Creating Effective Readout Software

5th Edition

XL3

The Definitive Guide

O'REILLY®

Chuck Musciano & Bill Kennedy

XL3 and Penn DAQ Programmer's Manual

Richie Bonventre

Don't have any questions. It will just work, ok?

Contents

1 Introduction to the XL3

The SNO electronics is a custom designed system, much of which was produced by Penn. The system has 9728 channels, distributed over 304 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 analog signals are digitized and stored along with various ids and error information in 96 bits (12 bytes) on local digital memory on 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 running with scintillator compared to the original water Cherenkov mode. 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 pmts, with backgrounds at similar energies occurring 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 electronics crates serially to check for available data. The memory of each FEC was read over the SNOBUS backplane. The translation from VME to the SNOBUS protocol was done with pairs of custom-built translator boards, called XL1 and XL2. The XL1s sit in the central VME crate and the XL2s sit in the front end crate. These boards had no local intelligence; all readout was done from the central DAQ computer and so 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 heavy water as typical SNO data rates were a mere 16 kb/s (2kB/s), which rose to roughly 2Mb/s (250kB/s) during data runs with the highest rate calibration sources. At this rate, the system was pushed to it's 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 2.5Mb/s, too fast for us to reliably run the system. The readout electronics thus needed to be upgraded to handle this increased rate.

The XL1/XL2 translator pair has been replaced with a new custom crate readout board, the XL3. This board resides within each front end crate where the XL2 use to be and is capable of reading data from the FECs independent of any central DAQ computer. Data is now pushed, rather than 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 parallelization increases the bandwidth of the experiment by a factor of 19, even before accounting for the increase in the local readout speed.

To provide the ethernet interface for the new board, a commercially built evaluation 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, a state machine written in VHDL is configured to pull data across the backplane using the SNOBUS protocol. A small C-program using the open source Light Weight IP (LWIP) libraries is then run on the PowerPC with no operating system to pull the data from the FPGA across the Xilinx peripheral local bus (PLB) and store it in local

RAM memory. LWIP is used to implement the tcp/ip stack and send the data to the DAQ. This hybrid approach allows fast direct control of readout with the SNOBUS protocol as well as the ease of C programming to allow flexibility on the ethernet side.

The XL3 board was designed by keeping intact much of the old XL2 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.

2 Setting up Xilinx tools

2.1 Installation

The XL3 code was developed in EDK version 10.1 service pack 3. To install, you will need to download ISE version 10.1, EDK version 10.1, ISE 10.1 service pack 3 upgrade, and EDK 10.1 service pack 3 upgrade. You will need a license key for both ISE and EDK. Make sure to install ISE before EDK. If you are on linux, make sure you uncheck the box for installing the cable drivers. Once installation is finished, you will need to find xps_ll_temac_soft_core_permanent_eval.lic and place a copy in the path to xilinx install/10.1/EDK/data/core_licenses directory.

If you are on windows, you should be pretty much done with the installation. For linux users, we will next need to configure some libraries. First find what version of libdb4.* you have in /usr/lib, then do

```
sudo ln -s /usr/lib/libdb-4.* /usr/lib/libdb-4.1.so
```

Next, we need to fake the gmake utility

```
sudo ln -s make /usr/bin/gmake
```

If you don't want to use acrobat reader, or dont have it installed,

```
sudo ln -s evince /usr/bin/acroread
```

Now we will need to install a bunch of other dependencies:

```
sudo apt-get install libusb-dev libstdc++5 build-essential gtkterm
sudo apt-get install fxload portmap libmotif3 libmotif-dev rlwrap
```

Now we need to get something to replace the cable drivers so the jtag cable will work. We will use an alternative version available via git

```
git clone git://git.zerfleddert.de/usb-driver
```

Move to the directory created and type make. It should create a file called libusb-driver.so copy that file to

```
{path to xilinx install}/10.1/usb-driver/.
```

We will next set up a bash command to configure your development environment. In your .bashrc, add

```
x101sp3() {
local xil_home
xil_home = {path to xilinx install}/10.1
. $xil_home/ISE/settings64.sh
. $xil_home/EDK/settings64.sh
export PATH="$xil_home/EDK/gnu/powerpc-eabi/lin/bin:$PATH"
export PATH="$xil_home/EDK/gnu/microblaze/lin/bin:$PATH"
export XIL_IMPACT_USE_LIBUSB=1
export DISPLAY=":0"
alias inserter='AWT_TOOLKIT=MToolkit inserter.sh"
alias analyzer='AWT_TOOLKIT=MToolkit analyser.sh"
}
```

Now open a new terminal or source your bashrc and type x101sp3. You can now set up the usb driver by moving to the path to xilinx install/10.1/usb-driver/ directory and doing

```
sudo ./setup_pcusb
sudo ./setup_pcusb
```

The first time you execute this command you may get some error messages, but you can ignore them and just execute it a second time. This time it should work fine. Everything should now be configured, so once you enter x101sp3 in a terminal, you should be able to run ise, edk, xps, impact, etc.

2.2 Configuring the ML403 Project

When creating a project for the ML403, the architecture is virtex4, device size is xc4vfx12, package is ff668, and speed grade is -10. Once the project is created, go to Software-¿Sofware Platform Settings and in the Software Platform tab, check the lwip130 box to include the lwip library. In the OS and Libraries tab, under lwip130 make sure that api_mode is set to RAW_API.

Create a new Sofware Application Project, right click it and select Compiler Options. Check the "Use Custom Linker Script" box and select the linker script included in the ml403_code repository. This sets up the memory spaces on the RAM and ensures that space is reserved for our fifo queues. Under the Debug and Optimization tab, set the Optimization Level to "High (-O3)", and if you would like to do debugging, check the Generate Debug Symbols box. Finally in the Paths and Options tab, make sure that "-llwip4" is in the Libraries to Link against box. You can now add the source and header files to the application.

You will need to assemble the hardware IP cores used in the project. Click on System Assembly View and then the Bus Interfaces tab on the right hand window and the IP Catalog

in the left. First, add the buses. There should be one proc_sys_reset IP, one jtagppc_cntlr IP, and three plb_v46 IPs. Add a ppc405_virtex4, and expand it to set up its Bus Connections. RESETPPC should be connected to the reset bus, JTAGPPC should be connected to the jtag bus. IPLB0 and DPLB0 should be connected to one plb bus, and IPLB1 and DPLB1 should each be connected to one of the other two. The first bus will be used to talk to most of the various peripherals, while the second two are used only for input and output from the RAM. Next add a mpmc IP, which is the ram controller. Connect DPLB1 to SPLB1 and IPLB1 to SPLB0. We will also use faster bram which exists in the FPGA, so create a bram_block and then an xp3_bram_if_cntlr and connect it to the first plb bus and to PORTA on the bram_block. Next add an xps_intc which allows the peripherals to interrupt, and connect it to the plb bus. An xps_uartlite IP allows you to printf to the serial port, and also needs to be hooked up to the plb bus. Right click it and select Configure IP to set the baud rate and parity. For ethernet, create an xps_ll_fifo and an xps_II_temac. Link them together and connect them both to the plb bus. Finally, add a clock_generator IP.

