TDAS OS-9 ISKP
User's Guide

Version 2.4

Florian Schwandt

May 12, 1994

Contents

What's New ?

The predecessor of this document (TDAS OS-9 ISKP Programmer's Guide) is definitely out of date. Furthermore, the information presented there was sometimes to detailed, sometimes to roughly sketched and not always useful. This document is an approach to

- reflect changes since the first version of the manual (knowing that all attempts are in vain to produce a documentation which corresponds your current software) and
- focus more on the "user" side of TDAS. TDAS internals are not relevant when people try to use this as an introduction.

Note, that the older document is *not* updated. It is still available for comparison or details (which may not reflect the actual state of the software). Many things have been taken from the old document, however, with sometimes only slight modifications. New features described here (among others) are

- the entry point for user code has changed a lot,
- multiple trigger sources can be handled independently,
- tapes can be mounted in append mode,
- the directory structure has changed a bit.

New features which are not fully discussed here, but in different documents include

- a variety of *control processes*, providing to read, set, monitor and archive-to-tape parameters like high-voltage systems, discriminator thresholds and programmable delays [die93].
- an interface for writing host-data to the TDAS tape via TCP/IP [die94]
- a number of host-based programs for online data display and analysis [gas93, sch94].

You're welcome to send comments, corrections and improvements to `schwandt@merlin.iskp.uni-bonn.de`.

Chapter 1

Introduction

This chapter will give you an introduction to the TDAS hard- and software. After a short look on TDAS's history and features, some references for further reading are recommended. Then, hardware requirements are discussed, and the basic mechanism of data readout is presented.

1.1 Overview

TDAS (**T**emporary **D**ata **A**cquisition **S**ystem) is a data acquisition software designed for acquiring, archiving (and viewing) experimental data for small to medium-sized experiments in modern physics.

It is able to handle most standards in "front-end"-electronics e.g. CAMAC, Fastbus or VME. Raw data are stored on a tape unit connected to the local SCSI bus, e.g. an EXAbyte 8500 tape, which is capable of archiving approximately 5 GByte of data with a transfer rate up to 500 KByte/sec. Data can be transmitted "online" via ethernet to a host computer for event display, experiment control and analysis. Host software for this task is not discussed in this document.

TDAS was originally developed at the GSI (Gesellschaft für Schwerionenforschung) Darmstadt, where it was aimed to provide data acquisition for many experimental groups at different facilities. It was called "temporary", because it should provide a possibility for measurements until the data acquisition system which was planned in the first place and was in preparation was ready to work. A description of the original TDAS version can be found in [lin89]. The current software version now in use at the ISKP Bonn has been modified in several respects. The original structure, however, has been preserved, and so the name "TDAS" remains.

This manual is aimed at the experimentalist who wants to tailor the TDAS software according to his or her needs and use it for a special experiment. Note that this manual refers *only* to the original TDAS ISKP version (called *master version* hereafter), which is used as a point of departure for your specific experiment. In contrast to the GSI version, where TDAS was maintained centrally for all experiments, the ISKP version serves as a framework where new functionality can be added. The programmer is then responsible for all changes and additions. New master versions with enhanced functionality are constantly evolving.

To use TDAS for your experiment, you have to *program* the required readouts into TDAS. This approach was chosen in contrast to other, existing TDAS versions, which provide user CAMAC readout via ASCII-files at runtime and therefore require no programming effort.

The reason for this is, that especially for modern front-end electronic modules with e.g. zero-suppression or compressed readout facilities, the readout cannot be handled by simple, sequential lists of CAMAC commands. Instead, they require logic decisions and full arithmetic to be performed at runtime. This prohibits the use of plain sequential lists, but requires a lot of features from a programming language like e.g. "C". This is particularly true when TDAS ISKP is to be applied to COSY experiments. It was decided to do *all* CAMAC readout via programming and avoid a hybrid design (both CAMAC lists *and* programmed CAMAC readout).

The programming abilities required for implementing CAMAC readout, however, are few, and with a short introduction, you should be able to do every readout you want.

Note, that the information contained here is by no means exhaustive. For questions concerning TDAS directly, its source code is always the first instance of information!

Note further, that the ISKP master version no longer contains explicit support for FAST-BUS and/or VME readout, although the principal way of interfacing to these standards still exists. Probably some more work is needed to (re)implement them in the current version.

1.2 What You Should Know - References

Since you want to adapt TDAS by means of programming, you should be familiar with the terms and concepts of programming in general. This implies a principle understanding of computer architecture, the basic concepts of CPUs, memory, I/O, filesystems, networks, compiling and linking programs and so on.

If you just want to do some data readout from one or several CAMAC crates, you need not bother with TDAS's structure. A basic knowledge of the "C" programming language as well as some understanding of CAMAC should be sufficient for fitting TDAS to your needs. If you want to "dig deeper" into TDAS internals, you should equip yourself with knowledge about the following topics (else you can skip the rest of this section):

TDAS is written mainly in C with some hardware dependent parts written in 68K assembly language. You should therefore be familiar with C. If not, you should probably first read [ker77] and do some smaller programming exercises in C before changing TDAS sourcecode. Note, that due to the available compiler, TDAS is written in the older, so called K+R C-"standard", which lacks many features of the modern ANSI-C standard and makes programming more exciting. [mic84] describes all features of the OS-9 C-compiler and assembler.

You will most likely not need to change program parts in assembly, so knowledge of 68K assembly language is not essential, but may prove helpful. If you already know some other assembly language, 68K will be easy to learn. Refer to [hil84] for a complete description.

TDAS runs under OS-9, a realtime, (multiuser) and multitasking operating system. Since TDAS uses many features of OS-9, you should be acquainted to the concepts of processes, I/O redirection, pipes, (memory) modules, events, signals, files and directories. See the glossary in this manual for a short help, a full description of needed terms is found in [mic87]. For details concerning the ISKP installation of OS-9 consult [sch92].

TDAS's user interface is written as a *shell-script* for the so-called Z-shell (zsh). Changes in the user-interface may require a Zshell-manual [zel88].

For details on VME and related concepts, you should read [pet89]. Also look at the glossary in this manual. Details about the E6 microcomputer can be found in [elt91].

Communication with a host computer (e.g. a UNIX workstation) is done by means of the

TCP/IP protocol with the *socket* programming interface. You will find a small introduction in [mic89], consult [com91] for more detailed information.

As the main source for experimental data considered here is CAMAC, you should know the principles of CAMAC programming. See e.g. [lec90, Appendix A] for a brief introduction.

If you want to change the mass storage medium, you may need to alter the SCSI interface routines in TDAS. Refer to your mass storage manual for special requirements. [fuj91] gives the basics of SCSI interface.

Interfacing to front-end electronics is done via VSB-VDB (see below). You should have [str89], [str89a] and perhaps [str89b] at hand.

1.3 Hardware Requirements

In the following, needed hardware components are listed and described. Don't worry about the confusing abbreviations (especially those related to VME). See the glossary in this manual for a quick help. The OS-9 part of the TDAS software runs on an *ELTEC EUROCOM-6 CPU* Board manufactured by ELTEC elektronik mainz. It consists of a Motorola 68030 CPU, with interfaces to VMEbus, VSB, ethernet and SCSI. It is a standard VME CPU module with up to 8 or 16 Mbyte of quad-ported RAM (VMEbus, VSB and CPU). For further details, refer to [elt91]. Connection to front-end electronics is done via the VSC (VSB to differential bus cable) over distances up to 50m to either

- one or more dedicated CAMAC controllers (CBV, CVI),
- opt. EBI interfaces (Aleph event builder to VDB interface),
- opt. remote VME crates via other VSC interfaces there.

Data storage is done on a cartridge tape unit (EXB-8500) connected to the E6 SCSI interface. If no host computer communication is desired, no ethernet interface is required.

So for a minimal TDAS configuration, you need at least

- an arbitrary VME crate with an ELTEC E6 CPU board,
- a VSC (VSB to differential cable) interface connected to the backplane of the VME crate (Struck),
- a VDB cable (a screened 80 x twisted pair cable with two 96-pin connectors on both sides) connected to the VSC (Struck),
- a standard CAMAC crate containing a CBV CAMAC controller with a DMS dataway display (Struck) connected to the VDB cable and
- a CAMAC module in slot no. 1 of the CAMAC crate, which is able to provide a *LAM* (Look At Me) signal whenever a physical event occurs. In the following, this module is called the *IOLAM-box* (see below).
- An EXABYTE 8500 tape unit connected to the E6 SCSI bus.

