
openPET

OpenPET Developer's Guide Documentation

Release 2.3.2

OpenPET

May 03, 2017

1	Abbreviations	1
2	Overview of OpenPET Framework	3
2.1	Introduction	3
2.2	System Hardware, Firmware, and Software Structures	10
3	Specifications	15
3.1	Introduction	15
3.2	Timing Signals	15
3.3	Power	19
3.4	Crate	19
3.5	Support Board	19
3.6	Detector Board	27
3.7	Coincident Interface & Multiplexor Boards (CI-1 & MB-1)	34
3.8	Host PC Interface Board	34
3.9	Firmware	36
4	Implementations	39
4.1	Introduction	39
4.2	System Addressing	39
4.3	System Commands and Responses	39
4.4	Data Modes	59
4.5	Serial Peripheral Interface	73
4.6	List Mode Data	80
4.7	Bootup Sequence	82
4.8	System Reset	84
5	Source Code	85
5.1	Introduction	85
5.2	Firmware and Software	85
5.3	OpenPET Library	85
5.4	OpenPET Control and Analysis Tools	98
6	OpenPET Firmware Parameters	101
6.1	Main FPGA	103
6.2	IO FPGA	107
6.3	Detector Board FPGA	111

7	OpenPET Software Interface	115
7.1	QuickUSB	115
7.2	Ethernet	120
8	System Troubleshooting	125
8.1	Firmware	125
8.2	Embedded Software	127
9	What Exists Now	129
10	How To Be An OpenPET Developer	131
10.1	Signup as Developer	131
10.2	BitBucket Repository	131
10.3	Code Style Requirements	133
10.4	C Programming (NOIS II)	133
11	Appendices	135
11.1	Application Notes	135
12	Acknowledgements	137
13	References	139
14	Index	141

CHAPTER 1

Abbreviations

SB: Support Board
DB: Detector Board
MB: Multiplexer Board (also called high-speed data transceiver board)

CI: Coincidence Interface Board
CU: Coincidence Unit
DU: Detector Unit
CDU: Coincidence/Detector Unit

CUC: Coincidence Unit Controller
DUC: Detector Unit Controller
CDUC: Coincidence/Detector Unit Controller

FPGA: Field-programmable gate array
NIOS II: A 32-bit embedded-processor architecture designed specifically for the Altera FPGAs
VHDL: VHSIC hardware description language

CRC: Cyclic redundancy check

FSM: Finite State Machine

Overview of OpenPET Framework

Introduction

This document describes the OpenPET electronics system in order to help enable other individuals to contribute to the development of this system. The purpose of the OpenPET electronics is to provide a system that can be used by a large variety of users, primarily people who are developing prototype nuclear medical imaging systems. These electronics must be extremely flexible, as the type of detector, camera geometry, definition of event words, and algorithm for creating the event word given the detector outputs will vary from camera to camera. This implies that users must be able to modify the electronics easily, which further implies that they have easy access to documentation, including the schematics and documents needed to fabricate the circuit boards (Gerber files, bill of materials, etc.) and source code (for both firmware and software). They also need support in the form of instructions, user manuals, and a knowledge base, and they want fabricated circuit boards to be readily available.

The OpenPET system includes “open source” hardware, firmware, and software that will be expanded by LBNL with the help of an active community of developers.

System Overview

The OpenPET system architecture is shown in [Fig. 2.1](#). There are four types of custom electronics boards in the system: the Detector Board (DB), the Support Board (SB), the Coincidence Interface Board (CI), and the Multiplexer Board (MB). The Support Board plays two roles in the system, depending on the firmware.

The general data flow is that analog signals from detector modules provide the inputs to the Detector Board. This board processes the analog signals to create a singles event word, which is a digital representation of this single gamma ray interaction. The singles event words are passed to the Support Board loaded with detection firmware, whose main function here is to multiplex the singles event words from multiple Detector Boards. The singles event words are then passed through the Coincidence Interface Board to the Multiplexer Board, which can provide a further layer of multiplexing for singles event words, if necessary. Next the multiplexed singles event words are passed to another Support Board loaded with coincidence firmware, which searches through the singles event words for pairs that are in time coincidence and then forms coincidence event words. These coincidence event words are then passed to the Host PC. Optionally, the Support Board with coincidence firmware can act as a multiplexer and pass unaltered singles event words to the Host PC.

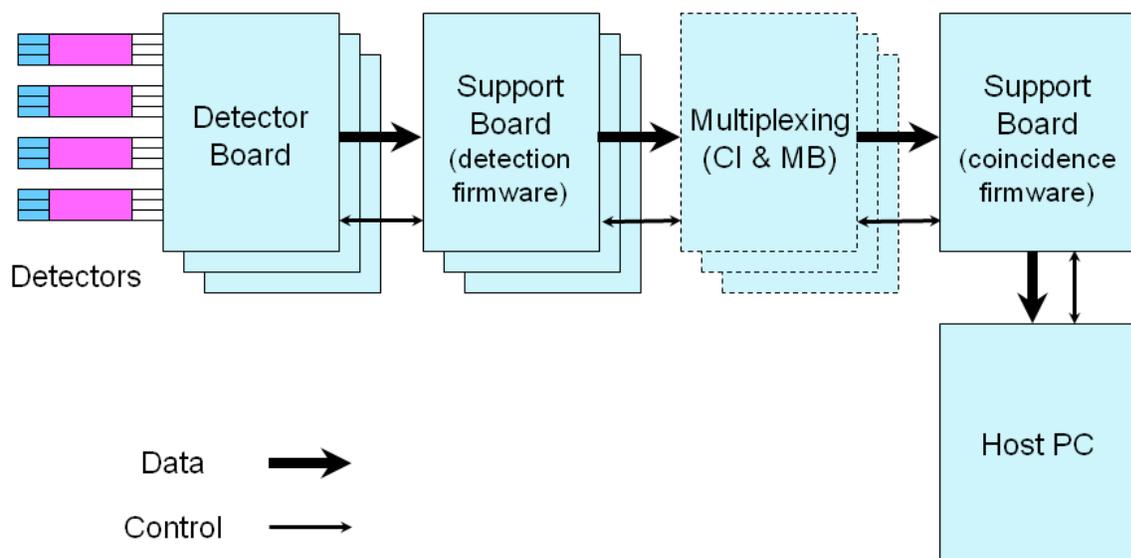


Fig. 2.1: Schematic of the OpenPET system architecture.

The OpenPET components are housed in an assembly whose form factor is the same as a 12-slot VME crate that accommodates 6U boards. A Support Board essentially replaces the backplane of the VME crate and all the other boards plug into it. The plug-in boards have the same form factor as a VME 6U board, except that the position of the connectors is offset (compared to true VME boards) to prevent OpenPET boards from being plugged into standard VME systems and vice versa.

Definitions

Support Crate

A Support Crate (Fig. 2.2) is conceptually similar to a VME crate (with controller), namely an intelligent support structure that “functional” boards can be plugged into. It consists of a mechanical frame with 12 plug-in slots, a Support Board (that has a considerable amount of programmable processing power and also acts as a backplane), power supplies, cooling fans, and appropriate boards plugged into slots 9-11. Slots 0-8 are vacant. Slot 9 holds a Host PC Interface Board, which is used to communicate with the Host PC; this board is optional. Slot 10 holds a User IO Board, which allows users to interface to external components such as EKG signals and motor controllers; this board is optional. Slot 11 holds a Debugging Board, which has interfaces to logic analyzers, a number of diagnostic LEDs, an external clock input, and a JTAG connector; this board is optional. Some ancillary components (such as DRAM memory and a QuickUSB board) are also necessary for a functioning Support Crate. By programming the Support Board with appropriate (but different) firmware, the Support Crate becomes part of either a Detector Unit or a Coincidence Unit.

Detector Unit

A Detector Unit (DU), as shown in Fig. 2.2, consists of a Support Crate with between one and eight Detector Boards plugged into slots 0-7. Each Detector Board can process up to 32 analog input signals. A Detector Unit can therefore process up to 256 analog signals, which corresponds to 64 conventional block detector modules with 4 analog outputs per module. In a Small System (Fig. 2.5), Slot 8 is typically empty (if data is transferred to the host computer through USB or Ethernet via the Host PC Interface Board plugged into Slot 9). In a Standard (Fig. 2.4) or Large System (Fig. 2.5), a Coincidence Interface Board must be plugged into Slot 8 of the Detector Unit. There are two versions of the Coincidence Interface Board: Coincidence Interface Board-1 (CI-1) for the Standard System and Coincidence Interface Board-8 (CI-8) for the Large System. At present, the Coincidence Interface Board-8 has not been designed or specified. These boards transfer event data and bidirectional control data between the Detector Unit and a Coincidence Unit.

In a Detector Unit, the Support Board that acts as a backplane for the Support Crate is loaded with detection firmware. In this case, the Support Board with related firmware and software is called a **Detector Unit Controller** (DUC).

In a Small System, the Support Board in the Detector Unit is programmed to multiplex outputs from the Detector Boards, process coincident events, and pass the coincident events to the host computer. It can also be programmed to multiplex singles events and pass them to the host computer. In a Standard or Large System, the Support Board in the Detector Unit is programmed to multiplex singles events from the Detector Boards and forward them to a Coincidence Unit, as shown in Fig. 2.4 and Fig. 2.5. In this specialized case, the Support Board with related firmware and software is called a **Coincidence/Detector Unit Controller** (CDUC).

Coincidence Unit

The Coincidence Unit (CU) for a Standard System consists of a Support Crate with between one and eight Multiplexer Boards plugged into slots 0-7. The Support Board is loaded with firmware to perform the coincidence processing. Each Multiplexer Board communicates with one Detector Unit via the Coincidence Interface Board using a cable. Similar to the Coincidence Interface Board, there are two versions of the Multiplexer Board: Multiplexer Board-1 (MB-1) for the Standard System and Multiplexer Board-8 (MB-8) for the Large System. At present, the Multiplexer Board-8 has not

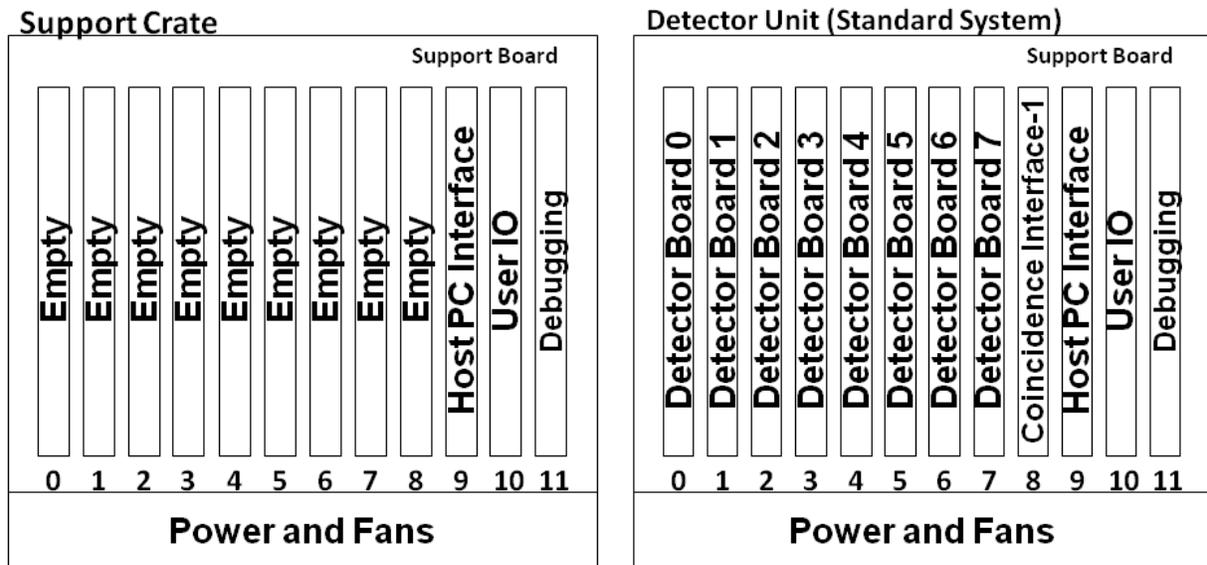


Fig. 2.2: Support Crate (left) and Detector Unit (right). A Detector Unit is a Support Crate with up to 8 Detector Boards (in Slots 0-7). For a Small System, Slot 8 is usually empty. For Standard and Large Systems, a Coincidence Interface Board must be plugged into Slot 8.

been designed or specified. The Coincidence Unit's Support Board is programmed to do the coincidence processing and pass the coincident events to the host computer, although it can also function as a multiplexer and forward singles events. Data is transferred to the Host PC either through USB or Ethernet via the Host PC Interface Board plugged into Slot 9.

In a Coincidence Unit, the Support Board that acts as a backplane for the Support Crate is loaded with coincidence firmware. In this case, the Support Board with related firmware and software is called a **Coincidence Unit Controller (CUC)**.

In the Coincidence Unit for the Standard System, each MB-1 connects with only one Detector Unit via a single cable, allowing up to 64 Detector Boards (or 512 block detector modules) in the system. In the Coincidence Unit for a Large System, the MB-8s plugged into slots 0-7 connect via cables (one cable per Detector Unit) with up to 8 Detector Units, allowing up to 512 Detector Boards (or 4096 block detector modules) in the system. The MB-8s are programmed to serve as multiplexers for events coming from up to 8 Detector Units. Due to the nature of multiplexing, this allows a larger number of channels to be serviced without increasing the maximum total event rate (singles or coincidence).

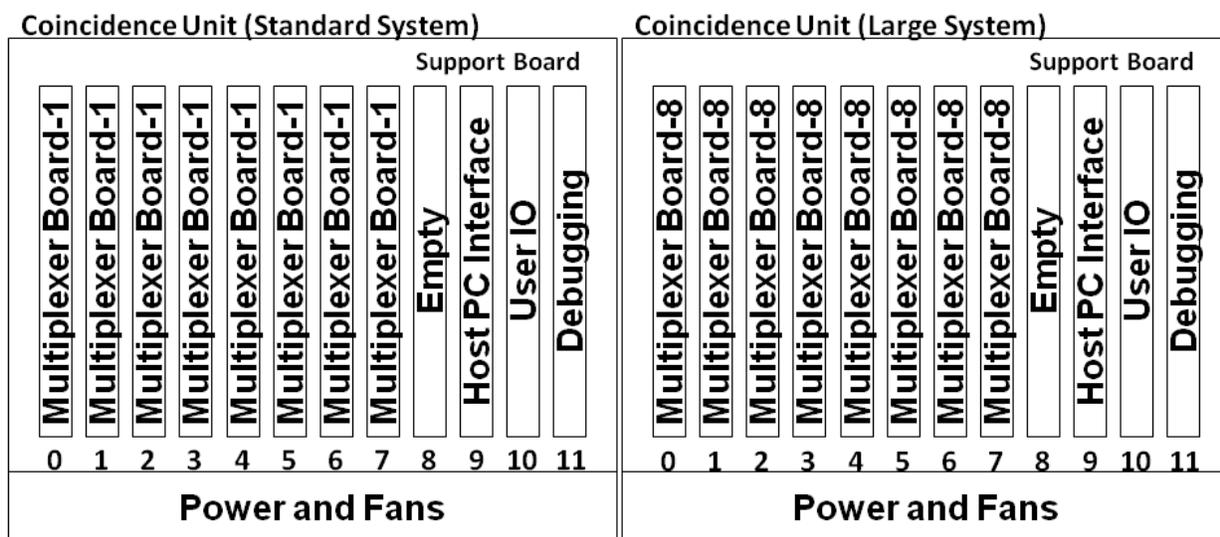


Fig. 2.3: Coincidence Unit for a Standard System (left) and Large System (right). For both Standard and Large Systems, a Coincidence Unit is a Support Crate with up to 8 Multiplexer Boards (in slots 0-7). In a Large System, the Multiplexer Boards function as multiplexers.

OpenPET can be configured either as a **Small System** (Fig. 2.5), **Standard System** (Fig. 2.4), or **Large System** (Fig. 2.5), with the difference largely due to the number of analog signals that can be read out. To determine which system configuration you need, see the [User's Guide System Configuration](#) section.

OpenPET Standard System

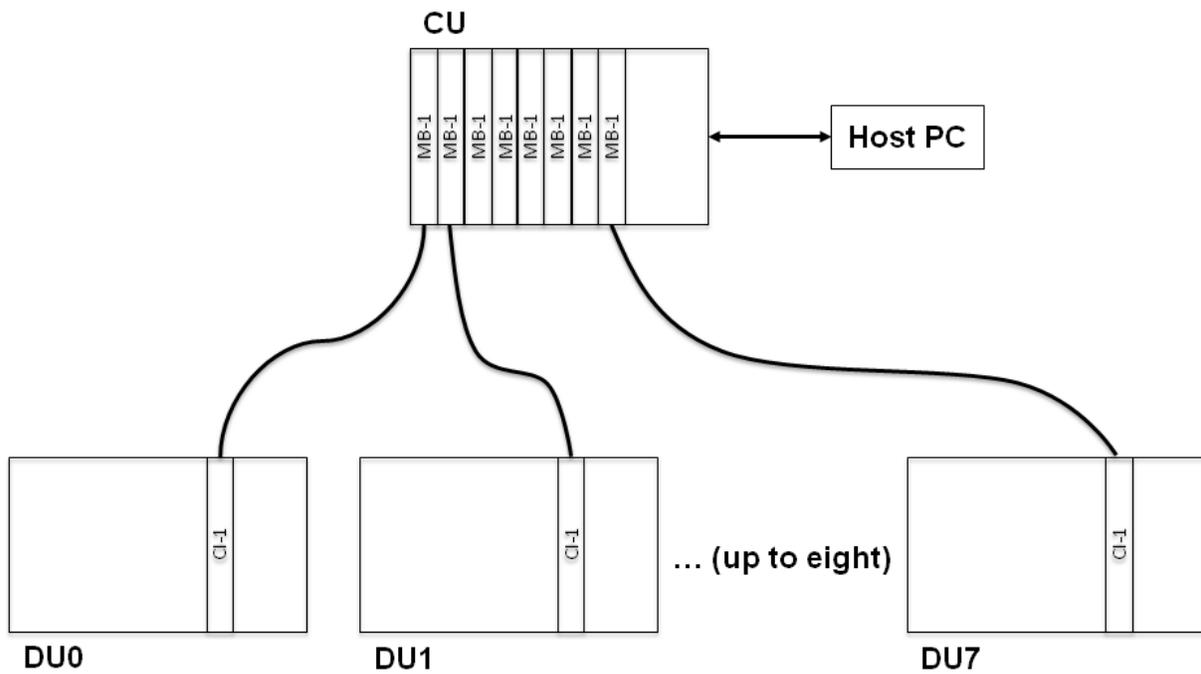


Fig. 2.4: Configuration of an OpenPET Standard System. Slots 0-7 in the Coincidence Unit contains a Multiplexer Board-1 that services a single DU. Slot 8 in the Detector Unit contains a Coincidence Interface Board-1 that transfers event data and bidirectional control between the DU and CU.

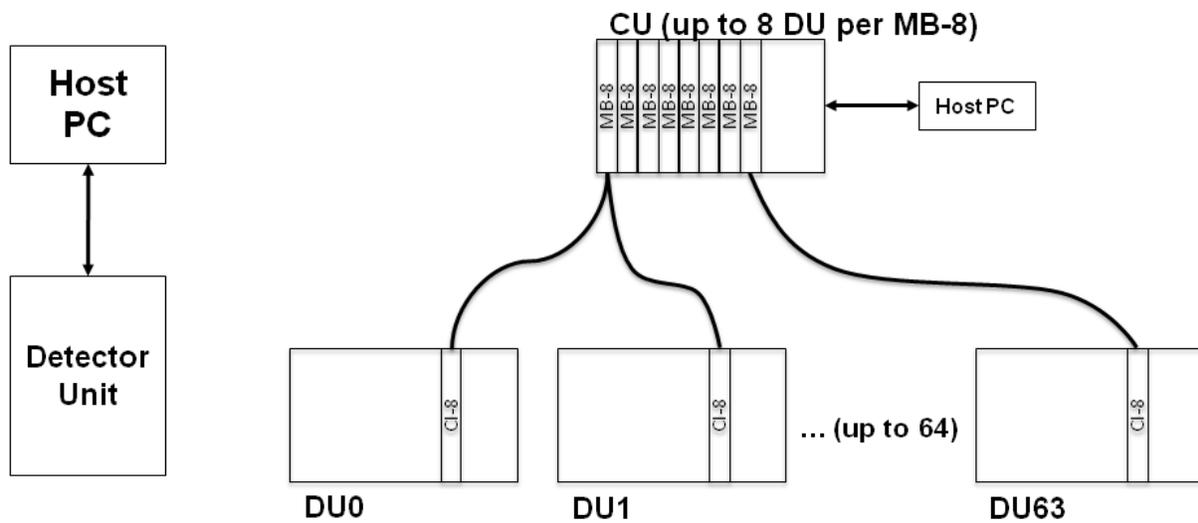


Fig. 2.5: Configuration of an OpenPET Small System (left) and a Large System (right). In a Small System, the Support Board in the Detector Unit performs the coincidence processing. In a Large System, each of slots 0-7 in the Coincidence Unit contains a Multiplexer Board-8 that functions as a multiplexer and services between 1 and 8 Detector Units. Slot 8 in the Detector Unit contains a Coincidence Interface Board-8 that transfers event data and bidirectional control between the DU and CU.

System Hardware, Firmware, and Software Structures

System Hardware Structure

The basic hardware structure for a standard OpenPET system is shown in [Fig. 2.6](#).

The host computer is equipped with an USB/Ethernet/Fiber-Optic connection and high-speed hard disk. It sends commands through USB/Ethernet/Fiber-Optic to the Coincidence Unit via the Host PC Interface board. It reads data from the Coincidence Unit through USB/Ethernet/Fiber-Optic and saves them to the hard disk in real time. Currently only a USB communication interface with the host computer has been implemented.

The command bus is shown in red in [Fig. 2.6](#). Four LVTTTL single ended lines are used to pass bi-directional command data between the boards. This command bus uses a custom serial digital bus protocol, consisting of a Clock line (CLK), Data In line (DI), Data Out line (DO), and Chip Selection line (CS). In the current protocol, the Chip Selection line is not utilized.

The data bus is shown in blue in [Fig. 2.6](#). Sixteen LVDS differential pairs are used to transfer list mode data upstream from the Detector Boards to the host computer.

In the general data flow, analog signals from detector modules are inputted into a Detector Board, which processes these analog signals to create a digital singles event word. The singles event words are passed to a Support Board loaded with detection firmware (SB-DUC), which multiplexes the singles event words from up to eight Detector Boards. The singles event words are then passed through a Coincidence Interface Board to a Multiplexer Board, which can provide a further layer of multiplexing for singles event words, if necessary. Next the multiplexed singles event words are passed to another Support Board loaded with coincidence firmware (SB-CUC), which searches through the singles event words from up to eight Multiplexer Boards for pairs that are in time coincidence and then forms coincidence event words. These coincidence event words are then passed to the host computer. Optionally, the Support Board with coincidence firmware can act as a multiplexer and pass unaltered singles event words to the host computer.

Each Detector Board has an Analog Front End (AFE) and a single Field-Programmable Gate Array (FPGA), as shown in [Fig. 2.6](#).

Several versions of the Detector Board will be designed, which will differ in how the analog inputs are processed by the front-end circuitries and in the number of analog input channels per Detector Board. The details of each design is described separately in [Detector Board](#) (page 27). In general, the Analog Front End includes circuitries for analog signal conditioning, analog-to-digital conversion, constant fraction or raising edge discrimination, and timing-to-digital conversion. The Detector Board FPGA performs the necessary computation for event processing, debugging, testing, and calibration tasks. Four single LVTTTL lines are used to program the Detector Board FPGA using the custom serial digital bus protocol described above. These signals are provided by the Support Board.

Each Support Board uses three FPGA: a master FPGA (Main FPGA) and two slave FPGAs (IO FPGA). The event multiplexing and forwarding is shared among the three FPGAs due to limited pin count.

The Support Board Main FPGA performs the logical functions of both the master FPGA and the support microprocessor. As the master FPGA, it primarily passes events from the slave IO FPGAs to the Coincidence Interface Board. As a support microprocessor, Nios II is used to program logical blocks in the FPGA to run executable files programmed in C. It is responsible for loading all firmware into the slave FPGAs and Detector Board FPGA, as well as the contents of other registers that are on the DB and SB. For instance, the support microprocessor interprets high-level commands sent from the host computer and executes these commands. This execution may involve controlling the Detector Board, such as by loading firmware into the DB FPGA on the DB; or it may involve higher-level functions, such as performing a calibration by instructing the DB to produce calibration data, analyzing the forthcoming calibration events, computing calibration parameters, and loading these parameters into the detector memory on the DB. It also monitors the event stream and can insert diagnostic information (such as event rates) into the event stream.

The slave IO FPGA passes events to the Main FPGA in different ways depending on the mode of the system. In Scope mode, it queues all the events with no loss of data. For singles events, the IO FPGA acts primarily as an arbitrator, taking

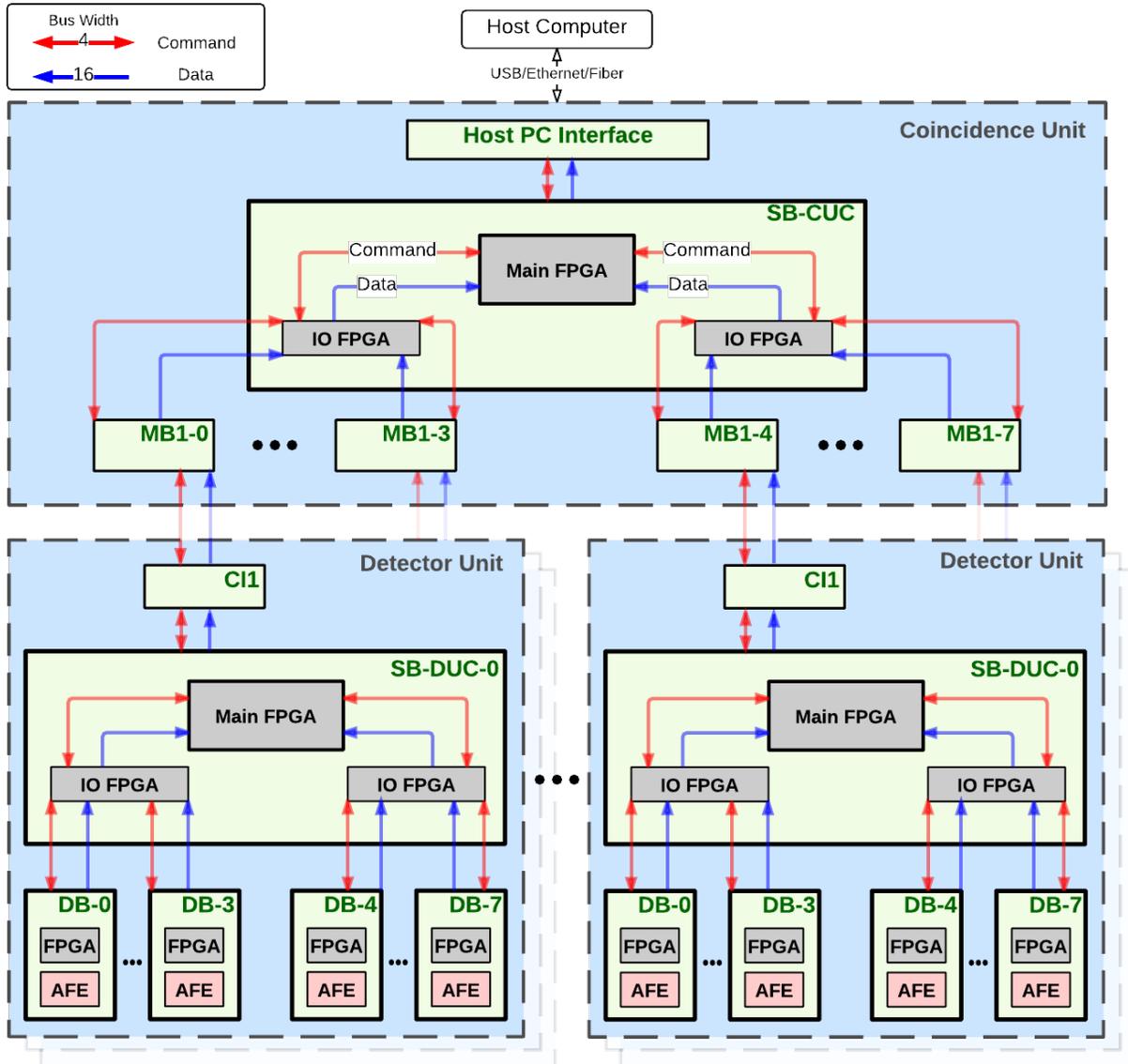


Fig. 2.6: Standard system hardware structure.

up to 16 individual singles events in a single time frame and passing up to four of them to the master FPGA. Clearly some loss results, and the multiplexing algorithm is designed to ensure that this loss is unbiased. Each IO FPGA also serves as a fan-in / fan-out for communication between the support microprocessor and the individual Detector Boards, and the two IO FPGAs can communicate with each other. The digital signals between each IO FPGA and the Main FPGA are identical to the bus IO signals between the IO FPGA and a single Detector Board (see *Timing Signals* (page 15)).

System Firmware and Software Structure

(Software: expand on User Guide section 1.4 and Framework section 1.3.2?? Also see details in Framework section 4.

Firmware: expand on User Guide section 1.4 and Framework section 1.3.2?? Also see details in Framework section 5.)

The OpenPET firmware and software structures are based on a computer network tree topology. The configuration strategy needs to fulfill the following two basic requirements:

1. Compatibility with different types of detector modules (e.g., single analog channel addressing, single crystal addressing for a conventional block detector, etc.);
2. Compatibility with different sized systems (e.g., Small, Standard and Large Systems).

In addition, the configuration strategy needs to be implementable, flexible and reliable.

The firmware and software structure is shown in Fig. 2.7, corresponding to the hardware structure shown in Fig. 2.6. As described in the previous section, All FPGAs (two on supportboard and one on detectorboard) implement software (using NIOS II soft core) and firmware using HDL. The Support Board loaded with coincidence firmware and software is called the Coincidence Unit Controller (CUC). The Support Board loaded with detection firmware and software is called the Detector Unit Controller (DUC).

The computer network tree topology structure is such that the Host PC is the top level and the Detector Boards are the bottom level. The command flow follows a specific path from the source to its destination by writing a specific source and destination address in the system command line. The current OpenPET system uses a 80-bit command word (can be extended) that includes the 16-bit command ID, the 16-bit source and the 16-bit destination addresses, and the 32-bit payload. For a detailed discussion, see *System Commands and Responses* (page 39).

The source address is a 16-bit number that defines where the command originates. Typically, commands that originate at the Host PC have a source address of 0×4000 . The destination address is a 16-bit number that identifies where the response should be received and processed. Both the source and destination addresses have the same address format, as shown in Fig. 2.8.