You will now need to import the custom IP that contains the VHDL state machine. Go to Hardware-¿Create or Import Peripheral. Click Import Existing Peripheral, To an XPS Project, and then enter the name of the file in the pcores directory of the ml403_code repository. Once the IP has been imported, add one to the project and connect it to the plb bus.

Now we will set up the port connections. Click the Ports tab in the right hand window. The powerpc should have a signal from the interrupt IP to EICC405EXTINPUTIRQ, and a signal from the clock called proc_clk_s to CPMC405CLOCK. The plb buses should all have a signal from the reset IP to SYS_Rst, and a signal from the clock called sys_clk_s to PLB_Clk. The RAM should have many signals that are hopefully auto configured since I will not be detailing their configuration here. It should also have a reset signal and three clocks: one to MPMC_Clk_200MHz, one to MPMC_Clk90, and one to MPMC_Clk0. Again I hope the temac is auto configured. The custom IP has many ports that need to be created. Ext_Clock should be connected to a clock signal like sys_clock_s. For Dtack, Gated_strobe, V_all_ok, Write_star, Memreg, Board_Select, Addr_Bus, Data_Bus, Leds, ML403_Happy, and XL3_Alarm, click the select box and pick “Make External”, which will automatically create an external port that you can assign to a pin on the FPGA. We can now set up the clock frequencies since we have made our clock signals. Back in the Bus Connections tab, right click the clock IP and go to configure IP. The Input clock CLKIN should be connected to dcm_clk_s and should be set to 100,000,000 Hz. The output clocks are sys_clk_s (100 MHz), proc_clk_s (300 MHz), DDR_SDRAM_mpmc_clk_90_s (100 MHz), clk_200mhz_s (200 MHz), and temac_clk_s (125 Mhz). The DDR_SDRAM_mpmc_clk_90_s should also have a phase shift set by clicking the Ports Overview tab and entering 90 into the Phase Shift field.

Next click the Addresses tab in the right hand window. Set the state machine Size to 64K, and the bram to 8K. The interrupt, xps_ll IPs, and uart should all be set to 64K, the powerpc to 256, and the RAM to 64M. Then click Generate Addresses in the top right.

Now in the Project tab in the left hand window, double click the UCF File to open it up in the editor. You will want to base it off the ucf file included in the ml403_code repository,

changing any port names that are different to match.

Finally, check in xl3_utils.c where the fifo queues are allocated and make sure they are properly placed in memory that has been allocated to the RAM in the linker script and in the Addresses window in the System Assembly View, and check that the peripheral address is the same as that for the custom IP.

3 XL3 Hardware

Much of the XL3 hardware is the same as the old XL2. A description of the XL2 hardware and a copy of the schematics can be found at `http://nubar.hep.upenn.edu/snoplus/DAQ/`. The XL3 schematics should also soon be available at the same location.

The XL3 hardware can be broken up into five main parts, as shown on the first page of the schematic.

The slow control interface is the simplest block. It is merely a connector for a cable to the slow control system. Currently there are three signals used by slow control. The ML403_RESET comes from the slow controls to the XL3 and is a 5V TTL logic line. When this line is driven high, the power to the ML403 is disconnected. When it is then brought low, power returns, and so this can be turned on for a few milliseconds to reset the ML403. V_ALL_OK and TEMP_ALL_OK are also 5V TTL logic lines that go from the XL3 to the slow controls. They are high if the voltages and temperatures are ok, and drop low when there is a problem. It is not latched so it will remain low only for as long as the problem remains.

The ML403_BLOCK shows the connection to the evaluation board. There are two 3x32 pin connectors. Each connector has one row that is all connected to ground. One has the whole 32 bit data bus, and the other has address and control lines. In addition, one connector has 4 VCC pins. These are directly connected to the power lines on the ml403, and so all the current to power the evaluation board goes through these pins. The AP280 ic connected to this line is a switch that connects or disconnects these pins to VCC of the rest of the XL3 based on the ML403_RESET signal.

The XILINX_INTERFACE block is where the signals are converted from the LVCMOS 3.3V signals used on the ML403 to the 5V TTL logic used for the rest of the XL3. The data lines are bidirectional with the direction controlled by the WRITE* signal. The address and control signals go from the ML403 to the XL3, and the alarms go from the XL3 to the ML403.

The SB_BUS_INTERFACE block is almost identical to what it was on the XL2. The GTL16612s translate from the 5V TTL logic of the XL3 to the GTL logic levels used on the SNOBUS backplane. Again the address and control lines go one way from the XL3 to the backplane. The data lines are sent from the XL3 to the backplane when WRITE is high and GATED_STROBE arrives. The lines are opened from the backplane to the XL3 when WRITE is low and GATED_STROBE arrives, and they are latched when LEBA (latch everything B to A) arrives. The DATA_AVAILABLE lines and DTACK from the backplane are always enabled from the backplane to the XL3.

The relay signals, the clocks, the xilinx programming lines, and the reset also all come from XL3 and are sent to the backplane. The powers for the XL3 - VCC, VEE, VP8, VP24, VM24 - all come from the backplane, and from these lines V33P, V08P, VGTLTERM, V2M, V15P and V15M are generated. VGTLTERM and VGTLREF (V08P) are sent back to the backplane to make the GTL lines work.

The XL3_REGISTERS block is subdivided into the different hardware registers on the XL3. In the CSR_REG, we have the xilinx programming lines, the crate address lines, and the logic reset. The Xilinx chips on the FECs are programmed over four lines. Data in, clock, and xilinx active are all one way signals from the XL3 to the FEC and are converted to the backplane voltage levels with HCT125s. There is a maximum allowed low time for the clock signal, so it is sent through a one-shot to get low going pulses and is pulled high after the HCT125. The Done/Program* line is a wired-AND signal (can only be drive low or set to high impedance, and then is pulled high). The XL3 pulls it low for a few clock cycles to tell the xilinx to start programming, and then releases it. The xilinx chip will hold it low for as long as it is being programmed, and when it completes successfully, it will let it go high again. The crate geo address lines come straight off the backplane and I believe are set by jumpers on the crate itself. The logic reset is a one shot that is fired whenever the reset register address is written to. It is ored with another one shot that is configured to fire once when the power turns on. This reset signal clears all the XL3 hardware registers and sends a reset signal to the SNOBUS backplane.

