Contents

1 Preface

The data acquisition system TDAS which is the subject of this manual was mainly developed during the second half of 1991 by members of the GSI collaborations of the CAVE B experients 4π, ALADIN, LAND. The main reason for building it was to overcome the time until the GSI general data acquisition system would be available. So the name TDAS was chosen to be **T**emporarily **D**ata **A**cquisition **S**ystem. However - as things typically evolve - the general GSI data acquisition system took more time to get into an operational state and therefore TDAS was more and more worked at. So the Temporarily data acquisition system became a **T**enable data acquisition system. It is still in use at the ALADIN and LAND experiments. The 4π experiment has a processor farm for online data reduction. Such a setup requires the data acquisition to support event pipelining. TDAS does not support event pipelining. The 4π experimenters therefore switched their data acquisition system to be a hybrid of the main GSI data acquisition system which supports event pipelining and TDAS which supports a variety of features described later in this manual they would not not miss. Especially during the setup phase of the experiments it sometimes happened that the data acquisition system crashed pretty often. During that time we called it **T**ouchy **D**ata **A**cquisition **S**ystem. However during the production runs where TDAS ran quite stable **T**enable **D**ata **A**cquisition **S**ystem was a more suitable name. So the **T** in the name of our data acquisition system can be considered as a wild card for the appropriate situation.

This manual comprises of three main parts. In the first part the variety of the hardware scenarios which are supported by the TDAS concept are described. The second part – the user manual itself – describes how to setup and operate TDAS and it's utilities. The third part is a technical manual which describes the implementation details of the whole data acquisition system. In the appendix you will find a little lexicon of the most important terms that will be used throughout this manual. It is only intended to give the reader a rough idea whats behind the appropriate item. For detailed information please refer to the there cited references. However I will put there some useful hints you wont find in the documentation.

1.1 general features of *TDAS*

- fully stand alone
- runs on single os9 based 68k processor
- local taping via Exabyte 8200/8500
- online event monitoring via TCP/IP or Cern RPC supported for any workstation platform
- Camac, Fastbus and VME readout supported
- supports GSI standard hardware
- support of host based experiment control via databases
- flexibel - subsystems can be easily switched from stand alone mode to combined mode with main TDAS
- intelligent subsystems outside the scope of TDAS (for example dectectors with complete readout from collaborators) can be easily connected via dual-ported-memories using a simple – easy to implement – protocol
- fast – minimum event cycle time (depending on CPU): < 200 µs

2 *TDAS* Hardware

In the following chapters the variety of hardware scenarios supported by the TDAS concept are described. In the next sections sample trigger setups and the basic components for a running TDAS are defined together with the variety of hardware options which may be incorporated in the TDAS concept. Chapter 2.6 describes the setup of the GSI experiment S022, where most of the TDAS features came to operation.

2.1 *TDAS* CAMAC Requirements for Trigger Input the I/O-LAM Box

Any data acquisition system needs some inputs to trigger the readout and define the *readout-type* and some outputs for clearing the hardware after readout and for defining the dead-time. Since almost any setup has at least one CAMAC crate for maintenance we chose CAMAC devices for that purpose. Figure 2.1 shows how the so called *system-crate* (crate number 15) has to be set-up and cabled for TDAS. All signals, which will be discussed in the following, are NIM signals. NOTE: in the following circuit diagrams logically identical signals are connected to each other for simplicity. However since any NIM input is terminated with 50Ω, an output which is connected to more than one input has to be driven with *fan-in/out* units. These driver units are omitted in the circuit diagrams for simplicity.

For handling the trigger signals, there is a CAMAC device avaliable at GSI - the I/O LAM box - which serves all requirements of *TDAS*. It must be mounted in station 1 of the *TDAS system-crate*. The I/O-LAM box has a latched trigger input - the LAM in (**LI**). It can be programmed to interrupt the data acquisition process and therefore serves as a readout trigger input for *TDAS*. The busy flip-flop (**BF**) is another flop-flop of the I/O-LAM box which is set synchronously to the LAM flip-flop (**LF**). The ouput of the busy flip-flop defines the dead time of *TDAS*. These two signals (**LI, BF**) are the minimum trigger signals required by *TDAS*. All additional signals discussed in the following are needed for higher flexibility; however small test setups may omit them.

10µs before releasing the dead time (**BF**) *TDAS* supplies an about 100ns pulse (**P2**) for clearing the hardware. In case of a Fastbus setup with two AEBs in one crate this clear signal is required for all Fastbus modules since the AEBs can only clear all - and not just their associated - modules with a Fastbus broad-cast call. Therefore all Fastbus modules (if any) have to be cleared with the **P2** signal. If the trigger uses fast-clears then all modules are already connected to a general clear signal, thus **P2** may simply be ORed into that signal chain. However independent of how the clear is cabled it is a good style to read all CAMAC devices (except the spill scaler) with F2 (read and clear). There are some modules around which require a *F2* clear after readout (for example *AD811*).

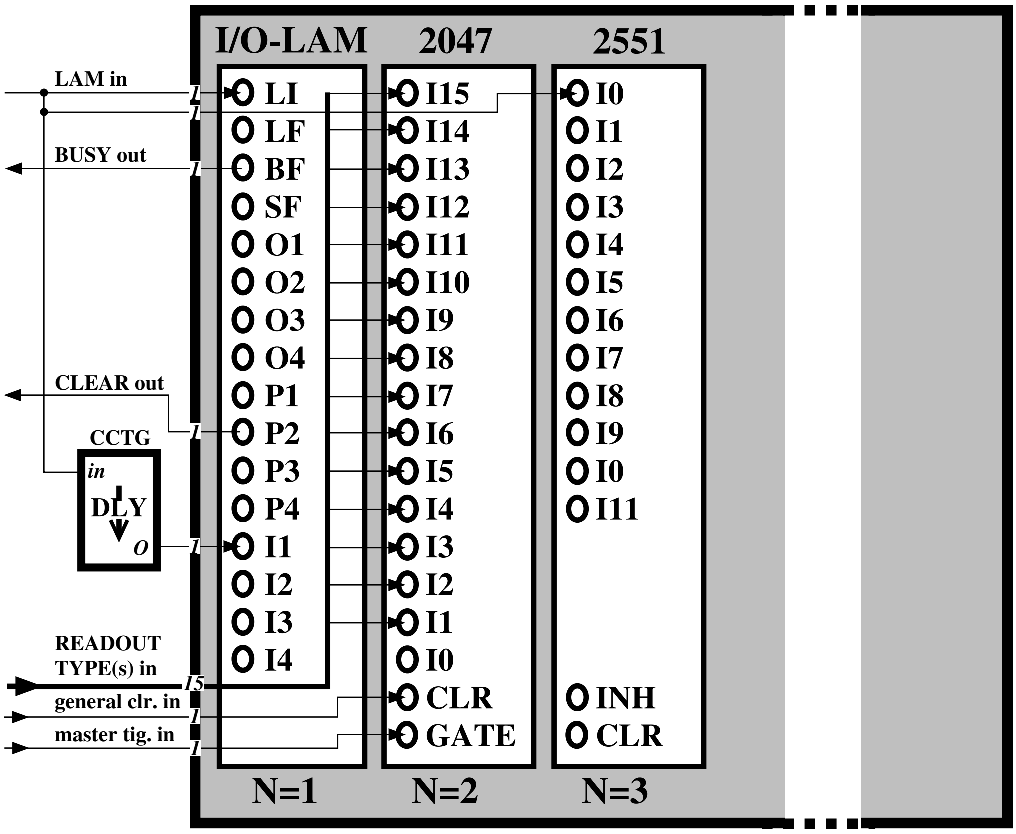

Figure 2.1: CAMAC requirements for TDAS
The first three CAMAC stations of the so called system-crate (crate number 15) have to be setup in the scheme sketched above. However by setting software flags (chapter 4) the modules in stations 2 and 3 may be omitted with the loss of some functionality. As an example of how to set up these input- and output signals refer to chapter 2.2 and 2.3.

The CAMAC modules need some conversion time before they can be read out. This conversion time can be guaranteed by a delay loop at the start of the CAMAC-readout. However this means wasting life time. A better approach is the use of the CAMAC conversion time gate (**CCTG**) as sketched in figure 2.1. It works as follows: As long as the input **I1** of the I/O-LAM box is set the CAMAC readout is inhibited. Therefore the CAMAC conversion time can be guaranteed by connecting the **I1** input to a one-shot-timer (for example MOD 2255 dual timer), which has the appropriate conversion time (~150µs) set up and is triggered with the *TDAS* trigger input (**LI**). In this scenario the CPU response time to the **LI** and the time for starting all the subprocessors runs parallel to the CAMAC conversion time.

Any setup has a series of different trigger types, which require different handling by the data acquisition system. For example there is typically compressed readout and uncompressed readout with a scaled down rate for monitoring baselines and pedestals. In addition there are maintenance triggers like pulser, cosmic and the *spill-off* triggers. The *spill-off* trigger is especially important for a couple of reasons: first of all it starts buffer state transitions inside *TDAS* so the next spill starts with an empty buffer. During the *spill-off* event the spill scalers are read and cleared. In addition the scaler data structures are updated for the scaler display program *TDASscaldisp* (refer to chapter 4).

In order to tell *TDAS* what kind of readout - the *readout-type* - is requested a pattern box (for example type 2047) is required in station 2 of the *system-crate*. It should be gated with the *master-trigger* and cleared with the *general-clear* signal. This pattern unit is read out every event. The bit pattern is then priority encoded in software. The *readout-type* therefore corresponds to the bit number of the highest bit that was set in the pattern unit and ranges from 1 to 15. The input signals of the pattern unit can be directly connected to the trigger box (refer to chapter 2.2), which has an appropriate NIM *after-dead-time* output for every trigger input. It is therefore possible to set an arbitrary bit at the pattern box, and therefore an arbitrary *readout-type*, if a particular trigger is accepted. *TDAS* will pass this *readout-type* to all subprocessors and to the CAMAC readout code. The 15 available *readout-types* may be assigned arbitrarily, except *readout-type* 15, which is assigned to the *spill-off* event. If however for simple test setups no different *readout-types* are required, the *readout-type* may be fixed to a certain value using the *fixed_readout_type* option in the self documenting configuration file *TDASpar.cnf* (refer to chapter 4). In such a setup *TDAS* will not read station 2 - the pattern unit is not needed.

The scaler in station 3 of the *system-crate* is used by *TDAS* to check the trigger logic. TDAS expects the readout trigger (**LI**) to be input to channel 0 of that scaler (for example type 2551). The other inputs are undefined and may be used as spill scalers. The local event counter is compared to channel 0 of that scaler every event. If a mismatch is found an error counter is incremented in the global TDAS data structures, which may be monitORed with *TDASmonitor* (refer to chapter 4). If this check is not desired it may be switched off with the *no_scaler_check* option in the configuration file *TDASpar.cnf.*

The following outputs of the I/O-LAM box signal the current *TDAS* status (NOTE: P1 and O1 are reserved for specific applications):

signal	**description**
LF	This output is set by the LAM in readout trigger (LI) and cleared by *TDAS* in the interrupt handler. The duration of this signal shows the current interrupt latency of the data acquisition CPU. It depends on CPU type and clock and is of the order of 30-100µs.
BF	This output is set as the LF output by the LI input and cleared by *TDAS* after readout. The BF output defines the common dead-time.
O2	This output is active during a *TDAS* buffer state transition of *TDASacquire*. If the data acquisition rate is faster than the taping rate, a buffer state transition may take several seconds!
O3	This output is active during the camac readout only.
O4	This output is active if *TDASacquire* is in the data fetch state of a certain subprocessor. If for example three subprocessors are connected to the *TDAS event-builder*, the O4 will become active three times per event. The duration of the O4 is a measure of the size and data transfer rate of the dual ported memory of the slave system.

2.2 A Simple Trigger

Figure 2.2 shows a simple trigger setup for *TDAS*. The central component is a GSI trigger box TB8000. We strongly recommend to use that module since it contains all the features needed here. If there are more than eight channels, several TB8000 units may be used by oring their trigger output and feeding the *common-deadtime* to all inhibit inputs. Each of the eight channels per TB8000 can be enabled or disabled and scaled down. Every input has a corresponding after dead time lemo output (**ADT**), which can be used to build the *readout-type* at the pattern unit in station 2 of the *system-crate*. The TB8000 has also individual ECL outputs for the trigger inputs before dead time (**BDT**), after dead time (**ADT**) and after scale down (**ASD**). These outputs are well suited to be directly connected to the spill scaler (for example use LRS 4434 or similar devices). There are two identical ECL outputs of the after dead time trigger. The second one is typically connected to a pattern register (for example LRS 4448) to determine the trigger pattern of the specific event.

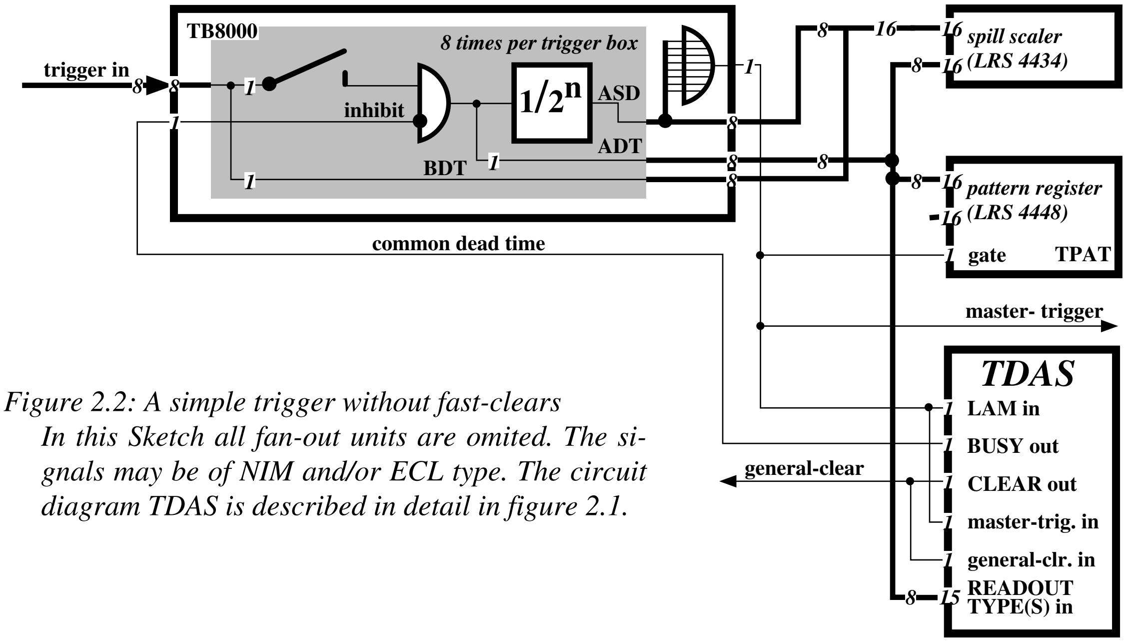

Figure 2.2: A simple trigger without fast-clears
In this Sketch all fan-out units are omited. The signals may be of NIM and/or ECL type. The circuit diagram TDAS is described in detail in figure 2.1.

The ORed trigger output of the trigger box forms the so called *master-trigger*, which is the source for all starts/stops, gates, etc. of the various modules. It is typically driven with a 16-fold fan-out unit. The **BF** output, the *common-dead-time*, is input to the inhibit of the trigger box. *TDAS* produces a clear signal of about 10µs before releasing the dead time. This clear should be connected to all modules (excluding scalers) - the *general-clear* signal. Since the trigger described here does not incorporate *fast-clears,* the clear of the modules may be done in software thereby avoiding the *general-clear* signal tree.

2.3 A Trigger with Fast-Clear

Figure 2.3 shows a sample trigger setup with an appropriate *fast-clear* logic. A *fast-clear* kind of trigger is required any time when a detector in the trigger decision logic that may produce a veto is too slow to produce that signal before the analog delays (typically a couple of 100ns) expire. The typical work around is to trigger all modules with some pretrigger, the *master-trigger* and to wait a certain time - the fast clear gate (**FCG**). If during that time interval a veto occurs, all modules are cleared again with the *general-clear* and the data acquisition system is not triggered. Of course the common dead time has to be set during the time interval between the *master-trigger* and some settling time after the *general-clear*.

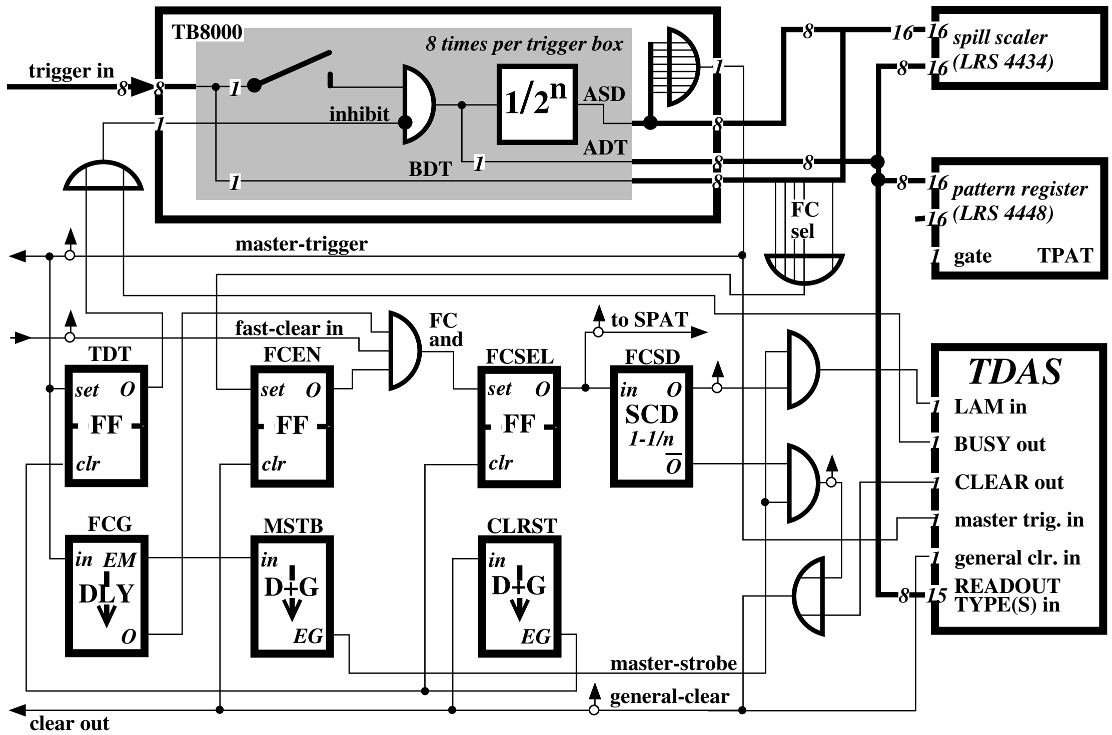

Figure 2.3: A sample fast-clear trigger logic
In this Sketch all fan-out units are omited. The signals may be of NIM and/or ECL type. The circuit diagram TDAS is described in detail in figure 2.1. For details please refer to the text.

