Receiving L1A Control Data from MACTRIS+ via Ethernet

By Molly Taylor on May 26, 2016

Table of Contents

Introduction 3
Packet-Receiving Software 4
    How to Compile 4
    Configuring MACTRIS+ 4
    Setting Up Banjo-1 5
    Sending Packets from MACTRIS+ 5
Walkthrough of Code 6
    main Function 6
    get_packet Function 6
    got_packet Function 6
    extract Function 7
Packet Format 8
    General Layout 8
    Ethernet Header 8
    IP Header 8

Introduction

One of the new features of MACTRIS+ is its capability to send control data directly to Level 3 (L3) via Ethernet – specifically to Banjo-1. We use this to save data on all Level 1 Accepts (L1A) about trigger type, timestamp, veto data, etc., regardless of whether the event is passed on to Level 2. With the old MACTRIS, we only had the capability to record control information about events that went on to Level 2 (L2A). Thus the new capabilities of the MACTRIS+ FPGA necessitate new software to handle the data and write it into a file once it arrives at Banjo-1. This documentation is about the software that receives the Ethernet packets from MACTRIS+ and the format of the .bin file it writes to Banjo-1.

Packet-Receiving Software

The software that receives Ethernet packets from MACTRIS+ on Banjo-1 is written in C.

How to Compile

The code is compiled with the g++ compiler on Banjo. Note that the code will not compile correctly if you attempt to do it on Banjo-1. The steps for compiling the code are outlined below.

1. ssh into Banjo and log in to your account.
2. Currently the latest code is located in **/sudaq/mitay** and is titled *nosleep4.c*. Navigate to this directory and make sure you see it there. If preferred, you can copy it elsewhere, but it must be somewhere Banjo-1 can access.
3. In order to compile the code, you must have the *k0to_config.h* header file in the same directory as *nosleep4.c*. You can find a copy in **/sudaq/mitay**.
4. Now you can actually compile the code. Because it uses lpcap to capture the Ethernet packets, you will need some extra arguments. An example is below.
   ```
   g++ nosleep4.c –o nosleep4 –lpcap –lpthread -lrt
   ```
5. Now an executable file named *nosleep4* has been generated. You may use it as you see fit.

Configuring MACTRIS+

So you've compiled the code, and now you want to run some tests to make sure it's receiving packets correctly before you set it up with your run control script or whatever. There are a few things you have to set up before this is possible. First, you've got to have your MACTRIS+ board set up in its crate and connected to Banjo-1 with an Ethernet cable. Then, after you have downloaded the latest firmware onto MACTRIS+, it must be configured to send out simulated Ethernet Packets. The steps for doing this are outlined below.

1. ssh to the VME processor that connects to MACTRIS+. Ask Monica if you don't know the IP address or login information. Go to the **daq_dma** directory.
2. To configure the board to accept multiple triggers, you must change the value of the *cc* register. An example is below, where the MACTRIS+ board is in slot 6.
   ```
   ./vme7700 write 6 cc a01304ff
   ```
3. Now you have to configure the Ethernet to send out simulated data via bit 2 of VME register *1cc*. If you are curious, bits 0 and 1 correspond to GTP Transceiver channels 0 and 1. The command to enable sending simulated Ethernet data is below
   ```
   ./vme7700 write 6 1cc 4
   ```
4. After this you are pretty much ready to go, but it is important to know that you can change the number of triggers that are sent out at once by modifying VME register *a4*. The top 16 bits set the space between triggers in units of # of clock cycles, and the bottom 16 bit set the number of triggers sent per a single pulse of the *3c* register. To send out just one trigger at a time, use the following command.
   ```
   ./vme7700 write 6 a4 60000001
   ```
5. You can also change the number of L1A put in each Ethernet packet by modifying the bottom 16 bits of VME register *11c*. By changing the top 16 bits of this register, you can control interpacket delay. However, unless you know what you're doing, it's best to leave those alone.

Setting Up Banjo-1

Ok, now that MACTRIS+ is all configured, you've got to set up Banjo-1 to receive the Ethernet packets and write them to a file. Luckily this requires very little effort. Even so, there is an excellent outline below.

