Ba$\overline{\text{B}}$ar Note # **282**
April 12, 1996

The DAQ System for the Ba$\overline{\text{B}}$ar Calorimeter Beamtest, October 1995.

P.D. Dauncey, N.I. Geddes,
Rutherford Appleton Laboratory

Abstract

A description of the DAQ system used for the 1995 calorimeter beamtest is presented. The beamtest took place at PSI, Zurich in October 1995. The hardware and software used in the DAQ are documented and the data format and experience with the system are also described.

Contents

1 Introduction

This report serves several purposes

1. To describe the hardware and software associated with the (RAL part of) the BaBar calorimeter 1995 beamtest DAQ.
2. To serve as a short User Guide for the beamtest DAQ system.
3. To document the data structures used.
4. To document our experience of using LynxOS.

Throughout, things which are entered at the keyboard are shown in typewriter style; for example

```
more ~daq/daq/src/daq_readout.c
```

2 Basic Shift Instructions

The system runs the LynxOS operating system, which to a user looks very similar to Unix. When on shift, log on as user daq (ask someone for the password) using the Falco terminal. Two commands allow you to control and monitor everything; `daq_menu` will start the run control menu and `daq_hist` the run monitoring menu; see section 3 for details.

2.1 Run Control

The daq_menu program gives an easy interface to the DAQ control and this should always be used. In normal use, only starting and stopping runs is required, although after crashes or power-downs, the DAQ must be reinitialised (see below).

2.2 Emu Messages and Errors

Some of the DAQ programs give information messages. These are routed using a utility called emu to the VT320 terminal. In addition, the latest messages can be displayed on any terminal, using daq_menu.

Error messages are also routed in the same way, so it is important to keep an eye on the VT320 terminal.

All messages are logged into a file /tmp/emulog, which is renamed every few days to emulog_1995_mm_dd (where mm and dd are the month and day) and a new emulog is started. These log files are flat files and so can be read directly, e.g. using `more`, but a more user-friendly output can be obtained by using the `daq_reademu` command (see below).

2.3 Using the Exabyte Tape Drive

In normal use, data should be written to exabyte, not disk. However, there are a few things to be careful of when using the tape.

The tape is always rewound when it is inserted or removed from the drive, so it must remain in the drive between runs. It can take several minutes to rewind a complete exabyte tape. A run must not be started while the rewind is in progress, so always wait for the green light on the drive to come on before trying to write to the tape. Also, the run must always be stopped before removing the tape.

There is no way to tell if a tape is almost full, so the person on shift has to be careful not to put too much data on each tape. It is much better to have many partially filled tapes than to lose data from a run. The data can be concatenated later by copying onto other exabytes.

Each event is about 300 bytes, so 3M events are 1 Gbyte (which is the total size of the disk data area). At 100Hz, this is about 8 hours of data-taking. The exabyte drive is only low density and so a tape can store 2 Gbytes or 6M events (16 hours) of data. If we take 2M events in a standard run, then it would be sensible to change the tape after every such run or two runs.

2.4 Bookkeeping

As part of the shift duties, records of the runs written to disk or tape should be kept. Each run with output enabled creates a small flat file, /tmp/PRT.nnnnnn, where nnnnnn is the run number, which has the basic information of date, time, numbers of events etc. These files are human readable, e.g. using `more`. Although there is no printer attached to the DAQ system while at PSI, they can be printed directly from the Alpha workstation psw236, to a printer in the PSI computer centre, using

```
vpp /mnt/psfe90.psi.ch/tmp/PRT.nnnnnn
```

2.5 Cold Start or Reboot

Occasionally, the system may get into a mess and will need to be reboot'ed. There are two ways to do this; by resetting the VME crate or by typing `reboot`. The advantage of the latter method is that using options, you can speed up the reboot sequence. Resetting off the crate, by pressing the "Cold Start" button (or powering it off and on, which should only be done in emergencies), will perform many checks on startup, particularly fsck on the disk, which takes about 10 minutes. The command `reboot -a`, for comparison, will take about 1 minute. Note, `reboot` can only be issued by user root.

After a reboot, follow these steps:

1. Log on as user daq.
2. Type `daq_menu` to start the run control menu.
3. Type `1` "Initialisation" to go to the initialisation menu.
4. Type either `2` "Initialise All for Disk" or `3` "Initialise All for Tape" to start all the necessary processes.
5. Type `1` "Check Processes" to check things have started correctly; daq_readout, daq_monitor, daq_check and daq_todisk or daq_totape should be running.
6. Type `X` "Return to Main Menu".

It should now be possible to start a run.

3 User Interface

The user interaction with the DAQ system is through two simple menu interfaces, daq_menu and daq_hist.