The basic components like the trigger box, the spill scaler and pattern register are similar to those of figure 2.2. However the *master-trigger* and *general-clear* are different. The sketched trigger logic works as follows: If there is a valid trigger condition during the life time, the trigger box TB8000 will fire a 200ns pulse at its' common trigger output - the so called *master-trigger*, which then triggers the whole readout electronics. Synchronously to the *master-trigger* the trigger dead time flip-flop (**TDT**) is set. Such a flip-flop may be built by using a dual timer MOD 2255 with the gate length set to infinity. The output of the **TDT** flip-flop is ORed with the *TDAS* dead time (**BF**). This signal - the common dead time - is fed to the inhibit input of the trigger box TB8000. The *master-trigger* also starts a gate generator. This gate (typical length about 10µs) forms the *fast-clear* gate (**FCG**). Only during that gate are *fast-clear* signals accepted. The *fast-clear* veto is not reasonable for all kind of triggers. For example pulser and baseline triggers would be vetoed every time. Therefore there is a *fast-clear* selection signal, which is derived from a logical or (**FC sel**, may be done with a NIM fan-in/fan-out unit) of the triggers for which a *fast-clear* veto is reasonable. This signal has to be latched (**FCEN**) during the *fast-clear* gate to be valid. If a *fast-clear* signal is received during the *fast-clear* gate and the *fast-clear* enable flip-flop is set (**FC and**), then the valid *fast-clear* is latched (**FCSEL**). There is usually another pattern register with a gate (about 10-20µs) depending on the *fast-clear* gate length, which is similar to the trigger pattern register (**TPAT**). This register (**SPAT**) is used for maintenance trigger information of the event. For instance **SPAT** determins if this is a scaled down *fast-clear* or if there was another beam particle during the conversion of the detectors (pile up). Therefore the **FCSEL** is fed to the **SPAT.** Since it is allways good practise to monitor the rejected events the valid *fast-clears* are scaled down in the next stage of that signal chain (**FCSD**). The *fast-clear* is inhibited every nth (~200) trigger to have a sample of the events, that are typically vetoed. Parallel to the end of the *fast-clear* gate another delay and gate generator (**MSTB**) is triggered (use for example DG8000). This may be accomplished with the *end-marker* of the *fast-clear* gate generator (**FCG**). The specified delay (typically 200ns) corresponds to the signal propagation time of the *fast-clear* input signal through the trigger logic to the output of the scale down unit **FCSD.** After that propagation time the **MSTB** delay and gate generator fires a gate (use 200ns), which forms the master-strobe. Depending on the outputs of the scale down unit **FCSD** either a *general-clear* or a *TDAS* readout trigger is produced. If a readout trigger is produced the *TDAS* computers will process the event and before clearing the *TDAS* dead-time (**BF**), which is ORed to the trigger dead time (**TDT**), a clear pulse will be fired, which must be ORed with the *general-clear*. All modules need some settling time to process a hardware clear. This is accomplished by starting another delay and gate generator (**CLRST**) with the *general-clear*. We typically use 5 - 10µs, which is rather long. After that time expires the trigger dead time flip flop (**TDT**) is reset with the gate pulse of the **CLRST** gate generator. The common dead time finishes when the clear settling time (**CLRST**) and the *TDAS* clear settling time (10µs) have both expired. Since the described trigger setup is complicated enough to produce a lot of problems, it is a good practise to use spill scalers to monitor some of its' relevant signals, like the ones maked with in figure 2.3.

2.4 Special *Spill-Off* Trigger Handling

The *spill-off* trigger is relatively important for several reasons. First, *TDAS* starts several actions that are probably time consuming like buffer state transitions. This is therefore intentionally done in the *spill-off* phase. *TDAS* tries to start the next spill with empty buffers (see section 4). Second, all spill scalers are read in the spill off event. Therefore some logic must be supplied to ensure, that every *spill-off* event is read out and not inhibited by the dead time. Figure 2.4 shows an appropriate sample setup.

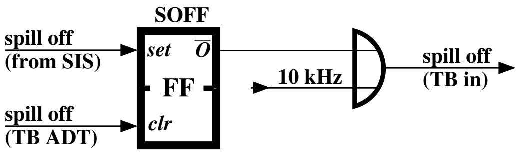

Figure 2.4: A sample trigger input for the spill-off trigger
This logic ensures that the spill-off trigger from the synchrotron is not lost during the dead time.

The *spill-off* event from the SIS synchrotron is latched (**SOFF**). The output of that latch enables a 10kHz oscillator, which is fed to the *spill-off* trigger input of the trigger box (**TB in**). When the *spill-off* trigger is accepted (*spill-off* signal 'after dead time of the trigger box' (**TB ADT**) the *spill-off* flip-flop is reset.

2.5 A Minimal *TDAS* Setup

Figure 2.5 shows a sketch of a minimal data acquisition setup. The *event-builder* is typically a VME CPU. However it may be also the intelligent CAMAC Controller CVC.

If local taping is required, an EXABYTE drive may be connected to the SCSI bus of the *event-builder*. Both (high density, EXB 8500 and low density, EXB 8200) are supported. There is no customization required for the type of the connected tape. Since *TDAS* inquires it from the tape itself. The only restriction is that the tape drive is expected at SCSI ID 5 and LUN 0 (this is the default for the VAXEN, too). The EXABYTE has to be jumpered appropriately.

If the data rate is small, it may be also sent through ethernet using TCP/IP to a host computer to be stored there. All the GSI CAVEB front end computers have no hard discs installed, they have virtual disks, which are served by a host computer - currently a VAX .

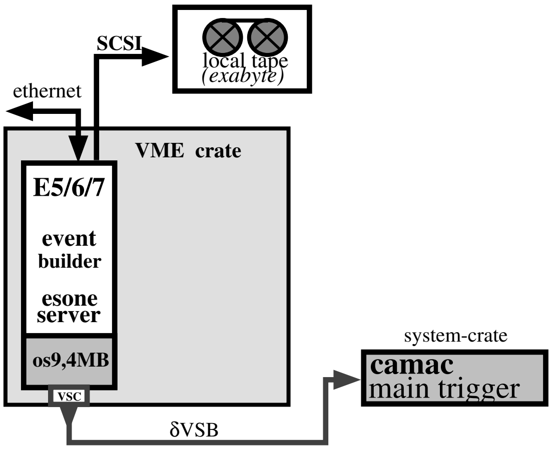

Figure 2.5: Minimal TDAS setup
This sketch shows a minimal TDAS setup with one VME CPU and the CAMAC system-crate.

Since the trigger (see chapters 2.1 - 2.3) input of *TDAS* was chosen to be a CAMAC module, the I/O-LAM box, at least one CAMAC crate must be connected to the *event-builder*. This is done by using the differential VSB bus extention δVSB (see chapter A), which was developed at GSI. First there is a slim card the VSC (see chapter A) which is connected to the VSB port of the *event-builder* at the back side of the VME crate. The other end of the VSC forms the differential VSB bus. For the CAMAC *system-crate* branch the VSC must be jumpered to use VSB window 5 and to be a slave on the VSB side (the *event-builder* is the master). Mount the crate controller CBV in the CAMAC crate and connect it to the VSC with δVSB cable. This is all of the computer hardware setup that is required for *TDAS*. The trigger logic and the CAMAC crate were described in chapters 2.1 - 2.3.

2.6 A Sample Setup Combining most of the *TDAS* Features

The minimal hardware requirements for *TDAS* were described in chapter 2.5. The variety of scenarios that are possible within the *TDAS* concept is sketched in figure 2.6, where almost all TDAS features stated in chapter 1.1 were implemented. The whole setup was comprised of 9 CPUs. The *event-builder* sits in the central VME crate. It has the local EXABYTE tape connected at its' SCSI bus. All front end processors are served by the VAX cluster via ethernet. They are all accessible with telnet.

The *event-builder* has two δVSB branches. This is accomplished by mounting a two slot VSC backplane on the back side of the VME crate between the VSC and the VSB backplane. It consists essentially of two female 96 pin wire-wrap connectors, where all pins are parallel connected. Such devices are available in the Elektronik-Experimente department of GSI. In the two branch VSB setup there are two VSC cards connected to the same VSB bus of a given processor. Therefore the VSB terminators (not the δVSB terminators) must be removed from the second VSC. NOTE: not all VSC cards have their VSB terminators (three SIL resistor arrays) on sockets. To have a different VSB base of the second branch the VSB window of the second VSC must be set to 6. On the E5, E6, E7 the VSB address map is identical like:

VSB window #	VSB base address	CAMAC branch #
5	$ F0000000	0
6	$ F4000000	1

The crate number offsets to the VSB base address like:

<VSB crate address> = *<VSB base address(branch)>* + *<crate number>* · $200000

The δVSB branch 0, to which the TDAS *system-crate* must be connected, connects all CAMAC and Fastbus crates in the counting house to the *event-builder*. Branch number 1 connects everything in the CAVE. It is a good idea to seperate the short branch in the counting house from the long one to the CAVE.

The VME addresses of the various devices in the branch may be set arbitraritly. It is strongly recommended not to use VME standard (A24) addressing. Use VME extended (A32) addressing whenever possible (only some old CES products don't support the A32 addressing mode).

All Fastbus crates in this setup are read out by two AEBs to decrease the data reduction time. Since a Fastbus block mode transfer takes about the same time to process like as the setup, even the readout speed of the modules is doubled with two AEBs.

There are two readout processors in the CAVE - one for each setup: the *MINIBALL* CAMAC hardware and the *MUSIC* flash adc. Their dual-ported memories for the data and control exchange are intentionally mounted in the crate of the *event-builder*. In this setup the data transfer through the long and slow δVSB cable (data rate about 2-4MB/sec) is parallelized and the *event-builder* may then read the data much faster from these dual-ported memories directly through VME (about 15 MB/sec). The other option to connect the readout processors in the CAVE directly to the *event-builder* as VSB slaves (the E6 and E7 have a VSB master/slave interface) works also. It is cheaper since there is no need for an additional VME/VSB memory per front end processor. But it produces more CPU load on the *event-builder* and is much slower since it can read the front end processors only sequentially!

Since the collaborators from *MSU* brought their own CAMAC crate controllers, the front end processor of their setup does not use CA/BVs but instead the *CES* CAMAC branch driver.

The *MUSIC* readout front end processor has a special setup, which is both importend and not obvious. There are two δVSB branches connected to that CPU. Two δVSB branches are not absolutely required here - if there was not a hardware bug in the DL350 flash-adc-system. The DL350 reacts on any δVSB cycle independent if it is selected or not; unfortunately there is no practical fix possible. The only alternative is therefore to give it a separate branch!

The LAND Fastbus crate, which was in the CAVE, was connected as second branch to the *event-builder*. Since the LAND *event-builder* crate is setup near by the general *event-builder* in the counting house, the LAND subsystem was simply connected by moving the δVSB cable from one *event-builder* to the other. However the LAND control branch stayed connected to the LAND *event-builder* during the whole experiment.

The beam *event-builder* had a special state: all sub systems discussed so far are so called synchronous sub systems. This means they must react on any readout command. The beam *event-builder* is a so called asynchronous sub system - it may respond to a command. This is achieved in the following way: The main *event-builder* sends the readout command to the asynchronous sub systems. After finishing the whole readout of all synchronous sub systems the main *event-builder* looks in the communication structurs of the asynchronous sub systems. If there is some data to transfer it will be put into the data stream. If not, it finishes the readout ignoring this sub system(s). This scenario has the advantage to produce only negligible additional dead time due to the data transfer of the asynchronous sub event. The main purpose of the beam TDAS is for focussing and monitoring the beam with typically *Stelzer* counters. For calibration purposes however it was requested to have some of these beam TDAS events in the main data stream. Therefore a coincidence trigger was set up, which produced a special coincidence *readout-type*. If the beam TDAS recognized such a coincidence *readout-type*, it produced the appropriate event in the appropriate communication structure and the main TDAS then copied it into the data stream. The beam TDAS had plenty of time to produce that tiny sub-event since the typical readout time of the main TDAS is of the order of several ms.

During our experiments it turned out that one very important feature of the data acquisition system is the flexibility to switch any given sub sytem to a stand-alone mode. Normally during the setup phase of an experiment many people want to do incompatible things with the setup. The only way to perform these tests in parallel is to give every major sub setup its' own data acquisition system, which can be later easily incorporated into the main data acquisition system. All sub-systems were connected to a so-called *system-crate* (labelled as *main trigger* in figure 2.6) connected to. Therefore any sub-system could run its own TDAS using the appropriate CAMAC *system-crate* for the trigger input. No recabling of the δVSB and therefore no changes in the address mappings were required. For the MINIBALL and MUSIC sub-systems a stand alone TDAS was run on the appropriate front end CPU communicating to the same readout program through the same memory as the main TDAS. For the LAND setup the δVSB cable was moved to the LAND *event-builder*. To diagnose errors appearing during the data taking, when the CAVE was closed and therefore there was no access to the test *system-crates*, another test CPU was mounted in the VME crate of the main *event-builder*. Therefore, any sub-system could be taken out of the main TDAS and put into the test TDAS without any hardware change. Of course the test TDAS was run with the identical triggers as the main TDAS.

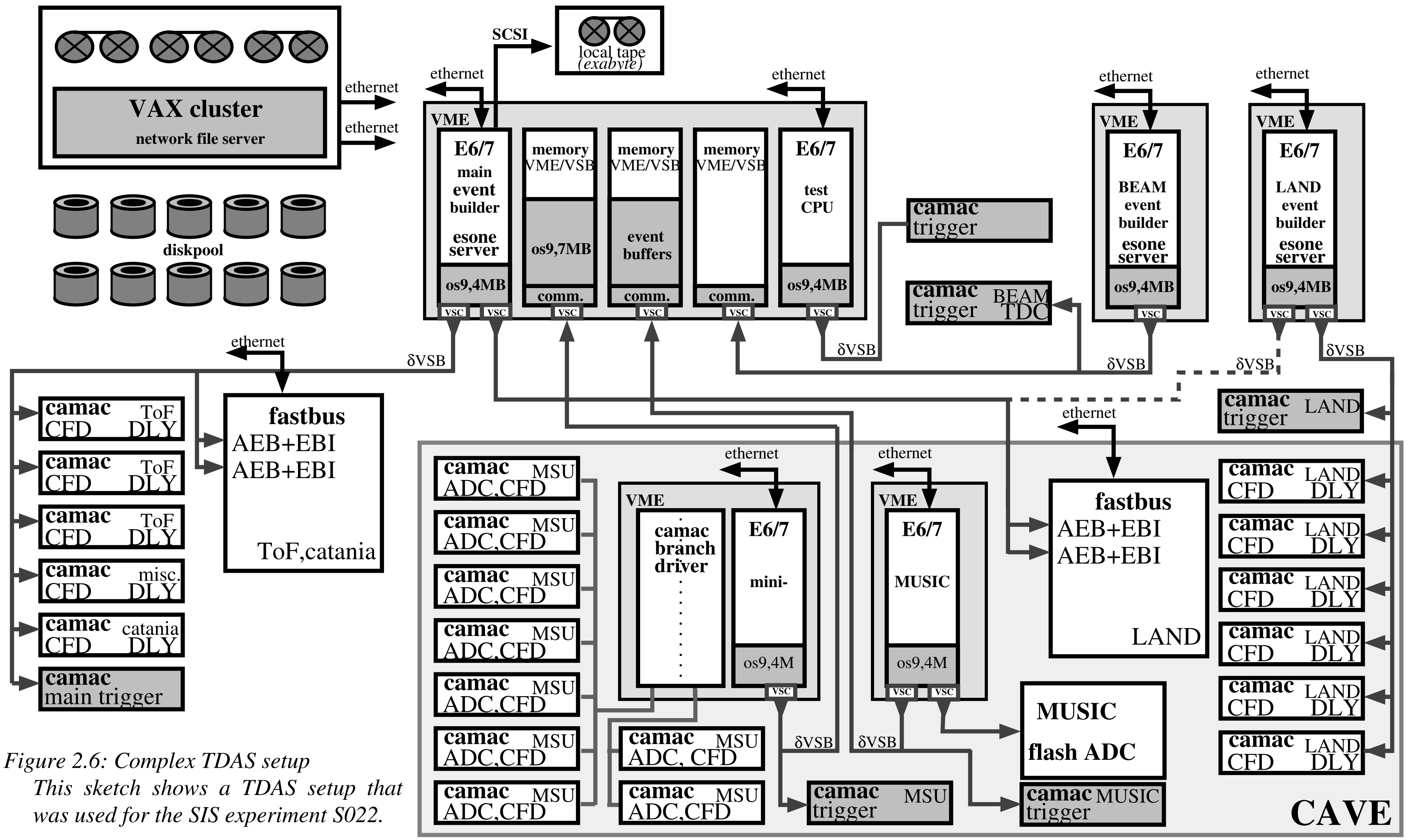

Figure 2.6: Complex TDAS setup

This sketch shows a TDAS setup that was used for the SIS experiment S022.

2.7 Concluding Remarks

The various components of the data acquisition may be combined arbitrarily as shown in figure 2.6. Since a lot of hardware comes into operation a lot of hardware problems typically arises. Not all of them are obvious but all of them are critical! There has not been any setup which worked from the first moment!

The only workaround is testing - testing - testing!

There are a variety of tools which are well suited to test and diagnose the various possible problems. Major problems canarise from the δVSB bus. Typically some wires of the thick cables break in the connectors. This generally leads typically to errors with the *memorytest* or *borertest* programs. How to use these programs and how to interprete the diagnostic outputs is described in chapter 4. It has also happened, that broken CAMAC crate controllers (typically CAVs) spoiled a whole branch. The module needed only to be present to spoil any cycle on the differential VSB bus. One good approach is to have a CAMAC dataway display in the last crate of a branch or to have a memory there, on which one of the previously noted test programs may be run. This checks at least all cables to some extent. However there is another serious problem related to the differential VSB bus. A broken wire of a δVSB cable typically but need not necessarily leads to reproducable *memorytest* or *borertest* errors. It may also spoil a whole branch, since the branch is then only partially terminated. Typical test results are *memorytest* errors in any crate of the branch and a very sensible dependence of the errors on cable length and number and type of the modules connected to the branch (they may partially damp the reflections due to the open line(s)).

Another source of problems arises from the typical incompatibility of different VME modules. Allmost all VME device violates at least some of the design specs! If these violation(s) will cause errors strongly depends on the environment of the setup. The only good confidence is to test. To check the data transfer to and from a VME memory use the *memorytest* program. To check the communication between two CPUs, it is possible to boot the E6 and E7 processors with one megagyte of memory reserved from the use of OS9 . It may be used for mutual memory testing.

A good handle to check the communication between the TDAS *event-builder* and the sub processors is the program *TDAScpudbg* (see chapter 4).

3 *TDAS* User Manual

This part of the manual shall help the user to operate and set the software up to run the *TDAS* data acquisition system. What hardware scenarios are supported within the *TDAS* concept and how to set them up and especially how to build a trigger was already described in the hardware part of this manual (chapter 2). Please remember it is really very important to seriously test all hardware components that are involved into a given setup (refer to *memorytest, borertest, tapetest* etc. in chapter 4). *TDAS* was designed to be as failure tolerant as possible. However broken hardware can cause anyway a lot of trouble, especially since hardware problems often look like software or configuration problems and are therefore highly misleading.

