The Midas DAQ System

— Version Refer to New documented features —

A versatile and compact Data Acquisition System

Stefan Ritt, Pierre-Andre Amaudruz

Contents

Welcome to the world of Midas

Midas is a versatile Data acquisition System for middle range physics experiments.

This document will try to answer most of your questions regarding installation, setup, running, and development.

Here is a list of highlights:

- General description of the Midas system.
- Setup and acquisition examples.
- Q and A listing.
- Library calls reference.

If you think Midas can help you for your projects and you want to get more info on it, feel free to browse through the following Web sites:

EU, CDN

Enjoy!

1 New documented features

New documented features.

Some of the midas features are not yet fully documented or even referenced anywhere in the documentation. This section will maintain an up-to-date information with a log of the latest documentation on past and current features. It will also mention the wish list documentation on current developments.

Current doc revision: 1.8.3-4

Software version: 1.8.3

Latest tarball : 1.8.3-3

Wish list ...

1.8.3-4

- **elog task** Added comment to this section.
- **mevb task** Midas event Builder task. This task provides multiple frontend data gathering/filtering/concatenation mechanism. Code in the cvs

1.8.3-3

- **ESONE CAMAC standard functions**: esone.c inclusion of cclnk , cculk, ccrgl.
- miniexpt directory: update files for proper build.
- **appendix A: Data format**: New pictures.
- **ODB /Experiment Tree**: New key **"Edit run number"**.

1.8.3-2

- **ODB /Logger Tree** Specifing independent Data directory for each logging channel.
- **ODB /Logger Tree** Optional History directory, ELog directory declaration.
- **Midas Library** Addition of doc on cm_msg1, db_delete_key , db_send_change_record.

1.8.3-1

- **Custom page** Custom midas web page. Allow the makeup of personalized midas web page control with access to any ODB field.
- **Start page** Addition of **Edit on Start** description for web start command.
- **Midas Library** Addition of Midas Library function description (db_xxxx).
- **Pre-processor flags** List of pre-processor flags for midas tunning (appendix F: Midas build options and consideration).

2 Introduction

What is Midas, what can you do with it.

Names

... A few words...

Acquiring, collecting and analyzing data is the essence of mankind to satisfy his urge for understanding natural phenomena by comparing "real" events to his own symbolic representation. These fundamental steps paved human evolution and in the world of science they have been the keys to major steps forward in our understanding of nature. Until the last couple of decade's -when "Silicium" was still underground, the PPP protocol (Paper, Pencil and Patience) was the basic tool for this "unique" task. With the development of the "Central Processing Unit", data acquisition using computers wired to dedicated hardware instrumentation became available. This has allowed scientists to sit back and turn their minds towards finding solutions to problems such as "How do I analyze all these data?" Since the last decade or so when "connectivity" appeared to be a powerful word, the data acquisition system had to adapt itself to that new vocabulary.

Based on this sudden new technology, several successful systems using de-centralization of information have been developed. But the task is not simple! If the hardware is available, implementing a true distributed intelligence environment for a particular application requires that each node have full knowledge of the capability of all the other nodes. Complexity rises quickly and generalization of such systems is tough. Recently more pragmatic approaches emerged from all this, suggesting that central database information on a system may be more adequate, especially since processing and networking speed are not a "real" concern these days. MIDAS and its predecessor HIX may be counted part of the precursor packages in the field.

The old question: "How do we analyze all these data?" still remains and may have been the driving force behind this evolution :-).

->What is Midas?

2.1 What is Midas?

The Maximum Integrated Data Acquisition System (MIDAS) is a general-purpose system for event based data acquisition in small and medium scale physics experiments. It has been developed at the Paul Scherrer Institute (Switzerland) and at TRIUMF (Canada) between 1993 and 2000 (Release of Version 1.8.0).

Midas is based on a modular networking capability and a central database system. MIDAS consists of a C library and several applications. They run on many different operating systems such as UNIX like, Windows NT, VxWorks, VMS and MS-DOS. While the system is already in use in several laboratories, the development continues with addition of new features and tools. Current development involves RTLinux for either dedicated frontend or composite frontend and backend system.

For the newest status, check the MIDAS home page:

`http://midas.psi.ch` or `http://midas.triumf.ca`

->What can MIDAS do for you?

2.2 What can MIDAS do for you?

MIDAS has been designed for small and medium experiments. It can be used in distributed environments where one or more frontends are connected to the backend via Ethernet. The frontend might be an embedded system like a VME CPU running VxWorks or a PC running Windows NT or Linux. Data rates around 1MB/sec through standard Ethernet and 6.1MB/sec over Fast Ethernet can be achieved.

For small experiments and test setups the front-end program can run on the back-end computer thus eliminating the need of network transfer, presuming that the back-end computer has direct access to the hardware. Device drivers for common PC-CAMAC interfaces have been written for Windows NT and Linux. Drivers for PC-VME interfaces are commercially available for Windows NT.

For data analysis, users can write a complete analyzer or use the standard MIDAS analyzer which uses HBOOK routines for histogramming and PAW for histogram display.

The MIDAS package contains also a slow control system which can be used to control high voltage supplies, temperature control units etc. The slow control system is fully integrated in the main data acquisition and act as a front-end with particular built-in control mechanism. Slow control values can be written together with event data to tape.

->Components

3 Components

General information on the internal structure.

->Skip to Quick Start

Midas system is based on a modular scheme that allows scalability and flexibility. Each component operation is handled by a sub-set of functions. but all the components are grouped in a single library (libmidas.a, libmidas.so(UNIX), midas.dll(NT)).

The overall C-code is about 40'000 lines long and makes up over 160 functions (version 1.7.x). But from a user point of view only a subset of these routines are needed for most operations.

Each Midas component is briefly described below but throughout the documentation more detailed information will be given regarding each of their capabilities. All these components are available from the "off-the-shelf" package. Basic components such as the **Buffer manager**, **Online Database**, **Message System**, **Run Control** are by default operationals. The other needs to be enabled by the user simply by either starting an application or by activating the components through the **Online Database**.

- **Buffer Manager** Data flow and messages passing mechanism.
- **Message System** Specific Midas messages flow.
- **Online Database** Central information area.
- **Frontend** Acquisition code.
- **Midas Server** Remote access server (RPC server).
- **Data Logger** Data storage.
- **Analyzer** Data analyzer.
- **Run control** Data flow control.
- **Slow Control system** Device monitoring and control.
- **History system** Event history storage and retrival.
- **Alarm system** Overall system and user alarm.
- **Electronic Logbook** Online User Logbook.

Buffer Manager The "buffer manager" consists of a set of library functions for event collection and distribution. A buffer is a shared memory region in RAM, which can be accessed by several processes, called "clients". Processes sending events to a buffer are called "producers", processes reading events are called "consumers".

A buffer is organized as a FIFO (First-In-First-Out) memory. Consumers can specify which type of events they want to receive from a buffer. For this purpose each event contains a MIDAS header with an event ID and other pertinent information.

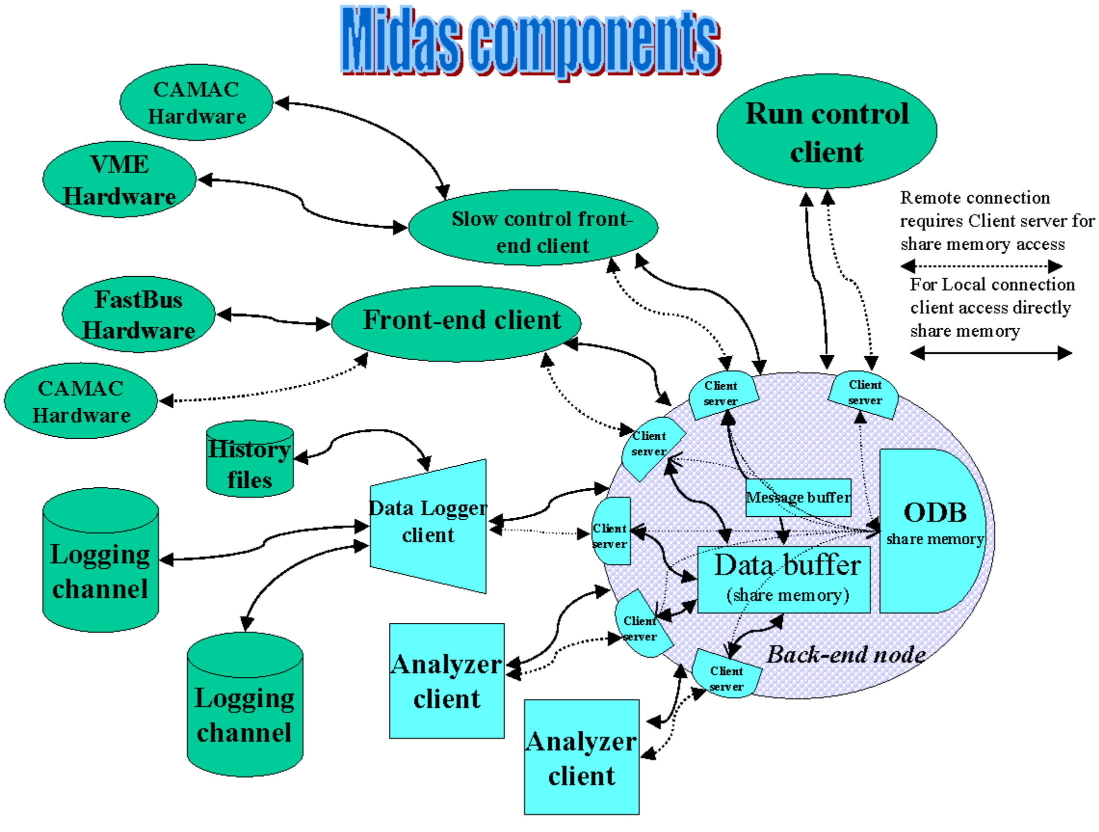


Figure 1: *Midas layout.*

Buffers can be accessed locally or remotely via the MIDAS server. The data throughput for a local configuration composed of one producer and two consumers is about 10MB/sec on a 200 MHz Pentium PC running Windows NT. In the case of remote access, the network may be the essential speed limitation element.

A common problem in DAQ systems is the possible crash of a client, like a user analyzer. This can cause the whole system to hang up and may require a restart of the DAQ inducing a lost of time and eventually precious data. In order to address this problem, a special watchdog scheme has been implemented. Each client attached to the buffer manager signal its presence periodically by storing a time stamp in the share memory. Every other client connected to the same buffer manager can then check if the other parties are still alive. If not, proper action is taken consisting in removing the dead client hooks from the system leaving the system in a working condition.

Message System Any client can produce status or error messages with a single call using the MIDAS library. These messages are then forwarded to any other clients who maybe susceptible to receive these messages as well as to a central log file system. The message system is based on the buffer manager scheme. A dedicated buffer is used to receive and distribute messages. Predefined message type contained in the Midas library covers most of the message requirement.

Online Database In a distributed DAQ environment configuration data is usually stored in several files on different computers. MIDAS uses a different approach. All relevant data for a particular experiment are stored in a central database called "Online Database" (ODB).

This database contains run parameters, logging channel information, condition parameters for front-ends and analyzers and slow control values as well as status and performance data.

The main advantage of this concept is that all programs participating in an experiment have full access to these data without having to contact different computers. The possible disadvantage could be the extra load put on the particular host serving the ODB.

The ODB is located completely in shared memory of the back-end computer. The access function to an element of the ODB has been optimized for speed. Measurement shows that up to 50,000 accesses per second local connection and around 500 accesses per second remotely over the MIDAS server can be obtained.

The ODB is hierarchically structured, similar to a file system, with directories and sub-directories. The data is stored in pairs of a key/data, similar to the Windows NT registry. Keys can be dynamically created and deleted. The data associated to a key can be of several type such as: byte, words, double words, float, strings, etc. or arrays of any of those. A key can also be a directory or a symbolic link (like on Unix).

The Midas library provides a complete set of functions to manage and operate on these keys. Furthermore any ODB client can register a hot-link between a local C-structure and a element of the ODB. Whenever a client (program) changes a value in this sub-tree, the C-structure automatically receives an update of the changed data. Additionally, a client can register a callback function which will be executed as soon as the hot-link's update has been received. For more information see ODB Structure.

Midas Server For remote access to a MIDAS experiment a remote procedure call (RPC) server is available. It uses an optimized MIDAS RPC scheme for improved access speed. The server can be started manually or via inetd (UNIX) or as a service under Windows NT. For each incoming connection it creates a new sub-process which serves this connection over a TCP link. The Midas server not only serves client connection to a given experiment, but takes the experiment name, as parameter meaning that only one Midas server is necessary to manage several experiments on the same node.

Frontend The *frontend* program refers to a task running on a particular computer which hs access to hardware equipment. Several *frontend* can be attached simultaneously to a given experiment. Each *frontend* can be composed of multiple *Equipment*.

- *Equipment* is a single or a collection of sub-task(s) meant to collect and regroup logically or physically data under a single and uniquely identified event.

This program is composed of a general framework, which is experiment independent, and a set of template routines for the user to be filled. This program will:

- Registers the given *Equipment(s)* list to the Midas system.
- Provides the mean of collecting the data from the hardware source defined in each equipment.
- Gathers these data in a known format (Fixed, Midas, Ybos) for each equipment.
- Sends these data to the buffer manager.
- Collects periodically statistic of the acquisition task and send it to the Online Database.

The frontend framework takes care of sending events to the buffer manager and optionally a copy to the ODB. A "Data cache " in the frontend and on the server side reduces the amount of network operations pushing the transfer speed closer to the physical limit of the network configuration.

The data collection in the frontend framework can be triggered by several mechanisms. Currently the frontend supports four different kind of event trigger:

- *Periodic events* Scheduled event based on a fixed time interval. They can be used to read information such as scaler values, temperatures etc.
- *Polled events:* Hardware trigger information read continuously which in turn if signal is asserted will trigger the equipment readout.
- *Interrupt events:* Generated by particular hardware device supporting interrupt mode.
- *LAM events:* Generated only when pre-defined LAM is asserted:
- *Slow Control events:* Special class of events that are used in the slow control system.

Each of these types of trigger can be enabled/activated for a particular experiment state, Transition State or a combination of any of them. Examples such as "read scaler event only when running" or "read periodic event if state is not paused and on all transitions" are possible.

Dedicated header and library files for hardware access to CAMAC, VME, Fastbus, GPIB and RS232 are part of Midas distribution set For more information see Frontend code.

Data Logger The data logger is a client usually running on the backend computer (can be running remotely but performance may suffer) receiving events from the buffer manager and saving them onto disk, tape or via FTP to a remote computer. It supports several parallels logging channels with individual event selection criteria. Data can currently be written in four different formats: *MIDAS binary*, *YBOS binary*, *ASCII* and *DUMP* (see Midas format, YBOS format).

Basic functionality of the logger includes:

- Run Control based on:
  - event limit
  - recorded byte limit
  - logging device full.
- Logging selection of particular event based on Event Identifier.
- Auto restart feature allowing logging of several runs of a given size without user intervention.
- Recording of ODB values to a so called History System
- Recording of the ODB to all or individual logging channel at the beginning and end of run state as well as to a separate disk file in a ASCII format.

For more information see ODB /Logger Tree.

Midas Analyzer As in the front-end section, the analyzer provided by Midas is a framework on which the user can develop his/her own application. This framework is based on HBOOK histogramming functions from the CERN library which is not included in the MIDAS distribution. This analyzer takes care of receiving events (a few lines of code are necessary to receive events from the buffer manager), initializes the HBOOK system and automatically books N-tuples for all events. It calls user routines for event analysis.

The analyzer is structured into "stages", where each stage analyzes a subset of the event data. Low level stages can perform ADC and TDC calibration, high level stages can calculate "physics" results. The same analyzer executable can be used to run online (receive events from the buffer manager) and off-line (read events from file). When running online, generated N-tuples are stored in a ring-buffer in shared memory. They can by analyzed with PAW immediately without stopping the run.

When running off-line, the analyzer can read MIDAS binary files, analyze the events, add calculated data for each event and produce a HBOOK RZ output file which can be read in by PAW later. The analyzer framework also supports analyzer parameters. It automatically maps C-structures used in the analyzer to ODB records via Hot link. To control the analyzer, only the values in the ODB have to be changed which get automatically propagated to the analyzer parameters. If analysis software has been already developed, Midas provides the functionality necessary to interface the analyzer code to the Midas data channel. Support for languages such as C, FORTRAN, PASCAL is available.

Run Control As mentioned earlier, the Online Database (ODB) contains all the pertinent information regarding an experiment. For that reason a run control program requires only to access the ODB. A basic program supplied in the package called ODBEdit provides a simple and safe mean for interacting with ODB. Through that program essentially all the flexibility of the ODB is available to the user's fingertips.

Three "Run State defines the state of Midas *Stopped*, *Paused*, *Running.* In order to change from one state to another, Midas provides four basic "Transition" function *Tr_Start*, *Tr_pause*, *Tr_resume*, *Tr_Stop.* During these transition periods, any Midas client register to receive notification of such message will be able to perform its task within the overall run control of the experiment.

In addition to these main transitions states, the *Tr_Start* and *Tr_Stop* have another 2 sub-states defined as *Tr_preStart*, *Tr_postStart*, *Tr_preStop*, *Tr_postStop.* These sub-states allows proper synchronization of data flow through the system.

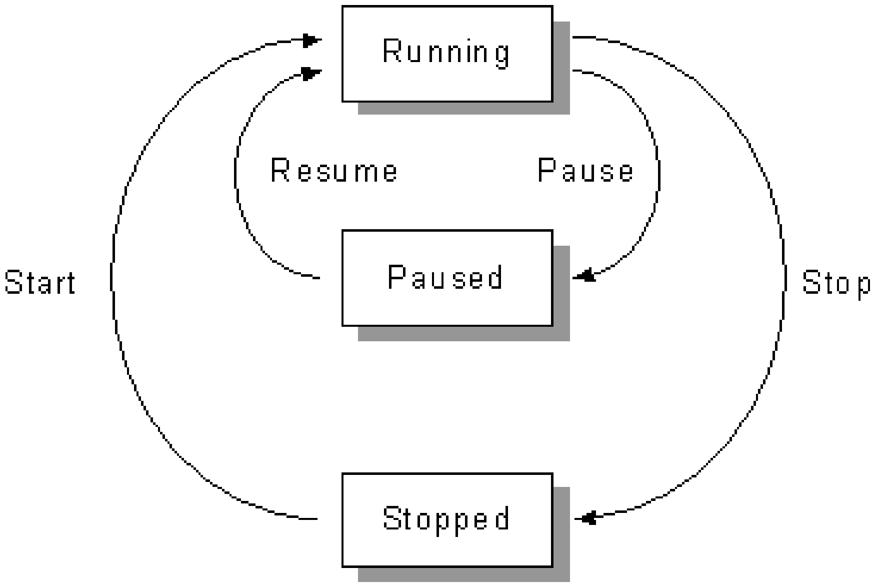


Figure 2: *Transitions*

Slow Control system The Slow control system is a special front-end equipment or program dedicated to the control of hardware module based on user parameters. It takes advantage of the Online Database and its "hot link" capability. Demand and measured values from slow control system equipment like high voltage power supplies or beam line magnets are stored directly in the ODB.

To control a device it is then enough to modify the demand values in the database. The modified value gets automatically propagated to the slow control system, which in turn uses specific device driver to control the particular hardware. Measured values from the hardware are periodically send back to the ODB to reflect the current status of the sub-system.

The Slow control system is organized in "Classes Driver ". Each Class driver refers to a particular set of functionality of that class i.e. High-Voltage, Temperature, General I/O, Magnet etc. The implementation of the device specific hardware is left to the hardware device driver. The current MIDAS distribution already has some device driver for general I/O and commercial High Voltage power supply system (see appendix B: Supported hardware. The necessary code composing the hardware device driver is kept simple by only requiring a "set channel value" and "read channel value". For the High Voltage class driver, a graphical user interface under Windows NT/9x is already available. It can set, load and print high voltages for any devices of that class. For more information see Slow Control.

History system The MIDAS history system is a recording function imbedded in the Midas logger. Parallel to its main data logging function of defined channels, the Midas logger can store slow control data and/or periodic events on disk file. Each history entry consists of the time stamp at which the event has occurred and the value[s] of the parameter to be recorded.

The activation of a recording is not controlled by the history function but by the actual equipment (see Frontend code). This permits a higher flexibility of the history system such as dynamic modification of the event structure without restarting the Midas logger. At any given time, data-over-time relation can be queried from the disk file through a Midas utility mhist task.

The history data extraction from the disk file is done using low level file function giving similar result as a standard database mechanism but with faster access time. For instance, a query of a value, which was written once every minute over a period of one week, is performed in a few seconds. For more information see History System, ODB /History Tree.

Alarm system The Midas alarm mechanism is a built-in feature of the Midas server. It acts upon the description of the required alarm set defined in the Online Database (ODB). Currently the internal alarms supports the following mechanism:

ODB value over fixed threshold At regular time interval, a pre-defined ODB value will be compared to a fixed value.

Midas client control During Run state transition, pre-defined Midas client name will be checked if currently present.

General C-code alarm setting Alarm C function permitting to issue user defined alarm.

The action triggered by the alarm is left to the user through the mean of running a detached script. But basic aalrm report is available such as:

- Logging the alarm message to the experiment log file.
- Sending a "Electronic Log message" (see Electronic logbook).
- Interrupt data acquisition.

For more information see Alarm System, ODB /Alarms Tree.

Electronic logbook The Electronic logbook is a feature which provide to the experimenter an alternative way of logging his/her own information related to the current experiment. This electronic logbook may supplement or complement the standard paper logbook and in the mean time allow "web publishing" of this information. Indeed the electronic logbook information is accessible from any web browser as long as the mhttpd task is running in the background of the system. For more information see Electronic Logbook, mhttpd task.

->Quick Start

4 Quick Start

The Howto for installation, quick test, troubleshooting.

Names

->Skip to Internal features

This section describes step-by-step the installation procedure of the Midas package on several platform as well as the procedure to run a demo sample experiment. In a second stage, the frontend or the analyzer can be moved to another computer to test the remote connection capability. Then finally the frontend and/or the analyzer can be modified to suit a real experiment.

The evaluation demo sample consists of:

1. A frontend code frontend.c, which doesn't access any hardware but internally, generates event structure using a random number generator.
2. An analyzer program analyzer.c, which receives these events and creates histograms in a shared memory region, which can be displayed with PAW.

But prior running the demo, the MIDAS software has to be installed. This process is composed of several steps, which can be summarized as follow:

1. Copy of the MIDAS library and header files to system include and library directories.
2. Build the Midas executable.
3. Copy (Install) the MIDAS logger and an editor for the online database (ODBEdit) to a system executable directory.
4. Install the Midas server to allow remote connection to the ODB and the to buffer manager.

5. Compile the sample frontend and sample analyzer.
6. Start the frontend and the analyzer on the same node.
7. Start the ODBEdit program to control the experiment.

4.1 UNIX installation

MIDAS version 1.x officially supports the following UNIX systems: Linux, Solaris, FreeBSD, OSF/1 (DEC UNIX) and Ultrix. Since it's highly portable, it can probably compiled on other system with minor adjustments (change of include files location).

New... rpm package is now available

See `ftp://midas.triumf.ca/pub/midas`

This rpm installation will replaces step 2 through 5 of the following hand installation.

Step 1 : Distribution Obtain the MIDAS distribution set via anonymous ftp from `ftp://midas.psi.ch/pub/midas` or `ftp://midas.triumf.ca/pub/midas` The UNIX version is called midas-x.xx.tar.gz where x.xx is the version number. Copy the distribution set to a directory of your choice. It is recommended to use <home>/midas where <home> is your home directory. For a public installation it is recommended to use /usr/local/ Then ftp the Z library which is called libz.a to the same directory.

Step 2 : Extraction Decompress and extract the distribution set by typing:

```
tar -zxvf midas-x.xx.tar.gz
```

If the GNU tar is not available the -z flag won't work. In this case the file has to be decompressed first and then untared:

```
gunzip midas-x.xx.tar.gz
tar -xvf midas-x.xx.tar
```

The gunzip program can be obtained from any GNU FTP site. For an index have a look at `http://www.gnu.org/order/ftp.html`. The extraction process creates a sub-directory <*.../midas-x.xx* where x.xx is again the version number. Following sub-directory structure is then created:

doc Documentation

drivers Hardware drivers

examples Example experiment

include C header files

src Source code

utils Utilities source code

vxworks Makefile for VxWorks

The PAW analyzer, which comes with the MIDAS distribution, can generate compressed output files. For that purpose it needs the Z library. Extract it from zlib the same way as the MIDAS distribution. It generates a directory zlib Then go to this directory and compile the Z library: cd zlib gmake In case of problem refer to the README file in the zlib directory.

Step 3 : Installation Compile and install the MIDAS system files. In the midas-x.xx directory, enter gmake If the GNU make program is not available, obtain it from the above-mentioned source. It is necessary for the automatic detection of the operating system it is running under. Alternatively, find all ifeq - endif combinations in the makefile and evaluate them manually since the standard make doesn't understand these statements. Then enter as the super-user:

```
make install
```

If you don't have super-user privileges, ask you system administrator to assist you in this step. The installation will copy the MIDAS library, MIDAS tools and header files into system directories, usually /usr/local/lib, /usr/local/bin and /usr/local/include. Edit the makefile if you want to change these directories. Make sure the CERN library is installed properly. The MIDAS analyzer needs libpacklib.a which is usually installed under /cern/pro/lib.

Step 4 : Compilation Compile the sample experiment. Create a working directory that contains the frontend and analyzer program (in the following example called online). Copy the sample experiment source files to that directory:

```
cd ~ (to go to your home directory)
mkdir online
cd online
cp <home>/midas/midas-x.xx/examples/experiment/* .
```

Edit the makefile in the working directory to select the correct operating system and the proper directories. Then build the example frontend and analyzer:

```
make
```

This creates two files frontend and analyzer. Set the environment variable MIDAS_DIR to point to the working directory: setenv MIDAS_DIR <home>/online This command could be added to your login shell (.login, .cshrc).

Step 5 : Networking If you plan to run the frontend later on another computer, the MIDAS server program has to be started. You can either start it manually by entering:

```
mserver
```

or via inetd add the following line to your */etc/services* file (Superuser required) :

```
midas 1175/tcp
```

Then add this single line to your */etc/inetd.conf* file :

```
midas stream tcp nowait root /usr/local/bin/mserver /usr/local/bin/mserver
```

This assumes that the mserver program is installed at */usr/local/bin/*. Send a hang-up signal to inetd to reload the modified configuration file:

```
ps -A | grep inetd
```

```
<note the process id>

kill -HUP <id>
```

Each time you make a remote access to this computer, inetd will now start a copy of mserver.

To connect to different experiments, the server has to know in which directories and under which user names the experiments are running. For this purpose a list of all experiments running on a machine has to be defined in a file called exptab. This file is located under /etc and contains one line for each experiment composed of the experiment name, the directory and the user name. Create this file with an editor containing the following line to define an experiment called "Sample": Sample <home>/online <your name> where Sample is the experiment name, <home> is your home directory (like /usr/users/john) and <your name> is your login name.

4.2 Windows NT installation

Under Windows NT ...

Step 1 : Distribution Obtain the MIDAS distribution set via anonymous ftp from `ftp://midas.psi.ch/pub/midas` The NT version is called midas-x.xx.exe where x.xx is the version number. Copy the distribution set to a directory of your choice. It is recommended to use c:\. If you use a different drive than c: substitute the drive letter in the following instructions by the one that you use. Then ftp the Z library, which is called zlib104.zip to c:\zlib

Step 2 : Tree Extract the distribution set by executing it:

```
gunzip midas-x.xx
```

The extraction process creates a sub-directory \midas-x.xx where x.xx is again the version number. Following sub-directory structure is then created:

doc Documentation

drivers Hardware drivers

examples Example experiment

include C header files

src Source code

utils Utilities source code

nt Makefiles for Visual C++

nt\lib Libraries for Windows NT

nt\bin Program binaries for Windows NT

nt\directio DirectIO kernel driver for hardware access under Windows NT

nt\service Files needed to install the Midas Server as a NT Service

Step 3 : Extraction Install the MIDAS system files. In the midas-x.xx directory, enter

```
install
```

This will install the MIDAS library under c:\midas\nt\lib, the MIDAS programs under c:\midas\nt\bin and the include files under c:\midas\include. Edit the file install.bat if you want to change these directories. Set the "path" environment variable to the MIDAS executables at c:\midas\nt\bin.

To do so:

- right-click on the "My Computer" icon on the desktop.
- Select "Properties" from the menu.
- On the dialog box, click on the "Environment" tab.
- Under "System variables", search and select "Path".
- On the "value" field, go to the end of the line and add the MIDAS executable directory. The current path should be similar to the following line: %SystemRoot%\system32;%SystemRoot%;c:\midas\bin.
- Press the "Set" than the "OK" button.
- Make sure the CERN library is installed properly. The MIDAS analyzer needs packlib.lib that is usually installed under c:\cern\lib.

Step 4 : Compilation Compile the sample experiment. Create a working directory that contains the frontend and analyzer program (in the following example called online). Copy the sample experiment source files to that directory:

```
cd c:\
mkdir online
copy c:\midas-x.xx\examples\experiment\*.*
```

Build the example frontend and analyzer (this assumes that you have installed the command line tools of Visual C++):

```
nmake -f makefile.nt
```

This creates the two files frontend.exe and analyzer.exe. Set the environment variable MIDAS_DIR to point to the working directory. Open the "Environment" dialog box as under Step 3. Then enter "MIDAS_DIR" in the "Variable" box and "c:\online" in the "Value" field (without quotation marks). Press the "Set" and "OK" buttons.

Step 5 : Networking If you plan to run the frontend later on another computer, the MIDAS server program has to be started. You can either start it manually by entering:

```
mserver
```

Or as a Windows NT service, you can configure it by running the install.bat file. Make sure you have administration privileges on the PC:

```
c:\midas-x.xx\nt\service\install
```

To connect to different experiments, the server has to know in which directories and under which user names the experiments are running. For this purpose a list of all experiments running on a machine has to be defined in a file called exptab. This file is located under C:\winnt\system32 and contains one line for each experiment describing the experiment name and the directory. Create this file with an editor containing the following line:

```
sample c:\online
```

This defines the experiment Sample attached to the directory c:\online.

4.3 vxWorks installation

Under VxWorks, only the frontend code is supported by Midas. This means that tools such as ”odbedit”, ”mlogger” etc, are not being compiled under VxWorks. Only the code doing the actual data acquisition (frontend acquisition code, Slow control code) will be generated by the makefile provided in the package. In the other hand, your host computer should have the capability to generate code specific for your hardware running VxWorks. Installation of VxWorks has been done so far only under UNIX like host machine. The cross compiler has to be then available for the following operations.

Step 1 : Distribution Go to the midas directory tree under vxworks.

```
cd .../midas-x.xx/vxworks
```

Step 2 : VxWorks Version Pick the proper makefile.xxx which suit your hardware and copy it to makefile.

```
Wed> ls -l
total 20
drwxr-xr-x   4 midas    midas        1024 Mar  9 14:45 ./
drwxr-xr-x  12 midas    midas        1024 Apr 18 13:34 ../
drwxr-xr-x   2 midas    midas        1024 Mar  7 14:53 CVS/
-rwxr-xr-x   1 midas    midas        4778 Mar  7 14:53 makefile.68k_psi*
-rwxr-xr-x   1 midas    midas        5616 Mar  7 14:53 makefile.68k_tri*
-rwxr-xr-x   1 midas    midas        4670 Mar  7 14:53 makefile.ppc_tri*
drwxr-xr-x   2 midas    midas        1024 Apr 17 09:56 ppcobj/

Wed> cp makefile.ppc_tri Makefile
```

Step 3 : X-Compiler path Modify the makefile to match your VxWorks installation distribution. (The example here refers to /vw/include for VxWorks system include files).

```
Wed> more makefile.ppc_tri
# directories
#
# System directories
SYSINC_DIR = /vxworks-ppc/include

<edit> makefile SYSINC_DIR =
```

Step 4 : Building Build the VxWorks Midas library as well as the Midas frontend code skeleton. This operation should create a *./ppcobj* with the generated files (Midas library, drivers, etc)

```
Wed> make
/vxworks-ppc/bin/ccppc -c -g -finline-functions -Winline -I../include -I/vxworks-ppc/include -ansi
  -fstrength-reduce -fkeep-inline-functions -DOS_VXWORKS  -DPPCxxx -o ppcobj/midas.o ../src/midas.c
/vxworks-ppc/bin/ccppc -c -g -finline-functions -Winline -I../include -I/vxworks-ppc/include -ansi
  -fstrength-reduce -fkeep-inline-functions -DOS_VXWORKS  -DPPCxxx -o ppcobj/system.o ../src/system.c
/vxworks-ppc/bin/ccppc -c -g -finline-functions -Winline -I../include -I/vxworks-ppc/include -ansi
  -fstrength-reduce -fkeep-inline-functions -DOS_VXWORKS  -DPPCxxx -o ppcobj/mrpc.o ../src/mrpc.c
```

```
/vxworks-ppc/bin/ccppc -c -g -finline-functions -Winline -I../include -I/vxworks-ppc/include -ansi
 -fstrength-reduce -fkeep-inline-functions -DOS_VXWORKS  -DPPCxxx -o ppcobj/odb.o ../src/odb.c
/vxworks-ppc/bin/ccppc -c -g -finline-functions -Winline -I../include -I/vxworks-ppc/include -ansi
 -fstrength-reduce -fkeep-inline-functions -DOS_VXWORKS  -DPPCxxx -o ppcobj/ybos.o ../src/ybos.c
rm -f ppcobj/libmidas.o
/vxworks-ppc/bin/ldppc -o ppcobj/libmidas.o -r ppcobj/midas.o ppcobj/system.o ppcobj/mrpc.o
 ppcobj/odb.o ppcobj/ybos.o
/vxworks-ppc/bin/ccppc -g -finline-functions -Winline -I../include -I/vxworks-ppc/include -ansi
-fstrength-reduce -fkeep-inline-functions -DOS_VXWORKS  -DPPCxxx  -c -o ppcobj/esone.o
../drivers/bus/esone.c
```

Step 5 : Tree template Go to your application directory and copy the frontend code example and the corresponding makefile.

```
cd ~ (to go to your home directory)
mkdir online
cd online
cp .../midas-x.xx/examples/experiment/makefile.vxw makefile
cp .../midas-x.xx/examples/experiment/frontend.c .
```

Step 6 : Building Modify the makefile to fit your VxWorks configuration and build the frontend code under VxWorks. This last operation if successful will create a *./ppcobj* directory with a file called *mfe.o*. This file will have to be loaded into the VxWorks machine through any mean provided by the VxWorks system.

```
<edit> Makefile
gmake UFE=frontend
```

Step 7 : Running If the VxWorks kernel has been built with NFS support, through a telnet session or through the console, the frontend code can be started in the following way:

```
cd "<nfs pointing to .../midas-x-xx/vxworks/ppcobj>"
> ld < libmidas.o
> cd "<nfs pointing to your host online dir>/ppcobj"
> ld < mfe.o
> mfe ("<Midas_host_name>","<Midas_expt_name>")
```

Or as a background task:

```
> taskSpawn "mfe", 121, spTaskOptions, 100000, mfe,"<Midas_host_name>","<Midas_expt_name>"
```

4.4 DOS installation

The full Midas system can not be installed under MS-DOS due OS resources limitation. But in the other hand, MS-DOS is suitable to run a single Midas application such as a single client. Usually this client is of the type of a frontend with hardware data collection capability. In order to connect this MS-DOS client to the remote Midas system, a TCP/IP stack has to be primarily installed and running. MIDAS has been tested with PC/TCP from FTPSoftware. Special files needed for compilation of MIDAS client under MS-DOS are included in the msdos directory. It contains project files (*.prj) for Turbo C++ and Borland C++. The include directory contains some files which replace the standard include files (usually under \bc\include). Care has to be taken to guaranty that the msdos\include directory precedes the \bc\incude directory in the include files path (usually specified under Options/Directories in Borland C++). Instead of creating a MIDAS library, client programs contain the files midas.c, system.c, odb.c and mrpc.c directly in their project files. The PC/TCP library lpctcp.lib, which contains the socket routines, has to be linked to the executable. Only the large memory model is supported. Using the operating system MS-DOS today may seems old fashioned. But it has to be considered that a MIDAS frontend does not need a multi-process operating system because it contains its own scheduler. A multi-process operating system only puts additional load on the computer due to context switching which is not necessary in that case. One disadvantage of using MS-DOS could be the lack of 32bit support especially for the floating-point operations, which could be necessary within the frontend code for event filtering. The main disadvantage of using MS-DOS is the fact that programs cannot be loaded remotely like under VxWorks. To overcome this problem, the following scheme has been successfully used. The frontend program is compiled on the back-end PC running Windows NT in the counting house that runs the freeware NFS server SOSS . The frontend computer (MS-DOS) mounts the directory that contains the frontend executable program (usually c:\online). It then starts the frontend in a loop. This can be done in the autoexec.bat file:

```
...
<mount back-end c: as drive n:>
n:
cd online
:loop
frontend -h <host name>
goto loop
```

If the frontend program needs to be modified, it is recompiled on the NT computer, which replaces frontend.exe. Then the frontend is stopped with the ODBEdit command >sh frontend. The loop in autoexec.bat restarts then the new frontend is automatically reloaded and started.

4.5 Local connection test...

After successful installation of the Midas package, the necessary steps for running the sample experiment consists on starting the frontend, the logger and the analyzer in three different windows from your working directory (called "online" in the above installation):

```
Wed>frontend
Wed>analyzer
Wed>mlogger
```

These three programs will connect to the local online database and buffer manager. In the frontend window will display a status information (stopped) and statistics about "trigger events" and "scaler events". The trigger events will be generated as fast as possible (in a real experiment there will be a hardware trigger) while the scaler event will be generated once every ten second. To control the experiment, start the run control program (ODBEdit).

```
odbedit
[local:sample:S]/>
```

This program allows access to the ODB and is used for run control. Try these commands:

Show all connected clients. Shows the logger, the frontend, the analyzer, the ODBEdit itself.

```
[local:sample:S]/>Scl
```

Go to ”runinfo” sub-directory.

```
[local:sample:S]/>cd runinfo
```

Show runinfo values. Shows undefined run number.

```
[local:sample:S]/>ls
 State                         1
 Online Mode                   1
 Run number                    4
 Transition in progress        0
 Requested transition          0
 Start time                    Wed Feb 23 17:50:35 2000
 Start time binary             951357035
 Stop time                     Wed Feb 23 17:51:12 2000
 Stop time binary              951357072
```

Change the run number.

```
 [local:sample:S]/>set "run number" 123
```

Start a run. The frontend window shows run state ”Running” and statistics number changing.

```
[local:sample:S]/>start
 Run number [124]:
 Are the above parameters correct? ([y]/n/q):
 Starting run #124
```

Stop run.

```
[local:sample:S]/>stop
 [local:sample:R]/Runinfo>stop
 13:05:16 [ODBEdit] Run #124 stopped
```

Show available commands.

```
[local:sample:S]/>help
Database commands ([] are options, <> are placeholders):

alarm                   - reset all alarms
cd <dir>                - change current directory
chat                    - enter chat mode
chmod <mode> <key>      - change access mode of a key
                          1=read | 2=write | 4=delete
cleanup                 - delete hanging clients
copy <src> <dest>       - copy a subtree to a new location
create <type> <key>     - create a key of a certain type
create <type> <key>[n]  - create an array of size [n]
...
```

Note that ODBEdit supports "UNIX tcsh" command line editing. Use the tabulator to complete a directory name and the arrow keys to recall previous commands.