1.4 The IOLAM-Box

For the task of translating a trigger input pulse (NIM) to a LAM (look-at-me) signal in the CAMAC standard, usually a special module is needed. Its functions are simple, but convenient, since almost all experiments need to react on trigger signals instead of doing readouts in fixed time-intervals or "polling" the front-end modules.

The minimal functionality needed for TDAS is a CAMAC-module in slot 1 of a CAMAC-Crate, which is capable of

- setting a LAM-signal on an incoming trigger pulse. This LAM must be resetable by a CAMAC-command.
- providing a BUSY-FF, which is used to indicate the readout-time (which means dead-time for most experiments). It must be set at the same time as the LAM-FF and reset by CAMAC-Commands.

There exists a dedicated module (originally developed at the GSI), which provides additional functions. The master-version makes use of these features, so if you want to use a different module or simulate this functions by A combination of other standard CAMAC modules, you will have to do some modifications in the readout-routines. See the appendix for a complete description of the GSI IOLAM-box.

By convention, the IOLAM box must be in slot 1 of crate 15.

1.5 CAMAC Access

From the programming point of view, all access to front-end modules is done in a *memory-mapped* way, i.e. CAMAC cycles are generated by "touching" defined addresses in the VSB address space. Each such address is not a memory location, but corresponds to a specific combination of CAMAC parameters C,N,A and F. CAMAC readout is then simply done by fetching the contents of a specific address in that address space, CAMAC write cycles (programming front-end modules) is done by writing into these locations.

Fortunately, you need not calculate these addresses by hand, but they are hidden inside "C" - macros with a simple syntax. So all you need to know is C, N, A and F of your CAMAC command to be issued. So, CAMAC access is made transparent to the user. An example readout would look like:

```
...
int data;
...
data = camac(15,4,0,0); /* CAMAC read with C=15, N=4,A=F=0 */
...
```

See the section 4.2 for more information on how to address CAMAC.

If you're interested in details, you can consult [str89] for further information about the VSB address space, for the way camac access is mapped to C-macros (`#define`), see `TDASROOT/DEFS/camac.h` (For information on the directory structure, see below).

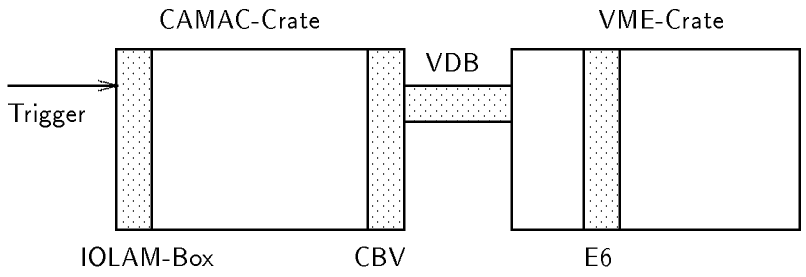

Figure 1.1: A trigger pulse (NIM) signals a physical event to the IOLAM-Box. This generates a LAM, the CBV issues an interrupt via VDB / VSB to the E6 microcomputer. Then the readout via VDB is started.

1.6 Basic Readout Scheme

Since TDAS readout is initiated by the IOLAM-box, as will be described in the following, the CAMAC-crate containing this module is called the *eventcrate*. If a physical event occurs, the IOLAM-box must generate a LAM signal. This signal is translated by the CBV to a VSB interrupt (via the VDB differential cable), which finally generates an interrupt in the E6 CPU. TDAS provides an interrupt routine, which is then called by the operating system OS-9. This interrupt routine handles all readout of front-end electronics specified.

1.7 Multiple Trigger Sources

If there are several, independent components to be read out for the experiment (e.g. the main ADC system for digitizing and a monitoring system being triggered by a pulser), TDAS provides the possibility to support multiple, independent trigger sources.

Corresponding to every trigger source, there must be a readout routine (a "C" function). The connection between trigger source and readout function is done via CAMAC scaler (counter) channels, one for each independent trigger source. The logical OR of the trigger sources is then fed into the LAM IN input of the IOLAM-box to generate an interrupt. From the TDAS side, first, the scaler is read out and the counts are compared with the values from the previous readout. Channels, which have incremented (should be by "one"), signal a trigger for this specific trigger source has arrived, and the corresponding readout routine is called.

Currently, TDAS can be operated in three modes:

- Just a single Trigger source. No Scaler is required, there is only one readout routine (option: `NO_SCALER`)
- Up to eight independent trigger sources, with integral information about the deadtime (i.e. only the *total* deadtime caused by TDAS readout together for all trigger sources is measured (option: `COMBI_SCALER` in slot 2). See figure 1.2 for a sample setup.
- Up to ten independent trigger sources, with deadtime information for each trigger source maintained separately. This means that both the counts for each trigger during the time, when TDAS is not busy with readout and the counts for each trigger with TDAS busy

or not are registered. This provides a convenient way to measure the trigger during busy caused by TDAS for each trigger source individually by subtracting the two scaler values. (option: `ALL_BUSY_SCALER`, two scalers in slots 2 and 3).

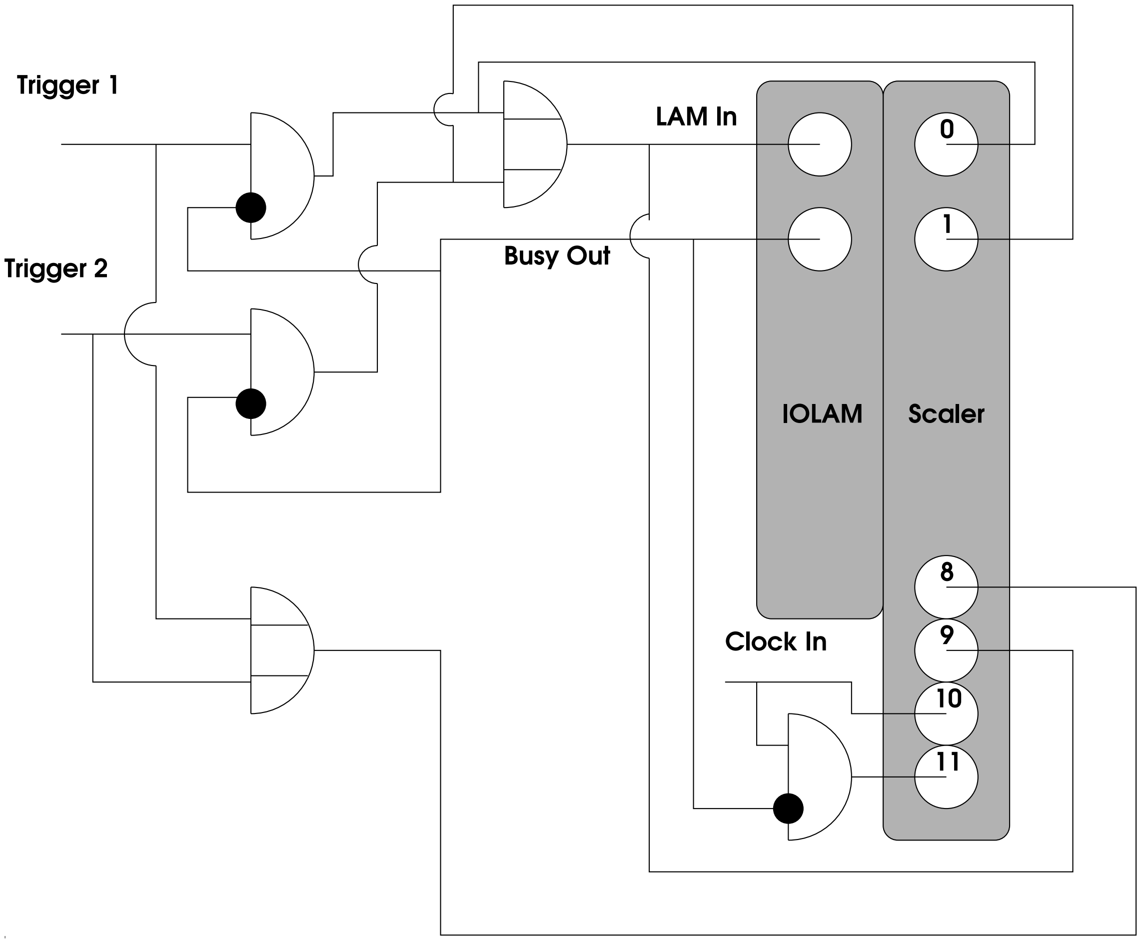

Figure 1.2: A sample trigger setup for two independent trigger sources. Scaler channels 0 and 1 register which trigger(s) have occurred. Clock-counters channel 10 and 11 provide a rough means for measuring deadtime

For the last two modes, a constant clock could also be fed into the scaler(s) as well, providing a means for measuring not only the *trigger during busy*, but also the *elapsed time during busy*, presuming a known and constant clock applied to two scaler channels. One channel should get the clock pulses "as is", the other again "vetoed" with the busy-signal from the IOLAM box. This method of deadtime measurement is not very accurate, however. Since common scaler modules are typically non-latching (i.e. the current scaler contents cannot be "frozen" instantaneously for all channels), so that the elapsed time between the readout of subsequent channels leads to a shift in counts, this information should not be taken for normalization. If it is used in spite of that, however, TDAS can show you a rough sketch of the current deadtime via TDASmonitor.

