The XL3 Manual and the Penn DAQ User Guide

December 10, 2010

Contents

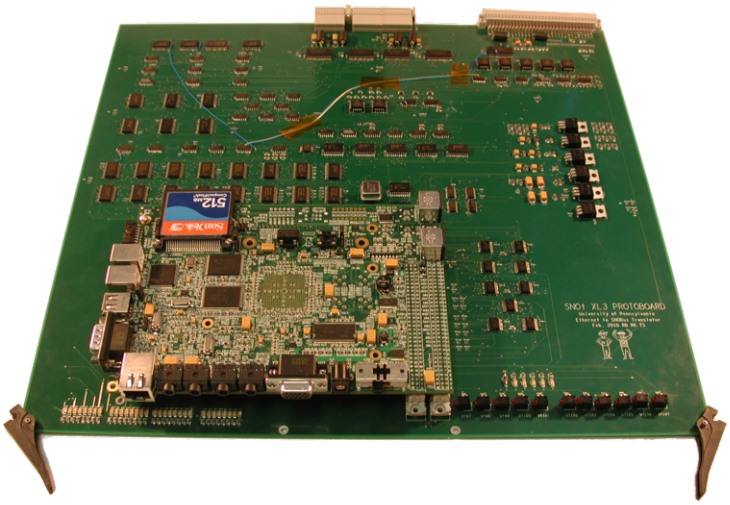

Figure 1: The Prototype XL3 board

1 Introduction

This section provides a short intro to why we made a XL3 in the first place, if you don't really care why we did it and just want to start using the XL3 or the Penn DAQ code, the skip ahead to Section 2 or Section 3 respectively.

The SNO electronics is a custom designed system, produced almost entirely by Penn. The system has 9728 channels, distributed over 300 Front End Cards (FECs) residing in 19 front end electronics crates. The communication protocol within each front end electronics crate is a custom SNOBUS interface, designed to be VME-like but with differential signals to reduce electronic pickup. Each channel records three charge measurements and one time measurement. These four channel level measurements are digitized and stored as 96 bits (12 Bytes) on local digital memory in each FEC.

Although the vast majority of the SNO electronics system can remain intact for SNO+, there will have to be changes to accommodate the increased data rates expected for the scintillating target compared to the original water Cherenkov target. In SNO, a typical solar neutrino event candidate illuminated about forty photomultiplier tubes. In SNO+, a neutrinoless double beta decay candidate will illuminate around 1500 photomultipliers, and on the average, the background from two neutrino double beta decay will illuminate half that number at a rate of several hertz.

During SNO data runs, a central data acquisition (DAQ) computer in a standalone VME crate polled each of the 19 front end electronic crates to check for available data. The memory of each FEC was read over the SNOBUS backplane. The translation from VME protocol to the SNOBUS protocol was done with pairs of custom-built translator cards. One of these cards, the XL1, sits in the central VME crate while the other, the XL2, sits in each front end crate. The translator cards had no local intelligence; all readout was done from the central DAQ computer and crates could only be read out one at a time, limiting the maximum data rate of the experiment. This bandwidth limitation was not a problem with a heavy water target as typical SNO data rates were a mere 16 kbit/s (2 kB/s), which rose to roughly 2 Mbit/s (250 kB/s) during data runs with calibration sources. At a sustained rate of 2 Mbit/s, the system was pushed to its limit, and often the appearance of a few noisy channels were enough to cause the buffers to fill and data to be lost, occasionally terminating the calibration run. For SNO+, the expected nominal data rate will be roughly 2.5 Mbit/s, too fast for us to reliably run the system. The readout electronics will therefore need to be upgraded to handle the expected increase in data rate. Additionally, since the trigger in SNO was an

anolog current sum, the expected data rate implies that the average trigger sum will be significantly higher in SNO+. For this reason the analog trigger, the MTC/A, is being upgraded as well. Details of this upgrade, the MTC/A+, are disscussed elsewhere (see the nubar wiki).

The XL1/XL2 translator pair has been replaced with a new custom crate readout card, the XL3. This card resides within each front end electronic crate and is capable of reading data from the FECs independent of any central DAQ computer. Data is now pushed, rather then pulled, to the DAQ over ethernet using standard TCP/IP protocols. All crates can push data independently and are connected via a standard switch. This increases the bandwidth of the experiment by at least a factor of 19, before accounting for the local readout speed itself.