4.6 Remote connection test...

So far the demo sample experiment has demonstrate the basic operation of the Midas system in a local environment (see Local connection). Before switching to a "real" experiment with remote frontend, a secondary demo test can be done to verify the remote capability of Midas and to get you familiarize with it.

Move the frontend program (frontend.c) to another computer connected through network to the back-end (analyzer) computer. This can be a VME CPU, a PC running Linux or Windows NT. Or Cross-compile the frontend program (frontend.c) and move the image to the remote computer if running the VxWorks or MS-DOS. Start the Midas server by hand (back-end computer) if configuration through the inetd has not been used (valid only for UNIX machine). <system_dir>/mserver& On the frontend computer, start the frontend task specifying the host name (back-end computer) as well as the experiment name using the -h and -e flags. The experiment name has to be specified only if more than one experiment is defined in the exptab file on the back-end computer. frontend -h <host-name> [-e Sample]

Running ODBEdit on either computer will allow you to control the run by issuing command as described earlier (start, stop, etc.) After successfully passing this remote test, the next step would be the setup of a hardware trigger. Refer to appendix D: Computer Busy Logic to learn how to setup a "computer busy" logic. A hardware trigger signal has to be made available to the frontend computer through an I/O unit or through an interrupt signal (in CAMAC also called look-at-me signal or LAM) in order to activate the proper software sequence for data collection. Since the frontend framework is hardware independent, all hardware accesses have to be done in the user part of the frontend. It is recommended to use first the polling mode. For that purpose code has to be inserted in the function poll_trigger_event() for performing constant check for the presence of the hardware signal. The readout code read_trigger_event() which will be called upon that later condition, will collect the predefined data. The last instruction in this routine should be the rearming of the hardware (Clear, reset trigger signal) to enable new event. FNoted that if the flag RO_ODB is defined in the equipment definition for the trigger event, a copy of the trigger event is sent to the ODB under /Equipment/Trigger/Variables periodically. The contents of the event can be checked with ODBEdit. Once the frontend is running correctly, the user analyzer can be modified to suit the needs of the experiment. This chapter describes the usage of MIDAS in more detail. It explains how to move from the sample experiment to a "real" experiment. Details of the frontend, the logger, the analyzer and the run control are covered.

4.7 Defining an Experiment

Every experiment under Midas has its own ODB. In order to differentiate them, an experiment name and directory are assigned to each experiment. This allows several experiments to run concurrently on the same host using a common Midas installation. Whenever a program participating in an experiment is started, the experiment name can be specified as a command line argument or as an environment variable. A list of all possible running experiments on a given machine is

kept in the file **exptab** which is located under /etc (UNIX) or
winnt
system32 (Windows NT).

Different experiments can run under the same user name or under different user names. The Midas system supports multiple experiment running contemporary on the same computer. Even though it may not be efficient, this capability makes sense when the experiments are simple detector lab setups which shared hardware resources for data collection. In order to support this feature, Midas requires a uniquely identified set of parameter for each experiment that is used to define the location of the Online Database.

These parameters are: *The experiment name* and *experiment directory*

Several means for declaration of those parameters are available: *exptab* file for remote access to this node. Each computer has to have an experiment list stored in a system wide area to allow remote clients to locate the experiment directory. The file has a unique name exptab and is composed of two entries per line describing the experiment. Since the exptab file is kept in a system location, one has to have super-user privileges to modify it. If you don't have it, ask your system administrator to add your experiment to the *exptab* file for you (*/etc/exptab* for Unix OS like).

```
Sample /usr/users/mydir/online LabTest1 /home/alllab/labt1
```

System environment for global setting within a user area:

- *MIDAS_DIR* points to the directory where the experiment ODB resides. This is used only for local connection i.e. from within the same computer. That directory uniquely defines the experiment. setenv MIDAS_DIR <my_dir>
- *MIDAS_SERVER_HOST* defines the name of the host containing the experiment database. This environment variable sets the default Midas host name for the remote connection. setenv MIDAS_SERVER_HOST myhost
- *MIDAS_EXPT_NAME* defines the default name of the experiment to connect to. setenv MIDAS_EXPT_NAME sample
- *Application argument* Every Midas application has two valid arguments to specify the host and experiment to contact. -h <host name>: supersedes the default system environment MIDAS_SERVER_HOST if defined. -e <experiment name> : supersedes the default system environment MIDAS_EXPT_NAME if defined. Odbedit odbedit -e sample odbedit -e sample -h myhost

4.8 Defining a frontend

The frontend code (the part doing the actual data acquisition form the hardware) is defined in a C-code file such as: midas-x.xx/examples/experiment/frontend.c This example generates MIDAS events under two different equipment's i.e.: Trigger and Scaler. After compilation, the image can be started by typing: > frontend

4.9 Defining an Analyzer

Names

Several ways are available to the user for getting an event analyzer set up: Using the standard MIDAS analyzer together with PAW as described in (???) Writing a complete new analyzer from scratch using a template such as the /examples/lowlevel/consume.c. Adapting an existing analyzer package (FORTRAN, PASCAL, C) and turning it into a MIDAS consumer. In all the cases, the software approach is always the same and uses the same tools for basic event retrieval. The following section describes the second option, which is the most complete example.

4.9.1 Creating an Analyzer

Receiving events from MIDAS is relatively simple and consists of three distinct steps:

- Connecting to an experiment
- Requesting events
- Supplying a callback routine which receives these events

The code for these steps can look like this:

```
#include <stdio.h>
#include "midas.h"


void process_event(HNDLE hbuf, HNDLE request_id,
                   EVENT_HEADER *pheader, void *pevent)
{
  printf("Received event #%d\\r",
         pheader->serial_number);
}

main() {
INT status, request_id;
HNDLE hbuf;

  status = cm_connect_experiment("", "sample",
             "Simple Analyzer", NULL);
  if (status != CM_SUCCESS)
19     return 1;
20
21   bm_open_buffer("SYSTEM", EVENT_BUFFER_SIZE,
22                  &hbuf);
23   bm_request_event(hbuf, 1, TRIGGER_ALL,
24     GET_ALL, request_id, process_event);
25
26   do {
27     status = cm_yield(1000);
28   } while (status != RPC_SHUTDOWN &&
29            status != SS_ABORT);
30
31   cm_disconnect_experiment();
32
33   return 0;
34 }
```

This program connects to the experiment "Sample" on the local computer. It opens event buffer "SYSTEM" (which is the default event buffer) and requests all events with ID=1. When an event of that type is received, the serial number of this event is printed. The lines have following meaning:

Line 2: This is the standard MIDAS header file. Make sure the compiler knows its location (usually add the -I/usr/local/include flag under UNIX and /I
midas
include under Windows NT). Line 5/6: This is the callback routine receiving events. pheader points to the event header (for a description of its structure refer to Appendix B: MIDAS Event Format) and pevent points to the data area of the event. Line 16-19: These lines connect to the experiment "Sample" on the local computer. The declaration of the cm_connect_experiment() function is: INT cm_connect_experiment(char *host_name, char *exp_name, char *client_name, void (*func)(char*)); Where: host_name is the IP name of the host to connect to (empty if to connect to the local host). exp_name is the experiment name, client_name is the name of the calling program as it can be seen by others and the func routine can be supplied to read in a password if security has been enabled. Line 21/22: The event buffer "SYSTEM" is opened. If successful, a non-zero handle for this buffer is returned in hbuf, which can be used, in the following calls. Line 23/24: This call requests a certain type of events. The syntax is: INT bm_request_event(INT buffer_handle, short int event_id, short int trigger_mask, INT sampling_type, INT *request_id, void (*func)(HNDLE,HNDLE,EVENT_HEADER*,void*));

The buffer handle is the one obtained from bm_open_buffer. The event_id and trigger_mask select a certain type of event. Only events of that type are received. The event ID is directly compared with the one contained in the event header of each event, while the trigger mask is bit-wise compared with the trigger mask in the event.

The event is received only if the event ID's matches and the trigger masks have at least one bit in common. To receive all events from the system buffer, values of EVENTID_ALL and TRIGGER_ALL can be specified. The sampling type specifies events should be received. Two possible types are currently available:

- *GET_ALL* Specify that all events from a given type are to be received. If the analyzer processes these events slower than the frontend produces them, the frontend gets automatically blocked periodically to reduce the data rate.
- *GET_SOME* Tells the Midas buffer system to provide events to the analyzer on request. In other word, the analyzer will get events without holding the acquisition. If the analyzer

processes events faster than the frontend produces them, the analyzer receives all events. If the analyzer is slower than the frontend, events are skipped.

On some systems these different modes are called "May process" and "Must process". Line 26-29: This is the standard main loop in a MIDAS program. Since the system is event based, a central routine has to be called periodically to receive and distribute events. If an event is received which matches the request, the according callback routine (in this case process_event) is dispatched. The loop can be broken under an error condition such as SS_ABORT (network error) or exit request RPC_SHUTDOWN by other program. For this last case ODBEdit has a dedicated command to shutdown Midas client

```
[local]/>scl (to show all clients)
Name Host
Simple Analyzer pc810
ODBEdit pc810
[local]/>shutdown "Simple Analyzer"
11:01:02 [Simple Analyzer] Program Simple Analyzer on host pc810 stopped
```

The parameter to cm_yield is a time-out in milliseconds. It refers to a time after which the processing control is returned to the main task allowing user code to be activated in a regular time interval basis. A typical example would be the user command keystrokes handling. Line 31: Disconnects from the experiment. It is important to call this routine before a program stops to allow proper network disconnection. After successful compilation of the code, the linking procedure requires particular libraries: The MIDAS library which is installed in /usr/local/lib/libmidas.a under UNIX and in c:
midas
nt
lib
midas.lib under NT. The NT library is a shared library where midas.lib only contains references to the functions. The real code is contained in
winnt
system32
midas.dll, which is needed by the application during run time. If the application is moved to another PC where MIDAS is not installed, it is enough to move midas.dll to the Windows directory on the new PC.

4.9.2 Debugging an Analyzer

A common problem in DAQ is a crashing analyzer that has a GET_ALL request and therefore blocks the whole system. To overcome this problem, a watchdog system has been implemented. Every client attached to a buffer periodically signals that it is alive by writing the current time to a client region in the shared memory. This is done using the alarm()() function under UNIX and a timer under Windows NT. Whenever the process crashes or is killed, the other clients see that the crashed client does not update its time any more. In this case all requests from that client are removed from that buffer automatically to release blocked producers. Stopping the analyzer via Control-C or the kill command while the ODBEdit is still running can test this scheme. After a time-out of 10 seconds ODBEdit considers the analyzer to be dead and removes it from the ODB and all buffers. While the watchdog scheme works fine for normal operation, it can cause problems when running the analyzer inside a debugger. When the debugger stops the program at a break point, signals (UNIX) and timers (NT) are disabled. Since the application cannot signal that it is still alive, it gets removed from the ODB and all buffers after 10 seconds. Indicating to the other applications that the "alive" status of this client should not be checked can solve this problem. This can be done by calling the function cm_set_watchdog_params() at the begin of a program. The time-out of zero (second parameter) tells other applications to skip the "alive" test on this application

cm_set_watchdog_params(FALSE, 0);

At last it may also happen that a program which has its alive tests disabled crashes. In this case the producers involved in the experiment might be blocked forever. The only remedy in this case is to manually remove the crashed application with the ODBEdit command cleanup. This command removes all dead clients without checking their time-out value.

4.9.3 Byte swapping

Care has to be taken if the byte ordering (big endian and little endian) differs between frontend and back-end. This can happen for example when the frontend is a Motorola CPU (68k or PowerPC) and the back-end a DEC Alpha or Intel PC. The MIDAS RPC layer takes care of byte swapping automatically for all functions called remotely, but not for the data part of events since this is user defined. This problem has been resolved by leaving intact the byte order of the data within the frontend and moves that operation into the logger. The data then are swapped before going to the logging channel on the back-end computer. This choice has been made in order to reduce frontend CPU load and therefore decrease the data acquisition dead time. The online analyzer will also have to swap the event content on the fly, but will have little impact on the acquisition as it acquire event usually in GET_SOME mode. To swap individual words or long-words, the C macros WORD_SWAP() (16 bit), DWORD_SWAP() (32 bit) and QWORD_SWAP() (64 bit or double float) are included in the header file msystem.h. For events in MIDAS or YBOS format, functions are included in the MIDAS library, which swap a complete event:

- ybos_swap_event(DWORD *pevent): for YBOS events.
- bk_swap(void *pevent): for MIDAS events. Check function call bank or event

The pointer pevent points to the data area of the event. All banks are scanned and swapped according to the data type content.

4.9.4 Writing a FORTRAN Analyzer

An analyzer can also be written in FORTRAN. The only problem is that the MIDAS library is written in C, which uses different parameter passing schemes than FORTRAN. Therefore a set of "wrapper" routines have been written. These routines are contained in fmidas.c and export some of the MIDAS library functions in a way that they can be called easily from FORTRAN. The file fmidas.c needs to be compiled and linked to a FORTRAN analyzer. A additional FORTRAN include file midas.inc has been written which defines MIDAS functions and constants. By using these files, a FORTRAN analyzer looks very similar to the C analyzer:

```
  SUBROUTINE PROCESS_EVENT (HBUF, HREQ, HEADER, EVENT)
  INTEGER*4 HBUF, HREQ, HEADER(4), EVENT(*)
    WRITE (*,*) HEADER(2)
  END

  PROGRAM TEST
  INCLUDE 'midas.inc'
  INTEGER*4 STATUS,REQUEST_ID
  INTEGER*4 HBUF

  STATUS = CM_CONNECT_EXPERIMENT('','sample',
+            'Fortran Analyzer')
  IF (STATUS .NE. CM_SUCCESS) STOP


  CALL BM_OPEN_BUFFER('SYSTEM', EVENT_BUFFER_SIZE,
+                     HBUF)
  CALL BM_REQUEST_EVENT(HBUF, 1, TRIGGER_ALL,
+                       GET_ALL, REQUEST_ID)

  DO WHILE (STATUS .NE. RPC_SHUTDOWN .AND.
+          STATUS .NE. SS_ABORT)
    STATUS = CM_YIELD(1000)
  END DO

  CALL CM_DISCONNECT_EXPERIMENT()

  END
```

The only difference is the callback routine PROCESS_EVENT() which cannot be passed to BM_REQUEST_EVENT() as a parameter like in C. Therefore the function PROCESS_EVENT() is called directly from the system. Note that this function must be present in the FORTRAN program even if no events are requested. Otherwise one would get a linker error.

4.10 Running an Experiment

Names

A whole run can be controlled through the ODBEdit program. It contains functions to start/stop/pause and resume runs. Since all parameters, configuration and status information are contained in the ODB, the whole experiment can be monitored and controlled by accessing these data. The ODB is structured as a file system under which sub-trees are constructed. Each sub-tree can contain parameters, variables, information or other sub-tree that are specific to the declaration of the original sub-tree. Several sub-tree's name are reserved by the system in order to provide basic organization of the Online Database.

4.10.1 Starting/Stopping a Run

Run transitions can be made with the ODBEdit commands start, stop, pause and resume. When starting a run, the current run number is automatically incremented by one. The user is asked to confirm the run number. By pressing return the proposed value is accepted:

```
[local]/>start
Run number [2]:<return to accept "2">
Are the above parameters correct? ([y]/n/q):<return to accept "y">
Starting run #2
Run #2 started
The current run information is kept in the ODB under /Runinfo:
[local]/>cd /Runinfo
[local]/Runinfo>ls
State                          3
Run number                     2
Transition in progress         0
Start Time             Thu Jan 15 11:21:19 1998
Start Time binary      884866879
Stop Time              Thu Jan 15 11:21:12 1998
Stop Time binary       884866579
```

The State is the current run state: 1 for stopped, 2 for paused and 3 for running. The Start Time binary is in standard UNIX format (seconds since 1.1.1970) and can be used by programs to calculate the duration of the current run. The Stop Time is the time when the previous run has been stopped. When a logger writes to Exabyte tapes, starting and stopping of a run can take up to 60 seconds since mounting Exabyte tapes and writing EOF marks is very slow. Therefore it can happen that the run is stopping while someone on another computer tries to restart already the next run. Since this would confuse the system, a transition lock has been implemented. During a transition the Transition in progress flag from above is set to 1 causing all other transition requests to be blocked. If ODBEdit is killed during a transition, this flag might not be reset to 0 thus blocking all future transition request falsely. In this case the flag can be reset manually with:

[local]/>set ”/Runinfo/Transition in progress” 0

The MIDAS State model is flat, there is no hierarchy of components and sub-components like in other systems. Each client can register to receive a transition. The program making the transition (usually ODBEdit) then contacts all registered clients and their transition callback routine is executed via RPC.

4.10.2 Monitoring a Run

Checking the statistics and status data in the ODB can monitor an experiment. A utility called mstat comes with MIDAS that displays the most interesting data:

The first section displays general run information, the second section displays the statistics for the different equipment, the third section logging information and the last section shows a list of all active clients. This information can be found in the ODB in the directories /Runinfo,

/Equipment/<equipment name>/Statistics, /Logger and /System/Clients. If the frontend sends event copies to the ODB (via the RO_ODB flag), they can be checked under the Variables tree:

```
[local]/>cd /Equipment/Trigger/Variables
[local]Variables>ls
ADC0
                                 1378
                                 980
                                 797
                                 398
                                 3271
                                 1322
                                 1227
                                 1725
TDC0
                                 3289
                                 2911
                                 3883
                                 4065
                                 2784
                                 3626
                                 1130
                                 2691
```

As can be seen the trigger event contains two banks named ADC0 and TDC0 that contain eight values produced by the frontend of the sample experiment.

4.11 Troubleshootting

Midas is neither a ”bug free” nor a ”transparent” package. This documentation is an attempt to shed some light on the inside of the system and to make it a little bit more intuitive. But troubleshooting will remain especially hard when the user is not well familiar or has little experience with the system. The main appearance of a problem with Midas will be when either the frontend is not showing as client to the experiment, the Online Database is not accessible from the keyboard or when the data logging to a tape device (usually) is no longer working. In these cases, required action from the user is necessary to restore the operation of the system.

4.12 Crashed Frontend

If the frontend crashes due to malfunction code in the event readout routine, it normally disconnects the TCP connection from the back-end. In this case it can be simply restarted. If the run is still going on, the frontend automatically enters the running state and continues sending events. The only problem might be that the serial number of the events starts again at one.

If the frontend runs on an operating system which does not gracefully close down the TCP connection (this is true for VxWorks and MS-DOS), the back-end still assumes that the frontend is alive. If the frontend is then restarted, it won't be able to open the statistics record since it can only be open by one frontend. To solve this problem, watchdog messages are sent over the network between the back-end and the frontend to insure that the frontend is alive. If the frontend does not respond after 30 seconds, the connection is aborted by the back-end automatically. Therefore the best strategy to recover from a crashed frontend is to wait 30 seconds until the watchdog period expires. Then the frontend can be restarted safely.

4.13 Corrupted ODB

The ODB contents is kept in shared memory. When all clients exit, it is written to a disk file (.ODB.SHM) where it is loaded next time a new client starts. This ensures persistency even if the computer gets rebooted in between. Since the ODB is mapped to the address space of all local clients, they might overwrite some of the ODB contents if they contain malfunctioning code. This is especially true for user written analyzers which write over array boundaries etc. If this happens, the ODB contents might get corrupted. This leads to situation where ODBEdit displays strange contents or even crashes. In worst case ODBEdit won't start any more with an error "ODB full" or similar. To solve the problem, the disk file ODB.SHM (.ODB.SHM under UNIX) must be deleted. If ODBEdit is then restarted, the ODB is empty except the /System entry which is created when ODBEdit starts. To restore the previous ODB contents, the file last.odb can be loaded into the ODB. This ASCII ODB dump file gets written after each run in the data directory by the logger. This file represents the exact ODB state after the last run. The contents of this file might be loaded back from this file with ODBEdit:

cd <data directory> odbedit [local]/>load last.odb

Any changes in the ODB after the last run has been stopped are not contained in this file and have to be applied manually.

4.14 Tape problems

Tapes drives can produce write errors when using bad tapes or when the write heads are dirty. The logger then tries to stop the run, even if the last events cannot be written to tape. It is then recommended to rewind the tape with the ODBEdit rewind command, clean the tape heads with a cleaning cartridge and start a new tape. It also has to be checked that the problems don't arise from the SCSI bus because of wrong termination or too many devices on the bus. The logger releases the tape after a run has been stopped. Experienced users might use the mtape utility to manipulate the tape between runs. If the tape contains a useless run which should be overwritten, it might be spooled back with the bsf command of the mtape utility. If a new tape is inserted into a tape drive that contains already some data, the tape is spooled forward to the end of the recorded data. If the tape should be overwritten, it has to be erased manually with the mtape utility by writing an EOF directly at the beginning of the tape.

5 Internal features

The main internal components of the system.

Names

This section refers to the Midas built-in capabilities. The following sections describe in more details the essential aspect of each feature starting from the frontend to the Electronic logbook.

->Skip to Utilities

5.1 Frontend code

Names

Under MIDAS, experiment hardware is structured into "equipment" which refers to a collection of hardware devices such as: a set of high voltage supplies, one or more crates of digitizing electronics like ADCs and TDCs or a set of scaler. On a software point of view, we keep that same equipment term to refer to the mean of collecting the data related to this "hardware equipment". The data from this equipment is then gathered into an "event" and send to the back-end computer for logging and/or analysis.

The frontend program (image) consists of a system framework contained in mfe.c (hidden to the user) and a user part contained in **frontend.c**. The hardware access is only apparent in the user code. Several libraries and drivers exist for various bus systems like CAMAC VME or RS232. They are located in the drivers directory of the MIDAS distribution. Some libraries consist only of a header file, others of a C file plus a header file. The file names usually refer to the manufacturer abbreviation followed by the model number of the device. The libraries are continuously expanding to widen Midas support.

ESONE standard routines for CAMAC are supplied and permit to re-use the frontend code between different platform as well as different CAMAC hardware interface without the need of modification of the code.

The user frontend code consists of a several sections described in order below. Example of frontend code can be found under the *../examples/experiment* directory:

Global declaration Up to the *User global section* the declarations are system wide and should not be remove.

- *frontend_name* value can be modified to reflect the purpose of the code.
- *frontend_call_loop* enable the function *frontend_loop()* to be run on every frontend loop (see *frontend_loop()* function below).
- *display_period* defined in millisecond the time interval between refresh of a frontend status display. The value of zero disable the display. If the frontend is started in the background with the display enabled, the stdout should be redirected to the null device to prevent process to hang.
- *max_event_size* specify the maximum size of the expected event in byte.
- *event_buffer_size* specify the maximum size of the buffer in byte to be allocated by the system.

After these system parameters, the user may add his or her own declarations.

```
// The frontend name (client name) as seen by other MIDAS clients
char *frontend_name = "Sample Frontend";

// The frontend file name, don't change it
char *frontend_file_name = __FILE__;

// frontend_loop is called periodically if this variable is TRUE
BOOL frontend_call_loop = FALSE;

//a frontend status page is displayed with this frequency in ms
INT display_period = 3000;

//maximum event size produced by this frontend
INT max_event_size = 10000;

//buffer size to hold events
INT event_buffer_size = 10*10000;

// Global user section
// number of channels
#define N_ADC  8
#define N_TDC  8
#define N_SCLR 8

CAMAC crate and slots
#define CRATE      0
```

```
#define SLOT_C212 23
#define SLOT_ADC   1
#define SLOT_TDC   2
#define SLOT_SCLR  3
```

Prototype functions The first group of prototype(7) declare the pre-defined system functions should be present. The second group defines the user functions associated to the declared equipments. All the fields are described in detailed in the following section.

```
INT frontend_init();
INT frontend_exit();
INT begin_of_run(INT run_number, char *error);
INT end_of_run(INT run_number, char *error);
INT pause_run(INT run_number, char *error);
INT resume_run(INT run_number, char *error);
INT frontend_loop();

INT read_trigger_event(char *pevent, INT off);
INT read_scaler_event(char *pevent, INT off);
```

Remark Each equipment has the option to force it-self to run at individual transition time see *The Equipment structure / RO_RUNNING.* At transition time the system functions *begin_of_run(), end_of_run(), pause_run(), resume_run()* runs **prior** the equipment functions. This gives the system the chance to take basic action on the transition request (Enable/disable LAM) before the equipment runs.

Equipment definition See The Equipment structure for further explanation.

```
#undef USE_INT
EQUIPMENT equipment[] = {

  { "Trigger",                                   // equipment name
    1, 0,                                        // event ID, trigger mask
    "SYSTEM",                                    // event buffer
    EQ_INTERRUPT,                                // equipment type
    EQ_POLLED,                                   // equipment type
    LAM_SOURCE(CRATE, LAM_STATION(SLOT_C212)),   // event source crate 0
    "MIDAS",                                     // format
    TRUE,                                        // enabled
    RO_RUNNING |                                 // read only when running
    RO_ODB,                                      // and update ODB
    500,                                         // poll for 500ms
    0,                                           // stop run after this event limit
    0,                                           // number of sub events
    0,                                           // don't log history
    "", "", "",
    read_trigger_event,                          // readout routine
  },

  ...
  }
```

Pre-defined functions The sequence of function calls throughout the frontend code is the following:

frontend init, begin of run, [pause/resume], end of run, , frontend exit

INT frontend_init() This function run once only at the application startup. Allows hardware checking, loading/setting of global variables, hot-link settings to the ODB etc... In case of CAMAC the standard call can be:

```
  cam_init();                                 // Init CAMAC access
  cam_crate_clear(CRATE);                     // Clear Crate
  cam_crate_zinit(CRATE);                     // Z crate
  cam_inhibit_set(CRATE);                     // Set I crate
  return SUCCESS;
```

INT run_number, char *error) This function is called for every run start transition. Allows to update user parameter, load/setup/clear hardware. At the exit of this function the acquisition should be armed and ready to test the LAM. In case of CAMAC frontend, the LAM has to be declared to the Crate Controller. The function *cam_lam_enable(CRATE, SLOT_IO)* is then necessary in order to enable the proper LAM source station. The LAM source station has to alos be enabled (F26).

The argument *run_number* provides the current run number being started. The argument *error* can be used for returning a message to the system. This string will be logged into the *mdas.log* file.

```
  // clear units
  camc(CRATE, SLOT_C212, 0, 9);
  camc(CRATE, SLOT_2249A, 0, 9);
  camc(CRATE, SLOT_SC2, 0, 9);
  camc(CRATE, SLOT_SC3, 0, 9);

  camc(CRATE, SLOT_C212, 0, 26);              // Enable LAM generation

  cam_inhibit_clear(CRATE);                   // Remove I

  cam_lam_enable(CRATE, SLOT_C212);           // Declare Station to CC as LAM source

  // set and clear OR1320 pattern bits
  camo(CRATE, SLOT_OR1320, 0, 18, 0x0330);
  camo(CRATE, SLOT_OR1320, 0, 21, 0x0663);    // Open run gate, reset latch
  return SUCCESS;
```

ource, INT count, BOOL test) If the equipment definition is *EQ_POLLED* as a acquisition type, the *poll_event* will be call as often as possible over the corresponding poll time (ex:500ms see The Equipment structure) given by each polling equipment. The code below shows a typical CAMAC LAM polling loop. The *source* corresponds to a bitwise LAM station susceptible to generate LAM for that particular equipement. If the LAM is ORed for several station and is independent of the equipment, the LAM test can be simplified (see example below)

```
// Trigger event routines ----------------------------------------
INT poll_event(INT source, INT count, BOOL test)
    // Polling routine for events. Returns TRUE if event
    // is available. If test equals TRUE, don't return. The test
    // flag is used to time the polling.
{
  int   i;
  DWORD lam;

  for (i=0 ; i<count ; i++)
    {
      cam_lam_read(LAM_SOURCE_CRATE(source), &lam);
      if (lam & LAM_SOURCE_STATION(source)) // Any of the equipment LAM
   // *** or ***
      if (lam)                              // Any LAM (independent of the equipment)
         if (!test)
          return lam;
    }
  return 0;
}
```

Remark When multiple LAM source is specified for a given equipment like:

```
    LAM_SOURCE(JW_C,  LAM_STATION(GE_N)
                    | LAM_STATION(JW_N)),
```

The polling function will pass to the readout function the actual LAM pattern read during the last polling. This pattern is a bitwise LAM station. The content of the *pevent* will be overwritten. This option allows you to determine which of the station has been the real source of the LAM.

```
INT read_trigger_event(char *pevent, INT off)
{
  DWORD lam;

  lam = *((DWORD *)pevent);

  // check LAM versus MCS station
  // The clear is performed at the end of the readout function
  if (lam & LAM_STATION(JW_N))
  {
   ...
  }
  ...
}
```

r_event(char *pevent, INT off) Event readout function defined in the equipment list. Refer to further section for event composition explanation FIXED event construction, MIDAS event construction, YBOS event construction.

```
// Event readout -------------------------------------------------
INT read_trigger_event(char *pevent, INT off)
{
  WORD *pdata, a;

  // init bank structure
  bk_init(pevent);

  // create ADC bank
  bk_create(pevent, "ADC0", TID_WORD, &pdata);

  ...
}
```

INT run_number, char *error) These two functions are called respectively upon "Pause" and "Resume" command. Any code relevant to the upcoming run state can be include. Possible commands when CAMAC is involved can be *cam_inhibit_set(CRATE);* and *cam_inhibit_clear(CRATE);*

The argument *run_number* provides the current run number being paused/resumed. The argument *error* can be used for returning a message to the system. This string will be logged into the *mdas.log* file.

INT run_number, char *error) For every "stop run" transition this function is called and provides opportunity to disable the hardware. In case of CAMAC frontend the LAM should be disable.

The argument *run_number* provides the current run number being ended. The argument *error* can be used for returning a message to the system. This string will be logged into the *mdas.log* file.

```
  // set and clear OR1320 pattern bits or close run gate.
  camo(CRATE, SLOT_OR1320, 0, 18, 0x0CC3);
  camo(CRATE, SLOT_OR1320, 0, 21, 0x0990);

  camc(CRATE, SLOT_C212, 0, 26);            // Enable LAM generation
```

```
      cam_lam_disable(CRATE, SLOT_C212);        // disable LAM in crate controller
      cam_inhibit_set(CRATE);                   // set crate inhibit
```

INT frontend_exit() This function runs when the frontend is requested to terminate. Can be used for local statistic collection etc.

5.1.1 The Equipment structure

-> Next FIXED event construction

To write a frontend program, the user section (frontend.c) has to have an equipment list organized as a structure definition. Here is the structure listing for a trigger and scaler equipment from the sample experiment example frontend.c.

```
#undef USE_INT
EQUIPMENT equipment[] = {

  { "Trigger",            // equipment name
    1, 0,                 // event ID, trigger mask
    "SYSTEM",             // event buffer
    EQ_INTERRUPT,         // equipment type #else
    EQ_POLLED,            // equipment type
    LAM_SOURCE(0,0xFFFFFF),// event source crate 0, all stations
    "MIDAS",              // format
    TRUE,                 // enabled
    RO_RUNNING |          // read only when running
    RO_ODB,               // and update ODB
    500,                  // poll for 500ms
    0,                    // stop run after this event limit
    0,                    // number of sub events
    0,                    // don't log history
    "", "", "",
    read_trigger_event,   // readout routine
  },

  ...
  }
```

”trigger”,”scaler” : Each equipment has to have a unique equipment name defined under a given node. The name will be the reference name of the equipment generating the event.

1, 0 Each equipment has to be associated to an unique event ID and to a trigger mask. Both the event ID and the trigger mask will be part of the event header of that particular equipment. The trigger mask can be modified dynamically by the readout routine to define a sub-event type on an event-by-event basis. This can be used to mix ”physics events” (from a physics trigger) and ”calibration events” (from a clock for example) in one run and identify them later. Both parameters are declared as 16bit value. If the Trigger mask is used in a single bit-wise mode, only up to 16 masks are possible.

”SYSTEM” After composition of an ”equipment”, the Midas frontend mfe.c takes over the sending of this event to the ”system buffer” on the back-end computer. Dedicated buffer can be specified on those lines allowing a secondary stage on the back-end (Event builder to collect and assemble these events coming from different buffers in order to compose a larger event. In this case the event coming from the frontend are called fragment). In this example both events are placed in the same buffer called "SYSTEM" (default).

Remark If this field is left empty ("") the readout function associated to that equipment will still be performed, but the actual event won't be sent to the buffer. The positive side-effect of that configuration is to allow that particuliar equipment to be mirrored in the ODB if the RO_ODB is turned on.

EQ_xxx The field specify the type of equipment EQ_PERIODIC, EQ_POLLED or EQ_INTERRUPT. In any selected case, when the equipment will be required to run, a declared function will be call doing the actual user required operation. For the EQ_PERIODIC case, another parameter is necessary to set the periodicity of the call. In the case of the EQ_POLLING mode, the name of the routine performing the trigger check function is defaulted to poll_event()(present in frontend.c). As polling consists on checking a variable for a true condition, if the loop would be infinite, the frontend would not be able respond to any network commands. Therefore the loop count is determined when the frontend starts so that it returns after a given time-out when no event is available. This time-out is usually in the order of 500 milliseconds. For EQ_INTERRUPT, Midas requires complete configuration and control of the interrupt source. This is provided by an interrupt configuration routine interrupt_configure that has to be coded by the user in the user section of the frontend code. A pointer to this routine is passed to the system instead of the polling routine. The interrupt configuration routine has the following declaration:

```
INT interrupt_configure(INT cmd, INT source [], PTYPE adr)
{
  switch(cmd)
    {
    case CMD_INTERRUPT_ENABLE:
      cam_interrupt_enable();
      break;
    case CMD_INTERRUPT_DISABLE:
      cam_interrupt_disable();
      break;
    case CMD_INTERRUPT_ATTACH:
      cam_interrupt_attach((void (*)())adr);
      break;
    case CMD_INTERRUPT_DETACH:
      cam_interrupt_detach();
      break;
    }
  return CM_SUCCESS;
}
```

Under the four commands listed above, the user has to implement the adequate hardware operation performing the requested action. In **drivers** examples can be found on such a interrupt code. See source code such as hyt1331.c, ces8210.c.

- CMD_INTERRUPT_ENABLE: to enable an interrupt
- CMD_INTERRUPT_DISABLE: to disable an interrupt
- CMD_INTERRUPT_INSTALL: to install an interrupt callback routine at address adr.
- CMD_INTERRUPT_DEINSTALL: to de-install an interrupt.

LAM_SOURCE(0,0xFFFFFF) This parameter is a bit-wise representation of the 24 CAMAC slots which may raise the LAM. It defines which CAMAC slot is allowed to trigger the call to the readout routine. (See below **read_trigger_event()**).

"MIDAS" This line specifies the data format used for generating the event. The following options are possible: MIDAS, YBOS and FIXED. The format has to agree with the way the event is composed in the user read-out routine. It tells the system how to interpret an event when it is copied to the ODB or displayed in a user-readable form.

MIDAS and YBOS or FIXED and YBOS data format can be mixed at the frontend level, but the data logger (mlogger) is not able to handle this format diversity on a event-by-event basis. In practice a given experiment should keep the data format identical throughout the equipment definition.

TRUE "enable" switch for the equipment. Only when enable (TRUE) the related equipment is active.

RO_RUNNING Specify when the read-out of an event should be occurring (transition state) or be enabled (state). Following options are possible:

RO_RUNNING	Read on state "running".
RO_STOPPED	Read on state "stopped".
RO_PAUSED	Read on state "paused".
RO_BOR	Read after begin-of-run.
RO_EOR	Read before end-of-run
RO_PAUSE	Read when run gets paused.
RO_RESUME	Read when run gets resumed.
RO_TRANSITIONS	Read on all transitions.
RO_ALWAYS	Read independently of the states and force a read for all transitions.
RO_ODB	Equipment event mirrored into ODB under variables.

These flags can be combined with the logical OR operator. Trigger events in the above example are read out only when running while scaler events is read out when running and additionally on all transitions. A special flag RO_ODB tells the system to copy the event to the /Equipment/<equipment name>/Variables ODB tree once every ten seconds for diagnostic. Later on, the event content can then be displayed with ODBEdit.

500 Time interval for Periodic equipment (EQ_PERIODIC) or time out value in case of EQ_POLLING (unit in millisecond).

0 (stop after...) Specify the number of events to be taken prior forcing an End-Of-Run transition. The value 0 disables this option.

0 (Super Event) Enable the Super event capability. Specify the maximum number of events in the Super event.

0 (History System) Enable the MIDAS history system for that equipment. The value (positive in seconds) indicates the time interval for generating the event to be available for history logging by the mlogger task if running.

"","","" Reserved field for system. Should be present and remain empty.

read_trigger_event User read-out routine declaration (could be any name). Every time the frontend is initialized, it copies the equipment settings to the ODB under /Equipment/<equipment name>/Common. A hot-link to that ODB tree is created allowing some of the settings to be changed during run-time. Modification of "Enabled" flag, RO_xxx flags, "period" and "event limit" from the ODB is immediately reflected into the frontend which will act upon them.

This function has to be present in the frontend code and will be called for every trigger under one of the two conditions:

In polling mode The poll_event has detected a trigger request while polling on a trigger source.

In interrupt mode An interrupt source pre-defined through the interrupt_configuration has occurred.

Remark 1 The first argument of the readout function provide the pointer to the newly constructed event and point to the first valid location for storing the data.

Remark 2 The content of the memory location pointed by **pevent** prior its uses in the readout function contains the LAM source bitwise register. This feature can be exploited in order to identify which slot has triggered the readout when multiple LAM has been assigned to the same readout function.
Example:

```
... in the equipment declaration
    ...
    LAM_SOURCE(JW_C,  LAM_STATION(GE_N) | LAM_STATION(JW_N)), // event source
    ...
    "", "", "",
    event_dispatcher,   // readout routine
...

INT event_dispatcher(char *pevent)
{
  DWORD lam, dword;
  INT   size=0;
  EQUIPMENT      *eq;

  // the *pevent contains the LAM pattern returned from poll_event
  //  The value can be used to dispatch to the proper LAM function

  // !!!! ONLY one of the LAM is processed in the loop !!!!
  lam = *((DWORD *)pevent);

  // check LAM versus MCS station
  if (lam & LAM_STATION(JW_N))
  {
    ...
    // read MCS event
    size = read_mcs_event(pevent);
    ...
  }
  else if (lam & LAM_STATION(GE_N))
  {
    ...
    // read GE event
    size = read_ge_event(pevent);
    ...
  }
  return size;
}
```

Remark 3 In the above example, the Midas Event Header will contains the same Event ID as well as the Trigger mask for both LAM. The event serial number will be incremented by one for every call to *event_dispatcher* as long as the returned size is non-zero.