1. Login to your Banjo-1 account and go to **/sudaq/mitay**.
2. Enter `su` in the terminal and input the password so you can use the `sudo` command.
3. Now you just run the latest executable code and it should capture packets and write them to a file. You must also enter the run number, which will be appended to the file name, when you do this. A sample command is below, though obviously you should change the run number to best fit your purposes.
   ```
   sudo ./nosleep4 1234
   ```
4. Enter `C-c` whenever you want to stop listening for packets, and the code will close the file and terminate.

Sending Packets from MACTRIS+

Now that everything is set up, you can send packets from MACTRIS+ and receive them at Banjo-1. However, there are still a few things you must know. The MACTRIS+ Ethernet firmware works by writing control data to a large FIFO while live is high (i.e. the first two seconds of a spill) and then sending it out as soon as live is written low. So all you need to do to send packets from MACTRIS+ is write live high, send triggers, and then write live low. Recall that you can change the number of triggers sent out at once with the *a4* VME register. Refer to the section Configuring MACTRIS+ if you have forgotten. A table outlining relevant commands is below.

Write live high	`./vme7700 write 6 1c 1`
Send *a4* number of triggers	`./vme7700 write 6 3c 1`
Write live low	`./vme7700 write 6 1c 0`

Walkthrough of Code

The code is well commented, so it shouldn't be too hard to follow. Even so, I will attempt to describe the overall purpose of each function in an effort to mitigate future struggles.

main Function

The main purpose of the main function is to create to threads to capture the packets and then write them to file. To create the threads, called *iret1* and *iret2*, the function *pthread_create* is used. You can Google this for a more complete explanation, but I will outline the most relevant information. The first argument is the actual thread, declared earlier in main and passed by reference. The default value for the second argument is adequate, so we pass a null pointer. The third argument is the name of the function that will be run continuously. As you can see, it does not look like a regular function call, and there is a different mechanism for passing variables to the functions if it is required. That, in fact, is the purpose of the fourth argument. This should be a pointer to any variables that will be required by the requisite function.

get_packet Function

This function opens a handle for the Ethernet port, which is defined by the *dev* string. If you have plugged the cable into a different Ethernet port, you will need to alter the value of this variable. Furthermore, *get_packet* calls the *terminate* function, which ends the program and closes the data file when `C-c` is sent. If you want to change the way the program is terminated, for example by sending a different command, you have to modify the first argument of this function call from *SIGINT* to whatever is preferable. A cursory Google search will tell you what other options are available, or you can ask Stephanie.

The last thing *get_packet* does before it closes the handle and terminates is call *got_packet*. It uses the *pcap_loop* function to run *got_packet* over and over again – the desired number of times. Inputting '-1' as the second argument will run the function infinitely, until `C-c` is sent from the terminal. Inputting any positive number will run the function that number of times.

got_packet Function

The *got_packet* function has several different steps, which are outlined below.

1. The *filled* array is used to keep track of which parts of the buffer are full (i.e. contain data that has not yet been written to file) and which are ready to accept new data. It is initialized to saying the buffer is entirely empty, and whenever a 128-bit word is written to the buffer, a corresponding bit in the *filled* array is written high. The first part of *got_packet* cycles through the *filled* array, starting at the beginning, to find the first empty spot, the indices of which are recorded in variables *x* and *i*.
2. Once a spot for the new data is found, we create pointers to its future destination in the buffer and to its current destination in the packet (where the new packet is automatically written upon its reception). The actual data part of the packet (i.e. the payload) starts after the Ethernet header, IP header, and Josh's header containing the spill number and packet flag. Please note that variables *ETH_HEADER_SIZE* and *IP_HEADER_SIZE* are in units of 16-bit words, aka 2 octets.
3. The next step is actually copying the data into the buffer. The first step in doing this is determining the length of the payload, which is the purpose of the *get_length* function. Please note that packet length (consisting of IP header length plus payload length) is recorded in bytes 3-4 of the IP header. The *get_length* function will return the number of 128-bit words in the

payload. Then we just cycle through each words and copy it into the buffer, at the same time recording each position as being filled in the *filled* array. Other things to know are that we need to swap the bytes in each 16-bit word because of the way they come out of the FIFO on the receiving end. Then we increment the pointers by 8, which moves them forward 128 bits (each increment moves the pointer forward by the size of the pointer type – in this case a 16-bit integer).