3.1 Setting *TDAS* up for a new Experiment

Since a group does not more than one experiment at a time there was for every group that uses *TDAS* an own private area defined, where the whole group specific *TDAS* setup should be placed. After the experiment that area is typically saved for later inspection. In the current os9 installation, which uses the Cern network disks the group/experiment specific part of the *TDAS* setup is placed at the network disk */n10*, which is private to the given group. The global *TDAS* stuff is placed at */n5*, which is identical (read only) at every network disk mapping. Later os9 installations that use nfs will have a different root path names but all sub-trees discussed here will (hopefully) stay as they are.

There is one very important artifact of the network disk setup, which is so important that it cannot be stated often enough:

do not write to a given network disk from more than one front end computer !!!

never ever !!!

The network disk environment allows any number of front end computers to have the same network disk mapping – to have kind of the same (identical) disks connected. This means if one CPU modifies its network disk all other CPUs will immediately see that modification. **BUT** os9 does not support any kind of distributed lock management. Therefore writing to the same network disk from different front end computers will surely destruct the file system of that network disk. Therefore every user has its own network disk and is responsible for making sure that he uses only one front end computer to write to his account. Only the *TDAS* event-builder should write to the *TDAS* group disk */n10*. Be expecially careful when redirecting log/error files or when running the program *pedestal* parallel to the data acquisition! The default path for all log-files was therefore set to the local ram disk /r0.

At the *TDAS* group disk you will find (beside others) the following sub trees:

/n10/crashlog	at this path the crash dump files are written to by the utility *shutdown*
/n10/startup	at this path there is for every front end processor an individual additional startup file (it's name is equivalent to the internet name of the CPU), that is executed at the end of the generic startup. Subprocessor readout programs should be started here with the appropriate environment.
/n10/tftp_boot	here is the whole tftp boot stuff
/n10/data	path for placing pedestal data files (refer to the programs *pedestal, fastbus_readout*)
/n10/tdas/acquire	at this path the experiment specific Camac readout is placed. There may also be the (usually not needed) readout routines of sub-processors that are not *TDAS* conform.
/n10/tdas/dl350	if the sub-processor readout program *dl350_readout* shall be used with a private Camac subevent readout the appropriate code is placed here.
/n10/tdas/cnf	in this sub-tree there are all configuration files of all *TDAS* related processes
/n10/tdas/cmds	in this sub-tree are all *TDAS* images (they are mostly simple copies of the images in */n5/cmds*, to decouple a running *TDAS* setup from possible software upgrades at the generic disk */n5*)

First of all an up to date Camac readout code should be set up (refer to chapter 3.1.1). The next step is to program the configuration files and environment for the first tests.

If the environment shall be changed (refer to *TDASmenu*, chapter 4) the file */n10/.login* must be modified. If the event-builder has only 4MB of local memory and large buffers are required it is recommended to add a VME memory as external *TDAS* buffer (refer to `-e` option of *TDASstart* and environment variable `TDASstartopt<n>`). The size of one single *TDAS* buffer should be adjusted such that it can take a full spill to prevent buffer state transitions from occurring during the spill phase. The buffer fill ratio of the running system can be investigated (and eventually later readjusted) with *TDASmonitor*. For the first tests you may want to disable the use of any SCSI tape. For that purpose add the -n option to the environment variable TDASarchiveopt.

The next step is to edit the main *TDAS* configuration file */n10/tdas/cnf/TDASpar.cnf.* A sample configuration file with default options is placed at the *DOC* tree of the *TDAS* disk /n5. As first approach all definitions of sub-processor systems (*[a]synch_subcpu*) should be commented out. First of all the SIS experiment number must be defined. This number will be placed into the header of every event. The most important parameter in the *TDASpar* configuration file is the maximum event size. *TDAS* assumes <u>any</u> event to be smaller than this parameter, that of course depends on the number and type of the connected sub-processor systems (it must therefore become adjusted if a new sub-processor system is incorporated into *TDAS*). It cannot prove during readout if this condition is really true. Too large performance losses would be the consequence. However some consistency checks are done. But they may not find all error conditions. If an event is detected to be larger than this maximum size parameter a fatal error is produced shutting the whole data acquisition system down. If there are sporadic *bus-errors* showing up at the end of *TDAS* buffers, the maximum event size is typically set too small. If the event-builder front end cpu is configured not to use the os9 memory protection *ssm,* a wrong maximum event size parameter may also crash the whole computer! If you have a test setup without a *spill-off* trigger but with scalers to read you may set the parameter *spill_off_event_scale_down* to something like 100. If you are testing detectors and often change their operation mode, you should set the parameter *max_buf_age* to one second (1) to sample (refer to *TDAsaccessdc* chapter *4*) always the most up to date data. If however you are writing data to tape with probably low rate you should set this parameter to 1 minute (60) or more. The other parameters in the configuration file may keep their default.

You may now start *TDAS* up using *TDASmenu* (refer to chapter 4). If the startup command terminates with an error refer to the log and error files *tdas.log* and *tdas.err*. You may use the E and L option of the root menu of *TDASmenu.* The most typical error is a bus error. A bus error is a kind of an access violation. The event builder tried to access a physical address, where no device is responding. The physical violation address and it's possible interpretations (branch and crate numbers) are left in the error file *tdas.err*. You may use the utilities *memorytest* and *camc* to debug the hardware if the violation address looks all right. Otherwise check the configuration file and the CAMAC readout.

If the startup command succeeded try to begin a run again using *TDASmenu*. You should start a so called *test run*. In this case *TDAS* will not try to do any taping. The archive process will in that case discard the data with the specified rate (default 500kb/sec). If the command fails check again the log and error files. If the command succeeds start *TDASmonitor* (X option of the *TDASmenu* root menu) to see if there are events coming in. If this is not the case check the trigger logic. The LED displays of the I/O-LAM box (refer to chapter 2.1) are helpful understanding what is wrong with the trigger. If however the event counter is running but also some or all of the error counters, again check the trigger logic. The meaning of the error counters and the corresplnding possible errors in the trigger logic are discussed together with the description of the utility *TDASmonitor* at chapter 4.

3.1.1 Programming CAMAC Readout

There is typically a large variety of different CAMAC modules, that have do be read out. To have a maximum flexibility of the CAMAC readout, it was implemented as C-source code that is linked to the *TDASacquire* process. The CAMAC readout code comprises typically of one source file *camac.c*. A sample documented *camac.c* readout code is placed at */n5/doc/acquire* together with an appropriate *makefile*. The CAMAC readout sources should be placed at the group disk */n10/tdas/acquire*. There must be 5 functions in the source *camac.c*:

- int startup_camac()
  This function is called once at startup of the data acquisition system. Any CAMAC related general setup or initialization (if any) should be done here. If the function returns -1 the global variable *errno* is expected to contain the fatal error code.
- int *beginrun_camac(startpointer)
  This function is called every time a begin run command is executed. To start any run with a well defined state all CAMAC crates should be initialized and cleared. The *OS9 ULIB* function *setup_cbv* is well suited for that purpose. A begin run parameter event can be produced at the address specified by the integer pointer argument *startpointer*. The function returns the updated pointer to the first free integer word after the produced CAMAC parameter subevent or -1 in case of an error. In that case the global variable *errno* is expected to contain the fatal error code.
- int *readout_camac(startpointer)
  The function *readout_camac* is called every event. It shall contain the actual CAMAC readout code that produces the CAMAC subevent at the address specified by the integer pointer argument *startpointer*. The function returns the updated

pointer to the first free integer word after the produced CAMAC subevent. The current readout type (refer to chapter 2.1) is defined in the global variable *MT_pattern*. It can be used to disentangle normal readout triggers from spill/scaler triggers.

- int *endrun_camac(startpointer)
  This function is called every time an end run command is executed. Its main purpose is to produce a CAMAC parameter event for later inspection. Of course the begin- and end run parameter events should be equal. Comparing the begin- and endrun parameter events it is possible to detect setup changes during a run. The call conventions of *endrun_camac* are identical to those of *beginrun_camac*.
- int shutdown_camac()
  This function is called once before shutting down the data acquisition system. Any CAMAC related general cleanup (if any) should be done here. If the function returns -1 the global variable *errno* is expected to contain the fatal error code.

To perform effitiently CAMAC cycles a high performance C-preprocessor macro was implemented that allows both reading and writing CAMAC *BCNAF* cycles like the *camac OS9* utility (refer to chapter 4). For example to write a B=0,C=15,N=5,A=0,F=16 the hexadecimal value 10 the following C statement will be appropriate:

```
camac (0,15,5,0,16) = 0x10;
```

The camac macro is very effitient - the complete above C-statement will translate into a single 68k assembler statement! For dataless CAMAC cycles another shell macro is available. For example to clear the above module with B=0,C=15,N=5,A=0,F=9 the following C statement works:

```
touch (camac (0,15,5,0,9));
```

touch reads the address passed as argument and discards the result. To read a CAMAC data word at B=0,C=15,N=5,A=0,F=0 to store it to the integer pointer *startpointer* and automatically update that pointer the following C statements work:

```
*startpointer++ = camac (0,15,5,0,0);
```

To read modules with several channels loops can be constructed using the C *for* or *while* statements. To achieve high readout performance the argument *startpointer* should be copied to a local register variable in the function *readout_camac*. That register pointer should then be used like in the above example (refer also to the sample file *camac.c* at /n5/doc/acquire).

3.1.2 Reading non *TDAS* conform Sub-Processor Systems

To supply readout code for subprocessors that use a protocol that is not *TDAS* conform is usually not required. For reading non standard sub systems there is an additional C source code *subcpus.c* required that is compiled together with the *camac.c* source. The default *subcpus.c* file contains dummy functions that simply do nothing but immediately return. A in the source documented sample *subcpus.c* source file is placed at */n5/doc/acquire* together with the *camac.c* source. The file *subcpus.c* is structured like *camac.c*. However it contains one additional function:

- int start_subCPUs()
  This function is called once at startup of the data acquisition system. Any general setup or initialization (if any) should be done here. If the function returns -1 the global variable *errno* is expected to contain the fatal error code.
- int *begrun_subCPU(startpointer)
  This function is called every time a begin run command is executed. To start any run with a well defined state all slaves should be initialize and clear their assiciated hardware. A begin run parameter event can be produced at the address specified by the integer pointer argument *startpointer*. The function returns the updated pointer to the first free integer word after the produced subevent or -1 in case of an error. In that case the global variable *errno* is expected to contain the fatal error code.
- void start_subCPUs_readout()
  This function is additional as compared to the CAMAC readout functions. If a readout trigger is recieved all subprocessor systems recieve their readout command first to gain a parallel readout of all slaves and the CAMAC readout. So the *start_subCPUs_readout()* function shall send the readout type as readout command to all specified data acquisition slaves to start them reading the data.
- int *readout_subCPUs(startpointer)
  This function is called every event. It shall contain the actual readout code that fetches the appropriate subevents from the subprocessor systems. The function returns the updated pointer to the first free integer word after the produced subevent. The current readout type (refer to chapter 2.1) is defined in the global variable *MT_pattern*. It can be used to disentangle normal readout triggers from spill/scaler triggers.
- int *endrun_subCPUs(startpointer)
  This function is called every time an end run command is executed. Its main purpose is to produce a parameter event for later inspection. Of course the begin- and

end run parameter events should be equal. Comparing the begin- and endrun parameter events it is possible to detect setup changes during a run. The call conventions of *endrun_subCPUs* are identical to those of *begrun_subCPUs*.

- int stop_subCPUs()
  This function is called once before shutting down the data acquisition system. Any general cleanup (if any) should be done here. If the function returns -1 the global variable *errno* is expected to contain the fatal error code.

3.2 Running *TDAS*

All *TDAS* processes can receive and respond to commands directly. However for easy use there is a control program *TDASmenu* which contains the full *TDAS* functionality. It knows which processes to notify in case of a given command and what results are expected. The major functions of *TDASmenu* are grouped into submenus:

- **S:** TDASstart submenue
  In this submenue are the *startup* and *shutdown* commands.
- **A:** TDASacquire submenue
  In this submenue are the run control commands like *begin/end* and *suspend/resume* run commands and maintenance commands to edit the *TDAS* configuration file TDASpar.cnf
- **M:** TDASarchive mass storage submenu
  In this submenue are the commands that control the taping of the data acquisition system like *initializing* or *unloading* the tape.

Any of the menu commands or options can be executed by typing the specified letter or by moving the reverse selection bar with the cursor keys (a vt100 terminal type is the default) to the appropriate option and hitting the *Return* key.

To control *TDAS* simply log in at the *event-builder* that has the so called *system-crate* connected and type *tm*. To start the data acquisition type *S S*. The first *S* moves to the *TDASstart* submenue. The second *S* executes the startup command. *TDASmonitor* will show the result of the command in a short form. If an error occurs a detailed error report can be found in the *TDAS* error file, which can be read by using the *E* option of the *TDASmenu* root menu. To get the data acquisition into a running state it must be started with the discussed command sequence first. Then a run must be started (type *A B* to begin a run). *TDASmenu* asks if the run that shall be started is a dummy/test run. As first approach a dummy/test run is sufficient. If a taping run shall be started answer this question with no and enter a short run comment. This comment will be put into the data stream for later inspection. If a run could be successfully started there is a monitor utility *TDASmonitor* (refer to chapter 4) to investigate the current state of the data acquisition system. It may be started with the *X* option of the root menu of *TDASmenu.*

4 CAVE B Utilities

In this chapter all OS9 utilities, which were developed within the *TDAS* concept, will be described together with a typical example. All programs are standard OS9 and can be directly executed by typing their name. Every program supports a short online help, which is printed when it is called with the -? option. Unless otherwise stated, the command line options may appear anywhere and in any order in the command line. Command line options may be combined. For example -ad=24 enables the boolean option -a and sets the option -d to 24. Options enclosed in [] are optional.

The sources of all the described utilities are on the *TDAS* disk */n5*.

Several utilities have so many customization parameters that they could not all be specified at the command line. These utilities (for example *TDASacquire, fastbus_readout*) read a configuration file where all options should be defined. These files are self documenting ASCII files. A sample of each of these files can be found at the *doc/cnf* tree of the *TDAS* disk. All configuration files have the same general format:

- empty lines and lines beginning with a ! are comment lines and ignored
- every command line begins with a ASCII parameter identifier (case sensitive) and a list of arguments depending on the argument.
- the arguments are seperated with at least one blank
- blanks and tabs are handled equivalently
- the number of blanks does not matter, leading blanks are ignored
- If the same parameter is specified several times the last definition overwrites all previous ones

borertest

test/verify CAMAC

SYNTAX:

borertest [<opts>] (<branch>, <crate>, <station>) [<opts>]

ENVIRONMENT:

LOGPATH	/r0	generic path for writing log/error files
CAMAC_TYPE	not defined	Type of camac interface to use

FUNCTION:

borertest is used to test the CAMAC accesses. It needs a dataway display (BORER or the compatible GSI DMS1000) to be mounted at the specified <branch>, <crate> and <station>. The dataway display has an internal 24 bit register which can be written to and read from. Like *memorytest borertest* writes and verifies a random pattern to that CAMAC register. Every loop comprises of first writing and then verifying a pattern including the check to have in both cases a correct X and Q and a second fast writing and verifying of another pattern without X/Q check. The start pattern is 0 and is incremented every write/verify cycle by $654321. To abort the infinite test loop and get the total number of errors found type ’x’ at the standard input of *borertest*. There are several types of CAMAC interfaces supported, which may be selected by defining the environment variable CAMAC_TYPE:

not defined	(default) CAMAC crate controller is a (GSI) CBV
CBV	CAMAC crate controller is a (GSI) CBV
CBD	CAMAC crate controller is a (CES) CAMAC Branch Driver (CBD)
CVI	CAMAC crate controller is a (GSI) intelligent controller (CVI/CVC)

If a verify error occurs the same data item is read again. If the two read items are identical the error is considered a write error. If the item that was read the second time is identical to the item written in the first place then the error is considered as a read error. If a wrong X/Q status was detected an appropriate error message is produced, too. Every error is logged to the standard error output and the error log file together with a time stamp as shown below (all values are hexadecimal):

```
17.05.93 16:22:48 - borertest: Error 000:175
Iteration 00000001 bits 654ef7 wrote 654ef7 read 000000 write error
17.05.93 16:22:48 - borer_test: Error 000:175
Iteration 00000001 wrong x/q status (write no x no q; read no x no q)
```

In this case an error occurred at the first iteration - a zero was read back. Since the second verify read gave the same result as the first the error was considered as a write error. This is a typical error if there is no dataway display in the specified slot.

OPTIONS:

-q	quiet mode - don't print log/error messages
-n	don't produce log- and error files
-i	ignore wrong X/Q status

EXAMPLE:

```
borertest -i (0,15,20)
```

test the dataway display in branch 0, crate 15, station 20 ignoring eventually wrong X, Q status.

NOTE:

The X,Q status must be read with a second cycle from the CAMAC crate controller. Therefore two asynchronously running *borertest* programs will mutually destruct their X,Q status. *borertest* must not run parallel to the data acquisition system especially if the CAMAC readout uses *read-until-no-Q* compressed readout.

camac

read/write from/to CAMAC device

SYNTAX:

camac [<opts>] (<b>, <c>, <n>, <a>, <f>) [=[$]<store>] [<opts>]

ENVIRONMENT:

CAMAC_TYPE	not defined	Type of camac interface to use

FUNCTION:

Perform individual *bcnaf* CAMAC cycles. This command is the interactive variant of the *camac* C-preprocessor function. To *bcnaf*-write the value [$]<store> (add the optional '$' sign for hexadecimal values) use the optional assignment qualifier [=[$]<store>]. If that assignment qualifier is omitted a read cycle is assumed. The default output format is decimal.

There are for the various CAMAC modules shell scripts available at

/n5/scmds

which can be used for test and diagnose purposes. They are named corresponding to the module like *lrs2229*, *lrs4434*, *zero_fera*, *sc4800*, *rm2400, init_crate*.

There are several types of CAMAC interfaces supported, which may be selected by defining the environment variable CAMAC_TYPE:

not defined	(default) CAMAC crate controller is a (GSI) CBV
CBV	CAMAC crate controller is a (GSI) CBV
CBD	CAMAC crate controller is a (CES) CAMAC Branch Driver (CBD)
CVI	CAMAC crate controller is a (GSI) intelligent controller (CVI/CVC)

OPTIONS:

-d	debug mode; print the physical address accessed, too
-h	print output data in hexadecimal representation
-s	do only short (16 bit) cycles (only with CES branch driver CBD)
-n	don't print X, Q status (for reading result with shell scripts)

EXAMPLE:

camac (0,15,1,1,16) = $01

If there is an I/O-LAM box in station 1 of crate 15 at branch 0 then this command pulses the P1 output of that module. It is a F16A1 write cycle to that module with the hexadecimal data item 1.