To provide the ethernet interface for the new readout card, a commercially built board, the ML403 from Xilinx, acts as a daughterboard on the XL3. The ML403 has a Virtex-4 FPGA and an embedded PowerPC processor providing many options for the user. Inside the FPGA fabric of the Virtex-4, a simple state machine written in VHDL is configured to pull data across the SNOBUS backplane in each crate and store the digitized PMT signals in a local memory buffer. The additional memory on the ML403 daughterboard doubles the available memory in each crate. A small C-program, making use of open source Light Weight IP (LWIP) libraries, is run directly on the processor (no operating system) to send the data via TCP/IP to the central data acquisition computer. The Xilinx peripheral local bus (PLB) is used as the interface between the VHDL and the C-program. The 'hybrid' approach allows complete control over the SNOBUS side of the protocol, and the familiarity of programing in C, provides flexibility on the ethernet side.

The new XL3 board was designed by keeping intact most of the old translator schematics. The ML403 daughterboard is powered by the backplane of the front end crate and is attached by two connectors to the main board. The output of the board is contained to a single ethernet cable.

In late April 2010, the prototype XL3 was delivered and stuffed with components at Penn(see Figure 1). The rest of this paper documents how to communicate with the XL3 prototype in general and also how to run the Penn DAQ software. The production board is being built and no major changes to the software will need to be made. All the following details of communication with the prototype XL3 will be compatible with the production XL3. In section **??** any additional commands and details that pertain to the production board are explained.

2 The XL3 Manual

2.1 Loading new XL3 software

The great thing about the 'hybrid' design is that changes to the firmware (VHDL) and the software (C-code) can be changed easily. This section explains how to load new firmware and software onto the XL3. The source code for the XL3 software can be used as a reference. The src/ folder has all the C code, and the vhdl/ folder has all the FPGA code. All versions of the XL3 code are backed up on the nubar server at Penn in this location,

```
nubar:/proj/snoplus/sw/xl3/XL3_vX.YZ/
```

Where vX.YZ refers to the version number of the code. Version numbers with a 1.YZ are to be used with the prototype XL3 and version numbers with 2.YZ are for production XL3 (for example v1.08 was the last stable prototype version and 2.10 is the version for the prototype board as of December 2010).

The compact flash card on the ML403 is preloaded with an ace file that contains both the VHDL code for programming the FPGA as well as the C code that runs on the power pc. This code will run automatically every time you start up the board. If the code needs to be changed at all, either the new code has to be placed on the flashcard, or a J-TAG cable can download the new code.

New ace files can only be made with the Xilinx tools(**As of December 2010, only Penn has a licence to use the Xilinx tools**). To make a new ace file,

```
-Log into the computer with the Xilinx tools:
```

```
ssh neutrino@128.91.41.138
-Change into the directory that has the version of the XL3 code you want to load (ex. 1.07)
neutrino@neutrino:~$ cd XL3_v1.07
-Load up the environment:
neutrino@neutrino:~$ x101sp3
-Start a xmd session:
neutrino@neutrino:~$ xmd
-Run the command to make the ace file from the bit stream and the elf file
XMD% xmd -tcl genace.tcl -jprog -hw implementation/system.bit -elf code/executable.elf \
 -board ml403 -ace xl3.ace -start_address 0xFFFFFFFC
```

(It should be noted that there should/could be an executable file called "elf_and_bit_to_ace" that could be run instead of typing in the long command above.)

The name of the new ace file is "xl3.ace" and should be ∼3MB in file size. Once the new ace file is created it should be copied onto the compact flash card in this location, as this file name (Yes you overwrite the file called "system_bootload.ace"),

```
cp xl3.ace /media/ML403\ DEMO/ml403/bootload/system_bootload.ace
```

This is the ace file that the board runs on start up when the first three dip switches (see Figure 2) are set to the off position (which is the default). The compact flash card needs to be removed from the ML403 to have the new ace file loaded. One needs to be a little careful when removing and loading the compact flash card because the pins are somewhat fragile.

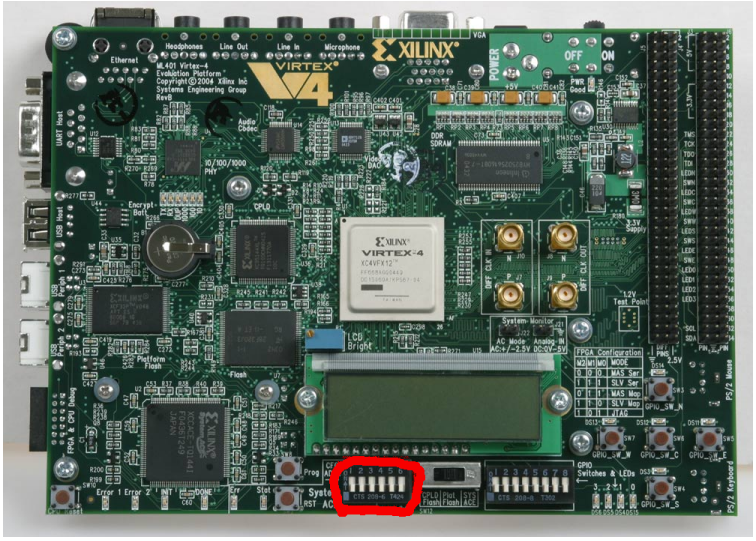

Figure 2: The ML403 evaluation board acts as a daughterboard on the XL3. The ML403 is set to boot off the compact flash card when the switch is set to System ACE. There are 2^3=8 locations in the compact flashcard memory to store an ace file. The dip switches, circled above, control which one of the 8 files is loaded on boot up. The dip switches should all be set to off (down in the orientation in the figure).