Remark 4 The return value should represent the number of bytes collected in this function. If the returned value is set to zero, The event will be dismissed and the serial number to that event will be decremented by one.

5.1.2

FIXED event construction

-> Next MIDAS event construction

The FIXED format is the simplest event format. The event length is fixed and maps to a C structure that is filled by the readout routine. Since the standard MIDAS analyzer cannot work with this format, it is only recommended for experiment, which use its own analyzer and want to avoid the overhead of a bank structure. For fixed events, the structure has to be defined twice: Once for the compiler in form of a C structure and once for the ODB in form of an ASCII representation. The ASCII string is supplied to the system as the "init string" in the equipment list.

Following statements would define a fixed event with two ADC and TDC values:

```
typedef struct {
  int adc0;
  int adc1;
  int tdc0;
  int tdc1;
} TRIGGER_EVENT;
char *trigger_event_str[] = {
"adc0 = INT : 0",
"adc1 = INT : 0",
"tdc0 = INT : 0",
"tdc1 = INT : 0",
} ASUM_BANK;
```

The *trigger_event_str* has to be defined before the equipment list and a reference to it has to be placed in the equipment list like:

```
{
...
  read_trigger_event, // readout routine
  poll_trigger_event, // polling routine
  trigger_event_str,  // init string
},
```

The readout routine could then look like this, where the <...> statements have to be filled with the appropriate code accessing the hardware:

```
INT read_trigger_event(char *pevent)
{
TRIGGER_EVENT *ptrg;

  ptrg = (TRIGGER_EVENT *) pevent;
  ptrg->adc0 = <...>;
  ptrg->adc1 = <...>;
  ptrg->tdc0 = <...>;
  ptrg->tdc1 = <...>;
  return sizeof(TRIGGER_EVENT);
}
```

5.1.3 MIDAS event construction

-> Next YBOS event construction

The MIDAS event format is a variable length event format. It uses ”banks” as subsets of an event. A bank is composed of a bank header followed by the data. The bank header itself is made of 4 fields i.e: bank name (4 char max), bank type, bank length. Usually a bank contains an array of values that logically belong together. For example, an experiment can generate an ADC bank, a TDC bank and a bank with trigger information. The length of a bank can vary from one event to another due to zero suppression from the hardware. Beside the variable data length support of the bank structure, onother main advantage is the possibility for the analyzer to add more (calculated) banks during the analysis process to the event in process. After the first analysis stage, the event can contain additionally to the raw ADC bank a bank with calibrated ADC values called CADC bank for example. In this CADC bank the raw ADC values could be offset or gain corrected.

MIDAS banks are created in the frontend readout code with calls to the MIDAS library. Following routines exist:

bk_init(), bk_init32() Initializes a bank structure in an event.

bk_create() Creates a bank with a given name (exactly four characters)

bk_close() Closes a bank previously opened with bk_create().

bk_locate() Locate a bank within anevent by its name.

bk_iterate() return bank and data pointers to each bank in the event.

bk_list() construct a string of all the bank name in the event.

bk_size() Returns the size in bytes of all banks including the bank headers in an event.

The following code composes a event containing two ADC and two TDC values, the <...> statements have to be filled with specific code accessing the hardware:

```
INT read_trigger_event(char *pevent)
{
INT *pdata;

  bk_init(pevent);

  bk_create(pevent, "ADC0", TID_INT, &pdata);
  *pdata++ = <ADC0>
  *pdata++ = <ADC1>
  bk_close(pevent, pdata);

  bk_create(pevent, "TDC0", TID_INT, &pdata);
  *pdata++ = <TDC0>
  *pdata++ = <TDC1>
  bk_close(pevent, pdata);

  return bk_size(pevent);
}
```

Upon normal completion, the readout routine returns the event size in bytes. If the event is not valid, the routine can return zero. In this case no event is sent to the back-end. This can be used to implement a software event filter (sometimes called "third level trigger").

```
INT read_trigger_event(char *pevent)
{
WORD *pdata, a;

  // init bank structure
  bk_init(pevent);

  // create ADC bank
  bk_create(pevent, "ADC0", TID_WORD, &pdata);

  // read ADC bank
  for (a=0 ; a<8 ; a++)
    cami(1, 1, a, 0, pdata++);

  bk_close(pevent, pdata);

  // create TDC bank
  bk_create(pevent, "TDC0", TID_WORD, &pdata);

  // read TDC bank
  for (a=0 ; a<8 ; a++)
    cami(1, 2, a, 0, pdata++);

  bk_close(pevent, pdata);

  return bk_size(pevent);
}
```

5.1.4 YBOS event construction

-> Next Deferred Transition

The YBOS event format is also a bank format used in other DAQ systems. The advantage of using this format is the fact that recorded data can be analyzed with pre-existing analyzers understanding YBOS format. The disadvantage is that it has a slightly larger overhead than the MIDAS format and it supports fewer different bank types. An introduction to YBOS can be found under:

`http://www-cdf.fnal.gov/offline/ybos/ybos.html`

The scheme of bank creation is exactly the same as for MIDAS events, only the routines are named differently. The YBOS format is double word oriented i.e. all incrementation are done in 4 bytes steps. Following routines exist:

ybk_init Initializes a bank structure in an event.

ybk_create Creates a bank with a given name (exactly four characters)

ybk_close Closes a bank previously opened with ybk_create().

ybk_size Returns the size in bytes of all banks including the bank headers in an event.

The following code creates an ADC0 bank in YBOS format:

```
INT read_trigger_event(char *pevent)
{
  DWORD i;
  DWORD *pbkdat;

  ybk_init((DWORD *) pevent);

  // collect user hardware data
  ybk_create((DWORD *)pevent, "ADC0", I4_BKTYPE, (DWORD *)(&pbkdat));
  for (i=0 ; i<8 ; i++)
    *pbkdat++ = i & 0xFFF;
  ybk_close((DWORD *)pevent, pbkdat);

  ybk_create((DWORD *)pevent, "TDC0", I2_BKTYPE, (DWORD *)(&pbkdat));
  for (i=0 ; i<8 ; i++)
    *((WORD *)pbkdat)++ = (WORD)(0x10+i) & 0xFFF;
  ybk_close((DWORD *) pevent, pbkdat);

  ybk_create((DWORD *)pevent, "SIMU", I2_BKTYPE, (DWORD *)(&pbkdat));
  for (i=0 ; i<9 ; i++)
    *((WORD *)pbkdat)++ = (WORD) (0x20+i) & 0xFFF;
  ybk_close((DWORD *) pevent, I2_BKTYPE, pbkdat);

  return (ybk_size((DWORD *)pevent));
}
```

5.1.5

Deferred Transition

-> Next Super Event

This option permits the user to postpone any transition issued by any requester until some condition are satisfied. As examples:

- It may not be advised to pause or stop a run until let say some hardware has turned off a particular valve.

- The start of the acquisition system is postpone until the beam rate has been stable for a given period of time.
- While active, a particular acquisition system should not be interrupted until the "cycle" is complete.

In these examples, any application having access to the state of the hardware can register to be a "transition Deferred" client. It will then catch any transition request and postpone the trigger of such transition until *condition* is satisfied. The *Deferred Transition* requires 3 steps setup:

1. **Register the deferred transition.**

```
//-- Frontend Init
INT frontend_init()
{
  INT    status, index, size;
  BOOL   found=FALSE;

  // register for deferred transition
  cm_register_deferred_transition(TR_STOP, wait_end_cycle);
  cm_register_deferred_transition(TR_PAUSE, wait_end_cycle);
  ...
}
```

2. **Provide callback function to serve the deferred transition**

```
//-- Deferred transition callback
BOOL wait_end_cycle(int transition, BOOL first)
{
  if (first)
  {
    transition_PS_requested = TRUE;
    return FALSE;
  }

  if (end_of_mcs_cycle)
  {
    transition_PS_requested = FALSE;
    end_of_mcs_cycle = FALSE;
    return TRUE;
  }
  else
    return FALSE;
}
```

3. **Implement the condition code**

```
  ... In this case at the end of the readout function...
  ...
INT read_mcs_event(char *pevent, INT offset)
{
  ...

  if (transition_PS_requested)
  {
    // Prevent to get new MCS by skipping re_arm_cycle and GE by GE_DISABLE LAM
    cam_lam_disable(JW_C,JW_N);
```

```
      cam_lam_disable(GE_C,GE_N);
      cam_lam_clear(JW_C,JW_N);
      cam_lam_clear(GE_C,GE_N);
      camc(GE_C,GE_N,0,GE_DISABLE);
      end_of_mcs_cycle = TRUE;
    }
    re_arm_cycle();
    return bk_size(pevent);
  }
```

In the example above the frontend code register for PAUSE and STOP. The second argument of the cm_register *wait_end_cycle* is the declaration of the callback function. The callback function will be called as soon as the transition is requested and will provide the Boolean flag *first* to be TRUE. By setting the *transition_PS_requested*, the user will have the acknowledgment of the transition request. By returning FALSE from the callback you will prevent the transition to occur. As soon as the user condition is satisfied (end_of_mcs_cycle = TRUE), the return code in the callback will be set to TRUE and the requested transition will be issued.

The *Deferred transition* shows up in the ODB under */runinfo/Requested transition* and will contain the transition code (see Transition Codes).

When the system is in deferred state, an ODBedit override command can be issued to **force** the transition to happen. **eg: odbedit> stop now, odbedit> start now** . This overide will do the transition function regarless of the state of the hardware involved.

5.1.6

Super Event

-> Next ODB Structure

The *Super Event* is a option implemented in the frontend code in order to reduce the amount of data to be transfered to the backend by removing the bank header for each event constructed. In other words, when an equipment readout in either *MIDAS* or *YBOS* format (bank format) is complete, the event is composed of the bank header followed by the data section. The overhead in bytes of the bank structure is 16 bytes for bk_init(), 20 bytes for bk_init32() and ybk_init(). If the data section size is close to the number above, the data transfer as well as the data storage has an non-negligible overhead. To address this problem, the equipment can be setup to generate a so called *Super Event* which is an event composed of the initial standard bank header for the first event on the super event and up to **number of sub event** maximum successive data section before closing of the bank.

To demonstrate the use of it, let see the following example:

- Define equipment to be able to generate *Super Event*

```
{ "GE",                 // equipment name
  2, 0x0002,            // event ID, trigger mask
  "SYSTEM",             // event buffer
  EQ_INTERRUPT,         // equipment type
  EQ_POLLED,            // equipment type
  LAM_SOURCE(GE_C, LAM_STATION(GE_N)),     // event source
  "MIDAS",              // format
```

This page was generated with the help of DOC++
http://www.linuxsupportline.com/~doc++

```
    TRUE,                 // enabled
    RO_RUNNING,           // read only when running
    200,                  // poll for 200ms
    0,                    // stop run after this event limit
    1000,                 // -----> number of sub event <-----  enable Super event
    0,                    // don't log history
    "", "", "",
    read_ge_event,        // readout routine
    },
    ...
```

- Setup the readout function for *Super Event* collection.

```
//-- Event readout

// Global and fixed -- Expect NWORDS 16bits data readout per sub-event
#define NWORDS 3

INT read_ge_event(char *pevent, INT offset)
{
  static WORD *pdata;

  // Super event structure
  if (offset == 0)
  {
    // FIRST event of the Super event
    bk_init(pevent);
    bk_create(pevent, "GERM", TID_WORD, &pdata);
  }
  else if (offset == -1)
  {
    // close the Super event if offset is -1
    bk_close(pevent, pdata);

    // End of Super Event
    return bk_size(pevent);
  }

  // read GE sub event (ADC)
  cam16i(GE_C, GE_N, 0, GE_READ, pdata++);
  cam16i(GE_C, GE_N, 1, GE_READ, pdata++);
  cam16i(GE_C, GE_N, 2, GE_READ, pdata++);

  // clear hardware
  re_arm_ge();

  if (offset == 0)
  {
    // Compute the proper event length on the FIRST event in the Super Event
    // NWORDS correspond to the !! NWORDS WORD above !!
    // sizeof(BANK_HEADER) + sizeof(BANK) will make the 16 bytes header
    // sizeof(WORD) is defined by the TID_WORD in bk_create()

    return NWORDS * sizeof(WORD) + sizeof(BANK_HEADER) + sizeof(BANK);
  }
  else
    // Return the data section size only
    // sizeof(WORD) is defined by the TID_WORD in bk_create()
```

```
        return NWORDS * sizeof(WORD);
    }
```

The encoded decryption of the data section is left to the user. If the number of words per sub-event is fixed (NWORD), the sub-event extraction is simple. In the case of variable sub-event length, it is necessary to tag the first or the last word of each sub-event. The content of the sub-event is essentially the responsibility of the user.

Remark 1 The backend analyzer will have to be informed by the user on the content structure of the data section of the event as no particular tagging is applied to the *Super Event* by the Midas transfer mechanism.

Remark 2 If the *Super Event* is composed in a remote equipment running a different *Endian* mode than the backend processor, it would be necessary to insure the data type consistency throughout the *Super Event* in order to guaranty the proper byte swapping of the data content.

Remark 3 The event rate in the equipment statistic will indicates the rate of sub-events.

5.2 ODB Structure

Names

-> Next Hot Link

The Online Database contains information that system and user wants to share. Basically all transactions for experiment setup and monitoring go through the ODB. It also contains some specific system information related to the "Midas client" currently involved in an experiment (/system).

Each ODB field or so called **KEY** is accessible by the user through either an interactive way (see odbedit task) or by C-programming (see db_ function in Midas Library).

The ODB information is stored in a "tree/branch" structure where each branch refers to a specific set of data. On the first invocation of the database (first Midas application) a minimal system record will be created. Later on each application will add its own set of parameters to the database depending on its requirement. For instance, starting the ODB for the first time, the tree **/Runfinfo, /Experiment, /System** will be created. The application **mlogger** (data logger) will add its own tree **/Logger/...**

As mentioned earlier, ODB is the main communication platform between any Midas application. As such, the content of the ODB is application dependent. Several "dormant" trees can be awaken by the user in order to provide extra flexibility of the system. Such "dormant" tree are **Alias, Script, Edit on Start , Security, Run parameters**. Any of those tree are not visible until the user actually create them. Each of these tree will be discussed in their respective utility.

/Alias Tree containing symbolic links list to any ODB location. Will appear in the Midas Web server main page (see mhttpd task).

/Script Tree containing list of shell scripts. Will appear as a button in the Midas web server main page (see mhttpd task).

/Edit on Start Tree containing symbolic links of run information to be requested during the run start up (see odbedit task).

/Run Parameters Tree containing run parameters which can be requested through the **Edit on Start** links (see odbedit task.

5.2.1 ODB /System Tree

The system tree contains information specific to each "Midas client" currenltly connected to the experiment. This information is not primarly for the user but may be informative in some respect to the reader.

```
[host:expt:Stopped]/>ls -r -l /system
Key name                        Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
System                          DIR
    Clients                     DIR
        29580                   DIR
            Name                STRING  1     32    17h  0   R    decay
            Host                STRING  1     256   17h  0   R    host1
            Hardware type       INT     1     4     17h  0   R    42
            Server Port         INT     1     4     17h  0   R    1227
            Transition Mask     DWORD   1     4     17h  0   R    329
            Deferred Transition DWORD   1     4     17h  0   R    6
            RPC                 DIR
                16000           BOOL    1     4     17h  0   R    y
                16001           BOOL    1     4     17h  0   R    y
        29638                   DIR
            Name                STRING  1     32    17h  0   R    MStatus
            Host                STRING  1     256   17h  0   R    host1
            Hardware type       INT     1     4     17h  0   R    42
            Server Port         INT     1     4     17h  0   R    1228
            Transition Mask     DWORD   1     4     17h  0   R    0
        Deferred Transition DWORD   1     4     17h  0   R    0
    29810                   DIR
        Name                STRING  1     32    17h  0   R    Nova_029810
        Host                STRING  1     256   17h  0   R    host
        Hardware type       INT     1     4     17h  0   R    42
        Server Port         INT     1     4     17h  0   R    1235
        Transition Mask     DWORD   1     4     17h  0   R    0
    29919                   DIR
        Name                STRING  1     32    17h  0   R    Epics
        Host                STRING  1     256   17h  0   R    host
        Hardware type       INT     1     4     17h  0   R    42
        Server Port         INT     1     4     17h  0   R    1237
        Transition Mask     DWORD   1     4     17h  0   R    329
        Deferred Transition DWORD   1     4     17h  0   R    0
        RPC                 DIR
            16000           BOOL    1     4     17h  0   R    y
            16001           BOOL    1     4     17h  0   R    y
    12164                   DIR
        Name                STRING  1     32    6s   0   R    ODBEdit
        Host                STRING  1     256   6s   0   R    host2
        Hardware type       INT     1     4     6s   0   R    42
        Server Port         INT     1     4     6s   0   R    4893
        Transition Mask     DWORD   1     4     6s   0   R    0
        Deferred Transition DWORD   1     4     6s   0   R    0
        Link timeout        INT     1     4     6s   0   R    10000
Client Notify               INT     1     4     6s   0   RWD  0
Prompt                      STRING  1     256   >99d 0   RWD  [%h:%e:%S]%p>
Tmp                         DIR
```

Remark 1 The key **Prompt** sets up the prompt of the ODBEdit program.

```
odbedit
[local:midas:Stopped]/>cd /System/
[local:midas:Stopped]/System>ls
Clients
Tmp
Client Notify                   0
Prompt                          [%h:%e:%S]%p>

[local:midas:Stopped]/System>set Prompt my_prompt>
my_prompt>set Prompt [Host:%h-Expt:%e:State:%s]Path:%p>
[Host:local-Expt:midas-State:S]Path:/System>set Prompt [Host:%h-Expt:%e-State:%S]Path:%p>
[Host:local-Expt:midas-State:Stopped]Path:/System>
```

5.2.2 ODB /RunInfo Tree

This branch contains system information related to the run information. Several time fields are available for run time statistics.

```
odb -e expt -h host
[host:expt:Running]/>ls -r -l /runinfo
```

```
Key name                     Type    #Val  Size  Last Opn Mode Value
------------------------------------------------------------------------
Runinfo                       DIR
    State                     INT     1     4     2h   0   RWD  3
    Online Mode               INT     1     4     2h   0   RWD  1
    Run number                INT     1     4     2h   0   RWD  8521
    Transition in progress    INT     1     4     2h   0   RWD  0
    Requested transition      INT     1     4     2h   0   RWD  0
    Start time                STRING  1     32    2h   0   RWD  Thu Mar 23 10:03:44 2000
    Start time binary         DWORD   1     4     2h   0   RWD  953834624
    Stop time                 STRING  1     32    2h   0   RWD  Thu Mar 23 10:03:33 2000
    Stop time binary          DWORD   1     4     2h   0   RWD  0
```

State Specifies in which state the current run is. The possible states are 1: STOPPED, 2: RUNNING, 3: PAUSED.

Online Mode Specifies the expected acquisition mode. This parameter allows the user to detect if the data are coming from a "real-time" hardware source or from a data save-set. Note that for analysis replay using "analyzer" this flag will be switched off.

Run number Specifies the current run number. This number is automatically incremented by a successful run start procedure.

Transition in progress Specifies the current internal state of the system. This parameter is used for multiple source of "run start" synchronization.

Requested transition Specifies the current internal of the Deferred Transition state of the system.

Start Time Specifies in an ASCII format the time at which the last run has been started.

Start Time binary Specifies in a binary format at the time at which the last run has been started This field is useful for time interval computation.

Stop Time Specifies in an ASCII format the time at which the last run has been stopped.

Stop Time binary Specifies in a binary format the time at which the last run has been stopped. This field is useful for time interval computation.

5.2.3 ODB /Equipment Tree

Every frontend create a entry under the /Equipment tree. The name of the sub-tree is taken from the frontend source code in the equipment declaration (frontend.c). More detailed explanation of the composition of that tree will be found throughout this document.

```
{
 "DspecCheck",      // equipment name
 ...
},
{
 "Scaler",      // equipment name
 ...
},
```

Example:

```
Key name                        Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
HistoCheck                      DIR
DSpecCheck                      DIR
HistoPoll                       DIR
HistoEOR                        DIR
DSpecEOR                        DIR
Scaler                          DIR
SuconMagnet                     DIR
TempBridge                      DIR
Cryostat                        DIR
Meters                          DIR
RFSource                        DIR
DSPec                           DIR
```

The equipment tree is then split in several sections which by default the system creates.

- Common : Contains the system information. Should not be overwritten by the user.
- Variables : Contains the equipment data if enabled (see below).
- Settings : Contains the equipment specific information that the user may want to maintain. In the case of a "Slow Control System" equipment, extended tree structure is created by the system.
- Statistics : Contains equipment statistics information such as event taken, event rate, data rate.

```
[local:S]ls -l -r /equipment/scaler
Key name                        Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
Scaler                          DIR
    Common                      DIR
        Event ID                WORD    1     2     16h  0   RWD  1
        Trigger mask            WORD    1     2     16h  0   RWD  256
        Buffer                  STRING  1     32    16h  0   RWD  SYSTEM
        Type                    INT     1     4     16h  0   RWD  1
        Source                  INT     1     4     16h  0   RWD  0
        Format                  STRING  1     8     16h  0   RWD  MIDAS
        Enabled                 BOOL    1     4     16h  0   RWD  y
        Read on                 INT     1     4     16h  0   RWD  377
        Period                  INT     1     4     16h  0   RWD  1000
        Event limit             DOUBLE  1     8     16h  0   RWD  0
        Num subevents           DWORD   1     4     16h  0   RWD  0
        Log history             INT     1     4     16h  0   RWD  0
        Frontend host           STRING  1     32    16h  0   RWD  midtis03
        Frontend name           STRING  1     32    16h  0   RWD  feLTNO
        Frontend file name      STRING  1     256   16h  0   RWD  C:\online\sc_ltno.c
    Variables                   DIR
        SCLR                    DWORD   6     4     1s   0   RWD
                                        [0]             0
                                        [1]             0
                                        [2]             0
                                        [3]             0
                                        [4]             0
                                        [5]             0
        RATE                    FLOAT   6     4     1s   0   RWD
                                        [0]             0
                                        [1]             0
                                        [2]             0
                                        [3]             0
                                        [4]             0
                                        [5]             0
    Statistics                  DIR
        Events sent             DOUBLE  1     8     1s   0   RWDE 370
        Events per sec.         DOUBLE  1     8     1s   0   RWDE 0.789578
        kBytes per sec.         DOUBLE  1     8     1s   0   RWDE 0.0678543
```

5.2.4 ODB /Logger Tree

The /Logger ODB tree contains all the relevant information for the Midas logger utility (mlogger task) to run properly. This utility provides the mean of storing the physical data retrieved by the frontend to a storage media. The user has no code to write in order for the system to operate correctly. Its general behavior can be customized and multiple logging channels can be defined. The application supports so far three type of storage devices i.e.: Disk, Tape and FTP channel.

Default settings are created automatically when the logger starts the first time:

```
Key name                        Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
Logger                          DIR
    Data dir                    STRING  1     256   4h   0   RWD  /scr0/spring2000
    Message file                STRING  1     256   22h  0   RWD  midas.log
    Write data                  BOOL    1     4     2h   0   RWD  n
    ODB Dump                    BOOL    1     4     22h  0   RWD  y
    ODB Dump File               STRING  1     256   22h  0   RWD  run%05d.odb
    Auto restart                BOOL    1     4     22h  0   RWD  y
    Tape message                BOOL    1     4     15h  0   RWD  y
    Channels                    DIR
        0                       DIR
            Settings            DIR
                Active          BOOL    1     4     1h   0   RWD  y
                Type            STRING  1     8     1h   0   RWD  Disk
                Filename        STRING  1     256   1h   0   RWD  run%05d.ybs
                Format          STRING  1     8     1h   0   RWD  YBOS
                ODB Dump        BOOL    1     4     1h   0   RWD  y
                Log messages    DWORD   1     4     1h   0   RWD  0
                Buffer          STRING  1     32    1h   0   RWD  SYSTEM
                Event ID        INT     1     4     1h   0   RWD  -1
                Trigger Mask    INT     1     4     1h   0   RWD  -1
                Event limit     DWORD   1     4     1h   0   RWD  0
                Byte limit      DOUBLE  1     8     1h   0   RWD  0
                Tape capacity   DOUBLE  1     8     1h   0   RWD  0
            Statistics          DIR
                Events written  DOUBLE  1     8     1h   0   RWD  0
                Bytes written   DOUBLE  1     8     1h   0   RWD  0
                Bytes written toDOUBLE  1     8     1h   0   RWD  3.24316e+11
                Files written   INT     1     4     1h   0   RWD  334
```

Data dir Specifies in which directory files produced by the logger should be written. Once the Logger in running, this **Data Dir** will be pointing to the location of the **midas.log**, ODB dump files, history files, message files.

In the case of multiple logging channels, the data path for all the channels is defaulted to the same location. In the case where specific directory has to be assigned to each individual logging channel, the field **/logger/channel/<x>/Settings/Filename** can contain the full path of the location of the **.mid, .ybs, .asc** file. By finding the OS specific *SEPARATOR_DIR* (”/”, ”̈). The field **Filename** will overwite the global **Data Dir** setting for that particular channel.

History Dir This field is optional and doesn’t appear by default in the logger. If present the location of the History System files is reassigned to the defined path instead of the default **Data dir**.

Elog Dir This field is optional and doesn’t appear by default in the logger. If present the location of the Electronic Logbook files is reassigned to the defined path instead of the default **Data dir**.

Message file Specifies the file name for the log file which contains all messages from the MIDAS message system. The message log file is a simple ASCII file, which can be viewed at any time to see a history of what happened in an experiment.

Write data Global flag which turns data logging on and off for all channels. It can be set to zero temporarily to make a short test run without data logging. The key ”Write data?” is predefined logger key for enabling data logging. This action can be overridden by setting the active key to 1.

ODB Dump Specifies if a dump of the complete ODB should be written to the file specified by ODB Dump File.

ODB Dump File At the end of each run. If the file name contains a ”%”, this gets replaced by the current run number similar to the printf() C function. The format specifier %05d from above would be evaluated to a five digit run number with leading zeros like run00002.odb. The ODB dump file is in ASCII format and can be used for off-line analysis to check run parameters etc. For a description of the ASCII format see db_copy.

Auto restart When this flag is one, a new run gets automatically restarted when the previous run has been stopped by the logger due to an event or byte limit.

Tape message Specifies if tape messages during mounting and writing of EOF marks are generated. This can be useful for slow tapes to inform all users in a counting house about the tape status.

channels Sub-directory which contains settings for individual channels. By default, only channel ”0” is created. To define other channels, an existing channel can be copied:

```
[loca]]Logger>cd channels
[local]Channels>ls
0
[local]Channels>copy 0 1
[local]Channels>ls
0
1
```

The Settings part of the channel tree has the following meaning:

active turns a channel on (1) or off (0). Data is only logged to channels that are active.

Type Specify the type of media on which the logging should take place. It can be Disk, Tape or FTP to write directly to a remote computer via FTP.

Filename Specify the name of a file in case of a disk logging, where %05d is replaced by the current run number the same way as for the ODB dump files. In the case of a tape logging, the filename specifies a tape device like /dev/nrmt0 or /dev/nst0 under UNIX or \\.\tape0 under Windows NT. In FTP mode, the filename specifies the access information for the FTP server. It has the form: host name, port number, user name, password, directory, file name. The port number for normal FTP is 21 and 1021 for a Unitree Archive like the one used at the Paul Scherrer Institute. By using the FTP mode, a back-end computer can directly write to the archive. myhost.my.domain,21,john,password,/usr/users/data,run%05d.mid

Format Specifies the format to be used for writing the data to the logging channel. It can have four values: MIDAS, YBOS, ASCII and DUMP. The MIDAS and YBOS binary formats MIDAS Event Format and YBOS Event Format, respectively. The ASCII format converts events into readable text format which can be easily analyzed by programs which have problems reading binary data. While the ASCII format tries to minimize the file size by printing one event per line, the DUMP format gives a very detailed ASCII representation of the event including bank information, serial numbers etc, it should be used for diagnostics. Consistency of this type of format has to be maintained between the frontend declaration and the logger.

ODB Dump Specifies the complete dump of the ODB to the logging channel before and after every run. The ODB content is dumped in one long ASCII string reflecting the status at begin-of-run event and at end-of-run event. These special events have an ID of EVENT_ID_BOR and EVENT_ID_EOR (0x8000 and 0x8001) and a serial number equals to the current run number. An analyzer in the off-line analysis stage can restore the ODB to its online state.

Log messages This is a bit-field for logging system messages. If a bit in this field is set, the according system message is written to the logging channel as a message event with an ID of EVENT_ID_MESSAGE (0x8002). The bits are 1 for error, 2 for info, 4 for debug, 8 for user, 16 for log, 32 for talk, 64 for call messages and 255 to log all messages. For an explanation of these messages refer to Buffer, Event ID and Trigger.

Mask Specify which events to log. See Frontend code to learn how events are selected by their ID and trigger mask. To receive all events, -1 is used for the event ID and the trigger mask. By using a buffer other than the "SYSTEM" buffer, event filters can be realized. An analyzer can request all events from the "SYSTEM" buffer, but only write acceptable events to a new buffer called "FILTERED". When the logger request now only events from the new buffer instead of the "SYSTEM" buffer, only filtered events get logged.

Event limit, Byte limit and Tape capacity These fields can be used to stop a run when set to a non-zero value. The statistics values Events written, Bytes written and Bytes written total are checked respectively against these limits. When one of these condition is reached, the run is stopped automatically by the logger. Updates of the statistics branch is performed automatically every so often. This branch contains the number of events and bytes written. These two keys are cleared at the beginning of each run. The **Bytes written total** and **Files written** keys are only reset when a tape is rewound with the ODBEdit command rewind. The Bytes written total entry can therefore be used as an indicator if a tape is full. The Files written entry can be used off-line to determine how many files on tape have to be skipped in order to reach a specific run.

5.2.5

ODB /Experiment Tree

Under this tree, the Midas system stores special features for the user in order to facilitate his job on controlling a run. Initially only one empty key is defined labeled **Name** for the experiment name. The user can create four system keys in order to provide extra run control flexibility i.e.: **”Run Parameter/”**, **”Edit on Start/”**, **”Lock when running/”** and **”Security/”**.

```
Key name                        Type    #Val  Size  Last Opn Mode Value
--------------------------------------------------------------------------
Experiment                      DIR
    Name                        STRING  1     32    22s  0   RWD  chaos
    Run Parameter               DIR
        Beam Polarity           STRING  1     256   2h   0   R    negative
        Beam Momentum           FLOAT   1     4     2h   0   R    91
        2LT: log file name?     STRING  1     256   2h   0   R    cni05
        1LT: file name?         STRING  1     256   2h   0   R    files.cni.zero
        Comment                 STRING  1     256   2h   0   R    ch2 target
        Target Angle            FLOAT   1     4     2h   0   R    0
        Target Material         STRING  1     256   2h   0   R    ch2
    Edit on start               DIR
        Beam Momentum           FLOAT   1     4     2h   0   R    91
        Beam Polarity           STRING  1     256   2h   0   R    negative
        Target Material         STRING  1     256   2h   0   R    ch2
        Target Angle            FLOAT   1     4     2h   0   R    0
        1LT: file name?         STRING  1     256   2h   0   R    files.cni.zero
        Trigger 2               BOOL    1     4     2h   0   RWD  n
        2LT: log file name?     STRING  1     256   2h   0   R    cni05
        Comment                 STRING  1     256   2h   0   R    ch2 target
        Write data              BOOL    1     4     2h   0   RWD  y
    Lock when running           DIR
        Run Parameter           DIR
            Beam Polarity       STRING  1     256   2h   0   R    negative
            Beam Momentum       FLOAT   1     4     2h   0   R    91
            2LT: log file name? STRING  1     256   2h   0   R    cni05
            1LT: file name?     STRING  1     256   2h   0   R    files.cni.zero
            Comment             STRING  1     256   2h   0   R    ch2 target
            Target Angle        FLOAT   1     4     2h   0   R    0
            Target Material     STRING  1     256   2h   0   R    ch2
    Security                    DIR
        Password                STRING  1     32    16h  0   RWD  #$%#@DF564%*
        Allowed hosts           DIR
            host.sample.domain  INT     1     4     >99d 0   RWD  0
            pierre.triumf.ca    INT     1     4     >99d 0   RWD  0
            pcch02.triumf.ca    INT     1     4     >99d 0   RWD  0
            koslx1.triumf.ca    INT     1     4     >99d 0   RWD  0
            koslx2.triumf.ca    INT     1     4     >99d 0   RWD  0
            vwchaos.triumf.ca   INT     1     4     >99d 0   RWD  0
            koslx0.triumf.ca    INT     1     4     >99d 0   RWD  0
        Allowed programs        DIR
            mstat               INT     1     4     >99d 0   RWD  0
            mhttpd              INT     1     4     >99d 0   RWD  0
        Web Password            STRING  1     32    16h  0   RWD  pon4@#@%SSDF2
```

Name Specifies the name of the experiment.

Run Parameters Specifies a fix directory name where you can create and define keys which can be presented at Run start for run condition selection. The actual activation of any of those line is done via a "logical link key" defined in the Edit on Start/ sub-tree. The links don't have to point to run parameters necessarily. They can point to any ODB key including the logger settings. It can make sense to create a link to the logger setting which enables/disables writing of data. A quick test run can then be made without data logging for example:

```
[local]/>create key "/Experiment/Run parameters"
```

Then one or more run parameters can be created in that directory:

```
[local]Run parameters>create int "Run mode"
[local]Run parameters>create string Comment
```

Edit on Start Specifies a fix directory name where you can define an ODB link (similar to a symbolic link in UNIX) key to the pre-defined directory Run Parameters. Any link key present in this directory pointing to a valid ODB key will be requested for input during the run start procedure.

A new feature has been added to this section for the possibility of preventing the user to change the run number from the web interface during the start sequence. By defining the key **/Experiment/Edit on Start/Edit run number** as a boolean variable the ability of editing the run number is enabled or disabled. By default if this key is not present the run number is editable.

```
[local]/>create key "Experiment/Edit on start"
[local]/>cd "Experiment/Edit on start"
[local]/>ln "/Experiment/Run parameters/Run mode" "Run mode"
```

When a run is started from ODBEdit, all links in /Experiment/Edit on start are scanned and read in:

```
[local]/>start
Run mode [0]:1
Run number [3]:<return to accept>
Are the above parameters correct?
([y]/n/q): <return to accept "y">
Starting run #2
Run #2 started

[local]/>cd "Experiment/Edit on start"
[local]/>create BOOL "Edit run number"
```

Lock when running Specifies a fix directory for defining logical link keys to be set in Read only access mode while the run is in progress. The lock when running can contains logical link to key(s) for setting these keys protection to "read only" while running. In the example below, all the parameters under the declared tree will be switched to read only preventing any parameters modification during the run.

```
[local]/>create key "Experiment/Lock when running"
[local]/>cd "Experiment/Lock when running"
[local]/>ln "/Experiment/Run parameters" "Run parameter"
[local]/>ln "/Logger/Write Data" "Write Data?"
```

Security Specifies a fix directory name where information regarding security can be setup. By default, there is no restriction for user to connect locally or remotely to a given experiment. If an access restriction has to be setup in order to protect the experiment from unwilling access, a password mechanism has to be defined.

Password Specifies the encrypted password for accessing current experiment.

```
[local]/>passwd
Password:<xxxx>
Retype password:<xxxx>
```

To remove the full password checking mechanism, the ODB security sub-tree has to be entirely deleted using the following command:

```
[local]/>rm /Experiment/Security
Are you sure to delete the key
"/Experiment/Security"
and all its subkeys? (y/[n]) y
```

After running the odb command passwd, four new sub-fields will be present under the Security tree.

- Password
- Allowed hosts
- Allowed programs
- Web Password

Allowed hosts Specifies a fix directory name where allowed remote hostname can be defined for free access to the current experiment. While the access restriction can make sense to deny access to outsider to a given experiment, it can be annoying for the people working directly at the back-end computer or for the automatic frontend reloading mechanism (MS-DOS, VxWorks configuration). To address this problem specific hosts can be exempt from having to supply a password and being granted of full access.

```
[local]/>cd "/Experiment/Security/Allowed hosts"
[local]rhosts>create int myhost.domain
[local]rhosts>
```

Where <myhost>.<domain> has to be replaces by the full IP address of the host requesting full clearance.

Allowed programs Specifies a list of programs having full access to the ODB independently of the node they running from.

```
[local]/>cd "/Experiment/Security/Allowed programs"
[local]:S>create int mstat
[local]:S>
```

Web Password Specifies a separate password for the Web server access (mhttpd task). If this field is active, the user will be requested to provide the "Web Password" when accessing the requested experiment in a "Write Access". In all condition the Read Only Access" is available.

5.2.6 ODB /History Tree

This tree is automatically created when the logger is started. The logger will create a default sub-tree containing the following structure:

```
[local:midas:S]/History>ls -l -r
Key name                        Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
History                         DIR
    Links                       DIR
        System                  DIR
Trigger per sec.    /Equipment/Trigger/Statistics/Events per sec.
Trigger kB per sec. /Equipment/Trigger/Statistics/kBytes per sec.

[local:midas:S]/>cd /History/Links/System/
[local:midas:S]System>ls -l
Key name            Type  #Val Size Last Opn Mode Value
-----------------------------------------------------------
Trigger per sec.    LINK  1    46   >99d 0    RWD  /Equipment/Trigger/Statistics/Events per sec.
Trigger kB per sec. LINK  1    46   >99d 0    RWD  /Equipment/Trigger/Statistics/kBytes per sec.
```

A second sub-tree is added to the /History by the mhttpd task Midas web server when the button "History" on the main status page is pressed.

```
[local:midas:S]/History>ls -l -r Display
Key name                        Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
Display                         DIR
    Trigger rate                DIR
        Variables               STRING  2     32    36h  0   RWD
                                        [0]             System:Trigger per sec.
                                        [1]             System:Trigger kB per sec.
        Factor                  FLOAT   2     4     36h  0   RWD
                                        [0]             1
                                        [1]             1
        Timescale               INT     1     4     36h  0   RWD  3600
        Zero ylow               BOOL    1     4     36h  0   RWD  y
```

This define a default history display under the Midas web server as long as the reference to "System" is correct (see History System for more information regarding explanation on these fields.

Where the 2 trigger fields are symbolic links to the given path. The sub-tree **System** defines a "virtual" equipment and get by the system assigned a particular "History Event ID".