Starting from least significant bit (LSB):

(2:0)	Detector Board Address
(5:3)	Detector Unit Address
(8:6)	Multiplexer Board Address
(9)	Detector Unit Controller source/destination flag
(10)	Coincidence Unit Controller source/destination flag
(11)	Coincidence Detector Unit Controller source/destination flag
(12)	Multiplexer Board Controller source/destination flag
(13)	Not used
(14)	Host PC source/destination flag
(15)	Broadcast flag

As an example, if the Host PC sends a command, the source address Host PC flag bit will be set to 1. The Host PC can talk to any of the components lower on the network by specifying the destination address. If the Host PC needs to send a command to the SB-CUC, the CUC flag in the destination address is set to one. However, if the Host PC wants to communicate with a specific detector board, the full address including the multiplexer board, detector unit

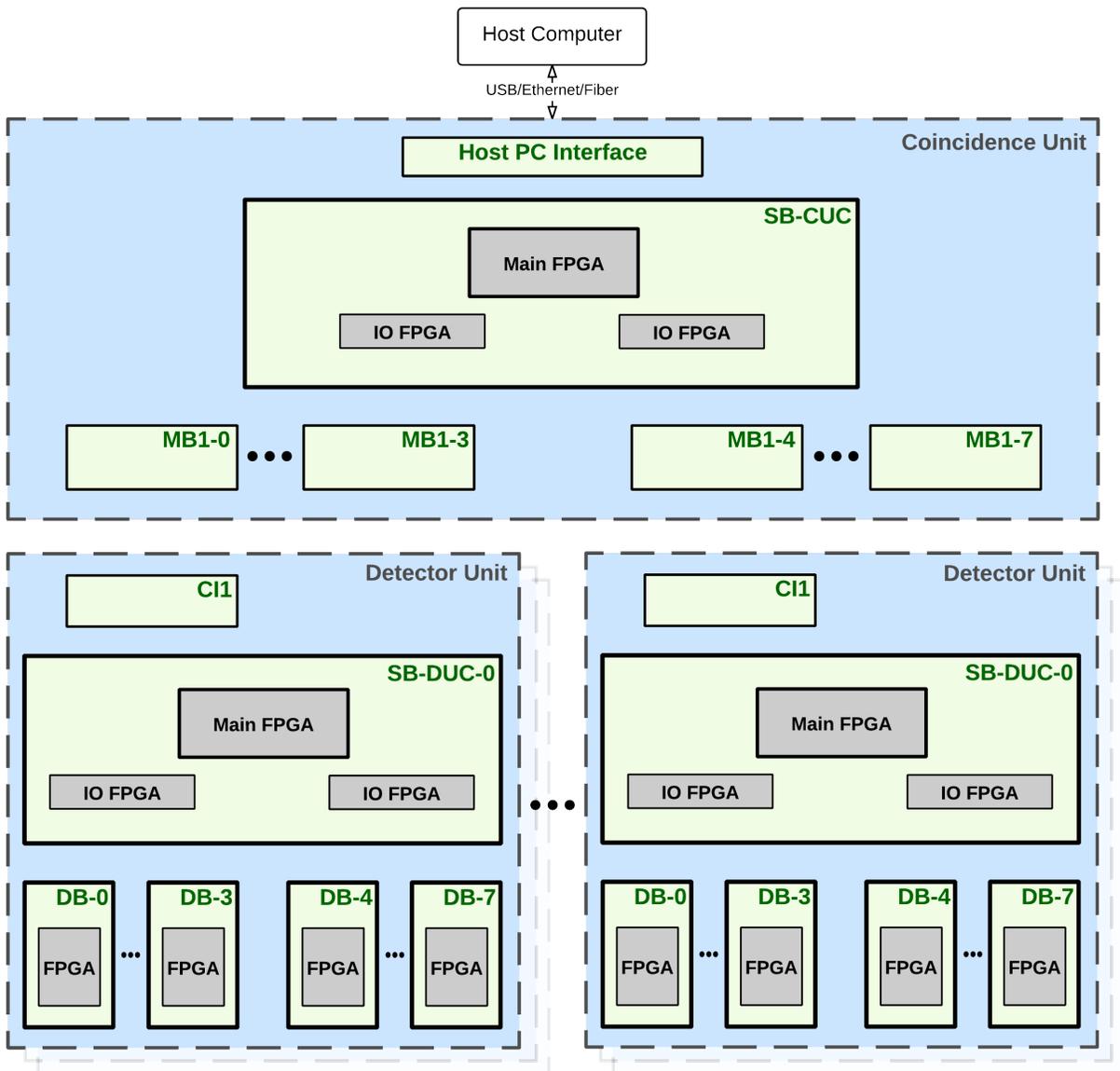


Fig. 2.7: Standard system firmware and software structure.

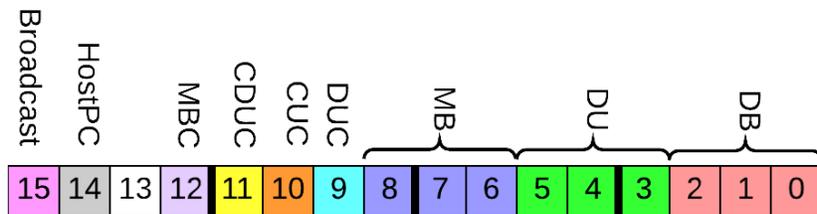


Fig. 2.8: Source/Destination address (16-bits)

and detector board address will be set so that the correct detector board will receive the command. The Host PC is the only node that can initiate commands. The components in the network tree below the Host PC can only reply.

If the Broadcast flag is set in the destination address, the source node will pass the command down to all of its "children" and the child specified in the destination address will be used to create the response.

If the CUC or CDUC flags are set in the destination address, that corresponding unit will execute the command and respond. Since there is no specific address required to send a command to the CUC or CDUC, any address listed is ignored. If the DUC flag is set, only the multiplexer board address is taken into account since a specific DUC corresponds to a specific multiplexer board.

Fig. 2.9 illustrates the structure of the firmware that exists on all FPGAs in the system.

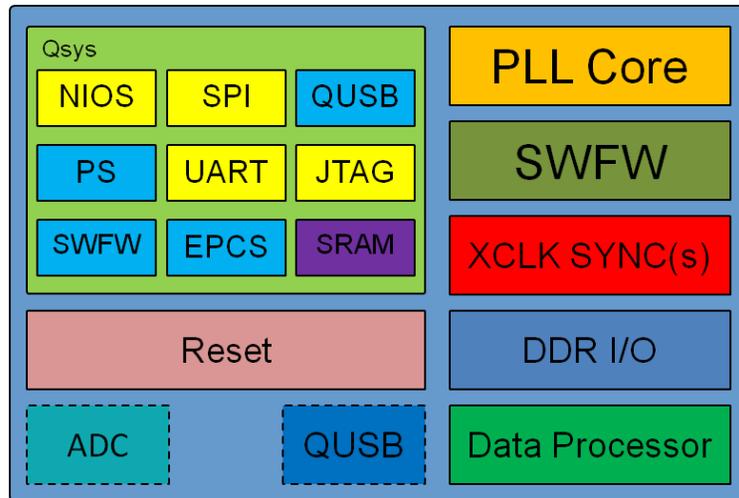


Fig. 2.9: Block diagram of firmware

The blocks with dotted lines denote aspects located in only specific components of the system. The ADC module exists only on the Detector Board FPGAs, and the QUSB module is only for the top node, for example CDUC. The Qsys subsystem component is provided by Altera and it currently uses the Avalon interface to communicate between its component. A customized SRAM module acts as an Avalon Slave is developed for OpenPET system in house due to the lack of compatible modules provided by Altera. The four blue blocks represent parallel I/O modules. The Reset block is a custom HDL logic used on all FPGAs. The data processor component contains HDL logic for the system mode such as singles mode or scope mode. DDR I/O block is a custom HDL logic which instantiates Altera's DDR megafunction to allow us to use double data rate transmission on the backplane of the SupportBoard. XCLK Sync(s) are custom HDL cores used to transfer clocks from one domain to the other. SWFW is a custom core used to allow the software running on Nios II to communicate with the firmware fabric. Finally, PLL core is an Altera megafunction with the same configuration settings on all FPGAs.

As far as compatibility with different sized systems, there are two other size options that deviate from the Standard System size. The first is a Small System which consists of a single detector unit with up to eight detector boards. The Support Board on the detector unit is loaded with both coincidence and detection firmware and software and thereby called a Coincidence/Detector Unit Controller (CDUC). The other system size is the Large System which is currently being developed.

Include DB->SB thru 96 pin connector (fig 35, pg 44 User Guide)

Quick USB: back of SB or Host PC Interface Board

Introduction

Need intro?

(Describe how the various elements interface with each other. Use simple diagrams from Framework, source code and hardware schematics. Example: Revisit Framework figure 5, pg 12 with specification details of clock frequency, protocol lines, etc.)

Timing Signals

Introduction

The system level timing signals are shown in [Fig. 3.1](#). There are two timing signals—the System Clock that is an 80 MHz clock signal, and the Time Slice Boundary that defines the beginning of a Time Slice. The firmware will support both “short” and “long” event words. In “short” mode, the Time Slice Boundary is generated every eight System Clock cycles (100 ns). In “long” mode, it is generated every sixteen System Clock cycles (200 ns). The choice creates a tradeoff—in “short” mode the dead time is a factor of two shorter, but the number of bits per event word is also a factor of two smaller (32-bit verses 64-bit).

The general concept is that the system divides time into small, fixed length Time Slices. During one Time Slice, each of the boards in a Standard System that output singles event words (i.e., DB, CI-1, and MB-1) can pass four singles event words. Thus, the maximum singles rate seen at the CI-1 output of each Detector Unit is 4 singles event words per Time Slice, or approximately 40 million “short” singles event words per second. Thus, 32 singles event words (four for each of the eight CI-1) enter the Coincidence Unit per Time Slice, or approximately 320 million “short” singles event words per second. In a Standard System, there are 8 Detector Units with 28 possible Detector Unit - Detector Unit combinations. So theoretically the Coincidence Unit can identify 448 coincident events per Time Slice (16 for

each of the 28 Detector Unit-Detector Unit combinations), corresponding to 4.48 billion coincidence event words per second. In practice, the maximum event rate is limited by the transfer rate between the Coincidence Unit and the Host PC, which is considerably slower.

(Make sure rest of document is consistent with the above paragraph)

Time Slice Boundary

The rising edge of the Time Slice Boundary defines the beginning of a Time Slice. The width of the pulse is one System Clock cycle, and the period is eight System Clock cycles. In general, it is generated on the Support Board in the Coincidence Unit (although it can be generated on the Support Board in a Detector Unit such as in the Small System), and then buffered through the rest of the system. Propagation delays will introduce skewing; therefore each FPGA that outputs data also outputs a copy of the Time Slice Boundary that is synchronized with its output data signals. The rising edge of the Time Slice is also offset by 3.125ns so that it happens before the rising edge of the Clock cycle. This way, the Time Slice is not missed.

Note: There is a 3.125ns difference between clock and slice rising edges. Slice precedes clock. As can be in Fig. 3.1.

Time Slice

As previously mentioned, the system divides time into small, fixed length Time Slices (100-200 ns or 8-16 clocks). All individual data processing operations must occur within a single Time Slice, which implies that only singles event words that occur in the same Time Slice can be combined to form a coincident event. While it can take significantly longer than one Time Slice to fully process a single event, the system is pipelined so that the processing is divided into smaller operations that each can be completed in a single Time Slice. It takes one Time Slice to transfer a singles event word.

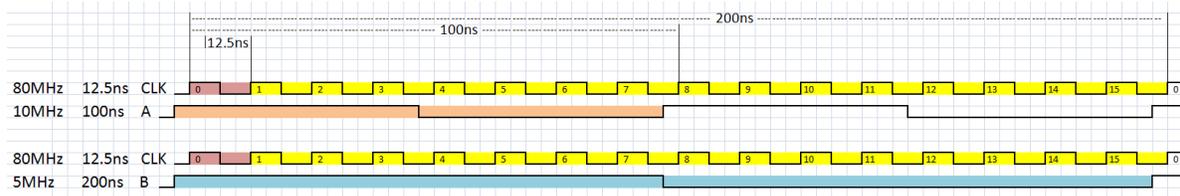


Fig. 3.1: System level timing signals. Slices A and B are 3.125ns early.

Currently, the time slice is set when the firmware is compiled. There is a variable `g_slice_div` that is either 0 or 1 which corresponds to a division by 8 or a division by 16. The former is for the 100ns time slice, and the latter is for the 200ns time slice as can be seen in Fig. 3.1. In the future, the time slice will be runtime user configurable so that the firmware does not need to be compiled every time the user wants to change the time slice.

Clocking

The clock signal flow of the OpenPET system in general is illustrated with Fig. 3.2. There are four clock domains that are explained in more detail below.

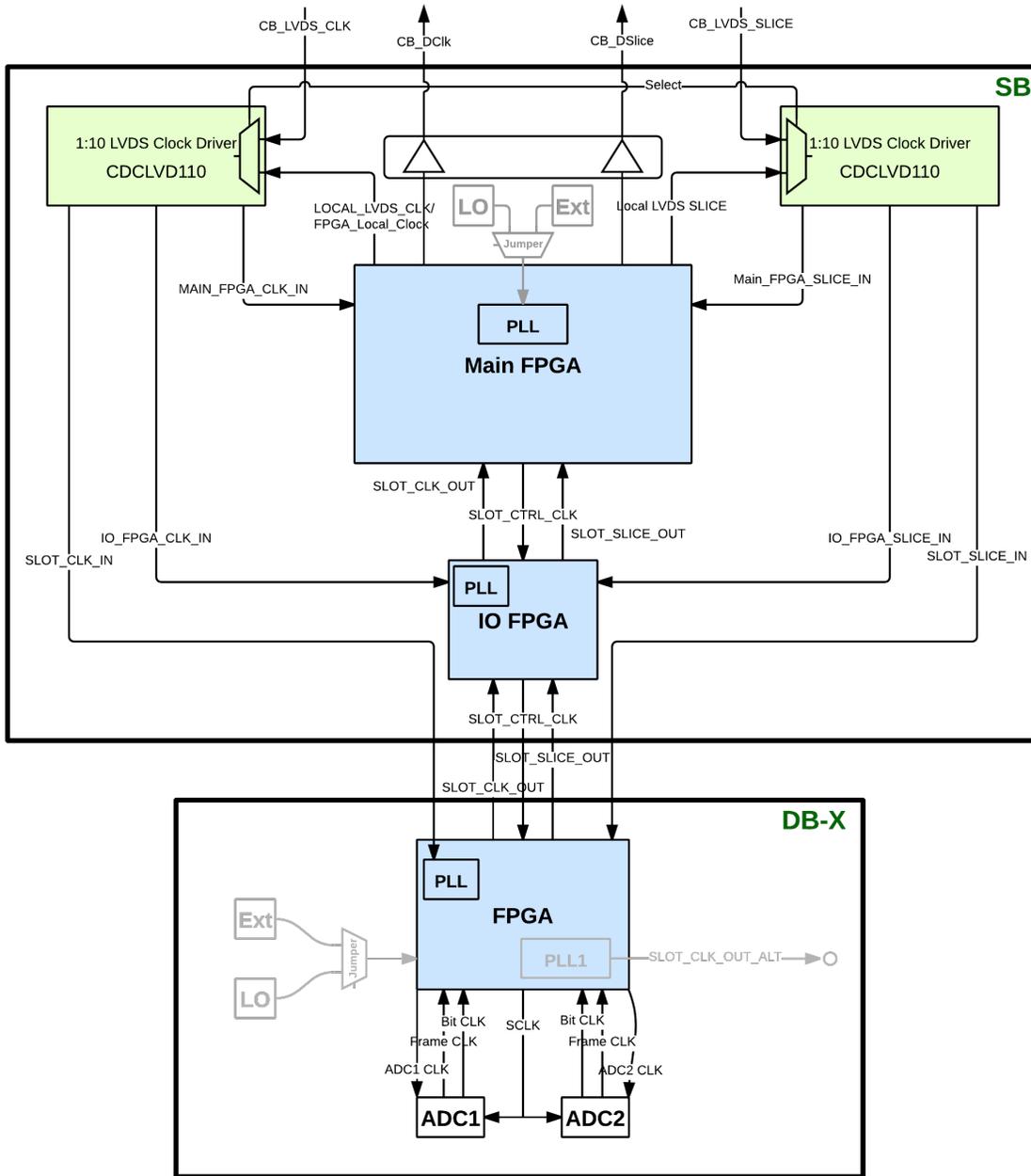


Fig. 3.2: Clock distribution diagram.

System Clock

The System Clock is an 80MHz clock. In general, it is generated by a local oscillator (crystal) or external signal on the Support Board in the Coincidence Unit (although it can be generated on the Support Board in a Detector Unit such as in the Small System), and then buffered through the rest of the system. Propagation delays will introduce skewing, therefore each FPGA that outputs data also outputs a copy of the System Clock that is synchronized with its output data signals. In general, each board in the system regenerates the clock using a phase-locked loop (PLL) in order to maintain signal quality and to minimize phase drift. The System Clock signal also goes to the NIOS II processor and other components connected to NIOS II.

ADC Clock

The ADC Clock is a 40MHz clock. This signal is derived from the 80MHz System Clock signal. An output from the Coincidence Unit (or Detector Unit) PLL goes to a clock distributor that takes the 80MHz signal and creates ten duplicate signals. Eight of these signals go to the Detector Boards, one signal per board. Each Detector Board takes the 80MHz signal and passes it through a PLL on its Main FPGA in order to maintain signal quality and to minimize phase drift. One of the PLL outputs is 40MHz and this is the ADC clock. It is used by the ADC on the board, and for signal processing of incoming data. It is also used for backplane transfer.

QuickUSB Clock

The QuickUSB clock is a 30MHz clock. It is essentially derived from the System Clock signal. It is an output of the PLL on the Main FPGA of the Support Board in the Coincidence Unit (or Detector Unit for a Small System). QuickUSB is used to send and receive data.

Ethernet Clock

The Ethernet Clock is not in use yet, but will be a 125MHz clock. An 80MHz signal from the PLL on the Main FPGA of the Support Board in the Coincidence Unit (or Detector Unit for a Small System) is sent to an additional PLL on the same Main FPGA. One of the outputs is a 125MHz signal that is sent to the PHY transceiver. Ethernet is used to send and receive data.

Small System Example

To describe the clock signals in more detail, we will use a small system example. In a small system, the support board on the CDUC connects to all the detector boards (Fig. 3.3).

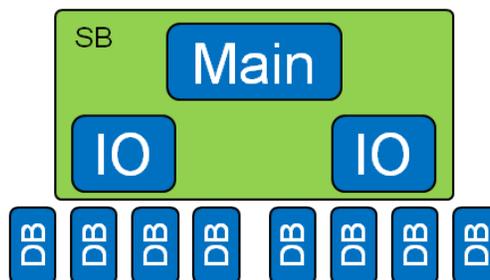


Fig. 3.3: Small system block diagram.

The clock signal schematic is shown in Fig. 3.2. For the purpose of this explanation and example, the support board is on the CDUC and is connected to eight identical detector boards labeled DB-X in the figure. Since the CDUC is

the top node of the system, any incoming and outgoing clock and slice signals at the top of the support board can be ignored.

On the support board, the clock signal that feeds into the Main FPGA PLL can come from two different places. Either a local oscillator (crystal) at 80MHz or an external clock connected through an SMA cable. The user defines which is used based on jumper placement. For the small system, the clock defaults to the local oscillator at 80MHz. This PLL's outputs feed the rest of the system. One 80MHz output travels to the NIOS processor and the U30 clock distributor (left side). The time slices are also generated with this PLL, both the 5MHz (200ns) and the 10MHz (100ns). The time slice signal goes to the U29 clock distributor (right side). Each time slice has its own PLL output. There is additionally a 30MHz output that goes to the QuickUSB. (There is a new Ethernet option which takes a 80MHz signal from the NIOS line and feeds another PLL. This PLL then outputs a 125MHz signal to a Marvell Ethernet PHY chip.)

For the clock signal (left side of Fig. 3.2), the U30 clock distributor takes the 80MHz input from the PLL and duplicates it ten times. There is a voltage source connected in order to accomplish this. One output is a feedback to the PLL, one splits and goes to both IOs, and each of the other eight go to a detector board.

For the time slice signal (right side of Fig. 3.2), the time slice signal (5MHz or 10MHz) travels from the PLL to the U29 clock distributor. This clock distributor also takes the input and duplicates it ten times. One output is a feedback to the PLL, one splits and goes to both IOs, and each of the other eight go to a detector board.

On the detector board, there are clock and slice inputs and outputs. The clock coming in is a 80MHz going into the PLL. The PLL has one output at 80MHz going to the NIOS. Another output goes to the ADC clocks at 40MHz which then go to ADC1 and ADC2.

From the Main FPGA, the SPI sets a control clock to the IO FPGA. This clock is 1MHz from the NIOS processor.

Power

Crate

A Support Crate (Fig. 3.4) consists of a standard 12-slot VME chassis that accommodates 6U x 160 mm boards, a custom Support Board backplane, power supplies, cooling fans, and appropriate boards plugged into slots 9-11 (Figure 2). Some minor assembly is required for the Support Crate; see the OpenPET User Guide Section 4.2 for detailed assembly instructions.

Elma Electronic Inc. supplies the OpenPET crate chassis installed with power supplies and cooling fans. It is an Elma type 12, 19" rack-mount enclosure with a 7U height. It includes a 740W Vicor power supply with +5 V at 40 A, -5 V at 40 A, +3.3 V at 80 A, +12 V at 6A, and ground outputs. The complete OpenPET chassis specifications are available on at <http://openpet.lbl.gov/downloads/hardware/>.

Support Board

Introduction

In an OpenPET Support Crate, the custom Support Board is mounted as a backplane on a standard 12-slot VME chassis that accommodates 6U boards. This Support Board primarily plays two roles, depending on the firmware. In a Detector Unit for a standard- or large-sized system, the Support Board is loaded with detection firmware and acts as a Detector Unit Controller (SB-DUC in Figure 6). In a Coincidence Unit for a standard- or large-sized system, the Support Board is loaded with coincidence firmware and acts as a Coincidence Unit Controller (SB-CUC in Figure 6).

The Support Board also plays a third role for the special case of a small system when it is configured as a Coincidence Detector Unit Controller (CDUC), which interfaces with the detector boards and performs coincidence functions. Basically the CDUC performs the functions of both the CUC and DUC.



Fig. 3.4: The assembled OpenPET Support Crate viewed from the (a) front and (b) back.

Repository

There are five directories in the [SupportBoard](#) repository that are outlined below.

doc

This directory contains the documentation for hardware, firmware, and embedded software.

fw

This directory contains the Altera firmware projects for the 3 FPGAs on the support board. In order to build the firmware as well as the embedded software that runs on the NIOS II CPU, the user should follow the simple commands below:

Building the firmware and flash images for the Support Board:

Windows x86 and x64 OSs:

1. Open your file browser i.e. Explorer and go to `./scripts` directory
2. Double click on 'buildFirmware.bat' to launch firmware building flow.
3. By default CDUC firmware is built. If you like to build another firmware edit 'buildFirmware.bat' and change CDUC to CUC or whatever you like.
4. If you don't want to edit the bat file. Open a Command Prompt and type `alt_win.cmd build-Firmware.sh CUC`
5. Follow the prompts.

Unix x86 and x64 POSIX OSs:

1. Open a terminal and navigate to `./scripts`
2. execute `buildFirmware.sh` by typing `./buildFirmware.sh`
3. By default CDUC firmware is built. If you like to build a firmware for another controller type `./buildFirmware.sh CUC`
4. Follow the prompts.

Note: It is possible to program the flash on the Main FPGA using the `flashboard` script. All the following compiled images will be stored in `./scripts/bin`

1. IO 1-4 SOF image
2. IO 5-8 SOF image
3. Main FPGA SOF image
4. `main.hex` is the ELF program that runs on the NIOS II CPU in the Main FPGA
5. `supportboard.jic` is the flash image which contains the four files listed above

To flash on Windows just double click on `flashboard.bat`. If you need to change the cable then edit the bat file and change the cable index number.

On Unix, just run `./flashboard.sh` or `./flashboard.sh 2` where 2 is the USB blaster cable index.

hw

This directory contains the PCB schematic and layout files using Orcad tools. It also has the bill of materials (BOM) for your convenience.

scripts

sw

This directory contains the embedded software which runs on the soft CPU core (NIOS II) in the Main FPGA. In order to build the ELF file associated with this project, one has to build the Qsys and Quartus projects in the firmware directory first. Follow the directions listed under “fw” section to build the embedded software using buildFirmware script.

Support Board with Detection Firmware

In an OpenPET Support Crate, the custom Support Board is mounted as a backplane on a standard 12-slot VME chassis that accommodates 6U boards. This Support Board primarily plays two roles, depending on the firmware. In a Detector Unit for a standard- or large-sized system, the Support Board is loaded with detection firmware and acts as a Detector Unit Controller (SB-DUC in Figure 6). In a Coincidence Unit for a standard- or large-sized system, the Support Board is loaded with coincidence firmware and acts as a Coincidence Unit Controller (SB-CUC in Figure 6).

The Support Board also plays a third role for the special case of a small system when it is configured as a Coincidence Detector Unit Controller (CDUC), which interfaces with the detector boards and performs coincidence functions. Basically the CDUC performs the functions of both the CUC and DUC.

In summary, digital IO is performed on the Support Board in a Detector Unit or Coincidence Unit with the following specifications (Fig. 3.5):

Through Main FPGA:

- 16 bits digital IO (4 bits direction control)
- 10 Bits LED bar
- Two logic analyzer connectors (16+1 bits each)

Through IO FPGA 1:

- 16 bits digital IO (4 bits direction control)
- 10 Bits LED bar
- One logic analyzer connector (16+1 bits each)

Through IO FPGA 2:

- 16 bits digital IO (4 bits direction control)
- 10 Bits LED bar
- One logic analyzer connector (16+1 bits each)

IO FPGAs

There are two slave IO FPGAs on the Support Board. These act primarily as a multiplexer for singles events, each taking up to 16 individual singles events that it can receive in a single time frame and passing up to 4 of them to the Main FPGA. Obviously, there is some possibility for data loss, and the multiplexing algorithm is designed to ensure

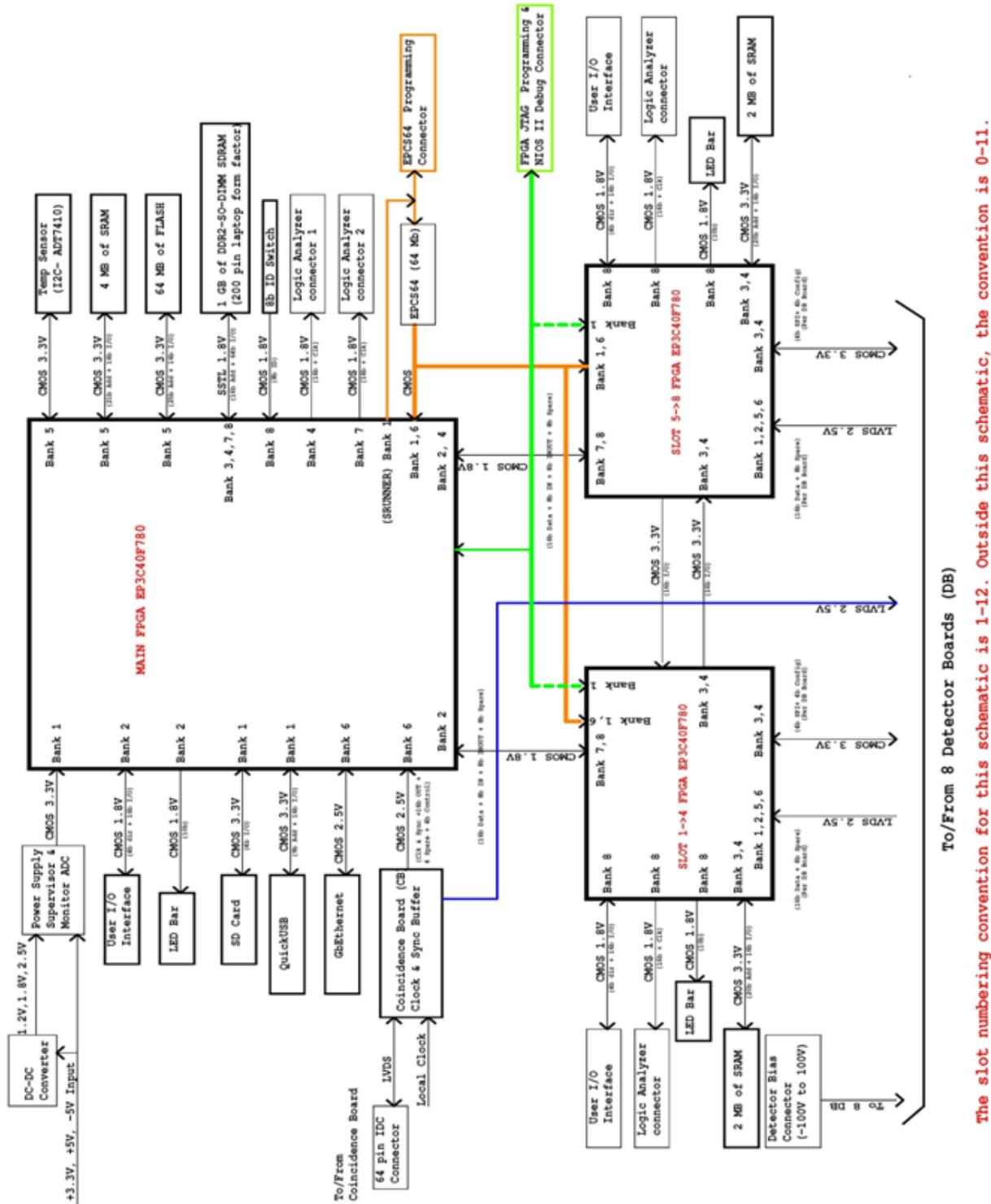


Fig. 3.5: Support Board overview schematic diagram (Page 1 Title Block Diagram).

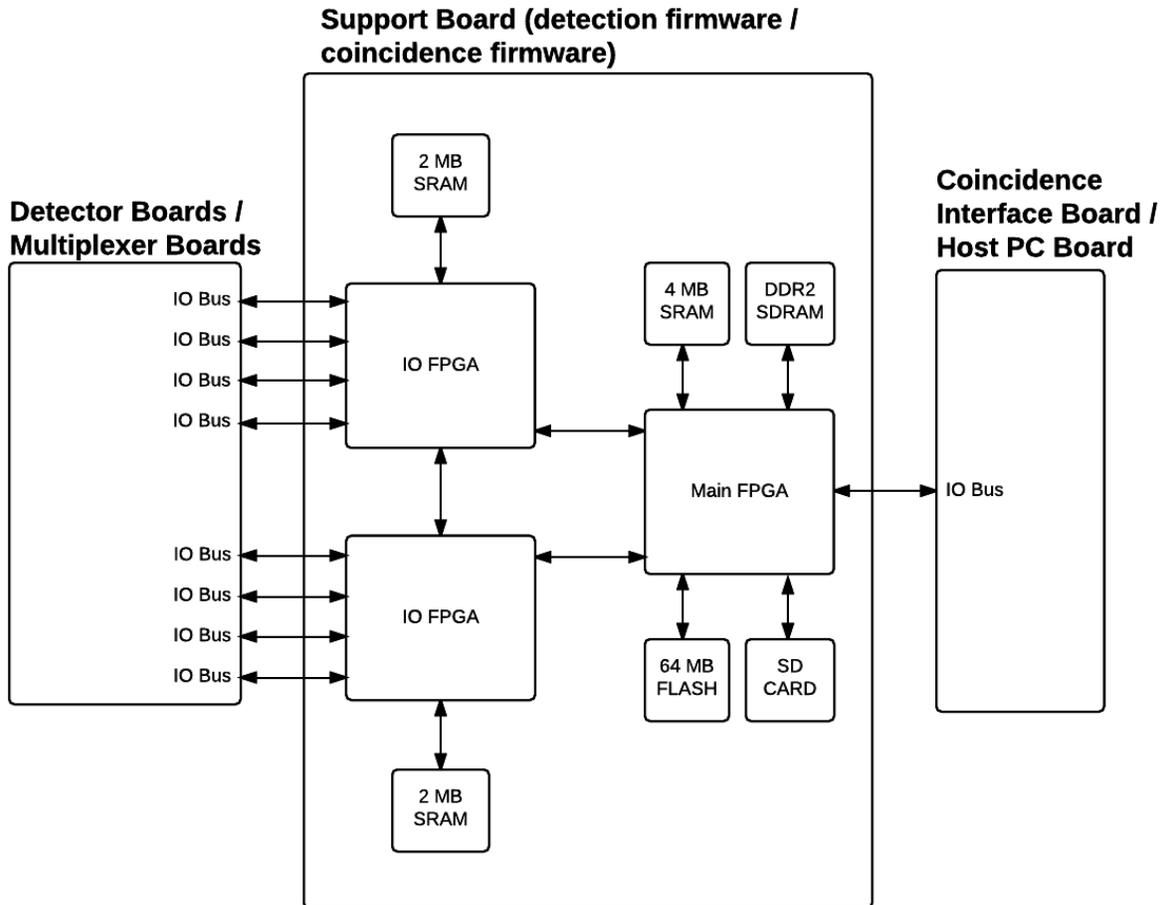


Fig. 3.6: Schematic of the Support Board, which is loaded with either detection firmware (when used in a Detector Unit) or coincidence firmware (when used in a Coincident Unit).

that this loss is unbiased. Each IO FPGA also serves as a fan-in and fan-out for communication between the Main FPGA and the individual Detector Boards, and the two IO FPGAs can communicate with each other.

