPGA first-order high-pass filters that can be applied to the eight amplifier signals routed to the DACs and threshold comparators (see above). These high-pass filters can be used to remove low-frequency local field potentials (LFPs) from wideband neural signals before detecting spikes using the programmable comparators. The bits in this TrigIn are used to enable the high-pass filters and to program their cutoff frequency.

To enable or disable the high-pass filters, apply a one or zero, respectively, to **WireInMultiUse** and pulse **TrigInDacHpf[0]**. To set the filter cutoff frequency, apply the filter coefficient to **WireInMultiUse** and pulse **TrigInDacHpf[1]**.

The filter coefficient is a 16-bit unsigned integer that is calculated as follows:

$$\text{Filter coefficient} = 65536 \cdot [1 - \exp(-2\pi \cdot f_{cutoff}/f_{sample})]$$

Note that the filter coefficient depends on the amplifier sampling frequency, so if this is changed, the filter coefficient should be updated appropriately.

TrigIn 0x45: TrigInAuxCmdLength

TrigInAuxCmdLength[0]: MaxAuxCmdIndex1
TrigInAuxCmdLength[1]: MaxAuxCmdIndex2
TrigInAuxCmdLength[2]: MaxAuxCmdIndex3
TrigInAuxCmdLength[3]: MaxAuxCmdIndex4
TrigInAuxCmdLength[4]: LoopAuxCmdIndex1
TrigInAuxCmdLength[5]: LoopAuxCmdIndex2
TrigInAuxCmdLength[6]: LoopAuxCmdIndex3
TrigInAuxCmdLength[7]: LoopAuxCmdIndex4

For each auxiliary command slot (**auxcmd1**, **auxcmd2**, **auxcmd3**, and **auxcmd4**), the user must specify the length of the command sequence. This **command sequence length** is the same for all SPI Ports and MOSI lines. When data acquisition is started, all auxiliary command sequences start at the beginning of their selected RAM banks and increment to the next command every sampling period. When the command index reaches the specified command sequence length, the command index resets to a **command loop index**. In many cases, the command loop index will be set to zero so that the entire command sequence repeats in its entirety. However, the command loop index may be set to a number greater than zero so that the first part of the command sequence is executed only once, and a later sequence is executed in a loop.

The RhythmStim USB-7310 FPGA code includes four 8192 x 32 bit RAM banks used to store commands for the four auxiliary command slots **auxcmd1**, **auxcmd2**, **auxcmd3**, and **auxcmd4**. The command sequence length for each auxiliary command slot is set by writing a number between 0 and 8191 to **WireInMultiUse** and pulsing one of the bits 0-3 of **TrigInAuxCmdLength**. The command loop index for each slot is set by writing a number between 0 and 8191 to **WireInMultiUse** and pulsing one of the bits 4-7 of **TrigInAuxCmdLength**.

For example, to set a command sequence length of eight (which will execute **nine** commands: indices 0-8) and a command loop index of zero, both for **auxcmd2**:

```
okCFrontPanel* dev;
// Code to open Opal Kelly board not shown here.
dev->SetWireInValue(0x1F, 8);  // 0x1F = WireInMultiUse
dev->UpdateWireIns();
dev->ActivateTriggerIn(0x45, 1);  // trigger write to MaxAuxCmdIndex2
dev->SetWireInValue(0x1F, 0);  // 0x1F = WireInMultiUse
dev->UpdateWireIns();
dev->ActivateTriggerIn(0x45, 5);  // trigger write to LoopAuxCmdIndex2
```

USB Software Endpoints: PipeIn Ports

USB-7310 PipeIn ports are virtual 8-bit buses that stream data bytes synchronously from the host computer to the FPGA. Up to 32 PipeIn ports (with addresses from 0x80 to 0x9F) are available for use with each Opal Kelly module. Each PipeIn word is 32 bits; for compatibility with earlier versions of RhythmStim USB-7310 in which PipeIns were only 16 bits, the FPGA is configured to only read the lowest 16 bits of the 32-bit PipeIns. The highest 16 bits are ignored and should be padded with 0s. The following C++ code shows how data is streamed over PipeIn ports.

```
okCFrontPanel* dev;
// Code to open Opal Kelly board not shown here.

// Our goal is to transmit 64 kBytes over the 16 lowest bits of a PipeIn,
// and pad the highest 16 bits with zeros.

// Allocate buffer to store 64 kBytes.
unsigned char dataToTransfer[65536];
unsigned char usbBuffer[65536 * 2];

// For every 16 bits (2 bytes) of data to transfer, pad with 2 bytes of zeros
// so the actual data is always aligned in the lowest 16 bits of the 32-bit PipeIn.
int num_16bit_words = 65536 / 2;
for (unsigned int i = 0; i < num_16bit_words; i++) {
     usbBuffer[4*i + 0] = dataToTransfer[2*i + 0];
     usbBuffer[4*i + 1] = dataToTransfer[2*i + 1];
     usbBuffer[4*i + 2] = 0;
     usbBuffer[4*i + 3] = 0;
}

// Write 64 * 2 kBytes from buffer to PipeIn.
long numBytesToWrite = 65536 * 2;
dev->WriteToPipeIn(0x80, numBytesToWrite, usbBuffer);
```

See the Opal Kelly FrontPanel User's Manual for more information on PipeIn endpoints.

PipeIn 0x80: PipeInAuxCmd1Msw
PipeIn 0x81: PipeInAuxCmd1Lsw
PipeIn 0x82: PipeInAuxCmd2Msw
PipeIn 0x83: PipeInAuxCmd2Lsw
PipeIn 0x84: PipeInAuxCmd3Msw
PipeIn 0x85: PipeInAuxCmd3Lsw
PipeIn 0x86: PipeInAuxCmd4Msw
PipeIn 0x87: PipeInAuxCmd4Lsw

TrigIn 0x42: TrigInRamAddrReset

These PipeIns are used to upload auxiliary command sequences from the host computer to the FPGA. Each of these PipeIns is backed by an 8192 x 16 bit RAM block on the FPGA, so up to 16384 bytes can be uploaded after the RAM blocks are reset by pulsing **TrigInRamAddrReset[0]**. Each command word is 32 bits wide, and there are two different PipeIns for each auxiliary command slot to receive the most significant word (MSW) and least significant word (LSW) of each command. Since PipeIns fundamentally transfer 8-bit bytes, the least significant byte of each word should be sent first. The FPGA is configured to only read the lowest 16-bits of a 32-bit PipeIn, so we pad the highest 16-bits with zeros that can be safely ignored instead of actual data that needs to be transferred.

Example writing commands to the **auxcmd1** slot:

```
okCFrontPanel* dev;
// Code to open Opal Kelly board and start SPI operation not shown here.

// Allocate buffers to command word lists.
unsigned int commandList[8192];
unsigned char commandBufferMsw[8192 * 4];
unsigned char commandBufferLsw[8192 * 4];
int i, numCmds;

// ...Assume that commandList has now been filled in with numCmds commands.

// Now break apart commands into bytes:
for (i = 0; i < numCmds; i++) {
     commandBufferLsw[4*i + 0] = (unsigned char)(commandList[i] & 0x000000ff) >> 0);
     commandBufferLsw[4*i + 1] = (unsigned char)(commandList[i] & 0x0000ff00) >> 8);
```

```
        commandBufferLsw[4*i + 2] = 0;
        commandBufferLsw[4*i + 3] = 0;

        commandBufferMsw[4*i + 0] = (unsigned char)(commandList[i] & 0x00ff0000) >> 16);
        commandBufferMsw[4*i + 1] = (unsigned char)(commandList[i] & 0xff000000) >> 24);
        commandBufferMsw[4*i + 2] = 0;
        commandBufferMsw[4*i + 3] = 0;
    }


    // Reset RAM address pointer to zero.
    dev->ActivateTriggerIn(0x42, 0);  // 0x42 = TrigInRamAddrReset
    // Write bytes to auxcmd1 MSW PipeIn from buffer.
    dev->WriteToPipeIn(0x80, 4*numCmds, commandBufferMsw);  // 0x80 = PipeInAuxCmd1Msw
    // Reset RAM address pointer to zero again.
    dev->ActivateTriggerIn(0x42, 0);  // 0x42 = TrigInRamAddrReset
    // Write bytes to auxcmd1 LSW PipeIn from buffer.
    dev->WriteToPipeIn(0x81, 4*numCmds, commandBufferLsw);  // 0x81 = PipeInAuxCmd1Lsw
```

