DETECTOR READOUT OVERVIEW 12/08/2003

The 90 GHz array will have 64 detectors whose resistance is inversely proportional to the amount of light falling on them. Accurate resistance measurements normally require 4 wires per resistor. With 64 detectors this is impractical. Instead time division multiplexing developed by NIST and Goddard will be used.

Time division multiplexing works by grouping the detectors into n groups with m bolometers per group. A single bolometer is read from each group at a time and the controlling electronics cycles through all m bolometers in the group. Each group requires 4 pairs of wires to read it out and m pairs of control wires are used to switch which bolometer is being read. As the control wires can be common to all groups (which are often referred to as columns) only $4n + m$ pairs of wires are needed to read out an array. With an 8 by 8 multiplexing system such as we intend to use this is 40 wires instead of 256.

Multiplexing Electronics Overview

Figure 1 shows Dominic Benford's block diagram of the multiplexing electronics. The multiplexing electronics consists of four parts, a computer (the same as is used to run the cryogenic hardware), a digital electronics box (DEB), an analog electronics tower, and the cold electronics (the multiplexers, amplifiers and bolometers).

The DEB contains one clock card, 8 single channel digital feedback (DFB) cards, and a general purpose IO interface card (described on the diagram as a facilities acquisition card). The later can be used to add information such as the status of the cal signal or an external clock into the time stream.

The analog tower contains one power card, one address card, one feedthrough card, one preamp card, and 4 bias cards. The feedthrough, preamp and bias cards all have 8 channels each with a SMB connector into the room and a stripline connector going into the cryostat. The preamp and bias cards are similar – they are programmable voltage sources.

A PC controls the whole system. The PC's serial ports have serial-fiber converters. Fiber links connect the serial ports with the DEB's clock card and analog tower's power card, which can send commands to other DEB or tower cards respectively.

Figure 2 shows the cold multiplexing circuitry. Each column of bolometers (marked TES for transition edged superconductor) is supplied with a bias voltage from the first of the bias cards in the tower (a separate channel is used for each column). As the optical signal landing on the bolometer changes its resistance, the current flowing through each TES changes. To read out a row, the cold address electronics puts a voltage across the 'row select' line, turning on the 1st stage SQUID amplifier (marked input SQUID on the diagram) – (SQUID stands for **s**uperconducting **qu**antum **i**nterference **d**evices and they are very sensitive current amplifiers.) The sum of the outputs of all the 1st stage SQUIDS

in each column is fed into another 2 stages of SQUID amplifiers before passing out of the cryostat via the preamp card. The preamp card also provides biasing (power) for the 3rd stage of amplification (which is often referred to as the series array). Biasing for the 2nd stage of amplification is provided by the second of the bias cards. Each channel from the preamp card goes to a separate DFB card in the digital electronics box where the signal is digitized.

Although SQUIDs are very sensitive their response is also periodic. To avoid problems associated with this the NIST/Goddard electronics uses a feedback system to keep the output from the preamplifier constant. Using a digital PI algorithm, the DFB cards output a voltage that gets feed back all the way to the input of the 1st stage SQUID (via the feedthrough card) and cancel out any changes in the signal from the bolometer. The output from the electronics to the computer (via fiber optic) is the size of this voltage. As the system multiplexes, the DFB cards have to change this feedback signal in sync with the switching of the 'row select' lines. The clock and address cards keep everything in sync.

As the 2nd and 3rd stage SQUIDS also have periodic outputs it is important to set up the array correctly so that they are functioning on the most responsive parts of their curves. The region of the response curve they are operating at is adjusted by applying a voltage to the 2nd and 3rd stage feedback connections (see figure 2). These voltages are provided by the third and fourth bias cards in the tower. To find the correct values, the DFB cards are used to apply a triangle wave to the 1st stage circuits. The feedback/bias values that give the maximum signal can then be found. A method for carrying this out automatically is currently under investigation. (see below)

The rest of this document describes each of the components of the detector read out system, version 2.1. On the GBT we expect to be using version 3.0 which is an update to the firmware running the DFB boards. Further details are given in the section on the DFB. All other components will stay the same and the methods for talking to the boards will not change. For more information on time division multiplexing, see Benford et al. Multiplexed Readout of Superconducting Bolometers. International Journal of Infrared and Millimeter Waves. 21:(12) 1909-1916.

The PC

The heart of the system is a PC running Red Hat Linux version 7.3 with the bigphysarea patch and the kernel updated to version 2.4.20. These patches are necessary to allow a custom input card to use large physical areas of memory. The PC controls the cryogenics system via a GPIB card (see cryogenics_system_overview_2 document). The PC also contains 3 serial ports which are used to control the multiplexing electronics, as well as a PCI Fiber DIO card to read the multiplexed data. The PCI Fiber card streams data from up to 16 fiber links (8 used, 8 unused), each one coming from a digital feedback card.

Serial ports

The PC contains three serial ports, COM0, COM2, and COM3:

COM0 connects to the digital electronics box's clock card.
COM2 connects to the analog electronics tower's power card.
COM3 connects to a relay box and is used for calibration purposes which are not relevant here.

COM0 and COM2 use serial to fiber converters to transmit data over optical fiber in simplex mode.

The Analog Tower

Addresses of cards in the analog tower are burned into the PROMs. Because of this, obtaining replacement cards is more difficult once each card in the system is assigned its permanent address. Therefore, until we have a working setup we will not assign cards (with their permanent addresses) to specific functions.