The digital signals between each IO FPGA and the Main FPGA are identical to the standard OpenPET Bus IO signals (Fig. 3.7): lines for the clock and Time Slice Boundary (both directions), 4 control lines, 4 FPGA programming lines, 16 event data lines, and eight user-definable data lines. There are also 32 user-definable digital lines between the two IO FPGAs, 16 sending data in each direction.

Main FPGA

A single physical Main FPGA performs the logical functions of both the master FPGA and the support microprocessor. Its FPGA-like functions are mostly limited to passing events from the IO FPGAs to the Coincidence Interface Board (providing multiplexing, if necessary).

Some of the logic blocks in the Main FPGA can be programmed using Nios II to be identical to microprocessor hardware, which then runs executable files programmed in C. This support microprocessor receives high-level commands from the Host PC via USB (or Gigabit Ethernet or Fiber-Optics), and then interprets and executes these commands. It is responsible for loading all the programs into the IO FPGAs and DB FPGA, as well as the contents of all the support memory, detector memory, and all other registers that are on the DB and SB. It also monitors the event stream and can insert diagnostic information (such as event rates) into the event stream or provide this information directly to the Host PC. Whenever possible, calibration routines are also performed on the support

Support Memory

There are multiple forms of memory on the Support Board, as shown in Fig. 3.5. The SRAM is accessed by the FPGA and provides its output within 1 clock cycle of being addressed so it is both reasonably fast and deterministic, which greatly simplifies incorporating it within FPGA algorithms. However, the SRAM capacity is fairly small. Thus, the Main FPGA is connected to 4 MB of SRAM and each of the IO FPGAs is connected to 2 MB of SRAM, for a total of 8 MB of SRAM on each Support Board. This memory is typically used to store look-up tables that apply real time calibration and corrections to the event data.

As the Main FPGA also emulates a microprocessor, RAM memory and disk storage are also necessary for it to function effectively. The RAM memory is provided via up to 1 GB of RAM that can be plugged into a DDR2 SDRAM connector (identical to that typically found in laptop computers). The disk storage is provided by a SD card (identical to that found in digital cameras) that is plugged into a SD card slot, when users desire disk storage that can be easily removed. Otherwise, standard disk storage is provided by a 64 MB FLASH memory chip that is connected to the Main FPG, in order to store the FPGA firmware needed for DB FPGAs, contents of all the support SRAM memory, detector memory, and all other registers that are on the DB and SB. This information can also be stored in the on-board EPCS memory, which is where it resides in the initial release.

In summary, the Support Board has the following memory available, as shown in Fig. 3.5:

Through Main FPGA:

- Flash memory
 - EPCS64 flash memory
 - 64 MB flash memory
- RAM
 - 4 MB SRAM
 - 1 GB DDR2-SO-DIMM SDRAM

Through IO FPGA 1:

- RAM

- 2 MB SRAM

Through IO FPGA 2:

- RAM
- 2 MB SRAM

Clock Conditioning

The clock-conditioning block consists of a PLL (phase-locked loop) that regenerates the system clock signal from the Support Board in a Coincidence Unit and passes it to the Support Board FPGAs in a Detector Unit and then to the Detector Boards. The block also includes space for a local clock oscillator (i.e., LO in Fig. 3.2), which is used to provide the system clock when the system is being used without a Coincidence Unit (i.e., when the support microprocessor passes events directly to the host computer).

Connectors

The connector that is soldered onto the SB in order to connect to a single DB (slot 0-7) is a 96 pin female connector, press fit for a 0.125" thick board, Vector Electronics RE96FSP (Digi-Key part number V1240-ND). This connector is also used for slots 8-10, and two of these connectors are used for slot 11. See below for more details.

Miscellaneous Devices

The Support Board also contains signals for miscellaneous devices, including temperature sensors, two RS232 interfaces, and power monitor.

(Add description of Slots 0-11, like the User Guide? Or describe elsewhere?)

Support Board with Coincidence Firmware

In a Coincidence Unit, the Support Board acts as a Coincidence Unit Controller (SB-CUC) when it is loaded with coincidence firmware, as shown in Figure 6. It also provides control and power for the Multiplexer Boards that are located in slots 0-7.

In the general data flow, singles event words are passed through a Coincidence Interface Board to a Multiplexer Board, which can provide a further layer of multiplexing for singles event words, if necessary. These multiplexed singles event words are then passed to the Support Board with coincidence firmware, which searches through the singles event words from multiple (up to eight) MBs for pairs that are in time coincidence and then forms coincidence event words. These coincidence event words are then passed to the Host PC through the Host PC Interface Board. Optionally, the Coincidence Unit Controller can act as a multiplexer and forward unaltered singles event words to the Host PC.

When the Support Board acts as a Coincidence Unit Controller, the role of the three FPGAs is similar to that described above. Each IO FPGA acts as a multiplexer for singles event words, taking up to 16 individual singles events from MBs and passing up to four of them to the Main FPGA, using a multiplexing algorithm that ensures unbiased loss. The Main FPGA acts as both a master FPGA and a support microprocessor. As a master FPGA, it primarily passes events from the IO FPGA to the Host PC Interface Board. As a support microprocessor, Nios II is used to program logical blocks in the FPGA to run executable files programmed in C. For instance, the Main FPGA is used to identify and form coincident event words.

High-level commands are sent via USB (or Ethernet or Fiber-Optic) from the Host PC to the Coincidence Unit. The NIOS II microprocessor is not connected directly to these communication interfaces. Instead, the NIOS II microprocessor talks to the communication interfaces through the register array implemented in the Main FPGAs.

Support Board with Detection & Coincidence Firmware

In the special case of a Small System, the Support Board is also capable of identifying coincident pairs of singles event words, formatting them into coincidence event format, and passing them to the support microprocessor (Main FPGA), which then passes them to the Host PC. Thus it can act as a full-featured PET data acquisition system, albeit with a limited number of input channels and output event rate capability. It can also be programmed to multiplex singles events and pass them to the Host PC.

Detector Board

Introduction

The overall purpose of the Detector Board is to process the analog inputs from the detector modules in the system and convert them into singles event words. Generally, this requires determining the energy, interaction position, and arrival time associated with a single gamma ray interaction that occurs in the detector module, as well as applying as many corrections as possible before the associated singles event word is generated.

In order to process an analog input signal, the Detector Board contains an analog front-end circuit with amplification and filtering to reduce the noise and bandwidth of the signal. The processed analog signal is subsequently digitized by an analog-to-digital converter (ADC) to capture the processed analog signal. For systems that require good timing resolution such as PET, additional circuitry may be implemented in the front end to split the analog input signal into a fast timing path where the signal is amplified with a high-bandwidth amplifier. The amplified signal is then triggered by a fast leading edge discriminator to create a timing pulse for the signal arrival, which is time stamped by a time-to-digital converter (TDC) implemented inside the FPGA.

The back end of the Detector Board primarily consists of an FPGA and static random access memory (SRAM). The FPGA processes the digitized signals from the ADCs and, if necessary, combines information from multiple channels to compute the deposited energy, the interaction position, and the event time. Appropriate calibration correction factors that are stored in the SRAM can also be applied to the data. The computed information are formatted into a singles event word and then transferred to the Support Board (i.e., SB-DUC) via the standard OpenPET Bus IO, as described in *Bus IO* (page 27). In addition, the firmware for the FPGA is loaded into the Detector Board by the Support Board via the standard Bus IO. Different firmware can be loaded into the DB FPGA to perform tasks other than event processing, such as debugging, testing, and calibration.

Bus IO

The Bus IO connecting the Detector Board to the Support Board (SB-DUC) is shown schematically in Fig. 3.7. The bus is divided into several blocks.

One block provides the timing signals using 4 LVDS differential pairs. There are two fundamental timing signals - an 80 MHz System Clock signal generated on the Support Board and a Time Slice Boundary signal that defines the beginning of a Time Slice. The Support Board sends a copy of these System Clock (CLK) and Time Slice Boundary (SLICE) signals, which are used to clock data from the Support Board to the Detector Board. Because there is propagation delay within the Detector Board, another copy of the System Clock and Time Slice Boundary signals (that is produced by the DB) is used to clock data from the Detector Board to the Support Board. In the OpenPET architecture, the system divides time into small, fixed time slices of either 100 ns (8 clocks) or 200 ns (16 clocks). Alternatively, custom time slices can be designed by modification of the firmware, if needed.

A second block defines 4 LVTTTL single ended lines to control data between the DB and SB. OpenPET uses a custom serial digital bus protocol comprised of a Clock line (CTRL_CLK), Data In line (CTRL_DI), Data Out line (CTRL_DO), and Chip Selection line (CTRL_CS). However, the Chip Selection line is not utilized in the current protocol. Alternatively, these control lines can be configured to support standard digital serial protocols such as I2C or SPI.

A third block in the Bus IO provides 4 LVTTTL single ended lines to program the FPGA on the Detector Board using serial protocols, which are generated by the Support Board. These FPGA programming lines are nCONFIG, DCLK, DATA0, and CONF_DONE.

A fourth block provides 16 LVDS differential pairs (grouped into 4 sets of 4 LVDS differential pairs) to transfer singles event words from the DB to SB. All individual operations must occur within a single time slice, which implies that only singles event words that occur in the same time slice can be combined to form a coincident event. A singles event word can be either a 32-bit word (100 ns time slice) or 64-bit word (200 ns time slice). Since it can take significantly longer than a single time slice to fully process a singles event, the system is pipelined so that the processing is divided into smaller steps that each can be completed in a single time slice. During one time slice, each set of 4 LVDS differential pairs can pass one singles event word (i.e. maximum of 4 singles event words per time slice). Thus, the maximum singles event rate that can be transferred out of each Detector Board is 40 million events per second.

Another block provides 8 spare LVDS differential pairs between the DB and SB for users to pass any information between these two boards.

The final block in the BUS IO supplies power to the Detector Board. Specifically, the Support Board supplies +5 V, +3.3 V, -5 V, and ground.

16-Channel Detector Board

[Fig. 3.8](#) shows the block diagram of the 16-channel Detector Board and [Fig. 3.9](#) shows a photograph. This Detector Board accepts up to 16 analog input signals, each of which is processed independently. The analog input signal is required to be negative polarity, ground referenced. The input analog signal is terminated with 50 ohms, and then split into two processing chains: an energy chain and a timing chain.

The processing circuit for one channel is shown in [Fig. 3.10](#). The input stage accepts voltages between -0.8 V to 0 V. The input voltage can be attenuated to fall within the acceptable range by changing the attenuation resistor values on the energy chain. Only the signal on the energy chain needs to be attenuated because the range of the input voltage is limited by the dynamic range of the ADC. In addition, there are 8 external differential LVDS IO pairs that can be used to interface to the DB FPGA. The flexibility of these digital IO will allow any digital inputs and/or outputs to be applied to the Detector Board.

On the energy chain, the input signal is amplified (OPA2694), and then split into an anti-aliasing filter (LTC6605-7) with a cut-off frequency at 7 MHz and a comparator (MAX964). The filtered signal is sent to a 12-bit ADC (ADS5282) that can digitize the signal at a sampling rate from 10 MSPS to 65 MSPS, with a typical sampling rate of 40 MSPS. Each channel of the ADC also has a programmable digital gain from 0 dB to 12 dB. A trigger is generated by the comparator and sent to the FPGA.

On the timing chain, the input signal is amplified with an x10 high-bandwidth (1.8 GHz) amplifier (THS4303) and then sent to a fast comparator (MAX9602) with low propagation delay dispersion (~30 ps) to trigger on the leading edge of the analog signal. We have selected a leading edge discriminator, since several groups have reported that a leading edge discriminator has equivalent or better performance than a constant fraction discriminator (although sometimes requiring amplitude correction for time walk). The generated timing pulse from the comparator is then sent to the DB FPGA, where a TDC is implemented to determine the time stamp of the arrival time of the timing signal.

Further details can be seen on the 16-channel Detector Board schematic shown in [Fig. 3.11](#) and available on the [OpenPET hardware page](#) (log-in required)

Repository

This describes the directories and build instructions for the [DetectorBoard16ChLBNL](#) repository.

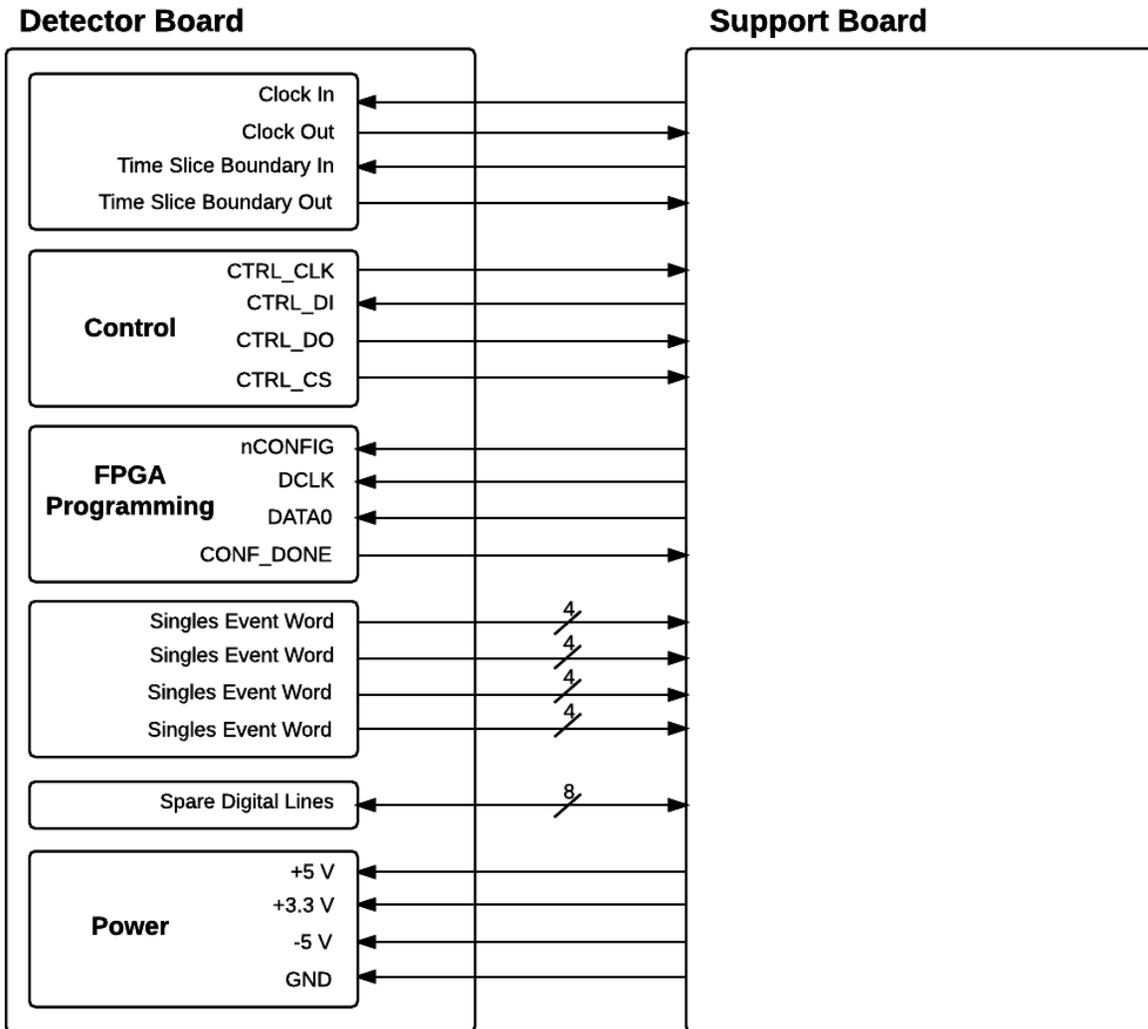


Fig. 3.7: Diagram of the BUS IO.

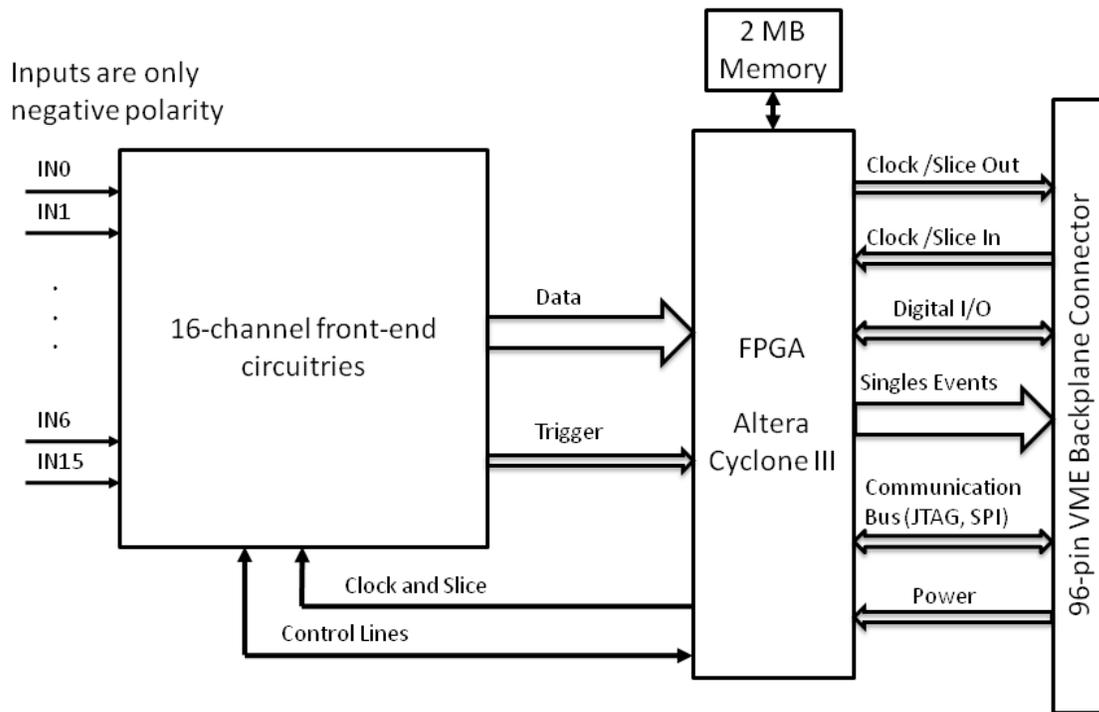


Fig. 3.8: Block diagram of the 16-channel Detector Board. (Make new version. Replace clock, power, etc lines from the FPGA to 96-pin as BUS IO – i.e. already shown in Fig 12.)

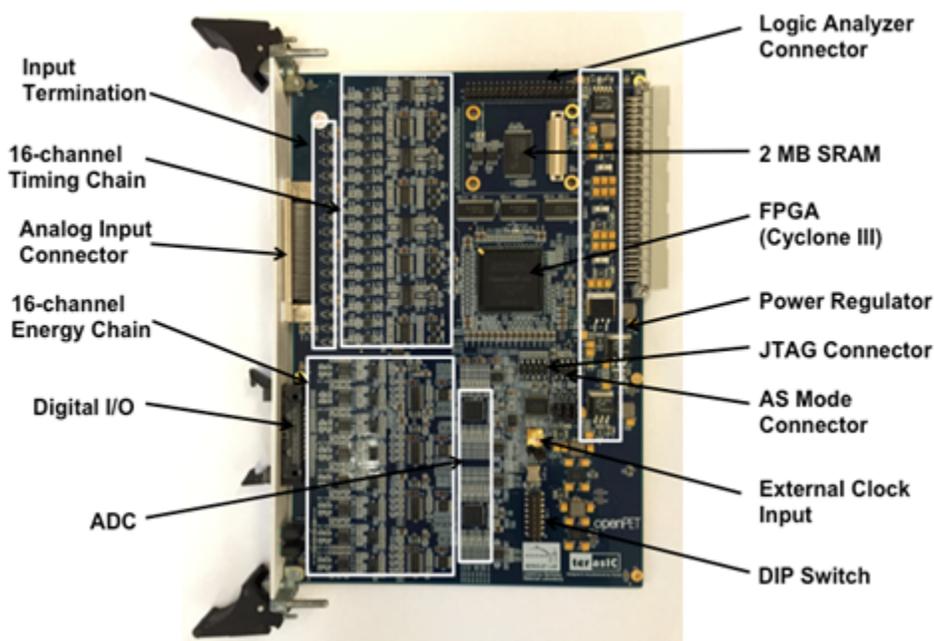


Fig. 3.9: Photograph of the 16-channel Detector Board.

doc

This directory contains the documentation for hardware, firmware, and embedded software.

fw

This directory contains the Altera firmware projects for the FPGA on the detector board. In order to build the firmware and flash images, the user should follow the simple commands below:

Windows x86 and x64 OSs:

1. Open your file browser (i.e., Explorer) and go to `./scripts` directory
2. Double click on 'buildFirmware.bat' to launch firmware building flow.
3. By default DB firmware is built. If you like to build another firmware edit 'build-Firmware.bat' and change DB to MB or whatever you like.
4. If you don't what to edit the bat file. Open a Command Prompt and type `alt_win.cmd buildFirmware.sh MB`
5. Follow the prompts.

Unix x86 and x64 POSIX OSs:

1. Open a terminal and navigate to `./scripts`
2. Execute `buildFirmware.sh` by typing `./buildFirmware.sh`
3. By default DB firmware is built. If you want to build a firmware for another board, type `./buildFirmware.sh DB`

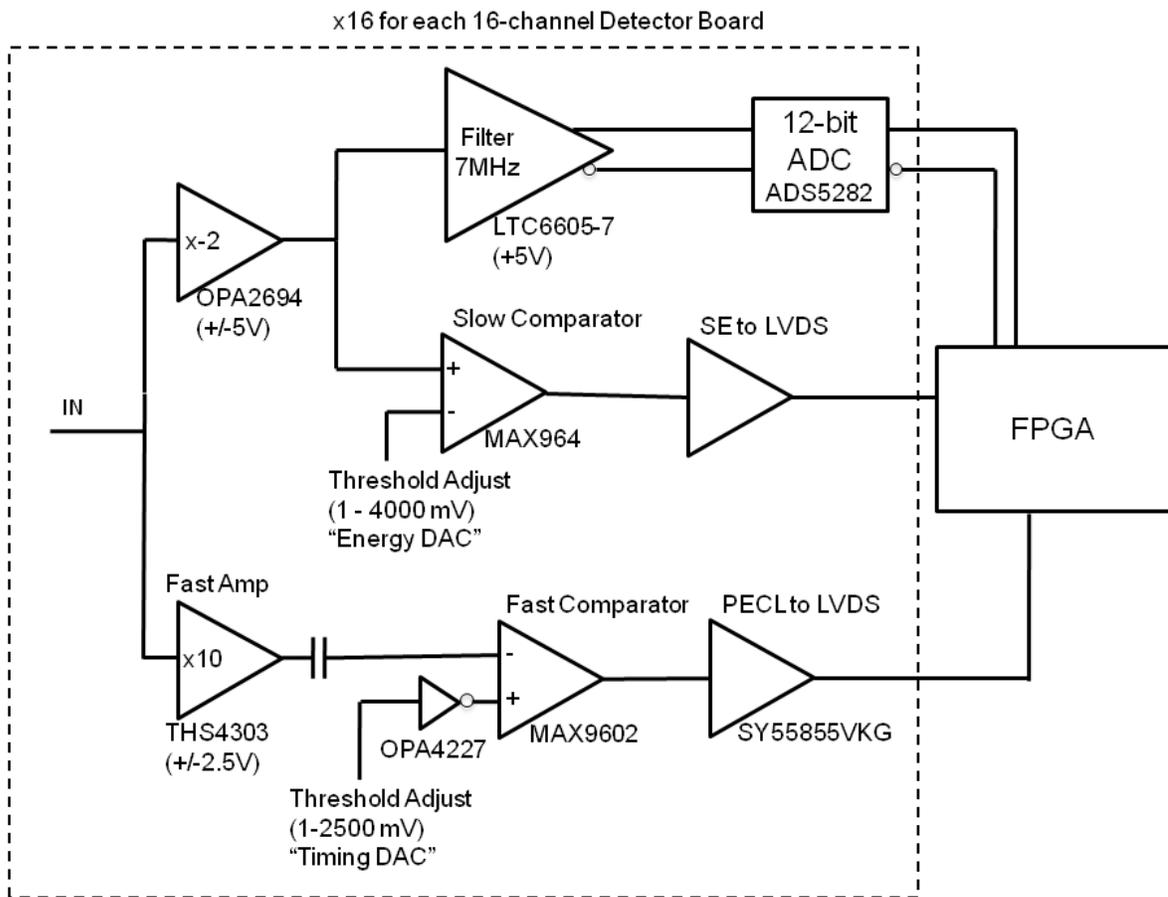


Fig. 3.10: Block diagram of the front-end circuitries of one channel of the 16-channel Detector Board.

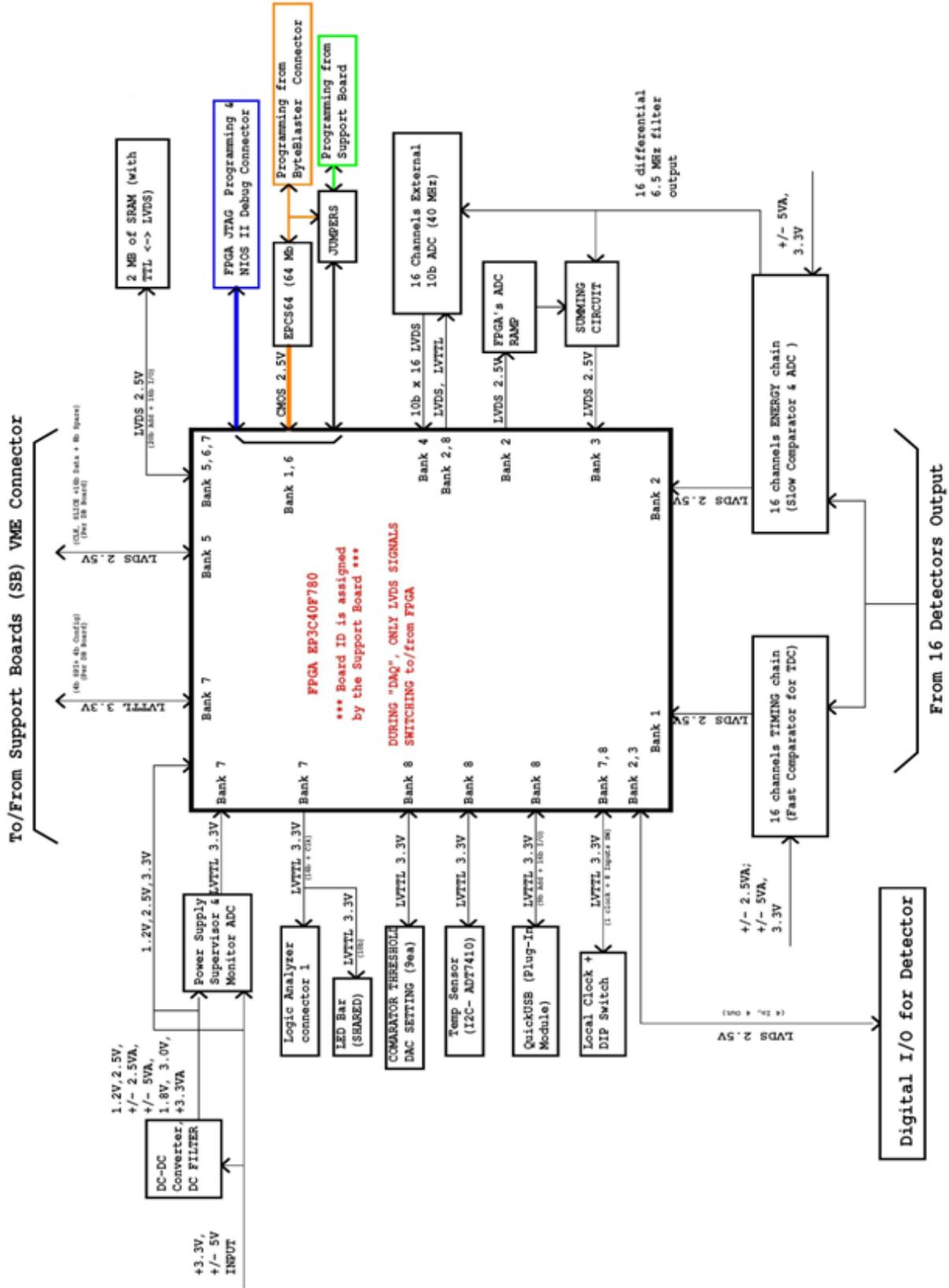


Fig. 3.11: 16-channel Detector Board overview schematic diagram (Title Block Diagram).

4. Follow the prompts.

Note: It is possible to program the flash on the FPGA using the flashBoard script. The user should first program the Support Board's flash and reboot the chassis before programming the detector board.

On Windows, just double click on flashboard.bat. If you need to change the cable, then edit the bat file and change the cable index number.

On Unix, just run `./flashboard.sh` or `./flashboard.sh 2` where 2 is the USB blaster cable index.

hw

This directory contains the PCB schematic and layout files using Orcad tools. It also has the bill of materials (BOM) for your convenience.

sw

This directory contains the embedded software which runs on the soft CPU core (NIOS II). In order to build the ELF file associated with this project, one has to build the Qsys and Quartus projects in the firmware directory first. Follow the directions listed under "fw" section to build the embedded software using buildFirmware script.

Coincident Interface & Multiplexor Boards (CI-1 & MB-1)

(Make figure similar to Fig. 3.5, showing lines connecting CI-1 to MB-1; same boards. Subset of standard Bus IO connects to CI-1 and MB-1 from SB: show or list in text with ref to Fig. 3.5.)

Host PC Interface Board

See [Host PC Interface Board Schematic](#)

Repository

This describes the directories and build instructions for the HostPC repository.

sw

This directory contains software which runs on the Host PC to interface with OpenPET electronics. The following list explains some of the files in this directory.

- `./dist/openpet/openpet.exe`: OpenPET command-line utility for Windows x64.
- `openpet.py`: OpenPET command-line utility source code (cross platform).
- `OpenPETlib.py`: OpenPET python Library (cross platform).
- `QuickUSB.py`: QuickUSB library (Developed by Bitwise).

- **ctrl_scope_example_simple_DEV.py**: A simple example python script to configure and acquire scope data from an OpenPET system.
- **ctrl_scope_example_intermediate_DEV.py**: An intermediate example python script to configure and acquire scope data from an OpenPET system.
- **ctrl_scope_example_advanced_DEV.py**: An advanced example python script to configure and acquire scope data from an OpenPET system.
- **plot.py**: Plots scope data using Matplotlib.
- **tdc.py**: Plots the tdc values for two channels.
- **ctrl_scope_example_simple.cmd**: A simple example Windows Batch script wrapper to configure and acquire scope data using openpet.exe.
- **README.netmap**: Instructions to setup high-speed ethernet acquisition i.e., 992 Mbps

Build Instructions

Windows x64 executable:

If you just want to use the executable you don't need to continue reading. Just run `../dist/openpet/openpet.exe` from your program or wrapper and enjoy the goodness.

Required software to run python scripts (not openpet.exe executable):

- Python 2.7.x (not Python 3.x branch)
- QuickUSB library 2.15.2
- Numpy 1.9+
- Matplotlib 1.4+ (for plotting data)

Installing required software to run python scripts (not openpet.exe executable):

- On Windows:
 - Are you running a 64-bit or a 32-bit Windows? Take a note of that. Typically win32 means 32-bits and amd64 means 64-bit.
 - Go to <https://www.python.org/downloads/windows/> and download the latest 2.7.* branch of Python
 - Configure the Python installer to add python to your PATH environment variables or do it manually by appending `C:Python27;C:Python27Scripts` to your PATH environment variables
 - Download and install latest Microsoft VC++ for Python27 from <http://aka.ms/vcpython27>
 - Launch Windows Command prompt and type the following to update pip to the latest version:

```
pip install pip --upgrade
```
 - Install numpy

```
pip install numpy
```
 - Install matplotlib

```
pip install matplotlib
```
 - Install netifaces

```
pip install netifaces
```
- On UNIX compatible OSs, use your distribution package manager to install Python 2.7, NumPy, Matplotlib, and netifaces

- Python: <https://www.python.org/downloads/>
- NumPy and Matplotlib: <https://www.scipy.org/install.html>

Building openpet.exe executable:

- To build the binary openpet.exe, use pyinstaller: `pip install pyinstaller`
- To create the binary (i.e., exe file) run `pyinstaller --onefile openpet.py`
 - This will generate openpet.exe in .distdirectory
 - If you don't want a single exe, drop the `-onefile` switch

Firmware

Throughput

Type	Rate (Mbps)
Single DB to SB	1280 (40MHz x 32 bits)
USB 2.0 theoretical maximum	480
QuickUSB measured maximum	330
GbE theoretical maximum	1250
GbE expected (no optimization)	500

The support board throttles the detector boards and IO FPGAs to adapt to the output speed.