3.1 daq_menu

This is the run control menu and is evoked by typing `daq_menu` when logged on as user daq. It operates by running the DAQ programs, listed in section 6, as separate processes, using system calls from C in response to the user's requests. It therefore does exactly the same as directly executing the programs. Section 6 should be consulted for details of the programs refered to in the following.

3.1.1 Main Menu

The first four items in the main menu are for run control, the next five are for monitoring the run and the last is to clean up after a crash. "Initialisation" and "Run Start" go to submenus, described below. "Run Pause (Toggle)" runs daq_togglepause which will pause a run or resume a paused run. "Run Stop" runs daq_runstop which causes the run to finish after the next event.

3.1.2 Initialisation Menu

Normally, only one of the second or third items need be selected. These start up all the necessary DAQ processes for either disk writing or tape writing.

The other items allow the user to restart selected processes if they have crashed and the user is sure it is safe to restart them. In general, this is not to be recommended; it is better to close down the DAQ completely, using "Closedown" in the main menu, and then reinitialise using one of the first two items in the Initialisation menu.

3.1.3 Run Start Menu

The first seven items each start a run of a particular type; see section 8 for details. The other two items allow the user to enable or disable data output (to disk or tape) and to move the stage. These last two must be performed before starting the run.

3.2 daq_hist

Displays the histograms being filled by daq_monitor. This program actually copies the histograms into HBOOK format for display via HPLOT.

It is evoked by typing `daq_hist` when logged on as user daq and should, in principle, work from any workstation or terminal, if the correct identifier is provided. The identifiers are those used by PAW; some examples are `1.hepvsk.rl.ac.uk` for a workstation, `7878` for a Falco and `7879` for an X-Window session.

Making a hardcopy of the plots is not possible directly online as there is no printer while at PSI. However, every menu has a “Send Next Plot to PS File” button which will create a postscript file from the next histogram selected. The file is written to /tmp/Hnnnnnn.ps, where nnnnnn is the histogram number. These files can then be printed from the Alpha workstation psw236, to a printer in the PSI computer centre, using

```
vpp -device=pts /mnt/psfe90.psi.ch/tmp/Hnnnnnn.ps
```

3.2.1 Data Histograms Menu

These plots are for the ADCs, TDC and wire chamber data. They are reset at the start of every run.

3.2.2 Calibration Histograms Menu

These plots show the results of a calibration run. They are only reset at the start of a calibration run and so can be viewed even if a data run has been started after the calibration run.

3.2.3 Slow Controls Histograms Menu

These plots show the temperatures. They are never reset, although stopping daq_monitor deletes the mbook shared memory, so the plots only contain values since daq_monitor was last restarted.

4 Hardware

The beamtest data-acquisition sytem is based around a Motorola MVME167 single board computer. This board has a M68040 CPU running at 25MHz, 16 Mbytes of on board memory and interfaces to the VME backplane, an external SCSI bus with a 2 Gbyte disk and exabyte drive, a printer port and up to four terminal ports. The VME crate also houses a CES CBD8210 CAMAC branch driver a Struck STR750 FADC controller, a STR755 8-channel FADC card and a VMEbus display module. The CAMAC branch driver connects to a single CAMAC crate containing a GEC CC2405 crate controller, a NE 9088 clock generator, an SEN OR2088 16-channel output register, a LRS 2228A 8-channel TDC, four SLAC ADC-32 32-channel ADCs, a LRS 4299 Databus Interface and a JEI DAC-16 16-channel DAC. There is also a NIM crate housing associated trigger logic modules.

4.1 MVME167 Computer

The MVME167 CPU runs the LynxOS real time operating system. To the user this looks just like a "standard" UNIX operating system. User accounts can be obtained from Neil Geddes or Paul Dauncey. The 2 Gbyte disk is configured as 3 separate partitions. The first (250 Mbytes) holds root file system (/), the second (750 Mbytes) holds the user files (/usr) and the third (1 Gbyte) is intended as the storage and staging area for beamtest data (/data).

Telnet access to the MVME167, called psfe90.psi.ch while at PSI (and hepvme6.rl.ac.uk while at RAL), is possible but currently it is only available from computers on the PSI site. In addition, the /usr, /tmp and /data areas are nfs mounted on the PSI psw236 Alpha machine and are visible as /mnt/psfe90.psi.ch/usr etc.

The default LynxOS boot sequence for the MVME167 has been modified slightly to tailor it to our requirements. Specifically:

1. The Local to VME memory map has been modified as indicated in table 1.

2. Device drivers for the CAMAC branch driver (cbddrvr) and VME interrupts (vmetrig) are installed.

3. The portable error message utility EMUX is started.

The file /usr/daq/bootfiles/BaBarboot contains the system specific boot sequence, which is executed by the system boot file (/bin/rc).