USB Software Endpoints: WireOut Ports

USB-7310 WireOut ports are virtual 32-bit wires that transfer data asynchronously from the FPGA to the host computer. Up to 32 WireOut ports (with addresses from 0x20 to 0x3F) are available for use with each Opal Kelly module. For compatibility with earlier versions of RhythmStim USB-7310 in which WireOuts were only 16 bits, in practice every WireOut only uses its lowest 16 bits. The full 32-bit word can still be read, but the highest 16 bits will simply be zeros. The following C++ code shows how data is received from WireOut ports.

```
okCFrontPanel* dev;
// Code to open Opal Kelly board not shown here.

// We must first execute UpdateWireOuts to refresh WireOut values on host computer.
dev->UpdateWireOuts();

// Read from WireOut 0x20 endpoint.
unsigned int fromFpga;
fromFpga = dev->GetWireOutValue(0x20);
```

WireOut 0x20: WireOutNumWordsLsb
WireOut 0x21: WireOutNumWordsMsb

These two 32-bit ports are used to transfer a 32-bit unsigned integer **NumWords** that indicates the total number of 16-bit words contained in the USB FIFO on the Opal Kelly board (see above for an explanation why each WireOut only contains 16 bits of data). Before executing a ReadFromPipeOut command, the host computer should first read this number to ensure that sufficient data is present in the FIFO buffer. Otherwise, underflow will occur and corrupted data will be transferred to the computer. (Note that NumWords reports the number of 16-bit **words** in the FIFO, while ReadFromPipeOut operates at the **byte** level.) These ports should also be monitored to prevent buffer overflow. The FIFO can hold slightly over 2^{26} = 67,108,864 words, but in practice it should never be allowed to get close to its maximum capacity.

WireOut 0x22: WireOutSpiRunning

WireOutSpiRunning[0]: **SpiRunning**

This bit is high while data acquisition is running and low when it has stopped.

WireOutSpiRunning[15:1]: unused

WireOut 0x23: WireOutTtlIn

This register returns the values of the 16 TTL input pins defined by RhythmStim USB-7310. (These pins are only sampled synchronously and updated when the RHS SPI commands are running.)

WireOut 0x24: WireOutDataClkLocked

WireOutDataClkLocked[0]: **DataClkLocked**

This bit goes high when the digital clock manager has successfully stabilized to a new frequency. When **WireInDataFreqPll** is changed and **DcmProgTrigger** is pulsed, it can take several milliseconds for the new clock frequency to stabilize. This pin should be monitored after frequency changes, and no data acquisition should be performed until this bit goes high.

WireOutDataClkLocked[1]: **DcmProgDone**

When **WireInDataFreqPll** is changed and **DcmProgTrigger** is pulsed, it can take several milliseconds before the digital clock manager is ready to be changed to yet another frequency. After any frequency changes this pin should be monitored, and no more frequency changes should be attempted until this bit goes high.

WireOutDataClkLocked[15:2]: unused

WireOut 0x25: WireOutBoardMode

WireOutBoardMode[3:0]: **BoardMode**

BoardMode is a 4-bit value set by direct inputs to the FPGA. These pins are typically connected to DIP switches that are set to identify different types of boards.

WireOut 0x3E: WireOutBoardId

This WireOut returns a constant value of 800 (decimal), identifying the board as a RhythmStim USB-7310-compatible device.

WireOut 0x3F: WireOutBoardVersion

This WireOut returns a constant value, identifying the version of RhythmStim USB-7310 running on the board (currently 1).

USB Software Endpoints: BTPipeOut Ports

USB-7310 BTPipeOut ports are virtual 8-bit buses (“block throttled” pipe outs) that stream data bytes synchronously from the FPGA to the host computer. Up to 32 BTPipeOut ports (with addresses from 0xA0 to 0xBF) are available for use with each Opal Kelly module. The following C++ code shows how data is streamed over BTPipeOut ports.

```
okCFrontPanel* dev;
// Code to open Opal Kelly board not shown here.

// Allocate buffer to store 64 kBytes.
unsigned char usbBuffer[65536];

// Read 64 kBytes from PipeOut 0xA0 into buffer.
long numBytesRead, numBytesToRead;
const int USB3_BLOCK_SIZE = 1024;
numBytesToRead = 65536;
numBytesRead = dev->ReadFromBlockPipeOut(0xA0, USB3_BLOCK_SIZE, numBytesToRead,
                                          usbBuffer);
// Note: numBytesRead will be negative if read failed.


// 16-bit words are sent least-significant-byte first.
unsigned int firstWord;
```

```
firstWord = (usbBuffer[1] << 8) | usbBuffer[0];
```

In order to minimize communication overhead and achieve maximum USB transfer speeds, Opal Kelly recommends reading from BTPipeOuts in relatively large data blocks. Reading approximately 30 milliseconds of accumulated amplifier data from the board at a time will provide adequate USB data rates in most cases. See the Opal Kelly FrontPanel User's Manual for more information.

BTPipeOut 0xA0: PipeOutData

This BTPipeOut is used to stream data from all RHS chips (and several additional data sources) through the FPGA to the host computer. The BTPipeOut is backed by a large FIFO implemented using the 1 GByte SDRAM (of which 128 MBytes are used) on the Opal Kelly board. This FIFO module allows continuously-streaming data (e.g., from multiple RHS chips to be transferred smoothly over a USB interface to a computer that will grab the data in bursts, and may be unresponsive for a brief time due to multitasking or other operating system overhead.

There is no mechanism in this FIFO to protect against underflow. That is, if the computer tries to read more data than the FIFO is currently holding, the FIFO will just repeat the last word after it runs out of data. To prevent underflow, it is essential for the host to monitor the value of **NumWords** (WireOut 0x20 and 0x21) and never attempt to read more words than the FIFO contains.

Example:

```
okCFrontPanel* dev;
// Code to open Opal Kelly board and start SPI operation not shown here.

// Allocate buffer to store data.
unsigned char usbBuffer[BUFFERSIZE];

// Wait until enough data is available in the FIFO...
unsigned int numWordsAvailable = 0;
while (numWordsAvailable < BUFFERSIZE / 2) {
     dev->UpdateWireOuts();
numWordsAvailable = 65536 * dev->GetWireOutValue(0x21) +
                        dev->GetWireOutValue(0x20);
}

// Remember, NumWords (WireOut 0x20 and 0x21) returns the number of 16-bit words in
// the FIFO, but ReadFromPipeOut() operates on bytes.  Factor of two difference!

// Read bytes from BTPipeOut 0xA0 into buffer.
long numBytesRead;
const int USB3_BLOCK_SIZE = 1024;
numBytesRead = dev->ReadFromBlockPipeOut(0xa0, USB3_BLOCK_SIZE, BUFFERSIZE,
                      usbBuffer);
```

RhythmStim USB-7310 Verilog Code Description

Verilog Source Code

The RhythmStim USB-7310 interface is described by a set of Verilog files that may be compiled into a bitfile using the free Xilinx Vivado software. While Intan Technologies provides a pre-compiled bitfile, developers may wish to modify the RhythmStim USB-7310 code for custom applications. The primary Verilog files are described below.

main.v: This is the top-level RhythmStim USB-7310 Verilog code that defines all FPGA I/O, executes the main state machine running all SPI I/O with the RHS chips, and defines connections to all other modules in the Verilog files listed below.

SDRAM_FIFO.v: This code makes use of 128 MBytes of the off-FPGA 1-GByte SDRAM to implement a large FIFO buffer. This Verilog file uses a number of other miscellaneous Verilog files to implement the SDRAM controller. Many of these files were derived from the Opal Kelly RamTest Verilog example.

variable_freq_clk_generator.v: This module sets up the programmable-frequency clock used to run the RHS chips at user-selectable sampling rates.

MISO_phase_selector.v: This small module implements the user-programmable delay in sampling MISO SPI lines to compensate for signal propagation delay on long cables.

RAM_bank.v and **RAM_block.v**: These files use on-FPGA memory to implement multiple 8192-word RAM banks for storing auxiliary command sequences.

DAC_output_scalable_HPF.v: This module implements a serial interface to an optional off-board Analog Devices AD5662 16-bit DAC, as well as variable gain, noise slicing, high-pass filtering, and threshold comparator functions.