Support Board

This section will detail the variables that are set in the firmware of the support board. The firmware must be compiled every time these values are changed in order to be put into effect. The following variables are found in main.vhd located in the support board firmware directory in the 'main' folder.

- `g_NODE_TYPE`

Variable that defines the support board function or type. For example, if the support board is in the CDUC, then the variable value is `c_NODE_CDUC`. The last term changes for each different node type, e.g., DUC or CUC.
- `g_slice_div`

Integer that defines the time slice size. It sets the clock to slice relation. If the value is 0 (default), divide by 8, and if it is 1, divide by 16. This correlates to a 100ns or 200ns time slice. Must be the same as the Detector Board value.
- `g_adc_channels`

Positive variable that specifies how many channels are connected to each ADC chip. For the LBNL 16-Channel Detector Boards, there are 8 channels connected to each ADC chip. To change this, a new board must be designed and built.
- `g_adc_number`

Positive variable that defines the number of ADC chips on the detector board. To change this, a new board must be designed and built. For the LBNL 16-Channel Detector Boards, there are 2 ADC chips. The `g_adc_channels x g_adc_number` equals the number of channels of the detector board (e.g., $8 \times 2 = 16$).
- `g_adc_resolution`

Positive variable that is the ADC resolution. This value is determined by the ADC chip used on the board. For the LBNL 16-Channel Detector Board, the ADC chip used is a 12 bit resolution.

- `g_QUSB_FD_WIDTH`

Positive variable that defines the QuickUSB data bus width. For the LBNL 16-Channel Detector Board, the QuickUSB module has a data bus width of 16 bits.

- `g_QUSB_ADR_WIDTH`

Positive variable that defines the QuickUSB address bus width. For the LBNL 16-Channel Detector Board, the QuickUSB module has an address bus width of 9 bits.

- `g_QUSB_CMD_PKTS`

Positive variable that sets the number of QuickUSB packets or write cycles to complete a single OpenPET command. It is currently set to 5 since OpenPET commands are 80 bits long so $16 (g_QUSB_FD_WIDTH) \times 5 (g_QUSB_CMD_PKTS) = 80$. If there is a different command length (still a multiple of 16 due to the QuickUSB data bus width), one can simply change the number of packets and recompile the firmware.

- `g_DATA_FIFO_DEPTH`

Positive variable that sets the FIFO depth of the QuickUSB on the Main FPGA. It should be more than twice the total number of samples where the larger, the better. For example, with 16 channels and a 256 sample size, this value is $2 \times 16 \times 256 = 8192$.

- `g_osc_num_adc_samples`

Positive variable that sets the total number of ADC samples per run. Currently, it is set to 256. It can be increased to 512, but space on the FPGA will need to be cleared to make room for it.

- `g_sng_max_pipeline_stages`

Positive type variable that is the maximum number of pipeline stages calculated by the following: $\text{ceil}(\text{app_processing_ticks}/\text{slice_width}) + 1$.

- `g_en_osc_mode`

Boolean that determines if oscilloscope mode data will be generated. If true, then it will generate oscilloscope mode data. If false, it will not.

- `g_sng_mode_type`

Integer that determines what kind, if any, singles mode data is generated. If the value is 0, no singles mode data is generated. If the value is 1 (default), it follows the LBNL example explained in [Singles Mode](#) (page 66).

Detector Board (16-Channel)

This section will detail the variables that are set in the firmware of the detector board. The firmware must be compiled every time these values are changed in order to be put into effect. These variables are found in `db16ch.vdh` located in the firmware directory of the `DetectorBoard16ChLBNL` repository.

- `g_en_debug`

Boolean variable that if set to true, debugging signals can be sent out of the detector board. If set to false, it cannot.

- `g_slice_div`

Integer that defines the time slice size. It sets the clock to slice relation. If the value is 0 (default), divide by 8, and if it is 1, divide by 16. This correlates to a 100ns or 200ns time slice. Must be the same as the Support Board value.

- `g_adc_channels`

Positive variable that specifies how many channels are connected to each ADC chip. For the LBNL 16-Channel Detector Boards, there are 8 channels connected to each ADC chip. To change this, a new board must be designed and built.

- `g_adc_number`

Positive variable that defines the number of ADC chips on the detector board. To change this, a new board must be designed and built. For the LBNL 16-Channel Detector Boards, there are 2 ADC chips. The `g_adc_channels` x `g_adc_number` equals the number of channels of the detector board (e.g., $8 \times 2 = 16$).

- `g_adc_resolution`

Positive variable that is the ADC resolution. This value is determined by the ADC chip used on the board. For the LBNL 16-Channel Detector Board, the ADC chip used is a 12 bit resolution.

- `g_tdc_type`

Integer that defines what kind of TDC logic is used. If the variable is set to 0, there is no TDC used. This frees up space on the FPGA for other processes. If the value is 1, the TDC is a waveunion type. This option has a higher resolution. If the value is 2, it is a multiphase TDC (default). See this [reference](#) for a more detailed description of the TDC types.

- `g_tdc_resolution`

Positive variable that defines the bit value of the TDC resolution. Currently, the system can have up to a 20 bit resolution.

- `g_en_osc_mode`

Boolean that determines if oscilloscope mode data will be generated. If true, then it will generate oscilloscope mode data. If false, it will not.

- `g_osc_num_adc_samples`

Positive variable that sets the total number of ADC samples per run. Currently, it is set to 256. It can be increased to 512, but space on the FPGA will need to be cleared to make room for it.

- `g_sng_mode_type`

Integer that determines what kind, if any, singles mode logic is generated. If the value is 0, no singles mode data is generated. If the value is 1 (default), it follows the LBNL example explained in [Singles Mode](#) (page 66).

- `g_sng_max_pipeline_stages`

Positive type variable that is the maximum number of pipeline stages calculated by the following: $\text{ceil}(\text{app_processing_ticks}/\text{slice_width}) + 1$.

Introduction

(Give overview of the comprehensive implementation plan roadmap.)

System Addressing

(Expand on Framework section 1.4)

Based on physical slot plugged in.

System Commands and Responses

OpenPET utilizes a standard 32-bit wide Serial Peripheral Interface (SPI) to facilitate serial communications between any parent node and its children. The communication protocol follows a [request-response](#) architecture, where a parent node writes a command to a single child or multiple children and reads back the response. For a gentle introduction on OpenPET commands, see the [Getting Started](#) section in the User's Guide.

OpenPET commands are 80-bits wide as shown in [Fig. 4.1](#). The first 16 most significant bits are the command ID, followed by the source address (16 bits), destination address (16 bits), and payload (32 bits).



Fig. 4.1: OpenPET command (80-bits)

Starting from the most significant bit (MSB):

- Command ID is defined below
- SRC/DST source/destination address is defined below
- Payload is defined per command (look in command folder)

The command id (Fig. 4.2) specifies the function of the command, using a 16-bit number. The most significant bit has two uses:

- used as a flag to denote a response/reply/acknowledgment from a node to its parent (direction=child-to-parent).
- used as a flag to denote a non-blocking command i.e., asynchronous command (direction=parent-to-child).



Fig. 4.2: Command ID (16-bits)

Starting from the least significant bit (LSB):

(14:0)	Command ID
(15)	Dual use flag
	(a) Child sets it to '1' when it responds to a parent
	(b) Parent sets it to '1' when it doesn't want to wait for the targeted child's response, i.e., non-blocking command or asynchronous command.
	Note: The targeted child will not reply to other commands if it is still busy executing this asynchronous command.

The source address is a 16-bit number that defines where the command originates. Typically, commands that originate at the Host PC have a source address of 0x4000. The destination address is a 16-bit number that identifies where the response should be received and processed. Both the source and destination addresses have the same address format, as shown in Fig. 4.3.

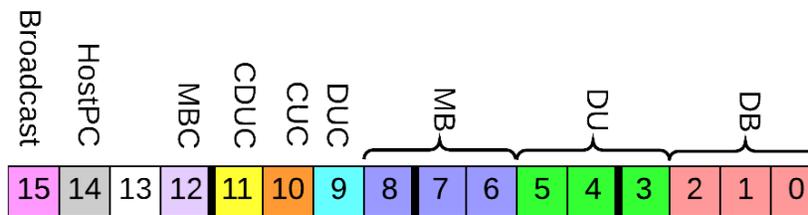


Fig. 4.3: Source/Destination address (16-bits)

Starting from least significant bit (LSB):

(2:0)	Detector Board Address
(5:3)	Detector Unit Address
(8:6)	Multiplexer Board Address
(9)	Detector Unit Controller source/destination flag
(10)	Coincidence Unit Controller source/destination flag
(11)	Coincidence Detector Unit Controller source/destination flag
(12)	Multiplexer Board Controller source/destination flag
(13)	Not used
(14)	Host PC source/destination flag
(15)	Broadcast flag

If the Broadcast flag is set in the destination address, the source node will pass the command down to all of its “children” and the child specified in the destination address will be read back to in order to create the response.

If the CUC or CDUC flags are set in the destination address, that corresponding unit will execute the command and respond. Namely, the response will not come from the unit specified in the destination address.

Finally, the payload is a 32-bit number that specifies additional information for each command; see below for examples.

Any operating system (Windows, GNU/Linux, or Mac OS) and programmable language (C, C++, Delphi, MATLAB, VB.NET, VC#, and Delphi) supported by QuickUSB can be used to interface with the OpenPET system. OpenPET provides multiple methods to control and configure the system. The simplest method is to use `openpet.exe` which is a Microsoft Windows executable that can configure and acquire data from an OpenPET system. Additionally, platform independent example Python scripts are also provided to streamline the configuration and acquisition process.

The executable `openpet` is used to control and configure the system. It has several optional arguments, list them by running `openpet` with `-h` or `--help` switches:

```
usage: openpet.py [-h] [-v] [-L] [-l] [-d DEVICEINDEX | -i INTERFACEINDEX]
  [-D IP]
  [-c ID DST PAYLOAD | -a DURATION | -sr FILE DST SIZE OFFSET | -sw FILE DST OFFSET]
  [-o FILE] [-t TIMEOUT] [-n RETRIES]

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         Show debugging info.
  -L, --List            List network devices.
  -l, --list           List quickusb devices.
  -d DEVICEINDEX, --device DEVICEINDEX
                        Quickusb device index.
                        List devices to see indexes.
                        DEFAULT=0
  -i INTERFACEINDEX, --interface INTERFACEINDEX
                        Network interface device index.
                        List devices to see indexes.
  -D IP, --destination IP
                        IPv4 address of of destination node.
  -c ID DST PAYLOAD, --command ID DST PAYLOAD
                        Sends a command to a destination
                        module with a specific payload.
  -a DURATION, --acquire DURATION
                        Acquire data for the specified
                        duration (in seconds). Partial
                        seconds are OK.
  -sr FILE DST SIZE OFFSET, --sram-read FILE DST SIZE OFFSET
                        Reads SRAM contents from OpenPET and
                        writes it to FILE.
  -sw FILE DST OFFSET, --sram-write FILE DST OFFSET
                        Writes FILE contents to SRAM.
  -o FILE, --outputfile FILE
                        File name to save acquired data.
  -t TIMEOUT, --timeout TIMEOUT
                        Timeout duration (in seconds) between
                        retries. Partial seconds are OK.
                        DEFAULT=0.200
  -n RETRIES, --retries RETRIES
                        Number of times I should try to
```

```
contact OpenPET System before  
giving up. DEFAULT=20
```

Tables 8.1 and 8.2 show a list of the current OpenPET commands, including their command ID and a brief description of their function. The payload is specified for each command in the examples that follow.

Table 4.1: Summary of the OpenPET system commands.

IDs	Name	Function
<i>0x0001</i> (page 46)	Ping	Sends a single ping request to destination.
<i>0x0002</i> (page 47)	Write Children Bitstream	Command Support Board(s) to configure all children boards from bitstream stored in EPCS.
<i>0x0003</i> (page 47)	Write System Acquisition Mode	Sets the system mode register in firmware to idle, scope, singles, etc.
<i>0x0004</i> (page 47)	Read System Acquisition Mode	Gets the system mode register from firmware.
<i>0x0005</i> (page 48)	Write System Acquisition Mode Settings	Sets the system mode settings register in firmware for the mode selected.
<i>0x0006</i> (page 49)	Read System Acquisition Mode Settings	Gets the system mode settings register from firmware.
<i>0x0007</i> (page 49)	Write System Acquisition Mode Action	Sets the system mode action register in firmware to reset, start, stop.
<i>0x0008</i> (page 50)	Read System Acquisition Mode Action	Gets the system mode action register from firmware.
<i>0x0009</i> (page 51)	Write Trigger Mask	Sets a mask to suppress triggers.
<i>0x000A</i> (page 51)	Read Trigger Mask	Gets the mask from firmware.
<i>0x000B</i> (page 51)	Write SRAM Data	Writes to external SRAM device. Auto-increments address.

<i>0x000C</i> (page 52)	Read SRAM Data	Reads from external SRAM device. Auto-increments address.
<i>0x000D</i> (page 52)	Zero out SRAM	SRAM content is zeroed out.
<i>0x000F</i> (page 53)	Reset	Reset all configurations.
<i>0x0101</i> (page 53)	Write TDC Configuration	Sets the TDC control register.
<i>0x0102</i> (page 54)	Read TDC Configuration	Gets the TDC control register.
<i>0x0103</i> (page 54)	Reset ADC Configuration	Command Detector Boards(s) to set ADCs registers to OpenPET default values.
<i>0x0104</i> (page 54)	Write ADC Register	Writes a register directly on ADC(s). See ADC datasheet for valid register maps.
<i>0x0105</i> (page 56)	Reset DAC Configuration	Command Detector Board(s) to set DACs registers to OpenPET default values.
<i>0x0106</i> (page 56)	Write DAC Register	Write a register directly on DAC(s). See DAC datasheet for valid register maps.
<i>0x0107</i> (page 58)	Write Sawtooth pulse(s)	Command DAC(s) to send sawtooth pulses for a given duration of time.
<i>0x0108</i> (page 59)	Write Firmware Threshold	Sets a firmware threshold level to trigger on.
<i>0x0109</i> (page 59)	Read Firmware Threshold	Gets the firmware trigger threshold.

Table 4.2: Summary of the OpenPET error codes for replies.

IDs	Name	Function
0x0000	(Reserved) Busy	Child doesn't have anything to reply yet.
0xFFFF	(Reserved) Dead	Dead, nonexistent, or not programmed.
0x7F00	SW Command is Unknown	Software (Running on NIOS) command id is unknown to node.
0x7F01	SW Command Timed Out	Software (Running on NIOS) command id has timed out.
0x7F02	Targeted Child is Dead	Targeted child is dead, nonexistent, or not programmed.
0x7F03	Targeted Child is Busy	Targeted child is busy processing previous command.
0x7F04	FW Command is Unknown	Firmware (Running on FPGA fabric) command id is unknown to node.
0x7F05	FW Command Timed Out	Firmware (Running on FPGA fabric) command id has timed out.
0x7F06	SW incomplete packet	Software (Running on NIOS) received incomplete OpenPET command
0x7F07	SW interrupt routine can't keep up	Software (Running on NIOS) received packets faster than it can handle.

Description and Examples of Command IDs

Command ID: 0x0001

Description: Sends a single ping request to destination. If broadcast on DST was specified, then the read back will be performed on the address provided.

Payload: None e.g. 0 or any value.

Examples:

Send a ping command using the `openpet` executable to destination 0x0002 i.e., DB in slot 2 with 0 payload. `openpet` will print out the date, time, message type, [S]ENT or [R]ECEIVED indicator, command id, destination (when sending) or source (when receiving) address, and the payload. Note that the MSB of the command ID is set to '1' when receiving.:

```
$ openpet -c 0x0001 0x0002 0
2015-09-01 12:30:00,529 INFO [S] 0x0001 0x0002 0x00000000
2015-09-01 12:30:00,733 INFO [R] 0x8001 0x0002 0x00000000
```

Just like the first example, however, using decimal numbers instead of hex:

```
$ openpet -c 1 2 0
```

The destination address has the broadcast flag set to 1. This will cause the SB to send the ping command to all its children, however, the reply will be read back only from 0x0002:

```
$ openpet -c 0x0001 0x8002 0
```

The asynchronous flag in the command id is set to 1. This will cause the SB to immediately respond to the HostPC regardless if the ping command was successful on 0x0002 or not. This feature becomes useful when sending commands that take minutes to complete. Note, that the targeted destination will not be able to respond until it completes the execution of the asynchronous command, however, other nodes can be accessed and controlled independently:

```
$ openpet -c 0x1001 0x0002 0
```

QuickUSB device index 1 is used instead of the default device:

```
$ openpet -d 1 -c 1 2 0
```

Using Ethernet instead of default QuickUSB device. By default the IP address of the OpenPET chassis is 10.10.10.2:

```
$ openpet -D 10.10.10.2 -c 1 2 0
```

The HostPC is willing to wait 1 second for the SB to provide a valid reply. The default timeout per try is 200ms:

```
$ openpet -t 1 -c 1 2 0
```

The HostPC tries a maximum of 3 times before giving up on the SB. The wait between each trial is 0.5 second:

```
$ openpet -n 3 -t 0.5 -c 1 2 0
```

More debugging information is displayed:

```
$ openpet -v -c 1 2 0
```

Command ID: 0x0002

Description: FPGA bitstream configuration is read from EPCS flash memory then written to all children. This command is executed on power-up by default.

Payload: None e.g. 0 or any value.

Examples:

Ask CDUC to configure all of its children i.e., detectorboard FPGAs (not io):

```
$ openpet -c 2 0x0800 0
```

Command ID: 0x0003

Description: Writes the System Acquisition Mode register in firmware.

Payload:



Fig. 4.4: System Acquisition Mode payload

```
Starting from least significant bit (LSB)
(3:0) Mode
(31:4) not used
```

Available Modes:

IDLE	0x0
Oscilloscope	0x1
Singles	0x2

Examples:

Broadcast to all nodes to set System Acquisition Mode to scope mode. The command acknowledgment is received from DB in slot 3:

```
$ openpet -c 3 0x8003 1
```

Command ID: 0x0004

Description: Reads the System Acquisition Mode register from firmware.

Payload: None e.g. 0 or any value.

Examples:

Read the System Acquisition Mode register from firmware from slot 5. The payload of the reply is the mode that was previously set i.e., 1=scope mode:

```
$ openpet -c 4 5 0xDEADFEED
2015-09-02 11:10:00,528 INFO [S] 0x0004 0x0005 0xDEADFEED
2015-09-02 11:10:00,732 INFO [R] 0x8004 0x0005 0x00000001
```

Command ID: 0x0005

Description: Writes the System Acquisition Mode Settings register in firmware. The 32-bit setting payload value depends on the mode set in Command ID 0x0003.

Payload:

- Payload when System Acquisition Mode = 0x1 = Oscilloscope (scope mode):

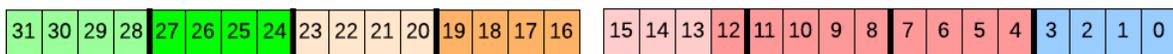


Fig. 4.5: System Acquisition Mode Settings payload for oscilloscope mode

```
Starting from least significant bit (LSB)
(3:0)   Reserved: Must be 0001
(12:4)  Total Number of ADC samples, see notes below (zero is accounted for)
(15:13) Reserved
(19:16) Number of ADC samples before energy trigger (2^4 = 16)
(23:20) Reserved
(27:24) Trigger window (2^4 = 16)
(31:28) Reserved
```

- Payload when System Acquisition Mode = 0x2 = Singles (singles mode):



Fig. 4.6: System Acquisition Mode Settings payload for singles mode

```
Starting from least significant bit (LSB)
(3:0)   Total number of ADC clock ticks to finish a single Event computation
         (2^4 = 16)
(7:4)   Reserved, max ADC clock ticks to process data (2^8 = 256)
(15:8)  Reserved
(31:16) Not Used
```

Note:

- **Scope Mode:**
 - Total Number of samples should be greater than samples before trigger + trigger window.
 - Total Number of samples should not exceed Firmware's maximum number of samples - (Number of Channel headers + Detector Board Header)
- **Singles Mode:**
 - Event computation clocks ticks should be greater than 1.
 - Number of pipeline stages is pre-defined in the firmware as a constant

- $\text{PipelineStages} = \text{ceil}(\text{EventCompuationClockTicks}/\text{SliceWidth}) + 1$, where $\text{ceil}()$ rounds the number to the next highest integer.

Examples:

Broadcast to all nodes to set System Acquisition Mode to scope mode. Then use data format = 1, 16 samples, 0 samples before trigger, and a trigger window to 2:

```
$ openpet -c 3 0x8003 1
$ openpet -c 5 0x8003 0x02000101
```

Broadcast to all nodes to set System Acquisition Mode to singles mode. Then TODO FIXME:

```
$ openpet -c 3 0x8003 2
$ openpet -c 5 0x8003 0x00000001
```

Note: If the user sets a parameter with a value that is too large, the system will set the parameter to the maximum possible value. This will be evident in the response payload. It will show the maximum possible value instead of the user's value. The user can then decide to continue or change it.

Command ID: 0x0006

Description: Reads the System Acquisition Mode Settings register from firmware.

Payload: None e.g. 0 or any value.

Examples:

Reads the System Acquisition Mode Settings register from firmware from slot 3. The payload of the reply is set to the settings previously specified, e.g., scope mode settings:

```
$ openpet -c 6 3 0xDEADFEED
2015-09-02 11:10:00,528 INFO [S] 0x0006 0x0003 0xDEADFEED
2015-09-02 11:10:00,732 INFO [R] 0x8006 0x0003 0x02000101
```

Command ID: 0x0007

Description: Writes the System Acquisition Mode Action register in firmware.

Payload:



Fig. 4.7: System Acquisition Mode Action payload

```
Starting from the least significant bit (LSB)
(3:0) Action
(31:4) not used
```

Available actions:

Reset	0x0
Stop	0x1
Run	0x2

Examples:

Reset the System Acquisition Mode and Settings (payload = 0x0):

```
$ openpet -c 7 0x8003 0x00000000
```

Stop the current system acquisition (payload = 0x1):

```
$ openpet -c 7 0x8003 0x00000001
```

First, broadcast to all nodes to set System Acquisition Mode to scope mode. Then use data format = 0, 16 samples, 0 samples before trigger, and a trigger window to 2. Finally, run the acquire data command (payload = 0x2):

```
$ openpet -c 3 0x8003 1
$ openpet -c 5 0x8003 0x02000100
$ openpet -c 7 0x8003 0x00000002
```

Command ID: 0x0008

Description: Reads the System Acquisition Mode Action register from firmware.

Payload: None e.g. 0 or any value.

Examples:

Reads the System Acquisition Mode Action register from firmware from slot 3. The payload of the reply is set to the action previously specified, e.g., reset = 0:

```
$ openpet -c 8 3 0xDEADFEED

2015-09-02 11:10:00,528 INFO [S] 0x0008 0x0003 0xDEADFEED
2015-09-02 11:10:00,732 INFO [R] 0x8008 0x0003 0x00000000
```

If action previously set to terminate the current action, e.g., stop = 1:

```
$ openpet -c 8 3 0xDEADFEED

2015-09-02 11:10:00,528 INFO [S] 0x0008 0x0003 0xDEADFEED
2015-09-02 11:10:00,732 INFO [R] 0x8008 0x0003 0x00000001
```

If action previously set to run the system acquisition, e.g., run = 2:

```
$ openpet -c 8 3 0xDEADFEED
2015-09-02 11:10:00,528 INFO [S] 0x0008 0x0003 0xDEADFEED
2015-09-02 11:10:00,732 INFO [R] 0x8008 0x0003 0x00000002
```

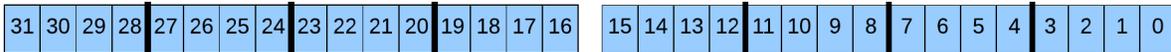
Command ID: 0x0009**Description:** Writes a mask to suppress triggers in firmware.**Payload:**

Fig. 4.8: Trigger mask payload

```
Starting from the least significant bit (LSB)
(31:0) 1 bit per channel. Set bit to 1 to enable trigger on that channel.
```

Example:

Set trigger masks for channels 0, 5, and 8 in detector board 5:

```
$ openpet -c 9 5 0x00000121
```

Command ID: 0x000A**Description:** Reads the trigger mask from firmware.**Payload:** None e.g. 0 or any value.**Example:**

Reads the trigger mask previously set in detector board 5:

```
$ openpet -c 10 5 0xDEADFEED
2015-09-02 11:10:00,528 INFO [S] 0x000A 0x0005 0xDEADFEED
2015-09-02 11:10:00,732 INFO [R] 0x800A 0x0005 0x00000121
```

Command ID: 0x000B**Description:** Writes to external SRAM device. Auto-increments address.**Payload:**

```
Starting from the least significant bit (LSB)
(31:0) Value to write to SRAM
```

Caveats: The Mode Action has to be in Reset for this command to work correctly.

Example:

Write some value to external SRAM in detector board 1:

```
$ openpet -c 11 1 0x12345678
2015-10-12 11:02:33,115 INFO [S] 0x000B 0x0001 0x12345678
2015-10-12 11:02:33,355 INFO [R] 0x800B 0x0001 0x12345678
```

Alternative: -sw option Write to SRAM in detector board 1 from file sramtest.bin starting from the top (offset = 0):

```
$ openpet -sw sramtest.bin 1 0
2015-10-12 11:05:22,956 INFO Writing content to SRAM.
2015-10-12 11:05:28,385 INFO Done SRAM writing.
```

Command ID: 0x000C

Description: Reads from external SRAM device. Auto-increments address.

Payload:

```
Starting from the least significant bit (LSB)
(31:0) SRAM address
```

Caveats: The Mode Action has to be in Reset for this command to work correctly.

Example:

Read data starting from SRAM address 0 in detector board 1:

```
$ openpet -c 12 1 0
2015-10-12 11:02:41,956 INFO [S] 0x000C 0x0001 0x00000000
2015-10-12 11:02:42,165 INFO [R] 0x800C 0x0001 0x12345678
```

Alternative: -sr option Read 100 values from SRAM to file sramtest.bin from detector board 1 starting from the top (offset = 0):

```
$ openpet -sr sramtest.bin 1 100 0
2015-10-12 11:03:11,296 INFO Reading SRAM content.
2015-10-12 11:03:15,635 INFO Done Reading SRAM Content.
```

Command ID: 0x000D

Description: Clears SRAM content. Sets to zero.

Payload: None e.g. 0 or any value.

Example:

Clears SRAM content on detector board 1:

```
$ openpet -c 13 1 0xDEADFEED
2015-10-09 11:10:00,528 INFO [S] 0x000D 0x0001 0xDEADFEED
2015-10-09 11:10:00,732 INFO [R] 0x800D 0x0001 0x00000000
```

Command ID: 0x000F

Description: System wide reset. It resets the configurations of firmware, software, and peripheral hardware to default values:

****Payload**:** **None** e.g. 0 **or** any value.

Example:

Broadcast a reset to all nodes and get a reply from 0x1:

```
$ openpet -c 0xF 0x8001 0
2015-10-09 11:10:00,511 INFO [S] 0x000F 0x8001 0x00000000
2015-10-09 11:10:00,733 INFO [R] 0x800F 0x8001 0x00000000
```

Command ID: 0x0101

Description: Sets the TDC control register.

Payload:

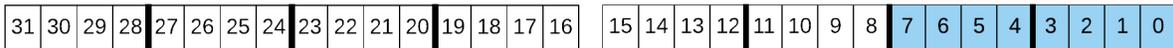


Fig. 4.9: Payload for TDC control register

```
Starting from the least significant bit (LSB)
(7:0)   TDC command
(31:8)  Not used
```

Available TDC commands:

Reset	0x80
Calibrate	0x02
Run	0x04

Examples:

Set the TDC control register to reset in detector board 5:

```
$ openpet -c 0x0101 5 0x00000080
```

Set the TDC control register to calibrate in detector board 5:

```
$ openpet -c 0x0101 5 2
```

Set the TDC control register to run in detector board 5:

```
$ openpet -c 0x0101 5 4
```

Command ID: 0x0102

Description: Gets the TDC control register.

Payload: None e.g. 0 or any value.

Example:

Reads the current TDC control register from detector board 5, e.g., run = 1:

```
$ openpet -c 0x0102 5 0xDEADFEED
2015-10-09 11:10:00,528 INFO [S] 0x0102 0x0005 0xDEADFEED
2015-10-09 11:10:00,732 INFO [R] 0x8102 0x0005 0x00000001
```

Command ID: 0x0103

Description: Resets ADC configuration. Commands Detector Board(s) to set ADC registers to OpenPET default values.

Payload: None e.g. 0 or any value.

Example:

Resets ADC registers on detector board 3 to default OpenPET values:

```
$ openpet -c 0x0103 3 0xDEADFEED
2015-10-09 11:10:00,528 INFO [S] 0x0103 0x0003 0xDEADFEED
2015-10-09 11:10:00,732 INFO [R] 0x8103 0x0003 0x00000000
```

Command ID: 0x0104

Description: Writes ADC register. See [ADS5282 datasheet](#) for valid register maps.

Payload:

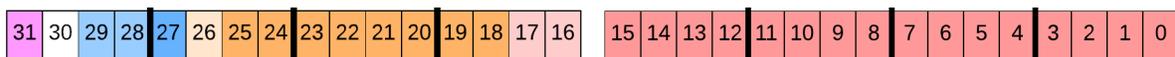


Fig. 4.10: Payload to write ADC register

```
Starting from the least significant bit (LSB)
(15:0)  ADC data (see ADS5282 datasheet, page 17)
(17:16) Reserved
(25:18) ADC register address (see ADS5282 datasheet, page 17)
(26)    Reserved
(27)    ADC chip address (0 for channel 0-7; 1 for channel 8-15)
(29:28) Reserved
(30)    Not used
(31)    Broadcast flag, i.e., run on all ADC chips
```

Note: The ADC address and ADC data for the corresponding DB channel when setting the ADC gain. Each gain is set using 4 bits ranging from 0-12 dB. The gain has to be set on four channels at a time (i.e., 16-bit ADC data).

ADC Chip Address	ADC Address	ADC Data (MSB to LSB)
0x0	0x2A	Channel 3 to 0
0x0	0x2B	Channel 4 to 7
0x1	0x2A	Channel 11 to 8
0x1	0x2B	Channel 12 to 15

Examples:

Sets the ADC gain to 6 dB for channels 4 to 7 on the Detector Board in slot 3 without broadcasting:

```
$ openpet -c 0x0104 0x0003 0x00AC6666
```

Payload breakdown (LSB to MSB):

- 0x00AC6666 (=0 0 00 0 0 00101011 00 0110011001100110)
- Bits 15-0: 0110011001100110 (see ADS5282 datasheet, page 17)
- Bits 17-16: Reserved
- Bits 25-18: 0x2B indicates for channel 4 to 7
- Bits 26: Reserved
- Bits 27: 0 indicates for channel 4 to 7
- Bits 29-28: Reserved
- Bits 30: Not used
- Bits 31: 0 indicates not a broadcast command

Sets the ADC gain to 8 dB for channels 8 to 11 on the Detector Board in slot 5 with broadcasting:

```
$ openpet -c 0x0104 0x0005 0x88A88888
```

Payload breakdown (LSB to MSB):

- 0x88A88888 (=1 0 00 1 0 00101010 00 1000100010001000)
- Bits 15-0: 1000100010001000 (see ADS5282 datasheet, page 17)
- Bits 17-16: Reserved
- Bits 25-18: 0x2A indicates for channel 8 to 11

- Bits 26: Reserved
- Bits 27: 1 indicates for channel 8 to 11
- Bits 29-28: Reserved
- Bits 30: Not used
- Bits 31: 1 indicates broadcast command

Command ID: 0x0105

Description: Resets DAC configuration. Commands Detector Board(s) to set DAC registers to OpenPET default values.

Payload: None e.g. 0 or any value.