The RLY_XL_TEST_REG block contains the relay, xilinx control, and test registers. The relay register is five bits that go directly to the backplane and can open or close the relays on the pmtics. The xilinx control register is eight bits that provide a little extra security in making sure that the FEC xilinx is never accidentally reprogrammed. In order to actually send the xilinx clock or data line from the CSR_REG, the signal XL_PERMIT must be high. This signal only goes high when bits 8-5 of the xilinx control register are set to 0101. In addition, when XL_PERMIT is high, the upper four address bits (A¡20-17¿) are replaced by the lower four bits of the xilinx control register. These bits must be set to 10XX in order for the FECs to be put into xilinx load mode. The test register is the only 32 bit register on the XL3 itself. It is mostly useful for lighting up leds. Note that a mod has made bit 0 of the test register responsible for enabling the local voltage readout adcs, so the test register should be left on 0xFFFFFFFF.

The DAV_MASK_REG allows you to control which data available lines are enabled. Coming off the backplane, the data available lines are driven low when the fifo has data to be read out, but the line is also low when no FEC is present in the slot. In order to make sure that we ignore any empty slots, there is a mask register that is anded with the data available lines.

There are three clocks on the XL3 that can be programmed with the CLOCK register: the adc, memory, and sequencer clocks. These are mostly used by the FECs. All three clocks are driven from a 16 MHz ECS2100AX oscillator. The clocks themselves are ics307-02s, and the programming instructions can be found on the datasheet (FIXME).

The VOLTAGE_MONITOR block contains registers for reading local low voltages and

temperatures on the XL3, as well as several alarms. Voltage monitoring exists on the FECs, but the ability to also check the local status of the XL3s as well as possibly a way way to check voltages without generating as much noise was desired. When writing to the voltage monitoring register, the lower eight bits are used to set up the alarm threshold dacs. These dacs are AD5724s and their programming instructions can be found on their datasheet (FIXME). These dacs set upper and lower limits for VP15, VM15, VCC, VEE, VP8, VP24, VM24, and a temperature sensor. The voltages are then put through comparators and if it crosses either upper or lower limit V_ALL_OK or T_ALL_OK is set low. In addition you can write to bits 11-9 of the voltage monitoring register to select a voltage or temperature sensor to measure. Once you select a voltage, you can read back bits 14-3 to get the 12 bit adc result.

The HV_CONTROL block has all the registers related to the high voltage. The hv csr register has various error bits and flags and must be written to in order to enable the hv supplies. The hv setpoint register sets the hv level for both supplies. Bits 11-0 control supply A and bits 27-16 control supply B. Coming out of the dacs the setpoints are a 0-10V signal. This signal first goes past a diode which caps the voltage at 9V, ensuring that the high voltage never goes above 2700V. It then goes through a switch which diverts the signal to ground if either the XL3_ALARM signal or the HV_KILL signal is high, otherwise it passes it through to the hv supply. The XL3_ALARM signal comes from the ML403, and the HV_KILL signal comes from a physical switch on the XL3 front panel. The hv readback registers allow you to monitor the HV voltage and current. The hv supply sends a 0-10V signal for each, which is then divided down to 0-5V. It alternates digitizing the currents and the voltages, and stores the most recent digitization of each in the register.

The XL3_LOGIC block is where the addresses are decoded to select the correct register, and some of the SNOBUS protocol is implemented. Five bits of the address are used to select a FEC or CTC, with a sixth used to select the XL3 (note that the XL3 can be selected at the same time as a single FEC). Once the address has been set, the VHDL state machine sets the GATED_STROBE signal high. This clocks the flip flops for whichever register has been selected. For read/writes to XL3 registers, the dtack signal is generated by delaying gated strobe. The FECs generate their own dtack when read from or written to. When this dtack is received, LEBA is set and the data lines are latched from the backplane.

FIXME add appendix with register and bit maps

4 VHDL State Machine

As seen in the hardware section, the ML403 communicates to the rest of the XL3 through roughly 64 pins: 32 data pins, 20 address pins, and an assorted group of control pins. The data pins are set up as GPIO pins, and the address pins are set up to be output only.

The state machine communicates with the PowerPC through six read and six write registers. The registers readable by the C code are: word1 word2 word3 error spare1 result v_all_ok. The register writeable by the C code are: data, address, start_bundle, start_command, reset. FIXME. Currently the state machine is clocked by the ml403's 300

MHz system clock. In it's idle state, it waits for the c code to either set start_bundle or start_command. If start_command is set, it reads in the address register and parses it to determine whether it's a read or write. It then puts the address and data out over the pins to the XL3, and once this is done, sets the gated strobe signal high. It waits for dtack to return from the XL3, and then reads the result of the read from the data pins. Once it has latched the result, it sets gated strobe low. It then waits for dtack to drop. Once dtack is low, it goes back to idle to await another command. Reading bundles is similar except that it does three reads in a row.

Each clock tick, the state machine checks that it is actually in one of its known states. If the register that holds the state glitches to any unrecognized value, the error register is set high to alert the PowerPC.

5 PowerPC code

The C code that runs on the PowerPC is based around the LWIP library. It is single threaded and uses interrupts and callbacks to send and receive packets. It runs in one main loop that is continually telling lwip to check its input buffers, and then allows the XL3 to do one action before repeating.

5.1 main

This file contains only the main function which has the main while loop. Before starting the loop, it calls init_XL3() to set up the VHDL registers, and initialize variables. It then initializes lwip and its callbacks using platform_setup_interrupts() and lwip_init(). Once lwip is ready it sets up a connection for the daq and then enters the main loop. Each time through the loop it calls xemacif_input which allows lwip to process incoming packets and call the callbacks. It then calls run_XL3 to do all other processing.

5.2 run_xl3

The function run_XL3() just calls the four functions that make the XL3 tick - check_connection(), update_status(), decide(), and do_what_was_decided(). Check_connection() maintains the connection to the daq, and update_status() checks the various fifo queues and polls the FECs for their memory and data available status.

```
int decide(void);
```

This function uses the XL3 status to determine what action should be done this time through the loop. In order to LWIP to function efficiently, it is important that the xemacif_input() function is called often and consistently and so nothing should hold up the loop for that long. Thus we determine the one and only thing we will do this time through the loop. The current decision tree is as follows:

```
If the daq connection is not setup or not ok:
    First do commands if there are commands to do and there is room in the command out q
    If you can't do a command, if there are bundles to be read, read bundles.
    If you can't do a command or read bundles, just idle.
If any packet failed to be sent correctly:
    Attempt to send the packet again.
If any error flag was just switched on:
    Send an error packet.
If any error flag was just switched off:
    Send an error packet.
If it is time to send a ping:
    Send a ping packet.
If any of the FECs have just had their fifo completely fill up:
    Send a screwed packet.
If any of the FECs have returned to a stable fifo level:
    Send a screwed packet.
If the command in queue is nearly full:
    If there is room in the command out queue, do a command.
    Otherwise send a command result packet.
(below is normal running)
If the message queue is over threshold:
    Send a message packet.
If the command out queue is over threshold:
    Send a command result packet.
If the command in queue is over threshold:
    Do a command.
If the command out queue is under threshold but not zero:
    Send a command result packet.
If the crate's average memory level is too high:
    If there is room in the bundle queue read a bundle.
    Otherwise send a megabundle packet.
If the bundle queue is over threshold:
    Send a megabundle packet.
If at least one FEC has data available:
    Read a bundle.
If the bundle queue is not empty:
    Send a megabundle packet.
Otherwise, idle.
```