ADC_input.v: This module implements a serial interface to an optional off-board Analog Devices AD7980 16-bit ADC.

stim_sequencer.v: This module implements a state machine that generates stimulation control bits for 16 RHS stimulator channels. A variety of stimulation waveforms (biphasic, biphasic with interphase delay, and triphasic), trigger methods (edge triggered, level triggered), and pulse train options are supported.

analog_out_sequencer.v: This module implements a state machine that generates voltage pulses signals for a 16-bit DAC. A variety of stimulation waveforms (biphasic, biphasic with interphase delay, triphasic, and monophasic), trigger methods (edge triggered, level triggered), and pulse train options are supported.

digout_sequencer.v: This module implements a state machine that generates digital pulses for 16 digital outputs. A variety of trigger methods (edge triggered, level triggered) and pulse train options are supported.

okLibrary.v: This Verilog file provided by Opal Kelly implements the communication endpoints (e.g., WireIns, BTPipeOuts) in the FPGA. It should not be modified.

xem7310.xdc: This file defines the location (i.e., pin number) and characteristics of all FPGA I/O signals.

main.bit: This is the bitfile that is generated by compiling all the above Verilog files with the Xilinx Vivado software. This file should be renamed to ConfigRHSController_7310.bit and copied into the directory containing the C++ executable file that opens and configures the Opal Kelly board.

Main State Machine Description

The heart of the RhythmStim USB-7310 code, in **main.v**, is a finite state machine that cycles through a pattern of 140 repeating states. These 140 states execute a single SPI cycle. (Refer to the RHS2116 datasheet for more information on the SPI communication protocol.)

Each SCLK period consists of four states labeled **ms_clkX_a** through **ms_clkX_d**, where X advances from 1 to 32. After 32 complete SCLK cycles, there are an additional two states (**ms_clk33_a** and **ms_clk33_b**) to create a delay between the falling edge of the last SCLK pulse and the rising edge of $\overline{CS}$. The $\overline{CS}$ signal is held high for 10 states labeled **ms_cs_a** through **ms_cs_j**. After all 140 states have completed, the variable **channel** is incremented. If **channel** exceeds 19, it is reset to 0. This variable tracks the repeating series of 20 commands send on the MOSI line (16 CONVERT commands for each amplifier channel and four auxiliary commands).

Data is transferred to the FIFO (and thence to the USB interface) during particular states of this main state machine. One 16-bit word can be transferred to the FIFO during each state.

- The 64-bit header magic number is sent during states **ms_clk1_b** through **ms_clk2_a** only when channel = 0.
- The timestamp is sent during states **ms_clk2_b** and **ms_clk2_c** only when channel = 0.
- Data streams 1-8 are sent during states **ms_clk2_d** through **ms_clk6_c**.
- The state of all on-chip stimulators (including stimulator on/off, polarity, amplifier settle on/off, and charge recovery on/off) are sent during states **ms_clk6_d** through **ms_clk14_c**.
- States **ms_clk14_d** through **ms_clk30_c** could be used to send more data streams in the future.
- States **ms_clk30_d** through **ms_clk32_c** are used to send data from the eight optional DACs only when channel = 19.
- States **ms_clk32_d** through **ms_cs_e** are used to send data from the eight optional ADCs only when channel = 19.
- State **ms_cs_f** is used to send the 16 TTL inputs only when channel = 19.
- State **ms_cs_g** is used to send the current value of the 16 TTL outputs only when channel = 19.

RhythmStim USB-7310 C++ API

Intan Technologies provides a basic, open-source C++ application programming interface (API) for controlling the RhythmStim USB-7310 FPGA interface described above. The API consists of three C++ classes: **RHXController**, **RHXRegisters**, and **RHXDataBlock**. These classes are defined in *.cpp and *.h files named **rhxcontroller**, **rhxregisters**, and **rhxdatablock**. Additionally, any application must link to the Opal Kelly library file **okFrontPanel.lib**.

The RhythmStim USB-7310 API is written using standard C++ and uses the **string**, **vector**, **cmath**, **mutex**, and **deque** classes from the C++ Standard Template Library (STL).

The public member functions of each class in the RhythmStim USB-7310 API are described below.

RHXController Class Reference

This class provides access to and control of the Opal Kelly XEM7310 USB/FPGA interface board running the RhythmStim USB-7310 interface Verilog code. Only one instance of the **RHXController** object is needed to control a RhythmStim USB-7310-based FPGA interface. Methods in this class are designed to be thread-safe using a **mutex** (mutual exclusion) variable to ensure this.

The public member functions of the **RHXController** class are listed below.

RHXController()

Constructor.

~RHXController()

Destructor.

bool isSynthetic()

Returns if this controller is synthetic (used for demonstration purposes when Intan hardware may not be present). Returns false if this RHXController represents a real Opal Kelly XEM7310-A75 board.

bool isPlayback()

Returns if this controller represents a playback controller (used for data playback purposes when new data acquisition is not desired). Returns false if this RHXController represents a real Opal Kelly XEM7310-A75 board.

AcquisitionMode acquisitionMode()

Returns the mode that this controller is running in. Acquisition modes are given using the **AcquisitionMode** enumeration; defined values are:

LiveMode
SyntheticMode
PlaybackMode

Returns LiveMode if this RHXController represents a real Opal Kelly XEM7310-A75 board.

vector<string> listAvailableDeviceSerials()

Lists all available Opal Kelly devices currently connected to this computer. Returns a vector of strings containing each device's serial number.

int open(const string &boardSerialNumber)

Finds an Opal Kelly XEM7310-A75 board attached to a USB port with the given serial number and opens it. Returns 1 if successful. Returns -2 if an XEM7310 board is not found.

int open()

Finds the first Opal Kelly board attached to a USB port and opens it. Returns 1 if successful. Returns -2 if an Opal Kelly board is not found. Note that this doesn't necessarily open a XEM7310 board, so if any other Opal Kelly boards are connected, the above function to open a specific board should be used instead.

bool uploadFpgaBitFile(string filename)

Uploads the RhythmStim USB-7310 configuration file (i.e., bitfile) to the Xilinx FPGA on the open Opal Kelly board. Returns true if successful.

void initialize()

Initializes RhythmStim USB-7310 FPGA registers to default values.

static void resetBoard(okCFrontPanel* dev_)

void resetBoard()

Resets the FPGA. This clears all auxiliary command RAM banks, clears the USB FIFO, and resets the per-channel sampling rate to its default value of 30.0 kS/s/channel. The static version of this method also requires a pointer to the **okCFrontPanel** device.

void resetFpga()

Performs a low-level reset of the FPGA. This can be called when closing an application to make sure everything has stopped.

bool setSampleRate(AmplifierSampleRate newSampleRate)

Sets the per-channel sampling rate of the RHS chips connected to the RhythmStim USB-7310 FPGA. Returns false if an unsupported sampling rate is requested. Sample rates are given using the **AmplifierSampleRate** enumeration; defined values are:

SampleRate1000Hz (not compatible with RHS XEM7310 Interface)
SampleRate1250Hz (not compatible with RHS XEM7310 Interface)
SampleRate1500Hz (not compatible with RHS XEM7310 Interface)
SampleRate2000Hz (not compatible with RHS XEM7310 Interface)
SampleRate2500Hz (not compatible with RHS XEM7310 Interface)
SampleRate3000Hz (not compatible with RHS XEM7310 Interface)
SampleRate3333Hz (not compatible with RHS XEM7310 Interface)
SampleRate4000Hz (not compatible with RHS XEM7310 Interface)
SampleRate5000Hz (not compatible with RHS XEM7310 Interface)
SampleRate6250Hz (not compatible with RHS XEM7310 Interface)
SampleRate8000Hz (not compatible with RHS XEM7310 Interface)
SampleRate10000Hz (not compatible with RHS XEM7310 Interface)

SampleRate12500Hz (not compatible with RHS XEM7310 Interface)
SampleRate15000Hz (not compatible with RHS XEM7310 Interface)
SampleRate20000Hz
SampleRate25000Hz
SampleRate30000Hz

double getSampleRate()

static double getSampleRate(AmplifierSampleRate sampleRate_)

Returns the current (non-static) or provided **AmplifierSampleRate** enumeration (static) as a per-channel sampling rate (in Hz) floating-point number.