4. Now we update the packet flag, which lets us know if the packet we just received is a first, middle, or last packet for the spill we are on. If only one packet is sent for a spill, the packet flag will say it is the last packet, rather than the first. This is why we primarily use the last-packet flag as a marker in the code. When the *get_packet_flag* returns true, it means this is the last packet and it is now time to write the contents of the buffer to file. Hence why we update the value of *packet_flag* after the contents of the latest packet have been written to the buffer.
5. Last, we update the *total_length*, which is reset after each spill has been written to file. It is used to keep track of how many 128-bit words of the *evt_buf* buffer need to be written to file – the buffer is much larger than the typical number of triggers we will receive each spill so we obviously don't want to write the whole thing. Not to mention any of the contents beyond *total_length* would be nonsensical. Then, we increment the *packet_counter*, which records the number of packets the code has successfully received and which is output as part of general statistics once the program has terminated.

extract Function

The *extract* function creates a file, to which it proceeds to write the contents of the Ethernet packets the program receives. It runs continuously during program execution. Please not that in order for this function to work with *pthread_create* it must return a void pointer, and only take void pointer arguments. The first thing that must be done is to create a pointer of the desired type (in this case *char*), and cast the arguments to this type of pointer. In the code, this is used to append the run number to the file name, thus making each file unique and ensuring data is not overwritten.

Once the file has been created, the function runs an infinite while loop that only ends when *terminate* is triggered by entering `C-c` in the terminal. This while loop writes data to the file after all the packets for a given spill have been received, as indicated by *packet_flag*. Prior to the data for each spill, it writes 14 bytes of 1s (used as a marker between spills) followed by the 2-byte spill number. Then it determines which part of the buffer is full, and writes that to file for *total_length* number of 128-bit words. Only after we have written all the new data to file do we go through and change the corresponding parts of the *filled* array to say empty. This is so that if a new packet is received before we finish writing to file, the old data will not be overwritten.

The last step in this process is to reset everything so a new spill is ready to be written to file. This consists of resetting *total_length* to zero, setting *packet_flag* back to false, and incrementing *spill_counter* to keep track of program statistics.

Packet Format

The figures below detail the contents of each Ethernet packet.

General Layout

Ethernet Header	IP Header	Spill Header	Payload

Ethernet Header

Octet	*Name*	*Content*
1-6	Destination MAC Address	L3 node MAC address
7-12	Source MAC Address	0x00.4B.30.54.4F.XX
13-14	MAC Type or Length	0x0800 for standard packets 0x8870 for jumbo packets

IP Header

Octet	*Name*	*Content*
1	IP Version & Header Length	0x45
2	Type of Service	0x00
3-4	Total Length of Payload + IP Header	Variable – in units of octets
5-6	Identification and Fragment ID	0x00.F2
7-8	Fragment Offset	0x00.00
9	Time-to-Live	0x40
10	Protocol	0x11 for UDP
11-12	IP Header Checksum	One’s complement sum
13-16	Source IP Address	192.68.55.1XX which is 0xC0.44.37.YY in hex
17-20	Destination IP Address	L3 node IP address

Spill Header

Bit	*Name*	*Content*
0-13	Spill Counter	Spill number in current run
14-15	Packet Flag	2’b00 for first packet of spill 2’b01 for middle packet of spill 2’b11 for last packet of spill

Payload

The payload is in units of 128-bit words. There is one 128-bit word of control information per L1A.

File Format

The format for the output file is essentially the data for each spill written one after the other. The beginning of the data for a given spill is marked by 112 bits of 1s followed by the 16-bit spill number, as a heading. Then there is 128 bits of control information for each L1A, with no delineating marker in between. The contents of each 128-bit word is are still under consideration, but the current contents are in the table below.

Bit	*Contents*
0-29	timestamp_data[29:0]
30-45	csum_out, csum_bit_out
46-63	sum_out
64-85	trg_data_out, raw_trg_data_out
86-103	veto_data
104-121	L1A_GATED_BUS
122-127	Currently free – 6'b000000