A version of TDAS supporting a latching scaler, thereby providing a means for correct deadtime evaluation is in preparation.

Moreover, the single trigger sources should also be "vetoed" during TDAS's deadtime to prevent wrong data being digitized or counted during this time. A convenient way to do this is via the *Busy-FF* output available at the front of the IOLAM-box.

See figure 1.2 for a sample setup with two independent trigger sources. For trigger sources which occur periodically (e.g. trigger by a constant clock), this setup can lead to problems. Another trigger coming in with a high frequency may (because of the veto-effect of TDAS's busy time) suppress a trigger source with low probability. This is especially true for timer (clock) generated readouts, which may be almost completely suppressed during high-deadtime periods.

To avoid this effect, the veto line given by "busy out" of the IOLAM-box must be cut for the trigger source where the lower statistics is expected.

For the `ALL_BUSY_SCALER` mode (sorry for that word...), the scaler channels corresponding to the trigger sources must be duplicated. Veto operation is then simply achieved by using the *inhibit* input of most commercially available scalers. In this setup, one scaler gets *all* the triggers, the other one just those *while not busy*.

Since most people prefer counting deadtimes themselves (doing the same job as described above, but each readout has its own (latching !) scaler somewhere in the crates), the `ALL_BUSY_SCALER` mode is rarely used, so there will be no example given here.

The actual option to be used is determined by editing a `#define` in in the include file `TDASROOT/ACQUIRE/setup.h`.

Chapter 2

The TDAS Software

This chapter gives you a survey of the various software components of TDAS. After introducing the main processes involved and the concepts of buffers and locks, the directory organisation is presented.

2.1 TDAS Processes - Producers and Consumers

TDAS's working scheme is depicted in figure 2.1. Several processes run in parallel to maximize the efficiency of the system. All data is first stored in *buffers*, which are reserved memory areas (OS-9 modules, actually). They are adjusted by the user in size and number by editing `TDASsymbols` (see section 2.2). At least three buffers must exist. These buffers are organized in a ring scheme, so that the sequence of buffers restarts at the first buffer, if the last buffer is done.

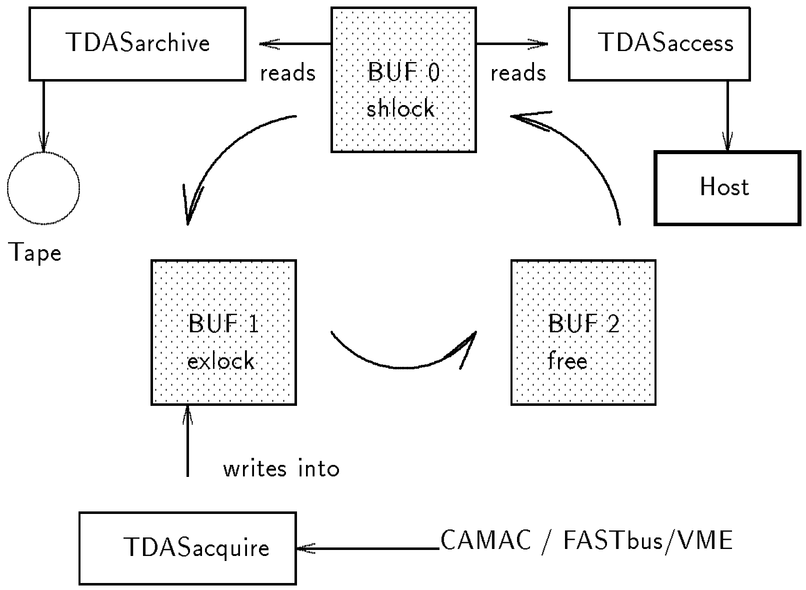

Figure 2.1: TDAS processes and buffers with their lock-state

Processes, which use these buffers are divided into two groups: *producers* and *consumers*. A producer process uses a buffer to write information (data) into it. On the other hand, a consumer process is only allowed to *read* data from a buffer. These actions must be synchronized, as it is obvious, that e.g. if a consumer and a producer share the same buffer, the data read by the consumer may become inconsistent if the producer overwrites old data. This Synchronization is achieved by the use of *locks*, which are in fact semaphores in a more general sense. They are implemented by OS-9 *events*. Don't mix up these events (signals) with the physical events! Normally, it is sufficient for the user to know that such a synchronization exists. You should never need to do something to the locking- and signal scheme. For the interested, the following describes buffertransition between different TDAS processes.

A buffer may be in one of three possible states:

- **FREE** The buffer is not in use by any process.
- **LOCKED SHARED** One or more processes are reading from this buffer.
- **LOCKED EXCLUSIVE**ly One process is writing into the buffer.

2.1.1 TDASacquire

The only producer in TDAS is TDASacquire, a process which receives the interrupt signal initiated by the IOLAM-box, and does the data readout of all specified components. It tries to lock the next buffer for writing (an exclusive lock, since no other process is then allowed to use this buffer), fills the data into this buffer and waits for the next interrupt. If the current buffer is full, TDASacquire does a *buffertransition*, which means to examine the next buffer's state. If it is not free, TDASacquire sleeps a certain amount of time and retries to find it in state free. If the buffer is free finally, it is locked exclusively. Then, the previously locked buffer is released, and data is filled into the new buffer. The above repeats everytime the current buffer becomes full, which means that less space than the maximum eventsize is left in the buffer.

Note that this means data-acquisition to stop, if the next buffer is not available and is not released by the other processes that lock it! It also means that a minimum of three ringbuffers must be supplied for the above scheme to work at all.

No other process is allowed to write into the buffers, so if there is the need to write e.g. high-voltage data (for archiving), a special communication scheme must be used to transmit the data to TDASacquire, which puts them into a buffer itself. See `TDASROOT/LIBSRC/TDASin.c` and [die94, die93] for details.

2.1.2 TDASarchive

The most important consumer process is TDASarchive, a process which tries to get a buffer (free or locked shared), locks it for reading (a shared lock, as it permits other consumers to also put a reading lock on that buffer), and does data saving on a mass storage medium (currently an EXAbyte tape unit). If all data is successfully stored on the medium, TDASarchive tries to lock the next buffer for reading. If not, the process waits for the next buffer to become free or shared locked, puts a lock on it itself and so on.

TDASarchive can actually be operated in two modes. One mode is real archiving data to tape, the other one means to dismiss data with a defined rate for testing purposes. The operation mode is selected via `TDASsymbols` (see section 2.2).

2.1.3 TDASaccessd / TDASaccessdc

Both TDASacquire and TDASarchive are necessary for the data acquisition to work. Other consumer processes are optional and are called TDASaccess-processes. These processes perform basically the same actions as TDASarchive, except for the fact that read data is not written onto a tape, but instead is transmitted via TCP/IP (Ethernet) to a remote host for online display and analysis.

To be more precise, the task of data transport via ethernet using TCP/IP has been split into two programs. TDASaccessd is a TCP/IP connector daemon, which only listens for incoming requests (from the remote host), accepts them and starts the actual data-transmitting processes (TDASaccessdc). So for each distinct connection to a remote host there runs a separate connected process, while the connector daemon is only running once. Sources for TDASaccessd (the daemon) are found in directory `ACCESSD`, while sources for `TDASaccessdc` (the connected, data transferring program) are located in `ACCESSDC`.