AmplifierSampleRate getSampleRateEnum()

Returns the current per-channel sampling rate as an **AmplifierSampleRate** enumeration.

static string getSampleRateString(AmplifierSampleRate sampleRate_)

Returns the provided **AmplifierSampleRate** enumeration as a human-readable string, from "20 kHz" to "30 kHz".

static AmplifierSampleRate nearestSampleRate(double rate, double percentTolerance= 1.0)

Returns the nearest supported sample rate to the desired rate, within the given tolerance, as an **AmplifierSampleRate** enumeration.

static AmplifierSampleRate nearestStimStepSize(double step, double percentTolerance= 1.0)

Returns the nearest supported stimulation step size to the desired step size, within the given tolerance, as a **StimStepSize** enumeration. Step sizes are given using the **StimStepSize** enumeration; defined values are:

StimStepSize10nA
StimStepSize20nA
StimStepSize50nA
StimStepSize100nA
StimStepSize200nA
StimStepSize500nA
StimStepSize1uA
StimStepSize2uA
StimStepSize5uA
StimStepSize10uA

static string getStimStepSizeString(StimStepSize stepSize_)

Returns the provided **StimStepSize** enumeration as a human-readable string, from "10 nA" to "10 uA"

void uploadCommandList(const vector<unsigned int> &commandList, AuxCmdSlot auxCommandSlot, int bank = 0)

Uploads a command list (generated by an instance of the **Rhs2000Registers** class) to a particular auxiliary command slot on the FPGA. Command slots are given using the **AuxCmdSlot** enumeration; defined values are:

AuxCmd1
AuxCmd2

AuxCmd3
AuxCmd4

void printCommandList(const vector<unsigned int> &commandList)

Prints a command list (generated by an instance of the **RHXRegisters** class) to the console in readable form, for diagnostic purposes.

int findConnectedChips(vector<ChipType> &chipType, vector<int> &portIndex, vector<int> &commandStream, vector<int> &numChannelsOnPort, bool synthMaxChannels = false)

Scan all SPI ports to find all connected RHS amplifier chips. Read the chip ID from on-chip ROM register to determine the number of amplifier channels on each port. This process is repeated at all possible MISO delays in the FPGA to determine the optimum MISO delay for each port to compensate for variable cable length.

This function returns three vectors of length maxNumDataStreams(): chipType (the type of chip connected to each data stream); portIndex (the SPI port number [A=1, B=2,…] associated with each data stream); and commandStream (the stream index for sending commands to the FPGA for a particular read stream index). This function also returns a vector of length maxNumSPIPorts(): numChannelsOnPort (the total number of amplifier channels on each port).

This function normally returns 1. A value of -1 or -2 can be returned under certain conditions if a USB Interface Board is being used, but for the RHS XEM7310 Interface this function will always return 1.

The synthMaxChannels argument is only used for synthetic controllers, and should always be left false by default for real RHS XEM7310 boards.

The type of chip is specified using the ChipType enumeration; defined values are:

NoChip
RHD2132Chip (not compatible with RHS XEM7310 Interface)
RHD2216Chip (not compatible with RHS XEM7310 Interface)
RHD2164Chip (not compatible with RHS XEM7310 Interface)
RHS2116Chip
RHD2164MISOBChip (not compatible with RHS XEM7310 Interface)

void selectAuxCommandLength(AuxCmdSlot auxCommandSlot, int loopIndex, int endIndex)

Specifies a command sequence end point (endIndex = 0-8191) and command loop index (loopIndex = 0-8191) for a particular auxiliary command slot (AuxCmd1, AuxCmd2, AuxCmd3, or AuxCmd4).

void setContinuousRunMode(bool continuousMode);

Sets the FPGA to run continuously once started (if continuousMode is set to true) or to run until maxTimeStep is reached (if continuousMode is set to false).

void setMaxTimeStep(unsigned int maxTimeStep)

Sets maxTimeStep for cases where continuousMode = false.

void run()

Starts SPI data acquisition.

bool isRunning()

Returns true if the FPGA is currently running SPI data acquisition.

unsigned int getNumWordsInFifo()

Returns the number of 16-bit words in the USB FIFO. The user should never attempt to read more data than the FIFO currently contains, as it is not protected against underflow.

unsigned int getLastNumWordsInFifo()

Returns the most recently measured number of 16-bit words in the USB FIFO. Does not directly read this value from the USB port, and so may be out of date, but does not have to wait on other USB access to finish in order to execute.

unsigned int getLastNumWordsInFifo(bool &hasBeenUpdated)

Returns the most recently measured number of 16-bit words in the USB FIFO. Does not directly read this value from the USB port, and so may be out of date, but does not have to wait on other USB access to finish in order to execute. The boolean variable **hasBeenUpdated** is set to indicate if this value has been updated since the last time this function was called.

unsigned int fifoCapacityInWords()

Returns the number of 16-bit words in the USB SDRAM FIFO can hold (67,108,864). The FIFO can actually hold a few hundred thousand words more than this due to the on-FPGA mini-FIFOs used to interface with the SDRAM, but this function provides a conservative estimate of maximum FIFO capacity.

void setStimCmdMode(bool enabled)

Enables or disables automatic stimulation command mode in the FPGA. Affects all RHS chips connected. If automatic simulation command mode is enabled, all auxiliary commands uploaded to the FPGA are ignored, and the four extra command slots during each sampling period are used to set stimulation parameters (i.e., turning stimulators on/off, setting stimulation polarity, activating amplifier settling for fast artifact recovery, and activating charge recovery) that are controlled by the Stimulation Sequencer state machines.

void programStimReg(int stream, int channel, StimRegister reg, int value)

Programs a value to a particular Stimulation Sequencer state machine on the FPGA. Also works for Analog Output Sequencers and Digital Output Sequencers with stream > 7. See description under **WireInStimRegAddr** and **WireInStimRegWord** above for details on the operation of the Sequencers. The desired Sequencer register must be specified using the **StimRegister** enumeration; defined values are:

TriggerParams
StimParams
EventAmpSettleOn
EventAmpSettleOff
EventStartStim
EventStimPhase2
EventStimPhase3
EventEndStim
EventChargeRecovOn
EventChargeRecovOff
EventAmpSettleOnRepeat
EventAmpSettleOffRepeat
EventEnd
DacBaseline

DacPositive
DacNegative

void configureStimTrigger(int stream, int channel, int triggerSource, bool triggerEnabled, bool edgeTriggered, bool triggerOnLow)

Configures a stimulation trigger for a Sequencer state machine. This function calls programStimReg and programs the TriggerParams register of a particular Sequencer.

void configureStimPulses(int stream, int channel, int numPulses, StimShape shape, bool negStimFirst)

Configures stimulation pulse parameters for a Sequencer state machine. This function calls programStimReg and programs the StimParams register of a particular Sequencer. The desired stimulation shape must be specified using the **StimShape** enumeration; defined values are:

Biphasic
BiphasicWithInterphaseDelay
Triphasic
Monophasic

Not all stimulation shapes are valid with all types of Sequencers. See description under WireIns **WireInStimRegAddr** and **WireInStimRegWord** above for details on the operation of the Sequencers.

void resetSequencers()

This function resets all Sequencer state machines on the FPGA. This is typically called after data acquisition is stopped. It is possible that a Sequencer could be in the middle of playing out a long pulse train (e.g., 100 stimulation pulses). If this function is not called, the pulse train will resume after data acquisition is restarted.

void setAnalogInTriggerThreshold(double voltageThreshold)

Sets the voltage threshold to be used for digital triggering from Analog In (ADC) ports. This function assumes additional circuitry has been added to the 16-bit ADCs to extend their input range to ±10.24 V.

void setManualStimTrigger(int trigger, bool triggerOn)

Sets the state of the manual (software) stimulation triggers 0-7 that can be used to trigger stimulation events.

void enableAuxCommandsOnOneStream(int stream)

Assuming that automatic stimulation command mode has been disabled, this function allows the user to select one specific RHS chip (stream 0-7) to receive auxiliary commands that have been uploaded to the FPGA. All other chips receive dummy commands (reading from ROM Register 255) during the auxiliary command slots. This allows the user to program chips independently (e.g., to set stimulation amplitudes differently on different chips).

void enableAuxCommandsOnAllStreams()

Assuming that automatic stimulation command mode has been disabled, this function sets all connected RHS chips to receive the same auxiliary commands that have been uploaded in the auxcmd1 – auxcmd4 slots on the FPGA.

void setGlobalSettlePolicy(bool settleWholeHeadstageA, bool settleWholeHeadstageB, bool settleWholeHeadstageC, bool settleWholeHeadstageD, bool settleAllHeadstages)

Since stimulation on one channel can create recording artifacts on nearby channels due to capacitive coupling in headstages, connectors, and/or electrode arrays, it is often useful to perform fast artifact recovery (i.e., amplifier settling) across an entire headstage. This function configures **WireInGlobalSettleSelect** (see above) to enable various modes of global amplifier settling.

void setAmpSettleMode(bool useFastSettle)

Selects the amplifier settle (fast artifact recovery) mode for all connected RHS chips. If useFastSettle is true, the traditional fast settle method is used. If useFastSettle if false, the low frequency cutoff select method is used (recommended).

void setChargeRecoveryMode(bool useSwitch)

Selects the charge recovery mode for all connected RHS chips. If useSwitch is true, the charge recovery switches are used. If useSwitch if false, the current-limited charge recovery drivers are used.

void setCableDelay(BoardPort port, int delay)

Sets the delay for sampling the MISO line on a particular SPI port (PortA, PortB, PortC, or PortD), in integer clock steps, where each clock step is 1/2800 of a per-channel sampling period. Cable delay should be updated after any changes are made to the sampling rate, since cable delay calculations are based on the clock period.