Example:

Resets DAC registers on detector board 3 to default OpenPET values:

```
$ openpet -c 0x0105 3 0xDEADFEED
2015-10-09 11:10:00,528 INFO [S] 0x0105 0x0003 0xDEADFEED
2015-10-09 11:10:00,732 INFO [R] 0x8105 0x0003 0x00000000
```

Command ID: 0x0106

Description: Writes DAC register. See [DAC LTC2634 datasheet](#) for valid register maps.

Payload:



Fig. 4.11: Payload to write DAC register

```
Starting from the least significant bit (LSB)
(9:0) DAC data (see LTC2634 datasheet, page 20)
(12:10) Reserved
(16:13) DAC address (see LTC2634 datasheet, page 20)
(20:17) DAC command (see LTC2634 datasheet, page 20)
(22:21) Reserved
(26:23) DAC chip address
(28:27) DAC type (00=energy, 01=timing, 10=reserved, 11=all)
(30:29) Not used
(31) Broadcast flag, i.e., run on all DAC chips for a given type
```

Note: DAC Command=0x3 to set DAC voltage (10-bit data)

Type	Bits 28-27	DAC Full Scale
Energy	00	4.096 V
Timing	01	2.500 V
Reserved	10	2.500 V

The DAC address and DAC chip select for the corresponding DB channel when setting the energy and timing DAC. Because a chip has four DAC, a single DAC voltage can be set to four channels with one command.

Channel	DAC Address	DAC Chip Address
0	0x0	0x0
1	0x1	0x0
2	0x2	0x0
3	0x3	0x0
4	0x0	0x1
5	0x1	0x1
6	0x2	0x1
7	0x3	0x1
8	0x0	0x2
9	0x1	0x2
10	0x2	0x2
11	0x3	0x2
12	0x0	0x3
13	0x1	0x3
14	0x2	0x3
15	0x3	0x3
0-3	0xF	0x0
4-7	0xF	0x1
8-11	0xF	0x2
12-15	0xF	0x3

- The timing DAC is usually set to a low threshold to obtain the best timing.
- The energy DAC determines whether readout is initiated, and so is usually set to a higher threshold to reduce noise triggers (see *16-Channel Detector Board* (page 28)).

Examples:

Sets DAC energy threshold to +1.150V for channel 4 on the Detector Board in slot 3 without broadcasting:

```
$ openpet -c 0x0106 0x0003 0x00860120
```

Payload breakdown (LSB to MSB):

- 0x000860120 (=0 00 00 0001 00 0011 0000 000 0100100000)
- Bits 9-0: 0100100000 (1024*1.15/4.096)
- Bits 12-10: Reserved
- Bits 16-13: 0x0 (channel 4 in chip select 0x1)
- Bits 20-17: 0x3 (see [DAC LTC2634 datasheet](#), page 20)
- Bits 22-21: Reserved
- Bits 26-23: 0x1 indicates this is the chip where channel 4 resides
- Bits 28-27: 00 indicates that the DAC type is energy
- Bits 30-29: Not used
- Bits 31: 0 indicates not a broadcast command

Sets DAC timing threshold to +0.5V for channel 12 on the Detector Board in slot 5 with broadcasting:

```
$ openpet -c 0x0106 0x0005 0x898600CD
```

Payload breakdown (LSB to MSB):

- 0x898600CD (=1 00 01 0011 00 0011 0000 000 0011001101)
- Bits 9-0: 0011001101 (1024*0.5/2.5)
- Bits 12-10: Reserved
- Bits 16-13: 0x0 (channel 12 in chip select 0x3)
- Bits 20-17: 0x3 (see [DAC LTC2634 datasheet](#), page 20)
- Bits 22-21: Reserved
- Bits 26-23: 0x3 indicates this is the chip where channel 12 resides
- Bits 28-27: 01 indicates that the DAC type is timing
- Bits 30-29: Not used
- Bits 31: 1 indicates a broadcast command

Command ID: 0x0107

Description: Writes Sawtooth pulse(s). Commands DAC(s) to send sawtooth pulse(s) for a given duration of time.

Payload:

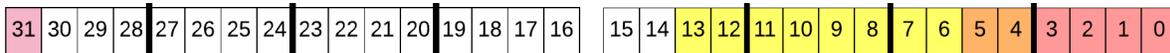


Fig. 4.12: Payload to write sawtooth pulse

```
Starting from the least significant bit (LSB)
(3:0) DAC chip address
(5:4) DAC type
(13:6) Number of sawtooth pulses to send
(30:14) Not used
(31) Broadcast flag, i.e., run on all DAC chips for a given type
```

Note: The time it takes to execute and finish this command depends on the number of sawtooth pulses you request. It is highly advised to send this command as an asynchronous command (non-blocking) by setting the MSB on CMD ID to '1'.

Example:

Command DAC to send 100 sawtooth pulses to detector board 3. DAC type is 00 which is energy, and chip address is 0x2. Broadcast flag is set to 1, and the CMD ID MSB is set to 1 for asynchronous command:

```
$ openpet -c 0x8107 0x0003 0x80001902
```

Command ID: 0x0108

Description: Sets a firmware threshold level to trigger on.

Payload:



Fig. 4.13: Payload to set firmware trigger threshold

```
Starting from the least significant bit (LSB)
(11:0)  Threshold value
(15:12) Reserved
(17:16) Mode (00=off, 01=on)
(31:18) Not used
```

Note: The minimum and maximum threshold values correspond to the minimum and maximum of the ADC signal which is 2Vpp. The 0V value is in the middle so to set at the 0V DC, set the MSB of the threshold value to 1. To set the maximum threshold value (1V), flip all bits 0 to 11 to 1. To set the minimum threshold value (-1V), flip all bits 0 to 11 to 0.

Example:

Sets the firmware threshold level to 0V and mode 'on' for detector board 5:

```
$ openpet -c 0x0108 0x0005 0x00010800
```

Command ID: 0x0109

Description: Gets the firmware trigger threshold.

Payload: None e.g. 0 or any value.

Example:

Retrieves the firmware trigger threshold previously set on detector board 5:

```
$ openpet -c 0x0109 0x0005 0xDEADFEED
2015-10-09 11:10:00,528 INFO [S] 0x0109 0x0005 0xDEADFEED
2015-10-09 11:10:00,732 INFO [R] 0x8109 0x0005 0x00010800
```

Data Modes

There are different data-gathering modes that can be created and used with the OpenPET system. Currently, there are three modes available with 2^{32} modes possible.

Mode Type	Data Description
<i>Scope Mode</i> (page 60)	Raw ADC data
<i>Singles Mode</i> (page 66)	Processed ADC data, e.g., energy
Idle Mode	Nothing processed or transferred

For each mode, there are actions that can be performed. Again, there are 2^{32} actions possible with three available so far.

Action	Description
Run	Launches a selected Mode
Stop	Causes a selected Mode to pause
Reset	Causes a selected Mode to reset to default

The settings for a selected mode is stipulated by a 32-bit word that is described in each mode section

Scope Mode

In Scope mode, raw ADC data is sent to an external disk. The data file is a binary file and shown in Fig. 4.14. It is a simple 32-bit format for easy parsing and manipulation. The headers are prepended to the raw data including information such as time of acquisition and all the acquisition settings. There is 4KB reserved for user defined content.

It uses a 16-bit wide bus to transfer 32-bits at a time via DDR. Each 32-bit packet has a 4-bit packet ID. Each detector board wraps its data with a DB header, and each channel in a given detector board wraps its data with a channel header (Fig. 4.15). All detector boards are synchronized when they start the acquisition.

Fig. 4.16, Fig. 4.17, and Fig. 4.18 show the bit maps of the headers and data pack mentioned above.

```
Starting from the least significant bit (LSB)
(5:0)   Number of channel header packets (i.e., 0 to 63)
(9:6)   Not used
(12:10) Detector board address (populated by parent)
(15:13) DU address (populated by parent)
(18:16) MB address (populated by parent)
(27:19) Not used
(31:28) Packet ID (must equal 0x4)
```

```
Starting from the least significant bit (LSB)
(19:0)  TDC data (currently using (14:0) with (19:15) for expansion)
(20)    Hardware trigger hit (energy)
(21)    Firmware trigger hit
(27:22) Channel address (i.e., 0 to 63)
(31:28) Packet ID (must equal 0x3)
```

```
Starting from the least significant bit (LSB)
(27:0)  Raw ADC data
(31:28) Packet ID (must equal 0x1)
```

For a given 32-bit packet or header, the Packet ID is the four most significant bits as shown in Fig. 4.19. It is used to verify the type of packet received at a parent. It also allows a node to identify, validate, and semi-confirm the integrity of the packet in the data path.

Offset	[31..24]	[23..16]	[15..8]	[7..0]	Comment
0	Epoch timestamp				Local Time
1	Reserved				
2	Acquisition Duration				From Software
3	Acquisition Mode Payload				See cmd id 0x0003
4	Acquisition Mode Settings Payload				See cmd id 0x0005
5	Reserved				future data format
6	Misc. Software Settings				processed/raw?
7	Reserved				
...					
9					
10	User Defined				
...					
999					
B=1000	Detector Board Header				M=Num of channels
B+1	Channel Header (0)				
B+2	ADC Sample (0)				N=number of ADC samples. *
B+3	ADC Sample (1)				
	...				
B+N	ADC Sample (N-1)				
B+1+N	Channel Header (1)				
	...				
	Channel Header (M-1)				
	...				
B+M*(1+N)	Detector Board Header				Next Detector Board
	...				
	Detector Board Header				Next Detector Board
	...				

Fig. 4.14: Binary file format for scope mode

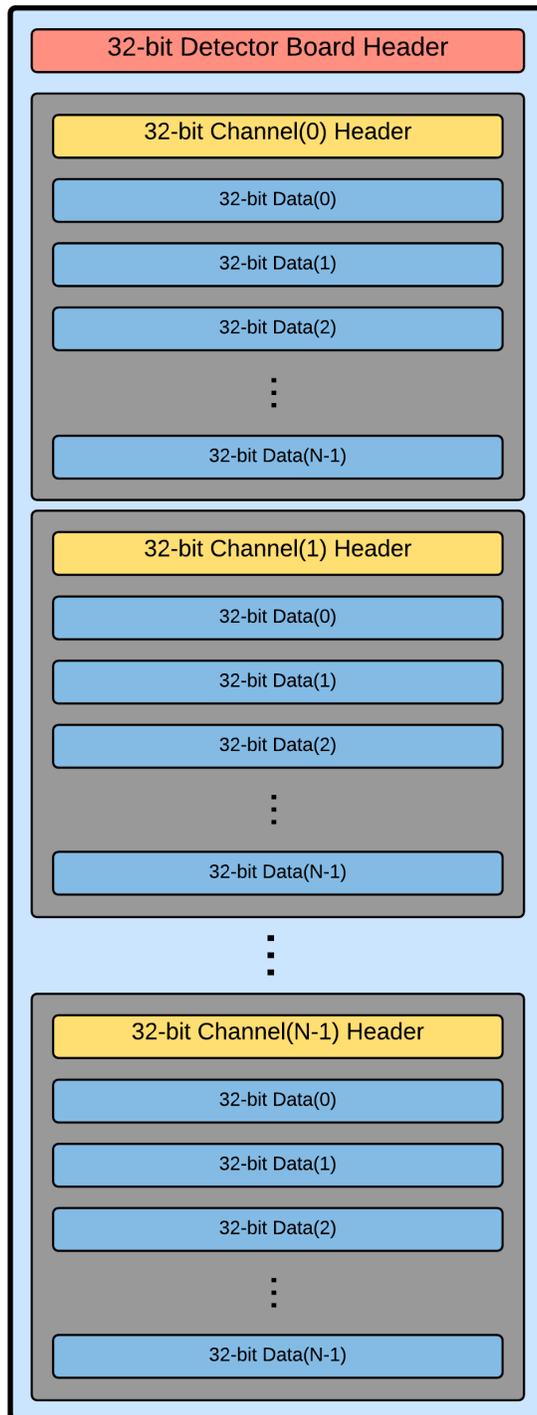


Fig. 4.15: Visual representation of the raw data from one detector board.

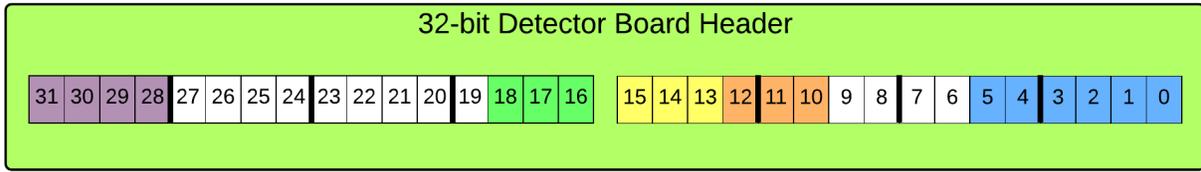


Fig. 4.16: Detector board header in scope mode

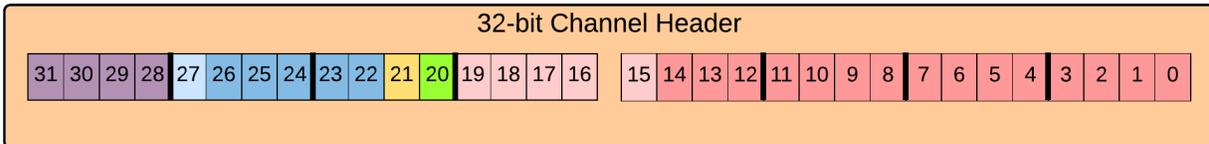


Fig. 4.17: Channel header in scope mode

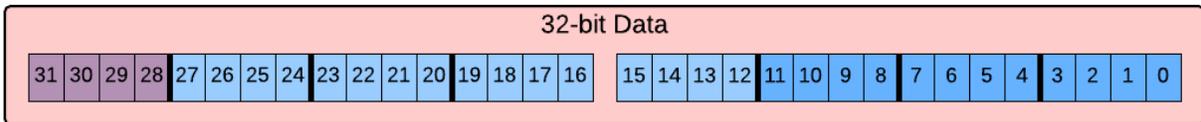


Fig. 4.18: Data packet in scope mode. In this example, the raw ADC data is from (11:0)

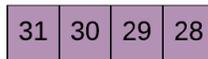


Fig. 4.19: Packet ID for scope mode. It is bits (31:28) for the scope headers and data packet above.

ID	Description
0x0	Reserved. Do NOT use
0x1	ADC data
0x2	Reserved
0x3	Channel header
0x4	Detector board header
0x5	DUC header
0x6	CUC header
0x7	CDUC header
0x8	MBC header
0x9	Host PC header
0xA-F	Not used

The bit assignment for the scope mode settings are described below with Fig. 4.20. They are set using *Command ID: 0x0005* (page 48).

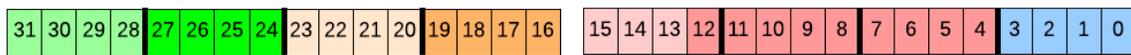


Fig. 4.20: Scope mode settings

```
Starting from the least significant bit (LSB)
(3:0)   Reserved (must equal 0001)
(12:4)  Total number of ADC samples ( $2^9 = 512$ ; zero is accounted for)
(15:13) Reserved
(19:16) Number of ADC samples before energy trigger ( $2^4 = 16$ )
(23:20) Reserved
(27:24) Trigger window ( $2^4 = 16$ )
(31:28) Reserved
```

Note:

- The total number of samples should be greater than the sum of the samples before the trigger and the trigger window.
- The total number of samples should not exceed the firmware's maximum number of samples minus the sum of the channel and detector board headers.

Also, the FIFO stores the ADC samples along with the detector board and channel headers. Since the FIFO always has a fixed size dependent on the firmware, the number of ADC samples is limited. Therefore, if the FIFO maximum size is 32, and the user sets the total number of ADC samples to be 32, there will only be 15 ADC samples because 16 slots are taken up by the 16 channel headers, and 1 slot is filled with the detector board header. If the user sets to collect only 5 ADC samples, then there will be 5 samples since the FIFO has room for all five. The maximum number of ADC samples that can be acquired is actually the FIFO size minus 17 (16 channel headers and 1 detector board header).

In scope mode, the data is transferred through multiplexing that can be explained through a small system example. With a single chassis, the multiplexing occurs in the support board. There is no data loss. For the IO FPGAs, there are four inputs, one for each detector board, and one output going to the Main FPGA. For the Main FPGA, there are two inputs, one for each IO FPGA, and one output going to the Host PC. A basic schematic is shown in Fig. 4.21. The code is reusable on both the Main and IO FPGAs.

Two options are available for multiplexing. The default setting is a synchronized data management across all detector boards, i.e., an aligned readout on all detector boards. It is fair to random triggers (round-robin), biased toward periodic

triggers, and has a longer dead time. The other option is a first-in-first-out scheduler. It is fair to periodic triggers, biased toward detector boards with higher trigger rates, and has a shorter dead time.

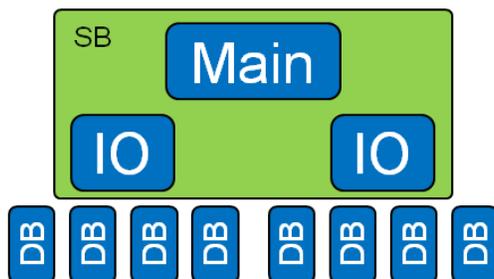


Fig. 4.21: Block diagram of a small system with the Main and IO FPGAs on the support board.

Data in the FPGAs is stored in a queue where the queue depth equals twice the number of inputs. For an IO FPGA, the queue depth is 8 (2×4 inputs), and for the Main FPGA, the queue depth is 4 (2×2 inputs). The code is generic and reusable for both FPGAs. As shown in Fig. 4.22, there is a write pointer and a read pointer. The write pointer tags an incoming block of data and then moves onto the next empty slot to tag the next incoming block of data. On the other end, once the data is read out, the queue slot is emptied and the read pointer advances to read the next block of data. These read and write pointers are circular pointers.

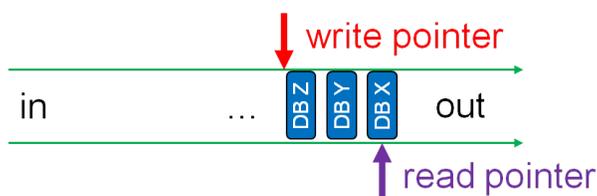


Fig. 4.22: Diagram of how data is read in the queue for scope mode.

Fig. 4.23 is the state machine diagram for the detector boards in scope mode.

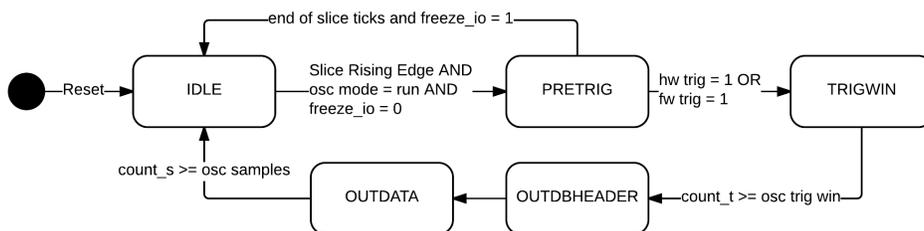


Fig. 4.23: Scope mode detector board state machine

It is initially in the IDLE stage where it is doing nothing but looking at the data coming into the system. It is not storing anything. There are three conditions that must be met for it to move onto the next stage. There must be a slice rising edge, the scope mode action must be set to RUN, and there cannot be a freeze input signal that restricts data flow. If these criteria are met, the next stage is the PRETRIG stage. Here, a user-defined number of samples before the trigger threshold are stored in the FIFO. The number of samples is set in the scope mode settings (Fig. 4.20) using bits (19:16). The FIFO can hold 256 samples, but in PRETRIG, it only stores the most recent 32 samples. Once a firmware or hardware trigger is detected on one channel, it moves on to the TRIGWIN stage. After that first trigger

is detected, there is a set period of time (trigger window) where the system stores information from other channels that subsequently triggered. Again, this window is set in the scope mode settings using bits (27:24). Once this time window has lapsed, the information is outputted. The detector board header is read out first followed by the first channel header and all of its data. There is a counter (count_s) that increments until the number of ADC samples is reached so that it knows when all the data has been outputted. Once all the data has been read out, the next channel header is outputted with all of its data and so on until the final channel is read. After all the channels have been read, the detector board returns to the IDLE stage.

If there is no trigger in the PRETRIG stage and the freeze input is flipped to one, the detector board goes back to IDLE. The “freeze” command occurs when there is lack of memory space somewhere along the data flow. It can also occur when an external issue causes a disruption in data flow (e.g., unplugged cable).

The entire system freezes by default starting with the source of the error and flowing to the bottom of the hierarchy (parent to child). For example, looking at a small system (Fig. 4.21), if the Main FPGA runs out of memory, it will send a freeze command to both its IOs which in turn will freeze all of their detector boards. If something happens to the Host PC, it will send a freeze command down to the QuickUSB which sends it to the Main FPGA which will give it to all the IO FPGAs which freezes all the detector boards.

As mentioned earlier, if a freeze command is given, all detector boards are frozen by default. However, there is another option that freezes only select detector boards and lets the rest continue to send data.

Singles Mode

In Singles mode, processed ADC data is received from the system. There are two ‘slice’ widths supported, 128-bit words and 256-bit words. Like Scope mode, it sends 32 bits at a time using DDR where each 32-bit packet has a 4-bit packet ID. There are customizable pipeline stages depending on the processing needs (user or algorithm defined). User-defined cores are isolated from the Singles top level core.

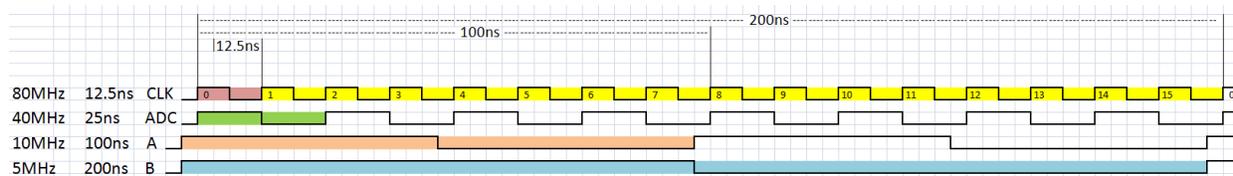


Fig. 4.24: Clock-Slice relationships. Slices A and B are 3.125ns early. In A, we can handle 4 ADC samples (12-bits each). In B, we can handle 8 ADC samples.

The data file is a binary file depicted in Fig. 4.25. It has similar characteristics to the scope mode data file. There are headers prepended to the singles data including time of acquisition and all acquisition settings. It is a simple 32-bit format for easy parsing and manipulation. Again, there is 4KB reserved for user-defined content.

Each individual piece of data recorded or event is either a 128-bit or 256-bit word. This word is broken up into N number of 32-bit packets as show in Fig. 4.26. As mentioned above, each packet has a 4-bit packet ID.

Below is a table showing the differences between the two slice choices or word sizes. It also shows how to calculate the number of packets N and the total bits transferred per slice.

- $R = \text{System Clock Period} / \text{ADC Clock Period}$
- $N = R * \text{Slice Period} / \text{System Clock Period}$

Offset	[31..24]	[23..16]	[15..8]	[7..0]	Comment
0	Epoch timestamp				Local Time
1	Reserved				
2	Acquisition Duration				From Software
3	Acquisition Mode Payload				See cmd id 0x0003
4	Acquisition Mode Settings Payload				See cmd id 0x0005
5	Reserved				future data format
6	Misc.Software Settings				processed/raw?
7	Reserved				
...					
9					
10	User Defined				
...					
999					
B=1000	Singles Packet (0)				Event can originate
B+1	Singles Packet (1)				from any DB.
	...				
B+N-1	Singles Packet (B+N-1)				N = Slice Width
B+N	Singles Packet (0)				Event can originate
B+N+1	Singles Packet (1)				from any DB.
	...				

Fig. 4.25: Binary file format in singles mode

#	32-bit wide word [31:0]
0	Packet 0
1	Packet 1
...	...
N-1	Packet N-1

Fig. 4.26: Diagram of a singles mode word. There are N number of 32-bit packets where N starts at 0.

Characteristics	Slice Choice 1	Slice Choice 2
Default System Clock Period	12.5ns (80MHz)	12.5ns (80MHz)
Default ADC Clock Period	25ns (40MHz)	25ns (40MHz)
Slice Period	100ns (10MHz)	200ns (5MHz)
R	$12.5/25 = 1/2$	$12.5/25 = 1/2$
N (Number of Packets)	$1/2 * 100/12.5 = 4$ packets	$1/2 * 200/12.5 = 8$ packets
Total Bits Transferred per Slice	$4*32 = 128$ bits	$8*32 = 256$ bits

As in scope mode, the Packet ID is used to verify the type of packet received at a parent. It also allows a node to identify, validate, and semi-confirm the integrity of the packet in the data path. The table below lists current IDs.

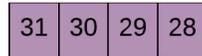


Fig. 4.27: Packet ID for singles mode. It is the four most significant bits in a 32-bit packet.

ID	Description
0x0	Reserved. Do NOT use
0x1	Singles Packet
0x2-F	Not used

Data received in singles mode uses arbitration to store it. This arbitration can be examined through a small system example. For a single chassis, arbitration happens in the support board. For an IO FPGA, there are four inputs, one from each detector board, and one output to the Main FPGA. For the Main FPGA, there are two inputs, one from each IO FPGA, and one output to the Host PC (Fig. 4.21). In this method, there is loss of data. The code on the Main and IO FPGAs is reusable.

The arbitrator uses a true random number generator to select one incoming event. No priority is given to any input slot. It works for all cases (e.g., 1-in-1-out, 2-in-1-out, 3-in-1-out, 4-in-1-out). Only one clock cycle is required for selection.

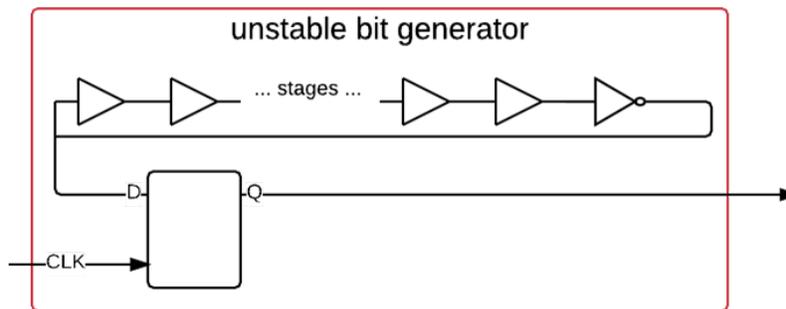


Fig. 4.28: Unstable bit generator used in the random selector

Fig. 4.28 is a diagram of the unstable bit generator used in the random selector (Fig. 4.29) for the arbitrator. It uses a programmable number of ring oscillator stages, and randomness is created from voltage, temperature, and near-by-logic variations. The output bit Q is unstable by design.

In the random selector, there are four instances of the unstable bit generator where each instance is a different stage. The outputs of the unstable bit generators are channelled through an XOR gate to add more randomness. The XOR output enables or disables a free running counter meaning if the XOR output is one, the counter increments, and if the XOR output is zero, the counter does not increment. In Fig. 4.29, the trapezoid labeled 'enable' is a multiplexor with a maximum of four inputs.

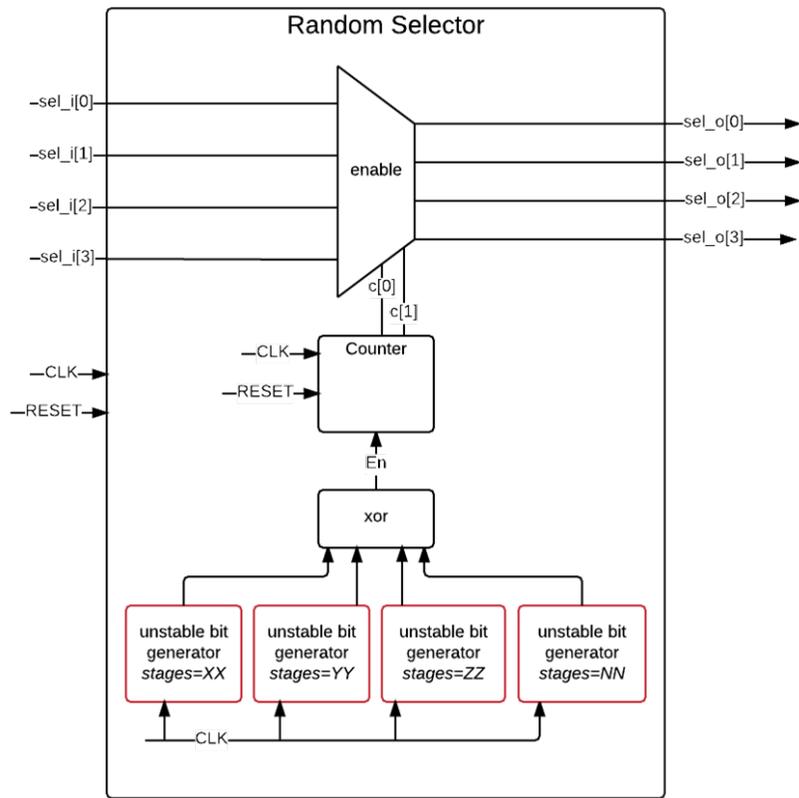


Fig. 4.29: True random number generator and arbitrator for singles mode

The output from the counter determines which input in the multiplexor is selected for output. For example, if there are four inputs to the multiplexor, the counter will select one of those four. If there are only two, the counter will select one of those two. More specifically, the counter's maximum value is the least common multiple of the number of multiplexor inputs. The counter output is the modulo of the counter value and the number of inputs going into the multiplexor. For example, if there are four inputs, the LCM is 12 so this is the maximum value the counter can reach. If the counter output value is 11, the modulo of 4 and 11 is 3 which corresponds to selecting the fourth input (the inputs are numbered 0 to 3). This whole system is designed to create a fair selection of the inputs.

The following is a minimalistic example of a detector board function and design in singles mode. The detector board can have user-defined cores composed of modules that manipulates the data for a desired output. Fig. 4.30 provides an example module showing default inputs and outputs. The thick I/O markers are buses, and the thin I/O markers are single bits. More can be added. The detector board interface intelligently instantiates the correct number of user-defined cores based on the number of pipeline stages available. The data is handled by the interface without any user intervention.

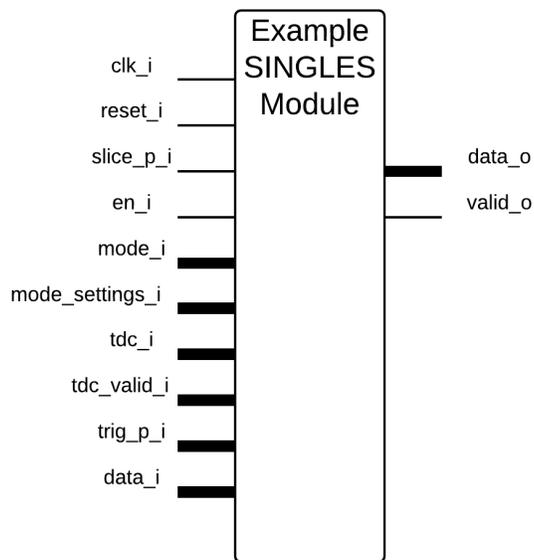


Fig. 4.30: Singles mode module example

Fig. 4.31 shows the state machine diagram for the example module above.

The detector board will enable each module within it sequentially to allow processing. Each module is initially in the IDLE stage where it waits for the conditions to move onto the PRETRIG stage. It must receive an enable signal from the detector board (en_i from Fig. 4.30) and the clock and slice must be '1' or the high value simultaneously. Once in the PRETRIG stage, it waits for a triggered channel signal to move onto the PROCESS/OUTPUT stage. If there is no trigger received within the time set by the user in the singles mode settings (Fig. 4.33), it returns to IDLE. Also, once it is in the PROCESS/OUTPUT stage, it will return to IDLE after that same amount of user-defined time.

The following is a basic OpenPET example that computes the energy or area under the curve of the waveform. There is up to 16 data points utilizing 5 pipeline stages. Fig. 4.32 is an example of one of the word packs.