NOTE:

The X,Q status must be read with a second cycle from the CAMAC crate controller. Therefore two simultaneously running *camac* programs will possibly mutually destruct their X,Q status. *camac* must not run parallel to the data acquisition system especially if the CAMAC readout uses *read-until-no-Q* compressed readout.

char_filter

filter character stream from *stdin* to *stdout*

SYNTAX:

char_filter [<opts>]

ENVIRONMENT:

none

FUNCTION:

Filter the characters read from the standard input and send the result to the standard output. This utility is useful for stripping control characters or case conversion of strings handled in shell scripts.

OPTIONS:

-u	convert to upper case
-l	convert to lower case
-a=<num>	strip all characters with the given ASCII code (decimal)

EXAMPLE:

```
date -m ! char_filter -a=58
```

This command removes all colons from the *date -m* output. The resulting 6 digit number corresponds to the time and may be used for creating unique files in shell scripts.

dl350_readout

TDAS sub-processor readout program for the *DL350* FADC

SYNTAX:

dl350_readout [<opts>]

ENVIRONMENT:

LOGPATH	/r0	generic path for writing the generic log/error files
TDAScnfpath	/n10/tdas/cnf	generic path for all *TDAS* configuration files
DL350_cnffile	dl350.cnf	configuration file for the *DL350* readout
DL350_opt_cnffile	dl350.cnf	optional configuration file for the *DL350* readout

FUNCTION:

dl350_readout is a readout program for the Struck *DL350* flash ADC system. It supports the standart *TDAS* communication protocol to interface to sub-processor systems. The base address and size of the dual-ported-memory, which is required for the data exchange, can be configured. *dl350_readout* runs on any VME processor running os9 (E5, E6, E7) with a VSB master interface. The *DL350* flash ADC (FADC) system must be connected to the local VSB bus. Unfortunately the *DL350* VSB slave interface does not work if other VSB slaves are connected to the same δVSB branch. The only work around is to supply for the *DL350* system an own δVSB branch with an own VSC window (*dl350_readout* assumes branch # 1, which corresponds to VSC window # 6).

After starting and initializing it's internal variables *dl350_readout* goes into the *TDAS* shut-down state. In this mode it checks only once per second the *TDAS* communication structure for a startup command. Therefore it consumes only negligible CPU time as long as there is no data acquisition started. Therefore *dl350_readout* is well suited to be started in the startup procedure of the given CPU without influencing normal use of that front end computer if there is no data acquisition running.

If a begin-run command is received, the configuration file is read and the *DL350* FADC system is completely reprogrammed depending on the specifications in the configuration file. All features that will be discussed in the following can be customized in that configuration file. A self documenting sample configuration file can be found at the *doc/cnf* subtree of the *TDAS* disk. The current setup of the *DL350* system is returned as a parameter event after a begin- or end-run command.

The *ALADIN* setup uses a modified *DL350* scanner (the scanner compresses the FADC data and deposits it in a so called hit-buffer-memory). The *ALADIN* scanner may be restarted. Therefore it is possible to effitiently compress the FADC data with different digital thresholds. Every sub scan with a given digital threshold is called a readout group. There are up to 4 readout groups available. What kind of scanner is mounted in the given *DL350* system can be specified in the configuration file. *dl350_readout* supports different groups with a standard *DL350* scanner, too. However the dead time will be much larger since the hit buffer memory must be read out after every sub scan. This feature was implemented to have at the *ALADIN* setup an at least slow going system with a standard scanner if the modified scanner breaks.

The *DL350* system, which is setup in the *CAVE* has in case of the *ALADIN* setup typically a CAMAC crate connected for reading some maintenance data. The implementation of that CAMAC readout is completely equivalent to the *TDAS* CAMAC readout (refer to chapter 3.1.1). However the source- and make files are put at the path */n10/tdas/dl350* instead of */n10/tdas/acquire*. The *dl350_readout* image, that is at the *CMDS* path of the *TDAS* disk can be used for testing. It simply excludes the local CAMAC readout.

For pedestal monitoring there was a pedestal readout mode implemented. If the (configurable) pedestal readout type is received, one of the four readout groups is read out with the digital threshold set to 0. The next pedestal readout will then read the next group (only). Therefore all groups are read in a cyclic order.

The *DL350* system showed a lot of strange responses. Therefore an other diagnostic readout type was implemented – the dump mode. If a dump mode readout type is received, the relevant time range of the whole FADC memory of all channels is read out by the CPU first. It produces a subevent that is compatible with a normal compressed event (except some additional words in the subevent header). Second the normal readout is done. The normal *DL350* compressed event is therefore appended to the dump mode sub-event. Off line diagnostic software in PAW will then compare the compressed event with the uncompressed raw data to prove the functionality of the *DL350* scanner. The dump mode should only be requested during the *spill-off* time since it produces both extreme large data and dead time!

The FADCs have a non linear response function. It typically calibrated by applying a triangular function (using a function generator) to the FADC's test input. For online monitoring of the stability of the FADC's non linear response function an other read out mode the so called test mode was implemented. Since the FADC test inputs should be short cut to ground in case of normal readout to suppress cross talk, a relais switch was required to selectively feed the test signal to the test inputs. *dl350_readout* assumes a *GSI* CAMAC *RM2400* relais multiplexer to be setup in the following way: the output 0 should be connected to all (24) test inputs of all FADC modules. The corresponding input 0 should be terminated with 0Ω, input 3 should be connected to the function generator supplying the appropriate triangular signal. If *dl350_readout* receives such a test mode readout type, it first switches the test inputs of the FADC modules to the function generator by programming the CAMAC *RM2400* module and reruns the FADC sampler to let the FADCs convert that signal. Then the *RM2400* is switched back to the default mode (0Ω). Finally the FADC system is normally read out producing an event with a pulse shape in every channel that corresponds to the FADC response function.

To save dead time the *DL350* scanner could be started with the common stop of the *DL350* sampler. But this imposes the following problem: the *dl350_readout* signals *TDAS* that it may end the dead time if the last sub-scan finished. Therefore the (long) read-out time of the hit buffer memory can be done during the life time of the system. If the next trigger occurs soon, it may restart the scanner while it's hit buffer memory is still being read out and therefore destroy the data of the previous event. If the scanner shall be started externally additional hardware must be supplied to prevent this race condition (an I/O-LAM box is needed in the local CAMAC crate). Figure 4.1 shows how to set the external start scan trigger up. Before signalling *TDAS* to clear the com-

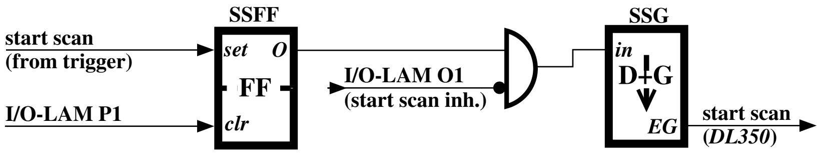

Figure 4.1: A sample trigger input for the DL350 external start scanning signal
This logic prevents the hit buffer memory from being destroyed while it is read out.

mon dead time *dl350_readout* sets the scart scan inhibit (I/O-LAM O1) signal and clears the start scan flip-flop (**SSFF**). If an other start scan trigger occurs during the readout time of the hit buffer memory it is latched (**SSFF**). The **AND** gate prevents *DL350* start scan gates from becoming fired as long as the start scan inhibit signal (I/O-LAM O1) is set.

The *DL350* scanner statistically produces incorrect data. Therefore several online consistency checks were implemented. If one of these checks fails the whole scanning process is retried (up to a maximum retry count). If after the maximal number of retries there is still an error found, the appropriate error flags are set. The following consistency checks are done after the completion of the *DL350* scanner:

- the total data size read from the scanner internal word count register must be equal to the data size word in the hit buffer memory
- there must be a double $ffff tag at the end of the data in the hit buffer memory
- (this check is optional) the last readout interval must have the specified wire number. There is typically somewhere a pulser ored to the raw data for calibration purposes. If this pulser goes into the last channel that is read out, then there must at least this channel/wire number show up in any event.

The following bits are defined in the error pattern (there may be several bits set in a single event), which is put into the sub-event header:

bit value (hex)	**corresponding error status**
1	the total size of the four readout groups was getting larger than 100kB and therefore truncated.
2	illegal scanning state: scanner <u>not</u> ready <u>and not</u> scanning; scanner will be reset and restarted
4	timeout (>5ms) waiting for (sub)scan ready
8	illegal scanning state after scanner was restarted
10	data overrun in the *DL350* hit buffer memory
20	sampler was found still running and therefore stopped by software - FADC data will be trash
40	wait for scan end timed out (> 5ms)
80	*dl350_readout* is <u>not</u> configured to support external start scanner logic but the scanner was started
100	data consistency check failed after maximum retries
200	(dump mode only) while reading the FADC memories a read verify failed.

The following bits are defined in the warning pattern (there may be several bits set in a single event), which is put into the sub-event header:

bit value (hex)	corresponding warning
40	sub scan end command did not finish within 10 µs, scanner was reset and sub scan end assumed
100	scanner was restarted after detecting inconsintent data
200	there was a word count mismatch found between the hit buffer memory and the SIM word count register
400	there was no double $ffff tag at the end of data
800	the optional check of the occurrence of the last wire number in the data failed

dl350_readout produces a data subevent with the following format (the compressed FADC data is an identical copy of the appropriate part of the hit buffer memory):

31 bit 16	15 bit 0
exclusive subevent length	
1000	(-)40
error pattern	readout format (1)
# of presamples	logical crate number
0 (spare)	0 (spare)
event counter low	warning pattern
$ffff	event counter high
relative start of hit	(wire #)<<16 | $ff
time channel 2	time channel 1
time channel 4	time channel 3
⋮	
(wire #)<<16 | $ff	$ffff
time channel 1	relative start of hit
time channel 1	time channel 1
⋮	
$ffff	$ffff

The exclusive subevent length is in units of 16 bit and excludes the first two header long words. The whole event consists of 16 bit items. Each interval of interest (hit) is preceded by the tag word $ffff. The next words specify the wire number and the relative start of the hit in time space. At the end of each readout group there may be two end markers ($ffff) instead of one for alignment.

The dump mode subevent is formatted as follows:

<table>
<tr><th>31 bit 16</th><th>15 bit 0</th></tr>
<tr><td colspan="2">exclusive subevent length</td></tr>
<tr><td>1001</td><td>(-)40</td></tr>
<tr><td>error pattern</td><td>readout format (1)</td></tr>
<tr><td># of presamples</td><td>logical crate number</td></tr>
<tr><td>scan range</td><td>phys. begin address</td></tr>
<tr><td>0 (spare)</td><td>warning pattern</td></tr>
<tr><td>$ffff</td><td>0 (spare)</td></tr>
<tr><td>relative start of hit</td><td>(wire #)<<16 | $ff</td></tr>
<tr><td>time channel 2</td><td>time channel 1</td></tr>
<tr><td>time channel 4</td><td>time channel 3</td></tr>
<tr><td colspan="2">⋮</td></tr>
<tr><td>(wire #)<<16 | $ff</td><td>$ffff</td></tr>
<tr><td>time channel 1</td><td>relative start of hit</td></tr>
<tr><td>time channel 1</td><td>time channel 1</td></tr>
<tr><td colspan="2">⋮</td></tr>
<tr><td>$ffff</td><td>$ffff</td></tr>
</table>

The number of channels that is read out per wire corresponds to the scan range. For alignment and simplicity of the dump mode readout it will be the next higher odd number.

The following parameter event is returned at a begin-run and end-run command. It mainly reflects the parameters read from the configuration file. The subevent has a substructure of variable length elements the so called *regions*. All parameter words (except the sub-evevnt header) are stored as long words with the upper 16 bit set to 0. The *slanted* not centered items are comments:

31 bit 16	15 bit 0
exclusive subevent length	
1000	(-)41
number of regions (4)	
offset to first region	
offset to second region	
offset to third region	
offset to fourth region	
offset to first word past fourth region	
region 1: SAM parameters	SAM clock mode
	start address
	end address
region 2: SIM parameters	SIM CSR
	scan range
	scan offset
	number of presamples
	start wire
	group 0 end wire
	group 0 dig. thr. left
	group 0 dig. thr. right
	⋮
	group 3 end wire
	group 3 dig. thr. left
	group 3 dig. thr. right
	event counter low
	event counter high
	pipeline out
region 3: FADC baseline DAC settings	base line DAC slot0 ch0
	⋮
	base line DAC slot23 ch7
region 4: FADC trigger threshold DAC settings	trig. thr. DAC slot0 ch0
	⋮
	trig. thr. DAC slot23 ch3

OPTIONS:

-d debug level (default 0). <num> is expected as a decimal number. (refer to *TDASstart* for the definition of the various debug levels; currently only debug value 16 is supported)

-s slow down – test *TDAS* running on same node

-x immediately execute implicit *TDAS* startup command

-i interactive mode – echo all messages to stdout

-f fire *DL350* front panel pulse and exit

-t do a readout speed test from the *DL350* hit buffer memory to the *TDAS* dual ported memory

-r run/stop sampler (period 2 seconds) for noise research

EXAMPLES:

```
-nt -ne -nx
setenv LOGPATH /r0
setenv TDAScnfpath /n10/tdas/cnf
setenv DL350_cnffile dl350.cnf
setenv DL350_opt_cnffile dl350_opt.cnf
del /r0/dl350_readout* >>>/nil
restart dl350_readout -d=24 >>>-/r0/dl350_readout.out
```

The command procedure above is typically put into the startup procedure of the *DL350* readout processor. It starts the readout of the *DL350* system with the appropriate environment in the background after deleting possibly existing old log and error files and killing possibly remaining old readout processes. *dl350_readout* does not produce any messages to its standard or error outputs. However these outputs are anyway redirected to a log file (*>>>-/r0/dl350_readout.out*) to capture system error messages like the bus trap trace back.

e6

setup utility for the VME processor E6

SYNTAX:

e6 [<opts>]

ENVIRONMENT:

none

FUNCTION:

This utility is used for setting up the various programmable hardware switches and options that are availble on the ELTEC VME CPU E6. It supports also an option to reconfigure the non volatile ram to configure the boot options.

OPTIONS:

-g perform general setup. This option is typically called once in the startup file. This option performs the following actions:

VME:

- no write posting
- set local/VME timeout to 64/16 µs
- disable local IRQ7
- clear sysfail
- clear AM source register
- set slave select 0 to D32,A32 no block mode support
- set slave select 1 to D32, AM source reg, no block mode support (effectively disable A24 slave select)
- set bus requester to level 3, fair mode timeout 8µs, disable local time out during bus request time
- set bus access mode to ROR (release on request)
- enable D32 mode on VME A24 master cycles
- set VME slave base address to $00000000

VSB:

- switch bounce mode off
- set VSB slave base address to $00000000

-x execute the command – don’t prompt for confirmation

-R pulse the VME sysreset (reset all modules in the crate)

-s pulse the SCSI bus reset signal for about one second

-c interactively reconfigure the on-board non volatile ram. The boot options are defined here. All options are prompted together with the current setup. Returning an empty line leaves the option unchanged.

-b=<addr> set the VME slave extended base address to <addr>. <addr> must be a 16 bit hexadecimal number. However only the upper 10 bits are used. They define A22-A31 of the VME extended slave address. In addition the *bus-error-status-register* of the *VIC068* is reset.

-B=<addr> set the VSB slave base address to <addr>. <addr> must be a 8 bit hexadecimal number. It specifies A24-A31 of the VSB slave address.

-t=<rate> redefine the os9 sysglob variable, that defines the system tick rate. Some of the VIC068 chips that were delivered on the E6en have a bug that changes the system tick interrupt rate if the VME Slave Select 0 Control register 0 is reprogrammed. With this option the os9 system tick rate can be readjusted.

-e enable external cache (50 MHz version only)

-d disable external cache (50 MHz version only)

-f=<num> set the front panel hex display to <num>. <num> must be a one digit hexadecimal number.

EXAMPLES:

e6 -g

This command is typically put into the startup routine of all E6 processors to have a common general setup. If a bus error occures during that command there is typically no or several VME arbiters installed in the VME crate. The E6 needs the correct 16MHz 50% duty cycle VME system clock for proper operation of it's VSB chip.

NOTE:

Not all software programmable options may be configured with the e6 program. The here stated default hardware initialization is not the optimal for any setup. However it turned out to work best with the largest variety of different modules. So in special cases changes to the default hardware initialization may be advised.

Since the *e6* program accesses CPU and operating system related resources, the sources of that program are not at */n5* but together with the os9 boot and driver stuff at the network disk *os9src0.disk*.

e7

setup utility for the VME processor E7

SYNTAX:

e7 [<opts>]

ENVIRONMENT:

none

FUNCTION:

This utility is used for setting up the various programmable hardware switches and options that are availble on the ELTEC VME CPU E7. It supports also an option to reconfigure the non volatile ram to configure the boot options.

OPTIONS:

-g perform general setup. This option is typically called once in the startup file. This option performs the following actions:

VME:

- no write posting
- set local/VME timeout to 64/16 µs
- disable local IRQ7
- clear sysfail
- clear AM source register
- set slave select 0 to D32,A32 no block mode support
- set slave select 1 to D32, AM source reg, no block mode support (effectively disable A24 slave select)
- set bus requester to level 3, fair mode timeout 8µs, disable local time out during bus request time
- set bus access mode to ROR (release on request)

VSB:

- switch bounce mode off
- set VSB slave base address to $00000000

-x execute the command – don't prompt for confirmation

-R pulse the VME sysreset (all modules in the crate are resetted)

-s pulse the SCSI bus reset signal, set hex display to <hexnm>, enable snooping

-c	interactively reconfigure the on-board non volatile ram. The boot options are defined here. All options are prompted together with the current setup. Returning an empty line leaves the option unchanged.
-b=<addr>	set the VME slave extended base address to <addr>. <addr> must be a 8 bit hexadecimal number. It defines A24-A31 of the VME extended slave address.
-B=<addr>	set the VSB slave base address to <addr>. <addr> must be a 8 bit hexadecimal number. It specifies A24-A31 of the VSB slave address.
-e	enable snooping, set hex display to <hexnm>.
-d	disable snooping, set hex display to <hexnm>.
-f=<hexnm>	define the front panel hex display value. <hexnm> must be a one digit hexadecimal number, the default is 4. This option does NOT write to the hex display. It is updated if one of the following options are set: -e -d -s.

EXAMPLES:

```
e7 -g
```

This command is typically put into the startup routine of all E6 processors to have a common general setup. If a bus error occures during that command there is typically no or several VME arbiters installed in the VME crate. The E6 needs the correct 16MHz 50% duty cycle VME system clock for proper operation of it's VSB chip.

NOTE:

Not all software programmable options may be configured with the e7 program. The here stated default hardware initialization is not the optimal for any setup. However it turned out to work best with the largest variety of different modules. So in special cases changes to the default hardware initialization may be advised.

Since the *e7* program accesses CPU and operating system related resources, the sources of that program are not at */n5* but together with the os9 boot and driver stuff at the network disk *os9src0.disk*.

eso_ser RPC server process for remote ESONE CAMAC access

SYNTAX:

eso_ser

ENVIRONMENT:

RPC_CLIENT_NAME *_<proto>.ETHERNET RPC protocol

FUNCTION:

eso_ser is a stateless server for remote ESONE CAMAC accesses [ESONE]. However all supported ESONE functions are available on the front end computer itself by linking to the *ulib.l* (refer to *cam_funcs.c*). The following ESONE functions are supported (for a detailt description of these routines and functions refer to the ESONE standard [ESONE]):

ctstat, cdreg, cgreg, cfga, csga, cfsa, cssa, cfubc, csubc, cfubr, csubr, cccz, cccc, ccci, ctci

On the host computers there is both a C- and a Fortran binding available (here it was put together with the high voltage control software into the CMS library called *setup*).