By the means of a shared lock, the data in a buffer may be read by several processes "simultaneously". Note however, that such a buffer is still unavailable to a producer (TDASacquire).

Furthermore, TDASaccessd can spawn other processes on request. This is used for transmission of host data to TDAS for archiving onto the same tape [die94].

Note, that all data transfer to the host is done by "handshakes": Everytime the host wants data, it has to *request* data explicitly before accessdc tries to lock buffers, copies the events and finally sends them.

Note further, that in contrast to TDASarchive, a small, but consistent sample of the currently locked buffer is *copied* to a local transmission buffer during locking. Then, the current buffer is *released* again while transmitting the local buffer data via TCP/IP. This means that a *data overrun* can take place in the meantime, i.e. TDASacquire refills the buffer which was just read by TDASaccessdc. This situation is detected by TDASaccessdc, and then a relock of the next free buffer has to take place. This means data loss for the TCP/IP connection, and so no complete set of events can be guaranteed to be transmitted to the host computer. The sample transmitted is consistent, because it is asserted that it contains complete containerevents (see section 3.3).

This possibility of data loss via TCP/IP must be accepted, since otherwise a (usually slow) TCP/IP transmission would cause TDAS data acquisition to stop (remember: TDASacquire cannot continue if the next ringbuffer is locked).

TDASarchive, however, keeps its buffer locked until the tape-write is completed, so no data loss can occur here.

To summarize:

Process	Purpose	Actions on buffers
TDASacquire	gets data (producer)	needs free buffer, locks exclusively, writes into buffer
TDASarchive	writes data to tape	needs free or shared locked buffer, locks shared
TDASaccessdc (optional)	transmits data to remote host for display and analysis	needs free or shared locked buffer, locks shared

2.2 Other TDAS Processes

There are also a number of processes not concerned with reading or writing into/from the buffers directly. The following gives a short description:

TDASmenu is a simple Zshell menu interface for starting, stopping and controlling the TDAS system during data acquisition. All menu-controlled functions are implemented in terms of Zshell-scripts. If you want to modify these, you should have a Zshell-manual [zel88] at hand.

TDASmonitor is the TDAS monitoring process, which indicates the current TDAS-status during data acquisition. It provides you with information about run number and status, amount of data written to tape, number of different triggers coming in, etc.

TDASstart Setup of all the interacting TDAS processes is not done manually. This directory contains the sources for TDASstart, the "bootstrapping-TDAS" process. It also controls termination of all TDAS processes. Options for affecting the runtime-behavior of TDAS are also controlled via this program.

TDASxxcontrol (where xx is hv, threshold, etc.). Processes to monitor and adjust high voltage systems, discriminator thresholds, etc. See [die93, die94].

TDASsymbols This is not a process, but a sort of parameter-setup-file for running TDAS. It contains the definition of zshell-variables which are read by TDASmenu and other TDAS processes (e.g. name of log- and errorfile, debug level, readout mode etc.). If you have a zshell running, you can simply type `TDASsymbols` from the system prompt to define all needed parameters. Note: TDAS will not run without these variables supplied! Use an editor to modify the values in TDASsymbols. TDASsymbols also defines a shorthand-name for TDASmenu, independent of its actual name. After running TDASsymbols, you may simply call `tm` at the system prompt to get the menu running.

The TDAS process hierarchy is depicted in 2.2. Invocation of TDAS processes is not done directly by the user. Instead, a menu-driven interface has been supplied to ease the task of setting up all processes which are executing concurrently. This is done by TDASmenu / TDASsymbols (see below). Whenever the user selects a TDAS startup-command from the shell, TDASmenu issued a commandline which performs the invocation of TDASstart with an appropriate command-line containing all user-selected options. The function of TDASstart will be explained below.

Communication between the various TDAS processes is done by *pipes* associated with *signals*. Each process sets up a *commandpipe*, which means an inputstream from which commands are received; and a temporary *resultpipe*, which is another datastream to indicate the command result to the caller. The occurrence of such a command is indicated by a *signal* (which is sent by the caller), so that no command polling is necessary. All commands are ASCII-strings written (or "echoed") into the processes command pipes. Typical commands issued by TDASmenu include "BEGIN_RUN", "END_RUN", "SUSPEND_RUN", "RESUME_RUN", etc.

TDAS shutdown (user-controlled killing of all processes) is also done via TDASmenu - TDASstart. TDASstart issues and end-of-run command (if appropriate) and then calls `texit()`, which kills all processes with their correct cleanup.

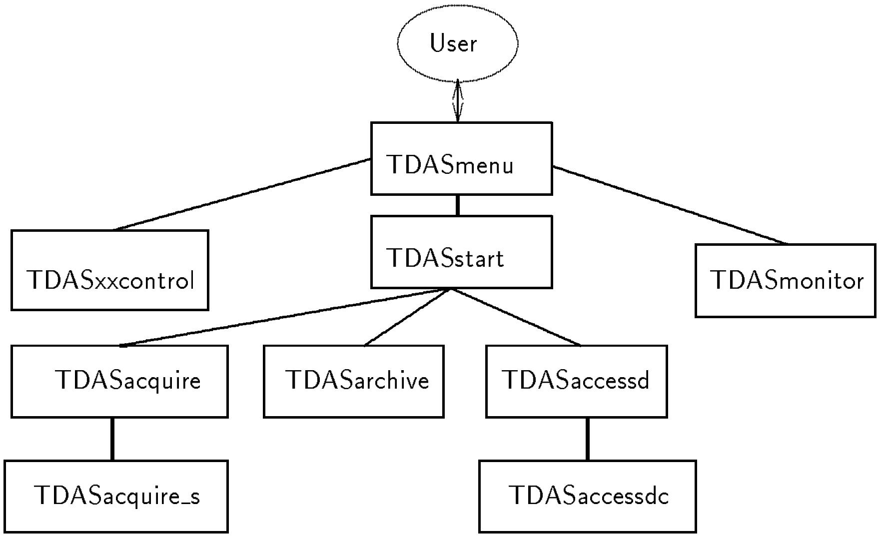

Figure 2.2: Hierarchy of TDAS main processes. Parent / child relationship is shown top to bottom. TDASxxcontrol can actually mean a variety of controlling processes (see text).

2.3 Directory Organisation

The TDAS software is organized into several directories below a "starting point"-directory which is called `TDASROOT`. You may freely choose `TDASROOT` to reside anywhere in your filesystem e.g. `/h0/TDAS` would be a good place for starters.

You should copy the whole TDAS-software below this directory, and do any modifications here freely to create your own experiment-dependent version.

For your experiment, there should be a directory with the name of your experiment under `TDASROOT`, e.g. `TDASROOT/EDDA`. This directory should contain all the files necessary for your specific customization. If not, copy an existing experiment directory as a starting point.

If you succeeded copying the master-version, you should find something like the following:

```
TDASROOT ---- ACQUIRE        sources for TDASacquire/acquire_s
          |
          --- ARCHIVE        sources for TDASarchive
          |
          ...                more source directories for TDAS
          |
          --- DEFS           .h header files
          |
          --- MENU --- SCMDS TDASmenu user interface / Zsh commands
          |
          ...                more directories for TDAS
          |
          --- EDDA --- CNF   experiment directory with configuration-
          |                  and runfiles subdirectory
          ...                more directories for the experiments
```

All directories containing sources have a `RELS` subdirectory, which holds the corresponding object code and is not listed here. Furthermore, version control is done via CVS/RCS (don't worry if you don't know this), so for every directory there is also a `CVS` subdirectory.

For your own customization of TDAS, the only relevant directories should be your *experiment directory*, containing all your specific sources as well as your own version of `TDASsymbols`, to adjust TDAS parameters for your needs, and the `CNF` directory, containing further configuration files. See the example section for a hypothetical experimental setup.

2.4 Signal versus Interrupt Mode

For data readout, there are two ways provided by TDAS: a clean, but slow (*signal*); and a "dirty", but fast (*interrupt*) way. These modes are explained to you in the following.