- 128-bit Word packs:
 - 6-bit energies for 16 channels = 6*16 bits
 - 9-bit address
 - 4-bit hit trigger counter

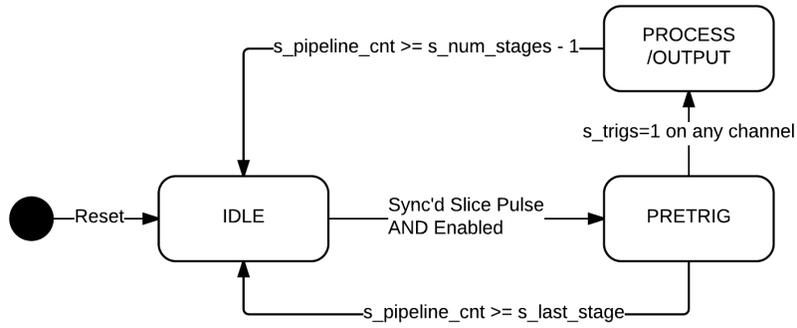


Fig. 4.31: Singles mode module state machine example

- 4-bit packet ID for 4 packets = 4*4 bits

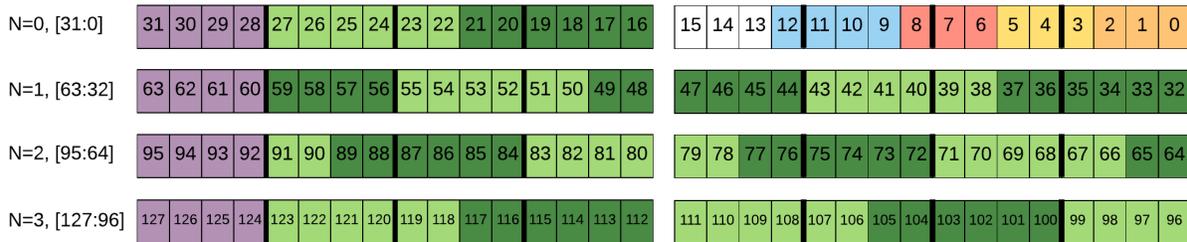


Fig. 4.32: Singles mode energy example word

For N = 0:

```

Starting from the least significant bit (LSB)
(2:0)  Detector board address (populated by parent)
(5:3)  DU address (populated by parent)
(8:6)  MB address (populated by parent)
(12:9) Number of channels that triggered
(15:13) Not used
(21:16) Channel 0 energy
(27:22) Channel 1 energy
(31:28) Packet ID
    
```

As previously mentioned, the word is broken up into N number of 32-bit packets. For the rest of packets after N = 0, (27:0) contains channel energy and (31:28) is the packet ID. Each channel energy is 6 bits long as illustrated by the alternating green sections in Fig. 4.32 for N > 0. Also, some channels are split over different packets.

Fig. 4.33 shows the bit assignment for the singles mode settings for this example. This is set by *Command ID: 0x0005* (page 48).

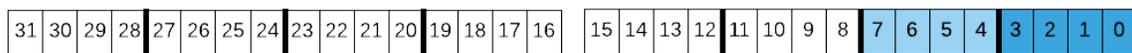


Fig. 4.33: Singles mode settings for this energy example

```
Starting from the least significant bit (LSB)
(3:0) Total number of ADC clock ticks to finish a single Event computation ( $2^4 = 16$ )
(7:4) Reserved (max ADC clock ticks to process data:  $2^8 = 256$ )
(15:8) Reserved
(31:16) Not used
```

Note:

- The event computation clock ticks should be greater than 1.
- The number of pipeline stages is predefined in the firmware as a constant.
- Pipeline Stages = $\text{ceil}(\text{Event Computation Clock Ticks}/\text{Slice Width}) + 1$, where $\text{ceil}()$ rounds the number to the next highest integer.

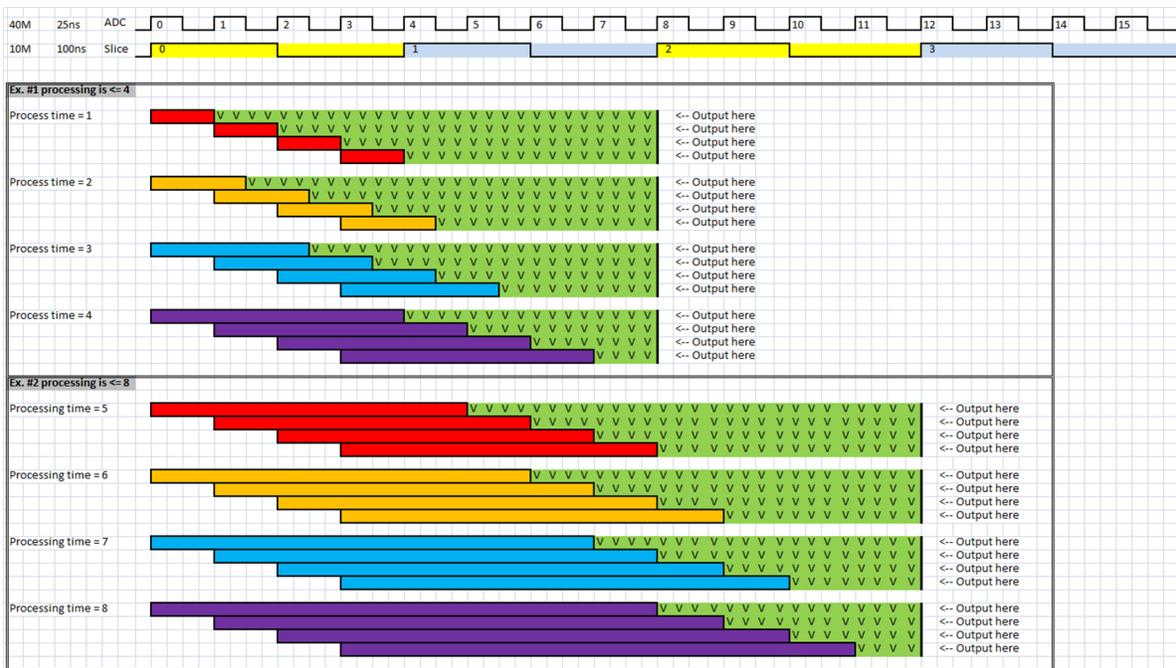


Fig. 4.34: Shows multiple examples of different processing ticks.

Since coincidence is important, OpenPET has a system of outputting data so that events that trigger in the same slice are processed and outputted in the same slice. In Fig. 4.34, there are eight examples divided into two sections. The top section shows when the processing time is 1 through 4 clock ticks (ADC domain), and the bottom section is for when the processing time is 5 through 8 clock ticks. The processing time is calculated by subtracting 1 from the maximum number of pipeline stages and multiplying the difference by N.

The timing of `valid_o`, `data_o`, and `slice_i` is shown in Fig. 4.35. In this simple example, the `valid_o` encapsulates `data_o`. The length of `valid_o` depends on the slice width and algorithm implemented. The MSB of `data_o`'s first byte indicates that it is a Singles byte. The address is zero because the address gets populated by the parent i.e., the IO FPGA.

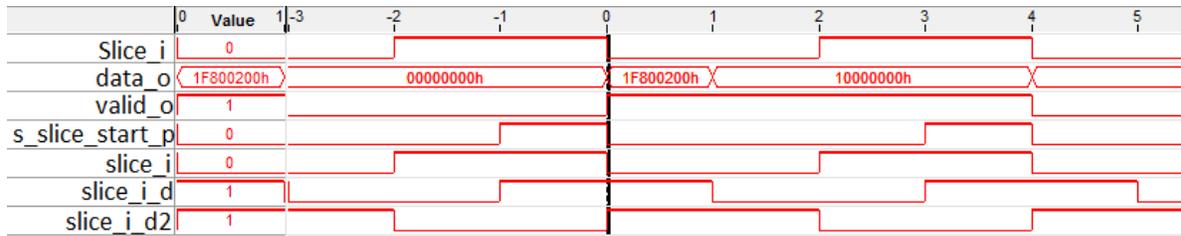


Fig. 4.35: Relationship between valid_o and data_o at the DB level.

Serial Peripheral Interface

OpenPET utilizes a standard 32-bit wide Serial Peripheral Interface (SPI) to facilitate serial communications. The SPI structure is a master-slave architecture to connect the parent node to its children. Fig. 4.36 and Fig. 4.37 show the configurations of the master and slave SPIs. In the OpenPET system, the master SPI is on either the CUC or CDUC depending on the size of the system.

Currently, the master SPI is configured to have ten slaves - two for the IO FPGAs and eight for either the DUCs or DBs. For example, if the system is small, the master SPI would be on the CDUC with two slaves for the IO FPGAs and eight slaves, one for each detector board. However, for a standard system, the master SPI is on the CUC with two slaves for the IO FPGAs and eight slaves, one for each DUC.

The DUC Support Board is a special case having a slave SPI as well as a master. Commands from a CUC (master SPI) flow to the slave SPI on the DUC. If the command destination is a detector board, the command then continues to the master SPI on the DUC to the slave SPI on the detector board. The master-to-slave SPI transition on the DUC is invisible to the user. The detector boards only have slave SPIs.

Fig. 4.38 gives an example waveform of a 32-bit SPI with broadcasting.

Fig. 4.39 shows the logic each SPI follows when receiving a command.

A command originates from a source, usually the Host PC, and consists of the command ID, source address, destination address, and the payload. The commands are constructed using the little-endian format. The master SPI on the parent node (e.g. CUC or CDUC) receives the command from the Host PC. It then processes the command and determines if the broadcasting flag is set to 1. If so, it will pass the command down to all its children and the response it receives will be from the child specified in the destination address. Additionally, if the CUC or CDUC flag is set to 1, the parent will execute the command itself and the reply will be from the parent instead of the child specified in the destination address.

After it determines the status of the broadcasting flag and performs any necessary actions, it will check the destination of the command. As mentioned in the previous paragraph, if itself is the destination, it will execute the command and reply to the source. If it is not the destination, it will write the command to the specified destination (if the broadcasting flag was not set to 1) and determine if the command is asynchronous. If it is, the C/R bit is set to 1, and the parent does not keep polling for a response. It simply sends the command. If the command is not asynchronous, the parent keeps reading until it receives a valid response from the child and performs the corresponding action.

When a command is passed from a parent node to a child node, it follows the steps listed in Fig. 4.40.

When reading a response from the destination, it follows Fig. 4.41.

Example

The following is an example of SPI communications between nodes. In this case, the parent node is the Host PC, the child node is the CUC or CDUC, and the grand child is the detector board.

1. The Host PC constructs a command:

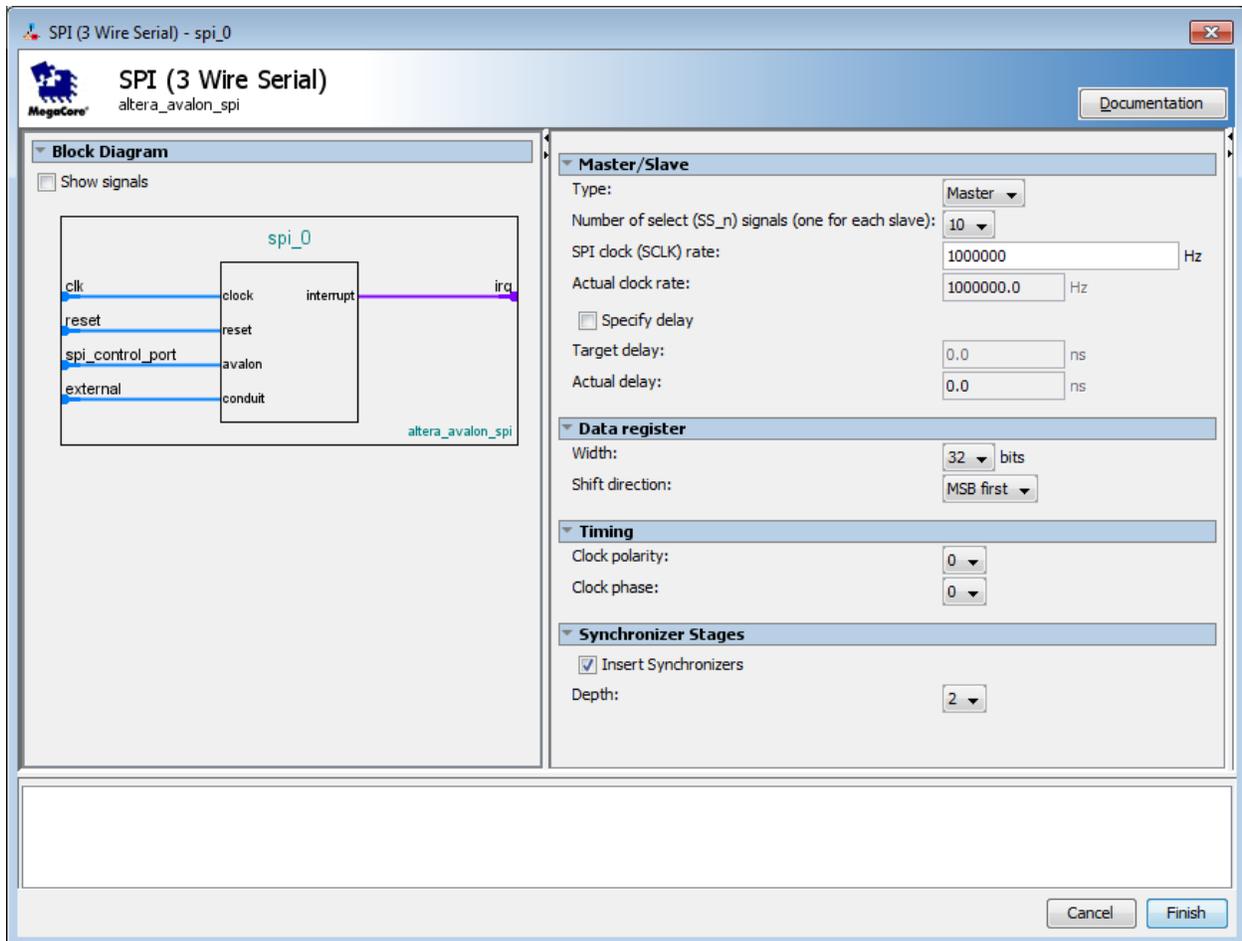


Fig. 4.36: Block diagram and specifications for the master SPI.

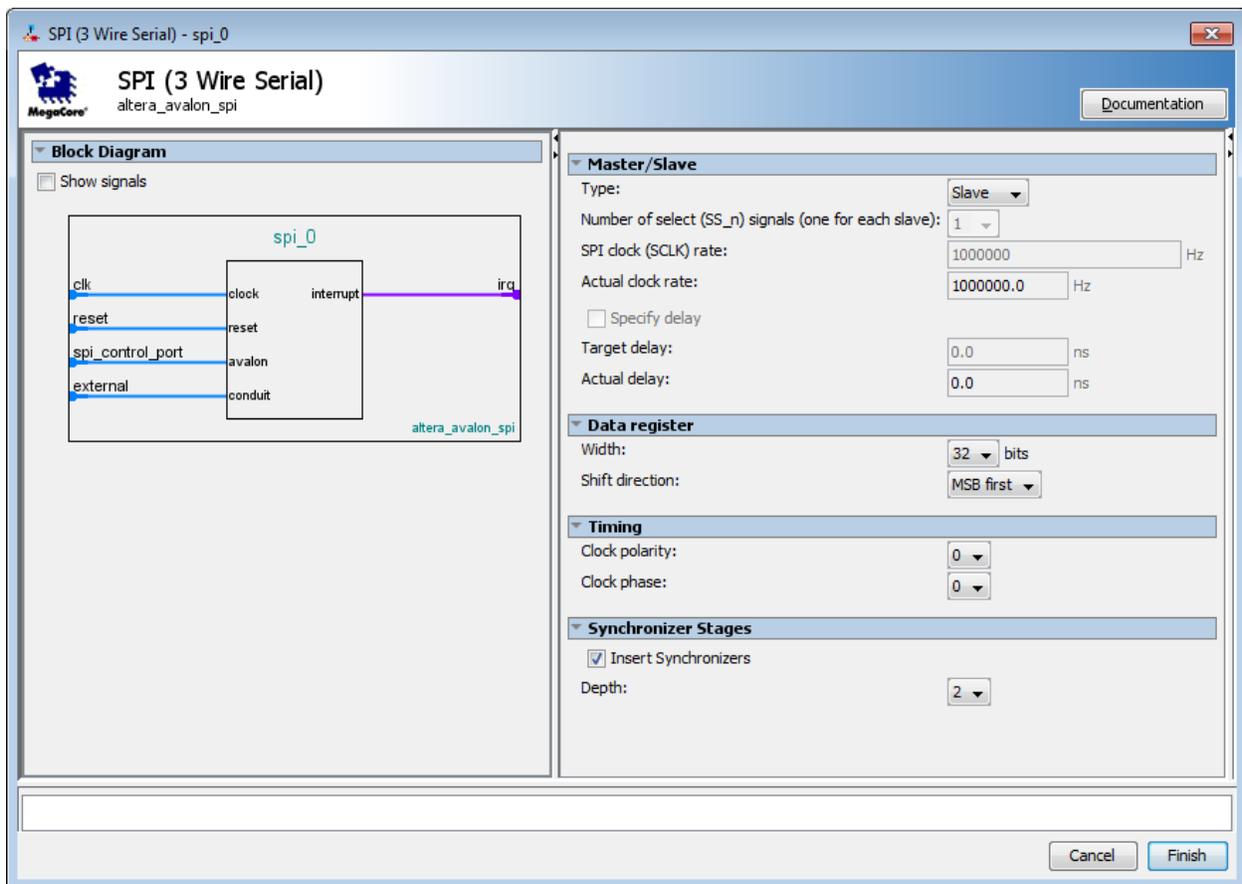


Fig. 4.37: Block diagram and specifications for the slave SPI.

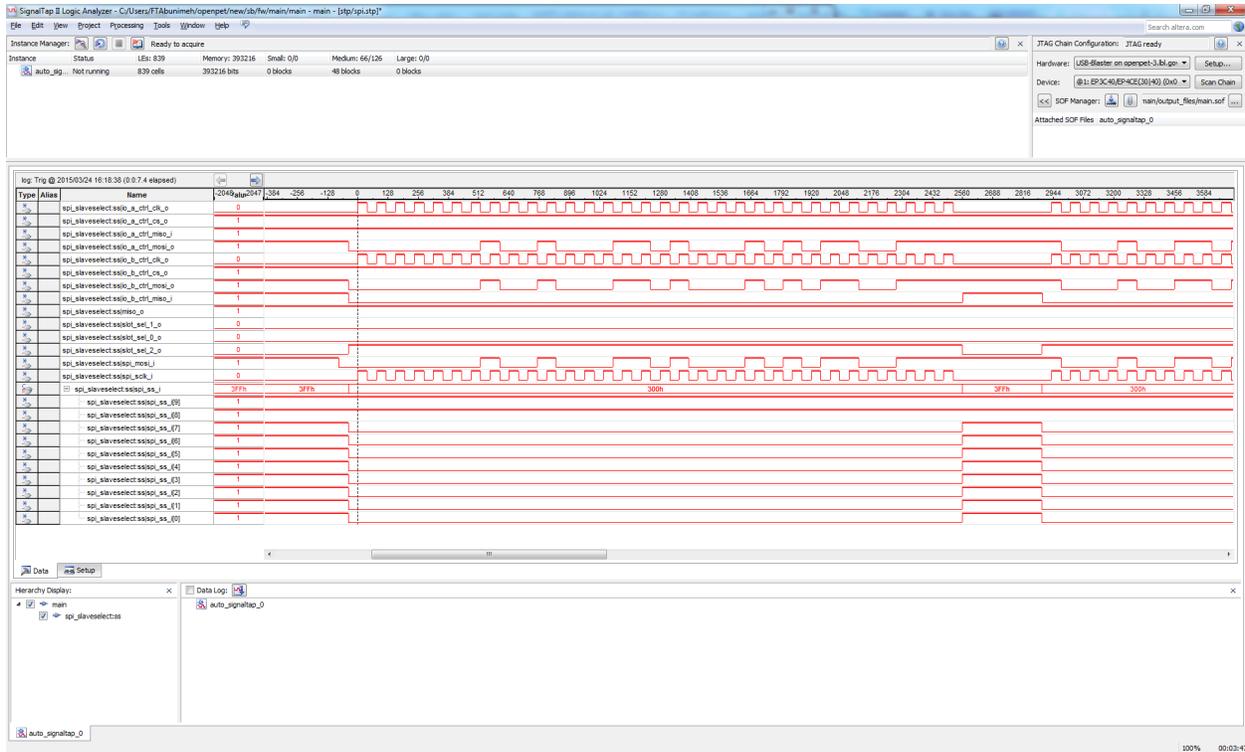


Fig. 4.38: Example waveform of a standard 32-bit SPI with the broadcasting flag set to 1.

cmd = 1 (PING)

src = 0x4000 - Host PC address

dst = 0x0003 - DB in slot 3

payload = 0x0000ABCD

2. Using QUSB, the Host PC writes that command to the first node, e.g., CUC or CDUC. It can communicate through any medium (ethernet, USB, fiber optic).
3. The Host PC will keep reading for a valid response from the child until the number of retries (“CMD_READ_RETRIES”) is reached. Between each trial, a time period of “CMD_READ_RETRIES_TIME” is observed. CMD_READ_RETRIES and CMD_READ_RETRIES_TIME are constants defined in the software running on the Host PC.
4. There are five possible responses:
 1. If the command ID is equal to the command ID sent, i.e. C/R flag is still 0 and not 1, then the child is out of memory. (when not in async/nonblocking calls)
 2. If the command ID is equal to “CMD_STDCMD_UNKNOWN”, then the child doesn’t understand this command.
 3. If the command ID is equal to “CMD_STDCMD_TIMEDOUT”, then the grandchild timed out.
 4. If the command ID is equal to “CMD_STDCMD_DEADCHILD”, then the grandchild is dead or doesn’t exist. Also, if the Host PC number of retries >= CMD_READ_RETRIES, then the child is presumed dead.
 5. If the command ID is equal to “CMD_STDCMD_BUSYCHILD”, then the grandchild is busy executing the previous command.
5. Once a child (CUC or CDUC) receives a command from HostPC, the following can occur:

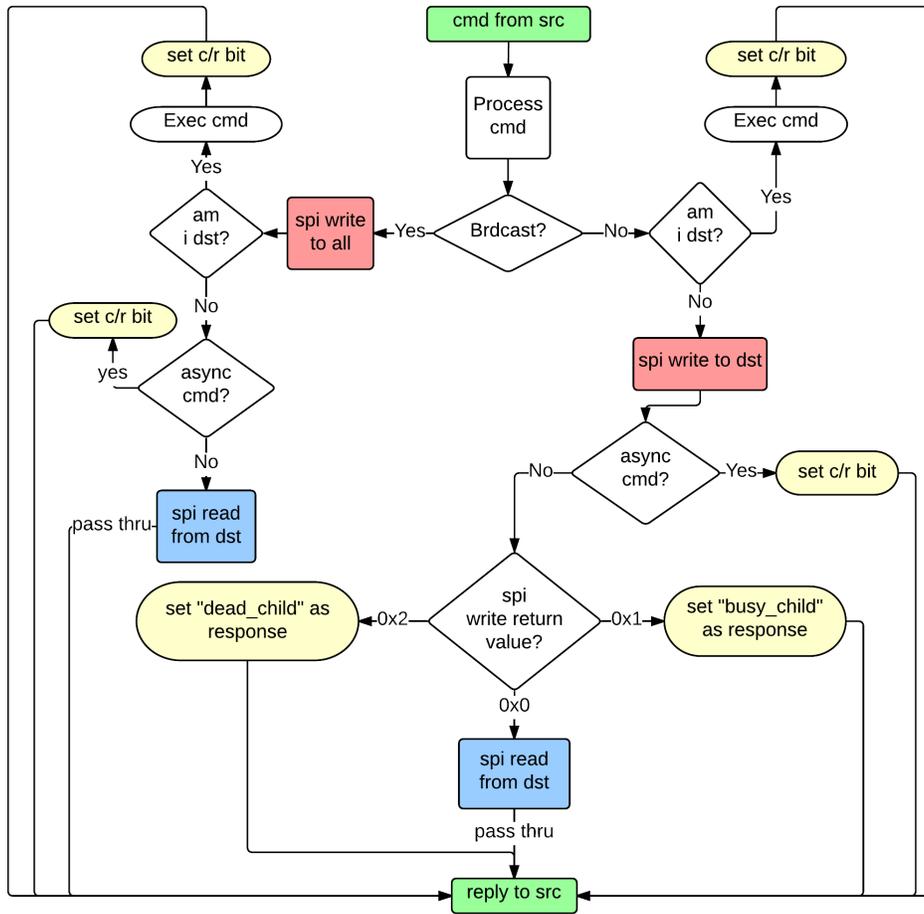


Fig. 4.39: Flowchart depicting the command flow logic

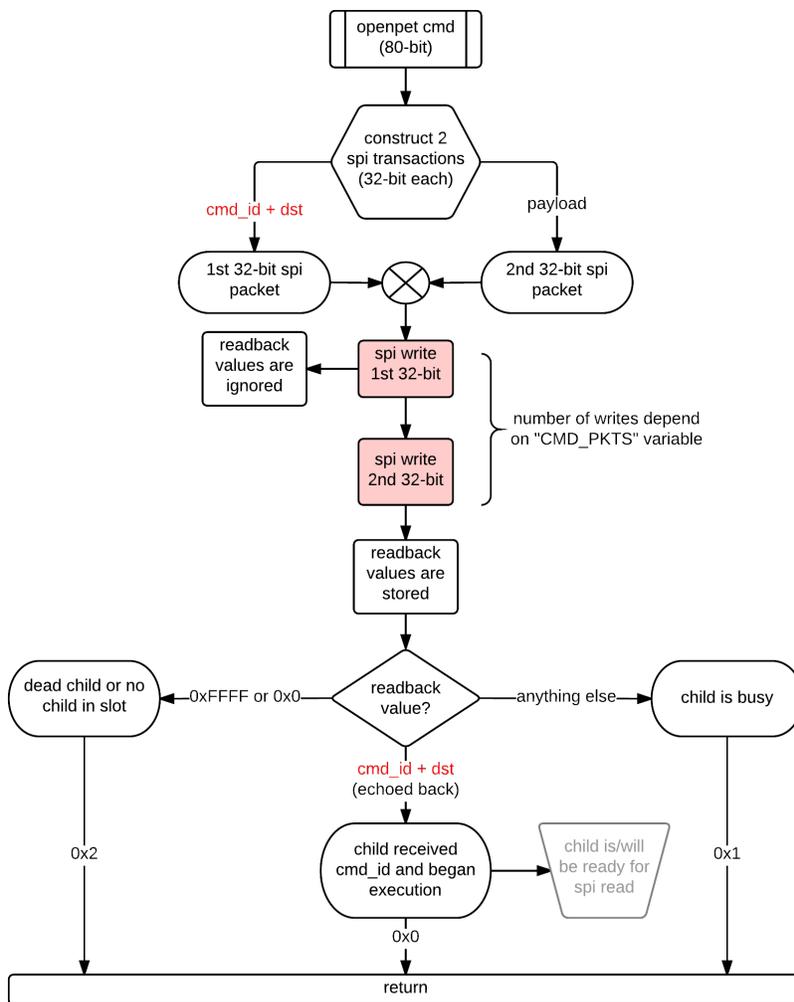


Fig. 4.40: SPI write command flowchart

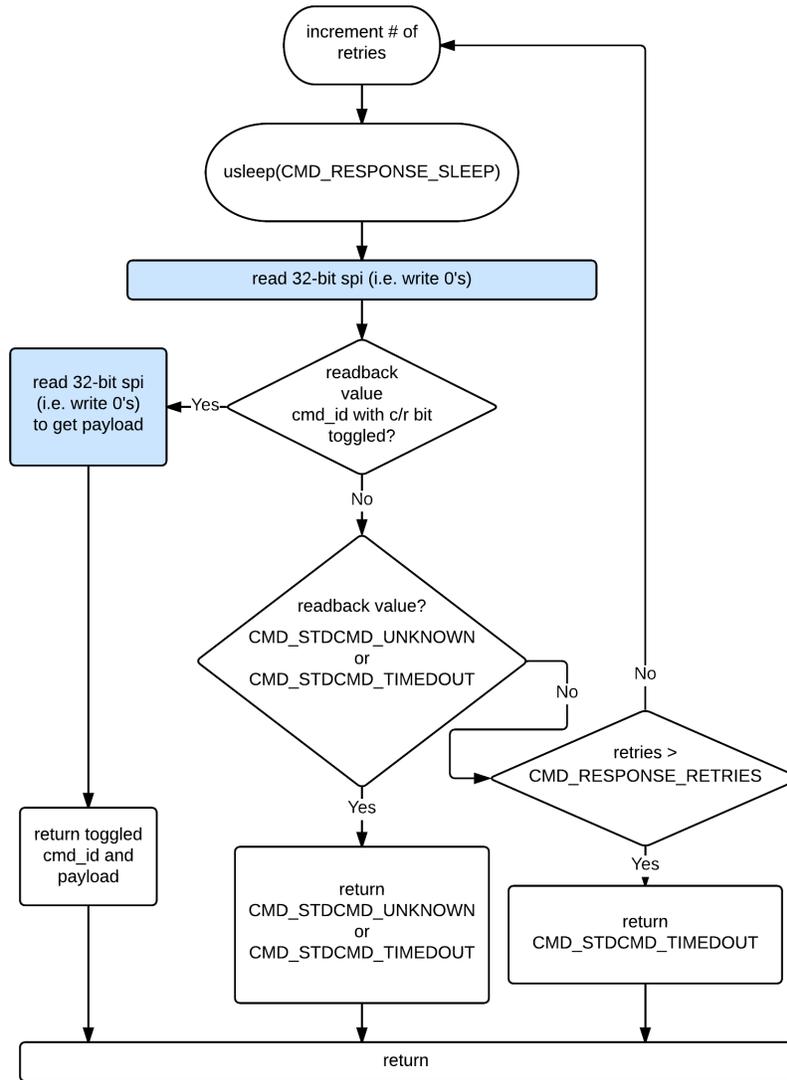


Fig. 4.41: SPI read command flowchart

1. If broadcast flag is set, the parent will pass the command down to all its children. The response will be from the child specified in destination address.
2. If the command destination is the CUC or CDUC (itself), it will be executed locally and returned with the proper response flag and payload. The reply will be from that unit and not from the child specified in destination address.
3. If the command is passed down (WRITE) to the grandchildren, the commands are modified into two 32-bit SPI transactions. The first transaction is 32-bits and contains the "C/R ID" (16 bits) + "DST address" (16 bits). The second contains the payload.
1. If grand child doesn't respond to WRITE, it doesn't exist or is dead, i.e. CMD_STDCMD_DEADCHILD is returned to parent.
2. Child will be polling (READ) grandchild for "CMD_RESPONSE_RETRIES" times with CMD_RESPONSE_SLEEP time period in between. CMD_RESPONSE_RETRIES and CMD_RESPONSE_SLEEP are defined in child's embedded software i.e. cmd.h
3. If the number of retries >= CMD_RESPONSE_RETRIES, then CMD_STDCMD_TIMEDOUT is sent back to parent. (as described in 4. C)
4. (1st 32-bit SPI transaction) If child receives CMD_STDCMD_TIMEDOUT or CMD_STDCMD_UNKNOWN from a grandchild, it will just pass them back to parent. There will be NO second transaction.
5. If child receives correct response with proper response flag set, then the child performs one more transaction to READ the payload from the grandchild. Once the payload is received, a response to parent will be constructed and ready to be READ/pollled by parent.
6. If while WRITING to a grandchild, the grandchild is still performing a task, the child will send back CMD_STDCMD_BUSYCHILD to parent. Because the child will be expecting the same command echoed back, if a different command is echoed back, then the grandchild is still working on the previous command.

Fig. 4.42 shows the debugging screen of sending a ping command (ID = 1) to a detector board. The first line beginning with [0] shows the command received which is broken down into its components in the next four lines. The command source is 0x4000 which is the Host PC, and the destination is Detector Board 3. The payload for this command is meaningless since it is merely a 'ping'. The slave SPI is then selected. The WRITING value (first [SPI] line) breaks down into 0x00010003 which is the command ID combined with the destination address and the payload 0x0000ABCD. Each is a 32-bit SPI transaction. Once the correct CMD ID and destination are read back ([SPI] line three), the command is executed. The master SPI then reads from the slave for the CMD ID and destination with the C/R bit set to 1 ([SPI] line four). After that, it reads the payload back. At this point, the command is complete, and a response is generated and sent back to the Host PC (final line).