The *eso_ser* protocol is based on the Cern RPC packet [RPC]. *eso_ser* needs the ethernet protocol number to listen to as an environment variable. To have some protection against unintended foreign accesses, each CAVEB group was assigned an own ehternet protocol number for the RPC transport as follows:

4π	<proto> = 5012
ALADIN	<proto> = 5011
LAND	<proto> = 5010

At the host computer the physical ethernet and protol number must be specified before running an ESONE client application. The physical ethernet address of a front end computer can be acquired with the moni -e os9 command. The eternet protocol number is defined in the table above. The ESONE client package is currently implemented only on Vaxen. Here the desired target front end computer must be defined with the logical name *ESOPACK*. The following command defines the ALADIN front end computer *e7kp3a* as the esone server:

```
define/job esopack 00_00_5B_07_00_26_5011.ethernet
```

OPTIONS:

none

EXAMPLE:

```
setenv RPC_CLIENT_NAME "*_5011.ETHERNET"
restart /n5/cmds/eso_ser >>>/nil ^200 &
```

The best way to start the ESONE server is to put it into the group startup file of the appropriate front end computer like in the example above. Make sure that the correct protocol number is specified.

NOTE:

If a non existing crate or branch is accessed a bus error will be the result. In such a case no X and no Q status is returned. In case of a read command the impossible data word $ffffffff (-1) is returned.

fastbus_readout

TDAS sub-processor readout program for the AEB

SYNTAX:

fastbus_readout [<opts>]

ENVIRONMENT:

LOGPATH	/r0	generic path for writing the generic log/error files
PEDESTAL_PATH	/n10/data	path for reading the individual pedestal parameter
TDAScnfpath	/n10/tdas/cnf	generic path for all *TDAS* configuration files
FBRD_cnffile	fbdef.cnf	configuration file for the Fastbus readout
FBRD_datfile	pedestal.dat	pedestal data file name

FUNCTION:

The program *fastbus_readout* runs on an Aleph-Event-Builder (AEB) as a Fastbus readout program of a *TDAS* sub-processor. LeCroy QDCs and TDCs are supported. It communicates to the *TDAS event-builder* through the dual-ported-memory EBI using the *TDAS* protocol for interfacing with intelligent sub-systems. Two different readout modes are supported: normal, compressed and uncompressed readout. If compressed readout is selected first all specified Fastbus modules are read out. Then for each channel an individual pedestal is subtracted and the channel is written to the data stream if it's count is larger than a certain threshold. The pedestal of all channels of all modules in a given crate can be determined with the *pedestal* program. *fastbus_readout* can directly read the *pedestal* output data file (environment FBRD_datfile) at the path PEDESTAL_PATH. The readout threshold above the given individual pedestal can be specified to be a certain absolute value or any factor of the rms value of the pedestal as determined with the *pedestal* program. If uncompressed readout is selected with the *readout-type* all channels are read out and there is neither the pedestal subtracted nor are the channels suppressed that are below the readout threshold.

fastbus_readout allocates its internal (large) buffers if it receives a begin-run command. Therefore it is possible to start that program in the startup procedure of the AEB without wasting too much of memory. As long as it is in shut-down state *fastbus_readout* consumes only negligible CPU time. What Fastbus modules and how many channels of each of them shall be read out can be defined in a self documenting configuration file (environment FBRD_cnffile) at the path TDAScnfpath. The required setup of any of these Fastbus modules can be also comfortably programmed there. The default is for both LeCroy TDC and QDC modules: test pulser disabled, MPI external, Wait disabled, gate front panel (QDC). Both the pedestal data file and the configuration file are read at every begin-run command and all Fastbus modules are reset and reprogrammed appropriately. A sample self documenting configuration file (called *fbdef.cnf*) can be found at the *doc/cnf* subtree of the *TDAS* disk */n5*.

fastbus_readout produces a data subevent with the following format (there is no difference between compressed and uncompressed readout concerning the format of the sub-event; the uncompressed readout simply contains all channels that are read out independent of the contents. In both modes the upper 16 bit of the data word contain the geographical address and the module's relative channel number; The *logical crate number* can be specified in the configuration file to disentangle the subevents from different Fastbus crates):

31 bit 16	15 bit 0
exclusive subevent length	
1001	(-)32
0 (spare)	logical crate number
module type (0 for LeCroy)	
first fastbus data word	
second fastbus data word	
⋮	
last fastbus data word	

this appendix follows if an error occurred (subevent type 32 will then be negative)
it's length is of course included in the exclusive subevent length
there is one entry for every module that returned an error during readout

fastbus error code
corresponding geographical address of module returning that error code
⋮
fastbus error code
corresponding geographical address of module returning that error code
inclusive length of error trailor

The exclusive subevent length is in units of 16 bit and excludes the first two header long words. The error trailor is appended only if at least one error occurred. The fastbus error codes are defined in the include file *cfbdef.h*. However the utility fbrw -e=<error_code> can be used for decoding it.

The following parameter event that is returned if a begin-run command is received. The pedestal and readout threshold data words are formatted like the Fastbus raw data words – the upper 16 bit define the geographical address and the module's channel number of that channel.):

31 bit 16	15 bit 0
exclusive subevent length	
1001	33
0 (spare)	logical crate number
module type (0 for LeCroy)	
pedestal of first fastbus channel	
readout threshold of first fastbus channel	
⋮	
pedestal of last fastbus channel	
readout threshold of last fastbus channel	

If an error occurs during the initialization phase, the initialization of the modules is cancelled and the error is logged to the error file. In that case an error status is returned and no parameter event is produced.

OPTIONS:

-d=<num> debug level (default 0). <num> is expected as a decimal number. (refer to *TDASstart* for the definition of the various debug levels;

currently only debug values 8 and 16 are supported; if debug value 8 is selected all associated Fastbus writes and verifies are logged together with the received commands.)

-i interactive mode – echo all messages to stdout

-a=<level> set Fastbus bus requester level to <level> (default 6). There may be only one potential Fastbus master requesting the bus at that level. So if two AEBs are mounted in one Fastbus crate one AEB must have a different level programmed.

EXAMPLES:

```
setenv LOGPATH              "/r0"
setenv PEDESTAL_PATH        "/n10/data"
setenv TDAScnfpath          "/n10/tdas/cnf"
setenv FBRD_cnffile         "fbdef.cnf"
setenv FBRD_datfile         "pedestal.dat"
fastbus_readout -a=6 >>>/nil &
```

The command procedure above can be typically found in the startup procedure of the AEBs that are connected to a readout branch. It starts the *fastbus_readout* program in the background with the specified environment. The here stated environment is of course highly setup dependant since if several Fastbus Crates and AEBs are incorporated into a given setup, several different configuration and pedestal data files must be available.

NOTE:

This utility runs only on the Fastbus CPU Aleph-Event-Builder (AEB). If it is started on another processor the following error output will occur:

```
FB_SEV_FATAL: Status code DEFAC0DE --> FECLSD - ID: 00000000
Detected by routine 'fb_open'
```

fbrw

read/write from/to Fastbus device

SYNTAX:

fbrw [<opts>] (<prim. addr.>, <sec. addr.>) [=[$]<store>] [<opts>]

ENVIRONMENT:

none

FUNCTION:

fbrw is a maintenance Fastbus utility equivalent to the *camac* utility for CAMAC devices. It can be used to program or check the CSR space or to readout Fastbus modules. *fbrw* uses geographic addressing. Therefore to access a module it needs its Fastbus slot, which is considered as <primary address>. Any subaddress inside the module is passed as <secondary address>. Like the *camac* utility read cycles are default. To write a data item to the specified address add the assignment option [=[$]<store>]. For hexadecimal values add a $ in front of the data item.

OPTIONS:

-d	debug mode
-h	print output data in hexadecimal representation
-s	access data space (default: control space)
-b	execute as broad cast cycle
-f	pulse front panel ECL IRQ acknowledge output of the AEB for about 100ns. This output can be ored into the trigger logic. The AEB can then produce pedestal triggers. If this option is specified all other arguments are ignored.
-c	fire the CAT calibration trigger output
-r	fire a Fastbus reset cycle
-e=<code>	decode the Fastbus error code <code> (hexadecimal)

EXAMPLE:

fbrw (20,0) -h

Read the CSR0 control register of the Fastbus module in slot 20.

NOTE:

This utility runs only on the Fastbus CPU Aleph-Event-Builder (AEB). If it is started on another processor the following error output will occur:

FB_SEV_FATAL: Status code DEFAC0DE --> FECLSD - ID: 00000000

Detected by routine ’fb_open’

fbscan

perform a Fastbus crate scan

SYNTAX:

fbscan

ENVIRONMENT:

none

FUNCTION:

fbscan performs a crate scan and tries to identify the Fastbus modules found by reading and decoding the upper 16 bit of the CSR0. For every Fastbus slot one line will be output with the appropriate CSR0 (or 0 if the slot is empty) in hexadecimal representation. If the module type is known it is also identified. A typical fbscan output of the first 5 slots looks like:

Slot # : 00 - csr0: 00000000 no FASTBUS board inserted

Slot # : 01 - csr0: 68580000 Struck Aleph Event Builder

Slot # : 02 - csr0: 00000000 no FASTBUS board inserted

Slot # : 03 - csr0: 10453300 LeCroy ADC Type 1885F

Slot # : 04 - csr0: 10490000 LeCroy TDC Type 1875F

fbscan is also useful to see if the Fastbus arbitration and interlock works (if not *fbscan* will terminate with an error).

OPTIONS:

none

EXAMPLE:

fbscan

A crate scan is performed - the CSR0 of every Fastbus station is shown and identified.

NOTE:

This utility runs only on the Fastbus CPU Aleph-Event-Builder (AEB). If it is started on another processor the following error output will occur:

FB_SEV_FATAL: Status code DEFAC0DE --> FECLSD - ID: 00000000

Detected by routine ’fb_open’

memorytest test/verify memory areas

SYNTAX:

memorytest [<opts>] (begin addr, end addr)

[<opts>]

ENVIRONMENT:

LOGPATH	/r0	generic path for writing log/error files

FUNCTION:

memorytest is used to test memory areas. If the memory is located offboard also all components like cables driver/receiver boards (for example VSC) are tested, too. The program needs two arguments the physical begin address (inclusive) and end address (excluseve) (both addresses hexadecimal) of the memory region as seen from the local CPU. Always the whole memory region is filled with some pattern and it is then verified to have that pattern. All memory accesses are 32 bit cycles (longwords). The test sequence is as follows: First write and verify the following fixed patterns to the whole memory region: $00000000, $FFFFFFFF, $55555555, $AAAAAAAA, $A5A5A5A5, $5A5A5A5A, $00000000. Second as back to back read/write cycles write the *exclusive ored* pattern of every memory location back to it and verify the whole address range for the correct result. The following patterns are use yielding the in brackets indicated new data items: $FFFFFFFF (–>$FFFFFFFF), $FFFFFFFF (–>$00000000), $55555555 (–>$55555555), $55555555 (–>$00000000), $55555555 (–>$55555555), $AAAAAAAA (–>FFFFFFFF), $FFFFFFFF (–>$00000000). Finally loop infinitely with random patterns in the memory range. It starts with pattern $77777777. The pattern is offset every memory location by $075BCD15 (decimal 123456789). Every loop the start pattern is offset by $DEADFACE. To abort the infinite test loop and get the total number of errors found type 'x' at the standard input of *memorytest*.

If a verify error occurs the same memory location is read again. If the two read items are identical the error is considered a write error. If the item that was read the second time is identical to the item written then the error is considered as a write error. Every error is logged to the standard error output and the error log file together with a time stamp like (all items shown are hexadecimal):

```
15.05.93 16:33:01 - memorytest: Error 000:175
Address f420a108 bits 00000004 wrote 5ef199d5 read 5ef199d1 write error
```

In this case an error occurred at address $F420A108 - bit number 2 (counting from 0) did not match. Since the second verify read gave the same result as the first the error was considered as a write error. If such a (write) error occurs on every address of the whole address range and if there are only some bits that don't match the most probable error is a broken δVSB cable or a δVSB receiver/transmitter card like the VSC. Intermittant random errors are more serious and hard to debug.

OPTIONS:

-q	quiet mode - don't print log messages
-n	don't produce log- and error files
-r	read only test. Every iteration the memory is copied first into a hash buffer, which is then verified with that memory region.
-s	speed test. A read/write speed test is performed at the specified memory region. For that purpose the memory range specified must be 32 bytes large.

EXAMPLE:

```
memorytest f4200100 f4210000
```

This command performs a memory test in the EBI with crate number 1 at δVSB branch 1. The first $100 bytes are excluded since there is the *fastbus_readout* communication structure located. Another more comfortable way to test the sub system's dual ported memories *TDAScpudbg*.

NOTE:

The *memorytest* program assumes that the operator does what he is doing - there are no checks included to protect the memory of the local CPU from being tested/discarded. Therefore the following command is a safe way to crash the front end computer:

```
memorytest 0 400000
```

If dual ported memories (like the EBI) are tested it is recommended to simultaneously run two *memorytest* programs on the two CPUs accessing that memory. This can be achieved by running *memorytest* of the first CPU (for example E6) in the lower half of that memory and the memory test of the second CPU (AEB) in the upper half of the memory. It is well possible that the individual memory tests succeed and the simultaneous one fails.

pedestal utilitiy for determining and monitoring Fastbus pedestals

SYNTAX:

pedestal [<opts>]

ENVIRONMENT:

TDAScnfpath	/n10/tdas/cnf	generic path for all *TDAS* configuration files
PEDESTAL_PATH	/n10/data	path for writing pedestal data files

FUNCTION:

The *pedestal* utility is used for determining the pedestal of the Fastbus QDCs and TDCs. It contains also a gain test mode. For proper operation of the *pedestal* program (the clear is done with a *8D* broadcast) a CAT module must be mounted somewhere in the Fastbus crate. First of all *pedestal* performs a crate scan to identify all modules mounted. Then every known module is internally (if not -f option specified) gated (QDC) and pulsed (TDC) 100 times in high and low range and the whole module is read out. For every individual channel the mean and rms of the pedestal is calculated and written to the file pedestal.dat at the path specified with the environment variable PEDESTAL_PATH. If this environment variable is not specified the default path /r0 is assumed. Finally an overall mean pedestal is calculated and all channels are printed out that differ more than 30% from that mean pedestal.

If *pedestal* is run in the gain test mode, it uses the built in self test function of the LeCroy TDCs and QDCs. They are able to internally gate (QDC) and start/stop (TDC) themselves depending on an analog voltage supplied at the back plane. The CAT module has a programmable DAC onboard which drives this signal. The gain test ramps the DAC, triggers the internal self test unit of all modules in the crate and reads them out. Then a stright line fit is applied to the data of every individual channel in both ranges. The resulting slope (gain), intercept (fit pedestal) and fit χ^2 is written to the file gain.dat at the same path like pedestal.dat. The fit pedestal of the QDCs should correspond to the *real* pedestal of the QDCs. However the TDCs should produce a negative fit pedestal while the actual pedestal (the LeCroy TDC is actually a time to charge converter with a QDC that is read out) should be zero. The very first channels of the Fastbus modules deviate from a linear behaviour. They should be excluded from the straight line fit. Therefore there is a configuration file pedestal.cnf that specifies the DAC range and increment per iteration for both gains (high gain/low range comes first). Here are typically the very smallest channels excluded. The configuration file is self documenting. *pedestal* tries to open the configuration file *pedestal.cnf* at the path specified with the environment variable TDAScnfpath.

The pedestal and gain data files are self documenting ASCII files. They have both the same explaning comment header. The header contains also the result of the crate scan. The two QDC and TDC ranges are identified *high* for high range/low gain and *low* for low range/high gain. Sample configuration and data files can be found at the *doc/cnf* subtree of the *TDAS* network disk. Both data files are default written in append mode. Therefore (for example drift) data can be accumulated in these files.

OPTIONS:

-p[=<num>]	produce a pedestal data file (default) with <num> (default 100) iterations
-g	produce a gain test data file
-c	compare the actual data with the data file and print different channels
-a	append write to the data file - do not discard it
-v	include the width/χ^2 in the bounds check
-f	(pedestal mode only) don't internally pulse the modules but use the external trigger logic. Since the pedestals of both the QDC and TDCs depend on the timing (gate length, arm/stop timing) for the experiment the real signals have to be used to produce relevant pedestals. Therefore the QDCs are programmed to accept only the front panel gate (The TDCs or their front panel and back plane trigger signals). Then before every iteration the AEB's front panel ECL IRQ acknowledge output is pulsed for about 100ns. This signal must be ored into the Fastbus master-trigger.
-u=<num>	use <num> as upper bound (in per cent) for the bound check of all channels (default 30% from general mean or 1% from individual result in case of -c option)
-l=<num>	use <num> as lower bound (in per cent) for the bound check of all channels (default 30 from general mean or 1% from individual result in case of -c option)

EXAMPLE:

pedestal -rfp

This command produces a new pedestal data file. A potentially existing one is discarded. The QDCs and TDCs are triggered with the external experimental hardware trigger. The so produced pedestals may be used by the *fastbus_readout* program for the online data reduction.

NOTE:

This utility runs only on the Fastbus CPU Aleph-Event-Builder (AEB). If it is started on another processor the following error output will occur:

FB_SEV_FATAL: Status code DEFAC0DE --> FECLSD - ID: 00000000

Detected by routine ’fb_open’

restart stop all processes with given name and restart it wit given options

SYNTAX:

restart <image path string> [<opts>]

ENVIRONMENT:

none

FUNCTION:

Kill all processes with the given name. Then restart the process with the given command line options. This utility is useful to restart server processes like daemons for example after a change of the environment.

OPTIONS:

The passed options depend on the program that is to be started.

EXAMPLE:

restart /n5/cmds/eso_ser >>>/nil ^200 &

This command kills a possibly running esone server process *eso_ser* and starts it again with its output redirected to the */nil* device and with a base priority of 200. Since the esone server is stateless it may be (re)startet any time. Without the restart utility the discussed command would possibly result in several simultaneous esone servers, which would certainly cause protocol problems.

NOTE:

The environment variable PATH is not searched for the image. Therefore the full image path string should be specified. Be careful using that utility. For example the command

restart shell

will cancel all other shells, which will effektively will log anybody off.

setprio

set process priority by name

SYNTAX:

setprio <priority> <process name list>

ENVIRONMENT:

none

FUNCTION:

Change the priority of the processes with the given mane(s) to <priority>. The process names are not considered as case sensitive. If the process name is specified as *ourselves* the calling process of *setprio* is modified. This utility is useful to change the priority of process classes like daemons or to modify a process with a known name but with an unknown process id in a shell script. *setprio* prints a log statement to *stdout*, of how many processes of the given name(s) have been modified.