5.2.7 ODB /Alarms Tree

This branch contains system information related to alarms. Currently the overall alarm is checked once every minute. Once the alarm has been triggered, the message associated to the alarm can be repeated at a different rate. The structure is split in 2 sections. The **"Alarms"** itself which define the condition to be tested and the **"Classes"** which defines the action to be taken when the alarm occurs. In order to make the system flexible, beside some default message logging (Classes/Write system message), each action may have a particular "detached script" spawned by it (Classes/Execute command).

```
odb -e expt -h host
[host:expt:Stopped]/Alarms>ls -lr
Key name                        Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
Alarms                          DIR
    Alarm system active         BOOL    1     4     6h   0   RWD  n
    Alarms                      DIR
        Test                    DIR
            Active              BOOL    1     4     31h  0   RWD  n
            Triggered           INT     1     4     31h  0   RWD  0
            Type                INT     1     4     31h  0   RWD  3
            Check interval      INT     1     4     31h  0   RWD  60
            Checked last        DWORD   1     4     31h  0   RWD  0
            Time triggered firstSTRING  1     32    31h  0   RWD
            Time triggered last STRING  1     32    31h  0   RWD
            Condition           STRING  1     256   31h  0   RWD  /Runinfo/Run number > 10
            Alarm Class         STRING  1     32    31h  0   RWD  Alarm
            Alarm Message       STRING  1     80    31h  0   RWD  Run number became too large
        wc3_anode               DIR
            Active              BOOL    1     4     31h  0   RWD  n
            Triggered           INT     1     4     31h  0   RWD  0
            Type                INT     1     4     31h  0   RWD  3
            Check interval      INT     1     4     31h  0   RWD  10
            Checked last        DWORD   1     4     31h  0   RWD  958070825
            Time triggered firstSTRING  1     32    31h  0   RWD
            Time triggered last STRING  1     32    31h  0   RWD
            Condition           STRING  1     256   31h  0   RWD  /equipment/chv/variables/chvv[6] < 900
            Alarm Class         STRING  1     32    31h  0   RWD  Alarm
            Alarm Message       STRING  1     80    31h  0   RWD  W c 3 Anode voltage is too low
        chaos                   DIR
            Active              BOOL    1     4     31h  0   RWD  n
            Triggered           INT     1     4     31h  0   RWD  0
            Type                INT     1     4     31h  0   RWD  3
            Check interval      INT     1     4     31h  0   RWD  10
            Checked last        DWORD   1     4     31h  0   RWD  0
            Time triggered firstSTRING  1     32    31h  0   RWD
            Time triggered last STRING  1     32    31h  0   RWD
            Condition           STRING  1     256   31h  0   RWD  /Equipment/B12Y/Variables/B12Y[2] < 3000
            Alarm Class         STRING  1     32    31h  0   RWD  Alarm
            Alarm Message       STRING  1     80    31h  0   RWD  CHAOS magnet has tripped.
    Classes                     DIR
        Alarm                   DIR
            Write system messageBOOL    1     4     31h  0   RWD  y
            Write Elog message  BOOL    1     4     31h  0   RWD  n
            System message interINT     1     4     31h  0   RWD  60
            System message last DWORD   1     4     31h  0   RWD  0
            Execute command     STRING  1     256   31h  0   RWD
            Execute interval    INT     1     4     31h  0   RWD  0
            Execute last        DWORD   1     4     31h  0   RWD  0
            Stop run            BOOL    1     4     31h  0   RWD  n
        Warning                 DIR
            Write system messageBOOL    1     4     31h  0   RWD  y
            Write Elog message  BOOL    1     4     31h  0   RWD  n
            System message interINT     1     4     31h  0   RWD  60
System message last DWORD   1     4     31h  0   RWD  0
Execute command     STRING  1     256   31h  0   RWD
Execute interval    INT     1     4     31h  0   RWD  0
Execute last        DWORD   1     4     31h  0   RWD  0
Stop run            BOOL    1     4     31h  0   RWD  n
```

Alarm system active Overall Alarm enable flag.

Alarms Sub-tree defining each individual alarm condition.

Classes Sub-tree defining each individual action to be performed by a pre-defined and requested alarm.

5.2.8

ODB /Script Tree

This branch permits to invoke scripts from the web page. By creating the ODB tree **/Script** every entry in that tree will be available on the Web status page with the name of the key. Each key entry is then composed with a list of ODB field (or links). The first ODB field should be the executable command followed by as many arguments as you wish to be passed to the script.

```
[host::expt:Stopped]/Script>ls
BNMR Hold
Continue
Real
Test
Kill
[host:expt:Stopped]/Script>ls -lr Continue
Key name           Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------
Continue           DIR
    cmd            STRING  1     128   39h  0   RWD /home/bnmr/perl/continue.pl
    Name           STRING  1     32    28s  0   RWD  bnmr1
    hold           BOOL    1     4     31h  0   RWD  n
```

5.2.9

ODB /Elog Tree

This branch decribes the Elog settings used through the Midas web server (mhttpd task) for setting up the different Elog page display.

```
[local:midas:S]/Elog>ls -lr
Key name                        Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
```

```
Elog                            DIR
    Email                       STRING  1     64    25h  0   RWD  midas@triumf.ca
    Display run number          BOOL    1     4     25h  0   RWD  y
    Allow delete                BOOL    1     4     25h  0   RWD  n
    Types                       STRING  20    32    25h  0   RWD
                                        [0]             Routine
                                        [1]             Shift summary
                                        [2]             Minor error
                                        [3]             Severe error
                                        [4]             Fix
                                        [5]             Question
                                        [6]             Info
                                        [7]             Modification
                                        [8]             Reply
                                        [9]             Alarm
                                        [10]            Test
                                        [11]            Other
                                        [12]
                                        [13]
                                        [14]
                                        [15]
                                        [16]
                                        [17]
                                        [18]
                                        [19]
    Systems                     STRING  20    32    25h  0   RWD
                                        [0]             General
                                        [1]             DAQ
                                        [2]             Detector
                                        [3]             Electronics
                                        [4]             Target
                                        [5]             Beamline
                                        [6]
                                        [7]
                                        [8]
                                        [9]
                                        [10]
                                        [11]
                                        [12]
                                        [13]
                                        [14]
                                        [15]
                                        [16]
                                        [17]
                                        [18]
                                        [19]
    SMTP host                   STRING  1     64    25h  0   RWD  trmail.triumf.ca
```

Email Defines the Email address for Elog reply.

Display run number Allows to disable the run number display in the Elog entries.

Allow delete Flag for permiting the deletion of Elog entry.

Types Pre-defined types displayed when composing an Elog entry. A maximum of 20 types are available. The list will be terminated by the encounter of the first blank type.

Systems Pre-defined categories displayed when composing an Elog entry. A maximum of 20 types are available. The list will be terminated by the encounter of the first blank type.

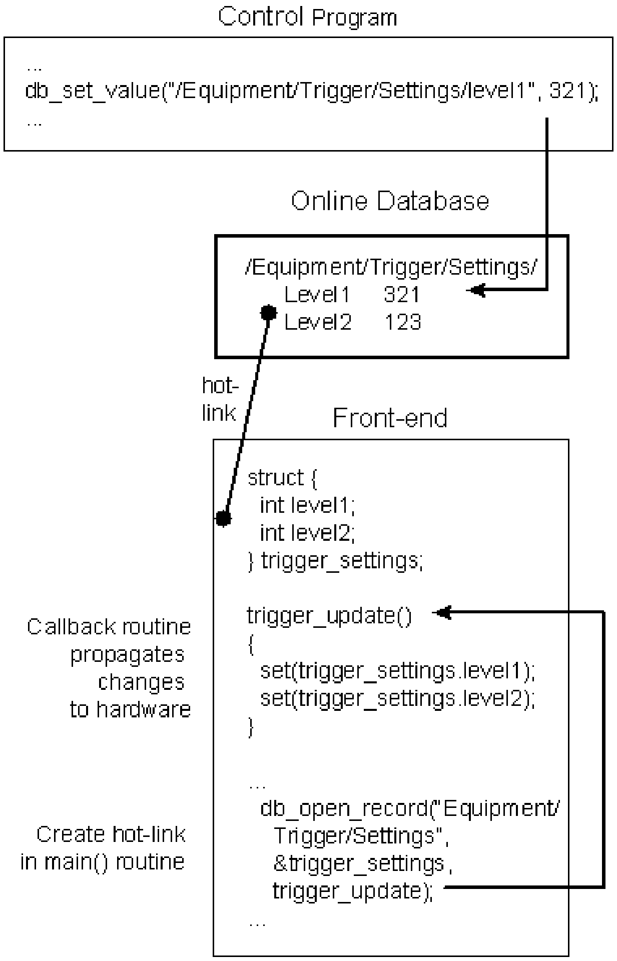
SMTP host Mail server address for routing the composed Elog message to the destination.

5.3

Hot Link

-> Next History System

It is often desirable to modify hardware parameters like discriminator levels or trigger logic connected to the frontend computer. Given the according hardware is accessible from the frontend code, theses parameters are easily controllable when a hot-link ODB is established between the frontend and the ODB itself.


Control Program
Online Database
hot-
link
Front-end
Callback routine propagates changes to hardware
Create hot-link in main() routine

```
...
db_set_value("/Equipment/Trigger/Settings/level1", 321);
...
```

```
/Equipment/Trigger/Settings/
     Level1    321
     Level2    123
```

```
struct {
  int level1;
  int level2;
} trigger_settings;

trigger_update()
{
  set(trigger_settings.level1);
  set(trigger_settings.level2);
}

...
  db_open_record("Equipment/
    Trigger/Settings",
    &trigger_settings,
    trigger_update);
...
```

Figure 3: *Changes in the ODB get propagated to the hardware by the frontend program.*

First the parameters have to be defined in the ODB under the Settings tree for the given equipment. Let's assume we have two discriminator levels belonging to the trigger electronics, which should be controllable. Following commands define these levels in the ODB:

```
[local]/>cd /Equipment/Trigger/
[local]Trigger>create key Settings
[local]Trigger>cd Settings
[local]Settings>create int level1
[local]Settings>create int level2
[local]Settings>ls
```

The frontend can now map a C structure to these settings. In order to simplify this process, ODBEdit can be requested to generate a header file containing this C structure. This file is usually called event.h. It can be generated in the current directory with the ODB command **make** which generates in the current directory the header file **experim.h**:

```
[local]Settings>make
```

Now this file can be copied to the frontend directory and included in the frontend source code. It contains a section with a C structure of the trigger settings and an ASCII representation:

```
typedef struct {
  INT       level1;
  INT       level2;
} TRIGGER_SETTINGS;

#define TRIGGER_SETTINGS_STR(_name) char *_name[] = {\
"[.]",\
"level1 = INT : 0",\
"level2 = INT : 0",\
"",\
NULL }
```

This definition can be used to define a C structure containing the parameters in frontend.c:

```
#include <experim.h>

TRIGGER_SETTINGS trigger_settings;
```

A hot-link between the ODB values and the C structure is established in the frontend_init() routine:

```
INT frontend_init()
{
HNDLE hDB, hkey;
TRIGGER_SETTINGS_STR(trigger_settings_str);

  cm_get_experiment_database(&hDB, NULL);

  db_create_record(hDB, 0,
    "/Equipment/Trigger/Settings",
    strcomb(trigger_settings_str));

  db_find_key(hDB, 0,
    "/Equipment/Trigger/Settings", &hkey);
  if (db_open_record(hDB, hkey,
      &trigger_settings,
      sizeof(trigger_settings), MODE_READ,
      trigger_update) != DB_SUCCESS)
    {
    cm_msg(MERROR, "frontend_init",
      "Cannot open Trigger Settings in ODB");
    return -1;
    }
  return SUCCESS;
}
```

The db_create_record() function re-creates the settings sub-tree in the ODB from the ASCII representation in case it has been corrupted or deleted. The db_open_record() now establishes the hot-link between the settings in the ODB and the trigger_settings structure. Each time the ODB settings are modified, the changes are written to the trigger_settings structure and the callback routine trigger_update() is executed afterwards. This routine has the task to set the hardware according to the settings in the trigger_settings structure.

It may look like:

```
void trigger_update(INT hDB, INT hkey)
{
  printf("New levels: %d %d",
    trigger_settings.level1,
    trigger_settings.level2);
}
```

Of course the printf() function should be replaced by a function which accesses the hardware properly. Modifying the trigger values with ODBEdit can test the whole scheme:

```
[local]/>cd /Equipment/Trigger/Settings
[local]Settings>set level1 123
[local]Settings>set level2 456
```

Immediately after each modification the frontend should display the new values. The settings can be saved to a file and loaded back later:

```
[local]/>cd /Equipment/Trigger/Settings
[local]Settings>save settings.odb
[local]Settings>set level1 789
[local]Settings>load settings.odb
```

The settings can also be modified from any application just by accessing the ODB. Following listing is a complete user application that modifies the trigger level:

```
#include <midas.h>

main()
{
HNDLE hDB;
INT   level;

  cm_connect_experiment("", "Sample", "Test",
                        NULL);
  cm_get_experiment_database(&hDB, NULL);
  level = 321;
  db_set_value(hDB, 0,
    "/Equipment/Trigger/Settings/Level1",
    &level, sizeof(INT), 1, TID_INT);

  cm_disconnect_experiment();
}
```

The following figure summarizes the involved components:

To make sure a hot-link exists, one can use the ODBEdit command **sor** (show open records):

```
[local]Settings>cd /
[local]/>sor
/Equipment/Trigger/Settings open 1 times by ...
```

5.4 History System

-> Next Alarm System

The history system is an add-on capability build in the data logger (see mlogger task) to record information in parallel to the data logging. This information is recorded with a time stamp and saved into "data base file" like for later retrieval. One set of file is created per day containing all the requested history events.

The history is working only if the logger is running, but it is not necessary to have any channel enabled.

The definition of the history event is done through two different means:

1. **frontend history event:** Each equipment has the capability to generate "history data" if the particular history field value is different then zero. The value will reflect the periodicity of the history logging (see The Equipment structure).
2. **"Virtual History event":** Composed within the Online Database under the specific tree "/History/Links" (see ODB /History Tree)

Both definition mode takes effects when the data logger gets a "start run" transition. Any modification during the run is not applied until the next run is started.

frontend history event As mentioned earlier, each equipment can be enabled to generate history event based on the periodicity of the history value (in second!). The content if the event will be completely copied into the history files using the definition of the event as tag names for every element of the event.

The history variable name for each element of the event is composed following one of the rules below:

bank event **/equipment/<...>/Variables/<bank name>**[] is the only reference of the event, the history name is composed of the bank name follwed by the corresponding index of the element.

bank event **/equipment/<...>/Settings/Names <bank_name>**[] is present, the history name is composed of the corresponding name found in the "Names <bank_name>" array. The size of this array should match the size of the **/equipment/<...>/Variables/<bank name**[]>.

```
[host:chaos:Running]Target>ls -l -r
Key name                        Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
Target                          DIR
    settings                    DIR
        Names TGT_              STRING  7     32    10h  0   RWD
                                        [0]             Time
                                        [1]             Cryostat vacuum
                                        [2]             Heat Pipe pressure
                                        [3]             Target pressure
                                        [4]             Target temperature
                                        [5]             Shield temperature
                                        [6]             Diode temperature
    Common                      DIR
...
    Variables                   DIR
        TGT_                    FLOAT   7     4     10s  0   RWD
                                        [0]             114059
                                        [1]             4.661
                                        [2]             23.16
                                        [3]             -0.498
                                        [4]             22.888
                                        [5]             82.099
                                        [6]             40
    Statistics                  DIR
...
```

fixed event The names of the individual element under **/equipment/<...>/variables/** will be used for the history name composition.

fixed event with array If the **/equipment/<...>/Settings/Names**[] exists, each element of the array will be referenced using the corresponding name of the **/Settings/Names**[] array.

ODB history event

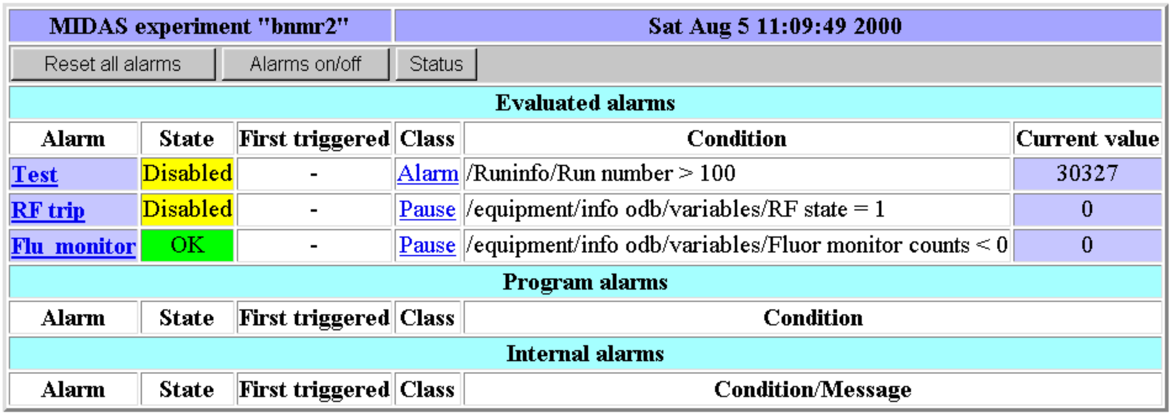
5.5 Alarm System

-> Next Slow Control System

The alarm system is built in and part of the main experiment scheduler. This means no separate task is necessary to beneficate from it, but this feature is active during **ONLINE** mode **ONLY**. Alarm setup and activation is done through the Online DataBase. Alarm system includes several other features such as: sequencing control of the experiment. The alarm capabilities are:

- Alarm setting on any ODB variables against threshold parameter.
- Alarm check frequency
- Alarm trigger frequency
- Customizable alarm scheme, under this scheme multiple choice of alarm type can be selected.
- Program control on run transition.

Beside the setup through ODBEdit, the Alarm can also be setup through the Midas web page..



Evaluated alarms

Alarm	State	First triggered	Class	Condition	Current value
Test	Disabled	-	Alarm	/Runinfo/Run number > 100	30327
RF trip	Disabled	-	Pause	/equipment/info odb/variables/RF state = 1	0
Flu_monitor	OK	-	Pause	/equipment/info odb/variables/Fluor monitor counts < 0	0

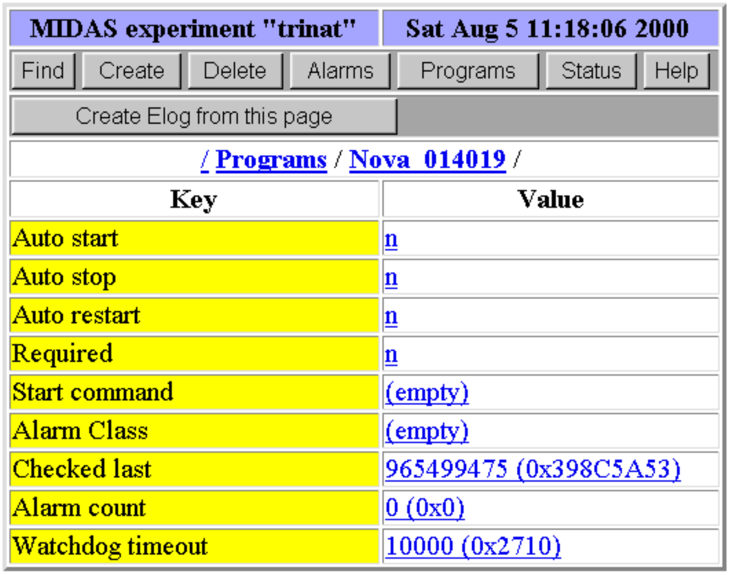
Program alarms

Alarm	State	First triggered	Class	Condition

Internal alarms

Alarm	State	First triggered	Class	Condition/Message

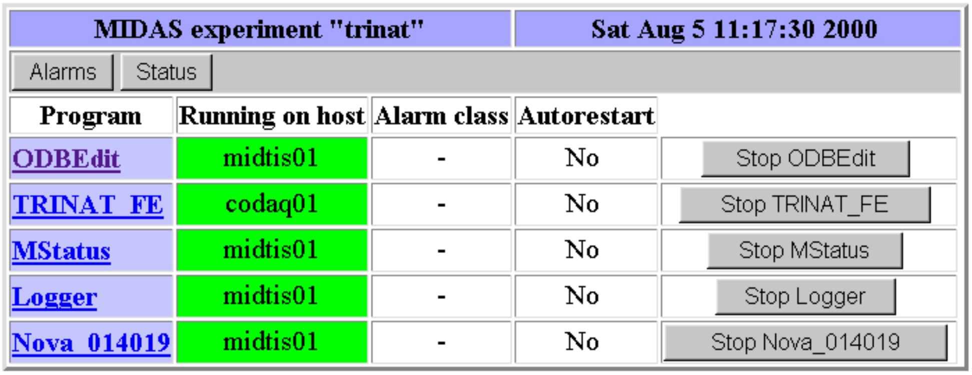
Figure 4: *Midas Web Alarm setting display.*



/ Programs / Nova_014019 /

Key	Value
Auto start	n
Auto stop	n
Auto restart	n
Required	n
Start command	(empty)
Alarm Class	(empty)
Checked last	965499475 (0x398C5A53)
Alarm count	0 (0x0)
Watchdog timeout	10000 (0x2710)

Figure 5: *Midas Web Alarm Program setting display.*



MIDAS experiment "trinat"				Sat Aug 5 11:17:30 2000
Alarms Status				
Program	Running on host	Alarm class	Autorestart	
ODBEdit	midtis01	-	No	Stop ODBEdit
TRINAT_FE	codaq01	-	No	Stop TRINAT_FE
MStatus	midtis01	-	No	Stop MStatus
Logger	midtis01	-	No	Stop Logger
Nova_014019	midtis01	-	No	Stop Nova_014019

Figure 6: *Midas Web Alarm Program status display.*

5.6

Slow Control System

-> Next Electronic Logbook

Instead of talking directly to each other, frontends and control programs exchange information through the ODB. Each slow control equipment gets a corresponding ODB tree under /Equipment. This tree contains variables needed to control the equipment as well as variables measured by the equipment. In case of a high voltage equipment this is a Demand array with contains voltages to be set, a Measured array which contains read back voltages and a Current array which contains the current drawn from each channel. To change the voltage of a channel, a control program writes to the Demand array the desired value. This array is connected to the high voltage frontend via a ODB hot-link. Each time it gets modified, the frontend receives a notification and sets the new value. In the other direction the frontend continuously reads the voltage and current values from all channels and updates the according ODB arrays if there has been a significant change. This design has a possible inconvenience due to fact that ODB is the key element of that control. Any failure or corruption of the database can results in wrong driver control. Therefore it is not recommended to use this system to control systems that need redundancy for safety purposes. On the other hand this system has several advantages:

- The control program does not need any knowledge of the frontends, it only talks to the ODB.
- The control variables only exist at one place that guarantees consistency between all clients.
- Basic control can be done through ODBEdit without the need of a special control program.
- A special control program can be tested without having a frontend running.
- In case of n frontends and m control programs, only n+m network connections are needed instead of n*m connection for point-to-point connections.

Since all slow control values are contained in the ODB, they get automatically dumped to the logging channels. The slow control frontends use the same framework as the normal frontends and behave similar in many respects. They also create periodic events that contain the slow control variables and are logged together with trigger and scaler events. The only difference is that a routine is called periodically from the framework that has the task to read channels and to update the ODB. To access slow control hardware, a two-layer driver concept is used. The upper layer is a ”class driver”, which establishes the connection to the ODB variables and contains high level functionality like channel limits, ramping etc. It uses a ”device driver” to access the channels. These drivers implement only very simple commands like ”set channel” and ”read channel”. The device drivers themselves can use bus drivers like RS232 or GPIB to control the actual device.

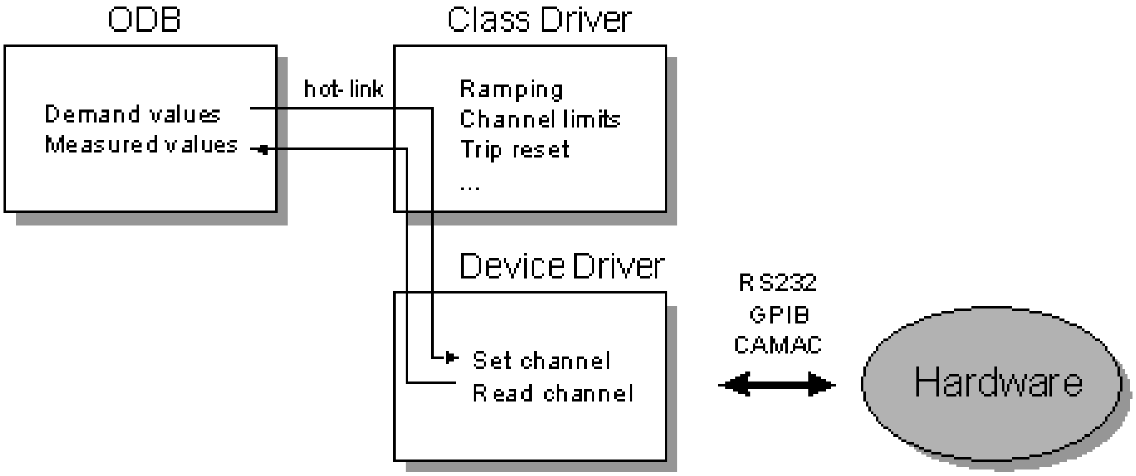


Figure 7: *Class driver and Device driver in the slow control system.*

The separation into class and device drivers has the advantage that it is very easy to add new devices, because only the simple device driver needs to be written. All higher functionality is inherited from the class driver. he device driver can implement richer functionality, depending on the hardware. For some high voltage devices there is a current read-back for example. This is usually reflected by additional variables in the ODB, i.e. a Current array. Frontend equipment uses exactly one class driver, but a class driver can use more than one device driver. This makes it possible to control several high voltage devices for example with one frontend in one equipment. The number of channels for each device driver is defined in the slow control frontend. Several equipment with different class drivers can be defined in a single frontend.

```
Key name                        Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
Epics                           DIR
    Settings                    DIR
        Channels                DIR
            Epics               INT     1     4     25h  0   RWD  3
        Devices                 DIR
            Epics               DIR
                Channel name    STRING  10    32    25h  0   RWD
                                        [0]             GPS:VAR1
                                        [1]             GPS:VAR2
                                        [2]             GPS:VAR3
        Names                   STRING  10    32    17h  1   RWD
                                        [0]             Current
                                        [1]             Voltage
                                        [2]             Watchdog
        Update Threshold MeasureFLOAT   10    4     17h  0   RWD
                                        [0]             2
```

```
                                    [1]             2
                                    [2]             2
Common                      DIR
    Event ID                WORD    1     2     17h  0   RWD  3
    Trigger mask            WORD    1     2     17h  0   RWD  0
    Buffer                  STRING  1     32    17h  0   RWD  SYSTEM
    Type                    INT     1     4     17h  0   RWD  4
    Source                  INT     1     4     17h  0   RWD  0
    Format                  STRING  1     8     17h  0   RWD  FIXED
    Enabled                 BOOL    1     4     17h  0   RWD  y
    Read on                 INT     1     4     17h  0   RWD  121
    Period                  INT     1     4     17h  0   RWD  60000
    Event limit             DOUBLE  1     8     17h  0   RWD  0
    Num subevents           DWORD   1     4     17h  0   RWD  0
    Log history             INT     1     4     17h  0   RWD  1
    Frontend host           STRING  1     32    17h  0   RWD  hostname
    Frontend name           STRING  1     32    17h  0   RWD  Epics
    Frontend file name      STRING  1     256   17h  0   RWD  feepic.c
Variables                   DIR
    Demand                  FLOAT   10    4     0s   1   RWD
                                    [0]             1.56
                                    [1]             120
                                    [2]             87
    Measured                FLOAT   10    4     2s   0   RWD
                                    [0]             1.56
                                    [1]             120
                                    [2]             87
Statistics                  DIR
    Events sent             DOUBLE  1     8     17h  0   RWDE 26
    Events per sec.         DOUBLE  1     8     17h  0   RWDE 0
    kBytes per sec.         DOUBLE  1     8     17h  0   RWDE 0
```

5.7 Electronic Logbook

The Electronic logbook is an alternative way of recording experiment information. This is implemented through the Midas web server mhttpd task. The definition of the options are implemented within the ODB data base under ODB /Elog Tree.

5.8 Log file

-> Utilities

Midas provides a general log file **midas.log** for recording system and user messages across the different components of the data acquisition clients. The location of this file is dependent on the mode of installation of the system.

hout /ODB /Logger Tree In this case the location is defined by either the **MIDAS_DIR** environment (see Environment variables) or the definition of the experiment in the **exptab** file (see Defining an Experiment). In both case the log file will be in the experiment specific directory.

with /Logger Tree The **midas.log** will be sitting into the defined directory specified by **Data Dir**.

midas.log file will contains system and user messages generated by any application connected to the given experiment.

The Message Macros definition provides a list of possible type of messages.

```
Fri Mar 24 10:48:40 2000 [CHAOS] Run 8362 started
Fri Mar 24 10:48:40 2000 [Logger] Run #8362 started
Fri Mar 24 10:55:04 2000 [Lazy_Tape] cni-043[10] (cp:383.6s) /dev/nst0/run08360.ybs 849.896MB file NEW
Fri Mar 24 11:24:03 2000 [MStatus] Program MStatus on host umelba started
Fri Mar 24 11:24:03 2000 [MStatus] Program MStatus on host umelba stopped
Fri Mar 24 11:27:02 2000 [Logger] stopping run after having received 1200000 events
Fri Mar 24 11:27:03 2000 [CHAOS] Run 8362 stopped
Fri Mar 24 11:27:03 2000 [SUSIYBOS] saving info in run log
Fri Mar 24 11:27:03 2000 [Logger] Run #8362 stopped
Fri Mar 24 11:27:13 2000 [Logger] starting new run
Fri Mar 24 11:27:14 2000 [CHAOS] Run 8363 started
Fri Mar 24 11:27:14 2000 [CHAOS] odb_access_file -I- /Equipment/kos_trigger/Dump not found
Fri Mar 24 11:27:14 2000 [Logger] Run #8363 started
Fri Mar 24 11:33:47 2000 [Lazy_Tape] cni-043[11] (cp:391.8s) /dev/nst0/run08361.ybs 850.209MB file NEW
Fri Mar 24 11:42:35 2000 [CHAOS] Run 8363 stopped
Fri Mar 24 11:42:40 2000 [SUSIYBOS] saving info in run log
Fri Mar 24 11:42:41 2000 [ODBEdit] Run #8363 stopped
Fri Mar 24 12:19:57 2000 [MChart] client [umelba.Triumf.CA]MChart failed watchdog test after 10 sec
Fri Mar 24 12:19:57 2000 [MChart] Program MChart on host koslx0 stopped
```

6 Utilities

The Midas applications.

Names

The Midas system provides several off-the-shelf programs to control, monitor, debug the data aquisition system. Starting with the main utility (odbedit) which provide access to the Online data base and run control.

6.1 odbedit task

Online DataBase Editor.

odbedit referes to the Online DataBase Editor. This is the main application to interact with the different components of the Midas system.

- Arguments

-h : help.

-h hostname :Specifies host to connect to. Must be a valid IP host name. This option supersedes the MIDAS_SERVER_HOST environment variable.

-e exptname :Specifies the experiment to connect to. This option supersedes the MIDAS_EXPT_NAME environment variable.

-c command :Perform a single command. Can be used to perform operations in script files.

-c @commandFile :Perform commands in sequence found in the commandFile.

-s size : size in byte (for creation). Specify the size of the ODB file to be created when no share file is present in the experiment directory (default 128KB).

-d ODB tree :Specify the initial entry ODB path to go to.

- Usage ODBEdit is the MIDAS run control program. It has a simple command line interface with command line editing similar to the UNIX tcsh shell. Following edit keys are implemented:

Backspace	Erase character left from cursor
Delete/Ctrl-D	Erase character under cursor
Ctrl-W/Ctrl-U	Erase current line
Ctrl-K	Erase line from cursor to end
Left arrow/Ctrl-B	Move cursor left
Right arrow/Ctrl-F	Move cursor right
Home/Ctrl-A	Move cursor to beginning of line
End/Ctrl-E	Move cursor to end of line
Up arrow/Ctrl-P	Recall previous command
Down arrow/Ctrl-N	Recall next command
Ctrl-F	Find most recent command which starts with current line
Tab/Ctrl-I	Complete directory. The command **ls /Sy<tab>** yields to **ls /System.**

ODBEdit treats the hierarchical online database very much like a file system. Most commands are similar to UNIX file commands like ls, cd, chmod, ln etc. The help command displays a short description of all commands.

```
[local:midas:Stopped]/>help
Database commands ([] are options, <> are placeholders):

alarm                   - reset all alarms
cd <dir>                - change current directory
chat                    - enter chat mode
chmod <mode> <key>      - change access mode of a key
                          1=read | 2=write | 4=delete
cleanup                 - delete hanging clients
copy <src> <dest>       - copy a subtree to a new location
create <type> <key>     - create a key of a certain type
create <type> <key>[n]  - create an array of size [n]
del/rm [-l] [-f] <key>  - delete a key and its subkeys
```

```
    -l                      follow links
    -f                      force deletion without asking
  exec <key>/<cmd>        - execute shell command (stored in key) on server
  find <pattern>          - find a key with wildcard pattern
  help/? [command]        - print this help [for a specific command]
  hi [analyzer] [id]      - tell analyzer to clear histos
  ln <source> <linkname>  - create a link to <source> key
  load <file>             - load database from .ODB file at current position
  ls/dir [-lhvrp] [<pat>] - show database entries which match pattern
    -l                      detailed info
    -h                      hex format
    -v                      only value
    -r                      show database entries recursively
    -p                      pause between screens
  make [analyzer name]    - create experim.h
  mem                     - show memeory usage
  mkdir <subdir>          - make new <subdir>
  move <key> [top/bottom/[n]] - move key to position in keylist
  msg [user] <msg>        - compose user message
  old                     - display old messages
  passwd                  - change MIDAS password
  pause                   - pause current run
  pwd                     - show current directory
  resume                  - resume current run
  rename <old> <new>      - rename key
  rewind [channel]        - rewind tapes in logger
  save [-c -s] <file>     - save database at current position
                            in ASCII format
    -c                      as a C structure
    -s                      as a #define'd string
  set <key> <value>       - set the value of a key
  set <key>[i] <value>    - set the value of index i
  set <key>[*] <value>    - set the value of all indices of a key
  set <key>[i..j] <value> - set the value of all indices i..j
  scl [-w]                - show all active clients [with watchdog info]
  shutdown <client>/all   - shutdown individual or all clients
  sor                     - show open records in current subtree
  start [number]          - start a run [with a specific number]
  stop                    - stop current run
  trunc <key> <index>     - truncate key to [index] values
  ver                     - show MIDAS library version
  webpasswd               - change WWW password for mhttpd
  wait <key>              - wait for key to get modified
  quit/exit               - exit
```

- Examples

```
>odbedit -s 512000
>odbedit -c stop
>odbedit
 [hostxxx:exptxxx:Running]/> ls /equipment/trigger
```

6.2 mstat task

Midas Status display

mstat is a simple ASCII status display. It presents in a compact form the most valuable information of the current condition of the Midas Acquisition system. The display is composed at the most of 5 sections depending on the current status of the experiment. The sections displayed in order from top to bottom refer to:

1. Run information.
2. Equipment listing and statistics. if any frontend is active.
3. Logger information and statistics if mlogger is active.
4. Lazylogger status if lazylogger is active.
5. Client listing.

- Arguments

-h : help

-h hostname : host name (see odbedit task)

-e exptname : experiment name (see odbedit task)

-l : loop. Forces mstat to remain in a display loop. Enter "!" to terminate the command.

-w time : refresh rate in second. Specifies the delay in second before refreshing the screen with up to date information. Default: 5 seconds. Has to be used in conjunction with -l switch. Enter "R" to refresh screen on next update.

- Usage

```
 >mstat -l
*-v1.8.0- MIDAS status page -------------------------Mon Apr  3 11:52:52 2000-*
Experiment:chaos       Run#:8699    State:Running          Run time :00:11:34
Start time:Mon Apr  3 11:41:18 2000

FE Equip.   Node              Event Taken    Event Rate[/s] Data Rate[Kb/s]
B12Y        pcch02            67             0.0            0.0
CUM_Scaler  vwchaos           23             0.2            0.2
CHV         pcch02            68             0.0            0.0
KOS_Scalers vwchaos           330            0.4            0.6
KOS_Trigger vwchaos           434226         652.4          408.3
KOS_File    vwchaos           0              0.0            0.0
Target      pcch02            66             0.0            0.0

Logger Data dir: /scr0/spring2000            Message File: midas.log
Chan.   Active Type      Filename            Events Taken   KBytes Taken
  0     Yes    Disk      run08699.ybs        434206          4.24e+06

Lazy Label     Progress  File name           #files         Total
cni-53         100[%]    run08696.ybs        15             44.3[%]
```

```
Clients:  MStatus/koslx0        Logger/koslx0         Lazy_Tape/koslx0
          CHV/pcch02            MChart1/umelba        ODBEdit/koslx0
          CHAOS/vwchaos         ecl/koslx0            Speaker/koslx0
          MChart/umelba         targetFE/pcch02       HV_MONITOR/umelba
          SUSIYBOS/koslx0       History/kosal2        MStatus1/dasdevpc
*-----------------------------------------------------------------------------*
```

6.3 analyzer task

online / offline analyzer

Names

analyzer is the main online / offline event analysis application. **analyzer** uses fully the **ODB** capabilities as all the analyzer parameters are dynamically controllable from the Online Database editor **odbedit task**.

- Arguments

-h : help

-h hostname : host name (see odbedit task)

-e exptname : experiment name (see odbedit task)

-D : start program as a daemon (UNIX only).

-i <filename1> <filename2> : Input file name. May contain a '%05d' to be replaced by the run number. Up to ten input files can be specified in one "-i" statement.

-o <filename> : Output file name. Extension may be .mid (MIDAS binary), .asc (ASCII) or .rz (HBOOK). If the name contains a '%05d', one output file is generated for each run.

-r <range> : Range of run numbers to analyzer like "-r 120 125" to analyze runs 120 to 125 (inclusive). The "-r" flag must be used with a '%05d' in the input file name.

-n <count> : Analyze only "count" events.

-n <first> <last> : Analyze only events from "first" to "last".

-n <first> <last> <n> : Analyze every n-th event from "first" to "last".

-f : Filter events. Write original events to output file only if analyzer accepts them (doesn't return ANA_SKIP).

-c <filename1> <filename2> : Configuration file name(s). May contain a '%05d' to be replaced by the run number. Up to ten files can be specified in one "-c" statement.

-p <param=value> : Set individual parameters to a specific value. Overrides any setting in configuration files

-w : Produce row-wise N-tuples in outpur .rz file. By default, column-wise N-tuples are used.

-v : Verbose output.

-d : Debug flag when started the analyzer fron a debugger. Prevents the system to kill the analyzer when the debugger stops at a breakpoint

-q : Quiet flag. If set, don't display run progress in offline mode.

-l : If set, don't load histos from last.rz when running online.

-L : HBOOK LREC size. Default is 8190.

-P <ODB tree> Protect an ODB subtree from being overwritten with the online data when ODB gets loaded from .mid.