Data written over COM2 to the analog tower consists of a 9-bit address, 3-bit subaddress, and a 16-bit data value. These 28 bits are packaged with 4 framing bits to produce 4 bytes, which are sent in the following fashion:

MSB b_7	b_6	b_5	b_4	b_3	b_2	b_1	LSB b_0	
0	d_6	d_5	d_4	d_3	d_2	d_1	d_0	Byte 1
0	d_{13}	d_{12}	d_{11}	d_{10}	d_9	d_8	d_7	Byte 2
0	a_1	a_0	sa_2	sa_1	sa_0	d_{15}	d_{14}	Byte 3
1	a_8	a_7	a_6	a_5	a_4	a_3	a_2	Byte 4

Where the **9 bit address** is reperesented as:
$a_8a_7a_6a_5a_4a_3a_2a_1a_0$
The **3 bit subaddress** value is represented as:
$sa_2sa_1sa_0$
And the **16 bit data** value is represented as:
$d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0$

Within each byte, the LSB (least significant bit) is sent first. Within each packet, Byte 1 is sent first.

Note that the MSB (most significant bit) of each byte sent over the serial port is a framing bit, with a 1 indicating the last byte of a transfer. Also note that communication is one way; there is no way for the PC to check that valid data has been received.

The exact meanings of address, subaddress, and value depend on the card being written to.

1) **DACs – bias and preamp cards**

Each of the DAC (digital to analog converter) cards (bias and preamp cards) contain 8 programmable voltage sources to properly biases various stages of amplification or provide the correct feedback voltage. In addition, the preamp card (which is connected to the 3rd SQUID stage) outputs an amplified version of its input to the SMB connectors, which gets routed to the DEB for sampling. The bias card's SMB connectors function as additional inputs which can be added to the DAC output, or used to monitor the voltage at the DAC output.

The meanings of the address, subaddress, and value fields are as follows:

Address	9 bits for the destination card address
Subaddress	Each DAC card has 8 channels indexed by these 3 bits
Data	A 16 bit DAC value corresponding to a voltage range of 0 (0x000) to 2.5 V (0xFFFF)

2) Address card

The meanings of the address, subaddress, and value fields are as follows:

Address represents the destination card address
Subaddress chooses one of several functions, which determines the meaning of **data**

Subaddress (binary)	Function	Data	High Bit	Low Bit
000	Program ILUT	LUT_ADDRESS	d15	d10
		PGM_DATA	d8	d4
001	Set Vref	14-bit DAC value	d15	d2
010	Set ILUT parameters	ILUT_DELTA	d15	d10
		ILUT_START	d8	d3
011	Start enable	All 1s for on, 0s for off	d15	d0
100	Set NSTEP	NSTEP	d7	d0
101	Enable address driver	All 1s for on, 0s for off	d15	d0

The address card contains a look up table (LUT) of 64 entries (address 0 to 63). Each entry contains a 5 bit word. The values stored are unsigned decimals. The values, which can range from 0 to 31, represent rows of SQUIDs. Upon reset, the system's LUT defaults to 0 through 31 for entries 0 through 31, and 0 for the remaining 32 entries. The user can choose to create a new LUT if desired.

The LUT determines which row of SQUIDs is accessed at any given time. The address card starts at **ILUT_START**, and progresses through the LUT, incrementing by **ILUT_DELTA**, addressing the row that has its number stored in the LUT. The card goes back to the start of the LUT after (**NSTEP**+1) rows have been addressed. "Addressing the row" means that the address card physically applies a voltage (**Vref**) to the bias line for that row.

For the Penn Array we wish to address rows 0 through 7, so the default LUT is correct. The other parameters should be set as:

ILUT_START = 0

ILUT_DELTA = 1
NSTEP = 7

The LUT can be changed using the **PGM_DATA** and **LUT_ADDRESS** fields. The value stored in PGM_DATA is stored at the LUT address pointed to by LUT_ADDRESS. So, if one wanted to change LUT entry 5 (address 4) with the value 25, one would program the card with the following data:

LUT_ADDRESS = "00100"
PGM_DATA = "11001"

Summary of parameters on address card:

- **LUT_ADDRESS** is a pointer to where in the LUT the data in **PGM_DATA** should be stored.
- **Vref** is the bias voltage used to 'turn on' 1 of 32 SQUID rows.\
- **ILUT_DELTA** is the amount by which the index is incremented.
- **ILUT_START** is the address to start indexing at.
- **Start_enable** enables/disables the Address Card to begin generating address data at its output.
- **NSTEP** is the number of transitions to make before going back to **ILUT_START** (also NSTEP = #rows – 1)
- **Enable address driver** allows the user to disable the Address Driver. It will not be used regularly.

The following sequence of bytes should be written to the analog tower to properly configure the address card for the Penn Array.

a8..a0 should be substitured with the address of the address card
x = don't care

b_7 (MSB)	b_6	b_5	b_4	b_3	b_2	b_1	b_0 (LSB)		
0	0	0	0	0	x	x	x	Byte 1	First 4 bytes set ILUT_DELTA and ILUT_START.
0	0	0	0	1	x	0	0	Byte 2	
0	a_1	a_0	0	1	0	0	0	Byte 3	
1	a_8	a_7	a_6	a_5	a_4	a_3	a_2	Byte 4	
0	0	0	0	0	1	1	1	Byte 5	Next 4 bytes set NSTEP = 7
0	x	x	x	x	x	x	0	Byte 6	
0	a_1	a_0	1	0	0	x	x	Byte 7	
1	a_8	a_7	a_6	a_5	a_4	a_3	a_2	Byte 8	

Vref must also be written, but this value has not yet been determined
After that:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0		
0	1	1	1	1	1	1	1	Byte 13	Start_enable
0	1	1	1	1	1	1	1	Byte 14	
0	a_1	a_0	0	1	1	1	1	Byte 15	
1	a_8	a_7	a_6	a_5	a_4	a_3	a_2	Byte 16	

The Digital Electronics Box

Each card in the DEB has an 8 bit DIP switch for setting the card address. The address assignments are shown below.