OPTIONS:

none

EXAMPLE:

setprio 32 TDASaccessdc

Set the process priority of all *TDAS* access connection handlers to 32.

setup_scsi_disk

utility for spinning up/down a SCSI disk

SYNTAX:

setup_scsi_disk [<opts>]

ENVIRONMENT:

none

FUNCTION:

Utility for starting/stopping a SCSI disk. Most of the SCSI disks are jumpered to perform an auto spin up at power up stage. However some devices (like the DEC RZ23) need a special command for doing so. This is useful if many SCSI disks are connected to the same power supply since the disks spin up sequence needs a much higher supply current than normal operation. The disks may then be started one after the other without overloading the power supply.

OPTIONS:

-i=<ID>	required option specifying the SCSI ID of the device
-t	perform a stop (spin down) instead of the default start (spin up)

EXAMPLE:

start_scsi_disk -i=3

Send a start command to the SCSI disk with the SCSI ID 3 and wait for command completion. If the disk is already started return immediately.

NOTE:

A spinned down disc cannot be read from or written to.

setup_vmv setup program for the CES VIC 8250 vertical bus module

SYNTAX:

setup_vmv [<opts>]

ENVIRONMENT:

none

FUNCTION:

The CES vertical bus module is a tripple ported 128k memory with a VMV master/slave, a VME master/slave and a VSB slave interface. The VME base address is selectable within the VME standard address space (A24) in 1MB steps with a hex switch (required -s option). The VSB interface is then programmed to have the specified VSB slave base address of the window of (fixed) 16MB size (-v option). The appropriate VME and VSB base addresses are printed out if the command succeeds.

OPTIONS: (both options are required!)

-s=<num>	setting of hex switch defining the VME slave address
--v=<num>	VSB slave address (A24-A31) in hexadecimal

EXAMPLE:

setup_vmv -a=e -v=f0

Sets the VME base address (A24) of the VIC 8250 to $E00000 and the VSB base address of the module to $F0000000.

NOTE:

As a matter of fact the VIC8250 behaves extremely strange if it is operated without a VSB terminator. Evan VME cycles fail (bus traps) without the VSB bus being terminated. If it was started without a VSB terminator cases have been reported, where after mounting the VSB terminator (e.g. mounting a VSC card) the power had to be cycled to get the module work properly.

shutdown

perform an orderly shutdown and reboot

SYNTAX:

shutdown

ENVIRONMENT:

SHUTDOWN_PATH	/n10/sysdump	path to write system crash dump files
HOSTNAME	e7kp3a	(unique) internet host name

FUNCTION:

If a front end computer seems to behave so strange that a reboot seem adequate and it is still possible to log into that computer, then the *shutdown* command is strongly recommended. *shutdown* is a shell script file that saves first the whole context of *TDAS* and os9 to the subtree with the name of the host computer as specified with the environment variable HOSTNAME at the path as specified by the environment variable SHUTDOWN_PATH. Then the front end computer is rebooted. This reboot comprises of a real reset as if the front panel butten would have been pressed. The system managers will thank you for using shutdown since later diagnostics is possible to isolate possible errors.

OPTIONS:

none

EXAMPLES:

shutdown

Reboot the front end computer and save prior to resetting it as much as possible information for later diagnostics.

NOTE:

In case of the front end processors E6 and E7 the reset that is fired is a VME System Reset – that means any board in the given VME crate and any device at any δVSB branch that is connected to any module in that crate will be reset!

TDASaccessd

daemon serving remote raw data access connection requests

SYNTAX:

tdasaccessd [<opts>]

ENVIRONMENT:

LOGPATH:	/r0	path for writing log/error files
HOSTNAME	e7kp3a	(unique) internet host name
TDASAccessPort	not defined	Internet port number if not 1999

SUPPORTED *TDAS* ONLINE COMMANDS:

STARTUP	*TDAS* is in started state - allow connection requests
TERMINATE	*TDAS* is in shut down state now - refuse connection requests
HELP	print help text on all or specified command

FUNCTION:

TDASaccessd is the connection demon for the *TDAS* raw data access. It has two states corresponding to the state of *TDAS*. If *TDASaccessd* is started it goes into the *TDAS* not started state. All remote connection requests are refused in this state. If *TDASaccessd* however receives a STARTUP all further connection requests are accepted and a connection handler *TDASaccessdc* is forked, which inherits the appropriate open socket. The actual data selecting and transferring is done by the connection handler *TDASaccessdc*. If *TDASaccessd* receives later a TERMINATE command, it goes back to the not started state and kills all *TDAS* connection handlers since they are about to loose their context. The STARTUP and TERMINATE commands are sent by *TDASstart* to *TDASaccessd* automatically. There is no need for the user to maintain *TDASaccessd.*

OPTIONS:

-d=<num> debug level (default 0). <num> is expected as a decimal number. (refer to *TDASstart* for the definition of the various debug levels)

NOTE:

TDASaccessd is typically not called directly but as a subprocess by *TDASstart*. It does not terminate at the TERMINATE command but goes to the stopped state. So it survives *TDAS* shutdowns. Therefore later connection requests do not time out but are immediately refused.

TDASaccessdc

Test

SYNTAX:

TDASaccessdc [<opts>]

ENVIRONMENT:

LOGPATH:	/r0	path for writing log/error files
TDASAccessEvtSize	not defined	default maximum event size if not *TDAS* maximum event size parameter (*TDASpar.cnf*) for allocating the internal event buffer

FUNCTION:

TDAsaccessdc is the connection handler transferring selected raw data to any remote host through it's open TCP socket, which it inherited from *TDASaccessd*. During the startup of *TDASaccessdc* the raw data filter and options are read from the open socket. Then it falls into an endless sampling loop, where it samples the raw data in the *TDAS* cyclic buffer structures. If an event is found that matches the filter criteria that were read at startup time, it is transferred to the host computer. There is no special handshaking required since this is already done at the TCP layer. *TDASaccessdc* will continue transferring data with the given filter criteria until it receives a *pipe-broken* error or it is terminated by *TDASaccessd*. Therefore the only way to change the raw data filter criteria and options is to close the connection to the data acquisition computer and restablish a new connection with the appropriate new definitions.

The following filter criteria may be applied to select one or several subevents out of a of a raw data event: First a special subevent type and subevent subtype may be selected (both optionally). In addition an optional bit mask mask can be applied to any word of the sub-event. The following logical operations are available (C syntax):

```
select_sub_event = bit_mask & data_word
select_sub_event = bit_mask == (bit_mask & data_word)
select_sub_event = bit_mask == data_word
```

The first option requires at least one bit of the bit-mask to be set in the data word. The second option requires that at least all the bits of the bit-mask are set in the data word and the last filter option requires that the data word and the bit-mask are identical. The 32 bit data word that shall be compred with the specified bit-mask is defined as offset (starting at 0) relative to the start of the sub-event. Up to 16 different filters can be defined to select several different subevent out of a given event.

TDASaccessdc copies the selected sub-event(s) first into a local buffer before it starts to transfer the raw data to the host. This achieves a minimum access time to the *TDAS* buffers and therefore a minimal blocking of the main acquisition processes *TDASacquire* and *TDASarchive*. By default the local *TDASaccessdc* buffer is allocated with the maximum event size (or what ever is specified by the environment variable TDASAccessEvtSize). However most of the application require only a small portion of the total event (for example while debugging small (sub)detectors). Therefore the size of the local buffer may also be specified as an optional parameter to reduce the amount of memory required per *TDASaccessdc* process.

The begin-run parameter event exists only for a very short time of one *TDAS* buffer round trip. Therefore it is also stored in the data module *TDASparbuf_m* for access of *TDASaccessdc*. With another option *TDASaccessdc* will send as very first event the full parameter event stored in the mentioned data module.

If *TDASaccessdc* has sampled an event and successfully transferred it to the host it will try to access the next event in the same *TDAS* data buffer. If however that buffer was rewritten in the mean time by *TDASacquire* there are two selectable strategies to recover:

go to the first event of the eldest available *TDAS* data buffer.
go to the first event of the newest available *TDAS* data buffer

The first option should be selected if *TDASaccessdc* transferres small or seldom subevents or in other words if *TDASaccessdc* is usually faster than *TDASacquire*. It will therefore receive access to the maximal possible available data. The second option should be selected if it is important to have alwais the most as possible up to date information. For example monitoring the spill structure with a multiscaler would require to display always the latest spill, therefore to access always the latest data.

OPTIONS:

-d=<num> debug level (default 0). <num> is expected as a decimal number. (refer to *TDASstart* for the definition of the various debug levels)

-s=<num> pathnum of open TCP socket to send selected raw data to

NOTE:

TDASaccessd is typically not called directly but as a subprocess by *TDASstart*.

All filters and *TDASaccessdc* options can be defined inside the *PAW* environment at the command subtree */RAW/...*

TDASacquire_s
TDASacquire

TDAS readout processes

SYNTAX:

TDASacquire [<opts>]

ENVIRONMENT:

LOGPATH	/r0	generic path for writing the generic log/error files
TDASlogfile	tdas.log	name of the generic *TDAS* log file
TDASerrfile	tdas.err	name of the generic *TDAS* error file
TDAScnfpath	/n10/tdas/cnf	generic path for all *TDAS* configuration files

SUPPORTED *TDAS* ONLINE COMMANDS (used by *tdasmenu*):

BEGRUN	begin a run
ENDRUN	end a run
SUSPENDRUN	suspend a run
RESUMERUN	resume a suspended run
__BEGRUN	TDAS internal command - don't use
__ENDRUN	TDAS internal command - don't use
HELP	print help text on all or specified command
TERMINATE	terminate process after proper cleanup

FUNCTION:

TDASacquire does the readout of all modules and compiles the event. It is the data acquisition kernel process, which also performs the event building. *TDASacquire_s* is a system state process which essentially acts as the readout interrupt handler and triggers the readout functions inside the scope of *TDASacquire*. *TDASacquire_s* is started and stopped by *TDASacquire*. These two processes can be considered as one functional unit.

The readout interrupt is expected as a VSB interrupt from an I/O-LAM box (which hardware setup is required to trigger *TDASacquire(_s)* is described in chapter 2.1). There were three different modes implemented to handle the readout. They are selected with the command line option -m. If the interrupt mode is selected, the readout routine(s) are called directly in the interrupt handler routine. This is the fastest option. However there are several disadvantages connected with that option. First if the data acquisition is run with high dead time which corresponds to a high percentage of the time the processor spends in the interrupt state, all user state processes – and especially the command interface *tdasmenu* and all monitor processes – will become extremely slow. The two other options (*signal, event*) are roughly equivalent. In both cases the interrupt handler only serves and clears the interrupt request from the I/O-LAM box and triggers then the actual readout. In case of the *signal* mode (-m=1) the readout is triggered with signal number 256. In case of the *event* mode (-m=2) an os9 event named *TDASevent_e* is created. If this event variable is signalled by the interrupt handler routine (becomes 1) the readout is executed. The *event* readout is strongly recommended. The additional time delay for propagating the *signal* or *event* trigger through the operating system is roughly depending on the CPU type and implementation 200µs. However the conversion time of the Fastbus modules is 250µs.

The actual readout comprises of the following steps. First the readout type is acquired from the pattern unit in station 2 of the *TDAS system-crate* (CAMAC branch 0, crate 15 – refer to chapter 2.1). If the pattern unit was not gated and contains the value 0 the readout trigger is considered as an error, the global error counter *readout type 0* is incremented and the modules are cleared without being read out. Otherwise the *readout-type* is passed as readout command to all synchronous sub-processors which themselves start immediately the readout of their associated modules and produce a so-called *sub-event*. If a socalled *coincidence-readout-type* was selected the asynchronous sub-processors are triggered, too. The principal difference between synchronous and asynchronous sub-processors is their handling by *TDASacquire.* The synchronus sub-processors are every event and must respond to the trigger. The asynchronous sub-processors are triggered only if a *coincidence-readout-type* is fired and may or may not respond to the readout command. The next step of *TDASacquire* is to perform the CAMAC readout. The CAMAC readout is highly experiment specific. It is set up as special C source file which is linked to the *TDASacquire* process. This routine must provide a CAMAC *sub-event.* After finishing the CAMAC readout *TDASacquire* waits for all sub-processors to finish their readout. If all sub-processors are at least in a state, where the data is read from all data acquisition modules, the dead time is terminated. Therefore some overlap between the actual sub-processor readout time and the system's life time is achieved. If a sub-processor finishes its readout it responds the size of the produced *sub-event* which is then transferred by *TDASacquire* and compiled into the main data stream. If all synchronous sub-systems are read out all asynchronous sub-processors are checked to be ready and to have valid data. If there is a valid asynchronous sub-event it is put into the data stream, if not this system is ignored. Asynchronous sub-processors cannot therefore produce additional dead time. However they have to be fast enough to produce their data in time. If any sub-processor returns an error status or if an asynchronous sub-processor was not ready the appropriate global error counter is incremented in the data module *TDASmain_m.*

At the end of the readout *TDAsacquire* checks if a buffer state transition is required. There are three possible reasons for a buffer state transition. First if a *spill-off readout-type* was fired. After processing the *spill-off* event *TDASacquire* switches to a new, free buffer to start the next spill with a free buffer. The size of a *TDAS* buffer should be adjusted such, that it can contain a full spill of data. Therefore the number of buffer state transitions which occur during the spill are kept minimal. If however less than *max_event_size* (parameter of the self configuration file *TDASpar.cnf*) bytes are left in the current buffer, therefore an event with maximal size would not fit into the buffer, a buffer start transition is enfoced, too. The last possible reason for buffer state transitions is typically required for test setups without spill triggers and with low rates. If a buffer is kept by *TDASacquire* for a longer time than the parameter *max_buf_age* specified in the configuration file *TDASpar.cnf* a buffer state transition is enforced, too. Therefore TDAS is prevented from keeping very old, stall data in it's buffers.

After the startup *TDAsacquire* goes into the end-run state. To read out data a run must be started (BEGRUN command). If *TDASacquire* receives begin-run command it reads the configuration file *TDASpar.cnf*, which contains all run options and a list of all sub-processors to incorporate into the acquisition system. Therefore all these parameters may change from run to run. If the begin-run command had the *dummy* option set, *TDASarchive* is informed not to store but do discard the data. In this case the run number will be 0 in the begin-run event. If however a data taking begin-run command was received, the run number in the file *TDASrun.cnf* is incremented. In this file there is also a log-message of every tape initialize and every data taking begin/end run sequence together with the run comment that was enered at begin run time. The run comment is expected in the file *TDASrun_comment.cnf.* A run may be ended with the ENDRUN command. A run may also be suspended without ending it (SUSPENDRUN command). In this case simply the common dead time is set inhibiting all triggers. This command is useful if during a run for example the beam tends to become unstable and it is currently unclear if this is a momentary or a serious problem. A suspended run may be resumed (RESUMERUN command) or ended (ENDRUN command). For the documentation of the *TDAS* internal commands (__BEGRUN, __ENDRUN) refre to *TDASarchive.*

All configuration and status files (*TDASrun_comment.cnf*, *TDASpar.cnf, TDASrun.cnf*) are expected at the path specified with the environment variable TDAScnfpath. Sample documented configuration files can be found in the *doc/cnf* subtree of the *TDAS* network disk.

OPTIONS:

-m=<num> readout mode (0: interrupt, 1: signal, 2: event).

-d=<num> debug level (default 0). <num> is expected as a decimal number. (refer to *TDASstart* for the definition of the various debug levels)

NOTE:

TDASacquire(_s) is typically not called directly but as a subprocess by *TDASstart*. *TDASacquire* needs the environment built by *TDASstart*.

Since the file *TDASrun.cnf* is updated every begin/end run and every tape initialize it may become lengthy during the time of an experiment. Only the first line of *TDASrun.cnf* is required for reading and updating the run number. The rest of the file is supplied for documentation purposes. It is therefore recommended to copy that file from time to time to a safe backup tree and to truncate it to it's first line. If that file does not exist it is created and the run number is reset to 1.

TDASarchive

Test

SYNTAX:

TDASarchive [<opts>]

ENVIRONMENT:

LOGPATH	/r0	generic path for writing the generic log/error files
TDASlogfile	tdas.log	name of the generic *TDAS* log file
TDASerrfile	tdas.err	name of the generic *TDAS* error file
TDAScnfpath	/n10/tdas/cnf	generic path for all *TDAS* configuration files

SUPPORTED *TDAS* ONLINE COMMANDS (used by *tdasmenu*):

INIT_TAPE	mount and initialize tape (needs as argument tape label)
UNLOAD_TAPE	write EOV header, rewind tape and anload it
SET_FILE_NAME	specify run file's name prefix on tape
SET_FILE_SIZE	specify maximum size of one run file in Megabytes
HELP	print help text on all or specified command
TERMINATE	terminate process after proper cleanup

FUNCTION:

TDAsarchive is responsible for putting the online data on a mass storage medium. If not started with the -n option *TDASarchive* tries at startup time to identify the kind of tape drive connected to the local SCSI bus at SCSI ID 5 (the SCSI target ID of the *TDAS* tape drive is specified at the *os9* device descriptor *exabyte0*). Currently supported devices are Exabyte EXB-8200 (low densitiy 8 mm video tape) EXB-8500 (high density 8 mm video tape) and EXB8500C (high density 8 mm video tape with hardware compression). There is no configuration for the specific tape device required except to set it's SCSI target ID to 5. If the tape device was found and successfully identified a potentially loaded volume is unloaded (unless -l option is specified) to prevent data tapes from accidentally being rewritten. After starting up *TDASarchive* begins to read the data from the cyclic *TDAS* buffer pool. If a begin run event is found and this run is not flagged as dummy run, a new tape file is opened. The file name (default *RUN_<run number>*.LMD) prefix (*RUN_*) may be changed with the online command *SET_FILE_NAME* at any time, the suffix containing the run number (4 digits) and the *.LMD* extension are however fixed. The physical format of the data on tape is conform to the ANSII standard [ANSItape]. These tapes can for example be directly mounted at any VAX and read with copy or dir.

The maximum capacity of a 8 mm high density video tape is about 4.5 GBytes. So in principle extremely large run data files could be produced, that are almost impossible to handle. Therefore a special feature was integrated into *TDASarchive* that prevents run data files from becoming larger than a ceratin limit (default about 150 MBytes). This limit can be changed any time with the command SET_FILE_SIZE to any value above 10 MBytes. If a tape file reaches that limit *TDASarchive* directly negotiates with *TDASacquire* to end the current run and to start a new tape run later with the same options (__BEGRUN, __ENDRUN commands).

Before a run data file can be written to tape a volume must be loaded, mounted and initialized. This is achieved by loading a 8 mm video cassette into the Exabyte drive and sending the INIT_TAPE command (use *tdasmenu*) to *TDASarchive*. If a begin run event is detected in the data stream with the tape flag on and there has not been a prior INIT_TAPE command *TDASarchive* cannot start writing data to tape until it receives the initialize command with the tape volume label as option. This status will result in a virtual *TDAS* crash since the *TDASacquire* will sooner or later be blocked by *TDASarchive* holding a lock on it's current buffer. This race condition can be easily resolved – even if there was another ENDRUN command sent to *TDASacquire* – by loading and initializing a tape volume with *tdasmenu.* If the end of the mounted volume is almost reached (physical size of the mounted volume minum 300 MB), the current run is ended and the tape is automatically unloaded. Loaded tapes can however be unloaded using the UNLOAD_TAPE command. If *TDASarchive* receives such an unload command and there is currently a run started, it first requests *TDASacquire* to end the run (__ENDRUN) and then unloads the tape.