- Usage

```
>analyzer
>analyzer -D
>analyzer -i run00023.mid -o run00023.rz -w
>analyzer -i run%05d.mid -o runall.rz -r 23 75 -w
```

6.3.1 The MIDAS Analyzer

->Multi Stage Concept

Users can write their own analyzer from scratch or use the standard MIDAS analyzer framework which uses the HBOOK package for histogramming. Using the MIDAS analyzer framework has following advantages: Events are received automatically, only a user routine has to be written to process the events. This concept is similar to the frontend. The analyzer is structured into "stages", where each stage analyzes a part of the event and adds some calculated data to it, which can be read by later stages. This simplifies the design of complex analyzers. The analyzer framework can receive events from a MIDAS buffer (online analysis) or from a file (off-line-analysis) without recompilation. The analyzer framework can produce output files which may contain a combination of raw and analyzed data. Output files can be in different formats like HBOOK RZ files which can be directly analyzed with PAW. An ODB dump contained in a data file can be retrieved and copied to the current ODB. This ensures that the same configuration values are used online and off-line. Additionally, parameters can be overloaded from off-line configuration files. Several files can be analyzed off-line each having its own configuration file. While HBOOK histograms have to be booked and filled manually from the user code, N-tuples can be booked automatically from one or more banks. This works also online where "live" N-tuples can be used to monitor an experiment with PAW. The following paragraphs explain these features in more detail and show how to use them.

6.3.2

Multi Stage Concept

->Analyzer parameters

In order to make data analysis more flexible, a multi-stage concept has been chosen for the analyzer. A raw event is passed through several stages in the analyzer, where each stage has a specific task. The stages read part of the event, analyze it and can add the results of the analysis back to the event. Therefore each stage in the chain can read all results from previous stages. The first stages in the chain typically deal with data calibration, while the last stages contain the code which produces "physical" results like particle energies etc. The multi stage concept allows collaborations of people to use standard modules for the calibration stages which ensures that all members deal with the identical calibrated data, while the last stages can be modified by individuals to look at different aspects of the data. The stage system makes use of the MIDAS bank system. Each stage can read existing banks from an event and add more banks with calculated data. Following picture gives an example of an analyzer consisting of three stages where the first two stages make an ADC and a MWPC calibration, respectively. They add a "Calibrated ADC" bank and a "MWPC" bank which are used by the third stage which calculates angles between particles:

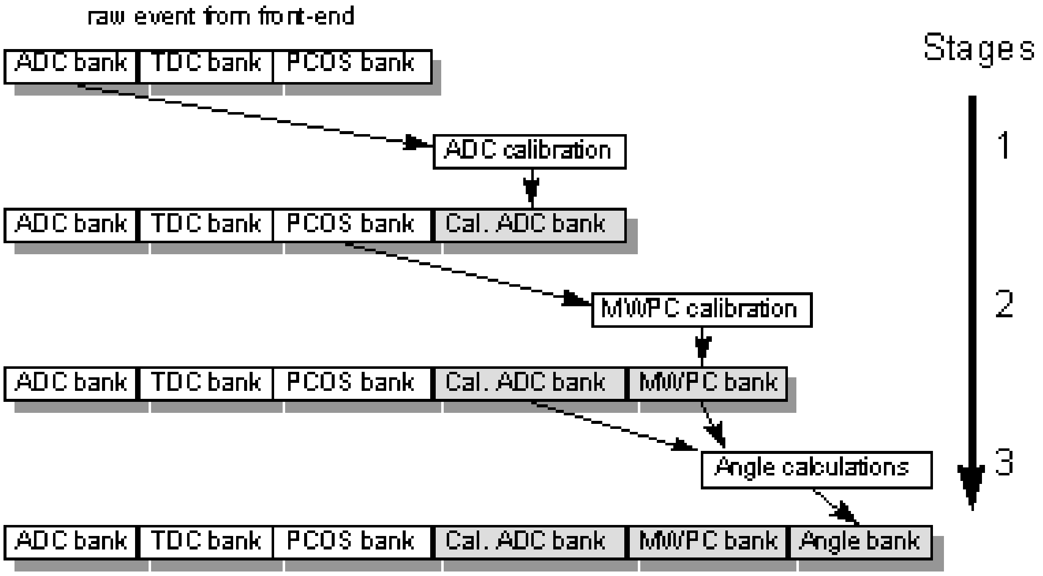


Figure 8: *Three stage analyzer.*

Example of a three stage analyzer Since data is contained in MIDAS banks, the system knows how to interpret the data. N-tuples can be booked automatically from any bank with a simple switch in the ODB. The user code for each stage is contained in a "module". Each module has a begin-of-run, end-of-run and an event routine. The BOR routine is typically used to book histograms, the EOR routine can do peak fitting etc. The event routine is called for each event that is received online or off-line.

6.3.3 Analyzer parameters

->ODB parameters for Analyzer

Each analyzer module can contain a set of parameters to control the behavior of the module or as configuration and calibration data. These parameters are kept in the ODB under /Analyzer/Parameters/<module name> and mapped automatically to C structures in the analyzer modules. Changing these values in the ODB can therefore control the analyzer. In order to keep the ODB variables and the C structure definitions matched, the ODBEdit command **make** generates the file **experim.h** which contains C structures for all analyzer parameters. If this file is included in all analyzer source code files, the parameters can be accessed under the name <module name>_param.

6.3.4 ODB parameters for Analyzer

->Writing the Code

When the analyzer is started for the first time, it will create a new tree in ODB. The default structure is composed of the following elements.

```
[host:expt:S]/Analyzer>ls -l
Key name                        Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
Parameters                      DIR
Output                          DIR
Book N-tuples                   BOOL    1     4     1m   0   RWD  y
Bank switches                   DIR
Module switches                 DIR
ODB Load                        BOOL    1     4     19h  0   RWD  n
Trigger                         DIR
Scaler                          DIR
```

Analyzer/Parameters This directory contains a default set of parameters which are passed to the modules. See the correspondence in the module.

```
[host:expt:S]/Analyzer>ls -lr Parameters
Key name                        Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
Parameters                      DIR
    ADC calibration             DIR
        Pedestal                INT     8     4     43m  0   RWD
                                        [0]             174
                                        [1]             194
                                        [2]             176
                                        [3]             182
                                        [4]             185
                                        [5]             215
                                        [6]          202
                                        [7]          202
        Software Gain           FLOAT   8     4     43m  0   RWD
                                        [0]          1
                                        [1]          1
                                        [2]          1
                                        [3]          1
                                        [4]          1
                                        [5]          1
                                        [6]          1
                                        [7]          1
        Histo threshold         DOUBLE  1     8     43m  0   RWD  20
    ADC summing                 DIR
        ADC threshold           FLOAT   1     4     43m  0   RWD  5
    Global                      DIR
        ADC Threshold           FLOAT   1     4     43m  0   RWD  5

---> file adccalib.c
#include "experim.h"
..
ANA_MODULE adc_calib_module = {
  "ADC calibration",             // module name
  "Stefan Ritt",                 // author
  adc_calib,                     // event routine
  adc_calib_bor,                 // BOR routine
  adc_calib_eor,                 // EOR routine
  adc_calib_init,                // init routine
  NULL,                          // exit routine
  &adccalib_param,               // parameter structure
  sizeof(adccalib_param),        // structure size
  adc_calibration_param_str,     // initial parameters
};
...
  // subtract pedestal
  for (i=0 ; i<n_adc ; i++)
    cadc[i] = (float) ((double)pdata[i] - adccalib_param.pedestal[i] + 0.5);

  // apply software gain calibration
  for (i=0 ; i<n_adc ; i++)
    cadc[i] *= adccalib_param.software_gain[i];

  // fill ADC histos if above threshold
  for (i=0 ; i<n_adc ; i++)
    if (cadc[i] > (float) adccalib_param.histo_threshold)
      HF1(ADCCALIB_ID_BASE+i, cadc[i], 1.f);
```

If more parameters are necessary, perform the following procedure:

1. modify/add new parameters in the current ODB.

   ```
   [host:expt:S]ADC calibration>set Pedestal[9] 3
   [host:expt:S]ADC calibration>set "Software Gain[9]" 3
   [host:expt:S]ADC calibration>create double "Upper threshold"
   [host:expt:S]ADC calibration>set "Upper threshold" 400
   [host:expt:S]ADC calibration>ls -lr
   ```

```
Key name                        Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
ADC calibration                 DIR
    Pedestal                    INT     10    4     2m   0   RWD
                                        [0]             174
                                        [1]             194
                                        [2]             176
                                        [3]             182
                                        [4]             185
                                        [5]             215
                                        [6]             202
                                        [7]             202
                                        [8]             0
                                        [9]             3
    Software Gain               FLOAT   10    4     2m   0   RWD
                                        [0]             1
                                        [1]             1
                                        [2]             1
                                        [3]             1
                                        [4]             1
                                        [5]             1
                                        [6]             1
                                        [7]             1
                                        [8]             0
                                        [9]             0
    Histo threshold             DOUBLE  1     8     53m  0   RWD  20
    Upper threshold             DOUBLE  1     4     3s   0   RWD  400
```

2. Generate **experim.h**

```
[host:expt:S]ADC calibration>make
"experim.h" has been written to /home/midas/online
```

3. Update the module with the new parameters.

```
---> adccalib.c
...
fill ADC histos if above threshold
for (i=0 ; i<n_adc ; i++)
if ((cadc[i] > (float) adccalib_param.histo_threshold)
 && (cadc[i] < (float) adccalib_param.upper_threshold))
    HF1(ADCCALIB_ID_BASE+i, cadc[i], 1.f);
```

4. rebuild the analyzer.

In the case global parameter is necessary for several modules, start by doing the step 1 & 2 from the enumeration above and carry on with the following procedure below:

1. Declare the parameter global in analyzer.c

```
// ODB structures
...
GLOBAL_PARAM     global_param;
...
```

2. Update ODB structure and open record for that parameter (hot link).

```
---> analyzer.c
...
```

```
sprintf(str, "/%s/Parameters/Global", analyzer_name);
db_create_record(hDB, 0, str, strcomb(global_param_str));
db_find_key(hDB, 0, str, &hKey);
if (db_open_record(hDB, hKey, &global_param
    , sizeof(global_param), MODE_READ, NULL, NULL) != DB_SUCCESS) {
  cm_msg(MERROR, "analyzer_init", "Cannot open \"%s\" tree in ODB", str);
  return 0;
}
```

3. Declare the parameter **extern** in the required module

```
---> adccalib.c
...
extern GLOBAL_PARAM  global_param;
...
```

Analyzer/Output Defines general analyzer behaviour and output data format.

```
[host:expt:S]Output>ls -l
Key name                        Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
Filename                        STRING  1     256   6m   0   RWD  run%05d.asc
RWNT                            BOOL    1     4     6m   0   RWD  n
Histo Dump                      BOOL    1     4     6m   0   RWD  n
Histo Dump Filename             STRING  1     256   6m   0   RWD  his%05d.rz
Clear histos                    BOOL    1     4     6m   0   RWD  y
Last Histo Filename             STRING  1     256   6m   0   RWD  last.rz
Events to ODB                   BOOL    1     4     6m   0   RWD  n
Global Memory Name              STRING  1     8     6m   0   RWD  ONLN
```

Output/Filename Analyzer data output ???? what are the other options?

Output/RWNT Raw Wise N-Tuple ???? but for online CW only?

Output/Histo Dump Enable the creation of a histogram save-set.

Output/Histo Dump Filename File name template for the histo dump. **Remark:** PAW++ browse the HBOOK automatically and therefore this field can be set to **his%05d.hbook**.

Output/Clear histos Enable the clearing of all histos at the begining of each run.

Output/Last Histo Filename Default name for the latest NTuple+histo file name. This file is read when the analyzer is restarted. **Remark:** that if the booking of the histograms has been changed in the analyzer, it is strongly suggested to remove "last.rz" file in order to prevent memory misalignment when restarting the analyzer.

Output/Events to ODB Enable the copy of the event to the ODB for debugging purpose.

Output/Global Memory Name Shared Memory name to allows PAW to attached to the N-Tuples and histograms.

Analyzer/Book N-tuples Enable the N-Tuple rebooking.

Analyzer/Bank switches Enable individual banks for N-Tuples generation.

```
[local:midas:S]/Analyzer>ls "Bank switches" -l
Key name                        Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
ADC0                            DWORD   1     4     1h   0   RWD  0
TDC0                            DWORD   1     4     1h   0   RWD  0
CADC                            DWORD   1     4     1h   0   RWD  0
ASUM                            DWORD   1     4     1h   0   RWD  0
SCLR                            DWORD   1     4     1h   0   RWD  0
ACUM                            DWORD   1     4     1h   0   RWD  0
```

Module switches Enable individual modules.

```
[local:midas:S]/Analyzer>ls "module switches" -l
Key name                        Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
ADC calibration                 BOOL    1     4     1h   0   RWD  y
ADC summing                     BOOL    1     4     1h   0   RWD  y
Scaler accumulation             BOOL    1     4     1h   0   RWD  y
```

Analyzer/ODB Load Enable the extraction of the ODB dump from the data file and the overwriting of the current ODB. This option is valid only during offline analysis. **Remark:** While taking data online (/Runinfo/Online mode = 1), if a offline analyzer is started it will overwite the online ODB even if the "ODB load" is disabled. Make sure you create a different experiment allocated strictly to the offline analyzer. The **Output** dir defines condition for the upcomming run.

Analyzer/Trigger Default analyzer module.

Analyzer/Scaler Default analyzer module.

6.3.5 Writing the Code

Names

->Online usage

An example analyzer is contained in the examples/experiment directory of the MIDAS distribution. The MIDAS analyzer framework mana.c is compiled and linked together with the main analyzer file analyzer.c which contains a list of analyzer modules. The source code files for the individual modules are adccalib.c, adcsum.c and scaler.c.

6.3.5.1 analyzer.c

The file analyzer.c contains the PAW common section which is defined with

PAWC_DEFINE(8000000);

This defines a section of 8 megabytes or 2 megawords. In case many histograms are booked in the user code, this value probably has to be increased in order not to crash HBOOK. If the analyzer runs online, the section is kept in shared memory. In case the operating system only supports a smaller amount of shared memory, this value has to be decreased. Next, the file contains the analyzer name

char *analyzer_name = "Analyzer";

under which the analyzer appears in the ODB (via the ODBEdit command scl). This also determines the analyzer root tree name as /Analyzer. In case several analyzers are running simultaneously (in case of distributed analysis on different machines for example), they have to use different names like Analyzer1 and Analyzer2 which then creates two separate ODB trees /Analyzer1 and /Analyzer2 which is necessary to control the analyzers individually. Following structures are then defined in analyzer.c: runinfo, global_param, exp_param and trigger_settings. They correspond to the ODB trees /Runinfo, /Analyzer/Parameters/Global, /Experiment/Run parameters and /Equipment/Trigger/Settings, respectively. The mapping is done in the analyzer_init() routine. Any analyzer module (via an extern statement) can use the contents of these structures. If the experiment parameters contain an flag to indicate the run type for example, the analyzer can analyze calibration and data runs differently. The module declaration section in analyzer.c defines two "chains" of modules, one for trigger events and one for scaler events. The framework calls these according to their order in these lists. The modules of type ANA_MODULE are defined in their source code file. The enabled flag for each module is copied to the ODB under /Analyzer/Module switches. By setting this flag zero in the ODB, modules can be disabled temporarily. Next, all banks have to be defined. This is necessary because the framework automatically books N-tuples for all banks at startup before any event is received. Online banks which come from the frontend are first defined, then banks created by the analyzer:

```
...
// online banks
{ "ADC0", TID_DWORD, N_ADC, NULL },
{ "TDC0", TID_DWORD, N_TDC, NULL },

// calculated banks
{ "CADC", TID_FLOAT, N_ADC, NULL },
{ "ASUM", TID_STRUCT, sizeof(ASUM_BANK),
  asum_bank_str },
```

The first entry is the bank name, the second the bank type. The type has to match the type which is created by the frontend. The type TID_STRUCT is a special bank type. These banks have a fixed length which matches a C structure. This is useful when an analyzer wants to access named variables inside a bank like asum_bank.sum. The third entry is the size of the bank in bytes in case of structured banks or the maximum number of items (not bytes!) in case of variable length banks. The last entry is the ASCII representation of the bank in case of structured banks. This is used to create the bank on startup under /Equipment/Trigger/Variables/<bank name>.

The next section in analyzer.c defines the ANALYZE_REQUEST list. This determines which events are received and which routines are called to analyze these events. A request can either contain an "analyzer routine" which is called to analyze the event or a "module list" which has been defined above. In the latter case all modules are called for each event. The requests are copied to the ODB under /Analyzer/<equipment name>/Common. Statistics like number of analyzed events is written under /Analyzer/<equipment name>/Statistics. This scheme is very similar to the frontend Common and Statistics tree under /Equipment/<equipment name>/. The last entry of the analyzer request determines the HBOOK buffer size for online N-tuples. The analyzer_init() and analyzer_exit() routines are called when the analyzer starts or exits, while the ana_begin_of_run() and ana_end_of_run() are called at the beginning and end of each run. The ana_end_of_run() routine in the example code writes a run log file runlog.txt which contains the current time, run number, run start time and number of received events.

6.3.5.2

<module.c>

Each module source code file defines itself in a ANA_MODULE structure which contains the module name, author, callback routines for events and run transitions, and a reference to the analyzer parameters for this module. In the BOR callback usually histograms are defined. The event routine reads banks from the event via bk_locate(), does its calculations, fills histograms and then creates calculated banks with bk_create()/bk_close() similar like the frontend. If a module returns 0 instead of SUCCESS, the event is not written to the output. This way event filtering might be implemented. To create new calculated values and parameters for an analyzer module, they first have to be created in the ODB. To create the calculated value new_sum in bank ASUM for module ADC summing, one enters in ODBEdit:

```
[local]/>cd /Equipment/Trigger/Variables/ASUM
[local]ASUM>cr float "New sum"
```

The parameter offset for module ADC summing is created with:

```
[local]/>cd /Analyzer/Parameters/ADC summing
[local]ADC summing>cr float Offset
```

The ODB command **make** now creates experim.h with these structures:

```
typedef struct {
  float     sum;
  float     new_sum;
} ASUM_BANK;

typedef struct {
  float     adc_threshold;
  float     offset
} ADC_SUMMING_PARAM;
```

The ASCII representations of these structures in event.h are used to create the ODB entries if they are not present. The new variables can now be used in the summing module like:

```
ASUM_BANK *asum;
  asum->new_sum = ... - adc_summing_param->offset;
```

6.3.6 Online usage

->Offline usage

Compile the analyzer as described in ???. To run the analyzer online, enter:

analyzer [**-h <host name>**] [**-e <exp name>**]

where <host name> and <exp name> are optional parameters to connect the analyzer to a remote back-end computer. This attaches the analyzer to the ODB, initializes all modules, creates the PAW shared memory and starts receiving events from the system buffer. Then start PAW and connect to the shared memory and display its contents

```
PAW > global_s onln
PAW > hist/list
    1  Trigger
    2  Scaler
 1000  CADC00
 1001  CADC01
 1002  CADC02
 1003  CADC03
 1004  CADC04
 1005  CADC05
 1006  CADC06
 1007  CADC07
 2000  ADC sum
```

For each equipment, a N-tuple is created with a N-tuple ID equal to the event ID. The CADC histograms are created from the adc_calib_bor() routine in adccalib.c. The N-tuple contents is derived from the banks of the trigger event. Each bank has a switch under /Analyzer/Bank switches. If the switch is on (1), the bank is contained in the N-tuple. The switches can be modified during runtime causing the N-tuples to be rebooked. The N-tuples can be plotted with the standard PAW commands:

```
PAW > nt/print 1
...
PAW > nt/plot 1.sum
PAW > nt/plot 1.sum cadc0>3000
```

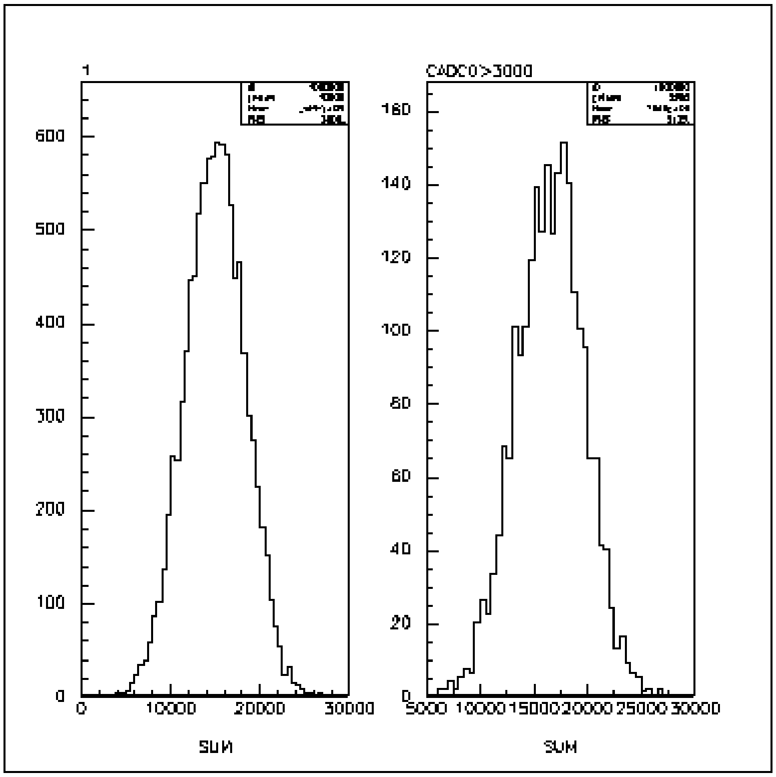


Figure 9: *PAW output for online N-tuples.*

While histograms contain the full statistics of a run, N-tuples are kept in a ring-buffer. The size of this buffer is defined in the ANALYZE_REQUEST structure as the last parameter. A value of 10000 creates a buffer which contains N-tuples for 10000 events. After 10000 events, the first events are overwritten. If the value is increased, it might be that the PAWC size (PAWC_DEFINE in analyzer.c) has to be increased, too. An advantage of keeping the last 10000 events in a buffer is that cuts can be made immediately without having to wait for histograms to be filled. On the other hand care has to be taken in interpreting the data. If modifications in the hardware are made during a run, events which reflect the modifications are mixed with old data. To clear the ring-buffer for a N-tuple or a histogram during a run, the ODBEdit command [**local**]/>**hi analyzer <id>**

where <id> is the N-tuple ID or histogram ID. An ID of zero clears all histograms but no N-tuples. The analyzer has two more ODB switches of interest when running online. The /Analyzer/Output/Histo Dump flag and /Analyzer/Output/Histo Dump Filename determine if HBOOK histograms are written after a run. This file contains all histograms and the last ring-buffer of N-tuples. It can be read in with PAW:

```
PAW >hi/file 1 run00001.rz
PAW > ldir
```

The /Analyzer/Output/Clear histos flag tells the analyzer to clear all histograms and N-tuples at the beginning of a run. If turned off, histograms can be accumulated over several runs.

6.3.7

Offline usage

->analyzer task

The analyzer can be used for off-line analysis without recompilation. It can read from MIDAS binary files (*.mid), analyze the data the same way as online, and the write the result to an output file in MIDAS binary format, ASCII format or HBOOK RZ format. If written to a RZ file, the output contains all histograms and N-tuples as online, with the difference that the N-tuples contain all events, not only the last 10000. The contents of the N-tuples can be a combination of raw event data and calculated data. Banks can be turned on and off in the output via the /Analyzer/Bank switches flags. Individual modules can be activated/deactivated via the /Analyzer/Module switches flags.

The RZ files can be analyzed and plotted with PAW. Following flags are available when the analyzer is started off-line:

-i [filename1] [filename2] ... Input file name(s). Up to ten different file names can be specified in a -i statement. File names can contain the sequence "%05d" which is replaced with the current run number in conjunction with the -r flag. Following filename extensions are recognized by the analyzer: .mid (MIDAS binary), .asc (ASCII data), .mid.gz (MIDAS binary gnu-zipped) and .asc.gz (ASCII data gnu-zipped). Files are un-zipped on-the-fly.

-o [filename] Output file name. The file names can contain the sequence "%05d" which is replaced with the current run number in conjunction with the -r flag. Following file formats can be generated: .mid (MIDAS binary), .asc (ASCII data), .rz (HBOOK RZ file), .mid.gz (MIDAS binary gnu-zipped) and .asc.gz (ASCII data gnu-zipped). For HBOOK files, CWNT are used by default. RWNT can be produced by specifying the -w flag. Files are zipped on-the-fly.

-r [range] Range of run numbers to be analyzed like -r 120 125 to analyze runs 120 to 125 (inclusive). The -r flag must be used with a "%05d" in the input file name.

-n [count] Analyze only count events. Since the number of events for all event types is considered, one might get less than count trigger events if some scaler or other events are present in the data.

-n [first] [last] Analyze only events with serial numbers between first and last.

-n [first] [last] [n] Analyze every n-th event from first to last.

-c [filename1] [filename2] ... Load configuration file name(s) before analyzing a run. File names may contain a "%05d" to be replaced with the run number. If more than one file is specified, parameters from the first file get superseded from the second file and so on. Parameters are stored in the ODB and can be read by the analyzer modules. They are conserved even after the analyzer has stopped. Therefore, only parameters which change between runs have to be loaded every time. To set a parameter like /Analyzer/Parameters/ADC summing/offset one would load a configuration file which contains:

```
[Analyzer/Parameters/ADC summing]
Offset = FLOAT : 123
```

Loaded parameters can be inspected with ODBEdit after the analyzer has been started.

-p [param=value] Set individual parameters to a specific value. Overrides any setting in configuration files. Parameter names are relative to the /Analyzer/Parameters directory. To set the key /Analyzer/Parameters/ADC summing/offset to a specific value, one uses -p "ADC summing/offset"=123. The quotation marks are necessary since the key name contains a blank. To specify a parameter which is not under the /Analyzer/Parameters tree, one uses the full path (including the initial "/") of the parameter like -p "/Experiment/Run Parameters/Run mode"=1.

-w Produce row-wise N-tuples in output RZ file. By default, column-wise N-tuples are used.

-v Convert only input file to output file. Useful for format conversions. No data analysis is performed.

-d Debug flag when started the analyzer from a debugger. Prevents the system to kill the analyzer when the debugger stops at a breakpoint.

6.4 mlogger task

Multi channel Data logger and history data collector.

mlogger is the main application to collect data from the different frontend under certain condition and store them onto physical device such as *disk* or *tape*. It also act as an history event collector if either the history flags is enabled in the frontend equipment (see The Equipment Structure or if the ODB tree /History/Links is defined (See History System). See the ODB /Logger Tree for reference on the Logger ODB tree structure.

- Arguments

-h	: help
-h hostname	: host name (see odbedit task)
-e exptname	: experiment name (see odbedit task)
-D	: start program as a daemon (UNIX only).

- Usage

```
>mlogger -D
```

- Remarks

1. As soon as the mlogger is running, the history is mechanism is enabled.
2. The data channels as well as the history logging is rescanned automatically at each "begin of run" transition. In other word, additional channel can be defined while running but effect will be taken place only at the following begin of run transition.
3. The default setting defines a data "Midas" format with a file name of the type "run%05d.mid". Make sure this is the requested setting for your experiment.
4. Once the mlogger is running, you should be able to monitor its state through the mstat task or through a web browser if the mhttpd task is running.

6.5 lazylogger task

Multi channel background data copier.

lazylogger is an application which decouples the data aquisition from the data logging mechanism. The need of such application has been dictated by the slow response time of some of the media logging devices (Tape devices). Delay due to tape mounting, retension, reposition imply that the data acquisition has to hold until operation completion. By using **mlogger** to log data to disk in a first stage and then using **lazylogger** to copy or move the stored files to the "slow device" we can keep the acquisition running without interruption.

- Multiple lazylogger can be running comtemporary on the same computer, each one taking care of a particular channel.
- Each lazylogger channel will have a dedicated ODB tree containg its own information.
- All the lazylogger channel will be under the ODB **/Lazy/<channel_name>/....**

- Each channel tree is composed of three sub-tree **Settings, Statistics, List**.

Self-explanatory the **Settings and Statistics** contains the running operation of the channel. While the **List** will have a dynamic list of run number which has been sucessfully manipulate by the Lazylogger channel. This list won't exists until the first successful operation of the channel is complete.

- Arguments

-h : help.

-h hostname : host name.

-e exptname : experiment name.

-D : start program as a daemon.

-c channel : logging channel. Specify the *lazylogger* to activate.

-z : zap statistics. Clear the statistics tree of all the defined *lazylogger* channels.

-t : talking messages. Enable some of the info messages to be tagged with the TALK option. In order to be spoken either mspeaker or mlxspeaker has to be running.

- ODB parameters (Settings/)

```
Settings                     DIR
    Maintain free space(%)   INT     1     4     3m   0    RWD  0
    Stay behind              INT     1     4     3m   0    RWD  -3
    Alarm Class              STRING  1     32    3m   0    RWD
    Running condition        STRING  1     128   3m   0    RWD  ALWAYS
    Data dir                 STRING  1     256   3m   0    RWD  /home/midas/online
    Data format              STRING  1     8     3m   0    RWD  MIDAS
    Filename format          STRING  1     128   3m   0    RWD  run%05d.mid
    Backup type              STRING  1     8     3m   0    RWD  Tape
    Execute after rewind     STRING  1     64    3m   0    RWD
    Path                     STRING  1     128   3m   0    RWD
    Capacity (Bytes)         FLOAT   1     4     3m   0    RWD  5e+09
    List label               STRING  1     128   3m   0    RWD
```

Maintain free space As the Data Logger (mlogger) runs independently from the Lazylogger, the disk will contains all the recorded data files. Under this condition, Lazylogger can be instructed to "purge" the data logging device (disk) after successful backup of the data onto the "slow device". The *Maintain free space(%)* parameter controls (if none zero) the percentage of disk space required to be maintain free.

The condition for removing a data file is defined as:

The data file corresponding to the given run number following the format declared
under "Settings/Filename format" IS PRESENT on the "Settings/Data Dir" path.
AND
The given run number appears anywhere under the "List/" directory of ALL the Lazy
channel having the same "Settings/Filename format"as this channel.
AND
The given run number appears anywhere under the "List/" directory of that channel

Stay behind This parameter defines how many consecutive data file should be kept between the current run and the last lazylogger run.
Example with "Stay behind = -3" :
- Current acquisition run number 253 -> run00253.mid is being logger by mlogger.

- Files available on the disk corresponding to run #248, #249, #250, #251, #252.
- Lazylogger will start backing up run #250 as soon new run 254 will start.
- ”Stay behind = -3” will correspond to 3 file untouch on the disk (#251, #252, #253).

The negative sign instructs lazylogger to **always scan the entire ”Data Dir” from the oldest to the most recent file sitting on the disk at the ”Data Dir” path** for backup. If the ”Stay behind” is positive, lazylogger will **backup starting from** x behind the current acquisition run number. Run older will be ignored.

Alarm Class Specify the Alarm class to be used in case of triggered alarm.

Running condition Specify the type of condition for which lazylogger should be actived. By default lazylogger is **ALWAYS** running. In the case of high data rate acquisition it could be necessary to active lazylogger only when the run is either pauses, stopped or when some external condition is satisied such as ”Low beam intensity”. In this later case, condition based on a single field of the ODB can be given to established when the application should be active.
Example:

```
odbedit> set "Running condition" WHILE_ACQ_NOT_RUNNING
odbedit> set "Running condition" "/alias/max_rate < 200"
```

Data dir Specify the Data directory path of the data files. By default if the ”/Logger/Data Dir” is present, the pointed value is taken otherwise the current directory where lazylogger has been started is used.

Data format Specify the Data format of the data files. Currently supported format are: **MIDAS** or **YBOS**.

Filename format Specify the file format of the data files. Same format as given for the data logger.

Backup type Specify the ”slow device” backup type. Default **Tape**. Can be **Disk** or **Ftp**.

Execute after rewind Specify a script to be run after completion of a lazylogger backup set (see below ”Capacity (Bytes)”).

Path Specify the ”slow device” path. Three possible type of Path:

- For Tape : **/dev/nst0** (UNIX like).
- For Disk : **/data1/myexpt**
- For Ftp : **host,port,user,password,directory**

Capacity (Bytes) Specify the maximum ”slow device” capacity in bytes. When this capacity is reached, lazylogger will close the backup device and clear the ”List Label” field to prevent further backup (see below). It will aslo rewind the stream device if possible.

List label Specify a label for a set of backed up files to the ”slow device”. This label is used only internaly by lazylogger for creating under the ”/List” a new array composed of the backed up runs until the ”Capacity” value has been reached. As the backup set is complete, lazylogger will clear this field and therefore prevent any further backup until a none empty label list is entered again. In the other hand the list label will remain under the ”/List” key to display all run being backed up until the corresponding files have been removed from the disk.

Statistics/ ODB tree specifying general information about the status of the current lazylogger channel state.

List/ ODB tree, will contain arrays of run number associated to the array name backup-set label. Any run number appearing in any of the array is considered to have been backed up.

- Usage

lazylogger requires to be setup prior data file can be moved. This setup consists in 4 steps:

Step 1 Invoking lazylogger once for setting up the appropriate ODB tree and exit.

```
>lazylogger -c Tape
```

Step 2 Edit the newly created ODB tree. Correct the setting field to match your requirement.

```
  > odbedit -e midas
[local:midas:Stopped]/>cd /Lazy/tape/
[local:midas:Stopped]tape>ls
[local:midas:Stopped]tape>ls -lr
Key name                        Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
tape                            DIR
    Settings                    DIR
        Maintain free space(%)  INT     1     4     3m   0   RWD  0
        Stay behind             INT     1     4     3m   0   RWD  -3
        Alarm Class             STRING  1     32    3m   0   RWD
        Running condition       STRING  1     128   3m   0   RWD  ALWAYS
        Data dir                STRING  1     256   3m   0   RWD  /home/midas/online
        Data format             STRING  1     8     3m   0   RWD  MIDAS
        Filename format         STRING  1     128   3m   0   RWD  run%05d.mid
        Backup type             STRING  1     8     3m   0   RWD  Tape
        Execute after rewind    STRING  1     64    3m   0   RWD
        Path                    STRING  1     128   3m   0   RWD
        Capacity (Bytes)        FLOAT   1     4     3m   0   RWD  5e+09
        List label              STRING  1     128   3m   0   RWD
    Statistics                  DIR
        Backup file             STRING  1     128   3m   0   RWD  none
        File size [Bytes]       FLOAT   1     4     3m   0   RWD  0
        KBytes copied           FLOAT   1     4     3m   0   RWD  0
        Total Bytes copied      FLOAT   1     4     3m   0   RWD  0
        Copy progress [%]       FLOAT   1     4     3m   0   RWD  0
        Copy Rate [bytes per s] FLOAT   1     4     3m   0   RWD  0
        Backup status [%]       FLOAT   1     4     3m   0   RWD  0
        Number of Files         INT     1     4     3m   0   RWD  0
        Current Lazy run        INT     1     4     3m   0   RWD  0
[local:midas:Stopped]tape>cd Settings/
[local:midas:Stopped]Settings>set "Data dir" /data
[local:midas:Stopped]Settings>set "Capacity (Bytes)" 15e9
```

Step 3 Start lazylogger in the background

```
>lazylogger -c Tape -D
```

Step 4 At this point the lazylogger is running and waiting for the "list label" to be defined before starting the copy procedure. mstat task will display information regarding the status of the lazylogger.

```
> odbedit -e midas
[local:midas:Stopped]/>cd /Lazy/tape/Settings
[local:midas:Stopped]Settings>set "List label" cni-043
```

- Remarks

    1. For every major operation of the lazylogger a message is sent to the Message buffer and will be appended to the default Midas log file (midas.log). These messages are the only mean of finding out What/When/Where/How the lazylogger has operate on a data file. See below a fragment of the **midas.log** for the chaos experiment. In this case the **Maintain free space(%)** field was enabled which produce the cleanup of the data files and the entry in the **List** tree after copy.

```
Fri Mar 24 14:40:08 2000 [Lazy_Tape] 8351 (rm:16050ms) /scr0/spring2000/run08351.ybs file REMOVED
Fri Mar 24 14:40:08 2000 [Lazy_Tape] Tape run#8351 entry REMOVED
Fri Mar 24 14:59:55 2000 [Logger] stopping run after having received 1200000 events
Fri Mar 24 14:59:56 2000 [CHAOS] Run 8366 stopped
Fri Mar 24 14:59:56 2000 [Logger] Run #8366 stopped
Fri Mar 24 14:59:57 2000 [SUSIYBOS] saving info in run log
Fri Mar 24 15:00:07 2000 [Logger] starting new run
Fri Mar 24 15:00:07 2000 [CHAOS] Run 8367 started
Fri Mar 24 15:00:07 2000 [Logger] Run #8367 started
Fri Mar 24 15:06:59 2000 [Lazy_Tape] cni-043[15] (cp:410.6s) /dev/nst0/run08365.ybs 864.020MB file NE
Fri Mar 24 15:07:35 2000 [Lazy_Tape] 8352 (rm:25854ms) /scr0/spring2000/run08352.ybs file REMOVED
Fri Mar 24 15:07:35 2000 [Lazy_Tape] Tape run#8352 entry REMOVED
Fri Mar 24 15:27:09 2000 [Lazy_Tape] 8353 (rm:23693ms) /scr0/spring2000/run08353.ybs file REMOVED
Fri Mar 24 15:27:09 2000 [Lazy_Tape] Tape run#8353 entry REMOVED
Fri Mar 24 15:33:22 2000 [Logger] stopping run after having received 1200000 events
Fri Mar 24 15:33:22 2000 [CHAOS] Run 8367 stopped
Fri Mar 24 15:33:23 2000 [Logger] Run #8367 stopped
Fri Mar 24 15:33:24 2000 [SUSIYBOS] saving info in run log
Fri Mar 24 15:33:33 2000 [Logger] starting new run
Fri Mar 24 15:33:34 2000 [CHAOS] Run 8368 started
Fri Mar 24 15:33:34 2000 [Logger] Run #8368 started
Fri Mar 24 15:40:18 2000 [Lazy_Tape] cni-043[16] (cp:395.4s) /dev/nst0/run08366.ybs 857.677MB file NE
Fri Mar 24 15:50:15 2000 [Lazy_Tape] 8354 (rm:28867ms) /scr0/spring2000/run08354.ybs file REMOVED
Fri Mar 24 15:50:15 2000 [Lazy_Tape] Tape run#8354 entry REMOVED
...
```

2. Once lazylogger has started a job on a data file, trying to terminate the application will result on producing a log message informing about the actual percentage of the backup being completed so far. This message will repeat it self until completion of the backup and only then the lazylogger application will terminate.
3. If an interruption of the lazylogger is forced (kill...) The state of the backup device is undertermined. Recovery is not possible and the full backup set has to be redone. In order to do this, you need:
   (a) To rewind the backup device.
   (b) Delete the /Lazy/<channel_name>/List/<list label> array.
   (c) Restart lazylogger with the -z switch which will "zap" the statistics entries.
4. In order to facilitate the recovery procedure, **lazylogger** produces an ODB ASCII file of the lazy channel tree after completion of successful operation. This file (**Tape_recover.odb**) stored in **Data Dir** can be used for ODB as well as lazylogger recovery.