In general it first checks that everything is ok, and deals with any errors or alarms. It then deals with any queue that is near full. It then prioritizes sending messages and sending command results to the daq first, followed by doing new commands, to make sure that the XL3s are always as responsive to the daq as possible. Once there are no new commands to

do, it then first checks whether the crate is filling up, and if it is, it prioritizes reading out the FEC fifos before sending packets. Otherwise it will always send out megabundle packets as soon as they are ready.

```
int do_what_was(int decided);
```

This function is just a switch on the result of the previous one, and calls a single function each time.

run_XL3.h contains the definitions for the return values of decide().

5.3 platform

This file contains xilinx specific code that deals with setting up the interrupts.

5.4 xl3_types.h

This header contains definitions of the basic packet types used by the XL3:

```
typedef struct
{
  uint16_t packet_num;
  uint8_t packet_type;
  uint8_t num_bundles;
} XL3_CommandHeader; //!< Header for every xl3 packet

typedef struct
{
  XL3_CommandHeader cmdHeader;
  char payload[XL3_MAXPAYLOADSIZE_BYTES];
} XL3_Packet; //!< all packets to and from xl3 look like this

typedef struct
{
  uint32_t word1;
  uint32_t word2;
  uint32_t word3;
} PMTBundle; //!< Bundle from FEC

typedef struct
{
  uint32_t cmd_num; // id number unique for the packet it came from
  uint16_t packet_num;    // number of the packet that created this command
  uint16_t flags;   // 0 = ok, 1 = there was a bus error
```

```
  uint32_t address; // address =
                    //           spare MEMREG WRITE* SPARE Board_select<5..0> Address<19
  uint32_t data;
} FECCommand; //!< Register read or write

typedef struct
{
  uint32_t howmany;
  FECCommand cmd[MAX_ACKS_SIZE];
} MultiFC; //!< many register reads or writes
```

5.5 queues

The header has all the queue data structures defined:

```
 typedef struct
 {
   FECCommand queue[MAX_FEC_COMMANDS];
   unsigned int first;
   unsigned int count;
 } FECCommandQueue;

 typedef struct
 {
   PMTBundle queue[MAX_BUNDLES];
   unsigned int first;
   unsigned int count;
 } PMTBundleQueue;

typedef struct
{
  XL3_Packet queue[MAX_MESSAGES];
  unsigned int first;
  unsigned int count;
} MessageQueue;
```

Methods for allocating and accessing queue structures live in queues.c.

```
FECCommandQueue* fcq_alloc(uint32_t address);
```

Allocates a block of memory at address and zeros it, mercilessly. There is no check on address, so "please" make sure it's safe to write there. In the EDK project, I have reserved a block of the SDRAM of size 0x00A00000 at 0x00100000. See the linker script for details. If you regenerate it, you MUST put back in the fifo_mem block!!

```
void fcq_free(FECCommandQueue* queue);
```

Deallocates the memory reserved for queue. The code actually never calls this anywhere, since there is really no condition in which we "quit" the application.

```
void fcq_push(FECCommandQueue* queue, FECCommand cmd);
```

Push a FECCommand cmd onto the end of a software FIFO FECCommandQueue. If it doesn't overflow (only enough space for MAX_FEC_COMMANDS is allocated), it adds it to the buffer and increments 'count.'

```
FECCommand fcq_pop(FECCommandQueue* queue);
```

Pops a FECCommand off the beginning of a software FIFO FECCommandQueue. If it doesn't underflow (ALWAYS CHECK that count¿0 before popping!), it removes the command at the 'first' pointer, increments 'first,' decrements 'count,' and returns the popped command.

```
FECCommand build_cmd(FECCommand* newcmd);
```

As it is now, takes a pointer to a FECCommand and returns it as an actual FECCommand struct. Sort of pointless... I think this started as something more useful, like building a user-defined command.

Equivalent functions exist for both PMTBundleQueues and MessageQueues.

5.6 bus

Contains all the methods that deal with the VHDL state machine and communicate with the rest of the XL3 and the crate.

```
int do_command(void);
```

This function is called to execute a single command from the command in queue. It pops a command off, and writes it's address and data to the VHDL registers. It then sets the start_command bit high to tell the state machine to begin. It will then wait for the read_ready to be set high from the VHDL, which tells it that the state machine has finished the command and the results are ready to be read out. If it times out waiting for read_ready, or if it sees the error bits written to by the VHDL, it resets the state machine and returns an error. The results of the command are then pushed into the command out queue.

```
uint32_t do_specified_command(uint32_t data, uint32_t address);
```

This method is very similar to do_command but it does not use the command in and out queues. Instead it takes the data and address as an argument and returns the result of the command.

```
int read_bundle(void);
```

This method loops through the FECs from slot 0 to slot 15, finds the first one with data available, and reads up to 120 bundles from it, or until the FECs fifo is empty, whichever comes first. Like in do_command(), it can time out while waiting for the read_ready signal from the VHDL. We need to be extra careful here in case there is a glitch midway through the bundle. Since each bundle takes three reads, it is possible to read out part of a bundle and become offset in the fifo. If this is not corrected, every following bundle read out will have parts of two bundles and will be undeciferable. Thus, whenever there is an error during a bundle read, the code attempts to resync the fifo. After reading each bundle, it is pushed onto the pmt bundle queue.

```
int read_specified_bundle(int slot, PMTBundle *p_bundle);
```

This method is identical to read_bundle except that you must specify which slot to read from. It will then read one and only one bundle, and returns a pointer to it instead of pushing it onto the bundle queue.

```
int resync_bundle_readout(int slot);
```

This is the method called when there is an error during a bundle read. It checks the fifo difference pointer for the slot in question, and reads out longwords until the difference is a multiple of three, indicating it is filled with full bundles.

```
void deselect_fecs(void);
```

This method just clears the address register read by the VHDL to ensure that the address pins are all set low across the backplane and no board is selected.