OPTIONS:

-d=<num>	debug level (default 0). <num> is expected as a decimal number. (refer to *TDASstart* for the definition of the various debug levels)
-n	no data taking mode – no tape drive is connected.
-r=<rate>	data discard rate in kilo bytes per second (default 500)
-t	taping mode, send data to tape drive connected at SCSI ID 5
-l	don’t unload tape

NOTE:

TDASarchive is typically not called directly but as a subprocess by *TDASstart*. *TDASarchive* needs the environment built by *TDASstart*.

Since *TDASacquire* may – depending on the trigger rate and the CPU load – not immediately react on an internal __ENDRUN command and since there may be all *TDAS* buffer filled with valid data, the actually specified maximum data run file size (default 150 MB) is decremented by twice the total size of the *TDAS* buffers plus 1MB.

TDAScpudbg

debugger for handling *TDAS* sub-processors

SYNTAX:

TDAScpudbg [<opts>]

ENVIRONMENT:

TERM	vt220	currently used terminal type
PORT	terminal device	currently used terminal device descriptor
TDAScnfpath	/n10/tdas/cnf	generic path for all *TDAS* configuration files

FUNCTION:

TDAScpudbg is an interactive utility for maintaining *TDAS* sub-processor systems. It supports *DCL* like command line editing. At startup the configuration file *TDASpar.cnf* at the path specified with the environment variable TDAScnfpath is interpreted. In this file there is for every sub-system a specific mnemonic, which is used inside *TDAScpudbg* to identify that system. All internal *TDAS* commands like startup, begin-run, readout, end-run, shut-down can be sent to any or all of the sub-systems. If the command produced data it can be dumped. It is possible to select how many data words to display as well as if the data should the begin or the end (use negative word count) of the sub-event(s) found in the specific dual-ported-memory. There is online help available (type help). *TDAScpudbg* is especially useful if another sub-processor system is intended to be included into the *TDAS* acquisition. Using *TDAScpudbg* it is possible to first diagnose that (new) system (of course after editing the file *TDASpar.cnf*) for example with the command cyclestartup and then to put it into the startup state simply by sending the specific system the startup command. A end-run / begin-run sequence will then incorporate that new system into the *TDAS* data stream.

OPTIONS:

none

EXAMPLES:

TDAScpudbg <commandfile.inp

Call *TDAScpudbg* with it's standard input redirected to the file *commandfile.inp*. TDAScpudbg will execute all commands in that script file and then exit..

NOTE:

TDAScpudbg does not consider if there is *TDAS* running or not. Therefore it is possible to send commands to sub-systems of the running data acquisition system with surely fatal results. Be careful when using *TDAScpudbg*!

TDASmenu

interactive menu driven control program for *TDAS*

SYNTAX:

TDASmenu or tm

ENVIRONMENT:

TERM	vt220	currently used terminal type
LOGPATH	/r0	generic path for writing the generic log/error files
TDASlogfile	tdas.log	name of the generic *TDAS* log file
TDASerrfile	tdas.err	name of the generic *TDAS* error file
TDAScnfpath	/n10/tdas/cnf	generic path for all *TDAS* configuration files
TDASroot	/n5	root path for the whole *TDAS* software
TDASshare	/n5/cmds	path for *TDAS* generic programs
TDASexec	/n10/tdas/cmds	path for experiment specific programs
TDASarchiveopt	-n	options to pass to *TDASarchive*
TDASstartopt1	"-m=2"	options for *TDASstart*
TDASstartopt2	""	additional options for *TDASstart*
TDASstartopt3	""	additional options for *TDASstart*
TDASstartopt4	""	additional options for *TDASstart*
TDASstartopt5	""	additional options for *TDASstart*
TDASdebug	24	debug option for all *TDAS* processes
EVENTcrate	15	Crate number of *TDAS* CAMAC *system-crate*

FUNCTION:

TDASmenu (or *tm*) is the interactive menu driven interface program the the *TDAS* process thread. Any *TDAS* action can be controlled with this program (it is actually implemented as a shell script). The menu items may be selected using cursor keys and pressing <return> or by typing the key in front of the displayed option. *TDASmenu* has a special sub-menu for every of the main *TDAS* processes. There is the *TDASstart* sub-menu for starting and stopping the data acquisition. Here the environment variables TDASstartopt*, TDASdebug, TDASarchiveopt are required. Since there is a certain limitation on the maximal length of a environment variable that is unfortunately *shell* dependent, there are 5 similar environment variables TDASstartopt* available (typically the -e options tend to become lengty). The actual startup command will be a merge of all of these 5 environment variables. The whole *TDAS* startup command is created as a temporary shell script at the path */r0/tmp/.TDASstartcom* and is then executed from there. In the *TDASacquire* sub-menu runs may be started, ended, suspended and resumed. The *TDAS* general configuration file *TDASpar.cnf* can be edited there. The third sub-menu is for controlling *TDASarchive*. Here tapes can be loaded and initialized or unloaded. The *TDASarchive* options specifying the maximal tape file size and the tape file name prefix can be changed from their default here, too.

If during startup old *TDAS* log or error files are found or if the C-option is selected in the *TDASmenu* root menu, the *TDAS* log and error files are saved to the subtree *LOGSAVES* at the path specified with the environment variable TDAScnfpath. These copies have as an extention the current date and time of the copy command. Therefore all log and error messages should be available for later inspection.

OPTIONS:

none

EXAMPLES:

TDASmonitor

Call the interactive, menu driven *TDAS* control program.

NOTE:

TDASmenu is an interactive tool for controlling the data acquisition. It calls for every function a special shell script at the subtree *scmds/scmds* of the path specified with the environment variable *TDASroot*. These scripts can also be called directly for example with rsh for remotely controlling *TDAS*.

TDASmonitor

Test

SYNTAX:

TDASmonitor

ENVIRONMENT:

TERM	vt220	currently used terminal type
PORT	terminal device	currently used terminal device descriptor
LOGPATH:	/r0	path for writing log/error files

FUNCTION:

TDASmonitor is a monitor utility for the overallstatus of the data acquisition system. It mainly acquires it's information from the *TDAS* main data module *TDASmain_m*. The display screen is mainly self describing. However the displayed error counters (they are reset at a begin of a run) are worth to be noticed here.

First there is the total number of readout triggers that led to any kind of an error and second there is the total number of events where at least one of the sub-processor system signalled a readout error (synchronization timeouts to asynchronous sub-systems are excluded here).

The next error counter is very important: The *LAM during dead time* error counter signals that there is something wrong with the dead time handling. If a readout trigger is fired, it is latched at the LAM in (**LI**) flip-flop at the I/O-LAM box (chapter 2.1). *TDASacquire_s* resets that flip-flop as soon as possible in the interrupt handler routine (typically about 40µs after the trigger depending on the implementation and speed of the *event-builder* CPU). Before releasing the dead-time (busy flip-flop of the I/O-LAM box) *TDASacquire* checks if this **LI** flip-flop was set again in the mean time. If this condition is true then a trigger was fired during the dead time and the mentioned error counter is incremented.

TDASacquire reads the pattern unit in the CAMAC *system-crate* that defines the *readout-type* before it calls the various readout routines. If that pattern unit was not gated, *TDASacquire* will read 0. In this case no *readout-type* is defined and the *TDAS* global error counter *readout type 0* is incremented. Events with *readout-type* 0 are discarded.

There is a CAMAC scaler in the *TDAS system-crate* that is used to count the physical readout triggers. This counter is compared to the *TDAS* internal event counter. If a mismatch is found the error counter *scaler mismatch* is incremented. If the CAMAC scaler count is higher than the *TDAS* event counter (the typical case), the *TDAS* event counter is adsusted to the appropriate value, discarding the not matching event numbers. If the CAMAC scaler has a lower count than the internal *TDAS* event counter, the CAMAC scaler is incremented until the two counters are equal and the *TDAS* global error counter *CAMAC scaler adjustments* is incremented, too.

OPTIONS:

none

EXAMPLES:

TDASmonitor

Display the continuously current *TDAS* status.

NOTE:

TDASmonitor needs as a *TDAS* display program the *TDAS* environment to work. Therefore it will terminate immediately if the data acquisition is not running.

TDASmonitor uses *curses* for optimized screen handling. It can therefore be installed for any terminal type using the *termcap* environment.

TDASscaldisp

TDAS scaler display utility

SYNTAX:

test [<opts>]

ENVIRONMENT:

TERM	vt220	currently used terminal type
LOGPATH:	/r0	path for writing log/error files
TDAScnfpath	/n10/tdas/cnf	generic path for all *TDAS* configuration files
SCLDSP_cnffile	display.cnf	configuration file name

FUNCTION:

TDASscaldisp is a scaler display utility for *TDAS* online scaler data. It accesses the fetches the scaler data by linking to the data module *TDASscalbuf_m*. Since the scaler setup is highly experiment specific the user must make sure to correctly update that data module. However there is a C-utility function available that does most of that job. Typically the scalers are read out during the *spill-off* event. At the end of that readout there exists the scaler sub-event somewhere in the memory of the event-builder. At this stage of the readout the C-routine *(void)update_scaler_data_module(((int*)ptr, (int)channels)* can be used to update the scaler data module for later display of the most up to date scaler data. It's arguments are a pointer *ptr* to the begin of the scaler data in the *spill-off* sub-event, *channels* is the number of (32 bit) scaler words that shall be transferred. *update_scaler_data_module* does not only copy the scaler data to the data module but also produces in addition the minimal (different from 0) maximal and cumulative count per individual scaler channel. Therefore the size of the scaler data module (environment variable TDASscalsize) must be at least 4 times of *size*.

The kind of display is highly configurable. The display (there are 9 different displays available) is splitted into 23 lines (plus one head-line) with 4 colomns of 20 characters each. Any of these cells may be chosen to display any of the following kind of scalers: the current value, the minimal, maximal and the cumulative value. In addition the ratio of any type of any thwo channels can be specified (for example for displaying the dead-time). Every displayed scaler channel is identified with a specifiable 8 character long mnemonic. The various scaler channels are identified by their order relative offset (starting at 1) to the begin of the scaler data module (*(int*)ptr* during readout). How the various displays are defined is read from a configuration file at startup of *TDASscaldisp*. The configuration file *display.cnf* must be located at the tree specified with the environment variable TDAScnfpath. The file is self documenting. A sample *display.cnf* file can be found at the *doc/cnf* subtree of the *TDAS* disk */n5*.

OPTIONS:

none

EXAMPLES:

TDASacaldisp

Start the scaler display program *TDASscaldisp.*

NOTE:

TDASscaldisp needs as a *TDAS* scaler display program the *TDAS* environment to work. Therefore it will terminate immediately if the data acquisition is not running or if the scaler data module was not updated (at least once) prior to calling *TDASscaldisp.* This happens typically at test setups where no *spill-off* occur.

TDASmonitor uses *curses* for optimized screen handling. It can therefore be installed for any terminal type using the *termcap* environment.

TDASstart

program to start/stop the *TDAS* data acquisition

SYNTAX:

TDASstart [<opts>]

ENVIRONMENT:

LOGPATH	/r0	generic path for writing the generic log/error files
TDASprcpath	/r0	path for the stdout/stderr log files of all *TDAS* subprocesses
TDASlogfile	tdas.log	name of the generic *TDAS* log file
TDASerrfile	tdas.err	name of the generic *TDAS* error file
TDAScnfpath	/n10/tdas/cnf	generic path for all *TDAS* configuration files
TDASparsize	40000	maximum size in bytes of the *TDAS* parameter event
TDASscalsize	4000	maximum number of scaler channels (long words) to pass to *TDASscaldisp*
TDASerrdatsize	100	maximum number of header words to save of error events
TDASerrtrlsize	100	maximum number of trailor words to save of error events

SUPPORTED *TDAS* ONLINE COMMANDS (used by *tdasmenu*):

HELP	print help text on all or specified command
TERMINATE	terminate process after proper cleanup

FUNCTION:

TDASstart produces the whole *TDAS* environment and starts all *TDAS* subprocesses.

First of all the generic log- and error files are created. Potentially existing older log files are discarded.

In the next step all *TDAS* data modules are created and initialized. Data modules are a special os9 feature. They are mainly a system wide named common block. Any process may link to it and depending on its protection read/write it. If the main data module *TDASmain_m* exists it is assumed that *TDASstart* was run earlier with the `-c` option. In that case all data modules will be kept in memory at shut down time. (The default is to remove (unlink) all data modules from the os9 memory to make the space available for possibly other memory consuming applications.)

If all data modules could be created without error the required os9 *events* are created. These os9 *events* are used as lock mechanism between the various *TDAS* subprocesses.

At this stage of the startup the *TDAS* environment is completely created. If there are *TDAS* subprocesses remaining from an earlier run, they are stopped at this time (except *TDASaccessd*). Then *TDASstart* forks all *TDAS* subprocesses including *TDASaccessd* if it is not already running. It passes to all subprocesses the debug level, which was passed as command line option (-d). The debug level is interpreted as a bit field, where every bit enables a certain class of logmessages:

bit value	**debug log message action**
2	stamp any *TDAS* buffer state transition of *TDASacquire* and *TDAsarchive* to the log file.
4	log any signal that is recept.
8	log any command that is recept.
16	log all state transitions.
32	disable timeouts at the communication with *TDAS* subprocessors. This flag is useful when debugging a readout program on a subprocessor and prevents *TDAS* from timing out during the possibly long debug phase.
1024	this proces is run in the context of a debugger and not in the context of *TDASstart*. (*TDASstart* was run with -f option)

TDAsaccessd is forked with its standard (error)output redirected to the */nil* device. The second process that is started is *TDASarchive*. *TDASarchive* receives as additional command line options the options that were passed to *TDASstart* with the -p option. The standard (error)output (*stderr*, *stdout*) of *TDASarchive* is redirected to the file *TDAsarchive.log* at the path specified with the environment variable TDASprcpath. The next step is to fork *TDAsacquire* similar to *TDASarchive*. The readout mode (-m option) is passed as additional command line option. Readout mode 2 (event) is strongly recommended. The standard (error)output of *TDASacquire* is correspondingly redirected to the file *TDAsacquire.log*. *TDASstart* redirects it's own standard (error)output to the file *TDASstart.log*. All these process specific log files are not used by

TDAS and should be empty. However there are operating system routines that produce error messages (like the trace back) at the standard error or standard output. These messages will deposit in the process specific log files.

After all subprocesses were started *TDASstart* waits for all of them to reach their startup state. This is achieved by negotiating with the data module TDASmain_m.

If all *TDAS* subprocesses reached the startup state *TDASstart* sends a startup command to *TDASaccessd* enabling remote connection requests to access online data.

At this stage the startup is complete and *TDASstart* sends the success message to the result pipe /pipe/startup_result. If during the startup sequence an error occurs an appropriate message is written to the *TDAS* error file and an error status is sent to the result pipe.

The last duty of *TDASstart* is to handle potential shutdwon commands (TERMINATE) and to react on fatal errors of one of the child processes. If a terminate command is received *TDASstart* ends a run if required first. Then the terminate command is passed to all child processes (including *TDASaccessd*) and *TDASstart* waits for the completion of all child processes (excluding *TDASaccessd*). Finally all *TDASmonitor* and *TDASscaldisp* processes are terminated. At this stage of shutting down the data acquisition there is no process left, that would potentially use the *TDAS* environment. So *TDASstart* removes all *events* and later all data modules (if it was not run earlier with -c option). If a child process dies a fatal error is considered and *TDASstart* handles this condition equivalent to a terminate command.

OPTIONS:

-m=<num> readout mode (0: interrupt, 1: signal, 2: event). This option is passed to *TDASacquire.*

-d=<num> debug level (default 0). <num> is expected as a decimal number.

-n=<num> number of internal data buffer data modules to use (default 3).

-s=<num> size in bytes of each internal data buffer (default 100000).

-e=<a>:<s> base address <a> and size <s> (both hexadecimal) of external buffer. This option is useful to use an external memory (outside the scope of os9) for one or several *TDAS* buffers

-p="str" special options to pass to *TDASarchive*

-f don't fork subprocesses. Fork shell instead for debugging of *TDAS* subprocesses inside the *TDASstart* context.

-c create all os9 data modules and exit. With this option all TDAS data modules may be created. *TDASstart* is run with this option typically at startup time to overcome possible problems due to os9 memory fragmentation at a later time.

EXAMPLES:

```
TDASstart -d=24 -n=4 -s=1000000 -m=2 -p="-t" &
```

Start the data acquisition with debug option 24 (refer to page 85); use 4 os9 internal data buffers (they will be created as data modules) each having 1MB size; perform readout in event mode and run the archiving process *TDASarchive* in SCSI tape mode.

```
TDASstart -d=24 -n=4 -s=100000 -f
echo "helpresult help" >+/pipe/TDASacquire_p
srcdbg TDASacquire -m=2 -d=24 </pipe/TDASacquire_p >>-/r0/acquire.log
```

The above two command lines are usefule for source level debugging of *TDAS* sub processes like *TDASacquire*. The -f option of *TDASstart* prevents it from starting all subprocesses but requests to fork a shell. The secone command line is executed by that shell. It produces the pipe that is required as command input later. The command passed is a help command that is supported by all *TDAS* sub processes. The result of the help command will be placed into the file */pipe/helpresult* The thirt line calles the source level debugger. The standard input redirection to the pipe is important since all

TDAS sub processes install signals on their standard input, which causes problems if the debugger and the debugged program itself use the same input device. Additional commands can be sent to that pipe using the echo command or the *TDASmenu* utility.

NOTE:

TDASstart is typically not called directly to start/terminate the data acquisition but using the utility *tdasmenu*.

If *TDASstart* has created all *TDAS* data modules for later use (-c option) to prevent possible memory fragmentation problems, later changes to the environment that specify the size of these data buffers (TDASparsize, TDASscalsize, TDASerrdatsize, TDASerrtrlsize or -s option) will not be recognized. Unlink the main data module *TDASmain_m* or reboot the computer (use *shutdown*) to put the new environment into affect.

If an error occurres during the very first steps of *TDASstart* before the log file handling is set up (for example if the environment variables TDASlogfile or TDASerrfile cannot be found) the error and log messages may not be in the specified files at the specified paths. *TDASstart* will then place the error and log message in the files *TDASstart.err/TDASstart.log* at the path specified by the environment variable LOGPATH. IF this fails, too the error messages will be sent to *stdout/stderr*.

A Glossary

Throughout this manual some technical verbs and names are used, which will be described briefly in the following. For details please refer to the references cited here in . However you will find some hints and tricks here which we found out during the years of *TDAS* use.