Local start	Address end	(VME) Physical start	Address end	Access
0x00000000	0x000fffff			On Board DRAM
0x00100000	0xcfffffff	0x00100000	0xcfffffff	VME A32/D32 (Supervisor Data)
0xd0000000	0xdfffffff	0x00000000	0x00ffffff	VME A24/D16 (Supervisor Data)
0xe0000000	0xefffffff	0x00000000	0x00ffffff	VME A24/D32(Supervisor Data)
0xfff00000	0xfffeffff			On board Local I/O devices
0xffff0000	0xffffffff	0x00000000	0x0000ffff	VME short i/o A16/D32

Table 1: VME Memory Map for MVME167

4.2 CAMAC Branch Driver

The CES CBD8210 is configured as branch 7, via the thumbwheel on the front. This module allows CAMAC cycles to be executed under the control of the MVME167. The recommended method is to use the CAMAC library provided (see section 5) which implements all the standard CERN CAMAC routines. A front panel lemo input (labelled "INT2") gives a VME interrupt which is handled by the MVME167. All communication and interrupt handling is handled via a LynxOS device driver, cdbdrvr. The CAMAC device file is /dev/cbd though users should not need to access (open and close) this device explicitly as this is handled by the library routines.

4.3 CAMAC Modules

The SLAC ADCs have a 12-bit range and each bit corresponds to approximately 80 KeV for the high gain preamp channels and 2 MeV for the low gain channels. This gives about 320 MeV and 8 GeV full-scale in these ranges.

The TDCs have approximately an 11-bit range and are set to 50 psecs/bin. This gives a full scale range (around 2000 counts) of approximately 100 μsecs. Times over that (or no input) have the 12th bit (overflow) set in addition and so give a value around 4000 counts.

The wire chambers are read out via the LRS 4299 databus interface. No analogue information is read out from the wires; the data are a list of wires above threshold.

The DAC has a 12-bit range covering $\pm$10 V.

5 General Software

All the software required for the data aquisition is installed under the daq account. This software can be divided into two groups; software which has been installed and possibly tailored slightly but not written specifically for this system, and software which has been written specifically for the DAQ. The former are described in this section, while the latter are described in section 6.

5.1 Installed Packages

Most self contained tools/libraries/packages can be found in /usr/daq/products, however all compiled libraries can be accessed via links in /usr/daq/lib and required header files can be found in /usr/daq/include (package internal header files not normally required by users must be found in the appropriate products subdirectory). Where available, documentation for these various packages is linked to /usr/daq/doc. The products currently installed include:

- **camac**

  The lynxOS driver (cbddrvr) and function library (libcamlib.a) for the CES CAMAC branch driver. There is no specific documentation for this library but the CERN/ESONE CAMAC library writeup contains a description of all the function calls. Note, this writeup specifies the FORTRAN calling sequences and the C implementation is the same under LynxOS, hence, requiring pointers ("calling by reference") for most function arguments.

- **emux**

  Machine independent Error Message Utility programs and function library (libemu.a and msg_srs.o). The default message decoding and routing files, loaded at boot time, can be found in /usr/daq/config. There is a postscript format writeup, emux.ps.

- **f2c**

  Utility for converting fortran source code into C. There is no FORTRAN compiler on LynxOS. HBOOK and ZEBRA have been installed by first converting the source code to C using f2c and compiling with the standard C compiler. The executable is in /usr/daq/bin and full documentation can be found in /usr/bin/doc/f2c.ps.

- **mbook**

  A locally written histogramming package which fills simple arrays rather than full hbook histograms. The function calls available match closely the common hbook ones; mbook1(), mfill() etc. This is used for the monitoring histograms as it allows the use of shared memory, so a user can see the histograms from a separate process.

- **menu**

  A CERN package of C functions which may be used to construct screen menus on dumb terminals.

- **pbm**

  The Portable Buffer Manager. Derived from the buffer manager routines used by OPAL, pbm is a library of C functions providing facilities for declaring and manipulating data buffers. This is used as the basis of the test beam data acquisition. For a full description of pbm see /usr/dac/doc/pbm010.mem.

- **sk**

  Skeleton suite of data acquisition programs which use pbm, intended primarily as example programs.

- **texlib**

  A set of utility programs for source code management, based around archives (texlibs) of source code. This package is required only to extract the source code for pbm and sk, both of which use texlib libraries for code maintanence.

5.2 CERN Libraries

The CERN library FORTRAN package has been converted to C using f2c and installed. The conversion from FORTRAN to C produces C functions with slightly modified names (an underscore is added to the function name) and calling sequences. Jacket functions have been provided for the most common names and these correspond to the standard forms. See Paul Dauncey for details of these jacket routines.