5.7 xl3_utils

The c file contains various utility functions for the XL3, while the header contains a majority of the global variables, including all the counters, the pointers to the VHDL registers, and the queues.

```
int init_XL3(void);
```

This is the first method called from main. It initializes all the global variables, and sets up and zeroes the VHDL registers. It also allocates space in the ML403's ram for four fifo queues: a message queue that holds generic XL3_Packets to be sent back to the daq, a pmt bundle queue that holds PMTBundles, and a command in and command out queue that hold FECCommands. All the FECs are masked out of the data available mask to ensure that on boot up nothing happens. It then reads from the XL3_CS_REG to get the crate geo address, and compares this with a list of known addresses to get the crate number. It returns this number so it can be used to set the ip address.

```
int change_mode(uint32_t new_mode, uint32_t slot_mask);
```

This function is called to either enable or disable the XL3s automatic readout. When the readout is disabled, the XL3 ignores the bundle queue and the FECs data available status so bundles are never read out nor pushed to the daq. When the readout is switched on, it needs to be told a slot mask so that it can mask off the data available bits of any slots that have no FECs.

```
int read_local_voltage(int select, float *readout);
```

This method reads out the adc results for one of several voltages or temperature sensors on the XL3. It first writes to the XL3_VR_REG to select which voltage it wants to check, then reads it and converts the adc counts to a voltage.

```
int do_nothing(void);
```

This method begins by not doing anything, and then continues in a similar fashion for no time at all. After finishing nothing, it doesn't check that any condition is met or attempt to do anything else. It does return 1.

```
void d_printf(char *buffer);
```

This method prints to the serial port only if XL3 debugging has been enabled.

5.8 xl3_status

The header file contains all the status globals and definitions for the queue status flags and the c file contains all the methods that set the status globals.

```
void update_status(void);
```

This function is called each time through the main loop. It first blinks the ml403_happy light so you can always tell if the ml403 is looping correctly or is stuck. It then checks the ping. If it is time to send another ping packet, it sets the ping_requested flag. If a ping has been sent but no pong has been received, it ticks the ping timeout. Once it has timed out, it retries several times, but then aborts the connection and tries to reconnect to the daq. Next, it checks how full the command and message queues are. If it is in NORMAL_MODE, it also checks the data available register, the FEC fifo levels, and the bundle queue. Otherwise, it ignores them and set their status to empty. Finally, it checks whether any of the FECs fifos are overflowing, and if they are, sets the appropriate screwed flag. Once a screwed packet has been sent to the daq, it keeps track of when the fifo memory level drops back down. Once it remains below some critical level for a certain number of loops, the code determines that the FEC is ok again and sends the daq another screwed packet to update it on the new status.

```
void check_data_available(void);
```

If the data_available status is still nonzero, this method does nothing. When it is zero, it reads out the data available register, sets the appropriate data available flags for each FEC, and then sets the data_available status equal to the number of FECs with data to read out. Each time read_bundle reads from a FEC, it zeroes that FECs data available flag and decrements the data_available status. Thus it will loop all the way through the slots before coming back to slot 0 even if it gets more data in the meantime. If there is an error reading the data available register, it sets a flag and will send an error packet to the daq.

```
void check_fec_mem_level(void);
```

This method polls all the FECs in the data available mask for their current fifo difference pointer. If there is an error during these reads, it sets a flag and will send an error packet to the daq.

```
void check_X_queue(void);
```

Checks the queue count and compares it to a threshold level and a warning level specified in main.h.

5.9 lwip_functions

These files contain everything for handling sending and receiving packets. The header has the globals that buffer the last sent packet until it is verified to have arrived ok, as well as the definitions for all the packet types that can be sent by the XL3 to the daq. (FIXME see appendix A) There are two functions that are used to send packets. quick_response takes a packet as an argument and will immediately tcp_write it and return the result. This is ok if used sparingly and not during readout. There is some danger that this will fail since there is nothing limiting how often it is called and so it can overflow lwip's buffers. The other function, transfer_packet, is only called from the main loop and only once per loop. It builds packets by popping data off the queues. Since this is only done once a loop, we can be sure that the lwip functions are being called frequently enough that it will be written safely.

```
int transfer_packet(uint8_t type);
```

This method is used to send data from any of the queues to the daq. It first ensures that the connection is still ok, and that there is room in lwip's output buffers for the packet. Then, depending on the type of packet to be sent, it either pops message packets, FECCommand results, or PMTBundles out of the queues and puts them into a packet payload. A flag is set when the packet is written, and is only cleared when the write returns ERR_OK indicating that there were no errors. Otherwise it will try again to send the same packet.

```
int quick_response(XL3_Packet *aPacket);
```

Immediately sends the packet passed to it to the daq. Returns the error code of the write. This is the only way to send packets to the daq manually - otherwise you have to trust the main loop to decide to send it and then actually send it on its own time.

```
err_t recv_callback(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err);
```

This callback is called by lwip every time a packet is received from the daq connection. If the daq has disconnected, this is also called but with a null pointer for the pbuf. The XL3 expects all packets it receives to be exactly 1444 bytes, so it checks the length of the data that it receives, and parses each 1444 bytes as its own XL3_Packet. If there are leftover bytes, they are stored away and added on in front of the next packet that is received. For each packet that is received, it calls processBuffer() to determine what to do. Note that this is all done as an interrupt and is not part of the main loop.

```
void err_callback(void *arg, err_t err);
```

Called whenever lwip has an error. If it reports that the connection has somehow died, it changes the connection status so the next time through the loop it can try connecting again.

```
err_t connected_callback(void *arg, struct tcp_pcb *tpcb, err_t err);
```

Called whenever a connection is fully established.

```
int open_connection(void);
```

Calls tcp_connect() and sets up the error callback function.

```
int close_connection()
```

Calls tcp_close().

```
void check_connection(void);
```

Is called once at the beginning of every time through the main loop. If the connection to the daq is still ok, it does nothing. If the connection was reset, it waits a few loops and then creates a new connection and attempts to connect. This also occurs when penn_daq isn't running yet when you start the XL3. If the connection is aborted, it closes the connection and creates a new one to attempt to connect on. This happens whenever you miss too many pings or when the XL3 isn't plugged in to the network yet.

```
int setup_daq_connection(void);
```

This method sets up the network interface for connecting to the daq and allocates a tcp_pcb struct for the connection.

```
void mq_printf(char* string, int sendnow);
```

This method can be used to send message strings by packet to the daq. Each time it is called, the string it is passed is added on to current_string until it reaches 1440 bytes, at which point an XL3_Packet is created and is pushed onto the message queue. For messages that you want to push onto the message queue right away, you can set the sendnow argument to 1. Note that even with sendnow enabled, you still have to wait for the main loop to get to it in the message queue. If you really want to send a message immediately you need to construct your own packet and set it with quick_response.