Most users will probably find it more convenient to use **setCableLengthMeters** or **setCableLengthFeet** instead of using **setCableDelay** directly.

int getCableDelay(BoardPort port)

void getCableDelay(vector<int> &delays)

Returns the last delay set on a particular SPI port (PortA, PortB, PortC, or PortD), or all ports, in integer clock steps.

void setCableLengthMeters(BoardPort port, double lengthInMeters)

void setCableLengthFeet(BoardPort port, double lengthInFeet)

Sets the delay for sampling the MISO line on a particular SPI port (PortA, PortB, PortC, or PortD) based on the length of the cable between the FPGA and the RHS chip (in meters or feet). Cable delay should be updated after any changes are made to the sampling rate, since cable delay calculations are based on the clock period.

double estimateCableLengthMeters(int delay)

double estimateCableLengthFeet(int delay)

Based on a delay setting used in **setCableDelay**, these functions return the estimated cable length corresponding to this setting.

void setDspSettle(bool enabled);

Turns on or off the DSP settle function in the FPGA. (This only executes when CONVERT commands are executed by the RHS.)

void enableDataStream(int stream, bool enabled)

Enables (if enabled is true) or disables (if enabled is false) one of the eight available USB data streams (0-7).

int getNumEnabledDataStreams()

Returns the total number of enabled USB data streams.

ControllerType getType()

Returns the type of this controller. This should always return **ControllerStimRecord** for the RHS XEM7310 interface. The type is specified using the **ControllerType** enumeration; defined values are:

ControllerRecordUSB2 (not compatible with RHD XEM7310 Interface)
ControllerRecordUSB3 (not compatible with RHS XEM7310 Interface)
ControllerStimRecord

int maxNumDataStreams()

static int maxNumDataStreams(ControllerType type_)

Returns the total number of data streams for this controller's type (non-static) or a controller of the given type (static). This returns 8 for the RHS XEM7310 interface. The type of controller is specified with the **ControllerType** enumeration, defined above.

int maxNumSPIPorts()

static int maxNumSPIPorts(ControllerType type_)

Returns the maximum number of SPI ports for this controller's type (non-static) or a controller of the given type (static). This returns 4 for the RHS XEM7310 interface. The type of controller is specified with the **ControllerType** enumeration, defined above.

int boardMode()

static int boardMode(ControllerType type_)

Returns the board mode for this controller's type (non-static) or a controller of the given type (static). This returns 14 for the RHS XEM7310 interface. Note that this doesn't actually read the FPGA's digital input pins set on the physical controller (this is done through the **getBoardMode()** method) and instead just reports the software variable associated with this controller.

static int numAnalogIO(ControllerType type_, bool expanderConnected_)

Returns the number of Analog Inputs/Outputs for a controller of the given type. For the RHS XEM7310 interface, this will return 8 if an I/O expander is connected, otherwise this will return 2.

static int numDigitalIO(ControllerType type_, bool expanderConnected_)

Returns the number of Digital Inputs/Outputs for a controller of the given type. For the RHS XEM7310 interface, this will return 16 if an I/O expander is connected, otherwise this will return 2.

static string getAnalogInputChannelName(ControllerType type_, int channel_)

Returns the name of the specified Analog Input channel for the given type. For the RHS XEM7310 interface, this will return "ANALOG-IN-1" to "ANALOG-IN-8".

static string getAnalogOutputChannelName(ControllerType type_, int channel_)

Returns the name of the specified Analog Output channel for the given type. For the RHS XEM7310 interface, this will return "ANALOG-OUT-1" to "ANALOG-OUT-8".

static string getDigitalInputChannelName(ControllerType type_, int channel_)

Returns the name of the specified Digital Input channel for the given type. For the RHS XEM7310 interface, this will return "DIGITAL-IN-01" to "DIGITAL-IN-16".

static string getDigitalOutputChannelName(ControllerType type_, int channel_)

Returns the name of the specified Digital Output channel for the given type. For the RHS XEM7310 interface, this will return "DIGITAL-OUT-01" to "DIGITAL-OUT-16".

static string getAnalogIOChannelNumber(ControllerType type_, int channel_)

Returns the number of the specified Analog I/O channel for the given type. For the RHS XEM7310 interface, this will return "1" to "8".

static string getDigitalIOChannelNumber(ControllerType type_, int channel_)

Returns the number of the specified Digital I/O channel for the given type. For the RHS XEM7310 interface, this will return "01" to "16".

static string getBoardTypeString(ControllerType type_)

Returns the type of the controller as a string. For the RHS XEM7310 interface, this will return "ControllerStimRecord".

void setAllDacsToZero();

Set all DACs to midline value (zero). DAC values are only updated when SPI ports are running.

void setDacManual(int value)

Sets the manual AD5662 DAC control (DacManual) WireIn to value (0-65536). DAC values are only updated when SPI ports are running.

void setLedDisplay(const int* ledArray)

Sets the eight red LEDs on the Opal Kelly XEM7310 board according to a length-8 integer array.

void setSpiLedDisplay(const int* ledArray)

Sets the eight red LEDs on the front panel SPI ports according to a length-8 integer array.

void enableDac(int dacChannel, bool enabled)

Enables (if enabled is true) or disables (if enabled is false) AD5662 DACs connected to the FPGA.

void setDacGain(int gain)

Scales the digital signals to all eight AD5662 DACs by a factor of 2^{gain}, where gain is between 0 and 7.

void setAudioNoiseSuppress(int noiseSuppress)

Sets the noise slicing region for DAC channels 1 and 2 (i.e., audio left and right) to +/-16 x noiseSuppress LSBs, where noiseSuppress is between 0 and 127. This improves the audibility of weak neural spikes in noisy waveforms.

void selectDacDataStream(int dacChannel, int stream)

void selectDacDataChannel(int dacChannel, int dataChannel)

Assigns a particular data stream (0-7) and amplifier channel (0-15) to an AD5662 DAC channel (0-7). Setting stream to 8 selects the DacManual source.

void enableDacHighpassFilter(bool enable)

Enables (if enabled is true) or disables (if enabled is false) the first-order high-pass filters implemented in the FPGA on all eight DAC/comparator channels. These filters may be used to remove low-frequency local field potential (LFP) signals from neural signals to facilitate spike detection while still recording the complete wideband data.

void setDacHighpassFilter(double cutoff)

Sets a cutoff frequency (in Hz) for first-order high-pass filters implemented in the FPGA on all eight DAC/comparator channels.

void setDacThreshold(int dacChannel, int threshold, bool trigPolarity)

Sets a threshold level (0-65535) and trigger polarity for a low-latency FPGA threshold comparator on a DAC channel (0-7). The threshold parameter corresponds to the RHS chip ADC output value, where the 'zero' level is 32768 and the step size is 0.195 μV. If the trigger polarity is set to true, RHS signals equaling or rising above the threshold produce a high digital output. If the trigger polarity is set to false, RHS signals equaling for falling below the threshold produce a high digital output. If the corresponding DAC is disabled, the digital output will always be low.

void setTtlOutMode(bool mode1, bool mode2, bool mode3, bool mode4, bool mode5, bool mode6, bool mode7, bool mode8)

Sets the TTL digital output mode of the FPGA for digital out ports 1-8. If mode is set false, the digital output pin is controlled by the Digital Output Sequencer. If mode is set true, the output is controlled by the low-latency threshold comparator connected to the waveform routed to the corresponding DAC. Digital out ports 9-16 are always controlled by Digital Output Sequencers.

void flush()

Flush all remaining data out of the FIFO. (This function should only be called when SPI data acquisition has been stopped.)

bool readDataBlock(RHXDataBlock* dataBlock)

Reads a data block from the USB interface and stores the data into an **RHXDataBlock** object dataBlock. Returns true if data block was available.