2.2 Booting up the XL3

When the XL3 is booted up, it will initialize itself, and then begin trying to connect to the server on a specific port and IP (port number == 6000+ crate number +1, such that the XL3 ports go from 6001 -6019 for the 19 different crates, and IP address == 10.0.0.0.[crate number + 1]). The XL3s are not individually

hardwired for a particular crate; they will read out the crate ID register on startup to determine what crate they are in and will set their IP accordingly (for example, the test stand at Penn is crate 2, so the IP address of a XL3 in the test stand is 10.0.0.3).

The XL3 will continue attempting to connect forever. Once it is connected it will never try to reconnect so if the connection is broken for some reason the XL3 will need to be rebooted to try and connect again. After connecting, the XL3 is in its initialization mode. This means that it will not do anything on its own, but will wait to receive packets with instructions on what to do. You can send it packets telling it to execute a command, or run crate_init, etc. You can also tell it to change to its normal running mode. At this point, it will begin automatically pushing any data it sees in the FECs out to the daq. If you still want to send it commands, you can fill up the command queue which it will execute as soon as it has a chance to, while still pushing out data words.

Most of the messages coming from the XL3 are shipped via ethernet (see Section 2.7), but there are some messages still sent over the com port. These messages are good for checking for ethernet related problems but for the most part all these problems have been solved (as of December 2010). These messages from the com port are not needed but if you want to see them, you will need a serial port on your computer to hook up to the UART port on the board. You will also need a serial terminal software program running on your computer("cutcom" on linux is a nice program). Set the baudrate to 19200, parity bit =none, data bits=8 and stop bits=1.

2.3 A fake FEC setup

This section is probalby only useful for Jarek and Richie

If one wants to test ethernet communication protocols only they can load up the FPGA with what we are calling a "fake FEC." A different ACE file is needed for the fake FEC setup. It can be found in the usual place and will be in a directory with a name like,

```
XL3_v1.03_ff/
```

The fake FEC setup simulates the XL3 connected to a very simple FEC. The same state machine that will run on the real XL3 is implemented, and all the output signals go to the io pins. But in addition, there is an additional state machine that mimics the most basic operation of the FEC or the actual XL3 board. Thus when you try to execute a read or write command, the fake FEC will receive the strobe, write or retrieve data from an internal register, and return dtack. This allows you to, if nothing else, write to a register and read back the same value as a check of the system. Many of the registers on the XL3 and the FEC are set up so they can be addressed correctly. When a write is executed correctly, the data returned is 0001ABCD.

If you want to test the way the XL3 will read data from the fecs during normal running, you can write to the data available registers and then switch the XL3 to normal running mode. In the current code (XL3_v1.03_ff), the data available register never clears, and so the XL3 will continue to think there is data and will read out from the fake FEC until you write 0 to the data available register. When the xl3 reads from FEC memory from the fake fec, the three long words it reads in are set to be first the 32 bit address you write to it, then two longwords that are each the fec number selected + "FECFEC". Commands to the XL3 or FEC require only two longwords, data and address. The address longword has the structure 31 30 29 28 27 26 25-20 19 - 0 spare spare memreg write* spare spare boardselect$< 5..0 > Addr < 19..0 >$ In source, registers.h has a more detailed map of the address space.

The packet structure that the XL3 expects is defined in lwip_functions.h. It is set up similarily as the SBC_Packet struct. For just doing one command, use 0x4. If you look at the function processBuffer in xl3_functions.c, you can see how each function expects the payload to be structured. For a single command, the XL3 expects the payload to be a FECCommand struct, which is outlined in queues.h. It will then return the same FECCommand struct, but with the data entry replaced with the value read in from the command.

2.4 Communicating with the XL3

This section is probalby only useful for Jarek and ORCA development The new packet structure is:

```
typedef struct
{
       uint16_t packet_num;
          uint8_t packet_type;
     uint8_t num_bundles;
} XL3_CommandHeader;

typedef struct
{
       XL3_CommandHeader cmdHeader;
          char payload[XL3_MAXPAYLOADSIZE_BYTES];
} XL3_Packet;
```

We have included a little bit more information in the new command header. We made it so that the total size was still the same so the overall size of the packet is unchanged. Packet_num is an incrementing number that identifies a particular packet. Packet_type is what was previously called cmdID, and tells the XL3 what kind of packet it is receiving. num_bundles just tells the DAQ how much data is in the packet.