5.10 process_packet

The header file contains all the packet types that the XL3 can receive from the daq (FIXME see appendix A). The c files contains the functions that deal with parsing XL3_Packets.

```
void processBuffer(XL3_Packet* aPacket);
```

This function is a switch on the xl3 packet type. Since this function is called from the recv callback, all the functions called from here also occur outside the main loop. For this reason it is important to ensure that no function is too long or can crash, or lwip may start having problems. For most packet types, a function is called, and then once it returns, any results are stuffed into the payload of the same packet and it is then pushed into the message queue to be sent back. The daq then just waits to get the same packet type back to know that a command has been executed.

```
int queue_cmds(XL3_Packet *aPacket);
```

This function casts the packet's payload as a MultiFC and then pushes each command there into the command queue to be executed whenever the main loop decides. The packet immediately sent back to the daq is only there to inform the daq that the commands have been queued; it does not contain the actual results of the reads/writes.

```
int do_single_cmd(XL3_Packet* aPacket);
```

This casts the packet's payload as a single FECCommand, executes the command immediately, and returns the results in the same packet. This is useful if you want to get the results right away and not wait for the main loop.

```
int do_multi_cmd(XL3_Packet* aPacket);
```

This is like do_single_cmd but with a MultiFC.

5.11 fec_utils

Various utilities that read or write to the FECs.

```
uint32_t board_id_read_single(int mb, int chip, int reg);
```

Does a serial read of one of the id chips on the FEC

```
int load_tac_bits(uint32_t select_reg, uint16_t *tacbits_ptr);
```

Clocks in to the cmos shift register the tac delay bits.

```
int reset_fifos(uint32_t slot_mask);
```

Resets the FEC fifos without resetting the rest of the board.

```
int check_total_count(int slot, uint32_t *data);
```

Reads the cmos total count register which tells you how many times that channel has fired. Since there can be errors reading this register, it reads it up to ten times until it reads it without the error bit showing up.

```
int reset_crate(void);
```

Just writes to the reset register on the XL3, which will fully reset the XL3 and all FECs, including fifos and xilinxi.

```
int xilinx_load(uint32_t slot_mask, uint32_t file);
```

Writes the xilinx code to the FEC xilinx chips. The bitstream is hardcoded into the data.h header file. First writes to the xlcon register to turn on xlpermit, then sets done/program* low and clocks in the data. When it's finished, waits to see done/program* come back high. If it doesn't, returns an error.

```
int XL3_clock_load(void);
```

Loads the three clocks on the XL3.

```
int build_crate_config(uint32_t slot_mask);
```

Updates the crates record of which FECs and DBs are in the crate. It first tries to read from each FEC's general csr register, and if it can't, it marks that slot as empty. Otherwise it reads out the id chips.

```
int dac_fullbuff(int slot);
```

```
int load_cmos_shift(int slot);

int fec_load_crate_add(int crate, uint32_t slot_mask);
```

Writes the crate address to each FECs general csr register so that it is properly recorded in each pmt bundle.

```
int set_crate_pedestals(u_short slot_mask, uint32_t pattern);
```

Sets the pedestal enable registers for all the fecs in the crate, using the XL3's knowledge of which slots are empty to avoid waiting for a bunch of writes that will never finish.

```
uint8_t read_voltage_value(uint32_t amask, int slot);
```

Attempts to read a voltage from the FEC several times until it manages to do it without the VMON_BUSY_BIT being set.

5.12 hv_utils

5.13 crate_init

Crate init packets must be sent from the daq to the XL3 17 times in order to actually start a crate init. The first 16 times they carry the database information for one of each of the FECs. The last time they have any leftover database info and tell it to start. Each time it calls crate_init_packet()

```
int crate_init_packet(char *payload);
```

First parses the packet to get the mb number. It keeps track of which FECs it has gotten database information for, and will not start if it misses some. After receiving data for all 16, on the next packet it starts the initialization. It then sends back in the final results packet the current crate configuration.

```
int crate_init(uint8_t xil_load, uint8_t hv_reset, uint8_t shift_only, uint16_t slot_mas
```

This method actually does the initialization. If shift_only is set, it only loads the cmos shift register, and uses this just as a cheezy way to get the database info all updated first. Otherwise, it resets the crate and loads the XL3 clocks. It then attempts to load the xilinx to each FEC. Next it updates the crate configuration. Then, for each FEC that is masked it, it tries to clear it by a write to the general csr register. This is the first place where it is easy to see whether the FEC is actually installed, so if there is an error here it quits instead of letting crate_init continue and take forever throwing lots of error messages. It then loads the dacs, the shift register, clears everything, sets the CTC delay, then loads the crate address into the FEC general csr register, and finally resets the XL3 mask register and test register.

5.14 load_dacs

A bunch of functions for loading different dacs on the FECs.

5.15 vmon_dacs

A bunch of functions for loading different dacs on the XL3s.

5.16 cald_test

This test is used to make sure that the three different charge adcs and the tac adc are functioning correctly. You can switch them to digitizing the output of a dac, so you can set the level of the dac and then make sure the output matches and is nice and linear. Note that you need to load the alternate bitstream to the FEC xilinx chips before this test will work.

```
int cald\_test(uint32_t slot_mask, uint32_t num_points, uint32_t samples, uint32_t upper
```

This function just uses the articles to determine what dac counts to sample at. For each point it calls caldac_get_point(). Every 100 points, it sends a CALD_RESPONSE_ID packet back to the daq. It does not respond with the original command packet until the whole test is done.

```
int caldac_get_point(uint16_t setval, uint16_t *adc, int slot);
```

It first sets the dac to the right value, and manually starts the adc conversion, then latches the results and reads them out.

5.17 fec_test

5.18 mem_test

5.19 vmon

5.20 zdisc

This test plays with the discriminator level until it is firing at the desired rate.

6 Penn DAQ

If you are looking for help using Penn DAQ, please refer to "Penn DAQ for dummies: The Official Penn DAQ User's Guide." This section is for code documentation and development. Penn DAQ is a multi threaded program that uses pthreads and unix sockets. Since it is multi threaded, it can have one thread constantly monitoring the sockets for new data. Whenever a

user requests a command be run, a new thread is spawned to execute it, and is deleted when it is done. This allows us to run tests or commands on different XL3s independently at the same time. All commands that work this way are made up of two functions; command_name(char *buffer) and pt_command_name(void *args). The first function is called when a command comes in and buffer is the command string. This function parses the string for arguments, and fills the struct command_name_t. This struct is used to pass all the arguments to a pthread. Before spawning a thread, it checks to make sure all XL3s and SBCs required by the command are free and not currently in use. If any of them are, it returns an error. Otherwise, it marks those XL3s/SBCs as in use and spawns a thread with the function pt_command_name. This function will actually perform the command and can communicate to the sockets it locked down. When the thread finishes or exits on an error, it marks those sockets as free again and sets a flag to let the main thread know that it is finished. The main thread will then free the thread to be used again.