long readDataBlocksRaw(int numBlocks, uint8_t* buffer)

Reads certain number of USB data blocks, if the specified number is available, and writes the raw bytes to a buffer. Returns the total number of bytes read.

bool readDataBlocks(int numBlocks, deque<RHXDataBlock*> &dataQueue)

Reads a specified number of data blocks from the USB interface and appends them to a queue. Returns true if the requested number of data blocks were available.

int queueToFile(deque<RHXDataBlock*> &dataQueue, ofstream &saveOut)

Writes the contents of a data block queue to a binary output stream. Returns number of data blocks appended to queue.

static int getBoardMode(okCFrontPanel* dev_)

int getBoardMode()

Reads four digital input pins on the FPGA (see **MC1 I/O Connections** section) and returns this as an integer. These pins will typically be hard-wired either high or low to encode a 4-bit number that identifies particular properties of the interface board. The static version of this method also requires a pointer to the **okCFrontPanel** device. This is typically used to read the board mode from the Controller and set an internal software variable, which is then reported with the **boardMode()** method.

int getNumSPIPorts(bool &expanderBoardDetected)

static int getNumSPIPorts(okCFrontPanel* dev_, bool isUSB3, bool &expanderBoardDetected, bool isRHS7310)

Reads four digital input pins on the FPGA (see **MC1 I/O Connections** section) and returns this as an integer. These pins will typically be hard-wired either high or low to encode a 4-bit number that identifies particular properties of the interface board. Sets **expanderBoardConnected** if the FPGA detects an I/O Expander. The static version of this method also requires a pointer to the **okCFrontPanel** device and boolean input **isRHS7310** which should always be set true for the RHS XEM7130 Interface.

void enableDacReref(bool enabled)

Enables or disables DAC re-referencing, where a selected amplifier channel is subtracted from all DACs in real time.

void enableDcAmpConvert(bool enable)

Enables or disables DC amplifier conversion.

void setDacRerefSource(int stream, int channel)

Selects an amplifier channel from a particular data stream to be subtracted from all DAC signals.

StreamChannelPair streamChannelFromWaveName(const string &waveName)

Returns the stream and channel number within that stream required to access an amplifier input specified by the string waveName.

Amplifier channels are specified as follows: “A-000”, “B-005”, “C-031”, etc. The first letter is a port designation (A-D in the 128-channel Stimulation/Recording Controller), followed by a hyphen, followed by a three-digit channel number ranging from 000-031 in RHS systems.

StreamChannelPair is a struct containing { int **stream**; int **channel**; } Invalid waveNames return **stream** = -1, **channel** = -1.

int pipeReadError()

Returns the Opal Kelly error code (see okFrontPanel.h for details) when a read from a BTPipeOut fails, often attributed to connection or interference problems on USB.

void setExtraStates(unsigned int extraStates)

Setting extraStates to a value greater than zero extends the number of internal clock cycles that CS is high from ten to ten plus the value of this variable. Under normal operation, this should be left at zero.

RHXRegisters Class Reference

This class creates and manages a data structure representing the internal RAM registers on an RHS chip, and generates command lists to configure the chip and perform other functions. Changing the value of variables within an instance of this class does not directly affect an RHS chip connected to the FPGA; rather, a command list must be generated from this class and then downloaded to the FPGA board using **RHXController::uploadCommandList**. Typically, one instance of **RHXRegisters** will be created for each RHS chip attached to the RhythmStim USB-7310 interface. However, if all chips will receive the same MOSI commands, then only one instance of **RHXRegisters** is required.

The public member functions of the **RHXRegisters** class are listed below.

RHXRegisters(ControllerType type_, double sampleRate_, StimStepSize stimStep_)

Constructor. Sets RHS register variables to default values for the given **ControllerType**. For RHS systems, this **ControllerType** should be ControllerStimRecord. The desired stimulation step size must be specified using the **StimStepSize** enumeration; defined values are:

StimStepSize10nA
StimStepSize20nA
StimStepSize50nA
StimStepSize100nA
StimStepSize200nA
StimStepSize500nA
StimStepSize1uA
StimStepSize2uA
StimStepSize5uA
StimStepSize10uA

void setDigOutLow(DigOut pin)

void setDigOutHigh(DigOut pin)

void setDigiOutHiZ(DigOut pin)

Sets an auxiliary digital output variable to indicate a low, high, or high impedance (HiZ) output for a digital output pin specified by the **DigOut** enumeration; defined values are:

DigOut1
DigOut2
DigOutOD

void enableDsp(bool enabled)

Enables or disables DSP offset removal filter.

double setDspCutoffFreq(double newDspCutoffFreq)

Sets the DSP offset removal filter cutoff frequency as closely to the requested newDspCutoffFreq (in Hz) as possible; returns the actual cutoff frequency (in Hz).

double getDspCutoffFreq()

Returns the current value of the DSP offset removal cutoff frequency (in Hz).

void enableZcheck(bool enabled)

Enables or disables impedance checking mode.

void setZcheckDacPower(bool enabled)

Powers up or down impedance testing DAC.

void setZcheckScale(ZcheckCs scale)

Selects the series capacitor C_S used to convert the voltage waveform generated by the on-chip DAC into an AC current waveform that stimulates a selected electrode for impedance testing. The capacitor is specified using the **ZCheckCs** enumeration; defined values are:

ZcheckCs100fF
ZcheckCs1pF
ZcheckCs10pF

int setZcheckChannel(int channel)

Selects the amplifier channel (0-15) for impedance testing.

void setAmpPowered(int channel, bool powered)

void powerUpAllAmps()

void powerDownAllAmps()

Powers up or down selected AC-coupled high-gain amplifiers on RHS chip.

void setDCAmpPowered(int channel, bool powered)

void powerUpAllDCAmps()

void powerDownAllDCAmps()

Powers up or down selected DC-coupled low-gain amplifiers on RHS chip.

void setStimEnable(bool enable)

Enables or disables stimulation globally on the entire chip by setting appropriate values for Registers 32 and 33.

void setStimStepSize(StimStepSize step)

Sets the stimulation current DAC step size for the entire chip. See Constructor description above for list of valid step sizes.

static double stimStepSizeToDouble(StimStepSize step)

Returns the numerical value (in amps) of a stimulation step size enumeration parameter. See Constructor description above for list of valid step sizes.

int setPosStimMagnitude(int channel, int magnitude, int trim)

int setNegStimMagnitude(int channel, int magnitude, int trim)

Sets the positive or negative stimulation magnitude (0 to 255, in DAC steps) and trim value (-128 to +127) for a particular stimulator channel (0-15). Returns -1 if any input parameters are out of range; otherwise returns 0.

void setChargeRecoveryCurrentLimit(ChargeRecoveryCurrentLimit limit)

Sets the charge recovery current limit globally. The currentLimit parameter is set by the **ChargeRecoveryCurrentLimit** enumeration; defined values are:

CurrentLimit1nA
CurrentLimit2nA
CurrentLimit5nA
CurrentLimit10nA
CurrentLimit20nA
CurrentLimit50nA
CurrentLimit100nA
CurrentLimit200nA
CurrentLimit500nA
CurrentLimit1uA

static double chargeRecoveryCurrentLimitToDouble(ChargeRecoveryCurrentLimit limit)

Returns the numerical value (in amps) of a charge recovery current limit enumeration parameter. See above for list of valid limits.

double setChargeRecoveryTargetVoltage(double vTarget)

Sets the target voltage for current-limited charge recovery. The parameter vTarget should specify a voltage in the range of -1.225 to +1.215 (with units of Volts). Returns the actual value of the target voltage (limited by DAC resolution).

int getRegisterValue(int reg)

Returns the value of a selected RHS RAM register (0-250), based on the current register variables in the class instance.

double setUpperBandwidth(double upperBandwidth)

double setLowerBandwidth(double lowerBandwidth, int select)

Sets the on-chip RH1, RH2, and RL DAC values appropriately to set a particular amplifier bandwidth (in Hz). Returns an estimate of the actual bandwidth achieved. If select = 1, the “A version” of the lower bandwidth is set; this is the version used in normal recording. If select = 0, the “B version” of the lower bandwidth is set; this is the version used to recover from stimulation artifacts if fast artifact recovery is performed using low frequency cutoff shifting.

unsigned int createRHXCommand(RHXCommandType commandType)

unsigned int createRHXCommand(RHXCommandType commandType, unsigned int arg1)

unsigned int createRHXCommand(RHXCommandType commandType, unsigned int arg1, unsigned int arg2)

unsigned int createRHXCommand(RHXCommandType commandType, unsigned int arg1, unsigned int arg2, unsigned int uFlag, unsigned int mFlag)