Up to this point we haven't been actually numbering our packets, so we attempted to come up with a scheme that will work for us. In our current plan, the DAQ numbers all packets it sends to XL3s. There can be a seperate numbering for each XL3 or one overall count, it doesnt matter. Most packets that the XL3 receive call for an immediate response. For example, when you send a packet telling the XL3 to do one register read/write, you will get a response with the result of the read/write; When you send a "crate init" packet, you get a response telling you if there were errors or not. These response packets are basically echos of the packet the DAQ sent, so the XL3 sends them back with the same packet_type and the same packet_num. So if the DAQ wants to read the test register, it will send a packet with packet_type "single register read" and lets say packet number 5. It will then expect a packet back from that XL3 also with packet_type "single register read" and packet_num = 5.

In addition, there are other kinds of packets that the DAQ should expect to get from the XL3, either during a run or during golden tests etc. Since these originate from the XL3, they are numbered by each XL3. So the DAQ will need to keep track of the numbering of these kinds of packets on an XL3 by XL3 basis. Examples of these kinds of packets are pmt bundle packets and messages for the daq to print out to the screen.

I'm not sure whether the ORCA code ever uses the XL3 command queues or actually needs the FEC-Command struct, but if it does, it has also changed as follows:

```
typedef struct
{
     uint32_t cmd_num;
      uint16_t packet_num;
       uint8_t flags;
        uint32_t address;
 uint32_t data;
} FECCommand;
```

Currently the way that the command queue works is that several kinds of packets from the DAQ can tell the XL3 to fill the command queue with a variable number of commands. The XL3 then executes each command whenever it can, and asynchronously pushes the results back to the DAQ. The XL3 groups the results of commands into packets on its own, so that it is necessary to be a little more careful keeping track

of commands. For example, if packet 5 comes to the XL3 and queues up 40 commands, and then packet 6 comes in and queues up 120, the XL3 will send the results back in groups of 80 so that the first packet the DAQ receives will have results of commands from both packet 5 and packet 6, and the results of commands from packet 6 will be spread out over two different packets. In order to make the DAQs life easier, when each command is queued on the XL3, it now carries with it the packet_num of the packet that created it, so that when the DAQ receives 80 command results in a packet, it can check each result to see which of the packets it sent to the XL3 it was created by. To make sure that no result is lost, the commands from each packet are then numbered, which is the cmd_num in the struct. So if packet 5 queued up 40 commands, they would all be in the queue with packet_num = 5 and cmd_num = 0-39. The DAQ can then check both of these values in the results to see that it has gotten as many as it expects back.

This may become important in new routines since doing register reads and writes the old way (one packet per read/write) is slower than it was before after rewriting some of the XL3 code to increase stability. It may become necessary to rewrite daq functions to more efficiently use packets, which may involve queueing up many commands at a time.

In newer versions of the XL3 code we have included the ability to ping the DAQ and determine when it has disconnected. In our version of the code, the XL3 pings and the DAQ pongs. If the XL3 gets no pong back after a certain amount of time, it closes the connection and starts trying to reconnect. The DAQ can also detect if the XL3 has been disconnected if it hasn't got a ping in a long time. All the DAQ needs to do for now is to expect periodically to get a "ping" type packet, and when it does, to send back a "pong" type packet (the XL3 will not respond to the pong packet).

2.5 Hardware, IPs, Ports and whatnot

To use the XL3 with either ORCA or PENN_DAQ, 4 pieces of hardware are needed,

1. A XL3
2. A SBC that communicates with the MTC/D
3. A DAQ computer (running linux or a mac with a intel chip)
4. A way of rebooting the XL3 via J-TAG, power cycle, reset button, Slow Control(not yet implemented)

The first three items above need to be connected on a local LAN network. The XL3 is preprogrammed to be on a 10.0.0.0 network. This is easy to change (src/main.h) but the Xilinx tools are needed. The default IP addresses are:

1. XL3 10.0.0.[crate_num +1]
2. SBC 10.0.0.20
3. DAQ 10.0.0.21

Below is the setup being used on the test stand at Penn. There are two options for the DAQ computer. The Mac_box running ORCA or the laptop running PENN_DAQ.

```
Mac box:
outside IP: 128.91.45.47
user    hep
passwd  XXXXXXXX
Internal IP: 10.0.0.21

SBC:
no outside IP:
user: daq
```

```
passwd XXXXXXXX
internal IP: 10.0.0.20

XL3:
no outside IP:
no user
no passwd
internal IP: 10.0.0.3  (crate_num+1 since ip of 0 is not allowed)

Penn DAQ laptop:
outside IP: 128.91.41.138  (this may change in the future!)
user: neutrino
passwd: XXXXXXXX
no internal IP: (we took it off the local network because the XL3 can only
connect to one DAQ at a time.)
If we use the PENN_DAQ we set the internal IP to the same as the
Mac_box 10.0.0.21
```