The libraries can be found in the standard location /cern/pro/lib. However, because of the f2c used to produce the libraries, the source is not in the standard location /cern/pro/src/car as patchy cards files, but can be found in /cern/pro/src/for/xxxxxx, where xxxxxx is the package name (hbook, zebra, hplot, higzX11).

6 DAQ Software

The locally written software specific to the data acquisition can be found in /usr/daq/daq. Seperate subdirectories contain the source code, binaries (also found in /usr/daq/bin), include files and utility scripts. The basic structure of the data acquisition suite is a collection of small programs, each performing a well defined task within the data acquisition and monitoring process. These separate programs communicate via shared memory segments. Separate memory areas are used for event buffering, daq parameters/constants and control functions. Users communicate with the main DAQ programs by running utilities which change values in the shared memory, principally in the control area, causing the main programs to change their behaviour. By convention the DAQ programs are all given names daq_xxxxxx.

The memory areas used for data buffering are created and managed by calling the appropriate pbm functions and more details of these area can be found in the pbm documentation. The memory area for mbook is created by a master mbook program and accessed by slave mbook programs. It is called /tmp/mbook.shm.

The only other area currently used is daq_control. This shared memory area is "used" by mapping a control_st C structure, as defined in the daq_control.h header file, onto the shared memory module. Accessing the structure members then accesses the shared memory itself. The control_st structure currently contains the following members:

- run_type : The current/last run type.
- run_status : The run status (active, paused etc).
- run_number : The current/last run number.
- run_start_date : The (cpu) date that the current/last run started, as yymmdd.

- run_start_time : Start time of current/last run, as hhmmss.
- run_end_date : The (cpu) date that the last run ended, as yymmdd. Zero during a run.
- run_end_time : End time of last run, as hhmmss. Zero during a run.
- trigger_number : The current/last event number.
- slowc_number : The current/last slow controls record number.
- cal_select : The calibration channel mask (calibration run only).
- major_cycle : The current/last major cycle number (calibration run only).
- minor_cycle : The current/last minor cycle number (calibration run only).
- write_enable : Flag set to turn on data output to disk or tape.
- debug_print : Flag set to turn on debug printing to emu.
- max_evtsize : Maximum event buffer size allowed.
- beam_momentum : Beam momentum in GeV, signed by the particle charge.
- stage_x_nominal : Requested horizontal stage position in cm.
- stage_y_nominal : Requested vertical stage position in cm.
- stage_x_actual : Measured horizontal stage position in cm.
- stage_y_actual : Measured vertical stage position in cm.
- stage_angle : Stage rotation angle in mrads.
- xtal_stagger : Crystal stagger in degrees.
- material_flag : Material flag (to be defined).

6.1 Essential DAQ Programs

Several daq_xxxxx programs must have been run, or be running continually, to enable the DAQ to function. In normal use, these are started using daq_menu.

- **daq_init**

  Open pbm configuration file, /usr/daq/config/daq.config, and initialise the pbm buffers. Create and initialise the daq_control area. The program accepts one parameter, the name of an alternative configuration file.

  This program must be run once to completion during initialisation. It will not appear in a process list, e.g. using `ps -a`, during normal running.

- **daq_readout**

  Acknowledge hardware trigger and read out front end electronics. All CAMAC commands are issued from this program. It is a "producer" in pbm terminology; it reserves a buffer and fills it with the required event data. This buffer will remain untill all the daq programs in the pbm stream have processed this event. The event data are actually written by mapping the event structure defined in the header file daq_event.h onto the data buffer. daq_event.h thus specifies the event format.

  This program must be started after daq_init and must run continually as a background process.

- **daq_monitor**

  Consumer program which monitors data quality and fills required monitoring histograms. Uses mbook rather than hbook to allow user monitoring of the histograms as they are filled. The program is a 100% consumer as far as pbm is concerned, but only fills data histograms every 50th event, simply to avoid slowing down the DAQ.

  The program must be started after daq_init and must run continually in the background.

- **daq_todisk** and/or **daq_totape**

  Writes events to disk and to tape. Either or both can be used. The events are blocked into 512-byte blocks to ensure good output speed.

By default when writing to disk, the program writes files named RUN.nnnnnn, where n is the current run number, into the /data directory. This may be changed by setting the environment variable DAQOUTPUT. If this variable is set then the data will be written to $DAQOUTPUT/RUN.nnnnnn.

At least one of these programs must be started after daq_init and must run continually in the background.

- **daq_check**

  Monitors the above processes to check they have not stopped and checks the /data disk area for space. Strictly speaking, this program is not necessary for the DAQ to function, but it is recommended that it is always started after the other processes listed above.