Returns a 32-bit MOSI command based on the **RHXCommandType** enumeration; defined values are:

RHXCommandConvert
RHXCommandCalibrate
RHXCommandCalClear
RHXCommandRegWrite
RHXCommandRegRead
RHXCommandComplianceReset

CONVERT and READ commands use arg1 to specify a register or channel; WRITE commands use arg1 and arg2 to specify a register and value, respectively. The optional arguments uFlag and mFlag should have a value of 0 or 1. Setting these to 1 asserts the U or M flags to update all triggered registers or clear the compliance monitor register, respectively. (See the “SPI Command Words” section of the RHS2116 chip datasheet for more details.)

int createCommandListRHSRegisterConfig(vector<unsigned int> &commandList, bool updateStimParams)

Creates a list of 128 commands to program most RAM registers on an RHS chip, read those values back to confirm programming, read ROM registers, and (if updateStimParams = true) update stimulation amplitudes (Registers 64-79 and 96-111) and other charge-recovery parameters (Registers 34-37). Returns the length of the command list.

int createCommandListRHSRegisterRead(vector<unsigned int> &commandList)

Creates a list of 128 commands to read all registers from an RHS chip without changing any values. Returns the length of the command list.

int createCommandListZcheckDac(vector<unsigned int> &commandList, double frequency, double amplitude)

Creates a list of up to 8192 commands to generate a sine wave of particular frequency (in Hz) and amplitude (in DAC steps, 0-128) using the on-chip impedance testing voltage DAC. If frequency is set to zero, a DC baseline waveform is created, which can be used when impedance testing is disabled to minimize on-chip noise. This function returns the length of the command list.

int createCommandListDummy(vector<unsigned int> &commandList, int n, unsigned int cmd)

Creates a list of dummy commands with a specific command. Returns the length of the command list (which should be n).

vector<double> getDspFreqTable()

static vector<double> getDspFreqTable(double sampleRate_)

Returns a size-16 vector containing all possible cutoff frequencies for the on-chip DSP offset removal filter (a one-pole highpass filter). The static function requires the system’s sample rate as an input parameter.

int createCommandListSetStimMagnitudes(vector<unsigned int> &commandList, int channel, int posMag, int posTrim, int negMag, int negTrim)

Creates a list of 128 commands to set the positive and negative stimulation magnitude and trim parameters for a single channel. Returns the length of the command list.

int createCommandListConfigChargeRecovery(vector<unsigned int> &commandList, ChargeRecoveryCurrentLimit currentLimit, double targetVoltage)

Creates a list of 128 commands to set the charge recovery current limit and target voltage. Returns the length of the command list. The parameter targetVoltage should fall within the range of -1.225 to +1.215 (units: Volts).

int maxCommandLength()

static int maxCommandLength(ControllerType type)

Returns size of on-FPGA auxiliary command RAM banks. For all RHS interfaces, this returns 8192.

int maxNumChannelsPerChip()

static int maxNumChannelsPerChip(ControllerType type)

Returns maximum number of amplifier channels per chip. For all RHS interfaces, this returns 16.

RHXDataBlock Class Reference

This class creates a data structure storing 128 data samples from a RhythmStim USB-7310 FPGA interface controlling up to eight RHS chips. Typically, instances of RHXDataBlock will be created dynamically as data becomes available over the USB interface and appended to a queue that will be used to stream the data to disk or to a GUI display.

The public member functions of the RHXDataBlock class are listed below.

RHXDataBlock(ControllerType type_, int numDataStreams_)

RHXDataBlock(const RHXDataBlock &obj)

Constructor. Allocates memory for a data block supporting the specified number of data streams.

~RHXDataBlock()

Destructor.

int samplesPerDataBlock()

static int samplesPerDataBlock(ControllerType type_)

Returns the number of samples in a single data block, which is always 128.

static unsigned int dataBlockSizeInWords(ControllerType type_, int numDataStreams)

unsigned int calculateDataBlockSizeInWords()

Returns the size of a USB data block (in 16-bit words) when numDataStreams data streams (0-8) are enabled.

void fillFromUsbBuffer(uint8_t* usbBuffer, int blockIndex)

Fills the data block with raw data from the nth data block in a USB input buffer in an RHXController object. Setting blockIndex to 0 selects the first data block in the buffer, setting blockIndex to 1 selects the second data block, etc.

void print(int stream)

Prints the contents of RHS registers from a selected USB data stream (0-7) to the console. This function assumes that the command string generated by RHXRegisters::createCommandListRHSRegisterConfig has been uploaded to the AuxCmd1 slot.

static bool checkUsbHeader(const uint8_t* usbBuffer, int index, ControllerType type_)

bool checkUsbHeader(const uint8_t* usbBuffer, int index)

Checks the first 64 bits of USB header against the fixed RhythmStim USB-7310 “magic number” to verify proper data synchrony.

void write(ofstream &saveOut, int numDataStreams)

Writes the contents of a data block object to a binary output stream in little endian format (i.e., least significant byte first).

uint32_t timeStamp(int t)

Returns the timestamp at index t (0-127) of this data block.

int amplifierData(int stream, int channel, int t)

Returns a single sample of amplifier data from the specified channel (0-15) on the specific stream (0-7), corresponding to the timestamp at index t (0-127) of this data block.

int auxiliaryData(int stream, int channel, int t)

Returns a single sample of auxiliary data from the specified channel (0-3) on the specific stream (0-7), corresponding to the timestamp at index t (0-127) of this data block.

int boardAdcData(int channel, int t)

Returns a single sample of Analog Input data from the specified on-board ADC channel (0-7), corresponding to the timestamp at index t (0-127) of this data block.

int ttlIn(int channel, int t)

Returns a single sample of TTL Input data from the specified on-board Digital Input channel (0-15), corresponding to the timestamp at index t (0-127) of this data block. Values are 1 (TTL high) or 0 (TTL low).

int ttlOut(int channel, int t)

Returns a single sample of TTL Output data from the specified on-board Digital Output channel (0-15), corresponding to the timestamp at index t (0-127) of this data block. Values are 1 (TTL high) or 0 (TTL low).

int dcAmplifierData(int stream, int channel, int t)

Returns a single sample of DC amplifier data from the specified channel (0-15) on the specific stream (0-7), corresponding to the timestamp at index t (0-127) of this data block.

int complianceLimit(int stream, int channel, int t)

Returns if compliance limit is flagged from the specified channel (0-15) on the specific stream (0-7), corresponding to the timestamp at index t (0-127) of this data block.

int stimOn(int stream, int channel, int t)

Returns if stimulation is active on the specified channel (0-15) on the specific stream (0-7), corresponding to the timestamp at index t (0-127) of this data block.

int stimPol(int stream, int channel, int t)

Returns the polarity of stimulation on the specified channel (0-15) on the specific stream (0-7), corresponding to the timestamp at index t (0-127) of this data block. 1 indicates positive current, 0 indicates negative current.

int ampSettle(int stream, int channel, int t)

Returns if amplifier settle is active on the specified channel (0-15) on the specific stream (0-7), corresponding to the timestamp at index t (0-127) of this data block. 1 indicates active amplifier settle, 0 indicates inactive amplifier settle.

int chargeRecov(int stream, int channel, int t)

Returns if charge recovery is active on the specified channel (0-15) on the specific stream (0-7), corresponding to the timestamp at index t (0-127) of this data block. 1 indicates active charge recovery, 0 indicates inactive charge recovery.

int boardDacData(int channel, int t)

Returns a single sample of Analog Output data from the specified on-board DAC channel (0-7), corresponding to the timestamp at index t (0-127) of this data block.

static int blocksFor30Hz(AmplifierSampleRate rate)

Returns the number of RHX data blocks that should be read over the USB interface each time for an approximate USB read rate of 30 Hz.

static int maxChannelsPerStream()

Returns the maximum number of amplifier channels that can be present per data stream. This returns 32 (although for RHS interfaces, no data stream will actually ever have more than 16 amplifier channels).

static int channelsPerStream(ControllerType type_)

int channelsPerStream()

Returns the number of amplifier channels that are present per data stream. For all RHS interfaces, this returns 16.

static int numAuxChannels(ControllerType type_)

int numAuxChannels()

Returns the number of auxiliary channels that are present per data stream. For all RHS interfaces, this returns 4.

int getChipID(int stream, int auxCmdSlot, int &register59Value)

Returns this chip's ID (-1 for no chip, 32 for RHS). This assumes that a command list created by **RHXRegisters::createCommandListRHSRegisterConfig** has been uploaded and run, and the resulting RHXDataBlock read first.

static uint64_t headerMagicNumber(ControllerType type_)

uint64_t headerMagicNumber()

Returns the RHS header magic number. For the RHS XEM7310 interface, this is 0x8d542c8a49712f0b.