Card	Address
Clock	0
DFB 1	1
DFB 2	2
DFB 3	3
DFB 4	4
DFB 5	5
DFB 6	6
DFB 7	7
DFB 8	8

Data written over COM0 to the DEB consists of a 7-bit address, 3-bit register address, and a 25-bit data value. Each card has a unique 7-bit address, and each card may have up to 8 registers, which are indexed using the respective fields. The 25 bit data value is stored in the given register on the given card.

These 35 bits are packaged with 5 framing bits to produce 5 bytes, which are sent in the following fashion:

MSB							LSB	
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	
d_6	d_5	d_4	d_3	d_2	d_1	d_0	0	Byte 1
d_{13}	d_{12}	d_{11}	d_{10}	d_9	d_8	d_7	0	Byte 2
d_{20}	d_{19}	d_{18}	d_{17}	d_{16}	d_{15}	d_{14}	0	Byte 3
r_2	r_1	r_0	d_{24}	d_{23}	d_{22}	d_{21}	0	Byte 4
a_6	a_5	a_4	a_3	a_2	a_1	a_0	1	Byte 5

Where the **7 bit address** is reperesented as:

$a_6a_5a_4a_3a_2a_1a_0$

The **3 bit register** address is represented as:

$r_2r_1r_0$

And the **25 bit data** value is represented as:

$d_{24}d_{23}d_{22}d_{21}d_{20}d_{19}d_{18}d_{17}d_{16}d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0$

Within each byte, the LSB is sent first, and within each packet, Byte 1 is sent first.

Note that the LSB of each byte sent over the serial port is a framing bit, with a 1 indicating the last byte of a transfer. This is in contrast to the analog tower communications protocol, which uses the MSB for framing. Also, note that communication is one way; there is no way for the PC to check that valid data has been received.

1) Clock card

The time division multiplexing system uses 4 clock signals. CLK is the master clock signal, running at 50 MHz. LSYNC is essentially a row clock; it pulses after a full row of SQUIDs has been read, telling the address card to switch to the next row. FRM pulses once the last row has finished (a frame is one cycle through all the rows). ADR_CK runs at twice the rate of LSYNC and is used by the address cards.

The clock card is responsible for CLK and FRM. The DFB cards make their own copy of LSYNC. The frequency of FRM is equal to (50 MHz / clock_divider).

The clock card contains 2 25-bit registers, described in the table below.

address	field	High Bit	Low Bit
1	stop	24	24
	start	23	23
2	clock_divider	24	0

Field names and definitions:

- STOP and START bits are used at the very start of turning the system on to first make sure the card is started. If the card does not start correctly (as evidenced by the yellow light on the card not blinking), STOP must be sent, followed by START again. There is no way to check from the system if it started correctly. We do not know yet if this will be a problem for us.
- clock_divider determines the frequency of the FRM clock. This frequency is equal to 50 MHz divided by the number stored in clock_divider

Other important info:

- The 25 bits of Register 1 consists of only 2 meaningful bits.
- There is no register 0.

2) Digital Feedback Card

Each DFB card uses an ADC to sample the SA output. A proportional/integral (PI) control algorithm is implemented by digital electronics, and the output is converted back to an analog voltage through a DAC. The DAC output is connected to the bias line of the first stage of SQUIDs.

The DFB cards are also responsible for sending all the multiplexed data to the PC for storage on hard disk. This is accomplished through direct fiber links to the PCI Fiber card.

The DFB cards also contain a programmable triangle wave generator. When active, the DAC outputs a triangle wave to the SQUID feedback, which is used to observe the full

response of the SQUIDs to different feedback levels. By monitoring the output, the optimal feedback level can be determined. The cycles per step can be changed to determine the frequency of the wave, and the number of steps and step size to determine the height and resolution of the wave. The value of the wave changes at a frequency of (2*CyclesperStep*line rate), and it changes NumSteps times, giving the triangle wave itself a frequency of (2*CyclesperStep*line rate / (NumSteps+1)). For example, if the line rate is 1 MHz, CyclesperStep is 1, NumSteps is 3, and StepSize is 1, then you will have a 500 kHz triangle wave ranging between 8 and 0 in steps of 1.

The digital feedback card contains 8 registers. The register structure is shown below.

address	size	addrsize	field name	High Bit	Low Bit	Arrayed?	Signed?
0	25	3	2*lsync_clock_divider	d24	d0	Yes	No
1	25	3	ADC _offset / lockpoint	d15	d0	Yes	No
2	25	3	Triangle Disable	d16	d16	Yes	No
			DAC_offset	d15	d0	Yes	No
3	25	3	FB on/off	d24	d24	Yes	No
			pval	d19	d10	Yes	Yes
			ival	d9	d0	Yes	Yes
4	25	3	triangle cycles/step	d23	d20	No	No
			triangle # of steps	d19	d16	No	No
			tri_step_size	d15	d0	No	No
5	25	3	START_EN	d7	d7	No	No
			Row number	d6	d1	No	No
			send_mode	d0	d0	No	No
6	25	3	#samples	d19	d0	No	No
7	25	3	global_reset	d24	d24	No	No
			NSTEP	d14	d8	No	No
			settling time	d7	d0	No	No

Field names and definitions:

- 2*LSYNC_CLOCK_Divider will be used for a user-programmable clock divider to generate LSYNC and ADR_CK pulses. As mentioned earlier, the DFB cards generate their own LSYNC clocks and its frequency is (50 MHz / lsync_clock_divider). ADR_CK runs at twice the rate of LSYNC and is used by the address cards.
- ADC offset/lockpoint – the DFB card attempts to lock the output at the ADC lockpoint.
- Triangle Disable turns off the triangle wave generator
- DAC offset – offset applied to DFB card's DAC (for both feedback and triangle wave purposes)
- FB on/off – single bit indicating whether or not the feedback algorithm is enabled
- Pval – proportional gain for DFB PI control algorithm
- Ival – integral gain for DFB PI control algorithm
- Triangle cycles per step – Value of triangle wave will change at a frequency of 2*CyclesperStep * line rate

- triangle # of steps – Number of times to count up values of the triangle wave.
- triangle step size – Size of step being incremented
- START_EN – enables addressing by address card
- Row number determines which row the arrayed registers will apply to. See below for more info on arrayed registers.
- send_mode – chooses between 2 different modes for streaming data to PCI card (see section on PCI Data Streaming for more information)
  - Mode 0: D2A[13..0], OVERFLOW, FRAME, ERROR[15..0]
  - Mode 1: ERROR[15..0], FRAME, OVERFLOW, D2A[13..0]
- # of samples – number of samples over which to average when calculating error signal – higher number means slower, but more accurate, response
- Global reset – starts the muxing over at the first row
- NSTEP – number of row transitions per frame, i.e. the number of rows being MUXed = NSTEP+1
- Settling_time – number of clock cycles to wait, and let input settle, before sampling input to calculate feedback

Important items:

- pval and ival are signed numbers, in 10-bit 2's complement notation
- Registers 0-3 are considered "Arrayed registers". Each DFB card controls a column, and thus must have room for different DAC offsets, P/I values, etc., for each of the rows it accesses. To write an arrayed value to a specific row, first set the row number (register 5) and then write any necessary arrayed registers. On the Penn Array, each DFB card will carry 8 different values for each of the fields in registers 0-3.
- Registers 4-7 are considered "Non-arrayed registers" in that each DFB card has only one set of values for these.

As of 14/8/2003 we expect to be updating the firmware in these cards. Changes include

- The PI feedback algorithm – a different equation & parameters will be used.
- Additional choices of waveform output (sine, sawtooth…..)
- Including clock pulses/frame numbers in the output data.
- Other additions to the output data.

The method of communication will be the same but some of the values to be passed will have to change.

3) Interface Card

The interface card interfaces the DEB to other systems. It provides the address card with a 2*LSYNC clock, as well as the FRM sync signal. It also has 4 general purpose analog inputs and 4 general purpose digital inputs. These inputs would come from an external system, on a separate ground plane. The data on these lines is synchronized with the Mark III data stream and sent into the PCI card.

Register	Field	High Bit	Low Bit
1	2*LSYNC_CLOCK divider	24	0
2	Enable ADC #1	24	24
	ADC #1 Ch. 0 conversion time	23	17
	ADC #1 Ch. 1 conversion time	16	10
3	Enable ADC #2	24	24
	ADC #2 Ch. 0 conversion time	23	17
	ADC #2 Ch. 1 conversion time	16	10
4	Status Dump enable	24	24

The ADC used in the Interface Card is the Analog Devices AD7732. The firmware will automatically configure it for a ±5V input range, No Chopping, and 24-bit Continuous Conversion Mode. These parameters are not user-configurable at this time. Sampling Rate for all four channels will be user-configurable. To calculate the sampling rate, use the following formula, where FW is the 7-bit Conversion Time parameter:

$$(((FW * 64) + 207) / 6.144) \text{ MHz}$$

The PCI Data Stream

DFB Cards

There are 2 possible modes for streaming from the DFB cards, and they produce slightly different data structures (they only differ in bit order).

- Mode 0: D2A[13..0], OVERFLOW, FRAME, ERROR[15..0]
- Mode 1: ERROR[15..0], FRAME, OVERFLOW, D2A[13..0]

The data is arranged from MSB to LSB, and is left-shifted out of the fiber link, so that the MSB is sent first. Exactly how the PCI card processes this is unclear right now. We know that FRAME becomes the first bit of the third byte written to the file, but the mode (0 or 1) for that situation was unknown, so either Mode 0 is written to file from right to left or Mode 1 is written LSB first.

Each data packet is 32 bits:

- The 14 bit **D2A** value represents the DAC output of the DFB card at that instant
- The 16 bit **ERROR** signal is the difference between the ADC lockpoint and the ADC sampled input (not an error code in the sense of a mistake, but in the sense of a difference between the desired value and the actual value).
- **OVERFLOW** is a single bit indicating whether or not the card overflowed (i.e., parameters should be changed or data disregarded)
- **FRAME** is a single bit indicating the start of a new frame of data

Interface Card

The interface card sends its digital and analog inputs to the PCI card. The bit stream consists of 2 32-bit words.

- Word 0: 0, FRAME, 0, 0, DIG[3..0], ADC0[23..0]
- Word 1: 0, FRAME, 0, 1, DIG[3..0], ADC1[23..0]

The data is arranged from MSB to LSB, and is left-shifted out of the fiber link, so that the MSB is sent first. Exactly how the PCI card processes this is unclear right now.

The 64 bits are relay the following information:

- **FRAME** is a single bit indicating that frame_sync is high
- **DIG[3..0]** are the 4 general purpose digital inputs.
- The 24 bit **ADCx** signals are the general purpose analog inputs.

SQUID Biasing Algorithm

We have been told there may be an algorithm in development that will automatically bias all of the SQUIDs properly. Until then, properly biasing the SQUIDs is a long procedure that must be repeated for each SQUID, and is not something the user should have to do each time.