6.6 mdump task

Event display utility.

This application allows to "peep" into the data flow in order to display a snap-shot of the event. Its use is particularly powerful during experiment setup. In addition **mdump** has the capability to operate on data save-set files stored on disk or tape. The main **mdump** restriction is the fact that it works only for events formatted in banks (i.e.: MIDAS, YBOS bank).

- Arguments for Online

-h : help for online use.

-h hostname : Host name.

-e exptname : Experiment name.

-b bank name : Display event containg only specified bank name.

-c compose : Retrieve and compose file with either Add run# or Not (def:N).

-f format : Data representation (x/d/ascii) def:hex.

-g type : Sampling mode either Some or All (def:S).
>>> in case of -c it is recommented to used -g all.

-i id : Event Id.

-j : Display bank header only.

-k id : Event mask.
>>> -i and -k are valid for YBOS ONLY if EVID bank is present in the event

-l number : Number of consecutive event to display (def:1).

-m mode : Display mode either Bank or Raw (def:B)

-p path : Path for file composition (see -c)

-s : Data transfer rate diagnositic.

-w time : Insert wait in [sec] between each display.

-x filename : Input channel. data file name of data device. (def:online)

-y : Display consistency check only.

-z buffer name : Midas buffer name to attach to (def:SYSTEM)

- Additional arguments for Offline

-x -h : help for offline use.

-t type : Bank format (Midas/Ybos).
>>> if -x is a /dev/xxx, -t has to be specified.

-r # : skip record(YBOS) or event(MIDAS) to #.

-w what : Header, Record, Length, Event, Jbank_list (def:E)
>>> Header & Record are not supported for MIDAS as it has no physical record structure.

- Usage
mdump can operate on either data stream (online) or on save-set data file. Specific help is available for each mode.

```
 > mdump -h
 > mdump -x -h

Tue> mdump -x run37496.mid | more
------------------------ Event# 0 --------------------------------
------------------------ Event# 1 --------------------------------
Evid:0001- Mask:0100- Serial:1- Time:0x393c299a- Dsize:72/0x48
#banks:2 - Bank list:-SCLRRATE-

Bank:SCLR Length: 24(I*1)/6(I*4)/6(Type) Type:Integer*4
   1-> 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000

Bank:RATE Length: 24(I*1)/6(I*4)/6(Type) Type:Real*4 (FMT machine dependent)
```

```
   1-> 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
------------------------ Event# 2 --------------------------------
Evid:0001- Mask:0004- Serial:1- Time:0x393c299a- Dsize:56/0x38
#banks:2 - Bank list:-MMESMMOD-

Bank:MMES Length: 24(I*1)/6(I*4)/6(Type) Type:Real*4 (FMT machine dependent)
   1-> 0x3de35788 0x3d0b0e29 0x00000000 0x00000000 0x3f800000 0x00000000

Bank:MMOD Length: 4(I*1)/1(I*4)/1(Type) Type:Integer*4
   1-> 0x00000001
------------------------ Event# 3 --------------------------------
Evid:0001- Mask:0008- Serial:1- Time:0x393c299a- Dsize:48/0x30
#banks:1 - Bank list:-BMES-

Bank:BMES Length: 28(I*1)/7(I*4)/7(Type) Type:Real*4 (FMT machine dependent)
   1-> 0x443d7333 0x444cf333 0x44454000 0x4448e000 0x43bca667 0x43ce0000 0x43f98000
------------------------ Event# 4 --------------------------------
Evid:0001- Mask:0010- Serial:1- Time:0x393c299a- Dsize:168/0xa8
#banks:1 - Bank list:-CMES-

Bank:CMES Length: 148(I*1)/37(I*4)/37(Type) Type:Real*4 (FMT machine dependent)
   1-> 0x3f2f9fe2 0x3ff77fd6 0x3f173fe6 0x3daeffe2 0x410f83e8 0x40ac07e3 0x3f6ebfd8 0x3c47ffde
   9-> 0x3e60ffda 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x3f800000
  17-> 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
  25-> 0x3f800000 0x3f800000 0x3f800000 0x00000000 0x3f800000 0x00000000 0x3f800000 0x3f800000
  33-> 0x3f800000 0x3f800000 0x3f800000 0x3f800000 0x00000000
------------------------ Event# 5 --------------------------------
Evid:0001- Mask:0020- Serial:1- Time:0x393c299a- Dsize:32/0x20
#banks:1 - Bank list:-METR-

Bank:METR Length: 12(I*1)/3(I*4)/3(Type) Type:Real*4 (FMT machine dependent)
   1-> 0x00000000 0x39005d87 0x00000000
...
```

- Examples

```
 > mdump -j
```

6.7

mevb task

Midas Event Builder.

Names

mevb is a event builder tool assembling multiple event fragment received from different event buffers.

In the case where overall data collection is handled by multiple physically separated frontend, it could be necessary to assemble these data fragments into a dedicated event. The synchonization of the fragment collection is left to the user which is done usually through specific hardware mechanism. Once the fragments are composed in each frontend, they are sent to the "Event Builder" (eb) where the serial number (pheader->serial_number) of each fragment is compared one event at a time for serial match. In case of match, a new event will be composed with its own event ID and serial number followed by all the expected fragments. The composed event is then sent to the next stage which is usually the data logger (mlogger).

The mhttpd task will present a specific section on its page for the "event builder" task if running.

- Arguments

-h	: help
-h hostname	: host name
-e exptname	: experiment name
-v	: Show wheel
-d	: debug messages
-D	: start program as a daemon

- Usage

```
Thu> mevb -e midas
Program mevb/EBuilder version 2 started
```

6.7.1 Function description

->ODB/EBuilder Tree

Each frontend channel needs to send its "midas event" (fragment in this case) to a dedicated midas buffer with a unique "Event Identifier". This is specified within the frontend code in the equipment definition (BUF1 instead of default SYSTEM, see also ODB /Equipment Tree):

```
EQUIPMENT equipment[] = {

  { "Trigger1",            // equipment name
    1, 0,                  // event ID, trigger mask
    "BUF1",                // event buffer
    ...
```

The user has the possibility of interfering into the event building process at several stages:

Begin of run Like in the frontend, a hook to the begin of run is available for initialization etc.

End of run Like in the frontend, a hook to the end of run is available for proper closure of private task etc.

Event-By_Event Once all the fragments for a given serial number (match), the user has the possibility to access these fragments for further "user fragment analysis" and/or appending private data to the built event through the mean of bank creation (see eb_user()).

In the case of serial number mismatch of "user fragment analysis" error, **THERE IS NO RECOVERY PROCESS AVAILABLE YET!**

6.7.2 ODB/EBuilder Tree

->EB Operation

The Event builder tree will be created with default settings from the mevb.h header file. The location of the tree is at the root level of the midas experiment. Each frontend fragment and the "built event" has its own subdirectory under the /EBuilder with a Settings tree defining the frontend characteristics and Statistics for status information.

The fields "Event ID", "Buffer" have to match the frontend equipment definition in order to garantee the proper data transfer.

```
[local:midas:S]/>ls -lr EBuilder
Key name                        Type    #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
EBuilder                        DIR
    Settings                    DIR
```

```
        Event ID                WORD    1     2     35h  0   RWD  1
        Trigger mask            WORD    1     2     35h  0   RWD  1
        Buffer                  STRING  1     32    35h  0   RWD  SYSTEM
        Format                  STRING  1     32    35h  0   RWD  MIDAS
        User build              BOOL    1     4     35h  0   RWD  n
        Event mask              DWORD   1     4     35h  0   RWD  3
        Hostname                STRING  1     64    43m  0   RWD  dasdevpc
    Statistics                  DIR
        Events sent             DOUBLE  1     8     38m  0   RWD  1883
        Events per sec.         DOUBLE  1     8     38m  0   RWD  0
        kBytes per sec.         DOUBLE  1     8     38m  0   RWD  0
    Channels                    DIR
        Frag1                   DIR
            Settings            DIR
                Event ID        WORD    1     2     35h  0   RWD  1
                Trigger mask    WORD    1     2     35h  0   RWD  65535
                Buffer          STRING  1     32    35h  0   RWD  BUF1
                Format          STRING  1     32    35h  0   RWD  MIDAS
                Event mask      DWORD   1     4     35h  0   RWD  1
            Statistics          DIR
                Events sent     DOUBLE  1     8     38m  0   RWD  1883
                Events per sec. DOUBLE  1     8     38m  0   RWD  1881.12
                kBytes per sec. DOUBLE  1     8     38m  0   RWD  0
        Frag2                   DIR
            Settings            DIR
                Event ID        WORD    1     2     35h  0   RWD  2
                Trigger mask    WORD    1     2     35h  0   RWD  65535
                Buffer          STRING  1     32    35h  0   RWD  BUF2
                Format          STRING  1     32    35h  0   RWD  MIDAS
                Event mask      DWORD   1     4     35h  0   RWD  2
            Statistics          DIR
                Events sent     DOUBLE  1     8     38m  0   RWD  1884
                Events per sec. DOUBLE  1     8     38m  0   RWD  1882.12
                kBytes per sec. DOUBLE  1     8     38m  0   RWD  0
```

6.7.3 EB Operation

->mevb Status/Bugs

Using the ”eb>” as the cwd for the example, the test procedure is the following:
cwd : midas/examples/eventbuilder -> refered as eb>

1. Build the mevb task:

```
eb> make
cc  -g -I/usr/local/include -I../../drivers -DOS_LINUX -Dextname -c ebuser.c
cc  -g -I/usr/local/include -I../../drivers -DOS_LINUX -Dextname -o mevb mevb.c \
          ebuser.o /usr/local/lib/libmidas.a  -lm -lz -lutil -lnsl
```

```
cc  -g -I/usr/local/include -I../../drivers -DOS_LINUX -Dextname \
          -c ../../drivers/bus/camacnul.c
cc  -g -I/usr/local/include -I../../drivers -DOS_LINUX -Dextname -o fe1 \
        fe1.c camacnul.o /usr/local/lib/mfe.o /usr/local/lib/libmidas.a \
-lm -lz -lutil -lnsl
cc  -g -I/usr/local/include -I../../drivers -DOS_LINUX -Dextname -o fe2 \
        fe2.c camacnul.o /usr/local/lib/mfe.o /usr/local/lib/libmidas.a \
-lm -lz -lutil -lnsl
eb>
```

2. Start the following 4 applications in 4 differents windows connecting to a defined experiment. – If no experiment defined yet, set the environment variable MIDAS_DIR to your current directory before spawning the windows.

   ```
   eb> pwd
   /home/midas/midas-1.8.3/examples/eventbuilder
   eb> setenv MIDAS_DIR /home/midas/midas-1.8.3/examples/eventbuilder
   eb> odbedit
   [local:Default:S]/>ls
   System
   Programs
   Experiment
   Logger
   Runinfo
   Alarms
   [local:Default:S]/>q
   eb>
   ```

   ```
   xterm1: eb> fe1
   xterm2: eb> fe2
   xterm3: eb> mevb
   xterm4: eb> odbedit
   ```

   ```
   [local:Default:S]/>ls
   System
   Programs
   Experiment
   Logger
   Runinfo
   Alarms
   Equipment
   EBuilder                        <--- New tree
   [local:Default:S]/>scl
   Name                Host
   Fe1                 dasdevpc    <--- fragment 1
   Fe2                 dasdevpc    <--- fragment 2
   EBuilder            dasdevpc    <--- Event builder
   ODBEdit             dasdevpc
   [local:Default:S]/>
   [local:Default:S]/>start now
   Starting run #2

   12:12:11 [ODBEdit] Run #2 started
   ```

```
[local:Default:R]/>stop

12:12:13 [ODBEdit] Run #2 stopped
12:12:16 [EBuilder] Run 2 Stop on frag#0; events_sent 144; npulser 0
12:12:16 [EBuilder] Run 2 Stop on frag#1; events_sent 144; npulser 0
[local:Default:S]/>
```

3. The xterm3 (mevb) should display something equivalent to the following, as the print statements are coming from the ebuser code.

```
 New Run 2
In eb_begin_of_run
 nfrag : 2
bm_empty_buffer:1
bm_empty_buffer:1
Event Serial1 Fragment#:1 Data size:56 Serial1 Fragment#:2 Data size:56 Serial1
Event Serial2 Fragment#:1 Data size:56 Serial2 Fragment#:2 Data size:56 Serial2
Event Serial3 Fragment#:1 Data size:56 Serial3 Fragment#:2 Data size:56 Serial3
Event Serial4 Fragment#:1 Data size:56 Serial4 Fragment#:2 Data size:56 Serial4
Event Serial5 Fragment#:1 Data size:56 Serial5 Fragment#:2 Data size:56 Serial5
...
Event Serial141 Fragment#:1 Data size:56 Serial141 Fragment#:2 Data size:56 Serial141
Event Serial142 Fragment#:1 Data size:56 Serial142 Fragment#:2 Data size:56 Serial142
Event Serial143 Fragment#:1 Data size:56 Serial143 Fragment#:2 Data size:56 Serial143
Event Serial144 Fragment#:1 Data size:56 Serial144 Fragment#:2 Data size:56 Serial144
In eb_end_of_run
Run 2 Stop on frag#0; events_sent 144; npulser 0
Time between request and actual stop: 3457 ms
In eb_end_of_run
Run 2 Stop on frag#1; events_sent 144; npulser 0
Time between request and actual stop: 3459 ms
```

4. The same procedure can be repeated with the fe1 and fe2 started on remote nodes.

```
eb> odb -e midas
[local:midas:S]/>scl
Name                Host
Fe1                 mid001.triumf.ca       <-- Node 1
Fe2                 mid002.triumf.ca       <-- Node 2
EBuilder            dasdevpc               <-- Node 3
ODBEdit             dasdevpc               <-- Node 3
[local:midas:S]/>

Thu> mevb -e midas
Program mevb/EBuilder version 2 started

 New Run 209
In eb_begin_of_run
 nfrag : 2
bm_empty_buffer:1
bm_empty_buffer:1
Event Serial1 Fragment#:1 Data size:56 Serial1 Fragment#:2 Data size:56 Serial1
Event Serial2 Fragment#:1 Data size:56 Serial2 Fragment#:2 Data size:56 Serial2
Event Serial3 Fragment#:1 Data size:56 Serial3 Fragment#:2 Data size:56 Serial3
Event Serial4 Fragment#:1 Data size:56 Serial4 Fragment#:2 Data size:56 Serial4
Event Serial5 Fragment#:1 Data size:56 Serial5 Fragment#:2 Data size:56 Serial5
...
Event Serial233 Fragment#:1 Data size:56 Serial233 Fragment#:2 Data size:56 Serial233
Event Serial234 Fragment#:1 Data size:56 Serial234 Fragment#:2 Data size:56 Serial234
Event Serial235 Fragment#:1 Data size:56 Serial235 Fragment#:2 Data size:56 Serial235
In eb_end_of_run
Run 209 Stop on frag#0; events_sent 235; npulser 0
Time between request and actual stop: 4488 ms
```

6.7.4

mevb Status/Bugs

->mevb task

Jan 17/2002 – Initial Version composed of:
Makefile : Build fe1, fe2, mevb.
fe1.c : frontend code for event fragment 1.
fe2.c : frontend code for event fragment 2.
mevb.h : Event builder header file.
mevb.c : Event builder core code.
ebuser.c : User code for event building.

- Under Linux if the FEx are remote and have already collected data without the mevb running, on the next run start, the mevb will exit with event mismatch. Make sure the mevb is started before the FEx or the FEx are "fresh" when mevb is launched. It appears when mevb is running under Windows this problem is not occuring.
- mdump needs to be updated for event information extraction from the /EBuilder tree. This will permit a proper event display from the SYSTEM buffer.

6.7.5

INT **eb_begin_of_run** (INT rn, char* error)

eb_begin_of_run() Hook to the event builder task at PreStart transition.

eb_begin_of_run() Hook to the event builder task at PreStart transition.

Return Value:	EB_SUCCESS	
Parameters:	rn	Run Number
	error	error string to be passed back to the system.

6.7.6

INT **eb_end_of_run** (INT rn, char* error)

eb_end_of_run() Hook to the event builder task at completion of event collection after receiving the Stop transition.

eb_end_of_run() Hook to the event builder task at completion of event collection after receiving the Stop transition.

Return Value:	EB_SUCCESS	
Parameters:	rn	Run Number
	error	error string to be passed back to the system.

6.7.7

INT **eb_user()** (INT nfrag, EBUILDER_CHANNEL* ebch, EVENT_HEADER* pheader, void* pevent, INT* dest_size)

event builder user code for private data filtering.

Hook to the event builder task after the reception of all fragments of the same serial number. The destination event has already the final EVENT_HEADER setup with the data size set to 0. It is than possible to add private data at this point using the proper bank calls.

The ebch[] array structure points to nfragment channel structure with the following content:

```
typedef struct {
    char  name[32];         // Fragment name (Buffer name).
    DWORD serial;           // Serial fragment number.
    char *pfragment;        // Pointer to fragment (EVENT_HEADER *)
    ...
} EBUILDER_CHANNEL;
```

The correct code for including your own bank is shown below where **TID_xxx** is one of the valid Bank type starting with **TID_** for midas format or **xxx_BKTYPE** for Ybos data format. **bank_name** is a 4 character descriptor. **pdata** has to be declared accordingly with the bank type. Refers to the ebuser.c source code for further description.

It is not possible to mix within the same destination event different event format!

```
// Event is empty, fill it with BANK_HEADER
// If you need to add your own bank at this stage

bk_init(pevent);
bk_create(pevent, bank_name, TID_xxxx, &pdata);
*pdata++ = ...;
*dest_size = bk_close(pevent, pdata);
pheader->data_size = *dest_size + sizeof(EVENT_HEADER);
```

Return Value:

Parameters:	`nfrag`	Number of fragment.
	`ebch`	Structure to all the fragments.
	`pheader`	Destination pointer to the header.
	`pevent`	Destination pointer to the bank header.
	`dest_size`	Destination event size in bytes.

6.8 mspeaker, mlxspeaker tasks

Midas message speech synthesizer.

mspeaker, mlxspeaker are utilities which listen to the Midas messages system and pipe these messages to a speech synthesizer application. **mspeaker** is for the Windows based system and interface to the FirstByte/ProVoice package. The **mlxspeaker** is for Linux based system and interface to the Festival. In case of use of either package, the speech synthesis system has to be install prior the activation of the **mspeaker, mlxspeaker**.

- Arguments

-h	: help
-h hostname	: host name
-e exptname	: experiment name
-D	: start program as a daemon

- Usage

```
> mlxspeaker -D
```

6.9 mcnaf task

CAMAC utility.

mcnaf is an interactive CAMAC tool which allow "direct" access to the CAMAC hardware. This application is operational under either of the two following conditions:

1. **mcnaf** has been built against a particular CAMAC driver (see CAMAC drivers).
2. A user frontend code using a valid CAMAC driver is currently active.

- Arguments

-h	: help
-h hostname	: host name
-e exptname	: experiment name
-f frontend name	: Frontend name to connect to.
-s RPC server name	: CAMAC RPC server name for remote connection.

- Building application
  The *midas/utils/makefile.mcnaf* will build a collection of **mcnaf** applications which are hardware dependent, see example below:

miocnaf *cnaf* application using the declared CAMAC hardware DRIVER (kcs2927 in this case). To be used with **dio** CAMAC application starter (see dio task).

mwecnaf *cnaf* application using the WI-E-N-ER PCI/CAMAC interface (see CAMAC drivers). Please contact midas@triumf.ca for further information.

mcnaf *cnaf* application using the CAMAC RPC capability of any Midas frontend program having CAMAC access.

mdrvcnaf *cnaf* application using the Linux CAMAC driver for either kcs2927, kcs2926, dsp004. This application would require to have the proper Linux module loaded in the system first. Please contact midas@triumf.ca for further information.

```
Thu> cd /midas/utils
Thu> make -f makefile.mcnaf DRIVER=kcs2927
gcc -O3 -I../include -DOS_LINUX -c -o mcnaf.o mcnaf.c
gcc -O3 -I../include -DOS_LINUX -c -o kcs2927.o ../drivers/bus/kcs2927.c
gcc -O3 -I../include -DOS_LINUX -o miocnaf mcnaf.o kcs2927.o  ../linux/lib/libmidas.a -lutil
gcc -O3 -I../include -DOS_LINUX -c -o wecc32.o ../drivers/bus/wecc32.c
gcc -O3 -I../include -DOS_LINUX -o mwecnaf mcnaf.o wecc32.o  ../linux/lib/libmidas.a -lutil
gcc -O3 -I../include -DOS_LINUX -c -o camacrpc.o ../drivers/bus/camacrpc.c
gcc -O3 -I../include -DOS_LINUX -o mcnaf mcnaf.o camacrpc.o  ../linux/lib/libmidas.a -lutil
gcc -O3 -I../include -DOS_LINUX -c -o camaclx.o ../drivers/bus/camaclx.c
gcc -O3 -I../include -DOS_LINUX -o mdrvcnaf mcnaf.o camaclx.o  ../linux/lib/libmidas.a -lutil
rm *.o
```

- Running application

- Direct CAMAC access: This requires the computer to have the proper CAMAC interface installed and the **BASE ADDRESS** matching the value defined in the corresponding CAMAC driver. For kcs2926.c, kcs2927.c, dsp004.c, hyt1331.c, the base address (CAMAC_BASE) is set to 0x280.

```
>dio miocnaf
```

- RPC CAMAC through frontend: This requires to have a frontend running which will be able to serve the CAMAC RPC request. Any Midas frontend has that capability built-in but it has to have the proper CAMAC driver included in it.

```
>mcnaf -e <expt> -h <host> -f <fe_name>
```

- Usage

  ....

6.10 mhttpd task

Midas Web server.

Names

This daemon application has to run in order to allow the user to access from a Web browser any Midas experiment running on a given host. Full monitoring and "Almost" full control of a particular experiment can be achieved through this Midas Web server. The color coding is green for present/enabled, red for missing/disabled, yellow for inactive. It is important to note the refresh of the page is not "event driven" but is controlled by a timer (see **Config** button). This mean the information at any given time may reflect the experiment state of up to n second in the paste, where n is the timer setting of the refresh parameter. Its basic functionality are:

- Run control (start/stop).

- Frontend up-to-date status and statistics display.
- Logger up-to-date status and statistics display.
- Lazylogger up-to-date status and statistics display.
- Current connected client listing.
- Slow control data display.
- Basic access to ODB.
- Graphical history data display.
- Electronic LogBook recording/retrival messages
- Alarm monitoring/control
- ... and more ...

mhttpd requires as argument the TCP/IP port number in order to listen to the web based request.

- Arguments

  -h : help

  -p port : port number, no default, should be 8081 for example.

  -D : start program as a daemon

- Usage

```
>mhttpd -p 8081 -D
```

- Description Once the connection to a given experiment is established, the main Midas status page is displayed with the current ODB information related to this experiment. The page is sub-divised in several sections:

Experiment/Date Current Experiment, current date.

Action/Pages buttons Run control button, Page switch button. At any web page level within the Midas Web page the main status page can be invoked with the <status> button.

Start... button Depending on the run state, a single or the two first buttons will be showing the possible action (Start/Pause/Resume/Stop) (see Start page).

ODB button Online DataBase access. Depending on the security, R/W access can be granted to operated on any ODB field (see ODB page).

CNAF button If one of the equipment is a CAMAC frontend, it is possible to issue CAMAC command through this button. In this case the frontend is acting as a RPC CAMAC server for the request (see CNAF page).

Messages button Shows the n last entries of the Midas system message log. The last entry is always present in the status page (see below) (see Messages page).

Elog button Electronic Log book. Permit to record permanently (file) comments/messages composed by the user (see Elog page).

Alarms button Display current Alarm setting for the entire experiment. The activation of an alarm has to be done through ODB under the **/Alarms** tree (See Alarm System)

Program button Display current program (midas application) status. Each program has a specific information record associated to it. This record is tagged as a hyperlink in the listing (see Program page).

History button Display History graphs of pre-defined variables. The history setting has to be done through ODB under the **/History** (see History System, History page).

Config button Allows to change the page refresh rate.

Help button Help and link to the main Midas web pages.

User button(s) If the user define a new tree in ODB named **Script** than any sub-tree name will appear as a button of that name. Each sub-tree (/Script/<button name>/) should contain at least one string key being the script command to be executed. Further keys will be passed as arguments to the script. Midas Symbolic link are permitted.

Example: The example below defines a script names **doit** with 2 arguments (run# device) which will be invoked when the button <doit> is pressed.

```
odbedit
mkdir Script
cd Script
mkdir doit
cd doit
create string cmd
ln "/runinfo/run number" run
create string dest
set dest /dev/hda
```

Version<1.8.3 Alias Hyperlink This line will be present on the status page only if the ODB tree **/Alias** or **/Alias new window** exists. This hyperlink will invoke the page within the current frame if placed in "Alias" or in a separate frame if placed in "Alias new window".

rsion >= 1.8.3 Alias Hyperlink This line will be present on the status page only if the ODB tree **/Alias**. The distinction for spawning a secondary frame with the link request is done by default. For forcing the link in the current frame, add the terminal charater "&" at the end of the link name.

Example: The example will create a shortcut to the defined location in the ODB.

```
 odbedit
 ls
 create key Alias
 cd Alias
 ln /Equipment/Trigger/Common "Trig Setting"
 ln /Analyzer/Output "Analyzer"

 create key "Alias new window"                    <-- Version < 1.8.3
 cd "Alias new window"
 ln /equipment/Scalers/Variables "Scalers Var"

or
 cd Alias
 ln /Equipment/Trigger/Common "Trig Setting&"      <-- Version >= 1.8.3
```

General info Current run number, state, General Enable flag for Alarm, Auto restart flag Condition of mlogger.

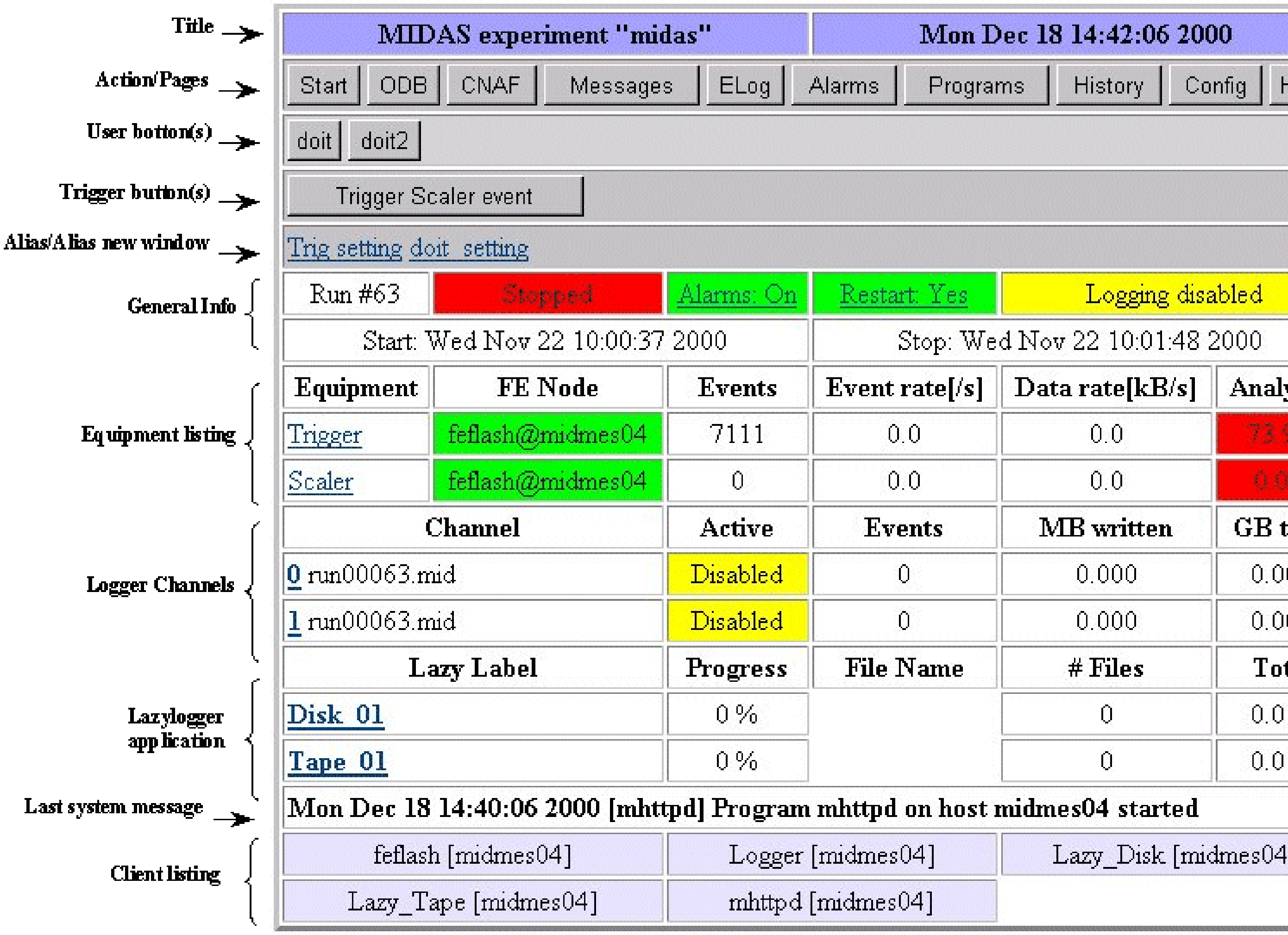
Equipment listing Equipment name (see Equipment page), host on which its running, Statistics for that current run, analyzed percentage by the "analyzer" (The numbers are valid only if the name of the analyser is "Analyzer").

Logger listing Logger list. Multiple logger channel can be active (single application). The hyperlink "0" will bring you to the odb tree /Logger/channels/0/Settings. This section is present only when the Midas application **mlogger task** is running.

Lazylogger listing Lazylogger list. Multiple lazy application can be active. This section is present only when the Midas application **lazylogger task** is running.

Last system message Display a single line containing the last system message received at the time of the last display refresh.

Current client listing List of the current active Midas application with the hostname on which their running.



MIDAS experiment "midas"		Mon Dec 18 14:42:06 2000		
Run #63	Stopped	Alarms: On	Restart: Yes	Logging disabled
Start: Wed Nov 22 10:00:37 2000		Stop: Wed Nov 22 10:01:48 2000		

Equipment	FE Node	Events	Event rate[/s]	Data rate[kB/s]	Analy
Trigger	feflash@midmes04	7111	0.0	0.0	73
Scaler	feflash@midmes04	0	0.0	0.0	0.0

Channel	Active	Events	MB written	GB t
0 run00063.mid	Disabled	0	0.000	0.00
1 run00063.mid	Disabled	0	0.000	0.00

Lazy Label	Progress	File Name	# Files	Tot
Disk_01	0 %		0	0.0
Tape_01	0 %		0	0.0

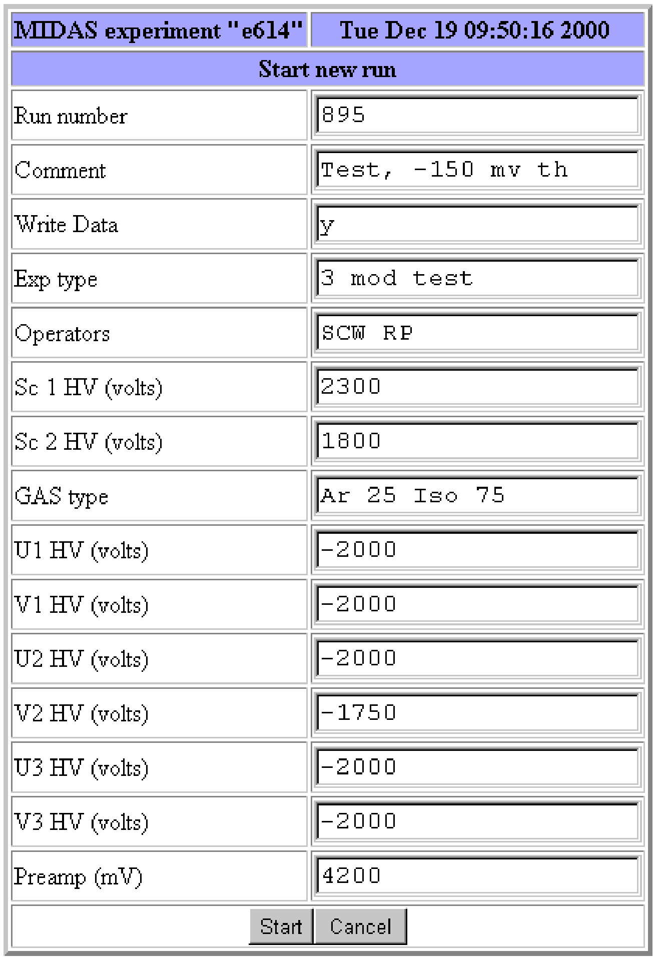
Mon Dec 18 14:40:06 2000 [mhttpd] Program mhttpd on host midmes04 started

feflash [midmes04]	Logger [midmes04]	Lazy_Disk [midmes04
Lazy_Tape [midmes04]	mhttpd [midmes04]	

Figure 10: *Midas Web server.*

6.10.1 Start page

Once the **Start** button is pressed, you will be prompt for experiment specific parameters before starting the run. The minimum set of parameter is the run number, it will be incremented by one relative to the last value from the status page. In the case you have defined the ODB tree **/Experiment/Edit on Start** all the parameters sitting in this directory will be displayed for possible modification. The **Ok** button will proceed to the start of the run. The **Cancel** will abort the start procedure and return you to the status page.



MIDAS experiment "e614"	Tue Dec 19 09:50:16 2000
Start new run	
Run number	895
Comment	Test, -150 mv th
Write Data	y
Exp type	3 mod test
Operators	SCW RP
Sc 1 HV (volts)	2300
Sc 2 HV (volts)	1800
GAS type	Ar 25 Iso 75
U1 HV (volts)	-2000
V1 HV (volts)	-2000
U2 HV (volts)	-2000
V2 HV (volts)	-1750
U3 HV (volts)	-2000
V3 HV (volts)	-2000
Preamp (mV)	4200

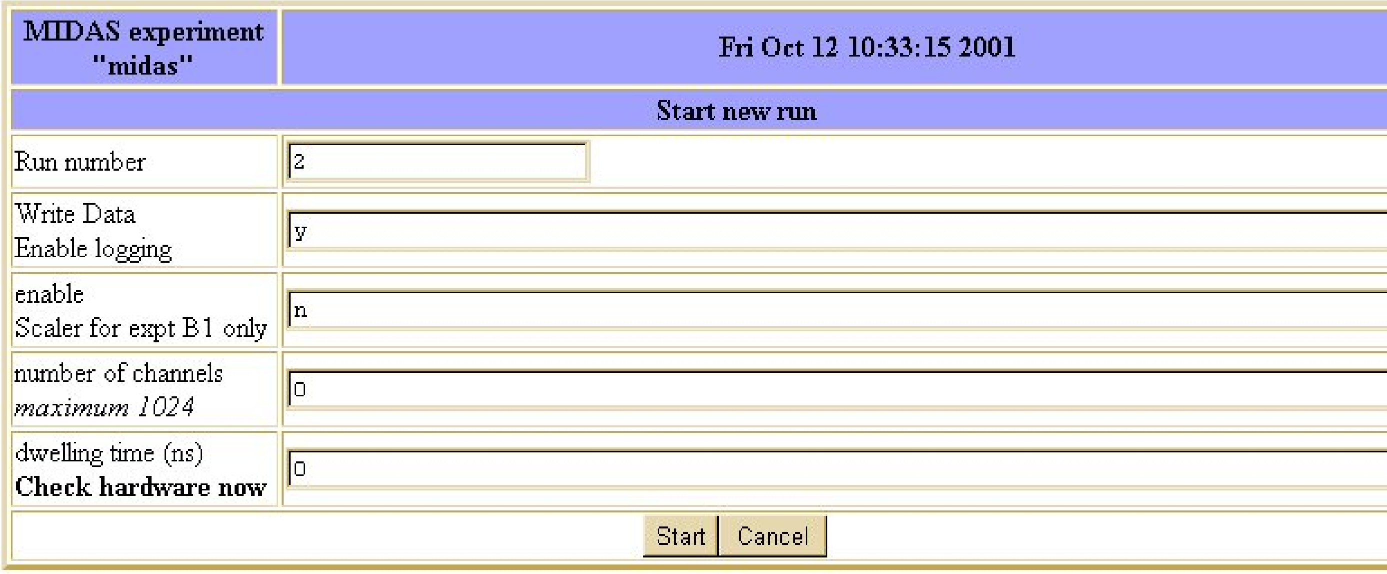
Figure 11: *Start run request page. In this case the user has multiple run parameters defined under "/Experiment/Edit on Start".*

The title of each field is taken from the ODB key name it self. In the case this label has a poor meaning and extra explanation is required, you can do so by creating a new ODB tree under experiment **Parameter Comments/**. Then by creating a string entry named as the one in **Edit on Start** you can place the extra information relative to that key (html tags accepted).