TDASacquire (the data-collecting process) is actually split into two processes, a "user-state" process and a "system-state"-process. This is due to the fact that interrupt-handling with physical I/O is only possible if a program runs in the so-called "system-state", which is the kernel's level of operation. This operating mode is the most powerful, but also the most dangerous one. Since it is the operating mode of OS-9 itself, it allows almost arbitrary control over all system resources. As a consequence, a failure in a "system-state"-process almost always causes a complete system hangup, where rebooting the system is all you can do. On the other hand, a user-mode program is controlled by OS-9, and may behave better on failure. So (by breaking things up into two processes)there is at least in principal a division between interrupt-controlling routines and "user code".

The system-state subprocess called `TDASacquire_s` is started by TDASacquire and contains the entry point for the interrupt-initiated readout. The "clean" way (see above) means to catch the interrupt (from IOLAM-box - VSB), and issue a READOUT signal to TDASacquire, which contains the user code. This receives the signal asynchronously and calls the appropriate readout-routine. The main disadvantage is that this mechanism takes about 0.3 msec, which means a deadtime of about this amount. A faster, but more dangerous way is to look for the memory address of the readout-routine (in TDASacquire), save it, and directly jump to this address during interrupt. This is significantly faster (readout times of about 0.1 msec), but causes user-written code to be executed during the sensitive time of interrupt processing. Coding errors and unconscious use of system calls will most likely cause the system to crash. So it is recommended that development of user-readout routines takes place while using so-called signal-readout-mode.

When choosing a nonzero debug level in `TDASsysmbols`, which may help in finding subtle errors during readout, be sure to select the *signal readout* mode. Since no I/O via the standard system calls is possible while in interrupt mode, no logging messages can be printed unless the readout mode is set to "signal".

The choice whether to do the readout by "signal" (slow, safe) or by "interrupt" (fast, risky) is made in the TDAS configuration file TDASsymbols in your experiment directory.

Chapter 3

Data Formats

The binary representation of the physical data in TDAS is given here. After presenting some notational conventions, the binary structure of *containerevents* and *subevents* is described. Finally, the format of data on tape is briefly discussed.

3.1 Bytes, Shorts and Longs

When talking about the binary representation of the experimental data, the following conventions are assumed:

Bytes consist of 8 bits and are used in both signed and unsigned flavours. In "C" available as `char` or `unsigned char`.

Shorts (short ints) are 16 bit wide and occur in both signed and unsigned flavours. The use of "word" meaning "short" should be avoided, since on true 32-bit machines (like the E6), the native machineword is a "long".

Longs (long ints) are 32 bit wide and occur in both signed and unsigned flavours. One some machines, a "long" means an 8-byte integer value. On most machines (like the E6 and many UN*X hosts), however, a long means 4 bytes, and we will stick to this assumption.

3.2 Alignment, Length Information

All length information is given in bytes. All data is long-aligned. This means: each datum starts on a memory address a, so that $(a \bmod 4) = 0$ for the sake of speed. This means that even for e.g. 8-bit ADC values, a whole long is used to store this value (This is due to the way CAMAC is addressed: every memory-mapped CAMAC address is capable of holding a long, so for all CAMAC readouts, 32 bits are returned).

This does not refer to fields like type / subtype (see below), or other "packed" data inside a long.

3.3 Containerevents and Subevents

The binary structure used by TDAS for representing physical events is shown in figure 3.1. There are two main classes of eventstructures: *containerevents* and *subevents*. Both

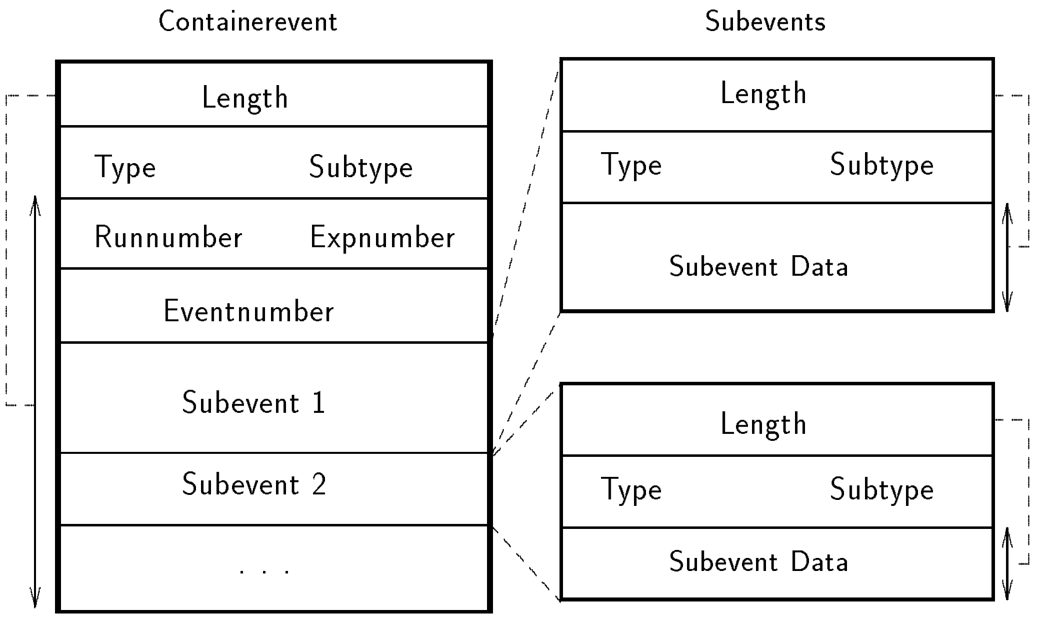

Figure 3.1: Layout of Container- and Subevent structure. The width of a box is 4 bytes (a long)

share the same header information: an unsigned long giving the total length (in bytes) of this containerevent / subevent (*excluding* this header), and a type/subtype identification (each an unsigned short), which is used to classify this event. This header is then followed by data, which meaning is given by the type/subtype information field. The important difference between container- and subevents is the fact that containerevents contain data + subevents, and subevents carry only their data.

Not all defined types/subtypes are used, but the user can (in principle) employ all defined types or add new types/subtypes according to his or her needs. Both types and subtypes are actually 16-bit values (shorts), but are given in a symbolic notation here for readability. The actual values for types and subtypes do not matter, and care should be taken to use them only in their symbolic notation (see `DEFS/ev_types.h`).

Type and Subtype information is used to label the data according to its meaning. Most host analysis software can "hunt for" type/subtype combinations to identify the user's data. The entire scheme makes no assumption about the *contents* of the containerevents / subevents, but uses it as a "transport layer" for transmission and identification only.

Valid types (with their subtypes) for containerevents are e.g.:

DUMMY This containerevent reflects no physical event. It can be used to "keep things going" by issuing one of these events in regular time-intervals, e.g. for preventing the tape to stop moving or an synchronous ethernet connection to block due to lack of data.

INVALID This eventtype can be used to mark illegal or corrupted events during acquisition.

CONTROL This containerevent indicates an *info event* like begin/end of run etc.

RAWDATA This containerevent contains subevents belonging to one physical event. This is the most common type used during run. It generally contains one complete readout of the front-end electronics. Various trigger sources of subsystems can be distinguished by setting different subtypes. By convention, the actual trigger pattern belonging to a containerevent is coded bitwise in this subtype.

... further user-defined containerevent-types

Each of these containerevents can further be distinguished by setting specific subtypes, e.g. RAWDATA is typically divided into "coming from triggersource 1" or "coming from triggersource 2" etc. by automatic setting of the subtypes TRIGGER01, TRIGGER02 etc. Control events can further be specified by setting their subtype to BEGIN_RUN, END_RUN, BEAM_ON, etc.

A container event furthermore always has the information runnumber, experimentnumber, and eventnumber. This is followed by zero or more subevents. For instance, a RAWDATA containerevent may contain one subevent coming from an ADC (or a whole ADC subsystem), followed by some scaler information. Then for both the ADC(s) and the scaler(s) a subevent would be generated.

Some of the possible Subevent types are:

PARAMETER Usually connected to CONTROL-type containerevents. Here various run parameters can be stored as part of the current run, e.g. values for the electronics, current beam energy.

TEXT Usually connected to CONTROL-type containerevents like BEGIN_RUN, END_RUN etc. It is primarily used for user run comments.

TIME This subevent contains a time stamp.

CAMAC_DATA For use with RAWDATA containerevents. This is the general type for CAMAC readout. Specified further by its subtype (FERA_ADC_COMPRESSED, etc).