6.2 Utility DAQ Programs

The other DAQ programs are not essential for running, but are used to control and monitor the run. Again, they are normally run using daq_menu.

- **daq_runstart**

  Start a new run. It takes one argument, the run type. It simply sets the flag run_status to RUNSTARTING in the daq_control memory area. This tells daq_readout to start a new run. daq_runstart will fail if a run is already started. This program runs to completion in the forground when invoked.

- **daq_runstop**

  Signal the end of a run. Sets the flag run_status to RUNENDING in the daq_control memory area which tells daq_readout to send an end of run (sweeper) record after the next data event and stop the run. This program runs to completion in the foreground when invoked.

- **daq_dump**

  Dump an event to the screen. Simply attaches to the pbm buffers and dumps the next event to the screen. The dump is given both as a straight hex dump of the buffer and as a formatted dump according to the event format definition contained in daq_event.h. This program runs to completion in the foreground when invoked.

- **daq_buffmon**

  Displays data flow through the daq system. For each program currently identified to pbm (by the configuration file), a column is printed showing the number of events and total data processed by that program. This display is updated automatically until killed (Ctrl–C). The program accepts one parameter, the update interval, in seconds. The default update interval is 5 seconds. The program runs continually in the foreground until explicitly killed.

- **daq_status** and **daq_stati**

  Show current status of the daq_control area. This is simply a formatted dump of the daq_control structure. daq_status runs to completion in the foreground when invoked, while daq_stati updates automatically every 5 seconds until killed (Ctrl–C).

- **daq_toggledebug**

  Toggles the flag debug_print on and off to enable and disable debug printout in daq_readout. The debug messages are sent to the emu screen. This causes a lot of output and so should be used with caution. This program runs to completion in the foreground when invoked.

- **daq_togglewrite**

  Toggles the flag write_enable on and off to enable and disable writing the data to disk or tape. daq_togglewrite will not change the write status while a run is in progress. This program runs to completion in the foreground when invoked.

- **daq_togglepause**

  Toggles the flag run_status between RUNPAUSED and RUNACTIVE to pause and restart a run. daq_togglepause will not change the run status if a run is not in progress. This program runs to completion in the foreground when invoked.

- **daq_tidy**

  Cleans up all DAQ processes and deleted all the shared memory files. It should be run after a crash or cold start to get the system back to a well defined state. This program runs to completion in the foreground when invoked.

- **daq_stage**

  Prompts for new coordinates to move the translation stage. daq_stage will not change the stage position while a run is in progress. This program runs to completion in the foreground when invoked.

- **daq_reademu**

  Prints the emu log file in a formatted style. It takes the input on stdin, so the usual method of using it is

  ```
  tail /tmp/emulog | daq_reademu
  ```

  This program runs to completion in the foreground when invoked.

7 Trigger Logic

The trigger logic is shown in figures 1 and 2 and is implemented in NIM.

7.1 Run Type Trigger Selection

Different run types have different triggers. The triggers are enabled and disabled by a CAMAC output register (OR) at the start of the run.

Refering to figure 1, OR channel 1 (OR 1) enables the beam scintillators, OR 2 the cosmic scintillators and OR 0 the 1kHz clock pulses. These are set on for beam runs, cosmic runs and clock runs respectively. In addition, OR 15 is turned on and off to generate a single pulse for each trigger in source runs, calibration runs and monitor runs.

These signals give a pretrigger, labelled "T" in figure 1. In addition, the signals form a preveto, labelled "V", to prevent triggers in events with more than one particle. Finally, allowance has been made for a veto scintillator, which would kill a trigger if it sees a signal in time with the trigger itself. This is labelled "K" but has no input presently.

Each trigger input is input to the TDC channels 0 to 5 to allow jitter in the timing to be checked and deduce which type of trigger caused the event. The final two channels of the TDC, channels 6 and 7, measure the 50MHz beam clock relative to the trigger to allow particle identification to be performed. Signal "G" here is the trigger created in the logic of figure 2. Two measurements are necessary as finite signal widths mean the input