This "parameter comment" option is available and visible **ONLY** under the midas web page, the **odbedit start** command will not display this extra information.

```
[local:midas:S]/Experiment>ls -lr
Key name                        Type   #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
```

```
Experiment                      DIR
    Name                        STRING  1     32    17s  0   RWD  midas
    Edit on Start               DIR
        Write data              BOOL    1     4     16m  0   RWD  y
        enable                  BOOL    1     4     16m  0   RWD  n
        nchannels               INT     1     4     16m  0   RWD  0
        dwelling time (ns)      INT     1     4     16m  0   RWD  0
    Parameter Comments          DIR
        Write Data              STRING  1     64    44m  0   RWD  Enable logging
        enable                  STRING  1     64    7m   0   RWD  Scaler for expt B1 only
        nchannels               STRING  1     64    14m  0   RWD  <i>maximum 1024</i>
        dwelling time (ns)      STRING  1     64    8m   0   RWD  <b>Check hardware now</b>

[local:midas:S]Edit on Start>ls -l
Key name                        Type   #Val  Size  Last Opn Mode Value
---------------------------------------------------------------------------
Write Data                      LINK    1     19    50m  0   RWD  /logger/Write data
enable                          LINK    1     12    22m  0   RWD  /sis/enable
number of channels              LINK    1     15    22m  0   RWD  /sis/nchannels
dwelling time (ns)              LINK    1     24    12m  0   RWD  /sis/dwelling time (ns)
```



MIDAS experiment "midas"	Fri Oct 12 10:33:15 2001
Start new run	
Run number	2
Write Data Enable logging	y
enable Scaler for expt B1 only	n
number of channels *maximum 1024*	0
dwelling time (ns) **Check hardware now**	0

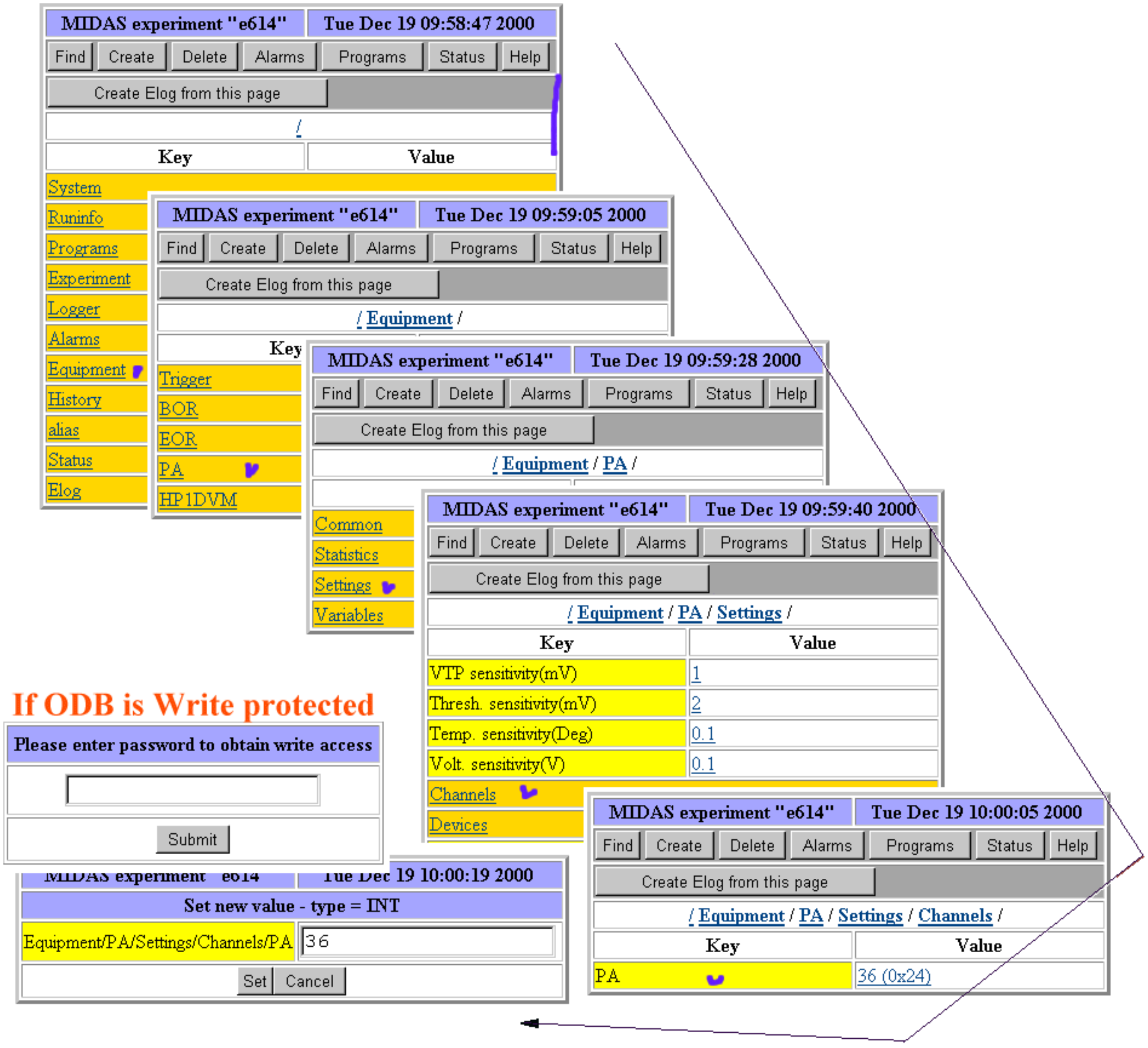
Figure 12: *Start run request page. Extra comment on the run condition is displayed below each entry.*

6.10.2

ODB page

The ODB page shows the ODB root tree at first. Clicking on the hyperlink will walk you to the requested ODB field. The example below show the sequence for changing the variable ”PA” under the /equipment/PA/Settings/Channels ODB directory. A possible shortcut

If the ODB is Write protected, a first window will request the web password.



Key	Value
System	
Runinfo	
Programs	
Experiment	
Logger	
Alarms	
Equipment	
History	
alias	
Status	
Elog	

Key
Trigger
BOR
EOR
PA
HP1DVM

Common
Statistics
Settings
Variables

Key	Value
VTP sensitivity(mV)	1
Thresh. sensitivity(mV)	2
Temp. sensitivity(Deg)	0.1
Volt. sensitivity(V)	0.1
Channels	
Devices	

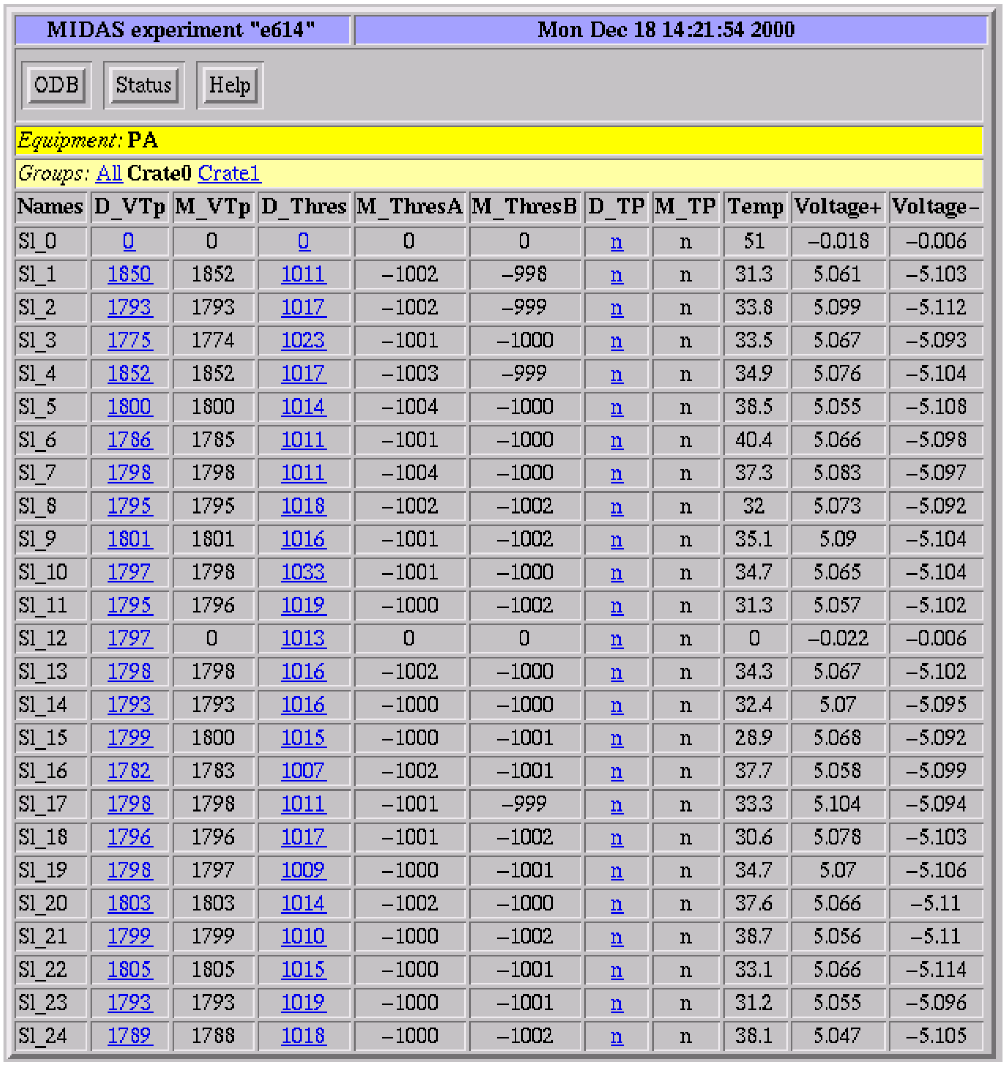
Key	Value
PA	36 (0x24)

Equipment/PA/Settings/Channels/PA 36

Figure 13: *ODB page access.*

6.10.3 Equipment page

The equipment names are linked to their respective **/Variables** sub-tree. This permit to access as a shortcut the current values of the equipment. In the case the equipment is a slow control equipment, the parameters list may be hyperlinked for parameter modification. This option is possible only if the parameter names have a particular name syntax (see History System).



MIDAS experiment "e614" Mon Dec 18 14:21:54 2000

Equipment: **PA**

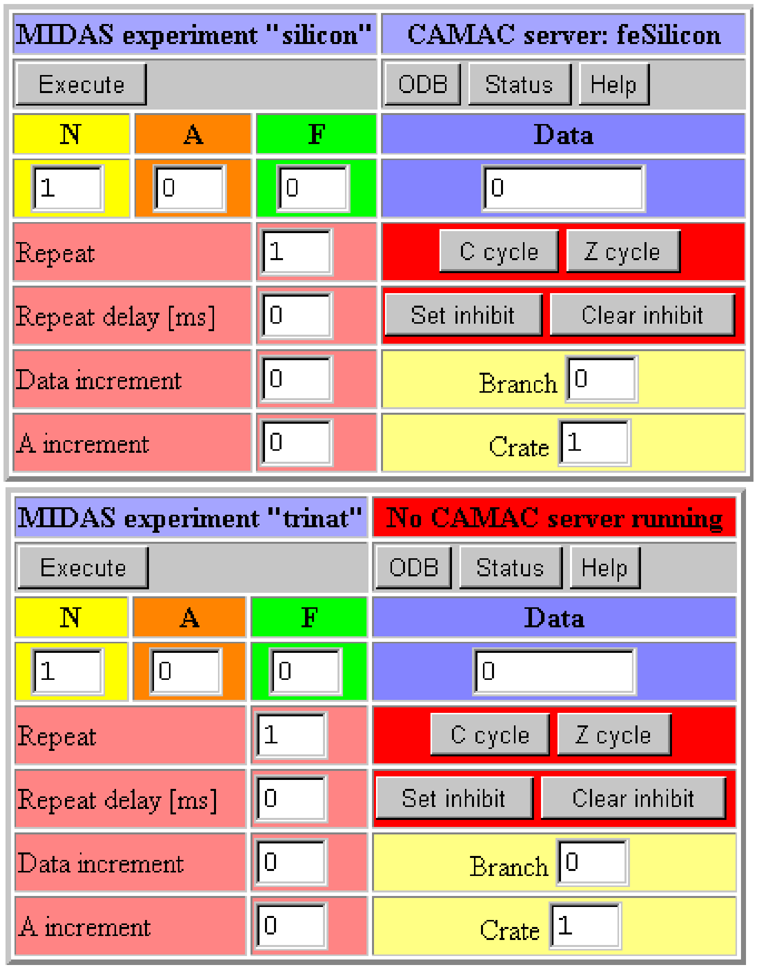
Groups: All **Crate0** Crate1

Names	D_VTp	M_VTp	D_Thres	M_ThresA	M_ThresB	D_TP	M_TP	Temp	Voltage+	Voltage–
Sl_0	0	0	0	0	0	n	n	51	–0.018	–0.006
Sl_1	1850	1852	1011	–1002	–998	n	n	31.3	5.061	–5.103
Sl_2	1793	1793	1017	–1002	–999	n	n	33.8	5.099	–5.112
Sl_3	1775	1774	1023	–1001	–1000	n	n	33.5	5.067	–5.093
Sl_4	1852	1852	1017	–1003	–999	n	n	34.9	5.076	–5.104
Sl_5	1800	1800	1014	–1004	–1000	n	n	38.5	5.055	–5.108
Sl_6	1786	1785	1011	–1001	–1000	n	n	40.4	5.066	–5.098
Sl_7	1798	1798	1011	–1004	–1000	n	n	37.3	5.083	–5.097
Sl_8	1795	1795	1018	–1002	–1002	n	n	32	5.073	–5.092
Sl_9	1801	1801	1016	–1001	–1002	n	n	35.1	5.09	–5.104
Sl_10	1797	1798	1033	–1001	–1000	n	n	34.7	5.065	–5.104
Sl_11	1795	1796	1019	–1000	–1002	n	n	31.3	5.057	–5.102
Sl_12	1797	0	1013	0	0	n	n	0	–0.022	–0.006
Sl_13	1798	1798	1016	–1002	–1000	n	n	34.3	5.067	–5.102
Sl_14	1793	1793	1016	–1000	–1000	n	n	32.4	5.07	–5.095
Sl_15	1799	1800	1015	–1000	–1001	n	n	28.9	5.068	–5.092
Sl_16	1782	1783	1007	–1002	–1001	n	n	37.7	5.058	–5.099
Sl_17	1798	1798	1011	–1001	–999	n	n	33.3	5.104	–5.094
Sl_18	1796	1796	1017	–1001	–1002	n	n	30.6	5.078	–5.103
Sl_19	1798	1797	1009	–1000	–1001	n	n	34.7	5.07	–5.106
Sl_20	1803	1803	1014	–1002	–1000	n	n	37.6	5.066	–5.11
Sl_21	1799	1799	1010	–1000	–1002	n	n	38.7	5.056	–5.11
Sl_22	1805	1805	1015	–1000	–1001	n	n	33.1	5.066	–5.114
Sl_23	1793	1793	1019	–1000	–1001	n	n	31.2	5.055	–5.096
Sl_24	1789	1788	1018	–1000	–1002	n	n	38.1	5.047	–5.105

Figure 14: *Slow control page.*

6.10.4 CNAF page

If one of the active equipment is a CAMAC based data collector, it will be possible to remotely access CAMAC through this web based CAMAC page. The status of the connection is displayed in the top right hand side corner of the window.



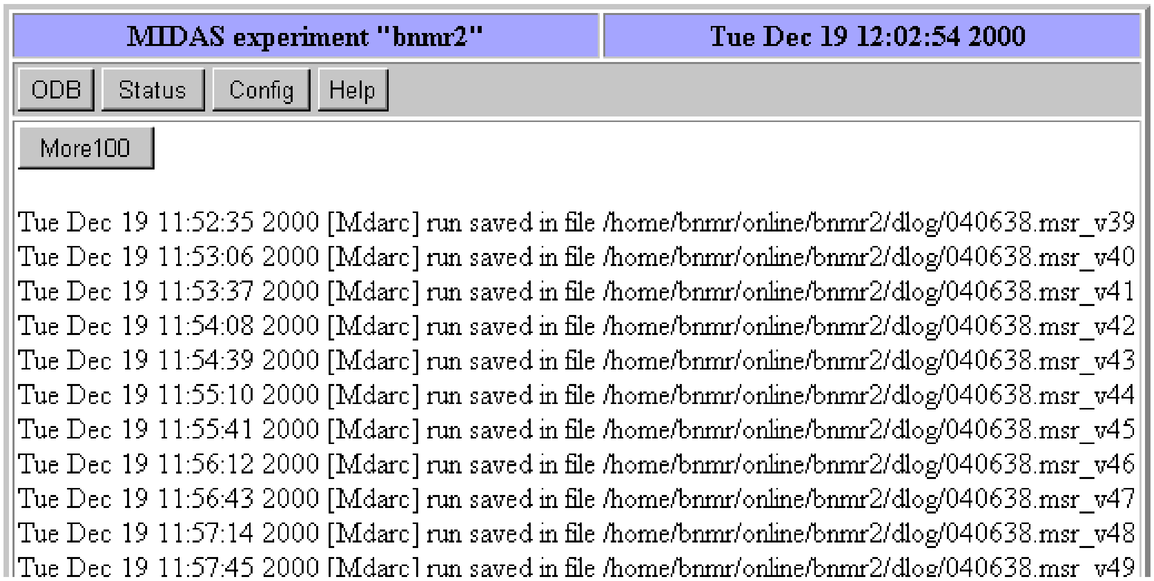
MIDAS experiment "silicon"			CAMAC server: feSilicon
Execute			ODB Status Help
N	A	F	Data
1	0	0	0
Repeat		1	C cycle Z cycle
Repeat delay [ms]		0	Set inhibit Clear inhibit
Data increment		0	Branch 0
A increment		0	Crate 1

MIDAS experiment "trinat"			No CAMAC server running
Execute			ODB Status Help
N	A	F	Data
1	0	0	0
Repeat		1	C cycle Z cycle
Repeat delay [ms]		0	Set inhibit Clear inhibit
Data increment		0	Branch 0
A increment		0	Crate 1

Figure 15: *CAMAC command pages.*

6.10.5 Message page

This page display by block of 100 lines the content of the Midas System log file starting with the most recent messages. The Midas log file resides in the directory defined by the experiment.

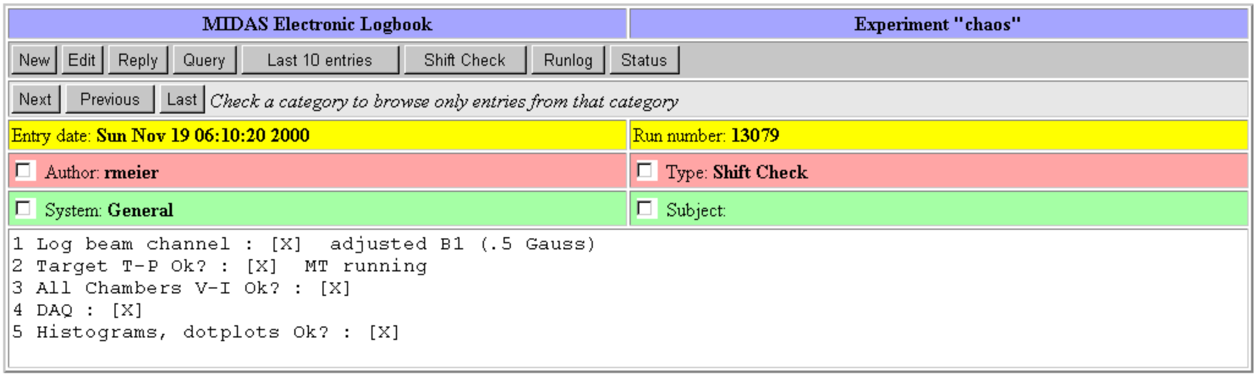


```
Tue Dec 19 11:52:35 2000 [Mdarc] run saved in file /home/bnmr/online/bnmr2/dlog/040638.msr_v39
Tue Dec 19 11:53:06 2000 [Mdarc] run saved in file /home/bnmr/online/bnmr2/dlog/040638.msr_v40
Tue Dec 19 11:53:37 2000 [Mdarc] run saved in file /home/bnmr/online/bnmr2/dlog/040638.msr_v41
Tue Dec 19 11:54:08 2000 [Mdarc] run saved in file /home/bnmr/online/bnmr2/dlog/040638.msr_v42
Tue Dec 19 11:54:39 2000 [Mdarc] run saved in file /home/bnmr/online/bnmr2/dlog/040638.msr_v43
Tue Dec 19 11:55:10 2000 [Mdarc] run saved in file /home/bnmr/online/bnmr2/dlog/040638.msr_v44
Tue Dec 19 11:55:41 2000 [Mdarc] run saved in file /home/bnmr/online/bnmr2/dlog/040638.msr_v45
Tue Dec 19 11:56:12 2000 [Mdarc] run saved in file /home/bnmr/online/bnmr2/dlog/040638.msr_v46
Tue Dec 19 11:56:43 2000 [Mdarc] run saved in file /home/bnmr/online/bnmr2/dlog/040638.msr_v47
Tue Dec 19 11:57:14 2000 [Mdarc] run saved in file /home/bnmr/online/bnmr2/dlog/040638.msr_v48
Tue Dec 19 11:57:45 2000 [Mdarc] run saved in file /home/bnmr/online/bnmr2/dlog/040638.msr_v49
```

Figure 16: *Message page.*

6.10.6 Elog page

The Electronic Log page show the most recent Log message recorded in the system. The top buttons allows you to either Create/Edit/Reply/Query/Show a message.



Entry date: **Sun Nov 19 06:10:20 2000** | Run number: **13079**

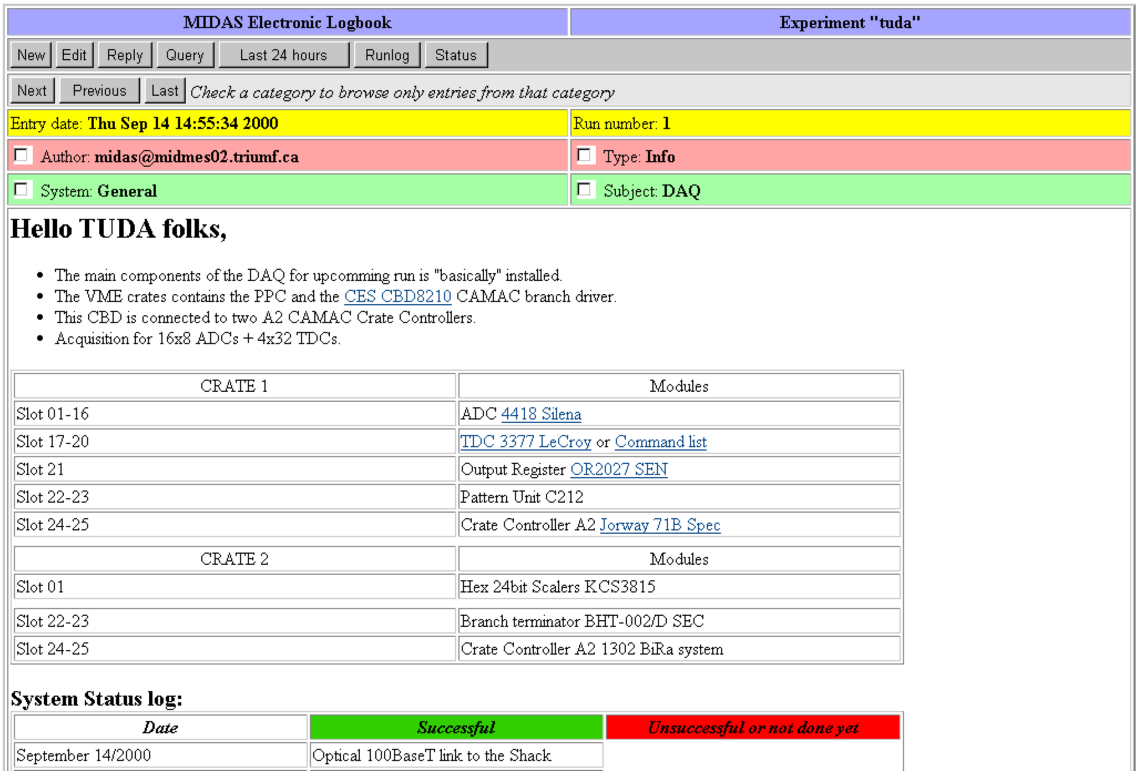
□ Author: **rmeier** | □ Type: **Shift Check**

□ System: **General** | □ Subject:

```
1 Log beam channel : [X]  adjusted B1 (.5 Gauss)
2 Target T-P Ok? : [X]  MT running
3 All Chambers V-I Ok? : [X]
4 DAQ : [X]
5 Histograms, dotplots Ok? : [X]
```

Figure 17: *main Elog page.*

The format of the message log can be written in HTML format.



MIDAS Electronic Logbook	Experiment "tuda"
Check a category to browse only entries from that category	
Entry date: **Thu Sep 14 14:55:34 2000**	Run number: **1**
□ Author: **midas@midmes02.triumf.ca**	□ Type: **Info**
□ System: **General**	□ Subject: **DAQ**

Hello TUDA folks,

- The main components of the DAQ for upcomming run is "basically" installed.
- The VME crates contains the PPC and the CES CBD8210 CAMAC branch driver.
- This CBD is connected to two A2 CAMAC Crate Controllers.
- Acquisition for 16x8 ADCs + 4x32 TDCs.

CRATE 1	Modules
Slot 01-16	ADC 4418 Silena
Slot 17-20	TDC 3377 LeCroy or Command list
Slot 21	Output Register OR2027 SEN
Slot 22-23	Pattern Unit C212
Slot 24-25	Crate Controller A2 Jorway 71B Spec
CRATE 2	Modules
Slot 01	Hex 24bit Scalers KCS3815
Slot 22-23	Branch terminator BHT-002/D SEC
Slot 24-25	Crate Controller A2 1302 BiRa system

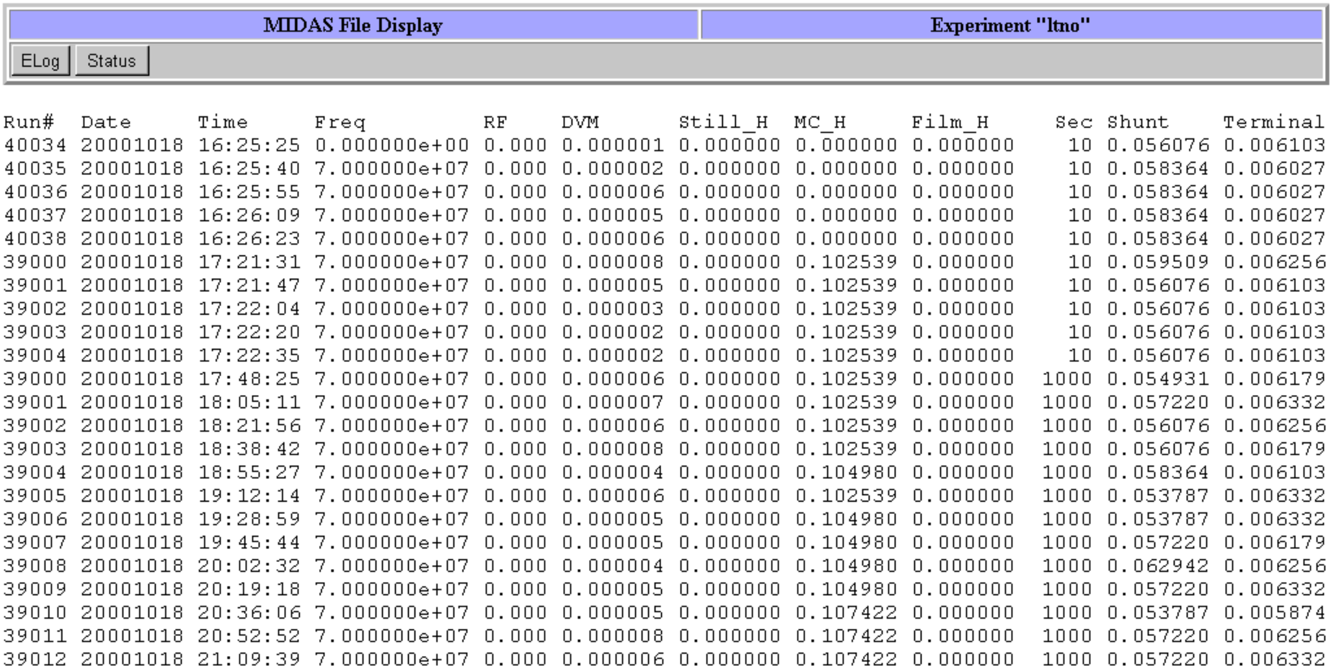
System Status log:

Date	*Successful*	*Unsuccessful or not done yet*
September 14/2000	Optical 100BaseT link to the Shack	

Figure 18: *HTML Elog message.*

The **runlog** button display the content of the file **runlog.txt** which is expected to be in the data directory specified by the ODB key **/Logger/Data Dir**. Regardless of its content, it will be displayed in the web page. Its common uses is to **append** lines after every run. The task appending this run information can be any of the midas application. Example is available in the *examples/experiment/analyzer.c* which at each end-of-run (EOR) will write to the runlog.txt some statistical informations.

When composing a new entry into the Elog, several fields are available to specify the nature of the message i.e: Author, Type, System, Subject. Under Type and System a pulldown menu provides multiple category. These categories are user definable through the odb under the tree **/Elog/Types**, **/Elog/Systems**. The number of category is fixed to 20 maximum but any remaining field can be left empty.



```
Run#  Date     Time     Freq          RF    DVM      Still_H  MC_H     Film_H    Sec Shunt    Terminal
40034 20001018 16:25:25 0.000000e+00 0.000 0.000001 0.000000 0.000000 0.000000   10 0.056076 0.006103
40035 20001018 16:25:40 7.000000e+07 0.000 0.000002 0.000000 0.000000 0.000000   10 0.058364 0.006027
40036 20001018 16:25:55 7.000000e+07 0.000 0.000006 0.000000 0.000000 0.000000   10 0.058364 0.006027
40037 20001018 16:26:09 7.000000e+07 0.000 0.000005 0.000000 0.000000 0.000000   10 0.058364 0.006027
40038 20001018 16:26:23 7.000000e+07 0.000 0.000006 0.000000 0.000000 0.000000   10 0.058364 0.006027
39000 20001018 17:21:31 7.000000e+07 0.000 0.000008 0.000000 0.102539 0.000000   10 0.059509 0.006256
39001 20001018 17:21:47 7.000000e+07 0.000 0.000005 0.000000 0.102539 0.000000   10 0.056076 0.006103
39002 20001018 17:22:04 7.000000e+07 0.000 0.000003 0.000000 0.102539 0.000000   10 0.056076 0.006103
39003 20001018 17:22:20 7.000000e+07 0.000 0.000002 0.000000 0.102539 0.000000   10 0.056076 0.006103
39004 20001018 17:22:35 7.000000e+07 0.000 0.000002 0.000000 0.102539 0.000000   10 0.056076 0.006103
39000 20001018 17:48:25 7.000000e+07 0.000 0.000006 0.000000 0.102539 0.000000 1000 0.054931 0.006179
39001 20001018 18:05:11 7.000000e+07 0.000 0.000007 0.000000 0.102539 0.000000 1000 0.057220 0.006332
39002 20001018 18:21:56 7.000000e+07 0.000 0.000006 0.000000 0.102539 0.000000 1000 0.056076 0.006256
39003 20001018 18:38:42 7.000000e+07 0.000 0.000008 0.000000 0.102539 0.000000 1000 0.056076 0.006179
39004 20001018 18:55:27 7.000000e+07 0.000 0.000004 0.000000 0.104980 0.000000 1000 0.058364 0.006103
39005 20001018 19:12:14 7.000000e+07 0.000 0.000006 0.000000 0.102539 0.000000 1000 0.053787 0.006332
39006 20001018 19:28:59 7.000000e+07 0.000 0.000005 0.000000 0.104980 0.000000 1000 0.053787 0.006332
39007 20001018 19:45:44 7.000000e+07 0.000 0.000005 0.000000 0.104980 0.000000 1000 0.057220 0.006179
39008 20001018 20:02:32 7.000000e+07 0.000 0.000004 0.000000 0.104980 0.000000 1000 0.062942 0.006256
39009 20001018 20:19:18 7.000000e+07 0.000 0.000005 0.000000 0.104980 0.000000 1000 0.057220 0.006332
39010 20001018 20:36:06 7.000000e+07 0.000 0.000005 0.000000 0.107422 0.000000 1000 0.053787 0.005874
39011 20001018 20:52:52 7.000000e+07 0.000 0.000008 0.000000 0.107422 0.000000 1000 0.057220 0.006256
39012 20001018 21:09:39 7.000000e+07 0.000 0.000006 0.000000 0.107422 0.000000 1000 0.057220 0.006332
```

Figure 19: *Elog page, Runlog display.*

6.10.7 Program page

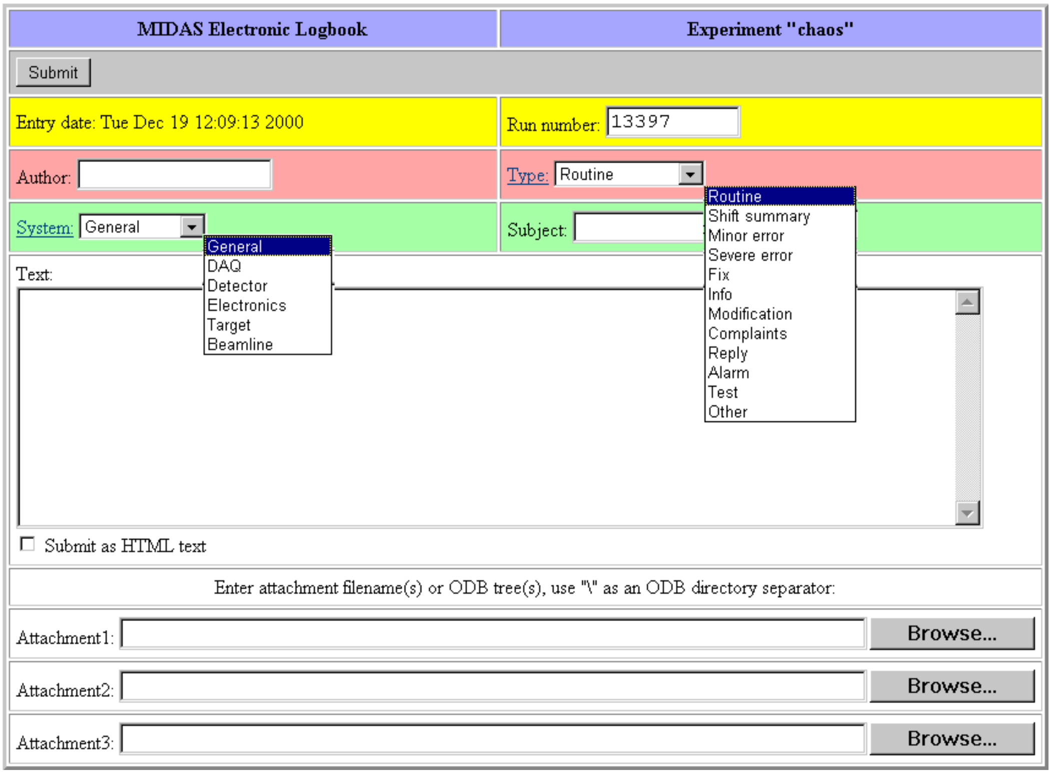
This page present the current active list of the task attached to the given experiment. On the right hand side a dedicated button allows to stop the program which is equivalent to the ODBedit command **odbedit> sh <task name>**.

The task name hyperlink pops a new window pointing to the ODB section related to that program. The ODB structure for each program permit to apply alarm on the task presence condition and automatic spawning at either the begining or the end of a run.

6.10.8 History page

This page reflects the History System settings. It lists on the top of the page the possible pannels defined in the ODB. A serie of buttons defines the full time scale of the graph and the ”<” ”>” ”>>” buttons permit the shifting of the graph in the time direction. The time unit is in minutes. The main graph will always display all the defined channels but clicking on the boxed channel names, single graph will supersede the page.

Elog message with the current history display as attachement for reference is possible using the **Elog** button.



MIDAS Electronic Logbook

Experiment "chaos"

Entry date: Tue Dec 19 12:09:13 2000

Run number: 13397

Author:

Type: Routine

System: General

Subject:

Text:

□ Submit as HTML text

Enter attachment filename(s) or ODB tree(s), use "\" as an ODB directory separator:

Attachment1:

Attachment2:

Attachment3:

Figure 20: *Elog page, New Elog entry form.*

6.10.9

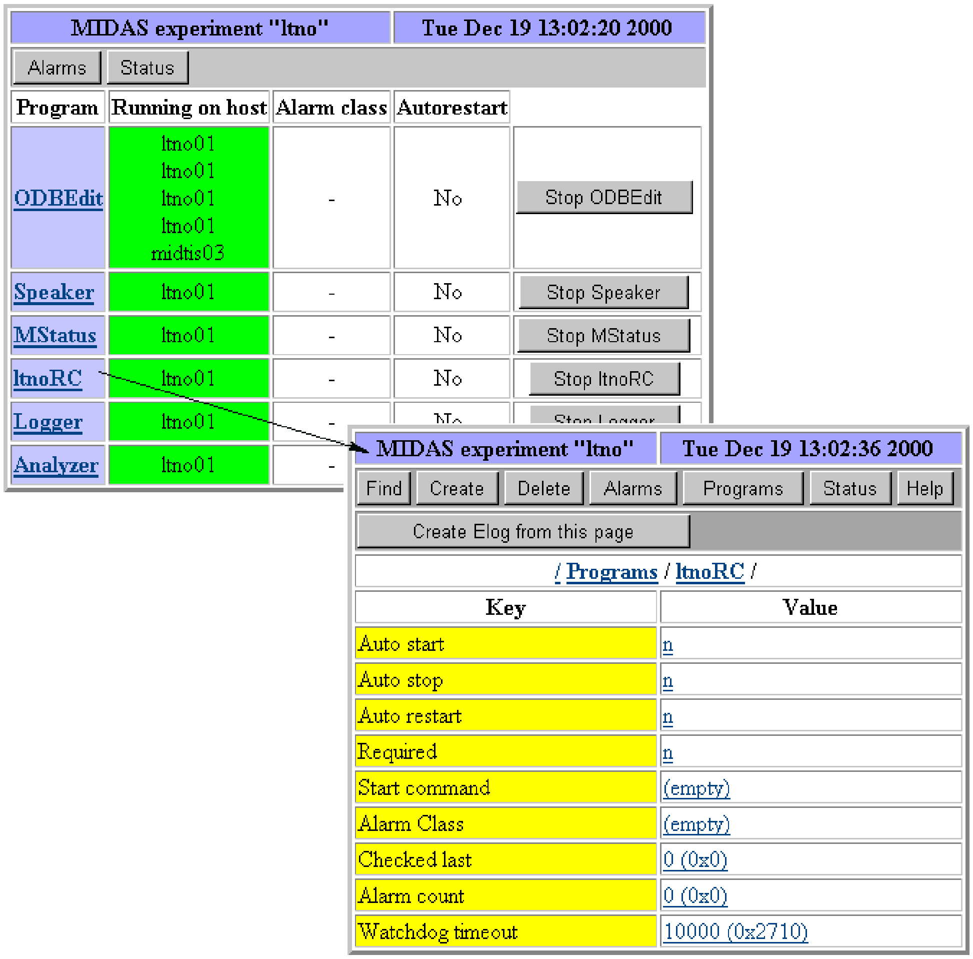
Custom page

The Custom page is available from the Midas version 1.8.3.

This page reflects the html content of a given ODB key under the **/Custom/** key. If keys are defined in the ODB under the **/Custom/** the name of the key will appear in the main status page as the **Alias** keys. By clicking on the Custom page name, the content of the **/Custom/<page>** is interpreted as html content.

The access to the ODB field is then possible using specific HTML tags:

- **<odb src="odb field">** Display ODB field.
- **<odb src="odb field" edit=1>** Display and Editable ODB field.
- <form method="GET" action="http://hostname.domain:port/CS/<Custom_page_key>"> Define method for key access.



MIDAS experiment "ltno"				Tue Dec 19 13:02:20 2000
Program	Running on host	Alarm class	Autorestart	
ODBEdit	ltno01 ltno01 ltno01 ltno01 midtis03	-	No	Stop ODBEdit
Speaker	ltno01	-	No	Stop Speaker
MStatus	ltno01	-	No	Stop MStatus
ltnoRC	ltno01	-	No	Stop ltnoRC
Logger	ltno01	-	No	Stop Logger
Analyzer	ltno01	-		

MIDAS experiment "ltno"	Tue Dec 19 13:02:36 2000
/ Programs / ltnoRC /	
Key	Value
Auto start	n
Auto stop	n
Auto restart	n
Required	n
Start command	(empty)
Alarm Class	(empty)
Checked last	0 (0x0)
Alarm count	0 (0x0)
Watchdog timeout	10000 (0x2710)

Figure 21: *Program page.*

- <meta http-equiv="Refresh" content="60"> Standard page refresh in second.
- <input type=submit name=cmd value=<Midas_page>> Define button for accessing Midas web pages. Valid values are the standard midas buttons (Start, Pause, Resume, Stop, ODB, Elog, Alarms, History, Programs, etc).
- <img src="http://hostname.domain:port/HS/Meterdis.gif&scale=12h&width=300"> Reference to an history page.