AEB, EBI:

The **A**leph-**E**vent-**B**uilder is a 20 MHz 68020 Fastbus module. It was originally designed for the ALEPH experiment at Cern (hence the name). The AEB has typically 1 or 4 MB memory and runs OS9. It has 68k Fastbus coprocessor onboard that supplies assembly language Fastbus calls. An appropriate Fastbus preprocessor package is supplied, which transforms the Fastbus mnemonics into define constant statements, which then run through the r68020 Microware assembler. There is also a library supplied (/n0/lib/fb.l) which contains all Fastbus-Standard-Routines (ANSI/IEE Standard 960-1986) [AEB, EBI].

The **E**vent-**B**uilder-**I**nterface is a module, which was designed to link the AEB to the standard GSI interface bus δVSB. It is another Fastbus module, which is mounted next to the AEB (left or right side). It is connected to the AEB at the front panel with a special 2x96 pin female connector. The EBI uses only the power pins from the Fastbus backplane. The second EBI connector is the differential VSB bus input. Therefore the only way to mount the EBI into the δVSB daisy chain is to use of the CAT - the δVSB T-piece.

CAMAC:

The CAMAC bus is a 24 bit I/O bus. A variety of special and general purpose modules are available. However it is a slow bus. A typical read cycle takes between 2 and 3 µs. Every module is individually addressed by a station select line called N. Every station has its own LAM which is handled in the CAMAC crate controller depending on its type. There are three global signals at the CAMAC backplane the user should be aware of - namely the *initialize, clear* and *inhibit*. The *initialize* signals all modules in the crate to perform a general reset function. The *clear* forces the modules to clear themselves without resetting their setup. The *inhibit* signal is the most dangerous since it is a toggle switch. It may be asserted for an arbitrarily long time. As long as *inhibit* is asserted all modules in the crate will not react to gates, triggers etc. - they are inhibited! Some of our crate controllers CAV,CVB assert the CAMAC *inhibit* after (power up) reset.

A CAMAC module is addressed with the Branch and Crate number (**B, C**) of the CAMAC Crate and the station number (**N**) inside the crate. The appropriate function (read, write, clear, self test etc.) is selected with a function code (**F**) and a subaddress (**A**). For easy use these CAMAC functions can be performed in C with a C-Preprocessor macro called CAMAC (B,C,N,A,F) or at OS9 with the command CAMAC (B,C,N,A,F). There is a variety of shell scripts available to handle the various modules. For details refer to chapter 4. Any CAMAC cycle finishes with two state signals **X** and **Q**, which are supplied by the selected module. X signals that the requested command was accepted and Q signals that there was valid data returned or accepted. For example a module that was cleared but not gated would signal a read-out command with X but no Q.

Cable Card:

Some δVSB devices (EBI, some CAV) have only one δVSB connector. To mount them inside a branch a δVSB T-piece is required – the Cable Card. But it is extraordinary fragile. I recommend not to use it whenever possible. In case of the EBI I built an aluminum support frame to fix the Cable Card to the EBI module. Otherwise it tends to fall off causing a hell of a lot of funny errors on the whole δVSB branch!

CAT:

The CAT module is a Fastbus module for controlling and testing of the LeCroy TDCs and QDCs. There is one CAT module recommended per Fastbus crate. The *pedestal* program does not work without a CAT module.

There is one important point which should be noted here. There are several general purpose signals on the Fastbus backplane, which LeCroy uses to control their modules. In particular there is the QDC gate, TDC arm, TDC common stop and general clear. Unfortunately the logical signals QDC gate and the TDC arm are assigned to the same physical backplane signal. In addition the TDCs that have (as the QDCs) all these signals in addition as individual front panel inputs cannot be programmed (like the QDCs) to select either the front panel or the backplane signals. They logically or both of them. Therefore the QDC gate of the CAT module, which is broad casted to that signal at the back plane, can only be used as the TDC arm signal. The QDCs have to have their gates supplied individually at their front panel input.

CAV, CBV, CVC:

The CAV and CBV are different (incompatible) versions of δVSB CAMAC crate controllers. Both devices map a LAM from station 1 to the δVSB interrupt line (the LAM signals from all other stations are ignored). However the CAV modules cannot disable the δVSB IRQ (**I**nterrupt **R**e**Q**uest). Therefore a broken CAMAC module in station 1 which pulls the LAM would certainly crash the data acquisition system. I therefore disabled the δVSB interrupt output of all ALADIN CAV modules. These devices have a little tag on their front panel which identifies that feature. I strongly recommend only to use CAVs with that option. The very first series of CAV modules have with respect to all later versions of the CAV/CBV the X and Q signals inverted. All ALADIN, LAND and 4π modules of that kind were modified to match the newer X, Q standard. However the CAMAC crate controllers of the TAPS collaboration are mostly of the incompatible, unmodified type. As long as X and Q are not tested during read-out (which is the default) this does not matter - however one should be aware of that fact. It is easy to find out what kind of X, Q response a CAV has. Read a module that has been cleared with the OS9 CAMAC command. If the response is X, no Q you got the right one - if the (impossible) response is no X, Q you better get another CAV. As a matter of fact the CBVs work more stably than the CAVs. If possible I would try to avoid using the CAVs in critical crates. Software handling of the CAV/CBV is described in chapter 4 of this manual.

The CVC is an intelligent CAMAC crate controller with a 68030 CPU and an optional signal processor onboard. It has also ethernet and SCSI interfaces. It is therefore conceivable to run a *TDAS* directly on the CVC as a single crate/branch data acquisition system. There is already an appropriate *TDAS* port available.

E5,E6,E7:

These VME processors build a family like the Motorolla 68k CPUs. The E5 has a 68020, the E6 a 68030 and the E7 a 68040 CPU. The main difference between the 68020 and the 68030 is the PMMU (**P**aged **M**emory **M**anagement **U**nit) being on chip of the 68030 and 68040. The MMU is of great importance since it supports a memory protection, which inhibits processes with for example overrunning pointers from destructing other processes or the operating system's global memory and thus crashing the CPU. All three VME boards have essentially the same address maps. They are all equipped with VME master/slave, VSB master, SCSI, ethernet and serial/parallel i/o. The E6 and E7 have a VSB slave interface in addition. NOTE: all these CPUs need the VME 16MHz system clock for the correct operation of their VSB interface gate array. So if there is no system controller in the crate then the VSB interface will fail! [E5, E6, E7]

ECL signals:

The trigger electronics uses typically two signal standards NIM and ECL signals. ECL signals are negative differential open drain current signals. Typical ECL cables are twisted pair flat cables. They cannot directly be monitored with an oszilloscope. However there are converters available which convert ECL into NIM signals and vice versa.

ESONE CAMAC standard library:

To have the opportunity of standard CAMAC software which is independent of the controller and host hardware there has been a standard defined. For details refer to chapter 4 of this manual. This library is available on the OS9 front-end computers as well as on all our host computers like VAXen, Ultrix and AIX workstations. It has a C and FORTRAN binding. If a CAMAC controlling program is run at one of these host computer all CAMAC calls are virtually routed through ethernet to the appropriate front end computer E5,6,7, where they are handled by a special ESONE server process *eso_ser*. After processing the commands *eso_ser* sends the result(s) back to the host computer. There are commands available which inhibit any CAMAC (remote or local) accesses outside *TDAS*. These locks are useful to inhibit unintended changes to the setup during the data taking phase.

Exabyte:

Exabyte is the name of the company that produces the SCSI 8mm tape drives which we use for data taking. There are essentially two versions available: a low density version (EXB 8200) with maximum 2 GB per volume and 250kB/sec data transfer rate and a high density version (EXB 8500) with maximum 4.5 GB per volume and 500 kB/sec data transfer rate. The tape itself is a standard 8mm video tape. Since the read/write head is a video head, the tape load is very slow. It takes about half a minute to load the tape and write the appropriate begin of tape file marks.

FastBus:

The Fastbus is a 32 bit mixed address and data bus. All CAVE B experiments have almost exclusively LeCroy TDC and QDC modules in their Fastbus setups. Therefore a standard read-out program *fastbus_readout* was supplied. For maintenance purposes there is a program *fbrw* which allows to perform single Fastbus cycles. The program *pedestal* is supplied for determining the pedestal of the Fastbus QDCs and for testing of the QDC and TDC modules. They have unfortunately a high failure rate. For details of all Fastbus related programs refer to chapter 4 of this manual.

Internet Name Convention:

All front-end computers can be accessed via Internet. They have a common name convention. Any Internet name consists of exactly 6 letters. The first two letters specify the CPU type, the next three letters specify the group that owes that CPU and the last letter is an alphabetic running index. As CPU types are the following combinations valid: *eb* (=AEB), *e5*, *e6*, *e7*. As group identifiers there are *kp3* for KP3, *lnd* for LAND, *4pi* for 4π, *elx* for the GSI "Elektronik und Experimente" division, *kos* for the KAOS setup, and *frs* for the fragment separator. So *e6kp3b* is the second E6 of KP3.

I/O LAM box:

The I/O LAM box is a general purpose CAMAC module. It has 4 general purpose NIM inputs, outputs and pulse outputs. In addition there is a special input signal which if pulsed sets the LAM of that module, which is used as read-out trigger of the data acquisition system. Parallel to setting the LAM another flip flop is set. The output of that flip flop is the *TDAS* common dead time. It is reset when the read-out finishes.

LAM:

The **L**ook **A**t **M**e signal which may be individually set by any CAMAC station was intended as a handle for a fast compressed read-out. Most of the CAMAC modules may be configured to set the LAM if they have at least one valid data word. Therefore a LAM scan would give then the list of stations to read out.

MPI:

If the trigger logic uses fast clears then there is typically a fast clear acceptance time (fast clear gate) defined within which the fast clear must occur. If the fast clear acceptance time expires without a fast clear the trigger is forwarded to the data acquisition system. The LeCroy Fastbus modules cannot be quickly cleared if they are in conversion mode. Therefore there is the so called **M**easure **P**ause **I**nterval which inhibits the Fastbus modules from starting the conversion during that time. It should be setup such that it does not expire before the fast clear gate. However if the LeCroy QDCs and TDCs are cleared during the conversion phase it takes about 5µs to clear compared to 500ns during the MPI. If the fast clear settling time is greater than 5 µs then the actual setting of the MPI does not matter.

NIM signals:

The trigger electronics uses typically two signal standards NIM and ECL signals. NIM signals are negative voltage signals. NIM cables use LEMO connectors. There are converters available which convert ECL into NIM signals and vice versa.

Read-out Type:

There is a variety of different kind of triggers in a data acquisition system. For example normal physics interaction triggers, base line / pedestal triggers, pulser, laser triggers, spill off / scaler trigger etc. For a given trigger not everything has to be read-out or special handling is required. For example in the spill off event the spill scaler must be reset. Therefore 15 different *read-out types* were made available. One of them is predefined; *read-out type* 15 is the spill off event. It has a special status since typically all spill scalers are read and cleared then. In addition, some other *TDAS* internal state transitions (refer to chapter 4 of this manual) are triggered.

SCSI:

The **S**mall **C**omputer **S**ystem **I**nterface is a popular bus for large storage devices like hard discs and tapes. Almost any workstation on the market uses SCSI for interfacing with their hard discs and tapes. The bus itself is comprised of a 50 wire flat cable, that is up to 10m long. The tape of the data acquisition system is directly connected to the front end computer via SCSI. NOTE: within the SCSI specification there is one 5V power line defined to be at the cable. Any neighboring wire has ground potential. So a wrong mounting of the SCSI cable may cause the cable to burn! [SCSI1,2]

Triggerbus:

In a real multiprocessor environment (*TDAS* is a strict master/slave environment) all front-end processors must be individually triggered for read-out. The trigger bus supplies these front end processors with a trigger read-out type and a 4 bit trigger index which is compared to the lower 4 bits of the event number. It also produces the common dead time by ORing all busys from all slaves. Any slave at the trigger bus may set a special so called subevent invalid signal which gains the opportunity of having an intelligent second level veto trigger. All features of the trigger bus are software programmable. Since *TDAS* is a master/slave kind of multiprocessor system in principle no trigger bus is required and since it was not available during the design phase the I/O LAM CAMAC module was used instead.

VME:

The VME bus was originally designed to be a multiprocessor bus for the Motorolla 68000. However in the mean time there have been several upgrades which made it a general purpose 32 bit address/data bus. Unfortunately as a consequence there are still modules on the market, which do not obey the latest revision of the VME bus and therefore cause a lot of trouble even if they are simply mounted in the VME crate without being selected. There is a VME controller chip onboard the E6 and E7, which is pretty popular and almost bug free. In contrast the CES FIC product is one of the other type. If such a device has to be used careful (memory)tests are strongly recommended!

VME supports three addressing schemes the short (A16) standard (A24) and the extended (A32) addressing mode. The short addressing mode supports only an address range of 64 kBytes and is used only for special devices. The standard (A24) addressing mode, which supports an address window of only 16MB (NOTE: there are CPUs and memories available with 16MB and more capacity) should not be used, too. However there are some (typically old CES products) modules around, which support only the VME standard addressing mode. Any VME device should be addressed in the A32 mode if at all possible. What addressing mode (and bus) is used during a certain bus cycle depends on the address and the address map of the given CPU [E5, E6, E7].

Another boring topic concerning VME is the bus arbitration. A typical VME crate does not necessarily have such a strongly required bus arbiter module. Therefore most of the VME CPUs have onboard VME arbiters of different quality. It is strongly recommended not to use the VME arbiter on the E5 in particular. On the other hand, exactly one CPU per crate (the one in slot 1) may have the VME arbiter option selected. This causes the disadvantage that identical VME devices have to be jumpered differently. Especially during beam times if things fail this causes serious trouble since people typically swap boards without considering this constraint. I recommend the following alternative. There are cheap and reliable VME arbitration modules available (price ca. DM 100) which may be mounted at the back side of the VME backplane at slot 1. Most of the crates have already been modified appropriately. However there are still crates without an arbiter around.

The VME bus has an arbitration daisy chain. Empty slots have to have that daisy chain signals (busgrant in/out number 0 to 3 and interrupt acknowledge) short cut. Therefore every VME backplane has a series of pins nearby to mount the appropriate jumpers if the slot is empty. Typically errors occur if an empty slot does not have these jumpers mounted or if there is a board mounted in a slot with the jumpers set. Things are unfortunately even more complicated. VME modules without a VME master interface like memory modules typically do not bridge the daisy chain signals themselves. This means that a VME slot that is occupied with such a memory module must have the daisy chain jumpers set! [VME]

VSB:

This bus was designed as a **V**ME **S**ubsystem **B**us. A typical VSB application is a series of VME CPUs each having an additional private VSB extension memory. Each CPU may then communicate to its' memory through its private VSB bus without occupying the common VME bus. The same idea was the reason for using it at GSI. Here we have VME CPUs in a VME crate where each CPU has its own subsystems connected at its VSB interface. It is therefore possible that all these processors which we call **F**ront **E**nd **P**rocessors (FEPs) read out their associated hardware in parallel. Only the final event building is done by the **E**vent **B**uilder (EB) another VME processor through VME. [VSB]

δVSB:

Since the VSB bus is a maximum 8 slot bus it cannot directly be used as an extension bus to connect remote hardware. Therefore an interface was designed at GSI by Jan Hofmann, which essentially translates the TTL VSB signals to differential signals (therefore the δ) with the advantage that up to 50 m cable may be between the first and the last δVSB device. The price due to the limited number of wires in a cable (differential signals have two wires per logical signal) is that this δVSB bus is a single master 32 bit (only) data bus. So the VSB arbitration, space and size signals are not available within the δVSB bus. NOTE: the δVSB cables are extremely fragile. They tend to produce intermittent contacts. Since the VSB bus is a multiplexed address/data bus the result is not only wrong data but also wrong addresses during δVSB transfers. So extreme caution and excessive testing is the only way to produce a stable data acquisition system.

VSC:

The VSC is an interface card which translates the VSB signals into δVSB signals. It is mounted at the back side of the VME P2 backplane directly to the appropriate CPU. On the other side of the VSC the δVSB cable is connected. Make sure that not more than 50 m (total length) of δVSB cables are connected. And that the δVSB is terminated exactly at both ends. The VSC maps only a sub part ($04000000, 64MB) of the full VSB address space to the δVSB interface (VSC windows). All δVSB devices (CAV, CBV, CVC, EBI) only decode the address bits within that window. It is therefore possible to have several δVSB branches per VME CPU. There are 2 slot VSB extension cards which are mounted between the VSC and VME P2 backplane. Then two VSCs may be connected to the same CPU. In such a setup the VSB terminators must be removed from the second VSC. However not all VSCs have their VSB (not δVSB terminators) on sockets. The VSC has 6 windows defined. However the first four are useless within the *TDAS* setup. Therefore the standard VSC window is 5 (VSB base address $F0000000). It is considered as VSB branch number 0. The VSB branch number 1 maps via VSC window 6 to the VSB base address $F4000000. There is another jumper on the VSC which is worth to be noted here, the slave select jumper. If the VSC is connected to a CPU – a VSB master – then the VSC must be jumpered to be a VSB slave. If however the VSC is connected to a VSB memory – a typical VSB slave – it must be jumpered to be a master on the VSB side. The δVSB interface is configured correspondingly.

VST:

The δVSB terminators VST are available in two versions. One version can be mounted directly on the VSC, CAV, CBV, CVC, EBI. The other version has a δVSB connector and may be plugged into the input of any δVSB device with two δVSB connectors (CAV, CBV). I recommend to use these since the directly mounted terminators are usually forgotten on the CAMAC crate controllers with the result of an illegal terminated δVSB branch.

B Bibiography

[AEB] ALEPH Event Builder
Dr. B. Struck user manual
BUGGY - A Debugger for the Aleph Event Builder
Dr. B. Struck user manual

[ANSITape] IBM
OS/VS Tape Labels Vs1 Release 3, VS2 Release 2
Edition GC26-3795-2

[E5] Eltec Eurocom E6 User manual
Eltec, Mainz

[E6] Eltec Eurocom E6 User manual
Eltec, Mainz

[E7] Eltec Eurocom E6 User manual
Eltec, Mainz

[EBI] Fastbus δVSB interface
GSI Manuals

[EXABYTE] EXABYTE 8200/8500 user manual
Exabyte Corporatione

[FASTBUS] Fastbus
A modular high speed data acquisition system for high energy physics
and other applications
ESONE committee ESONE/FB/01
FASTBUS stadndard routines
U.S. NIM Committee DOE/ER-0367
Aleph Event Builder FASTBUS library (A. Castro, A. Miotto Cern)

[HOFMANN] J.Hofmann
VSC VSB to differential VSB (δVSB) controller,
CAV1000 Camac crate controller,
EBI Fastbus δVSB interface
GSI Manuals

[M68K] M68020/30 32-Bit Microprocessor User’s Manual
Motorola

[OS9] OS9 operating system manuals
Microware Systems Corporation

[SCSI1,2] SCSI
American national Standard for Information Systems
Small Computer System Interface
ANSI X3.131-1986

[TDAS90] Th. Blaich, R. Freifelder, N.Herrmann, J.G. Keller, P.Koczon, M.Krämer,
Y. Leifels, V. Lindenstruth, W.F.J. Müller, R. Schmidt, K. Teh
GSI Scientific Report, p. 311, Darmstadt, 1990

[VME] The VMEbus specification, IEE1014-87
VMEbus International Trade Association

[VSB] The parallel sub system bus of the IEC 821 BUS
VMEbus International Trade Association