The XL3 in the test stand at Penn is set up so that people can log in (Jarek mainly) into the Penn DAQ laptop to reboot the XL3 whenever you want to. The XL3 is hooked up by J-TAG to the XL3.

```
ssh neutrino@128.91.41.138

neutrino@neutrino:~$ cd XL3_v1.03
neutrino@neutrino:~$ x101sp3
neutrino@neutrino:~$ xmd -xmp system.xmp -opt etc/xmd_ppc405_0.opt

XMD% dow a              (downloads the code to the Xilinx)
XMD% con                (starts the Xilinx code)

to stop the XL3, or if you want to restart it

XMD% stop

Then redownload the code (a is a soft link to code/executable.elf )
and restart it ( the "con" command )

Once the XL3 starts (after the con command) it will try to connect to a
DAQ at IP 10.0.0.21 on port 6003 (6000 +crate_num+1).
```

2.6 The readout loop

Once the XL3 is put into normal readout mode, it will continually poll the FECs for data and push anything it sees to the DAQ. Each time through its loop, it will check the status of the FECs, its own internal queues for data and commands, and any other alerts or status flags. It will then decide one action to execute based on the status before looping around again.

If no special flags are set and no commands are pending, the XL3 will go through its standard readout procedure. It first reads the data available register, and stores this value in a global variable. It then checks the fifo pointers to get the memory level of each FEC. At this point it also sets the GTID limit, as will be described later. It then goes through the FECs in order. Starting with the first one with data available high,

it reads out up to 120 bundles in a row, pushing each bundle into the pmt data out queue. The number read is meant to match the number of bundles that fit in a packet. It then masks off that bit in the global variable, and continues on to the next FEC with data available high. Once all the bits high in the global variable have been masked off, it starts back at the beginning, and checks the data available and memory levels again.

In order to let the event builder know when a certain event is finished, each time it restarts the readout loop the XL3 sends a packet containing just the maximum number of reads before the bundles currently in memory are completely read out. ORCA then periodically gets the GTID from the MTCD, and the event builder can combine this information to determine how much data to buffer.

Normally, the XL3 will decide to send out a PMT bundle packet as soon as there is 120 bundles in the queue. However, if there is a large burst of hits and the memory level of the FECs starts increasing faster than we can read out, the XL3 can shift its focus. After a certain point, it will stop sending packets and spend 100% of its cpu time reading out the FECs, using its own 64Mb of ram as an addition buffer. If its own buffer also begins to fill up, it will send an alert of an overflow, and will restart sending packets.

2.7 Messages from the XL3

All packets out of the XL3 that arent data or FECCommand results go through the message queue. These packets are generally either a response packet to a general ORCA command, or an ASCII message for the user. Response packets are pushed into the queue and are sent out as is from the main loop. ASCII messages are enqueued using the mq_printf command. To avoid sending too many packets with very short messages, strings are first buffered and then combined to get single strings as close to the maximum packet size as possible before being pushed into the message queue in a packet. Since you dont want messages getting stuck in the buffer if they arent long enough, one of the arguments for mq_printf allows you to manually flush the ASCII message buffer and immediately push the string into the message queue.

2.8 XL3 Commands

2.8.1 Change Mode

Set the XL3's mode. Packet payload is: uint32_t mode, uint32_t slot_mask. There are three possible modes: INIT, NORMAL, CGT. The XL3 starts in init mode. In this mode the readout loop skips over checking the data available lines and so will not read out any data from the FECs. In this mode it is safer to do longer commands on the XL3. In normal mode, the data readout loop functions as normal. CGT mode is used during the global trigger tests. The readout loop is slightly modified, so that instead of pushing bundles into the data out queue, the bundles are instead parsed to check for GT errors, and messages are enqueued when errors are found.

2.8.2 XL3 Test Cmd

This command is used to run diagnostic tests. It is mostly deprecated at this point.

2.8.3 Single Cmd

This command does one register read / write and then immediately returns the result. Packet payload is: FECCommand command. Uses the message out queue and not the cmd ack queue.

2.8.4 DAQ Quit

Stops the main loop and closes the connection.

2.8.5 FEC Cmd

Queues up commands in cmd queue. Safest way to do commands in NORMAL mode since it does not interupt data reading as much. Packet payload is: MultiCMD commands.

2.8.6 FEC Test

Tests the FEC registers. Packet payload is: uint32_t slot_mask.

2.8.7 Mem Test

Tests the FEC memory. Packet payload is: uint32_t slot_mask.