... further user-defined subevent types

For a complete definition of eventstructures and their type / subtype definitions, see `DEFS/ev_types.h`

3.4 Tape Data Format

Historically, TDAS writes its memory buffers to tape in the so called *ltf* (labeled tape facility) format. This format is based on

1. American National Standard Institute magnetic tape labels and file structure for information interchange ANSI X3.27-1978

2. The document/working paper: Draft Proposed Revision to ANSI X3.27- 1978 Public Review Comment on ANSI X3L5/83-28T 15-Oct-84 (describes the version 4 ANSI standard)

Based on these definitions, there are tools for handling these tape on some commercial platforms.

The format is, however, quite simple. A brief description, suitable for dealing with the tape contents from e.g. a UN*X host is (all byte counts starting with zero assumed):

- A 128 byte record containing the tape's volume label at bytes 4-9
- A filemark (a tape EOF marker, as it is written e.g. under UN*X with `mt eof 1`.
- 4 header records, 128 bytes each, containing among other things block- and recordsize and the name of the file (HDR1, bytes 4-21).
- A filemark
- Data records for the first file. They have a variable blocksize/recordsize of 16 * 1024 bytes maximum.
- A filemark
- 4 file trailer records of 128 bytes each.
- A filemark
- Then the sequence starts again with the header records for the second file.

Typically, there is host software available for copying and viewing the tape's data, so you need not worry about low-level access to the run file structure. A copying program may simply use the standard UN*X `dd -bs=16k` for copying the data part.

The structure inside the run data is simply the contents of the TDAS memory buffers, i.e. a sequence of containerevents as discussed above.

Chapter 4

Customizing TDAS

4.1 Compiling and Linking

Compilation and linking TDAS programs is done via `make`. In case you are not familiar with the `make`-utility, refer to [mic84].

Every subdirectory under `TDASROOT` which contains code to be compiled comes with a file called `makefile`. This file describes the dependencies of programs and their sources. `make` reads these dependencies, checks if any sourcefiles have changed, and performes the appropriate actions (usually start compilation of the modified sources). It also installs the the produced executable files in `TDASROOT/CMDS`, which contains all TDAS programs. Unless you want to add new source files to or remove existing sources from a program, or want to change compilation / linking options, you need not modify the makefiles.

For changing the readout-part as discussed below, typing `make` in your experiment-specific directory should be sufficient.

If you want a total recompilation of TDAS, type `make` from `TDASROOT`.

4.2 CAMAC commands

To read and write CAMAC data, use the `camac(C,N,A,F);` macro, with `C` meaning crate number (0-15), `N` meaning slot number (1-24), `A` meaning subaddress and `F` meaning function code. Reading is then done like e.g.

```
...
int data;

data = camac(15,5,0,0);
...
```

whereas writing is done with

```
...
int data=5;

camac(15,5,0,24) = data;
...
```

Simple dataless CAMAC cycles are performed with `touch(camac(C,N,A,F));`. For example, a clear command is done with

```
...
touch(camac(15, 10, 0, 9)); /* clear ADC via NAF 10,0,9 */
...
```

The `touch()` is used in order to read the VSB address corresponding to CNAF 15 , 10, 0, 9, but to dismiss any value located there. The CPU's read access causes the CAMAC command to be executed.

In fact, the preprocessor replaces `touch(x)` with `if(x);` and `camac(...)` by `*(adr)`, where `adr` is calculated according to the C,N,A,F values given and means an address in the VSB address space. So, after the preprocessor stage of compilation, the compiler sees a statement like

```
...
if (*(0xf1f8bc03));
...
```

which does not make much sense when only regarding C-language. But it in fact issues the CAMAC-command wanted *and* prevents the compiler from complaining. Totally equivalent for the CAMAC command would be `camac(15, 10, 0, 9);` , but this would cause the compiler to produce something like

```
cpp:
c68020:
"user.c", line xxx: **** warning - expression with little effect ****
```

- a reasonable idea, since from the compiler's point of view, it is something like `23;` or `x+1;`.

So the `touch()` macro simply suppresses these spurious compiler warnings without producing unnecessary code.

Accordingly, use `camac_q(C)` and `camac_x(C)` to check for the CAMAC q and x flags for crate number `C`, respectively. Use `camac_qx(C)` to check for the logical `OR` of of CAMAC q and x.

See `DEFS/camac.h` for a complete definition of `touch()` and `camac()` macros.

4.3 How to Write a Readout Function

All readout to be performed should be contained in the `user.c` file in your experiment directory. In general, each readout function is called by TDAS when it's due (okay, you must help a little, too).

Each incoming hardware trigger generates a containerevent with the appropriate header information filled in already. The user can then add his or her information by filling them into *subevents*.

For subroutines that are called when a new containerevent is generated, a pointer to the current event data area is passed as an argument (type `char *`). It is guaranteed to point to the first free byte after the information written already for the current event, so you can add your experiment-specific subevents here. The incremented startpointer must be returned to the caller (TDASacquire), pointing to the first data byte after all written subevents (the next "free space"). Care must be taken to obey this rule, since otherwise invalid subevents will be produced by your code (All containerevents are guaranteed to be valid and consistent by TDASacquire).

The readout routine may write zero, one or more subevents, one after the other, but the events written must be consistent with the event format as described in section 3.3.

To ease the task of writing consistent subevents, some "C"-macros are provided:

- `begin_sevt(ptr, TYP, SUBTYP)` makes you start a new subevent at the location `ptr`, giving it type `TYP` and subtype `SUBTYP`. `ptr` is incremented automatically to point now just *behind* this new subevent header.
- `end_sevt(ptr)` declares the current subevent to be finished at location `ptr`. Length calculation for the subevent is done automatically.

These macros also assert the *long alignment* discussed in section 3.2. With the assistance of these macros, a valid readout routine could look like:

```
/*>------------------------------------------------------------------------*/
char *readout_scaler(startpointer)
char *startpointer;
/*
 *   input: pointer to current free event buffer location
 *
 *   return: incremented startpointer
 */
/*<------------------------------------------------------------------------*/
{
register int *data;            /* pointer to write integer data */

begin_sevt(startpointer, SEVT_T_CAMAC_DATA, SEVT_ST_SCALER);
data = (int *) startpointer;   /* data is now right after sevt header */

*data++ = camac(15, 22, 0, 0); /* read scaler crate 15, slot 22, ch 0 */
*data++ = camac(15, 22, 1, 0); /* read scaler crate 15, slot 22, ch 1 */

startpointer = (char *) data;  /* the end of our data */
end_sevt(startpointer);        /* finish the sevt, calculate length */
return startpointer;           /* return the end of our subevent */
}                              /* end of readout_scaler */
```

Note the end of the readout function: It returns the *incremented startpointer*. This determines the point where TDAS truncates the current subevent, so watch out to return the correct value.

4.4 How to Get Your Readout Function Called

In `user.c`, there are some functions called by default. These routines must exist for TDAS to work properly. If you don't want any readout to be connected with them, simply do nothing inside these functions, but remember to return the startpointer which is passed to them unchanged.

user_begrun_camac(startpointer) This routine is called everytime a new run is started. Since a new run writes a BEGIN_RUN containerevent to tape, you have the chance to add subevents here. Follow the description above.

endrun_camac(startpointer) This routine is called everytime a run is ended. Since an ended run writes a END_RUN containerevent to tape, you have the chance to add subevents here. Follow the description above.

Apart from these fixed "readout" functions, you can define any number of own readout functions as described above. They can have any name, but must accept one `char *` argument and must return the incremented bufferpointer as explained above.

The readout functions are *registered* as "to be called" in the function `install_readout_functions()`.
This function is also used to assign readout functions to a *trigger pattern* given by the scaler values as discussed in section 1.7.

To connect your readout function with e.g. trigger source no. 1, simply use the function `assign_readout_function(1, func);`, where `func` is the name of your readout function.