The following screen shot (Fig. 4.43) shows the result of unsuccessfully sending a command to a detector board. Since the write-readback value is 0xF, there is a broken or non-existent detector board in that slot. Therefore, the corresponding response is generated (last line).

List Mode Data

(Use combination of User Guide section 2.5.1 and Framework section 3.)

Introduction

(Expand on Framework section 3.1.)

```

/cygdrive/c/altera/13.1
openpet@openpet-2 /cygdrive/c/altera/13.1
$ nios2-terminal.exe
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-0]", device 1, instance 0
nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate)

[0] Command Received [ 0xABCD 0x0000 0x0003 0x4000 0x0001 ]
ID = 0x0001
SRC = 0x4000
DST = 0x0003
PLD = 0x0000ABCD
This will be sent to [slave=0x00000008]
[SPI][slave=0x00000008] WRITING [ 0x00010003 0x0000ABCD ]
[SPI] write-readback [ 0x00000999 ]
[SPI] write-readback [ 0x00010003 ]
[SPI][slave=0x00000008] READING [ 0x80010003 ]
[SPI][slave=0x00000008] READING [ 0x0000ABCD ]
done with cmd
Responding back with [ 0xABCD 0x0000 0x4000 0x0003 0x8001 ]

```

Fig. 4.42: Example of a command successfully executed with Detector Board 3.

```

/cygdrive/c/altera/13.1
openpet@openpet-2 /cygdrive/c/altera/13.1
$ nios2-terminal.exe
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-0]", device 1, instance 0
nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate)

[0] Command Received [ 0xABCD 0x0000 0x0005 0x4000 0x0001 ]
ID = 0x0001
SRC = 0x4000
DST = 0x0005
PLD = 0x0000ABCD
This will be sent to [slave=0x00000020]
[SPI][slave=0x00000020] WRITING [ 0x00010005 0x0000ABCD ]
[SPI] write-readback [ 0xFFFFFFFF ]
[SPI] write-readback [ 0xFFFFFFFF ]
got junk
done with cmd
Responding back with [ 0xABCD 0x0000 0x4000 0x4000 0x7F02 ]

```

Fig. 4.43: Example of a command sent to a non-existent detector board.

Event Words

(Include single event mode and coincidence event mode. See Framework section 3.2.)

Status Words

(See Framework section 3.3.)

Coding and Decoding List Mode Data(?)

(See Framework section 3.4.)

Bootup Sequence

This bootup sequence describes the steps of the OpenPET system after the user has programmed the EPCS. It begins when the system powers on and ends with the detector boards waiting for the first SPI command. The basic steps can be explained through a small system example.

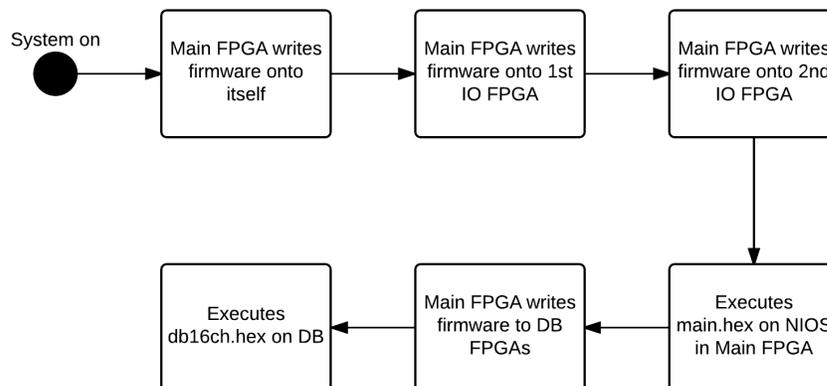


Fig. 4.44: Diagram of the bootup sequence for a small OpenPET system

In a small system, there is a single support board that connects to all the detector boards (Fig. 4.21). The basic sequence of steps is outlined in Fig. 4.44. On the support board is the EPCS chip that is used to configure all the FPGAs on the support board and detector boards. The table below and Fig. 4.45 show the structure of the EPCS. The table outlines the basic block locations. The blocks are further explained in Fig. 4.45.

Table 4.3: EPCS addressing

Block	Start Address	End Address
Page_0	0x00000000	0x00368E43
main.hex	0x00368E44	0x0037A13B

Note:

- The detector board images are deliberately written to a different location (0x00400000)

- All the addresses in this file are byte addresses

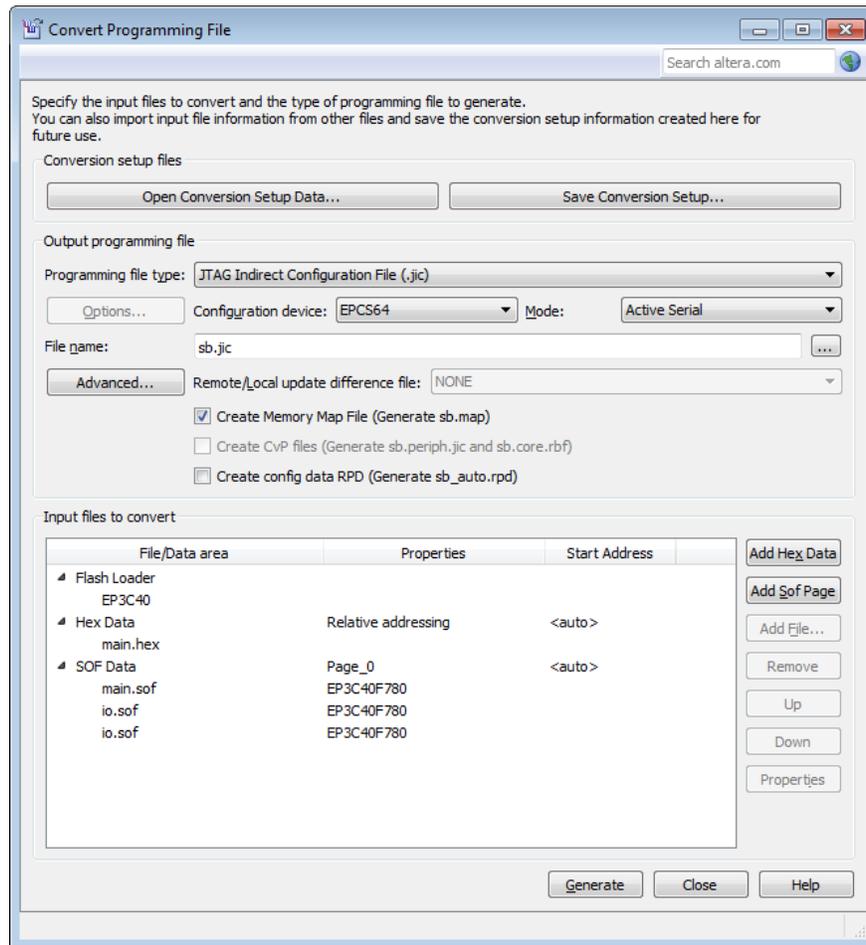


Fig. 4.45: EPCS hierarchy

Using the Altera boot loader, the Main FPGA configures itself first with the firmware from the EPCS. Next, it configures the IO FPGAs one at a time with different but identical firmware. In other words, there are two IO FPGA firmware blocks on the EPCS, and each gets loaded onto an IO FPGA (see Fig. 4.45). The IO sof contains the bitstream and an elf executable. Once the firmware is loaded onto the IO FPGA, the elf program is run. This program initializes and resets the IO FPGA and then idles as it waits for an SPI command from the user.

After the Main FPGA loads the firmware onto the IO FPGAs, it copies the main.hex program over to itself and executes it on NIOS. This program does multiple things. It initializes the Ethernet, initializes the current node (CDUC in this case), performs a soft reset, and then programs the detector boards. To program the detector boards, the Main FPGA uses what is called Active Serial Configuration which means the FPGA is the master device and the EPCS is the slave device. The Main FPGA accesses the EPCS to get the detector board images (see note above) and then programs all the detector boards simultaneously. The sof of the detector board includes the bit stream with an elf executable. The detector board FPGA runs the elf program which performs a reset and then waits for an SPI command. At this point the system is waiting for the user to tell it to do something. This concludes the bootup sequence.

System Reset

There are three different types of system resets possible.

The first is a hard reset that resets the entire system. This is the top level reset method. There is a button on either the support board or the Host PC board that will shut down any and all FPGAs running.

The second method is a firmware reset. This keeps the FPGAs running but essentially shuts down the system clock signal so the system still cannot do anything. It unlocks the PLL so that no clock signal can be outputted. When the PLL is locked, a clean system clock signal can be transmitted throughout the system. However, if the PLL is not locked, then no system clock signal can be sent through the system.

The third reset is the soft reset mentioned in the Bootup Sequence above. In this case, there is no physical button to press, and the PLLs are locked. This reset affects the QuickUSB and any firmware or software communication blocks. It takes about 10ms on bootup.

Introduction

(Describe source code modules: names, functions, how they tie in with previous higher level description shown in Fig. 2.7, etc.)

Firmware and Software

OpenPET Library

This section describes the python OpenPET library (OpenPETlib.py) used in the openpet executable. The library file can be found in the hostpc repository under the software directory.

Variables

- IMP_NETMAP - If netmap exists, set IMP_NETMAP to true. If not, set IMP_NETMAP to false.

Table 5.1: System Variables

Variable Name	Value	Description
QUSB_FD_WIDTH	16	Quick USB data bus width
QUSB_ADR_WIDTH	9	Quick USB address bus width
QUSB_CMD_PKTS	5	Number of Quick USB packets (write cycles) to complete a single OpenPET command. Current OpenPET command length is 80 bits so $5 \times 16 = 80$
UDP_WIDTH	8	
UDP_CMD_PKTS	10	
BITS_IN_BYTE	8	Number of bits in one byte (for convenience)
SRC_ADDR	0x4000	Workstation address such as the Host PC
QUSB_CMD_LEN		$QUSB_FD_WIDTH * QUSB_CMD_PKTS / BITS_IN_BYTE$ Command length in bytes
UDP_CMD_LEN		$UDP_WIDTH * UDP_CMD_PKTS / BITS_IN_BYTE$ Command length in bytes
UDP_CMD_LEN_OS		UDP_CMD_LEN unless $UDP_CMD_LEN < 46$. In which case, the value is 46 to prevent Windows error
UDP_DAT_LEN	1440	If IMP_NETMAP is true, then it equals 8945
PRINT_ACQ_PROG	200	Number of clock ticks between printing system status

Table 5.2: Command responses

Variable Name	Value	Description
CMD_STDCMD_UNKNOWN	0x7F00	Software (running on NIOS) command id is unknown to node.
CMD_STDCMD_TIMEDOUT	0x7F01	Software (running on NIOS) command id has timed out.
CMD_STDCMD_DEADCHILD	0x7F02	Targeted child is dead, nonexistent, or not programmed.
CMD_STDCMD_BUSYCHILD	0x7F03	Targeted child is busy processing previous command.
CMD_FWCMD_UNKNOWN	0x7F04	Firmware (running on FPGA fabric) command id is unknown to node.
CMD_FWCMD_TIMEDOUT	0x7F05	Firmware (running on FPGA fabric) command id has timed out.
CMD_STDCMD_INCOMP	0x7F06	Software (running on NIOS) received incomplete OpenPET command
CMD_STDCMD_FAST_USER	0x7F07	Software (running on NIOS) received packets faster than it can handle.
CMD_STDCMD_RESPONSE_MSK	0x8000	Command response mask

Table 5.3: Data IDs

Variable Name	Value	Description
DB_HDR_ID	0x40000000	Packet ID value in the DB header to verify that it is the DB header.
CH_HDR_ID	0x30000000	Packet ID value in the channel header to verify that it is the channel header.
DT_PKT_ID	0x10000000	Packet ID value in the data packet to verify that it is a data packet.
PKT_ID_MSK	0xF0000000	Packet ID mask
DHEADERLEN	1000	Data header length (words)

Table 5.4: QuickUSB variables

Variable Name	Value	Description
QUSB_major	2	QuickUSB driver major version number
QUSB_minor	15	QuickUSB driver minor version number
QUSB_rev	2	QuickUSB driver revision number
QUSB_TIMEOUT	50	QuickUSB timeout in milliseconds
DATABUFS	8	QuickUSB default buffer size
DATABUFSIZE	2x512x1024	Maximum buffer size

There is a final QuickUSB variable that sets the QuickUSB settings. It is a 2D array that is broken down below.

QUSBSETTINGS:

- [1, 0x0001] - Sets address 1 to 00000001.
 - Bit 7-1: Reserved
 - Bit 0: Sets data word width to 16 bits.
- [2, 0xC000] - Sets address 2 to 1100000000000000.
 - Bit 15: Disables incrementing the address bus.
 - Bit 14: Disables address bus
 - Bit 13-9: Unused
 - Bit 8-0: HSPP address value
- [3, 0x0002] - Sets address 3 to 0000000000000010.
 - Bit 15-14: Unused R/O
 - Bit 13: Sets FIFO packet end polarity to active low
 - Bit 12: Sets FIFO output enable polarity to active low
 - Bit 11: Sets FIFO read polarity to active low
 - Bit 10: Sets FIFO write polarity to active low
 - Bit 9: Sets FIFO empty flag polarity to active low
 - Bit 8: Sets FIFO full flag polarity to active low

- Bit 7: Sets IFCLK source to external clock
- Bit 6: Sets IFCLK speed to 30MHz
- Bit 5: Tri-state the IFCLK pin
- Bit 4: Sets IFCLK polarity to normal
- Bit 3: Sets GPIF clock mode to synchronous GPIF
- Bit 2: Reserved
- Bit 1-0: Sets HSPP configuration to GPIF master mode
- [5, 0x8010] - Sets address 5 to 1000000000010000.
 - Bit 15: Sets USB bus speed to allow high-speed (480Mbps)
 - Bit 14-8: Reserved
 - Bit 7-6: Unused R/O
 - Bit 5: Reserved
 - Bit 4-3: Sets CPU clock speed to 48MHz
 - Bit 2: Does not invert CLKOUT
 - Bit 1: Tri-state the CLKOUT pin

Table 5.5: SRAM commands

Variable Name	Value	Description
CMD_STDCMD_SRAM_WRITE	0x000B	Writes to external SRAM device. Auto-increments address.
CMD_STDCMD_SRAM_READ	0x000C	Reads from external SRAM device. Auto-increments address.

Table 5.6: UDP Variables

Variable Name	Value	Description
UDP_CMD_PORT	9955	
UDP_DAT_PORT	9956	
UDP_DAT_PORT_HEX		struct.pack(">H", UDP_DAT_PORT)

Table 5.7: UDP Commands

Command	Value	Description
c_UDP_CMD_ACK	0xAC	Request
c_UDP_CMD_LEN	0xBD	Payload length
c_UDP_CMD_MODE_ACQ	0xC1	Default mode
c_UDP_CMD_MODE_TST	0xC2	Test mode
c_UDP_CMD_MODE_NOACK	0xC3	Openloop test mode
c_UDP_CMD_MODE_IDL	0xC4	Openloop idle mode

Note: UDP commands are NOT 80-bit OpenPET commands

Table 5.8: Network friendly bytes

Variable Name	Description
UDP_B_REQ	struct.pack('>B',c_UDP_CMD_ACK) Construct byte for Request
UDP_B_ACQ	struct.pack('>B',c_UDP_CMD_MODE_ACQ) Construct byte for default mode
UDP_B_TST	struct.pack('>B',c_UDP_CMD_MODE_TST) Construct byte for test mode
UDP_B_LEN	struct.pack('>BH',c_UDP_CMD_LEN, UDP_DAT_LEN) Construct byte payload length
UDP_B_NOACK	struct.pack('>B',c_UDP_CMD_MODE_NOACK) Construct byte for openloop test mode
UDP_B_IDL	struct.pack('>B',c_UDP_CMD_MODE_IDL) Construct byte for openloop idle mode

Functions

- `__enter__(self):`
Class initialization function for proper QuickUSB handling.
- `__exit__(self, exc_type, exc_value, traceback):`
Class clean up function for proper QuickUSB handling.
- `__init__(self, qusbindx=0):`
Class constructor. Sets values for various settings of things such as QuickUSB, scope mode acquisition, DAC, etc.

The following table explains the variables in the OpenPET class constructor.

Variable	Description
qusb	Contains the QuickUSB object
qusb_valid	Used to check that QuickUSB is valid
ofilename	Default file name
Continued on next page	

Table 5.9 – continued from previous page

Variable	Description
ofile	Data output file
cmd_retries	Number of times to retry an OpenPET command
cmd_timeout	Length of time in seconds until timeout
d_acq_t	Length of time in seconds to acquire data (default = 10)
d_acq_osc_trig_win	Scope mode: trigger window (default = 5)
d_acq_osc_samples_before_trig	Scope mode: number of samples before trigger (default = 6)
d_acq_osc_samples	Scope mode: number of ADC samples (default = 32)
d_acq_osc_format	Scope mode: data format (default = 1)
d_dac_vpp	DAC resolution (default = 2 ¹⁰)
d_dac_v	DAC threshold (default = 200)
d_dac_type	DAC type (timing = 0, energy = 1)
d_process_data	Flag to determine if data will be processed (default = 1)
d_qu	Output queue
qusb_streamid	QuickUSB streaming ID
tbytes	Number of bytes QuickUSB collects
wbytes	Number of bytes written to disk
d_mode	OpenPET acquisition mode (scope, singles, etc)
d_mode_stgs	OpenPET acquisition mode settings
timelapse	Start time of acquisition
data	OpenPET data
outofmem	Flag to indicate if ran out of memory
eth	Contains Ethernet object
ip_src	IP source address
mac_src	MAC source address
mac_dst	MAC destination address
ip_dst	IP destination address
ethname	Name of Ethernet device
ip_brdst	Broadcasting IP
d_netmap	Ethernet netmap
eth_thru_test	Flag to enable/disable Ethernet throughput test
raw_pkt_id	Unique ID for tx packets
frame_loss	Frame loss detection
pktcnt	Counter for frames received

- `list_eth(self)`:
Gets the list of available network interfaces. If there are none, an error message will appear.
– **Returns:** List of network interfaces
- `list_qusb(self)`:
Gets the list of available QuickUSB devices. If there are no QuickUSB devices, an error message will appear.
– **Returns:** List of QuickUSB devices
- `init_qusb(self, devindex=0)`:
Initializes, checks, and configures QuickUSB. It checks that the correct QuickUSB model, DLL version, driver version, and firmware version are installed. If there is an error, a message will appear with information on what needs to be fixed.

Keyword Arguments	
devindex	QuickUSB device index

– **Returns:** QuickUSB object

- `init_eth(self, eindex, ip_dst):`

Initializes, checks, and configures Ethernet interface. It checks that the user supplied an Ethernet index. It also checks if the user supplied a destination IP address. If no IP address is given, then broadcast is used. If there is an error, a message will appear with information on what needs to be fixed.

Keyword Arguments	
<code>eindex</code>	Ethernet index
<code>ip_dst</code>	destination IP address

– **Returns:** True or False

- `rcmd(self):`

Reads OpenPET command or response

– **Returns:** Command read

- `wcmd(self, command):`

Writes OpenPET command

Keyword Arguments	
<code>command</code>	OpenPET command to be written

– **Returns:** QuickUsb.WriteCommand() return value

- `openpet_cmd(self, cmd_id, cmd_dst, cmd_payload):`

Executes OpenPET Hardware/Firmware/Software commands. If there is an error, a message will appear explaining what the error is. If QuickUSB is being used, `openpet_cmd_qusb` command is executed, and if Ethernet is being used, `openpet_cmd_eth` command is executed.

Keyword Arguments	
<code>cmd_id</code>	command ID
<code>cmd_dst</code>	destination address
<code>cmd_payload</code>	command payload

– **Returns:** Command ID, source, destination, and command payload (if available)

- `openpet_cmd_eth(self, cmd_id, cmd_dst, cmd_payload):`

Sends OpenPET command through Ethernet.

Keyword Arguments	
<code>cmd_id</code>	command ID
<code>cmd_dst</code>	destination address
<code>cmd_payload</code>	command payload

– **Returns:** Command ID, source, destination, and command payload (if available)

- `openpet_cmd_qusb(self, cmd_id, cmd_dst, cmd_payload):`

Sends OpenPET command through QuickUSB.

Keyword Arguments	
<code>cmd_id</code>	command ID
<code>cmd_dst</code>	destination address
<code>cmd_payload</code>	command payload

– **Returns:** Command ID, source, destination, and command payload (if available)

- `openpet_cmd_vresponse(self, cmd_id, response):`

Checks that the OpenPET response is valid. If there is an error, an error message is printed describing the error more specifically.

Keyword Arguments	
cmd_id	command ID
response	response received

– **Returns:** True or False if valid or invalid respectively.

- `opendatafile(self, mode):`

Creates and opens data output binary file

– **Returns:** File object

- `closedatafile(self):`

Closes data output binary file

– **Returns:** File object close() value

- `get_dac_bits(self, vol=None, dactype=None, vpp=None):`

Returns the correct bits for a given DAC voltage

Keyword Arguments	
vol	voltage requested
dactype	type of DAC, energy = 1 or timing = 0 (default)
vpp	range of the DAC, e.g., 2 ¹⁰ for 10-bit DAC

– **Returns:** 10-bit DAC data payload

- `resolve_osc_stg_payload(self, payload):`

Gets the scope mode settings from the payload. If no argument is provided, class values are returned

Keyword Arguments	
payload	32-bit scope mode settings payload

– **Returns:**

- * `d_acq_osc_trig_win` - trigger window
- * `d_acq_osc_samples_before_trig` - number of ADC samples before energy trigger
- * `d_acq_osc_samples` - total number of ADC samples
- * `d_acq_osc_format` - used internally for data handling

- `set_osc_stg_payload(self, d_acq_osc_trig_win, d_acq_osc_samples_before_trig, d_acq_osc_samples, d_acq_osc_format):`

Sets the scope mode settings with given function arguments and creates a corresponding payload. If no arguments are provided, class values are returned.

Keyword Arguments	
<code>d_acq_osc_trig_win</code>	trigger window
<code>d_acq_osc_samples_before_trig</code>	number of ADC samples before energy trigger
<code>d_acq_osc_samples</code>	total number of ADC samples
<code>d_acq_osc_format</code>	used internally for data handling (must be 1)

– **Returns:** 32-bit scope mode settings payload

- `getdsettings(self):`

Gets acquisition mode settings from CDUC

- **Returns:** Acquisition mode settings payload

- `setactionrun(self, ignore=0):`

Sets the CDUC's Acquisition Action to Run and broadcasts to all nodes.

Keyword Arguments	
<code>ignore</code>	if 1, skip the return value of OpenPET

- `setactionreset(self):`

Sets the CDUC's Acquisition Action to Reset and broadcasts to all nodes.

- `writeacqheader(self):`

Writes Acquisition Header to binary data file.

- `getdata(self, runreset=1):`

Start and acquire data through QuickUSB or Ethernet.

Keyword Arguments	
<code>runreset</code>	if 1, use default behavior

- `stopdata_qusb(self):`

Stop data acquisition from QuickUSB.

- `stopdata_eth(self):`

Stop data acquisition from Ethernet

- `ctype2uint32(self, data, nbytes):`

Converts ctype data to uint32 vector

Keyword Arguments	
<code>data</code>	Pointer to memory buffer
<code>nbytes</code>	Length of memory buffer

- **Returns:** uint32 data vector

- `data_cb(self, bstreamobj):`

Callback function for QuickUSB streaming API. This function also removes unnecessary zeros from the data.

Keyword Arguments	
<code>bstreamobj</code>	a pointer of type PQBULKSTREAM to QBULKSTREAM object

- `qu_writeraw(self, q):`

Queue function to write raw data to file. For Ethernet connections, this function removes unnecessary zeros in the data. This function is a wrapper function that calls `qu_write` in the end.

Keyword Arguments	
<code>q</code>	reference to Queue()

- `qu_write(self, data):`

Writes raw data to file.

Keyword Arguments	
<code>data</code>	data to write

- `loaddatafiletomemory(self, filename):`

Load file content to memory

Keyword Arguments	
filename	file object

– **Returns:** uint32 numpy array of file content

- `writesramtofile(self, filename, dst_addr, size=1024*1024*2, offset=0):`

Write SRAM content from OpenPET to disk

Keyword Arguments	
filename	file object
dst_addr	OpenPET destination address (don't use broadcast here)
size	number of bytes to write to file
offset	offset address in SRAM to read from

- `writefiletosram(self, filename, dst_addr, offset=0):`

Write SRAM content from disk to OpenPET

Keyword Arguments	
filename	file object
dst_addr	OpenPET destination address (don't use broadcast here)
offset	offset address in SRAM to write to

Note: Writes 32-bits at a time

- `str2num(self, x):`

Converts a string number from a decimal or hex to an integer.

Keyword Arguments	
x	string number

– **Returns:** Integer

- `show_acq_progress(self):`

Prints acquisition progress.

- `ipv4innetwork(self, ip_src, ip_dst, netmask):`

Checks is the IP address is within a network mask.

Keyword Arguments	
ip_src	IP address of local interface
ip_dst	IP address of OpenPET chassis
netmask	netmask of local interface

– **Returns:** True or False

- `build_nm_packet(self, byteit):`

Builds a raw UDP packet to be used in netmap.

Keyword Arguments	
byteit	byte to send

– **Returns:** UDP packet

- `send_raw_packet_nm(self, byteit):`

Sends raw packet using netmap tx ring.

Keyword Arguments	
byteit	byte to send

- `ip_chksum(self, phead):`

Calculates IP header checksum.

Keyword Arguments	
phead	IP header packet

- **Returns:** 16-bit checksum

OpenPET Control and Analysis Tools

OpenPET Control and Analysis Tools (OpenPET CAT) is a data acquisition and analysis software for the OpenPET electronics based on the ROOT framework. OpenPET CAT utilizes the object-oriented design in providing basic utilities, control and analysis tools for the OpenPET electronics. Using OpenPET CAT requires installing the [ROOT package](#).

Using OpenPET CAT is optional; users can run the executables “openpet” (see *System Commands and Responses* (page 39) section) to configure the system and to acquire data directly.

Installing ROOT

Go to [Downloading ROOT](#) for instructions on downloading and installing ROOT. Although ROOT is supported on many platforms, OpenPET only supports the Windows version of ROOT using the Microsoft Visual C++ compiler. Current OpenPET CAT supports ROOT version 5.34/30 and Microsoft Visual C++ 2010. If you plan to develop and compile OpenPET CAT codes, you also need to install Microsoft Visual C++ 2010 Express, which can be downloaded at no cost from [Visual Studio Downloads](#), and [Cygwin](#).

For instructions on how to use OpenPET CAT, see the [OpenPET CAT](#) section in the User's Guide.

OpenPET CAT Repository

The OpenPET CAT repository consists of six folders.

1. src - Contains all source files
2. inc - Contains all header files
3. bin - Once the code is compiled, contains all executables
4. lib - Once the code is compiled, contains the OpenPET CAT library
5. doc - Contains documentation
6. examples - Contains example macros and configuration files

Installing OpenPET CAT

First, clone the OpenPET HostPC repository and follow the instructions in the README file found in the sw directory. Next, clone the OpenPET CAT repository. After cloning, open cygwin and navigate to the src directory. Run the following commands to build OpenPET CAT.

```
$ make clean
$ make
```

After it is built, a GUI executable `openpet_cat.exe` will appear in the `bin` directory, and the OpenPET CAT library file `libopenpetcat.dll` will appear in the `lib` directory. If `ROOT` was previously installed, this library will also be copied to the directory `<installed ROOT directory>/bin` (e.g., `$ROOTSYS/bin`). If `ROOT` has not been installed, install `ROOT` as outlined above and then copy the library file into the `ROOT` `bin` directory.

The OpenPET CAT software is now ready to use. For further instructions, please refer to the [OpenPET CAT](#) section in the User's Guide.

(If optional tool, then this should be in a separate document?)

OpenPET Firmware Parameters

This section will outline the OpenPET firmware parameters in the Main, IO, and detector board field programmable gate arrays (FPGAs). These parameters are VHDL generics that can be changed during compilation.

Main FPGA

Entity - main

Name	Type	Description
g_NODE_TYPE	std_logic_vector(3 downto 0)	SupportBoard Function (Type)
g_slice_div	integer	Clk to slice relation. 0 (default) divide by 8. 1 means divide by 16.
g_adc_channels	positive	Number of ADC channels to deserialize
g_adc_number	positive	Number of ADC chips
g_adc_resolution	positive	ADC resolution
g_QUSB_FD_WIDTH	positive	QuickUSB Data bus width
g_QUSB_ADR_WIDTH	positive	QuickUSB Address bus width
g_QUSB_CMD_PKTS	positive	Number of QuickUSB packets (write cycles) to complete a single OpenPET command.
g_osc_num_adc_samples	positive	Total number of ADC samples per db
g_sng_max_pipeline_stages	positive	Maximum number of pipeline stages
g_en_osc_mode	boolean	Generate oscilloscope mode logic
g_sng_mode_type	integer	
6.1. Main FPGA		Generate singles mode logic. 0=no logic generated. 1=LBNL example (default)
g_en_qusb	boolean	

Component

qusb

Name	Type	Description
g_FD_WIDTH	positive	QuickUSB Data bus width
g_ADR_WIDTH	positive	QuickUSB Address bus width
g_CMD_PKTS	positive	Number of QuickUSB packets (write cycles) to complete a single OpenPET command
g_DATA_FIFO_DEPTH	positive	Depth of output data FIFO

swfw_cmd

Name	Type	Description
g_NODE_TYPE	std_logic_vector(3 downto 0)	SupportBoard Function (Type)
g_osc_num_adc_samples	positive	Total number of ADC samples per db
g_adc_number	positive	Number of ADC chips
g_adc_channels	positive	Number of ADC channels to deserialize
g_sng_max_pipeline_stages	positive	Maximum number of pipeline stages
g_en_tdc	boolean	Enable/disable TDC logic
g_en_osc_mode	boolean	Generate oscilloscope mode logic
g_sng_mode_type	integer	Generate singles mode logic. 0=no logic generated. 1=LBNL example (default)

processdata

Name	Type	Description
g_NODE_TYPE	std_logic_vector(3 downto 0)	SupportBoard Function (Type)
g_osc_fifo_depth	positive	Oscilloscope FIFO depth
g_en_osc_mode	boolean	Generate oscilloscope mode logic
g_sng_mode_type	integer	Generate singles mode logic. 0=no logic generated. 1=LBNL example (default)

ether_rgmii

Name	Type	Description
g_ip_src	std_logic_vector(31 downto 0)	Default source IP
g_mac_src	std_logic_vector(47 downto 0)	Default source mac
g_ip_dst	std_logic_vector(31 downto 0)	Default destination IP
g_mac_dst	std_logic_vector(47 downto 0)	Default destination mac
g_jumbo_dw	positive	Jumbo frames
g_udp_width	positive	Width of data bus
g_osc_num_adc_samples	positive	Depth of output data FIFO

Main FPGA VHD file

IO FPGA

Entity - io

Name	Type	Description
g_NODE_TYPE	std_logic_vector(3 downto 0)	SupportBoard Function (Type)
g_slice_div	integer	Clk to slice relation. 0 (default) divide by 8. 1 means divide by 16.
g_adc_channels	positive	Number of ADC channels to deserialize
g_adc_number	positive	Number of ADC chips
g_adc_resolution	positive	ADC resolution
g_osc_num_adc_samples	positive	Total number of ADC samples per db
g_sng_max_pipeline_stages	positive	Maximum number of pipeline stages
g_en_osc_mode	boolean	Generate oscilloscope mode logic
g_sng_mode_type	integer	Generate singles mode logic. 0=no logic generated. 1=LBNL example (default)

Component

swfw_cmd

Name	Type	Description
g_osc_num_adc_samples	positive	Total number of ADC samples per db
g_adc_number	positive	Number of ADC chips
g_adc_channels	positive	Number of ADC channels to deserialize
g_en_tdc	boolean	Enable/disable TDC logic
g_frame_width	positive	ADCClk_Freq / SliceClk_Freq
g_sng_max_pipeline_stages	positive	Maximum number of pipeline stages
g_en_osc_mode	boolean	Generate oscilloscope mode logic
g_sng_mode_type	integer	Generate singles mode logic. 0=no logic generated. 1=LBNL example (default)

processdata

Name	Type	Description
g_NODE_TYPE	std_logic_vector(3 downto 0)	SupportBoard Function (Type)
g_osc_fifo_depth	positive	Oscilloscope FIFO depth
g_frame_width	positive	ADCClk_Freq / SliceClk_Freq
g_sng_max_pipeline_stages	positive	Maximum number of pipeline stages
g_en_osc_mode	boolean	Generate oscilloscope mode logic
g_sng_mode_type	integer	Generate singles mode logic. 0=no logic generated. 1=LBNL example (default)

IO FPGA VHD file

Detector Board FPGA

Entity - db16ch

Name	Type	Description
g_en_debug	boolean	Enable/disable debugging
g_slice_div	integer	Clk to slice relation. 0 (default) divide by 8. 1 means divide by 16.
g_adc_channels	positive	Number of ADC channels to deserialize
g_adc_number	positive	Number of ADC chips
g_adc_resolution	positive	ADC resolution
g_tdc_type	integer	Generate TDC logic. 0=no logic 1=waveunion, 2=multiphase (default)
g_tdc_resolution	positive	TDC resolution
g_en_osc_mode	boolean	Generate oscilloscope mode logic
g_osc_num_adc_samples	positive	Total number of ADC samples per run
g_sng_mode_type	integer	Generate singles mode logic. 0=no logic generated. 1=LBNL example (default)
g_sng_max_pipeline_stages	positive	Maximum number of pipeline stages

Component

swfw_cmd

Name	Type	Description
g_osc_num_adc_samples	positive	Total number of ADC samples per db
g_adc_number	positive	Number of ADC chips
g_adc_channels	positive	Number of ADC channels to deserialize
g_frame_width	positive	ADCClk_Freq / SliceClk_Freq
g_sng_max_pipeline_stages	positive	Maximum number of pipeline stages
g_en_osc_mode	boolean	Generate oscilloscope mode logic
g_sng_mode_type	integer	Generate singles mode logic. 0=no logic generated. 1=LBNL example (default)

adc_16ch

Name	Type	Description
g_adc_channels	positive	Number of ADC channels to deserialize
g_adc_number	positive	Number of ADC chips
g_adc_resolution	positive	ADC resolution

processdata

Name	Type	Description
g_adc_channels	positive	Number of ADC channels to deserialize
g_adc_number	positive	Number of ADC chips
g_adc_resolution	positive	ADC resolution
g_tdc_resolution	positive	TDC resolution
g_osc_fifo_depth	positive	Oscilloscope FIFO depth
g_frame_width	positive	ADCClk_Freq / SliceClk_Freq
g_sng_max_pipeline_stages	positive	Maximum number of pipeline stages
g_tdc_type	integer	Generate TDC logic. 0=no logic 1=waveunion, 2=multiphase (default)
g_en_osc_mode	boolean	Generate oscilloscope mode logic
g_sng_mode_type	integer	Generate singles mode logic. 0=no logic generated. 1=LBNL example (default)

tcd

Name	Type	Description
g_en_debug	boolean	Enable/disable debugging
g_tdc_type	integer	Generate TDC logic. 0=no logic 1=waveunion, 2=multiphase (default)
g_adc_channels	positive	Number of ADC channels to deserialize
g_adc_number	positive	Number of ADC chips
g_tdc_resolution	positive	TDC resolution

sram

Name	Type	Description
g_dataw	positive	Data width
g_addrw	positive	Address width

Detector Board FPGA VHD file

OpenPET Software Interface

OpenPET provides a number of example Python scripts that can be used to operate the OpenPET system. However, developers may create their own scripts with additional functionality or rewrite the scripts in a different language. Please note that if these scripts are rewritten in a different language, the OpenPET library must also be rewritten or compiled such that it can be read by the new language. OpenPET uses Python's logging functionality to print status messages throughout the script, but this is optional when recreating the script although it is useful.