2.8.8 Crate Init

Initializes the FECs and loads the xilinx chips. First 16 packets send the database values, with packet payload: uint32_t mb_number, mb_const_t mb_constants. The 17th packet starts the crate initialization, and has packet payload: uint32 mb_number, uint32 xil_load, uint32 hv_reset, uint32 slot_mask, uint32 ctc_delay, uint32 shift_only. If it doesn't recieve all 16 database packets before it gets the 17th packet, the crate init will fail.

2.8.9 VMon Start

Reads out the voltages on the FECs. Packet payload is: uint32 slot_mask.

2.8.10 Board Id Read

Reads the ID registers of the motherboards, daughterboards, and HV cards. Packet payload is: uint32 slot_mask, uint32 chip_num, uint32 reg_num.

2.8.11 Zero Discriminator

This command reads the noise rates for a FEC, then changes the DAC values until the discriminator rate is lower than your accepted rate. Packet payload is: int slot, uint32 offset, float rate.

2.8.12 Fec Load Crate Add

Loads the crate address into the general CSR register for each FEC. Packet payload is: uint32 slot_mask, uint32 crate_num.

2.8.13 Set Crate Pedestals

Sets the pedestal enables on all the FECs to pattern. Packet payload is: uint32 slot_mask, uint32 pattern.

2.8.14 Deselect FECs

Sets the address lines of the ML403 to zero so that the FEC selects are all off.

2.8.15 Build Crate Config

Checks each FEC to see if it is connected, then reads in the ID registers.

2.8.16 Loadsdac

Loads a particular DAC in a FEC. Packet payload is: uint32 slot_mask, uint32 theDac, uint32 theDacValue.

2.8.17 Cald Test

Tests the ADCs using the calibration DAC. The DAC value is ramped from lower to upper. Packet payload is: uint32 slot_mask, uint32 num_points, uint32 samples, uint32 upper, uint32 lower.

2.8.18 State Machine Reset

Resets the FPGA state machine.

2.8.19 Multi Cmd

Like Single Cmd, executes commands immediately and then immediately returns the results bypassing the cmd in and out queues. Can do more than one command. Packet payload is: MultiFC commands.

2.8.20 Debugging Mode

Enables or disables print out to serial port. Packet payload is: uint32 on/off*.

2.8.21 Read Pedestals

Pushes num_read memory reads into the cmd in queue. Packet payload is: uint32 slot, uint32 num_read.

2.8.22 Pong

Response to XL3 ping.

2.8.23 Multi Loadsdac

Calls loadsDac several times. Packet payload is: int num_dacs, [uint32 slot_mask, uint32 theDac, uint32 theDacValue] x num_dacs.

2.8.24 Load Tac Bits

Loads the TAC bits into the shift register. Packet payload is: int crate, uint32 select_reg, uint16 tacbits[32].

2.8.25 Cmos GTValid

?

2.8.26 Reset Fifos

Resets the FEC FIFO. Packet payload is: uint32 slot_mask.

2.8.27 Set HV Relays

Clocks in 64 bit mask into relay. Packet payload is: uint32 buffer1, uint32 buffer2.

2.8.28 Get HV Status

Read out the HV voltage and current.

2.8.29 Read PMT Current

Reads PMT base currents from FEC voltage monitor. Packet payload is: uint32 slot, uint32 channel_mask.

2.8.30 Slot Noise Rate

Calculates the noise rate in a given FEC. Packet payload is: int slot, uint32 channel_mask, int period.

2.8.31 Setup Charge Inj

Sets up a FEC for charge injection. Packet payload is: uint32 slot, uint32 channel_mask.

3 The Penn DAQ Software Package

3.1 Needed software and install instructions

To use the PennDAQ to talk to the XL3 and MTC/D there are four items needed.

1. **penn_daq** - the server program that communicates with the user, the XL3 and the SBC
2. **tut** - a telnet like program that acts as the user interface to the server program
3. **OrcaReadout** - Code running on the SBC that handles the MTC/D communication
4. **ML403 code** Only needed as reference and only if the Xilinx tools are available can modifications be made.

A tarball of the latest version of all the software can be obtained by emailing either Rob or Richie. All the software is also backed up at Penn on the nubar server.

```
nubar:/proj/snoplus/sw/xl3/ORCA_dev/OrcaReadout
nubar:/proj/snoplus/sw/xl3/select_DAQ/daq/tut
nubar:/proj/snoplus/sw/xl3/select_DAQ/daq/mac_daq  (yes the PennDAQ is still called maq_dac FIXME)
nubar:/proj/snoplus/sw/xl3/XL3_v1.08/
```

Also all the IP address are hard coded. By default they are setup as in Section 2.5 but can be changed if needed (if how to change the IP's is not explained in the following subsections then email Rob if needed).