Suppose you had only one readout function `readout_scaler()` as above, then your `user.c` would contain:

```
/*>-------------------------------------------------------------------------*/
char *readout_scaler(startpointer)
char *startpointer;
/*
 *   input: pointer to current free event buffer location
 *
 *   return: incremented startpointer
 */
/*<-------------------------------------------------------------------------*/
{
register int *data;            /* pointer to write integer data */
...
return startpointer;
}
/*>-------------------------------------------------------------------------*/
void install_readout_functions()
/*
 *      just use :   'assign_readout_function( pattern, function_pointer );'
 *      to attach a trigger pattern to a ReadoutFunction.
 *
 *      int pattern : 1 .. 15 are reserved yet for hardware triggered readout
 *                     ( see setup_readout_functions for all )
 *
 *      function_pointer : just use the function name without parameters and
 *                          parentheses
 *                          e.g. : 'readout' is a pointer at the function
 *                                 'char *readout(char *)'
 */
/*>-------------------------------------------------------------------------*/
{
/* here comes the 'private' readout function assignment */
```

```
assign_readout_function(1,readout_scaler);
}
```

You can use up to 8 or 10 different readout functions here (because of only 8 or 10 distinct scaler channels for trigger sources with option `COMBI_SCALER` or `ALL_BUSY_SCALER`, respectively). Note, that "`1`" here means scaler channel zero, since scaler numbering starts with zero.

Chapter 5

Running TDAS

5.1 Preliminaries

First, get a copy of TDAS and make sure you have an experiment subdirectory set up for you. If not, copy an existing one. Make sure the hardware is set up correctly.

- Connect one or more CAMAC crates to the VSB. Make sure, there is a crate with crate number 15 (0F hex.), which contains the IOLAM-box (or a substitute).
- Connect the EXAbyte. Make sure the SCSI id is set to '3'.
- Choose the scaler mode (see section 1.7) in `ACQUIRE/setup.h`.
- Change to your experiment directory below `TDASROOT`.
- You may want to add new types/subtypes to `TDASROOT/DEFS/ev_types.h` for your setup. Edit this file to add new types.
- Edit your own `user.c` (in your experiment directory). See the example in the previous chapter.
- Type `make` in your experiment directory. This should cause all relevant sources to be recompiled automatically.
- Edit `TDASrun.cnf` in your experiment directory. Delete everything but the first and the next few lines (starting with "`#`") and set the run number in the first line to 0. This causes the run history to start with 1 when you start the next run. Save the file.
- Edit TDASsymbols to adjust all parameters. This should be sufficiently explained in TDASsymbols itself.
- Call TDASsymbols. This should cause all relevant TDAS parameters to be set.

Finally, you should be able to start the TDAS main menu with `tm`. See section 5.2.

5.2 TDAS main menu

The communication between the user and the TDASprocesses works by the more or less convenient interface of TDASmenu, which is the only process that the user has to deal with. Menu items start up or shut down the whole TDAS-system (which means to call TDASstart on one hand and to send a terminate - command on the other hand), begin a run, end a run, and perform various other functions by simple keypress. Note, that TDASmenu actually consists of zshell-scripts, and is not a compiled "C"-program. You can simply edit the file containing the TDASmenu commands and do changes according to your needs. For the sake of maintainability, many menu-items have been put to separate files, see the contents of the MENU/SCMDS (ShellCoMmanDS) directory. Once you started the TDAS menu interface, you should see something like:

```
**********************************************************************
*                            TDAS main menu                          *
**********************************************************************

Current Run Number:      1

S: TDASstart submenue
A: TDASacquire submenue
M: TDASarchive mass storage submenu
H: TDAShvcontrol submenu
D: TDASdelaycontrol submenu
T: TDASthresholdcontrol submenu
P: show all TDAS processes
X: call TDASmonitor
L: view log file
E: view error file
$: spawn new shell
Q: quit - return to OS9

**********************************************************************
```

Pressing the indicated key invokes the corresponding function or submenu. All submenus work in the same way with different menu entries. The most important entries will be discussed in the subsequent sections. Others may be discussed in different manuals.

5.3 Starting and Stopping TDAS - TDASstart submenu

With the TDASstart submenu, you can start up or shut down all TDAS processes. Starting means not to actually start a run, but to prepare everything for doing so. Normally, it is the first thing to do once you're in the TDAS menu.

Starting up TDAS may take a few seconds. If there are errors during startup, they will be shown on the screen and will (sometimes) be logged into the TDAS error file, which is viewable from the main menu.

After startup, all processes are ready for operation. You should continue with the mass storage submenu (see section 5.5) to mount an empty tape, or go directly to the acquire submenu to start a test run (see section 5.4).

Shutting down means to stop all TDAS processes. This is neccessary if e.g. you change readout commands in user.c (you need to recompile then!), or switch settings in `TDASsymbols`. You then have to do a shutdown/startup sequence to get TDAS running with the new parameters.

5.4 Run Control - TDASacquire submenu

From the TDAS main menu, run control is done via the TDASacquire submenu. Here you can start or stop runs, suspend (temporarily halt) and resume (restart after suspending) runs.

When starting a new run, you will be asked whether it will be a *dummy/test run*. This means that no data is written to tape and the run is not counted in the TDAS run history file `TDASrun.cnf` (in your experiment directory) , which contains the history of all runs started so far and the current run number as well. Everythings else works just like a "normal" run.

If everything seems to work allright, you can call TDASmonitor from the main menu and watch the progress of data acquisition.

You can stop a run any time. Note further, that a run is automatically ended when the *maximum run file size* is exceeded. If this happens, a new run is started automatically. The maximum run file size is adjusted via the mass storage submenu. This is done to prevent the single run files on the tape from becoming too big (to be handled conveniently for offline analysis).

5.5 Mass Storage Operation - TDASarchive submenu

Currently, mass storage is only possible using an EXAbyte8500 connected to the SCSI bus. It must be adjusted to have the SCSI ID 3. Tape mode is selected in `TDASsymbols`.

For test purposes, you can also select a "dummy" archiving mode via `TDASsymbols`, which dismisses data with a defined rate instead of putting them to tape.

Furthermore, to temporarily leave out a run on a tape cartridge, that is currently mounted, you can select "dummy run" when starting the run via the TDASacquire submenu. The run is then neither counted nor written to tape, but the readout is performed as usual.

For both "dummy" archiving mode and test run modes, host access to the data via TDASaccess processes is still possible and can be used to check the readout without wasting tape.

All mass storage functions are available via the TDAS mass storage submenu. To write onto a tape cartridge, it must be *mounted and initialized* first. You are prompted to enter a 6-character tape label.

Use this label to identify the tape. It should somehow identify the beam period and the tape sequence number, e.g. B01T05 is one of the best things you can make of a 6-character label.

This takes quite a few seconds, so don't be impatient. After mounting, you can start and stop runs freely via the TDASacquire submenu.

The tape can be dismounted and ejected with another menu entry here. *DON'T* remove the tape manually by pressing the eject button at the tape unit! At best, the last run may become unreadable. Furthermore, it may not be possible to use this cartridge for appending to it anymore!

You can *dismount and unload* the tape any time to change tapes. Furthermore, new runs can be appended to an existing tape (you will be prompted for the tape label to avoid misidentification).

You can further change the maximum tape file size here (see section 5.4).

When the tape is (almost) full, the current run will be closed and the cartridge ejected from the tape unit.

5.6 TDASmonitor

Selecting this menu point will show you the current status of TDAS. The readout mode, run number, data size of the current run and on the current tape is shown to you as well as information about different trigger sources and a rough approximation of the deadtime. Also, the last error or log message is shown to you at the bottom of the screen.

This the most useful program while TDAs is running. You should keep it running during data acquisition. TDASmonitor is exited by pressing ”q”.

Appendix A

The GSI IOLAM-Box

The GSI IOLAM-Box serves as a simple, but versatile CAMAC-module to concentrate the functions of LAM-generation, NIM input and output in a single width module.

It has the following features:

- 4 channels of NIM-level-outputs, O1 ...O4 with status LEDs
- 4 channels of NIM-level-inputs, I1 ...I4
- 4 channels of NIM-pulse-outputs, P1 ...P4
- 1 LAM-input (NIM-level), LI
- 3 FlipFlop outputs for LAM (LF), Busy (BF) and status (SF) with status LED for each

O1...O4 can be set by A(0)F(16) = x, where $x = \sum_{i=1}^{4} O_i * 2^{i-1}$ (the logical OR of the output lines to be set, where $O1 \cong 1$, $O2 \cong 2$, $O3 \cong 4$ and $O4 \cong 8$). The current state can also (together with the state of be LAM FF, BUSY FF and STATUS FF) be read via x = A(0)F(0).

I1...I4 as well as the current values of the FFs can be read via CAMAC command A(1)F(0). Bit-assignments to read lines as in O1...O4.