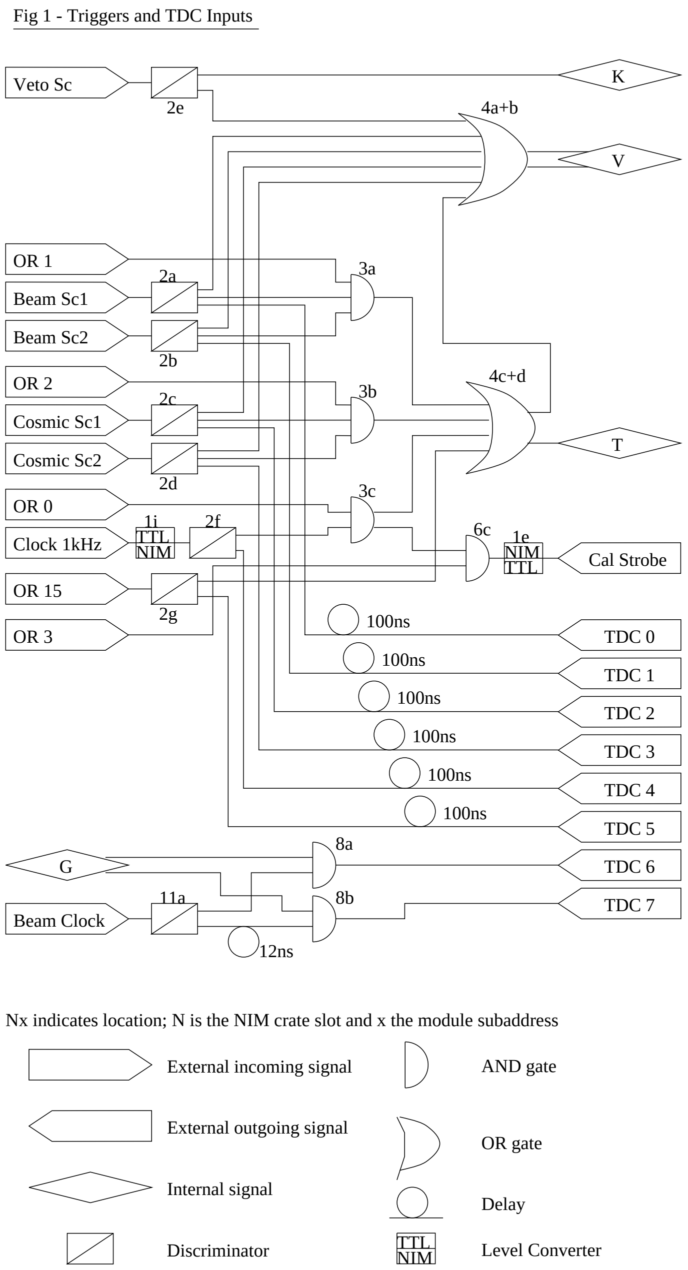

Figure 1: Triggers and TDC Inputs

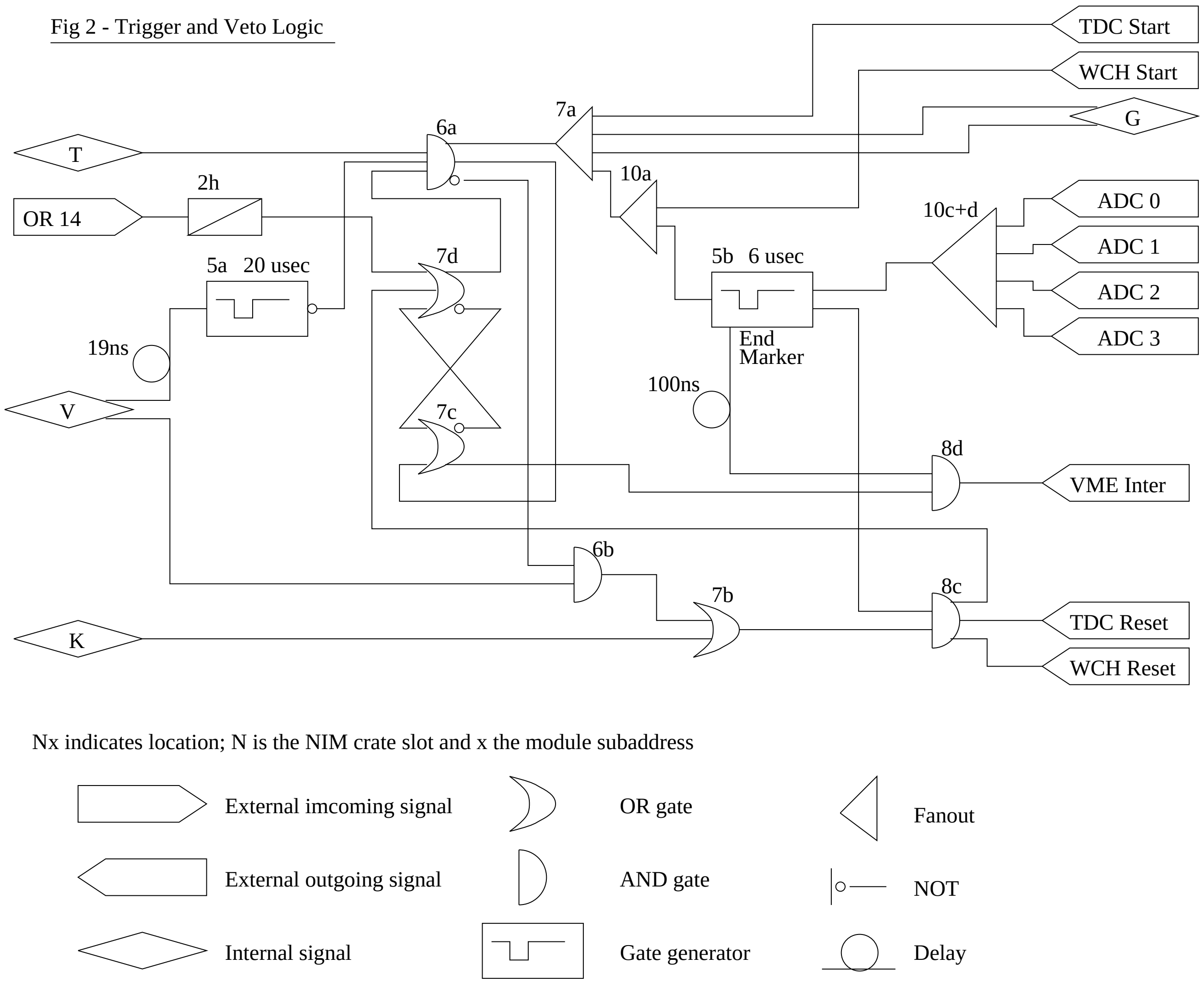

Figure 2: Trigger and Veto Logic

signal often overlaps the TDC start signal. Two inputs, one delayed by 9ns relative to the other, mean at least one of the two measurements should always be valid.

7.2 Trigger and Veto Logic

The trigger is required not only to generate gates for the ADCs and TDC but also to veto events where two particles have hit the crystals within a short time, which would lead to excess energy being measured.

Refering to figure 2, every preveto “V” signal starts a (regeneratable) 20 μsec gate. Unless this gate is off, a pretrigger “T” cannot give a real trigger. This therefore prevents a trigger from occuring when there is still some tail of signals from a previous particle. The 20 μsec time is the nominal time for a signal to have died away to a negligable size.

If the gate is off, a pretrigger will cause an ADC and TDC start and sets a latch which prevents further pretriggers causing starts. The latch remains on until reset in one of two ways described below.

If a pretrigger is not vetoed, it starts a 6 μsec gate for the ADCs, which then sample on the trailing edge; 6 μsec being the nominal peaking time for the preamps. Assuming no other particles arrive in this 6 μsec time, then the end marker of the 6 μsec gate causes an interupt and the event is read out, after which a signal is generated on OR 14 to reset the latch. However, if another particle arrives in this time and generates a preveto “V” or “K”, then the TDC and the latch are reset, no interupt is generated, and the system is ready for another trigger when the 20 μsec gate has finished. This therefore prevents pileup within the peaking time of the signal.

Note, the AC-coupling of the preamplifier means that undershoot from a previous energy deposit is possible; this has a timescale of hundreds of μsecs, which cannot be vetoed as above without giving far too large a deadtime.

8 Data Format

8.1 Run Types

There are seven types of run. These are:

1. Beam run : the normal run type for data-taking.

2. Cosmic run : similar but uses a different set of trigger scintillators.

3. Calibration run : scans the DAC voltage applied to the preamps. The run consists of "major cycles" (one for each DAC setting) and "minor cycles" (each event for a particular DAC setting). Currently, there are 44 major cycles, each of which has 50 minor cycles, although earlier runs used 20 major cycles of 100 minor cycles each. The calibration records (see below) are written out at the start of each major cycle, before any data events.

4. Source run : uses a computer-generated trigger, which simply fires as soon as the previous event has been read out.

5. Clock run : uses the 1kHz clock as the trigger.

6. Monitor run : similar to a source run, but the readout sleeps for 10 seconds between events. This it intended to be used when no run is in progress but power is on and the temperatures still need to be read out.

7. Dummy run : uses dummy versions of all CAMAC routines. This is for testing the DAQ software without any hardware.

Each record written is composed of two parts, the header and the body. All records have headers of the same format, while different types of record have different body formats. There are six types of record; start of run (SOR), data (EVT), pedestal (PED), calibration (CAL), slow controls (SLC) and end of run (EOR).

A start of run record is written out as the first record of every run. Data events are the records containing the ADC, TDC and wire chamber data and are written for all runs. Pedestal events are written out during beam and calibration runs only; 100 pedestal events are written immediately after the start of run record and then approximately every three minutes during the run. Calibration records are only written in calibration runs; they are written at the start of every major cycle and indicate that the DAC has changed value for the following events. Slow controls records are written every 10 seconds in all runs and finally, an end of run record is written out for all runs.