3.1.1 OrcaReadout

To install **OrcaReadout**, copy the ORCA_dev directory into the home folder on the SBC. Change into the ORCA_dev directory and typing 'make' should install it just fine. To run the program just type ./ORCA_dev/OrcaReadout

3.1.2 tut- telnet like client

To install **tut**, change into the select_DAQ/daq/ directory and invoke the gcc command.

```
cd select_DAQ/daq/
gcc -l readline -o tut tut.c
```

This software has not been tested on different platforms, so I hope it complies!

3.1.3 Penn_DAQ (called maq_daq still! FIXME)

An environment variable needs to be set,

```
export PENN_DAQ=/wherever/select_DAQ
```

In the select_DAQ/ directory there are many subdirectories and one makefile,

```
api/
dbio/
dispatch/
include/
Makefile
db/
db_utils/
snodb/
lib/
sys_util/
daq/
```

in many of these directories there are "Makefiles." Each makefile has to be made individually. The Makefiles need to be complied in the order listed below,

```
api/Makefile
dbio/Makefile
dbio/db_com/Makefile
dbio/tech_files/Makefile
dbio/tech_files/back/Makefile
db_utils/Makefile
snodb/Makefile
snodb/glue_files/Makefile
sys_util/Makefile
daq/Makefile
```

There is no script setup to do all this yet, but all this compilation only needs to happen once. Once all the makefiles in the subdirectories are made you can run the executable in the select_DAQ/daq directory called mac_daq.

```
cd select_DAQ/daq
./mac_daq
```

3.2 Running PennDAQ

PennDAQ is a set of programs that, together, form a simple DAQ program for the SNO+ experiment. Currently is can be used to run all the golden and silver tests (see Section **??**). In theory it could be used to record data but this has not been tested nor should it be. The code is based off of code by Paul Keener and friends written for the original SNO experiment, It has been heavily modified by Rob, Richie and Peter Downs. Unlike the code that ran on SNOPenn101 which communicated with the XL1 and SBC via a VME link, PennDAQ uses ethernet for all communications.

The program, still called mac_daq.c for legacy reasons that no one remembers, is a stream socket server. If this means nothing to you, I highly recommend reading: http://beej.us/guide/bgnet/ Seriously, go and read it. Quite an excellent guide. The majority of the communications in mac_daq are based off of his examples.

mac_daq.c is a server which accepts stream connections from 4 different types of clients:

1. Controller client - this client sends commands which are then parsed and executed. This client can be a remote computer or the same computer that the server (mac_dac.c) is being run on.

2. Single Board Computer / Master Trigger Controller (SBC/MTC) - a board which can be written to and have commands done to it, it also can be polled for data

3. XL3 board - an update of the XL2 board used in SNO, the XL3 reads data from the 16 FECs in each crate and sends it to the server.

4. View client - a client from which only data is received. No commands can be run from a viewer client. The idea is that any time there is a print statement in mac_daq, or data is displayed on screen, that text is also be sent to the viewer. This way people can spy on what you are doing!

To set up the PennDAQ there are four (yes 4!) programs that need to be set up. Some GUI or shell scripts could be made to stream line this but as of yet (December 2010), this has not happened. First, there is the actual DAQ server program running on your computer (mac_daq, but this should be called penn_daq). Second, there is a telnet replacement client that allows you to communicate to the DAQ (tut). Third, there is the readout code under ORCA_dev on the SBC in the VME crate (OrcaReadout). Finally, there is the XL3 code run on the ML403.

Once everything has been compiled on your system (see Section 3.1) you can begin to run PennDAQ. To start up the DAQ, open up a terminal and change into the $HOME/select_DAQ/daq/ directory and run mac_daq.

```
cd select_DAQ/daq/
./mac_dac
```

and then tut. penn_daq should automatically recognize the control client and connect to it. Then, ssh into the SBC and run OrcaReadout. At this point you can use the command "connect_to_SBC" in tut to connect penn_daq to OrcaReadout. Finally, if the XL3 has not connected automatically, reload the C code and restart the XL3 as explained above. The XL3 should automatically connect to penn_daq.

3.3 PennDAQ commands

This list will somewhat overlap with the list above, but is included for completeness.

3.3.1 print_connected

Prints a list of connected control clients, XL3s and SBCs.

3.3.2 stop_logging

Stop logging all output to a file.

3.3.3 start_logging

Start logging output to a file.

3.3.4 debugging_on[off]

Turn on XL3 output over serial port.

3.3.5 change_mode

Change the XL3 mode turn on or off data readout.

3.3.6 speed_test

Obsolete?

3.3.7 readout_test

Enables pedestals and gt pulser, then puts XL3 into normal mode to test continuous readout speed.

3.3.8 end_readout

Remove? Changes mode to init mode and disables pedestals.

3.3.9 start_pulser

Sets up the pulser on the MTCD.

3.3.10 stop_pulser

Turns off the pulser.

3.3.11 change_pulser

Change the frequency of the pulser.

3.3.12 test_func

Obsolete?