P1...P4 can be fired - producing a 100 nsec NIM pulse - via A(1)F(16) = x. Bit-assignments to pulse channels as in O1...O4.

LF is set by NIM-pulse on LI or via CAMAC command A(0)F(25) and is reset by CAMAC command A(0)F(10).

BF is set together with LF and reset by A(1)F(16) = 32

SF is set together with LF and reset by A(1)F(16) = 64

LI generating LAM can be disabled by A(0)F(24) and enabled by A(0)F(26). The FF state can also be read with A(0)F(0), with bit 4 meaning LF, bit 5 meaning BF and bit 6 meaning SF.

Glossary

CAMAC abbreviation for **C**omputer **A**pplication to **M**easurement **a**nd **C**ontrol. It is a modular standard for a data transfer protocol via a specially designed backplane, capable of operating at 1MHz with 16 or 24 bits wide access. Usually up to 25 Modules can be placed in a single Crate, which requires one dedicated module, the *controller* to connect to various other computer standards (VME, GPIB, QBUS, etc.) Many modules especially designed for nuclear physics data acquisition are available. Commands can i.g. be issued by specifying crate number (C), slot number (N), Subaddress (A) and function code (F), which are common to many modules.

CBV a (bad) abbreviation for **C**AMAC to VSB interface. A piece of hardware used as a CAMAC crate controller for connection to the VDB. See [str89] for description.

CVC an 'intelligent' version of the CBV containing a 68030 CPU, running under OS-9 and LYNX/OS. Includes SCSI, thin wire ethernet and some MByte of RAM and a somewhat faster CAMAC access.

CVI the older, 'intelligent' version of the CBV containing a 68030 CPU, possibly running under OS-9. Predecessor of the CVC.

E6 an abbreviation used for the EUROCOM 6 CPU, the microcomputer where OS-9 and TDAS is currently implemented and run.

Event another term meaning at least two different things. An event is either a physical event, indicated by a triggersignal (the detector has fired), or (in the OS-9 context) a communication means between processes related to *signals*, basically used for indicating a resource (e.g. a common memory area) as occupied or free. It is a special implementation of the more general concept of *semaphores*. For TDAS, e.g. the locking of buffers is done by events. See [mic87, Technical Manual].

FASTBUS is a newer modular standard for a data transfer protocol. It provides 32bit-wide, synchronous and asynchronous transfer via the ECL backplane as well as multiple processing features with multiple master modules (as opposed to single "master" operation of CAMAC). Up to 30MHz operation with 32bit-wide transfers can be performed.

I/O redirection Originally developed with UNIX, the means of "making this program's in- and output go there" is also built into OS-9. It means a flexible way of directing data streams to and from various processes or link them together (see "pipe").

LAM **L**ook **A**t **M**e. Each module in a CAMAC-crate has its own signal line for indication of special states (according to the modules function), e.g. an ADC may indicate "conversion complete" to the CAMAC controller. The IOLAM-Box uses its LAM-line for signaling occurrence of a physical event to the CBV.

Module a term used in (too) many contexts. A module can be: a hardware device, such as an ADC card; a file, usually containing source code; and in OS-9, a part of the system's memory (here the latter is assumed). It may contain an executable program, a common data storage area (e.g. in the case of TDAS the *buffers*, a shared library etc. Methods for module handling are provided by OS-9, e.g. creation, "linking" i.e. connecting a process to a module or deletion. See [mic87, Technical Manual, The Kernel, Memory Management]

Pipe A pipe is a resource usually supplied by the operating system (originally coming from UNIX, and also adopted for OS-9). It means a part of memory which is used for buffering data in a "first in - first out" scheme. Often used for process communication. Pipes can be unnamed (connected to processes "automatically" via a shell command) or named, in which case they behave very similar to ordinary files. See [mic87, Using Professional OS-9]

SCSI abbreviation for **S**mall **C**omputer **S**ystem **I**nterface. A device-independent 8 bit-wide parallel bus capable of transfer rates up to 5 Mbytes/sec over distances of 6m (single ended, inside a computer) or 25 m (twisted pair, differential cable for external devices). It is mainly used for connecting hard disks, tapes, etc. to a computer.

Semaphore A concept for access control to sharable resources, e.g. shared memory or files used by several processes "simultaneously". Often implemented as a variable capable of having the values *locked* and *free*, with the operation of testing and setting the variable made "unbreakable". This means that no taskswitch must take place between testing and setting the variable, which has to be guaranteeed by e.g. the operating system.

Signals are used for interprocess communication under OS-9. A process can send or receive signals to or from other processes via system calls, and may so synchronize its operation. See [mic87, Technical Manual].

Socket a simple way of interfacing TCP/IP from user programs. a socket is a unique number (similar to a file descriptor) describing the connection between the two participants in ethernet communication. Library calls are used to create, communicate with and delete sockets.

TCP/IP an abbreviation for **T**ransmission **C**ontrol **P**rotocol / **I**nternet **P**rotocol. A standard protocol used for data transmission over ethernet. A simple and convenient programmer interface is given by *sockets.*

VDB an abbreviation for **VSB** **D**ifferential **B**us. A standard for interfacing VSB over longer distances. Up to 50m total length can be achieved by using VDB.

VME abbreviation for **V**ersa **M**odule **E**urocard. Often used when actually meaning VME-bus, a computer architecture. Its basic features are a 32bit data and address bus, multiprocessing and up to seven interrupt levels. Data transfers can be handled up to 40Mbytes/second. It has become the *de facto* standard of the industry.

VME Crate a standard housing for VMEbus computer modules and peripheral devices. It provides the power supply and the backplane (which is the physical realization of the VMEbus).

VMS An operating system for Digital VAX computers. This has nothing to do with the other "three-letter-words" used in this manual. Got confused, finally ?

VSB an abbreviation for **V**ME **S**ubsystem **B**us. It is a "side bus" of the VMEbus, commonly used to reduce traffic on the main bus. It is 32 bit wide (address and data), and is often connected to via the rear of the VME crate backplane.

VSC a (bad) abbreviation for **VS**B to differential **C**able Interface, meaning VSB to VDB interface. A piece of hardware used to translate all signals from and to local VSB / remote VDB, respectively. See [str89b].

Bibliography

[com91] Douglas E. Comer: Internetworking with TCP/IP, Volume 1: Principles, Protocols and Architecture, Second Edition, Prentice Hall 1991

[die93] Olaf Diehl: TDAS HVcontrol User's Guide, Bonn University, ISKP internal report 1993

[die94] Olaf Diehl: Diploma Thesis, Bonn University, ISKP 1994, in preparation

[elt91] ELTEC electronik mainz: EUROCOM-6 Documentation 68030 CPU Board Revision 2 A, ELTEC electronic mainz GmbH 1991

[fuj91] Fujitsu: Fast Track to SCSI, Prentice Hall 1991

[gas93] Martin Gasthuber: XD manual version for XD system 1.1, Hamburg University, internal report 1993

[hil84] Werner Hilf, Anton Nausch: M68000 Familie Teil 1 - Grundlagen und Architektur, te-wi Verlag 1984

[ker77] Brian Kernighan, Dennis Ritchie: The C-Programming Language, Prentice Hall 1977

[lin89] Volker Lindenstruth: TDAS - Temporary Data Acquisition System, GSI Darmstadt internal report, 1989

[lec90] LeCroy Corporation: 1990 Research Instrumentation Catalog, LeCroy Corporation 1990

[mic84] Microware Systems Corporation: OS-9/68000 Language Manuals, Microware 1984

[mic87] Microware Systems Corporation: OS-9/68000 Operating System Manuals, Microware 1987

[mic89] Microware Systems Corporation: Using OS-9 Internet, Microware 1989

[pet89] Wade D. Peterson: The VMEbus Handbook, VITA 1989

[sch92] Horst Scheid: ?, ISKP 1992

[sch94] Florian Schwandt: YODA User's Manual, Bonn University, ISKP Internal Report 1994, in preparation.

[str89] Struck: Technical Manual STR610/CBV - CAMAC to VSB Interface, Dr. B.Struck 1989

[str89a] Struck: Technical Manual STR611/DMS - CAMAC Dataway Display, Dr. B.Struck 1989

[str89b] Struck: Technical Manual STR723 - Differential VSB Extension, Dr. B.Struck 1989

[zel88] Lukas Zeller: ZSH - An Advanced Shell for OS-9/68k, Lukas Zeller 1988