Currently, the procedure, in the context of the python GUIs, is as follows. Note that the following procedures work with an older version of the firmware and will not work in exactly the same way with our hardware, but this is a general idea of the necessary steps.

Turning on system – muxclock.py:
In the 'sync' tab: Issue stop command, then start command. Yellow light should blink (not go all the way up and remain on).

Basic procedure for properly biasing a SQUID – use dfbmuxgui.py (for DEB communication) and SQUID2.py (for analog biases):

1. Turn a triangle wave generator on and send this to the flux feedback (FB) line for the desired SQUID.
2. View series array (SA) output on scope – looks nearly sinusoidal.
3. Change DAC offset level to maximize amplitude without clipping. Princeton reports 40000 to be approximately correct.
4. Measure DC value of point on curve with greatest slope – pretty much the center of the sinusoidish output.
5. Remove the triangle wave and set the voltage on the bias card connected to the feedbackline so that the DC out is at approximately the same point as the center (biased at steepest point of curve) – this biases the SQUID correctly.

There are three stages of SQUIDs – the 1st stage, 2nd stage, and SA. To bias a full set of 3 SQUIDs the SA should be done first, followed by the 2nd stage and finally the 1st stage.

At this point, the DFB cards must be used to lock the feedback – use rack_addrboard.py

1. Select address
2. Width = 60 (we have no idea what that means, but NIST said so)

3. ADC high and ADC low should be approximately 14,200 – when ADC low is set low, the output gets very noisy
4. On scope, we should see triangle wave and that vaguely sinusoidal shape again from the SA.
5. We want to lock the feedback to the steepest point of the curve. There are four controls to adjust: DAC offset, ADC offset, P, and I
   i. DAC offset should be changed first. It ranges from 0 to 16,383. Adjust until the DAC offset is at the center (steepest part) of the curve like in previous steps.
   ii. Next find ADC offset. This is calculated by finding voltage at center of curve (where we biased using the DAC offset), then multiply by (2^{12} * number of samples per detector (default to 16)) = (4,096 * samples/detector)
   iii. Next choose P and I values (proportional and integral values for PI controller). Princeton uses 50 and 50 as defaults. Then, attempt to lock channels (click lock, then send register).
   iv. If overflow light (top light on address card – can also be read in PCI data stream), lower P and I values (10 and 10 often work).
   v. Try lock and send again. If more trouble, increase settling time.

Stream data to PCI card – pci_mux_control.py

1. Fill in appropriate fields – channel #s, # of buffers, delay, etc.
2. Click 'save' to stream data to disk

PCI Fiber DIO Board Device Driver

All information in this section has been obtained via reverse engineering the driver software, and has not been confirmed by anyone involved in the design of the board or driver.

NIST has provided a driver for the PCI Fiber DIO board. The driver is written in C and is available as a C header file (aa.c) or a Python module (AA).

Opening communication with PCI board

The open() function is used to obtain a file handler for the PCI board. The "file" to open is "/dev/ab0", and should be opened with "rwb" (read/write binary) permissions.

Driver capabilities

The driver contains implementations of the following standard functions:

- open
- poll
- ioctl
- release
- mmap
- fasync

Of these, the only one requiring further explanation is ioctl, since the rest are standard.

ioctl functions provided by driver

The ioctl function is actually an interface to several other, device-specific functions. All the ioctl functions used by the NIST software are listed below, along with their meanings.

Important: Some of these functions accept arguments. For all functions that require arguments, except ABIOCONFIGURE and ABIOSTART, the meaning of the argument is unclear, but the programs we have always pass 0 to the functions.

- ABIOGA2PBUFFER – returns the address of the start of the buffer – accepts argument
- ABIOGBUFFERSIZE – returns the size of the buffer (in kB) – accepts argument
- ABIORESET – resets board and counter in the driver
- ABIOCONFIGURE – writes a configuration word, passed as an argument, to the board – there is not much info on structure of configuration word – see below for more info
- ABIOSTART – begins the DMA (Direct Memory Access) transfer from the board – the argument passed is some form of configuration word
- ABIOSTOP – stops DMA transfer
- ABIOGMWTC – returns the number of buffers left in the transfer –accepts argument
- ABIOGBUFFNUMBER – returns the current buffer number – accepts argument
- ABIOGCOUNTER – returns the current counter value – accepts argument

The following ioctl functions exist in the driver but are not used in any of the sample software we have. Where possible, their purpose is noted. Any arguments are noted in the names.

- ABIORESETADDON – resets the add-on – we don't know what the "add-on" refers too, though
- ABIOSETLATENCY(arg) – sets the PCI latency timer to the value of arg, where 0x20 ≤ arg ≤ 0xF8
- ABIOSETNUMBUFFERS(arg) – sets the number of write buffers to arg
- ABIOSETBUFFERSIZE(arg) – sets the size of the buffers to arg
- ABIOGMCSR(arg) – prints the Bus Master Control Register information to the kernel log file
- ABIOGINTCSR(arg) – prints the Interrupt Control Service Register information to the kernel log file
- ABIOGINFOADDR(arg) – unknown
- ABIOGP2ABUFFER(arg) – get some value from PCI card called pd_phys_read – function is unknown
- ABIOGMRTC(arg) – ABIOGM**W**TC returns the number of buffers left to write; perhaps this returns the number left to read?

- ABIOGNUMBUFFERS(arg) – returns the number of buffers in the transfer, in total
- ABIOGCPUTIME(arg) – unknown

ABIOCONFIGURE/ABIOSTART Configuration words
No information is available on the precise meaning of most of the bits in the configuration words, written using ABIOCONFIGURE. However, the meaning of several complete words has been determined in context. Each configuration word is 4 bytes long, and the known ones are shown below in hex, along with meanings.