All data are either 2- or 4-byte integer words. This requires floating point variables to be converted to integer before being written. Several conversion factors (which have large values to prevent rounding errors) are used and these are stored in the start of run record (see below).

8.2 Header

The header consists of eight 4-byte integer words;

- `NDATA` the number of bytes in the whole record
- `NVER` the data format version number
- `NRUN` the run number
- `NEVT` the event number
- `IDATE` the event date in yymmdd format
- `ITIME` the event time in hhmmss format
- `IRTYPE` the run type
- `IETYPE` the record type

The names listed here correspond to the variables found in the btana common `/BTCRUN/` when analysing the data offline. Note, the event number for all records except data and pedestal events is set to zero.

The values for run and record types can be found in daq_event.h.

8.3 Start of Run Record Body

Following the header, a start of run record contains the following (all 4-byte integers):

- `GEVINT` conversion factor from GeV to integers
- `CMSINT` conversion factor from centimetres to integers
- `USCINT` conversion factor from μseconds to integers
- `RADINT` conversion factor from radians to integers
- `DEGINT` conversion factor from degrees to integers

- PBEAM beam momentum signed by the particle charge, in units of GEVINT
- STAGEX stage horizontal position, in units of CMSINT
- STAGEY stage vertical position, in units of CMSINT
- STAGEA stage rotation angle, in units of RADINT
- STAGGD crystal stagger angle, in units of DEGINT
- IMATER material configuration flag (to be defined)

The names listed here correspond to the (floating point) variables found in the btana common /BTCGPR/ when analysing the data offline.

The total length of the start of run records is therefore 76 bytes.

8.4 Data and Pedestal Event Body

The bodies for data and pedestal events are identical; these record types differ only through the trigger used; data events are run with a beam trigger and pedestal with a beam veto.

Following the header, a data or pedestal event contains the following (all 2-byte integers):

- ADC data; four words per crystal from the two gains of each of the two preamps. This is 100 words total for 25 crystals.
- TDC data; 8 words for the 8 TDC channels.
- Wire chamber data; the first word gives the number of wires read out and the following data list these wire numbers.

The total length of the data and pedestal records is $250 + 2N_{wires}$ bytes, rounded up to a multiple of 4 bytes for even values of N_{wires}.

8.5 Calibration Record Body

Following the header, a calibration record contains the following (all 2-byte integers):

- Major cycle number (starting from 1, not 0).

- DAC setting for the following data records, in DAC units.

The total length of the calibration records is therefore 36 bytes.

8.6 Slow Controls Record Body

Following the header, a slow controls record contains the following:

- Stage positions, x,y, as two 4-byte integers. Since the stage was removed for all runs, these numbers are always zero.
- Temperature data; the first word gives the number of temperature measurements and the following data give the digitised values. Between 8 and 16 temperature measurements are made. These are all 2-byte integers.

The total length of the slow controls records is therefore $42 + 2N_{temps}$ bytes, rounded up to a multiple of 4 bytes.

8.7 End of Run Body

These is no body for an end of run record. The total length of the end of run records is therefore 32 bytes.

9 Summary of Experience

Overall, the system ran successfully and almost continually for three weeks at around 80 Hz, writing to exabyte at 40 kbytes/sec. This was achieved without any attempts at tuning priorities between the processes and included some heavy interactive use by people on shift, particularly for examining plots.

Our experience of using LynxOS was generally very positive. For a Unix user, it is trivial to use and seems effectively identical for most uses.

Porting standard applications was straightforward. The pbm and emux packages were easy, while the f2c utility made the installation of CERNLIB from FORTRAN source code tedious but not complex. In fact, one of the biggest difficulties was getting a consistent set of source code for all the packages required. Providing the standard calling sequences to sit on top of the libraries was almost as time consuming as the initial porting work. One feature which we failed to achieve was shared memory within HBOOK, which would have allowed direct filling and viewing of HBOOK histograms in separate routines. Because this was not done, the purpose-built MBOOK package was written to perform the task.

The configuration of the MVME167 was straightforward as were the VME access and device drivers. The biggest problems occured because the version used, V2.1, was not completely POSIX compatible, so native LynxOS, rather than POSIX compliant, routines had to be used in places.

The reboot sequence defaulted to running fscheck on the whole 2GB disk, which proved very time consuming. This could be disabled, but as it turned out, the system was mainly stable during running and so this was not a great issue.

Full networking was not achieved, although IP was fully supported. The problem seemed to be with gateways from the LAN, not the system itself. Logins and ftp connections from within RAL or PSI (depending on where the system was at the time) worked well.