Example Usage

A simple **main.cpp** C++ program presented below opens an Opal Kelly XEM7310 board, configures it with the RhythmStim USB-7310 FPGA configuration bitfile (ConfigRHSController_7310.bit), and sets the clock generator for a 20 kS/s/channel sampling rate. The MISO sampling delay is set for a 3-foot cable. The program then modifies the default values of the RHS registers, generates command sequences for the four auxiliary command slots, and uploads these commands to the FPGA. Two command sequences, in AuxCmd1 and AuxCmd2, are executed once in a 128-sample SPI run. Register data from this brief run is read over the USB interface, and displayed on the console to confirm RHS register settings. Next is a one-second run, and complete data from the one-second acquisition is then saved in binary format to a file on disk.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <queue>

using namespace std;

#include "okFrontPanel.h"
#include "rhxcontroller.h"
#include "rhxregisters.h"
#include "rhxdatablock.h"

int main(int argc, char* argv[])
{
    // Create RHX Controller with 20 kHz per-amplifier sampling rate.
    RHXController *rhxController = new RHXController(ControllerStimRecord,
                                                     SampleRate20000Hz);

    // Open the first detected Opal Kelly device, load RhythmStim USB-7310 bitfile.
    vector<string> availableDevices = rhxController->listAvailableDeviceSerials();
    rhxController->open(availableDevices[0]);

    // Load RhythmStim USB-7310 bitfile and initialize
    rhxController->uploadFpgaBitfile("ConfigRHSController_7310.bit");
    rhxController->initialize();
    rhxController->enableDataStream(0, true);

    // We can set the MISO sampling delay which is dependent on the sample rate.
    // We assume a 3-foot cable.
    rhxController->setCableLengthFeet(PortA, 3.0);

    // Let's turn one LED on to indicate that the program is running.
    int ledArray[8] = {1, 0, 0, 0, 0, 0, 0, 0};
    rhxController->setLedDisplay(ledArray);

    // Set up an RHX register object.
    RHXRegisters *chipRegisters = new RHXRegisters(rhxController->getType(),
                                                   rhxController->getSampleRate());

    // Create command lists to be uploaded to auxiliary command slots.
    int commandSequenceLength;
    vector<unsigned int> commandList;

    // First, let's create a command list for the AuxCmd1 slot to configure
    // and read back the RHS chip registers.

    // Before generating register configuration command sequence, set
    // amplifier bandwidth parameters.
```

```
double dspCutoffFreq;
dspCutoffFreq = chipRegisters->setDspCutoffFreq(10.0);  // 10 Hz DSP cutoff
cout << "Actual DSP cutoff frequency: " << dspCutoffFreq << " Hz" << endl;

chipRegisters->setLowerBandwidth(1.0);       // 1.0 Hz lower bandwidth
chipRegisters->setUpperBandwidth(7500.0);    // 7.5 kHz upper bandwidth

commandSequenceLength =
    chipRegisters->createCommandListRHSRegisterConfig(commandList, false);
// Upload command sequence to AuxCmd1.
rhxController->uploadCommandList(commandList, RHXController::AuxCmd1);
rhxController->selectAuxCommandLength(RHXController::AuxCmd1, 0,
   commandSequenceLength - 1);
// rhxController->printCommandList(commandList); // optionally, print command list

// Next, we'll create a command list for the AuxCmd2 slot.  This command
// will create a 1 kHz, full-scale sine wave for impedance testing.
commandSequenceLength =
    chipRegisters->createCommandListZcheckDac(commandList, 1000.0, 128.0);
rhxController->uploadCommandList(commandList, RHXController::AuxCmd2);
rhxController->selectAuxCommandLength(RHXController::AuxCmd2, 0,
    commandSequenceLength - 1);
// rhxController->printCommandList(commandList); // optionally, print command list

// We'll upload dummy command sequences to slots AuxCmd3 and AuxCmd4.
commandSequenceLength =
    chipRegisters->createCommandListDummy(commandList, 128,
        chipRegisters->createRHXCommand(RHXRegisters::RHXCommandRegRead, 255));
rhxController->uploadCommandList(commandList, RHXController::AuxCmd3);
rhxController->uploadCommandList(commandList, RHXController::AuxCmd4);
rhxController->selectAuxCommandLength(RHXController::AuxCmd3, 0,
    commandSequenceLength - 1);
rhxController->selectAuxCommandLength(RHXController::AuxCmd4, 0,
    commandSequenceLength - 1);
// rhxController->printCommandList(commandList); // optionally, print command list


// Since our longest command sequence is 128 commands, let's just run the SPI
// interface for 128 samples.
rhxController->setMaxTimeStep(128);
rhxController->setContinuousRunMode(false);

// Start SPI interface.
rhxController->run();
//  Wait for the 128-sample run to complete.
while (rhxController->isRunning()) { }

// Read the resulting single data block from the USB interface.
RHXDataBlock *dataBlock =
    new RHXDataBlock(rhxController->getType(),
                     rhxController->getNumEnabledDataStreams());
rhxController->readDataBlock(dataBlock);

// Display register contents from data stream 0.
dataBlock->print(0);

// Let's save one second of data to a binary file on disk.
ofstream saveOut;
saveOut.open("binary_save_file.dat", ios::binary | ios::out);

deque<RHXDataBlock*> dataQueue;
```

```
    // Run for one second.
    rhxController->setMaxTimeStep(20000);
    rhxController->run();

    bool usbDataRead;
    do {
        usbDataRead = rhxController->readDataBlocks(1, dataQueue);
        if (dataQueue.size() >= 50) {   // save 50 data blocks at a time
            rhxController->queueToFile(dataQueue, saveOut);
        }
    } while (usbDataRead || rhxController->isRunning());

    rhxController->queueToFile(dataQueue, saveOut);

    saveOut.close();

    // Turn off LED.
    ledArray[0] = 0;
    rhxController->setLedDisplay(ledArray);

    delete dataBlock;
    delete chipRegisters;
    delete rhxController;

    return 0;
}
```

An elaborated version of this **main.cpp** program file is included with the RhythmStim USB-7310 API distribution files.

Reading binary data into MATLAB

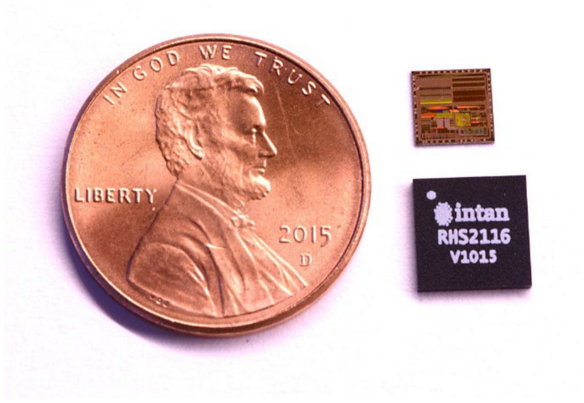
The C++ program above saves data from a single data stream to a binary output file. The following MATLAB code reads a saved file for the case where only one data stream is active:

```
fid = fopen(filename, 'r');

s = dir(filename);
filesize = s.bytes;

% allocate space to read the entire file
data = zeros(filesize, 1, 'uint8');

% read the entire file
data = fread(fid, filesize, 'uint8=>uint8');
fclose(fid);

% convert the remaining data from bytes to 2-byte unsigned integers
data = typecast(data, 'uint16');
swapbytes(data);

% convert uint16 datatype to double datatype
data = double(data);

L = length(data);

data = reshape(data, 51, L/51);

timestamp = data(1,:)';
amplifier_data = data(2:17,:)';
dc_amplifier_data = data(18:33,:)';
boardADC_data = data(34:41,:)';
boardDAC_data = data(42:49,:)';
TTLin = data(50,:)';
TTLout = data(51,:)';
```

Contact Information

This datasheet is meant to acquaint engineers and scientists with the RhythmStim USB-7310 USB/FPGA interface code developed at Intan Technologies. We value feedback from potential end users. We can discuss your specific needs and suggest a solution tailored to your applications.

For more information, contact Intan Technologies at:



www.intantech.com
support@intantech.com