0x80000000 – clear FIFOs – use before stopping card
(0x20000000 + channel_mask) – see next line
(0x40000000 + channel_mask) – these 2 commands, used consecutively, "setup the streaming" for a given channel – the channel mask is constructed by assigning each of the 16 lowest bits in the word to represent a channel on the PCI card. For example, if bits 0, 1, and 3 are set, then the PCI card will read data from Channels 0, 1, and 3.
0x80000000 + (delay * 0x10) – call immediately before issuing start command (with ABIOSTART) – the delay may range from 0 to 9 (default is 8)

Note that the last 3 words above are used by ABIOSTART in one of the sample programs, the one showing how to stream from multiple channels. There is a remote possibility that this is a mistake and they should be passed to ABIOCONFIGURE.

ABIOSTART accepts one additional word:

0x80000001 + (delay * 0x10) – tells PCI board to actually begin the data streaming

Using ioctl
ioctl is available in C, Python, and probably many other languages.

ioctl in C:
Header files needed:

```
#include <unistd.h>
#include <stropts.h>
```

Function prototype:

```
int ioctl(int fildes, int request, arg);
```

fildes here is the file descriptor to the PCI board
request is the name of the ioctl command, taken from the previous list
arg is the optional argument that some of these functions require.

If an invalid request or argument are passed to ioctl, (-EINVAL) is returned. If the particular command is supposed to return something, like ABIOGBUFFERSIZE (gets the buffer size), then that value is returned.

For example

```
/* assume fd is the file handler for the PCI card */
ioctl(fd, ABIOSTOP)       /* stops card */
bufsize = ioctl(fd, ABIOGBUFFERSIZE, 0);      /* 0 is passed as an argument */
```

Calling ioctl in Python
Modules needed:

```
import fcntl
```

In Python, ioctl is a member of the fcntl class. Actual file descriptors may be obtained from Python's higher level descriptors using the fileno() member function. Arguments to ioctl functions must be packed into a Python structure as an unsigned integer, which exposes the raw bit format of the data. Returned data must similarly be unpacked.

For example:

```
# assume fd is the file handler for the PCI card – to stop the card, use the following line
fcntl.ioctl(fd.fileno(), AA.ABIOSTOP)

# using an ioctl that requires an argument is more complicated:
# First, the argument (0 in this case) must be packed into a structure in bit form
arg = struct.pack("L", 0);
# Next, call the actual ioctl– note that it is from the AA module, and must be
# accessed as such (with AA. in front of it)
msg = fcntl.ioctl(fd.fileno(), AA.ABIOGBUFFERSIZE, arg);
# Finally the return value must be unpacked
data = struct.unpack("L", msg);
buffer_size = data[0];
```

Other functions necessary for using driver:
mmap – creates a memory map which allows us to read from an area of memory as if it were a file.

mmap in C:

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fildes, off_t offset);
```

mmap in Python:

```
import mmap
mmap.mmap(fildes, length, flags, protection);
```

The only important distinction between these two is the presence of the *addr argument in C. I do not know how this should be handled. Also, in Python, flags

and protection, along with the mmap function itself are all defined as members of the mmap class.

signal – links a specific signal (interrupt) with an interrupt handler function.

signal in C:

```
#include <signal.h>
void (*signal (int sig, void (*disp)(int)))(int);
```

signal in Python:

```
import signal
signal.signal(sig, handler)
```

Note that in Python signal is a member of the signal class. The PCI Fiber board asserts the SIGIO signal (signal.SIGIO in Python).

fcntl – fcntl is very similar to ioctl, but contains built-in functions. It is needed to finish configuring the interrupt handler, after the signal() call.

fcntl in C:

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fildes, int cmd, /* arg */ ...);
```

fcntl in Python:

```
import fcntl, FCNTL
fcntl.fcntl(fildes, cmd, arg);
```

Like ioctl, fcntl calls the "cmd" specified on the file descriptor. Relevant fnctl commands are F_SETOWN, F_GETFL, F_SETFL. fcntl's return value depends on the command being passed.

F_SETOWN sets the ownership of a process. We use it, immediately after configuring signal() with its handler, to finish the configuration.

In C:

```
fcntl(fd, F_SETOWN, getpid());   // getpid() is a function that gets the process ID
```

In Python:

```
fcntl.fcntl(fd.fileno(), FCNTL.F_SETOWN, os.getpid())
```

F_GETFL and F_SETFL get and set, respectively, the flags associated with a file descriptor.

In C:

```
flags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, flags | FASYNC);  // set flag for asynchronous notification
```

In Python:

```
flags = fcntl.fcntl(fd.fileno(), FCNTL.F_GETFL);
fcntl(fd.fileno(), FCNTL.F_SETFL, flags | FCNTL.FASYNC);
```

Reading Data

All information in this section has been obtained via reverse engineering the NIST Python software, and has not been confirmed by anyone involved in the design of the board or driver.

The PCI Fiber DIO card reads data from the DFB cards via direct fiber links. Each channel on the PCI card contains a memory buffer that, when full, causes the card to issue a SIGIO interrupt. Software must initialize the card for data streaming, listen for interrupts, read the data from the card and write it to a disk file, and finally stop the streaming.