3.3.13 mtc_init

Initialize the MTCD.

3.3.14 ped_run

This runs the pulser for a short amount of time, then reads in and analyses the results to check that the Qh/l/s/x, TAC, and that the channels and cells are all working.

3.3.15 crate_init

Initializes the FECs. It then writes the new crate configuration (which cards are where) into the database.

3.3.16 fec_test

Checks reading and writing to all the FEC registers.

3.3.17 mem_test

Checks reading and writing throughout the FEC memory.

3.3.18 vmon

Reads out all the voltages and temperature of the FEC.

3.3.19 board_id

Reads the id numbers of the MBs and DBs and HVCs.

3.3.20 crate_cbal

Tries to balance the QHs and QHl signals by adjusting the balance DAC and then reading out pedestals.

3.3.21 spec_cmd

Does one command immediately.

3.3.22 add_cmd

Adds one command to the command queue.

3.3.23 zdisc

Zeroes the discriminator as described above.

3.3.24 cgt_test_1

Checks that the GT bits are correct on the FECs. Puts the XL3 in cgt mode, then sends several soft gts. The XL3 keeps track of what GT it should be reading out from each channel and cell. If at any point there is an error, the XL3 sends a packet alerting the DAQ. If the DAQ doesnt recieve anything from the XL3, it assumes the test is going ok. With the quick option turned on (-q), will check the first 100 triggers, then will skip ahead to the 2^{16} rollover, and check that nothing happens around there.

3.3.25 cmos_m_gtvalid

This adjusts the gt delay channel by channel to ensure it falls in the gtvalid window. It adjusts DACs and twiddle bits and then takes some pedestals, checking to see when channels are no longer getting the trigger.

3.3.26 send_softgt

Sends one global trigger, as long as the pulser is enabled and set to source soft_gt (set frequency = 0).

3.3.27 read_bundle

Reads in one bundle from memory and parses it.

3.3.28 cald_test

Tests the ADCs using the calibration dac.

3.3.29 change_delay

Changes the GT delays.

3.3.30 sm_reset

Resets the FPGA state machine.

3.3.31 connect_to_SBC

Connects to the MTCD.

4 Current Known Bugs

1. Attempting to reconnect to the SBC will cause a seg fault in penn_daq
2. Attempting to run mtc_init twice in a row or running certain commands immediately after running mtc_init (like readout_test) will cause a seg fault.
3. Loading the xilinx to the mtcd will cause a lot of garbage to be printed out to penn_daq, can be safely ignored.

4. After running crate_cbal, the next time the pulser is set to a frequency instead of sourcing the software gt, it will briefly hold high flooding the FEC memory with garbage. Turning the pulser on and off briefly and then wiping the memory fixes this.

5. Sometimes when mem_test fails, it tries to send too many error messages to penn_daq and crashes, so if mem_test hangs for longer than 10 seconds, you should check to see if you forgot to plug in ram to your FEC. This should be fixed soon.

6. There is still no root replacement for fit_caldac, will be coming shortly.

7. Ped_run will always give a few extra events in each cell since timing the pulser is less accurate with packets.

8. Cmos_m_gtvalid takes FOREVER... get a cup of coffee or something (∼30 mins per FEC).

5 Running Silver and Golden tests

Penn_daq should now be sufficient for testing motherboards and daughterboards. We have tried to make everything as similar to the old sun tests as possible, so except for a few bugs and syntax changes, it should be the same. Here is the complete list of silver and golden tests for one FEC in slot 7 in crate 2:

```
Silver:
mtc_init -f
crate_init -c 2 -s 80 -x
fec_test -c 2 -s 80
mem_test -c 2 -s 7
crate_init -c 2 -s 80 -x
vmon -c 2 -s 80
board_id -c 2 -s 80
ped_run -c 2 -s 80

Golden:
crate_cbal -c 2 -s 80 -k 666
## not part of the test, just a bug fix:
start_pulser -c 2 -f 1
stop_pulser
## end bug fix
crate_init -c 2 -s 80 -1 666
ped_run -c 2 -s 80
zdisc -c 2 -s 80 -o 0 -r 100 -k 666
crate_init -c 2 -s 80 -1 666
mtc_init
cgt_test_1 -c 2 -s 80 -l FFFFFFFF
crate_init -c 2 -s 80 -1 666
cmos_m_gtvalid -c 2 -s 80 -g 400 -k 666
crate_init -c 2 -s 80 -C 113 -x
cald_test -c 2 -s 80
```

6 Interfacing with ORCA

Step 1: Email Jarek.
Step 2: Wait for him to do everything else.

7 Using the Xilinx Tools

This needs to be recorded so we don't forget how to do this stuff!!! Screen shots, samples and everything else would be helpful!

7.1 Installing ISE and EDK in linux

7.2 LWIP and the PLB bus