6.1 Core

6.1.1 main

Initializes all globals, parses all command line arguments, and reads in the configuration document. It then sets up all the listener sockets and begins the main loop. Each time through the loop it checks which sockets are available to check, and then select()s them. It then loops through every readable socket and calls read_socket();

6.2 Crate

6.2.1 Crate_init

Contains the crate_init command. This command first pulls

6.2.2 Run_Pedestals

6.3 db

7 Register Map

```
//XL3 registers
#define RESET_REG (0x02000000)
#define DATA_AVAIL_REG (0x02000001)
#define XL3_CS_REG (0x02000002)
#define XL3_MASK_REG (0x02000003)
#define XL3_CLOCK_REG (0x02000004)
#define RELAY_REG (0x02000005)
#define XL3_XLCON_REG (0x02000006)
#define TEST_REG (0x02000007)
```

```
#define HV_CS_REG (0x02000008)
#define HV_SETPOINTS (0x02000009)
#define HV_VR_REG (0x0200000A)
#define HV_CR_REG (0x0200000B)
#define XL3_VM_REG (0x0200000C)
#define XL3_VR_REG (0x0200000E)

//FEC registers
//Discrete
#define GENERAL_CSR (0x00000020)
#define ADC_VALUE_REG (0x00000021)
#define VOLTAGE_MONITOR (0x00000022)
#define PEDESTAL_ENABLE (0x00000023)
#define DAC_PROGRAM (0x00000024)
#define CALDAC_PROGRAM (0x00000025)
#define FEC_HV_CSR (0x00000026)
#define CMOS_PROG_LOW (0x0000002A)
#define CMOS_PROG_HIGH (0x0000002B)
#define CMOS_LGISEL (0x0000002C)
#define BOARD_ID_REG (0x0000002D)
//Sequencer
#define SEQ_OUT_CSR (0x00000080)
#define CMOS_CHIP_DISABLE (0x00000090)
#define FIFO_READ_PTR (0x0000009c)
#define FIFO_WRITE_PTR (0x0000009d)^M
#define FIFO_DIFF_PTR (0x0000009e)
//CMOS internal
#define CMOS_INTERN_TEST(num) (0x00000100 + 0x00000004 + 0x00000008*num)
#define CMOS_INTERN_TOTAL(num) (0x00000100 + 0x00000003 + 0x00000008*num)

//CTC registers
#define CTC_DELAY_REG (0x00000004)

// add to register value
#define READ_MEM (0x30000000)
#define READ_REG (0x10000000)
#define WRITE_MEM (0x20000000)
#define WRITE_REG (0x00000000)

#define FEC_SEL (0x00100000) // to select board #i, (0..15), use FEC_SEL * i
#define CTC_SEL (0x01000000)
#define XL3_SEL (0x02000000)
```

```
// define MTC register space
#define MTCRegAddressBase  (0x00007000)
#define MTCRegAddressMod   (0x29)
#define MTCRegAddressSpace (0x01)
#define MTCMemAddressBase  (0x03800000)
#define MTCMemAddressMod   (0x09)
#define MTCMemAddressSpace (0x02)

//MTC registers
#define MTCControlReg (0x0)
#define MTCSerialReg  (0x4)
#define MTCDacCntReg (0x8)
#define MTCSoftGTReg (0xC)
#define MTCPedWidthReg (0x10)
#define MTCCoarseDelayReg (0x14)
#define MTCFineDelayReg (0x18)
#define MTCThresModReg (0x1C)
#define MTCPmskReg (0x20)
#define MTCScaleReg (0x24)
#define MTCBwrAddOutReg (0x28)
#define MTCBbaReg (0x2C)
#define MTCGtLockReg (0x30)
#define MTCMaskReg (0x34)
#define MTCXilProgReg (0x38)
#define MTCGmskReg (0x3C)
#define MTCOcGtReg (0x80)
#define MTCC50_0_31Reg (0x84)
#define MTCC50_32_42Reg (0x88)
#define MTCC10_0_31Reg (0x8C)
#define MTCC10_32_52Reg (0x90)
```

8 Packet types

8.1 DAQ to XL3

8.1.1 DAQ_QUIT_ID

Tells the XL3 to disconnect from the daq and exit the main loop.

8.1.2 PONG_ID

The DAQ echos this packet type back when it gets a ping to let the XL3 know the connection is still alive.

8.1.3 CHANGE_MODE_ID

```
typedef struct{
  uint32_t mode;
  uint32_t davail_mask;
} change_mode_args_t;

typedef struct{
  uint32_t error_flags;
} change_mode_results_t;

#define INIT_MODE (1)
#define NORMAL_MODE (2)
```

Used to turn the XL3 readout on or off. When the readout is turned on (normal mode), a data available mask must also be specified to tell the XL3 which slots to read from.

8.1.4 STATE_MACHINE_RESET_ID

First clears the start_command and start_bundle registers, and then sets the VHDL reset register high. This should put the state machine back in the idle state and it should stay there. It then reads the error and spare1 registers to ensure that it actually is now in the idle state.

8.1.5 DEBUGGING_MODE_ID

```
typedef struct{
  uint32_t on_off;
} debugging_mode_args_t;
```

Turns on more verbose output to the serial port.

8.1.6 FAST_CMD_ID

```
typedef struct{
  FECCommand command;
} fast_cmd_args_t;

typedef struct{
  FECCommand command_result;
```

```
} fast_cmd_results_t;
```

Does one command immediately using do_specified_command and then sends the results back immediately with quick_response. The result of the read/write is put into the data field of the command, and the flag field is set to busstop.

8.1.7 MULTI_FAST_CMD_ID

```
typedef struct{
  MultiFC commands;
} multi_fast_cmd_args_t;

typedef struct{
  MultiFC command_results;
} multi_fast_cmd_results_t;
```

Just like fast_cmd except with many commands in a MultiFC. Does all the commands immediately and returns the results with quick_response.

8.1.8 QUEUE_CMDS_ID

```
typedef struct{
  MultiFC commands;
} queue_cmds_args_t;
```

Pushes each command from the MultiFC into the command queue. Returns the same packet, does not return the results of the commands.

8.1.9 FEC_LOAD_CRATE_ADD_ID

```
typedef struct{
  uint32_t slot_mask;
  uint32_t crate_num;
} fec_load_crate_add_args_t;

typedef struct{
  uint32_t error_flags;
} fec_load_crate_add_results_t;
```

Loads the crate address into each FEC's general csr register.

8.1.10 SET_CRATE_PEDESTALS_ID

```
typedef struct{
  uint32_t slot_mask;
  uint32_t pattern;
} set_crate_pedestals_args_t;

typedef struct{
  uint32_t error_flags;
} set_crate_pedestals_results_t;
```

Loads the pedestal enable registers on the FECs. Any FEC not in the slot mask that is known to be connected will have its pedestal enable register zeroed.