General Outline

1) Open a PCI device handler, memory map, and data storage file
2) Prepare an interrupt handler for when the PCI card buffer is full
3) Reset the board
4) Setup the channels for streaming
5) Start the DMA to read from the PCI card
6) When buffer is full, interrupt handler is called, which reads from the right memory map location, and write this data to the data file, and repeats DMA as necessary until streaming is complete.
7) When finished taking data, clear FIFOs on card and stop streaming

Detailed Steps – Essentially a line by line explanation of the Python streaming script – all specific functions referred to in this section are from Python, though in most instances there is a near exact correspondence with C functions. See above for more information on necessary functions. In most cases, C functions can be obtained simply be removing the class membership operation in functions (instead of fcntl.ioctl() just use ioctl(), instead of mmap.MAP_SHARED just use MAP_SHARED).

1. Open a PCI device handler, memory map, and data storage file
   - The PCI device handler is found at "/dev/ab0". It should be opened with mode "rwb" (read/write binary).
   - The memory map file descriptor (*memfd*) is located at "/dev/mem". It should be opened as a file, with os.O_RDWR as the mode (read/write).
   - Create a memory map (mem) to *memfd* using the *mmap* command.
     - Memory map size = buffer_addr + total_size.
       - buffer_addr is obtained from the ABIOGA2PBUFFER ioctl
       - buffer_size in kB is obtained from the ABIOGBUFFERSIZE ioctl
       - total_size = buffer_size in bytes * 2 /* purpose is somewhat unclear – we suspect that instead of 2, this number should = # of columns being MUXed*/
     - Flags = mmap.MAP_SHARED /* this allows other things to access the mapped memory */
     - Protection = mmap.PROT_READ | mmap.PROT_WRITE. /* read/write mode */

- Create a file handler for the data file. Name and location may vary. Open in "wb" mode (write binary)

2. Prepare interrupt handler
   - When the PCI board's buffers are full, it asserts a SIGIO signal, so before that we must link the SIGIO signal to our interrupt handler using the *signal()* function
   - Set ownership on these interrupts using the F_SETOWN command to the *fcntl()* function. The first argument should be the file descriptor for the PCI board. The last argument should be os.getpid() to get the proper process ID.
   - Set necessary FCNTL flags for the PCI file descriptor.
     - Get current flags using the FCNTL.F_GETFL argument passed to *fcntl()*
     - Set the flags to the current flags ORed with FCNTL.FASYNC to allow asynchronous notification.
3. Reset the board
   - Call the ABIORESET *ioctl* function.
4. Setup the channels for streaming
   - Build the channel mask
     - The channel mask is a bit mask with a 1 for each channel to include in the DMA, and a 0 for each channel not to.
     - Example 1 – to stream just to channels 0 and 1, channelmask=$0000\ 0011_b$
     - Example 2 – to stream channels 0-7, channelmask=0x00ff
   - Call ABIOSTART *ioctl* with 0x20000000+channelmask as argument /* there is a *remote* possibility that this should actually be ABIOCONFIGURE */
   - Call ABIOSTART *ioctl* with 0x40000000+channelmask as argument /* there is a *remote* possibility that this should actually be ABIOCONFIGURE */
5. Start the DMA to read from the PCI card
   - Calculate delayBits = "Delay" value (range 0-9, default 8) * 0x10
   - Call ABIOSTART ioctl with 0x80000001+delayBits
6. Interrupt handler takes over
   - Get the buffer number using the ABIOGBUFFNUMBER *ioctl.* Pass 0 as its argument.
   - Position the memory map pointer at the right place.
     - If buffer number = 1, position memory map at buffer_addr (using *seek()* function)
     - Else, position memory map at buffer_addr + buffer_size (the required offset)
   - Read a chunk of data of size buffer_size from the memory map using *read()*
   - Write this data to the disk data file.
     - If first_buffer=TRUE (which it is initialized to):
       - Find "frame offset" by proceeding through the first set of streamed data and examining the frame bit (see data stream section for more info). "Frame offset" = index of the packet containing a set frame bit (e.g., if the 3rd packet of streamed data contains a set frame bit, frame offset = 3). Note that the first two packets should be skipped since "the first few words are sometimes funny".
       - Set first_buffer=FALSE
       - Write data to disk file at offset

        - Python command used is data_fd.write(buffer(msg, self.offset)) – the buffer() function seems to convert the data into a set of chunks of address and data, with the offset added to the address – this was the only line of Python code that I could not decipher, and I felt it better to include the original code than to make a guess and possibly lose information
        - buffer_written = 1
    - Else if buffer_written != num_buffers
        - Write data to disk file – no offset needed since the file pointer is in the right place
        - Increment buffer_written
    - Else
        - Write data to disk file using an offset – Python command is data_fd.write(buffer(msg,0,self.offset)) – the meaning of this 0 is unclear
        - Stop streaming (go to step 7)

7. Stop streaming
    - Clear FIFOs using ABIOCONFIGURE *ioctl.* Pass 0x80000000 as the argument
    - Stop card using ABIOSTOP *ioctl.*

<u>Existing Software for SQUID biasing, multiplexing, and feedback</u>

NIST has provided us with many Python scripts that control the analog tower, digital box, and can stream data from the detectors to the hard disk. The capabilities of each are described below.