Warning: While developers are given the freedom to rewrite in a different language or add onto the existing scripts, the basic configuration sequence laid out by these examples is crucial and should not be changed.

QuickUSB

Simple Scope Example

This is a basic example that acquires data in scope mode using QuickUSB. The sequence of steps are outlined and described below.

1. Create an OpenPET object

```
op = OpenPET()
```

2. Initialize QuickUSB with the desired module

```
op.qusb = op.init_qusb()
```

In this example, the default QuickUSB deviceIndex (0) is used. One can also use deviceIndex 2 by putting 2 as the function argument.

```
op.qusb = op.init_qusb(2)
```

3. Set data acquisition duration in seconds, partial seconds are OK

```
op.d_acq_t = 10
```

Here, we set the acquisition time to 10 seconds.

4. Set Scope mode settings payload

```
oscpayload = 0x02000101
```

In this example, we set the settings for scope mode to be 16 samples, 0 before trigger, and 2 trigger window. For more information on this payload, please see *Command ID: 0x0005* (page 48).

5. Set target address and broadcast bit

```
trgt = 0x8003
```

In this example, we set the broadcast bit to 1 and the target to be DB 3.

6. Set mode to scope mode (payload = 0x1)

```
op.openpet_cmd(0x0003, trgt, 0x00000001)
```

7. Set scope mode settings

```
op.openpet_cmd(0x0005, trgt, oscpayload)
```

This sets the scope mode settings to the predefined values set in step 4. The target address is set in step 5. One can also read back the scope mode settings to check that they are correct as shown below.

```
cmd, srcp, dst, pyld = op.openpet_cmd(0x0006, trgt, 0)
```

8. Acquire data

```
op.getdata()
```

In this example, the default settings of this function are used. Namely, the Acquisition Action is set to Reset and then Run by default at the beginning and set to Reset at the end.

Note: The default ADC and DAC configurations are used in this example.

Intermediate Scope Example

This next example is more complex than the previous one in that some configurations that were left as default are now manually changed. The basic structure remains the same.

1. Create an OpenPET object

```
op = OpenPET()
```

2. Initialize QuickUSB with the desired module

```
op.qusb = op.init_qusb()
```

In this example, the default QuickUSB deviceIndex (0) is used. One can also use deviceIndex 2 by putting 2 as the function argument.

```
op.qusb = op.init_qusb(2)
```

3. (Optional) Change autogenerated file name

```
op.ofilename = "test.openpetd"
```

4. Set data acquisition duration in seconds, partial seconds are OK

```
op.d_acq_t = 10
```

Here, we set the acquisition time to 10 seconds.

5. Set the scope mode settings payload

```
osc_stg_pld = op.set_osc_stg_payload(win, btg, smp, fmt)
```

The arguments used here are integer variables that can be set to create the desired payload. For a more detailed explanation, see [OpenPET Library Functions](#)

6. Set target address and broadcast bit

```
trgt = 0x8003
```

In this example, we set the broadcast bit to 1 and the target to be DB 3.

7. Change ADC settings

Instead of keeping the default settings that are configured on powerup, the ADC settings can be changed by sending OpenPET commands. For example, the command below says to write to the "ADS5282" chip to output PAT_DESKEW fixed pattern on DB 6:

```
op.openpet_cmd(0x0104, 0x0006, 0x81140001)
```

One can also reset the ADC using the command ID 0x0103.

Additionally, the gain can be changed. More explanation on the payload for changing the gain can be found [here](#). The example below changes the gain on DB 1 to 6dB for channels 4 to 7 with broadcasting.

```
op.openpet_cmd(0x0104, 0x0001, 0x80AC6666)
```

8. Change DAC settings

The DAC settings can also be changed if desired. The example below tells the system to write to the DAC to set the energy threshold to 200mV on the set target address which in this case is DB 3.

```
op.openpet_cmd(0x0106, trgt, 0x8007E032)
```

There are also helper functions to determine what the payload should be for a given threshold value. First, set the timing and energy DAC mV threshold values. Then use the function `get_dac_bits()` to convert those mV threshold values into payloads. Once the payloads are created, the DAC can settings can be changed. An example of this process is below.

```
op.d_tdac_v = 100 # timing DAC threshold 100 mV range [0, 2500]mV
op.d_edac_v = 4096 # energy DAC threshold 100 mV range [0, 4096]mV
```

```
energydacpayload = 0x8007E000|op.get_dac_bits(dactype=1)
timingdacpayload = 0x8807E000|op.get_dac_bits(dactype=0)
op.openpet_cmd(0x0106, trgt, energydacpayload)
op.openpet_cmd(0x0106, trgt, timingdacpayload)
```

9. Set Acquisition Action to Reset.

```
op.openpet_cmd(0x0007, trgt, 0)
```

This is used to clear previous buffers, FIFOs, and registers in firmware.

Warning: This does not reset external peripheral devices like DAC and ADC settings

10. Set firmware threshold

```
op.openpet_cmd(0x0108, trgt, fwth)
```

This sets the firmware threshold to a specific value on the target detector board. To double check that the correct value was set, the payload can be read back using the following command.

```
cmd, srcp, dst, pyld = op.openpet_cmd(0x0109, trgt, 0)
```

For more information on how to set the firmware threshold, see *Command ID: 0x0108* (page 59).

11. Set mode to scope mode (payload = 0x1)

```
op.openpet_cmd(0x0003, trgt, 0x00000001)
```

To double check that the mode was set correctly, one can read back the mode setting from the target or from a different detector board as shown below:

```
cmd, srcp, dst, pyld = op.openpet_cmd(0x0004, trgt, 0)
cmd, srcp, dst, pyld = op.openpet_cmd(0x0004, 0x0001, 0)
```

12. Set scope mode settings

```
op.openpet_cmd(0x0005, trgt, osc_stg_pld)
```

The command above sets the scope mode settings to the previously generated payload. To double check that the settings are correct, read back scope mode settings.

```
cmd, srcp, dst, pyld = op.openpet_cmd(0x0006, trgt, 0)
```

If the scope mode settings are read back, there are two options. You can exit the system or reset the scope mode settings with the new values set by the firmware. Below is an example of setting the new values from the firmware.

```
w,b,s,f = op.resolve_osc_stg_payload(pyld) # Get values from payload_
↳that was read back
op.set_osc_stg_payload(w,b,s,f) # Set those new values
```

13. Set trigger mask

In this example, we show how to set the hardware trigger mask. The command sequence below outlines the basic command flow. First, turn all the channels on and reset the TDC and calibrate it. After the TDC calibrates, give the command to run the TDC control register. To double check, one can read back the current TDC control register. All these commands are sent to the target detector board.

```
op.openpet_cmd(0x0009, trgt, 0xFFFFFFFF) # All channels are ON
op.openpet_cmd(0x0101, trgt, 0x00000080) # Write TDC reset
op.openpet_cmd(0x0101, trgt, 0x00000002) # Write TDC calibration
time.sleep(5) # TDC calibrate for 5 sec
op.openpet_cmd(0x0101, trgt, 0x00000004) # Write TDC ctrl
cmd, srcp, dst, pyld = op.openpet_cmd(0x0102, trgt, 0) # read back_
↪tdc ctrl
```

14. Set Acquisition Action to Run

By default, the `getdata()` function will set the acquisition action to Run. However, it can also be done manually as shown here.

```
op.openpet_cmd(0x0007, trgt, 0x00000002)
```

15. Acquire data

Since the default functionality of `getdata` is not being used in this example, the argument of `getdata` is 0. In this example, we manually set the Acquisition Action to Reset and Run instead of doing it in the `getdata` function.

```
op.getdata(0)
```

16. Set Acquisition Action to Stop

```
op.openpet_cmd(0x0007, trgt, 0x00000001)
```

This command allows the user to tell the firmware to pause. This allows you to resume data acquisition without flushing the FIFOs in the FPGAs. However, if you do want to flush the FIFOs, simply set Action to reset and then run.

You can also check that the firmware has actually stopped by using the command below.

```
cmd, srcp, dst, pyld = op.openpet_cmd(0x0008, trgt, 0)
```

If the returned payload is correct, then it has stopped.

Simple Singles Example

This is a basic example that acquires data in singles mode using QuickUSB. The sequence of steps are outlined and described below.

1. Create an OpenPET object

```
op = OpenPET()
```

2. Initialize QuickUSB with the desired module

```
op.qusb = op.init_qusb()
```

In this example, the default QuickUSB deviceIndex (0) is used. One can also use deviceIndex 2 by putting 2 as the function argument.

```
op.qusb = op.init_qusb(2)
```

3. Set data acquisition duration in seconds, partial seconds are OK

```
op.d_acq_t = 10
```

Here, we set the acquisition time to 10 seconds.

4. Set Singles mode settings payload

```
oscpayload = 0x00000006
```

This sets the singles mode settings so that it will integrate over 6 ADC samples. For more information on this payload, please see *Command ID: 0x0005* (page 48).

5. Set target address and broadcast bit

```
trgt = 0x8003
```

In this example, we set the broadcast bit to 1 and the target to be DB 3.

6. Set mode to singles mode (payload = 0x2)

```
op.openpet_cmd(0x0003, trgt, 0x00000002)
```

7. Set singles mode settings

```
op.openpet_cmd(0x0005, trgt, oscpayload)
```

This sets the scope mode settings to the predefined values set in step 4. The target address is set in step 5. One can also read back the scope mode settings to check that they are correct as shown below.

```
cmd, srcp, dst, pyld = op.openpet_cmd(0x0006, trgt, 0)
```

8. Acquire data

```
op.getdata()
```

In this example, the default settings of this function are used. Namely, the Acquisition Action is set to Reset and then Run by default at the beginning and set to Reset at the end.

Note: The default ADC and DAC configurations are used in this example.

Ethernet

Simple Scope Example

This is a basic example that acquires data in scope mode using an Ethernet connection. The sequence of steps are outlined and described below.

1. Create an OpenPET object

```
op = OpenPET()
```

2. Initialize Ethernet

```
op.eth = op.init_eth(None, '10.10.10.2')
```

The destination IP is 10.10.10.2, and OpenPET will automatically find the correct ethernet interface.

3. Set data acquisition duration in seconds, partial seconds are OK

```
op.d_acq_t = 10
```

Here, we set the acquisition time to 10 seconds.

4. Set Scope mode settings payload

```
oscpayload = 0x02000101
```

In this example, we set the settings for scope mode to be 16 samples, 0 before trigger, and 2 trigger window. For more information on this payload, please see *Command ID: 0x0005* (page 48).

5. Set target address and broadcast bit

```
trgt = 0x8003
```

In this example, we set the broadcast bit to 1 and the target to be DB 3.

6. Set mode to scope mode (payload = 0x1)

```
op.openpet_cmd(0x0003, trgt, 0x00000001)
```

7. Set scope mode settings

```
op.openpet_cmd(0x0005, trgt, oscpayload)
```

This sets the scope mode settings to the predefined values set in step 4. The target address is set in step 5. One can also read back the scope mode settings to check that they are correct as shown below.

```
cmd, srcp, dst, pyld = op.openpet_cmd(0x0006, trgt, 0)
```

8. Acquire data

```
op.getdata()
```

In this example, the default settings of this function are used. Namely, the Acquisition Action is set to Rest and Run by default at the beginning and set to Reset at the end.

Note: The default ADC and DAC configurations are used in this example.

Intermediate Scope Example

Again, here is a more complex example using an Ethernet connection. The basic structure remains the same.

1. Create an OpenPET object

```
op = OpenPET()
```

2. Initialize Ethernet

```
op.eth = op.init_eth(None, '10.10.10.2')
```

The destination IP is 10.10.10.2, and OpenPET will automatically find the correct ethernet interface.

3. (Optional) Change autogenerated file name

```
op.ofilename = "test.openpetd"
```

4. Set data acquisition duration in seconds, partial seconds are OK

```
op.d_acq_t = 10
```

Here, we set the acquisition time to 10 seconds.

5. Set the scope mode settings payload

```
osc_stg_pld = op.set_osc_stg_payload(win, btg, smp, fmt)
```

The arguments used here are integer variables that can be set to create the desired payload. For a more detailed explanation, see [OpenPET Library Functions](#)

6. Set target address and broadcast bit

```
trgt = 0x8001
```

In this example, we set the broadcast bit to 1 and the target to be DB 1.

7. Change ADC settings

Instead of keeping the default settings that are configured on powerup, the ADC settings can be changed by sending OpenPET commands. For example, the command below says to write to the "ADS5282" chip to output PAT_DESKEW fixed pattern on DB 6:

```
op.openpet_cmd(0x0104, 0x0006, 0x81140001)
```

One can also reset the ADC using the command ID 0x0103.

Additionally, the gain can be changed. More explanation on the payload for changing the gain can be found [here](#). The example below changes the gain on DB 3 to 12dB for channels 8 to 11 without broadcasting.

```
op.openpet_cmd(0x0104, 0x0003, 0x08A8CCCC)
```

8. Change DAC settings

The DAC settings can also be modified. The example below tells the system to write to the DAC to set the energy threshold to 200mV on the set target address which in this case is DB 1.

```
op.openpet_cmd(0x0106, trgt, 0x8007E032)
```

There are also helper functions to determine what the payload should be for a given threshold value. First, set the timing and energy DAC mV threshold values. Then use the function

`get_dac_bits()` to convert those mV threshold values into payloads. Once the payloads are created, the DAC can settings can be changed. An example of this process is below.

```
op.d_tdac_v = 100 # timing DAC threshold 100 mV range [0, 2500]mV
op.d_edac_v = 4096 # energy DAC threshold 100 mV range [0, 4096]mV
energydacpayload = 0x8007E000|op.get_dac_bits(dactype=1)
timingdacpayload = 0x8807E000|op.get_dac_bits(dactype=0)
op.openpet_cmd(0x0106, trgt, energydacpayload)
op.openpet_cmd(0x0106, trgt, timingdacpayload)
```

9. Set Acquisition Action to Reset.

```
op.openpet_cmd(0x0007, trgt, 0)
```

This is used to clear previous buffers, FIFOs, and registers in firmware.

Warning: This does not reset external peripheral devices like DAC and ADC settings

10. Set firmware threshold

```
op.openpet_cmd(0x0108, trgt, fwth)
```

This sets the firmware threshold to a specific value on the target detector board. To double check that the correct value was set, the payload can be read back using the following command.

```
cmd, srcp, dst, pyld = op.openpet_cmd(0x0109, trgt, 0)
```

For more information on how to set the firmware threshold, see *Command ID: 0x0108* (page 59).

11. Set mode to scope mode (payload = 0x1)

```
op.openpet_cmd(0x0003, trgt, 0x00000001)
```

To double check that the mode was set correctly, one can read back the mode setting from the target or from a different detector board:

```
cmd, srcp, dst, pyld = op.openpet_cmd(0x0004, trgt, 0)
```

12. Set scope mode settings

```
op.openpet_cmd(0x0005, trgt, osc_stg_pld)
```

The command above sets the scope mode settings to the previously generated payload. To double check that the settings are correct, read back scope mode settings.

```
cmd, srcp, dst, pyld = op.openpet_cmd(0x0006, trgt, 0)
```

If the scope mode settings are read back, there are two options. You can exit the system or reset the scope mode settings with the new values set by the firmware. Below is an example of setting the new values from the firmware.

```
w,b,s,f = op.resolve_osc_stg_payload(pyld) # Get values from payload_
↳that was read back
op.set_osc_stg_payload(w,b,s,f) # Set those new values
```

13. Set trigger mask

In this example, we show how to set the trigger mask. The command sequence below outlines the basic command flow. First, turn all the channels on and reset the TDC and calibrate it. After the TDC calibrates, give the command to run the TDC control register. To double check, one can read back the current TDC control register. All these commands are sent to the target detector board.

```
op.openpet_cmd(0x0009, trgt, 0xFFFFFFFF) # All channels are ON
op.openpet_cmd(0x0101, trgt, 0x00000080) # Write TDC reset
op.openpet_cmd(0x0101, trgt, 0x00000002) # Write TDC calibration
time.sleep(5) # TDC calibrate for 5 sec
op.openpet_cmd(0x0101, trgt, 0x00000004) # Write TDC ctrl
cmd, srcp, dst, pyld = op.openpet_cmd(0x0102, trgt, 0) # read back
↳tdc ctrl
```

14. Set Acquisition Action to Run

By default, the `getdata()` function will set the acquisition action to Run. However, it can also be done manually as shown here.

```
op.openpet_cmd(0x0007, trgt, 0x00000002)
```

15. Acquire data

Since the default functionality of `getdata` is not being used in this example, the argument of `getdata` is 0. In this example, we manually set the Acquisition Action to Reset and Run instead of doing it in the `getdata` function.

```
op.getdata(0)
```

16. Set Acquisition Action to Stop

```
op.openpet_cmd(0x0007, trgt, 0x00000001)
```

This command allows the user to tell the firmware to pause, allowing you to resume data acquisition without flushing the FIFOs in the FPGAs. However, if you do want to flush the FIFOs, simply set Action to reset and then run.

You can also check that the firmware has actually stopped by using the command below.

```
cmd, srcp, dst, pyld = op.openpet_cmd(0x0008, trgt, 0)
```

If the returned payload is correct, then it has stopped.

Firmware

In order to troubleshoot the FPGAs on the support board and detector board, one can use a JTAG connection. For the support board, to test just the Main FPGA, use the JTAG along with the green jumper board as it is set up for normal operations (Fig. 8.1).

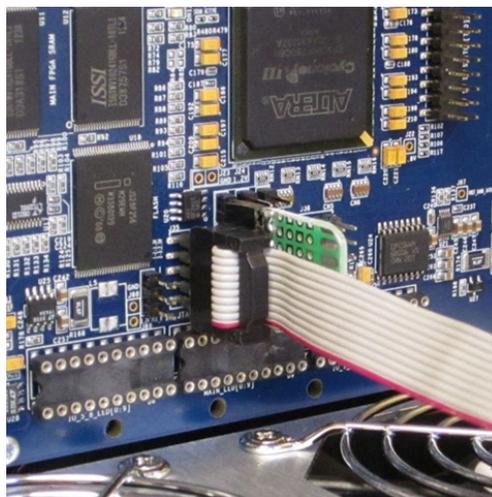


Fig. 8.1: Image of the JTAG connected to the support board.

To test the Main and IO FPGAs simultaneously, remove the green jumper board and use the JTAG chain depicted in Fig. 8.3. Each circle in the image represents a pin. The picture is rotated 90 degrees clockwise compared to how the pins are actually oriented on the board in the crate as can be seen when comparing Fig. 8.2 to Fig. 8.3. Jumpers and shunts are used to connect the pins.

For the detector board, only the JTAG need be connected. This is shown in Fig. 8.4.

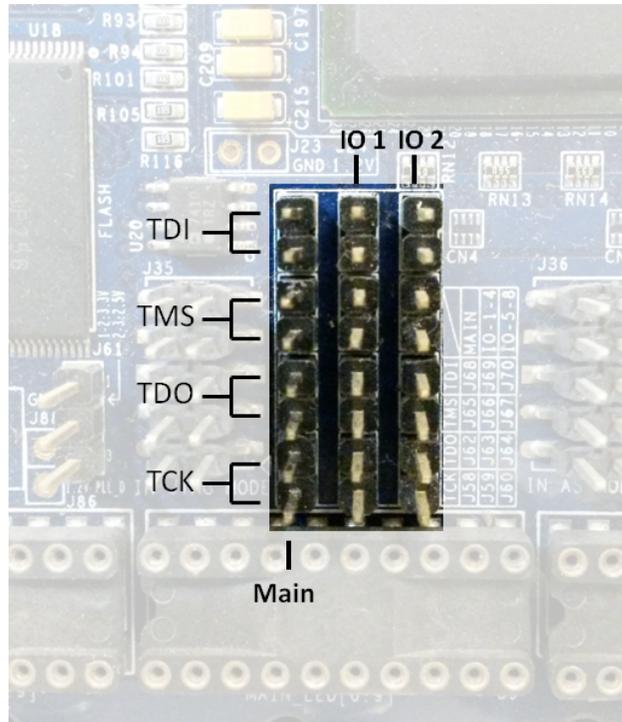


Fig. 8.2: Picture of the pins used in the JTAG chain

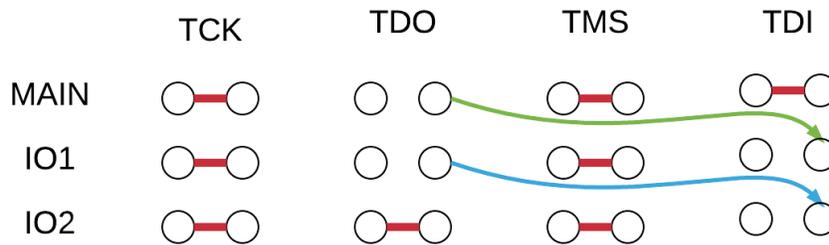


Fig. 8.3: Diagram of the JTAG chain for debugging Main and IO FPGAs on the support board.

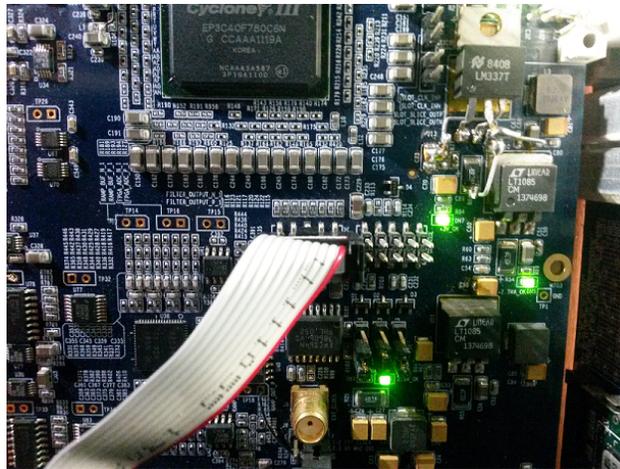


Fig. 8.4: Image of the JTAG connected to the detector board.

Embedded Software

With the JTAG connected, one can see what happens in NIOS for the support board and detector board. After connecting the JTAG to the desired board, open the NIOS command shell and then run the `nios2-terminal` executable. Now, when a command is sent to the system, the progress in NIOS can be seen. Fig. 8.5 shows an example of this output.

```

FTAbunimeh@OpenPET-WS /cydrive/c/Users/FTAbunimeh/openpet/faisal/sb/sw/main/app
$ nios2-terminal.exe
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster on openpet-3.lbl.gov [USB-01]", device 1, instance 0
nios2-terminal: <Use the IDE stop button or Ctrl-C to terminate>

[0] Command Received [ [0] 0x80 [1] 0x07 [2] 0x40 [3] 0x00 [4] 0x80 [5] 0x01 [6]
x00 [9] 0x00 ]
ID = 0x8007
SRC = 0x4000
DSI = 0x8001
PLD = 0x00000000
ASYNC = 1
This is broadcast command
[SPI][slave=0x000003FF] WRITING [ 0x80078001 0x00000000 ]
[SPI] write-readback [ 0xFFFFFFFF ]
[SPI] write-readback [ 0xFFFFFFFF ]
done with cmd
Responding back with [ 0x80 0x07 0x40 0x00 0x40 0x00 0x00 0x00 0x00 0x00 ]

```

Fig. 8.5: Image of the NIOS command shell when a command is sent to the system.

Additionally, one can see what happens during the bootup sequence of the support board as shown in Fig. 8.6.

If one wants to see NIOS on both the support board and detector board simultaneously, connect a JTAG to each board and open two NIOS command shells. In one window, run the `nios2-terminal` command specifying which JTAG cable is to be used by adding a parameter. In the command `nios2-terminal -c1`, the parameter `c1` indicates the cable in slot 1 is to be used. To set up the second window, run `nios2-terminal -c2` to view the information from the second board.

To determine what cables are connected to the system, there is a command `jtagconfig` that shows the list of cables. However, to determine which cable is connected to which board, one must physically look at the system setup.

For more NIOSII command-line tools, see this [page](#)

```
FTAbunimeh@OpenPET-WS /cygdrive/c/Users/FTAbunimeh/openpet/faisal/sb/sw/main/app
$ nios2-download -r -g main.elf && nios2-terminal.exe
Using cable "USB-Blaster on openpet-3.lbl.gov [USB-01]", device 1, instance 0x00
Resetting and pausing target processor: OK
Initializing CPU cache (if present)
OK
Downloaded 68KB in 1.1s (61.8KB/s)
Verified OK
Starting processor at address 0x008001B8
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster on openpet-3.lbl.gov [USB-01]", device 1, instance 0
nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate)

OpenPET LBNL SupportBoard-Main FPGA
Initializing PHY
External 1 GbE completed RGMII configuration.
Loading children FPGAs with bitstream stored on EPCS
Waiting for CMDs
```

Fig. 8.6: Image of the NIOS command shell during the bootup sequence.

CHAPTER 9

What Exists Now

(Previous sections describe the overall, long-term development plans. Describe here what has already been implemented in version 1.0 Oscilloscope mode.)

Use what subsections?

How To Be An OpenPET Developer

Signup as Developer

In order to have read/write access to source code on the OpenPET BitBucket repository, users are required to sign up for an OpenPET ‘developer’ account. If you don’t have a developer account, please send an email request to openpet@lbl.gov with the following information:

- Your name
- Institute/organization
- Names of individuals who will be working on the project
- What you intend to do with the source code
- Your potential contribution to this project

If your email does not include this complete information, then your request to be a Developer will not be granted. Once you have a developer account, you can view our BitBucket repository.

BitBucket Repository

(Describe how to use BitBucket, how to get code approved, etc.)

Getting Started

BitBucket Getting Started

Contributer vs. Developer

There are two different levels of access that are possible and will be determined by the OpenPET administration. A contributor is given read and write access which means he can clone, edit, and push changes directly from a local

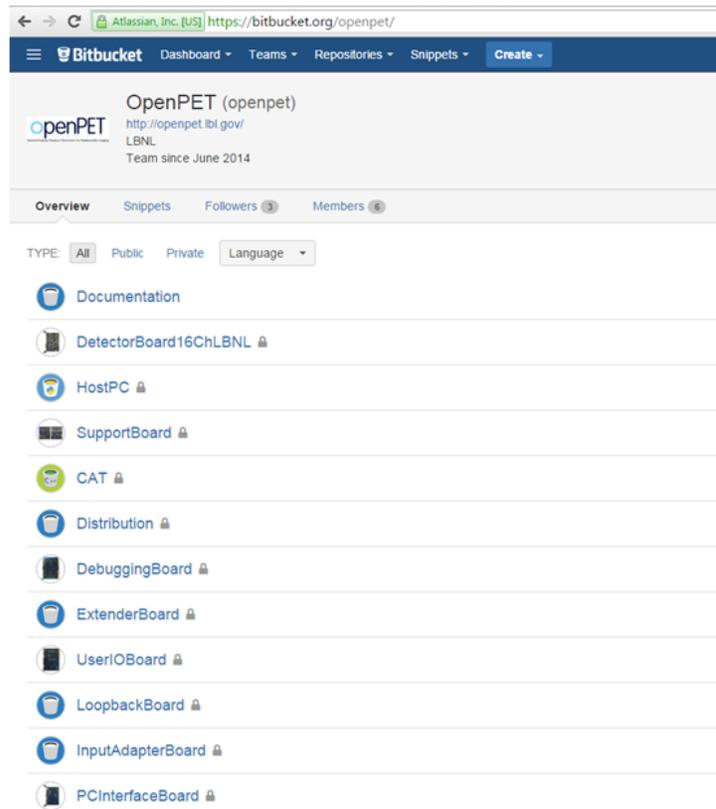


Fig. 10.1: OpenPET BitBucket repository

machine. A developer is given read access only, meaning he can view or clone the repository. For a developer to merge his work with the default branch, he must submit a pull request.

Code Style Requirements

(See Framework section 6.2.)

C Programming (NOIS II)

(See Framework section 6.2.1. Include program file style, comments style, examples, etc.)

VDHL Programming

(See Framework section 6.2.2?

Look up CERN Open Hardware standard)

VHDL CERN guide info at:

<http://www.ohwr.org/documents/24> <http://www.ohwr.org/attachments/554/VHDLcoding.pdf>

CHAPTER 11

Appendices

Application Notes

CHAPTER 12

Acknowledgements

Initial funding for the LBNL portion of this work was supported by the Director, Office of Science, Office of Biological and Environmental Research, Biological Systems Science Division of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Subsequent work is supported by the National Institutes of Health of the US Department of Health and Human Services under grant R01 EB016104.

Reference to a company or product name does not imply approval or recommendation by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.

CHAPTER 13

References

CHAPTER 14

Index

- genindex

Symbols

16-Channel Detector Board, 28

B

Bus IO, 27

C

Coincidence Unit, 5

Coincidence Unit Controller, 7

Coincidence/Detector Unit Controller, 5

D

Detector Unit, 5

Detector Unit Controller, 5

L

Large System, 7

S

Small System, 7

Standard System, 7

Support Board, 19

Support Crate, 5