8.1.11 BUILD_CRATE_CONFIG_ID

```
typedef struct{
  uint32_t slot_mask;
} build_crate_config_args_t;

typedef struct{
  uint32_t error_flags;
  hware_vals_t hware_vals[16];
} build_crate_config_results_t;
```

Checks each slot to see if there is a FEC there and if there is, reads the id chips for the FEC and the DBs.

8.1.12 LOADSDAC_ID

```
typedef struct{
  uint32_t slot_num;
  uint32_t dac_num;
  uint32_t dac_value;
} loadsdac_args_t;

typedef struct{
  uint32_t error_flags;
} loadsdac_results_t;
```

Loads one dac on the FEC.

8.1.13 MULTI_LOADSDAC_ID

```
typedef struct{
  uint32_t num_dacs;
  loadsdac_args_t dacs[50];
} multi_loadsdac_args_t;

typedef struct{
  uint32_t error_flags;
} mutli_loadsdac_results_t;
```

Loads up to 50 dacs one after the other.

8.1.14 DESELECT_FECS_ID

Zeroes the VHDL address register.

8.1.15 READ_PEDESTALS_ID

```
typedef struct{
  uint32_t slot;
  uint32_t num_reads;
} read_pedestals_args_t;
```

Pushes num_reads memory reads into the command queue. This is useful when the number of reads is many more than would fit in a normal QUEUE_CMDS packet.

8.1.16 LOADTACBITS_ID

```
typedef struct{
  uint32_t crate_num;
  uint32_t select_reg;
  uint16_t tacbits[32];
} loadtacbits_args_t;
```

8.1.17 RESET_FIFOS_ID

```
typedef struct{
  uin32_t slot_mask;
} reset_fifos_args_t;
```

8.1.18 READ_LOCAL_VOLTAGE_ID

```
typedef struct{
  uint32_t voltage_select;
} read_local_voltage_args_t;

typedef struct{
  uint32_t error_flags;
  float voltage;
} read_local_voltage_results_t;
```

8.1.19 CHECK_TOTAL_COUNT_ID

```
typedef struct{
  uint32_t slot_num;
} check_total_count_args_t;

typedef struct{
  uint32_t error_flags;
  uint32_t count[32];
} check_total_count_results_t;
```

8.1.20 SET_HV_RELAYS_ID

```
typedef struct{
  uint32_t mask1;
  uint32_t mask2;
} set_hv_relays_args_t;
```

8.1.21 GET_HV_STATUS_ID

```
typedef struct{
  uint32_t error_flags;
  float voltage_a;
  float voltage_b;
  float current_a;
  float current_b;
} ge_hv_status_results_t;
```

8.1.22 HV_READBACK_ID

```
typedef struct{
  float voltage;
  float current;
```

```
} hv_readback_results_t;
```

8.1.23 READ_PMT_CURRENT_ID

```
typedef struct{
  uint32_t slot;
  uint32_t channel_mask;
} read_pmt_current_args_t;

typedef struct{
  uint32_t error_flags;
  uint8_t pmt_current[32];
} read_pmt_current_results_t;
```

8.1.24 SETUP_CHARGE_INJ_ID

```
typedef struct{
  uint32_t slot;
  uint32_t channel_mask;
} setup_charge_inj_args_t;
```

8.1.25 FEC_TEST_ID

```
typedef struct{
  uint32_t slot_mask;
} fec_test_args_t;

typedef struct{
  uint32_t error_flag;
  uint32_t discrete_reg_errors[16];
  uint32_t cmos_test_reg_errors[16];
} fec_test_results_t;
```

8.1.26 MEM_TEST_ID

```
typedef struct{
  uint32_t slot_num;
} mem_test_args_t;

typedef struct{
  uint32_t error_flag;
  uint32_t address_bit_failures;
  uint32_t error_location;
  uint32_t expected_data;
```

```
  uint32_t read_data;
} mem_test_results_t;
```

8.1.27 VMON_ID

```
typedef struct{
  uint32_t slot_num;
} vmon_args_t;

typedef struct{
  float voltages[21];
} vmon_results_t;
```

8.1.28 BOARD_ID_READ_ID

```
typedef struct{
  uint32_t slot;
  uint32_t chip;
  uint32_t reg;
} board_id_args_t;

typedef struct{
  uint32_t id;
} board_id_results_t;
```

8.1.29 ZDISC_ID

```
typedef struct{
  uint32_t slot_num;
  uint32_t offset;
  float rate;
} zdisc_args_t;

typedef struct{
  uint32_t error_flag;
  float MaxRate[32];
  float UpperRate[32];
  float LowerRate[32];
  uint8_t MaxDacSetting[32];
  uint8_t ZeroDacSetting[32];
  uint8_t UpperDacSetting[32];
  uint8_t LowerDacSetting[32];
} zdisc_results_t;
```

8.1.30 CALD_TEST_ID

```
typedef struct{
  uint32_t slot_mask;
  uint32_t num_points;
  uint32_t samples;
  uint32_t upper;
  uint32_t lower;
} cald_test_args_t;
```

8.1.31 SLOT_NOISE_RATE_ID

```
typedef struct{
  uint32_t slot_num;
  uint32_t channel_mask;
  uint32_t period;
} slot_noise_rate_args_t;

typedef struct{
  uint32_t error_flags;
  float rates[32];
} slot_noise_rate_results_t;
```

8.2 XL3 to DAQ

8.2.1 CALD_RESPONSE_ID

```
typedef struct{
  uint16_t slot;
  uint16_t point[100];
  uint16_t adc0[100];
  uint16_t adc1[100];
  uint16_t adc2[100];
  uint16_t adc3[100];
} cald_response_packet_t;
```

8.2.2 PING_ID

```
typedef struct{
  uin32_t mem_level[16];
} ping_packet_t;
```

8.2.3 MEGA_BUNDLE_ID

```
typedef struct{
```

```
  PMTBundle bundles[MEGA_SIZE];
} mega_bundle_packet_t;
```

8.2.4 CMD_ACK_ID

```
typedef struct{
  MultiFC commands;
} cmd_ack_packet_t;
```

8.2.5 MESSAGE_ID

```
typedef struct{
  char message[1440];
} message_packet_t;
```

8.2.6 ERROR_ID

```
typedef struct{
  uint32_t cmd_in_rejected;
  uint32_t transfer_error;
  uint32_t xl3_data_avail_unknown;
  uint32_t bundle_read_error;
  uint32_t bundle_resync_error;
  uint32_t mem_level_unknown[16];
} error_packet_t;
```

8.2.7 SCREWED_ID

```
typedef struct{
  uint32_t screwed[16];
} screwed_packet_t;
```