AA.py – PCI card driver module

addr_board_dfb.py – programs the address card in the digital box – should not be run directly, but rather imported as a module for rack_addrboard.py – note that we will be using a newer address card in the analog tower, which makes the functioning of this script questionable

cardclass.py – programs the “baseline” card (from an older version of the digital box), which we do not have or need

clockcard.py – sets up the clock card in the digital box – muxclock.py seems to be a more complete version

dfbmuxgui.py – programs the digital feedback cards in the digital box

dfbmuxgui2.py – same as dfbmuxgui.py, with the added feature of a virtual oscilloscope that allows the user to click on the scope and automatically tune the SQUIDs to a given point

muxclock.py – sets up the clock card in the digital box – more complete than clockcard.py

params.py – contains a few variable definitions which initialize some global variables for data streaming programs

pci_mux_control.py – controls data streaming from up to 16 channels over the PCI fiber card, writing the information to disk

pci_writex_mux.py – reads a single buffer full of data from a single channel of the PCI fiber card, writing the information to disk

pci_writex_mux_multiple.py – reads many buffers full of data from a single channel of the PCI fiber card, writing the information to disk

pyserial.py – defines some functions used to transmit data over the serial port – imported as a module by squid2.py

rack_addrboard.py – configures address board in digital box – like addr_board_dfb.py, might not be useful for us since we have a different version of the address card – will have to be tested once the hardware is functioning

rack_baseline_mux.py – configures "baseline" card – unnecessary for us

squid2.py – communicates with analog tower, setting bias voltages

Nomenclature

ADC – Analog to Digital Converter
CLK – Master Clock
DAC – Digital to Analog Converter
DEB – Digital Electronics Box
DFB – Digital FeedBack
DIO – Digital Input/Output
DMA – Direct Memory Access
FIFO – First In / First Out
FRM – Frame Clock
LSB – Least Significant Bit
LSYNC – Line Synchronization Clock
LUT – Look-Up Table
MSB – Most Significant Bit
MUX – MUltipleX
PCI – Peripheral Control Interface
PI – Proportional + Integral control
PROM – Programmable Read-Only Memory
SA – Series Array

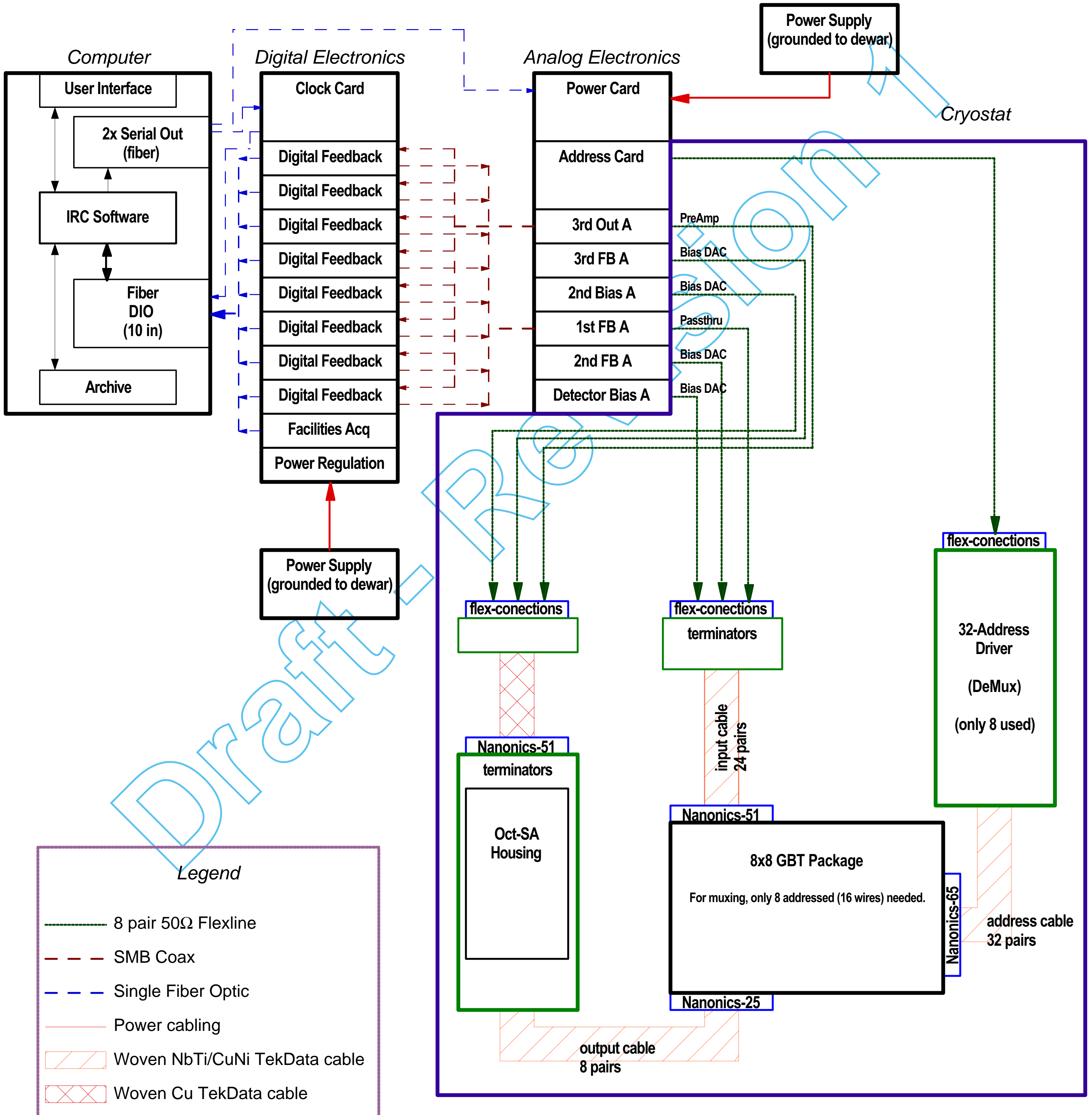
SQUID – Superconducting QUantum Interference Device

Notation

0110_b – the number 0110 is in binary

0x3F – the number 3F is in hexadecimal

Penn/GBT: 8x8 Array Electronics Block Diagram



Draft - Revision 1

Schematic for GBT 8x8 TES Bolometer Array

May 21, 2002 - Dominic Benford