The insertion of a new Custom page requires the following steps:

1. Create an initial html file using your prefered HTML editor.
2. Insert the ODB HTML tags at your wish.
3. Invoke ODBedit, create the Custom directory, import the html file.

- Example of loading the file mcustom.html into odb.

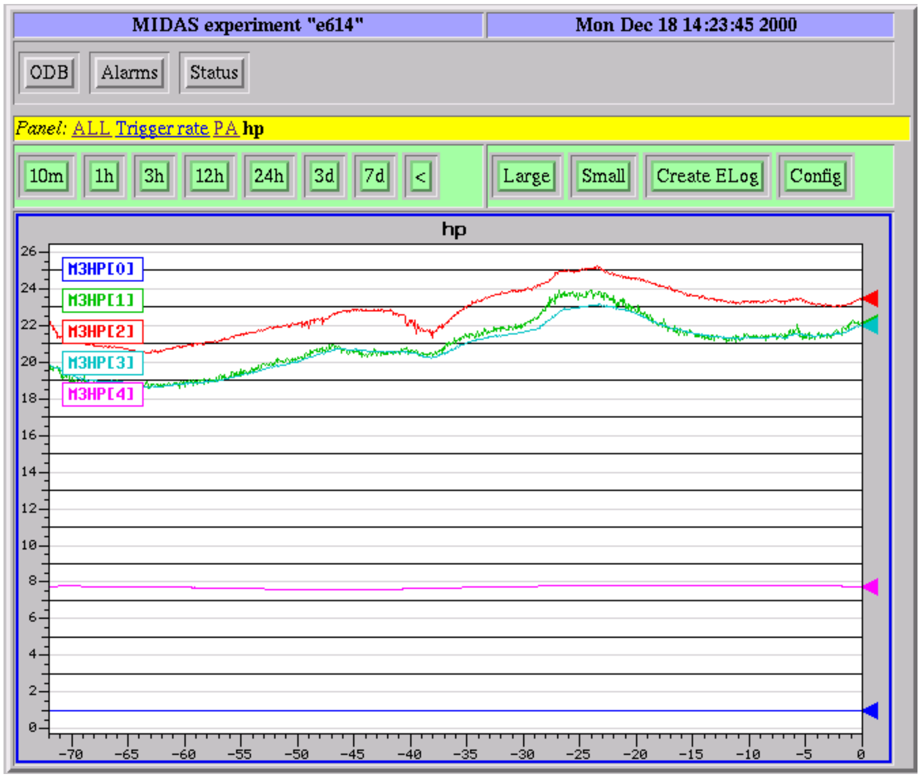


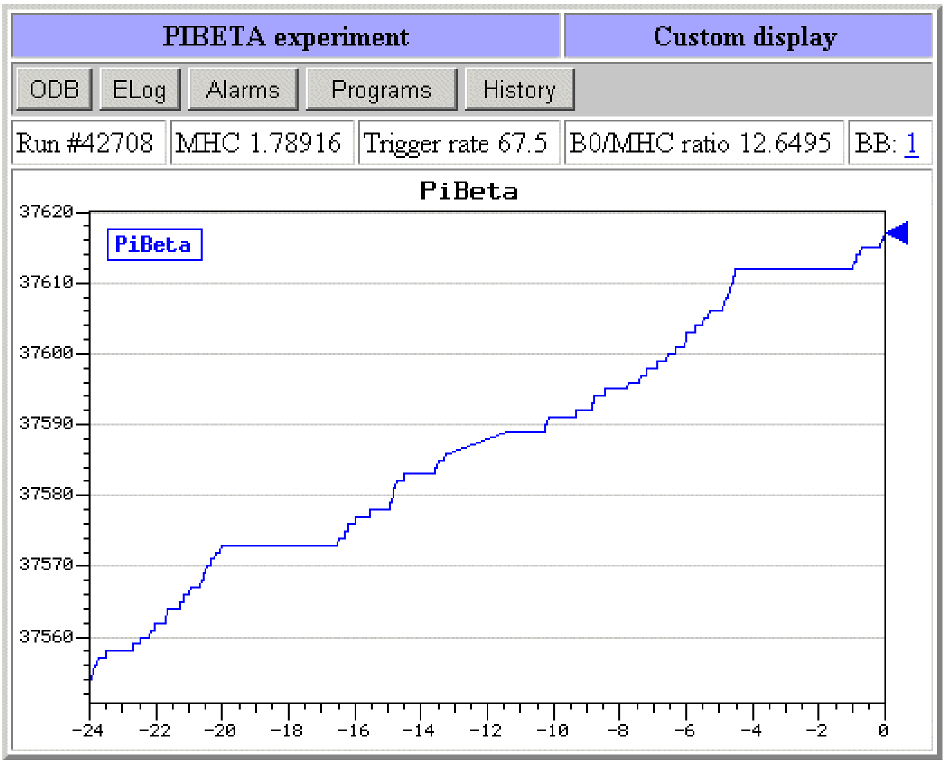
Figure 22: *History page.*

```
Tue> odbedit
[local:midas:Stopped]/>ls
System
Programs
Experiment
Logger
Runinfo
Alarms
Equipment
[local:midas:Stopped]/>mkdir Custom
[local:midas:Stopped]/>cd Custom/
[local:midas:Stopped]/Custom>import mcustom.html
Key name: Test&
[local:midas:Stopped]/Custom>
```

- Once the file is load into ODB, you can **ONLY** edit it through the web (as long as the mhttpd is active). Clicking on the **ODB(button) ... Custom(Key) ... Edit(Hyperlink at the bottom of the key)**. The Custom page can also be exported back to a ASCII file using the ODBedit command "export"

```
Tue> odbedit
[local:midas:Stopped]/>cd Custom/
[local:midas:Stopped]/Custom>export test&
File name: mcustom.html
[local:midas:Stopped]/Custom>
```

- The character "&" (will be changed to "#") at the end of the custom key name forces the



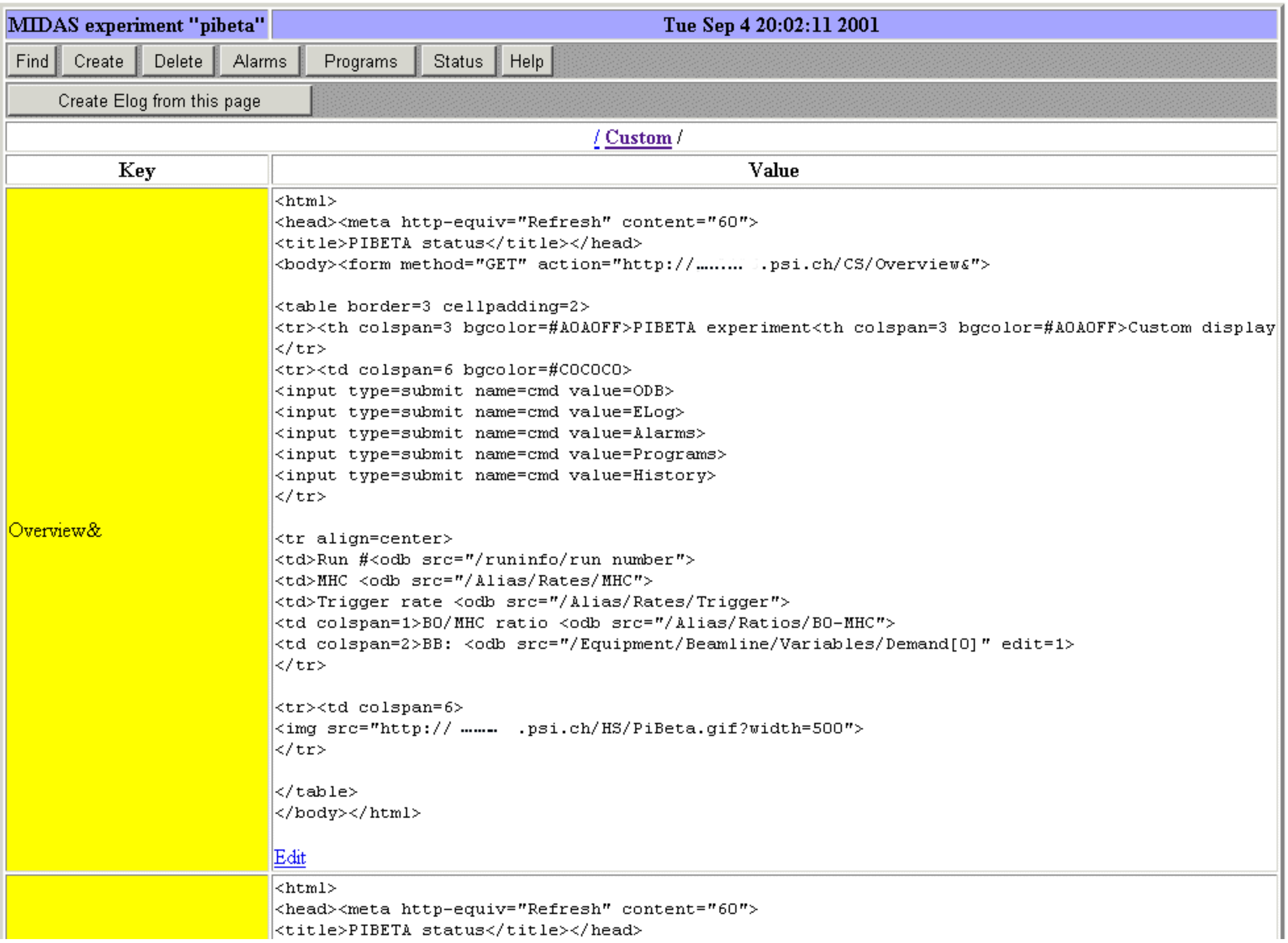
Run #42708 | MHC 1.78916 | Trigger rate 67.5 | B0/MHC ratio 12.6495 | BB: 1

Figure 23: *Custom web page with history graph.*

page to be open within the current frame. If this character is omitted, the page will be spawned into a new frame (default).

- If the custom page name is set to **Status** (no "&") it will become the default midas Web page!
- html code example mcustom.html

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
 <head>
 <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
 <meta name="GENERATOR" content="Mozilla/4.76 [en] (Windows NT 5.0; U) [Netscape]">
 <meta name="Author" content="Pierre-Andr Amaudruz">
 <title>Set value</title>
 </head>
 <body text="#000000" bgcolor="#FFFFCC" link="#FF0000" vlink="#800080" alink="#0000FF">
 <form method="GET" action="http://host.domain:port/CS/WebLtno&">
 <input type=hidden name=exp value="ltno">
  <center><table CELLSPACING=0 CELLPADDING=0 COLS=3 WIDTH="100%" BGCOLOR="#99FF99" >
  <caption><b><font face="Georgia"><font color="#000099"><font size=+2>LTNO
     Custom Web Page</font></font></font></b></caption>
  <tr BGCOLOR="#FFCC99">
  <td><b><font color="#FF0000">Actions: </font></b>
  <input type=submit name=cmd value=Status>
  <input type=submit name=cmd value=Start>
  <input type=submit name=cmd value=Stop>
```



Key | Value

Overview&

```
<html>
<head><meta http-equiv="Refresh" content="60">
<title>PIBETA status</title></head>
<body><form method="GET" action="http://…….. .psi.ch/CS/Overview&">

<table border=3 cellpadding=2>
<tr><th colspan=3 bgcolor=#A0A0FF>PIBETA experiment<th colspan=3 bgcolor=#A0A0FF>Custom display
</tr>
<tr><td colspan=6 bgcolor=#C0C0C0>
<input type=submit name=cmd value=ODB>
<input type=submit name=cmd value=ELog>
<input type=submit name=cmd value=Alarms>
<input type=submit name=cmd value=Programs>
<input type=submit name=cmd value=History>
</tr>

<tr align=center>
<td>Run #<odb src="/runinfo/run number">
<td>MHC <odb src="/Alias/Rates/MHC">
<td>Trigger rate <odb src="/Alias/Rates/Trigger">
<td colspan=1>B0/MHC ratio <odb src="/Alias/Ratios/B0-MHC">
<td colspan=2>BB: <odb src="/Equipment/Beamline/Variables/Demand[0]" edit=1>
</tr>

<tr><td colspan=6>
<img src="http:// ……. .psi.ch/HS/PiBeta.gif?width=500">
</tr>

</table>
</body></html>
```

Edit

```
<html>
<head><meta http-equiv="Refresh" content="60">
<title>PIBETA status</title></head>
```

Figure 24: *ODB /Custom/ html field.*

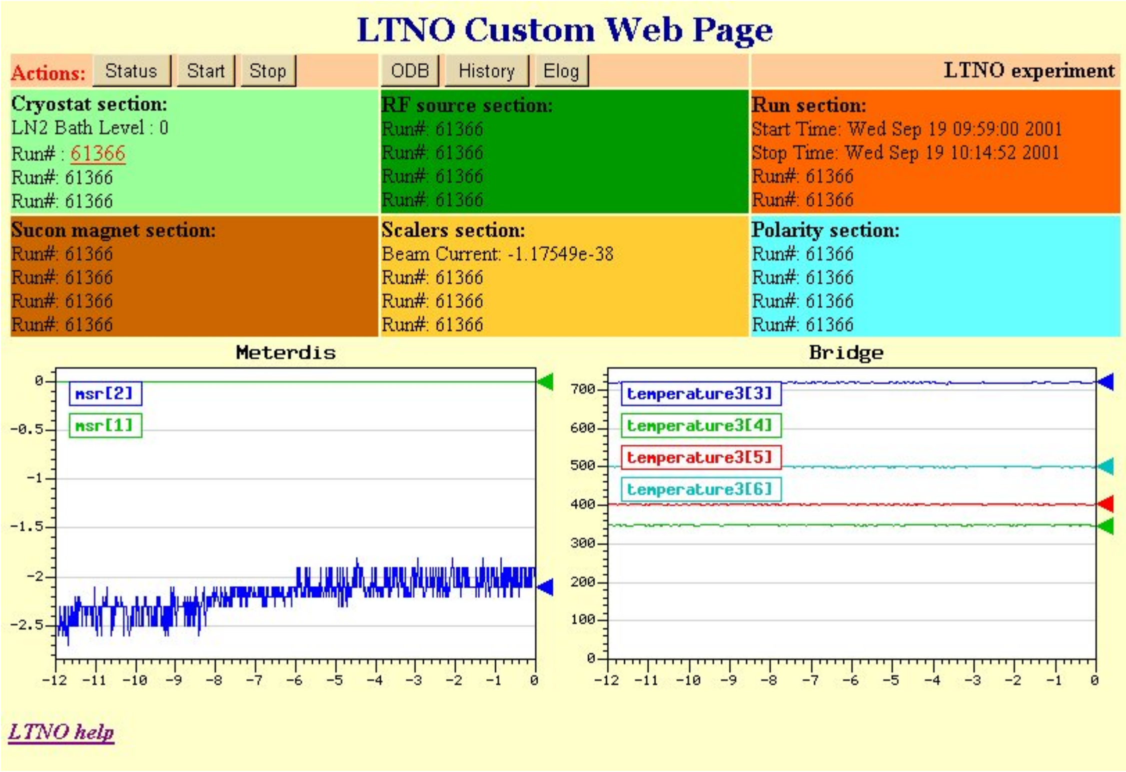
```
<td>
<input type=submit name=cmd value=ODB>
<input type=submit name=cmd value=History>
<input type=submit name=cmd value=Elog></td>
</td>

<td>
<div align=right><b>LTNO experiment </b></div>
</td>
</tr>

<tr>
<td><b>Cryostat section:</b>
<br>LN2 Bath Level : <odb src="/equipment/cryostat/variables/measured[12]">
<br>Run# : <odb src="/runinfo/run number" edit=1>
<br>Run#: <odb src="/runinfo/run number">
<br>Run#: <odb src="/runinfo/run number"></td>

<td WIDTH="100%" BGCOLOR="#009900"><b>RF source section:</b>
```

```
    <br>Run#: <odb src="/runinfo/run number">
    <br>Run#: <odb src="/runinfo/run number">
    <br>Run#: <odb src="/runinfo/run number">
    <br>Run#: <odb src="/runinfo/run number"></td>
    <td WIDTH="50%" BGCOLOR="#FF6600"><b>Run section:</b>
    <br>Start Time: <odb src="/runinfo/start time">
    <br>Stop Time: <odb src="/runinfo/stop time">
    <br>Run#: <odb src="/runinfo/run number">
    <br>Run#: <odb src="/runinfo/run number"></td>
    </tr>

    <tr>
    <td BGCOLOR="#CC6600"><b>Sucon magnet section:</b>
    <br>Run#: <odb src="/runinfo/run number">
    <br>Run#: <odb src="/runinfo/run number">
    <br>Run#: <odb src="/runinfo/run number">
    <br>Run#: <odb src="/runinfo/run number"></td>

    <td BGCOLOR="#FFCC33"><b>Scalers section:</b>
    <br>Beam Current: <odb src="/equipment/epics/variables/measured[10]">
    <br>Run#: <odb src="/runinfo/run number">
    <br>Run#: <odb src="/runinfo/run number">
    <br>Run#: <odb src="/runinfo/run number"></td>

    <td BGCOLOR="#66FFFF"><b>Polarity section:</b>
    <br>Run#: <odb src="/runinfo/run number">
    <br>Run#: <odb src="/runinfo/run number">
    <br>Run#: <odb src="/runinfo/run number">
    <br>Run#: <odb src="/runinfo/run number"></td>
    </tr>
    </table></center>

    <img src="http://host.domain:port/HS/Meterdis.gif?exp=ltno&scale=12h&width=400">
    <img src="http://host.domain:port/HS/Bridge.gif?exp=ltno&scale=12h&width=400">
    <b><i><font color="#000099"><a href="http://host.domain/index.html">
    <br> LTNO help</a></font></i></b>
    </body>
  </html>
```

6.11 elog task

Electronic LogBook utility.

Electronic Log utility. Submit full Elog entry to the specified Elog port.



LTNO Custom Web Page

LTNO experiment

Cryostat section:	RF source section:	Run section:
LN2 Bath Level : 0 Run# : 61366 Run#: 61366 Run#: 61366	Run#: 61366 Run#: 61366 Run#: 61366 Run#: 61366	Start Time: Wed Sep 19 09:59:00 2001 Stop Time: Wed Sep 19 10:14:52 2001 Run#: 61366 Run#: 61366
Sucon magnet section:	**Scalers section:**	**Polarity section:**
Run#: 61366 Run#: 61366 Run#: 61366 Run#: 61366	Beam Current: -1.17549e-38 Run#: 61366 Run#: 61366 Run#: 61366	Run#: 61366 Run#: 61366 Run#: 61366 Run#: 61366

LTNO help

Figure 25: *web page produced by mcustom.html.*

- Arguments

  -h : help

  -h hostname : host name

  -l exptname or logbook

  -u username password

  -f <attachment> : up to 10 files.

  – -a <attribute>=<value> : up to 20 attributes. The attribute ”Author=...” must at least be present for submission of Elog.

  – -m <textfile> — text>

  Arguments with blanks must be enclosed in quotes. The elog message can either be submitted on the command line or in a file with the -m flag. Multiple attributes and attachments can be supplied.

- Usage
  By default the attributes are ”Author”, ”Type”, ”System” and ”Subject”. The ”Author” attribute has to be present in the elog command in order to successfully submit the message. If multiple attributes are required append before ”text” field the full specification of the attribute. In case of multiple attachement, only one ”-f” is required followed by up to 10 file names.

```
>elog -h myhost -p 8081 -l myexpt -a author=pierre "Just a elog message"
>elog -h myhost -p 8081 -l myexpt -a author=pierre -f file2attach.txt \
                "Just this message with an attachement"
>elog -h myhost -p 8081 -l myexpt -a author=pierre -m file_containing_the_message.txt
```

```
>elog -h myhost -p 8081 -l myexpt -a Author=pierre -a Type=routine -a system=general \
                -a Subject="my test" "A full Elog message"
```

- Remarks

  1. ...

6.12 mhist task

History data utility.

History data retriever.

- Arguments

-h	:	help
-e Event ID	:	specify event ID
-v Variable Name	:	specify variable name for given Event ID
-i Index	:	index of variables which are arrays
-t Interval	:	minimum interval in sec. between two displayed records
-h Hours	:	display between some hours ago and now
-d Days	:	display between some days ago and now
-f File	:	specify history file explicitly
-s Start date	:	specify start date DDMMYY[.HHMM[SS]]
-p End date	:	specify end date DDMMYY[.HHMM[SS]]
-l	:	list available events and variables
-b	:	display time stamp in decimal format
-z	:	History directory (def: cwd).

- Usage
- Examples

```
--- All variables of event ID 9 during last hour with at least 5 minutes interval.
> mhist
Available events:
ID 9: Target
ID 5: CHV
ID 6: B12Y
ID 20: System

Select event ID: 9

Available variables:
0: Time
```

```
1: Cryostat vacuum
2: Heat Pipe pressure
3: Target pressure
4: Target temperature
5: Shield temperature
6: Diode temperature

Select variable (0..6,-1 for all): -1

How many hours: 1

Interval [sec]: 300

Date    Time    Cryostat vacuum Heat Pipe pressure Target pressure Target temperature      Shield temperat
Jun 19 10:26:23 2000    104444  4.614   23.16   -0.498  22.931  82.163  40
Jun 19 10:31:24 2000    104956  4.602   23.16   -0.498  22.892  82.108  40
Jun 19 10:36:24 2000    105509  4.597   23.099  -0.498  22.892  82.126  40
Jun 19 10:41:33 2000    110021  4.592   23.16   -0.498  22.856  82.08   40
Jun 19 10:46:40 2000    110534  4.597   23.147  -0.498  22.892  82.117  40
Jun 19 10:51:44 2000    111046  4.622   23.172  -0.498  22.907  82.117  40
Jun 19 10:56:47 2000    111558  4.617   23.086  -0.498  22.892  82.117  40
Jun 19 11:01:56 2000    112009  4.624   23.208  -0.498  22.892  82.117  40
Jun 19 11:07:00 2000    112521  4.629   23.172  -0.498  22.896  82.099  40
Jun 19 11:12:05 2000    113034  4.639   23.074  -0.498  22.896  82.117  40
Jun 19 11:17:09 2000    113546  4.644   23.172  -0.498  22.892  82.126  40
Jun 19 11:22:15 2000    114059  4.661   23.16   -0.498  22.888  82.099  40
```

— Single variable "I-WC1+_Anode" of event 5 every hour over the full April 24/2000.

```
 mhist -e 5 -v "I-WC1+_Anode" -t 3600 -s 240400 -p 250400
Apr 24 00:00:09 2000    160
Apr 24 01:00:12 2000    160
Apr 24 02:00:13 2000    160
Apr 24 03:00:14 2000    160
Apr 24 04:00:21 2000    180
Apr 24 05:00:26 2000    0
Apr 24 06:00:31 2000    160
Apr 24 07:00:37 2000    160
Apr 24 08:00:40 2000    160
Apr 24 09:00:49 2000    160
Apr 24 10:00:52 2000    160
Apr 24 11:01:01 2000    160
Apr 24 12:01:03 2000    160
Apr 24 13:01:03 2000    0
Apr 24 14:01:04 2000    0
Apr 24 15:01:05 2000    -20
Apr 24 16:01:11 2000    0
Apr 24 17:01:14 2000    0
Apr 24 18:01:19 2000    -20
Apr 24 19:01:19 2000    0
Apr 24 20:01:21 2000    0
Apr 24 21:01:23 2000    0
Apr 24 22:01:32 2000    0
Apr 24 23:01:39 2000    0
```

- Remarks

  1. ...

- Example History data can be retrieved and display through the Midas web page (see mhttpd task).

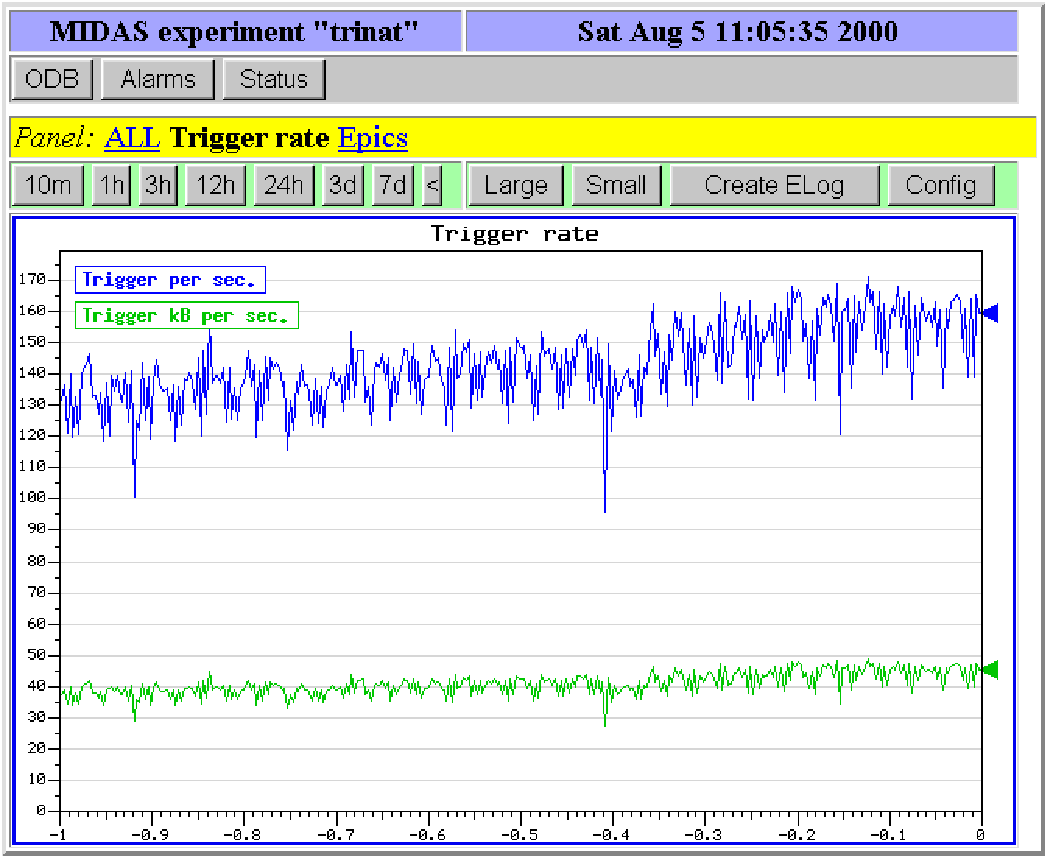


Figure 26: *Midas Web History display.*

6.13 mchart task

ODB data for stripchart utility.

mchart is a periodic data retriever of a specific path in the ODB which can be used in conjunction with a stripchart graphic program.

- In the first of two step procedure, a specific path in the ODB can be scanned for composing a configuration file by extracting all numerical data references *file.conf*.

- In the second step the mchart will produce at fix time interval a refreshed data file *file* containing the values of the numerical data specified in the configuration file. This file is then available for a stripchart program to be used for chart recording type of graph.

Two possible stripchart available are:

- **gstripchart** The configuration file generated by mchart is compatible with the GNU stripchart which permit sofisticated data equation manipulation. In the other hand, the data display is not very fency and provide just a basic chart recorder.
- **stripchart.tcl** This tcl/tk application written by Gertjan Hofman provides a far better graphical chart recorder display tool, it also permits history save-set display, but the equation scheme is not implemented.

- Arguments

-h	: help
-h hostname	: host name.
-e exptname	: experiment name.
-D	: start program as a daemon.
-u time	: data update periodicity (def:5s).
-f file	: file name (+.conf: if using existing file).
-q ODBpath	: ODB tree path for extraction of the variables.
-c	: ONLY creates the configuration file for later use.
-b lower_value	: sets general lower limit for all variables.
-t upper_value	: sets general upper limit for all variables.
-g	: spawn the graphical stripchart if available.
-gg	: force the use of gstripchart for graphic.
-gh	: force the use of stripchart (tcl/tk) for graphic.

- Usage

  The configuration contains an entry for each variable found under the ODBpath requested. The format is described in the gstripchart documentation.

  Once the configuration file has been created, it is possible to apply any valid operation (equation) to the parameters of the file following the gstripchart syntax.

  In the case of the use of the *stripchart* from G.Hofman, only the ”filename”, ”pattern”, ”maximum”, ”minimum” fields are used.

  When using mchart with -D Argument, it is necessary to have the *MCHART_DIR* defined in order to allow the daemon to find the location of the configuration and data files (see Environment variables).

```
chaos:~/chart> more trigger.conf
#Equipment:            >/equipment/kos_trigger/statistics
menu:                   on
slider:                 on
type:                   gtk
minor_ticks:            12
major_ticks:            6
chart-interval:         1.000
```

```
chart-filter:           0.500
slider-interval:        0.200
slider-filter:          0.200
begin:        Events_sent
  filename:     /home/chaos/chart/trigger
  fields:       2
  pattern:      Events_sent
  equation:     $2
  color:        blue
  maximum:      1083540.00
  minimum:      270885.00
  id_char:      1
end:            Events_sent
begin:        Events_per_sec.
  filename:     /home/chaos/chart/trigger
  fields:       2
  pattern:      Events_per_sec.
  equation:     $2
  color:        red
  maximum:      1305.56
  minimum:      326.39
  id_char:      1
end:            Events_per_sec.
begin:        kBytes_per_sec.
  filename:     /home/chaos/chart/trigger
  fields:       2
  pattern:      kBytes_per_sec.
  equation:     $2
  color:        brown
  maximum:      898.46
  minimum:      224.61
  id_char:      1
end:            kBytes_per_sec.
```

A second file (data file) will be updated a fixed interval by the *mchart* utility.

```
chaos:~/chart> more trigger
  Events_sent 6.620470e+05
  Events_per_sec. 6.463608e+02
  kBytes_per_sec. 4.424778e+02
```

- Examples
  - Creation with ODBpath being one array and one element of 2 sitting under variables/:

    ```
    chaos:~/chart> mchart -f chvv -q /equipment/chv/variables/chvv -c
    chaos:~/chart> ls -l chvv*
    -rw-r--r--   1 chaos    users         474 Apr 18 14:37 chvv
    -rw-r--r--   1 chaos    users        4656 Apr 18 14:37 chvv.conf
    ```

  - Creation with ODBpath of all the sub-keys sittings in variables:

```
mchart -e myexpt -h myhost -f chv -q /equipment/chv/variables -c
```

– Creation and running in debug:

```
chaos:~/chart> mchart -f chv -q /equipment/chv/variables -d
CHVV : size:68
#name:17 #Values:17
CHVI : size:68
```

– Running a pre-existing conf file (chv.conf) debug:

```
chaos:~/chart> mchart -f chv.conf -d
CHVV : size:68
#name:17 #Values:17
CHVI : size:68
#name:17 #Values:17
```

– Running a pre-existing configuration file and spawning gstripchart:

```
chaos:~/chart> mchart -f chv.conf -gg
spawning graph with gstripchart -g 500x200-200-800 -f /home/chaos/chart/chv.conf ...
```

– Running a pre-existing configuration file and spawning stripchart, this will work only if Tcl/Tk and bltwish packages are installed and the stripchart.tcl has been installed through the Midas Makefile.

```
chaos:~/chart> mchart -f chv.conf -gh
spawning graph with stripchart /home/chaos/chart/chv.conf ...
```

6.14 mtape task

Tape utility.

Tape manipulation utility.

- Arguments

-h	: help
-h hostname	: host name
-e exptname	: experiment name
-D	: start program as a daemon

- Usage
- Examples

```
>mtape
```

6.15 dio task

Frontend or mcnaf Direct IO to CAMAC launcher.

Direct I/O task provider (LINUX).

If no particular Linux driver is installed for the CAMAC access, the **dio** program will allows you to gain access to the I/O ports to which the CAMAC interface card is connected to.

- Arguments

application name : Program name requiring I/O permission.

- Usage

```
>dio miocnaf

>dio frontend
```

- Remarks

1. This ”hacking” utility restricts the access to a range of I/O port from 0x200 to 0x3FF.
2. As this mode of I/O access by-passes the driver (if any), concurrent access to the same I/O port may produce unexpected result and in the worth case freeze the computer. It is therefore important to ensure to run one and only one dio application to a given port in order to prevent potential hangup problem.
3. Interrupt handling, DMA capabilities of the interface will not be accessible under this mode of operation.

6.16 stripchart.tcl

Tcl/Tk history/ODB data stripchart display.

Graphical stripchart data display. Operates on mchart task data or on Midas history save-set files. (see also History System).

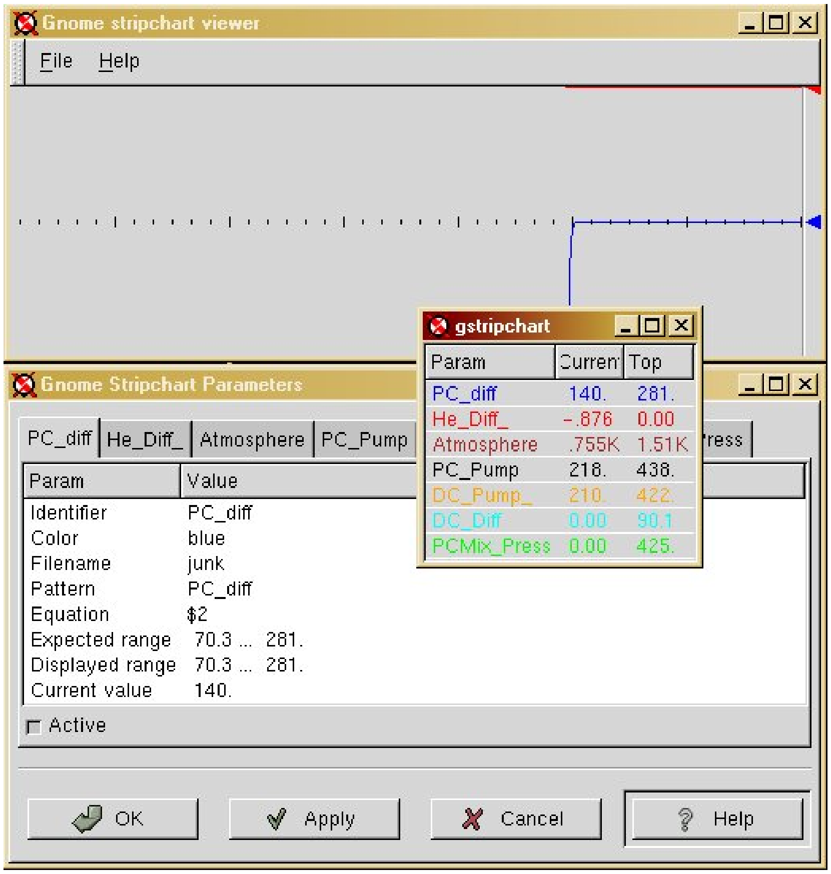
- Arguments

-mhist : start stripchart for Midas history data.

- Usage

- Examples

```
> stripchart.tcl -h
Usage: stripchart <-options> <config-file>
-mhist   (look at history file -default)
-dmhist  debug mhist
-debug   debug stripchart
config_file:  see mchart

> stripchart.tcl -debug
> stripchart.tcl
```

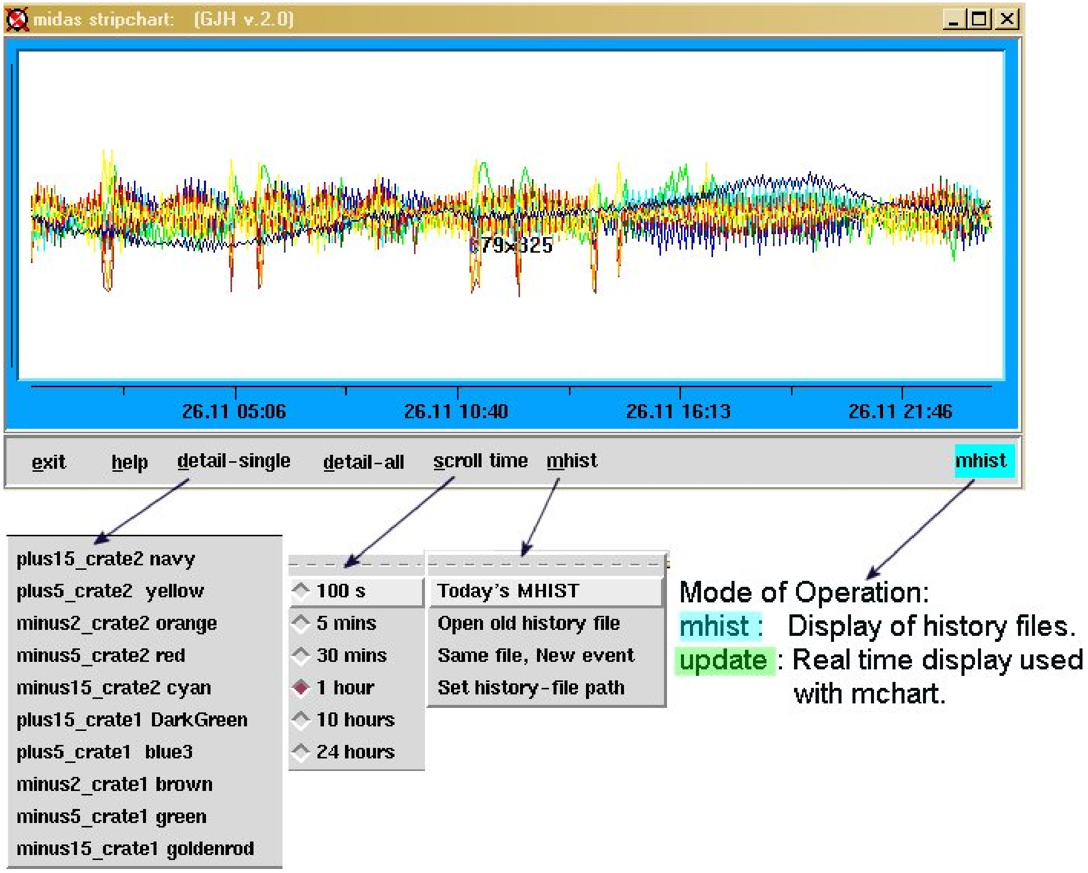


Param	Curren	Top
PC_diff	140.	281.
He_Diff_	-.876	0.00
Atmosphere	.755K	1.51K
PC_Pump	218.	438.
DC_Pump_	210.	422.
DC_Diff	0.00	90.1
PCMix_Press	0.00	425.

Param	Value
Identifier	PC_diff
Color	blue
Filename	junk
Pattern	PC_diff
Equation	$2
Expected range	70.3 ... 281.
Displayed range	70.3 ... 281.
Current value	140.

□ Active

Figure 27: *gstripchart display with parameters and data pop-up*



Mode of Operation:
mhist : Display of history files.
update : Real time display used with mchart.

Figure 28: *stripchart.tcl mhist mode: main window with pull-downs.*

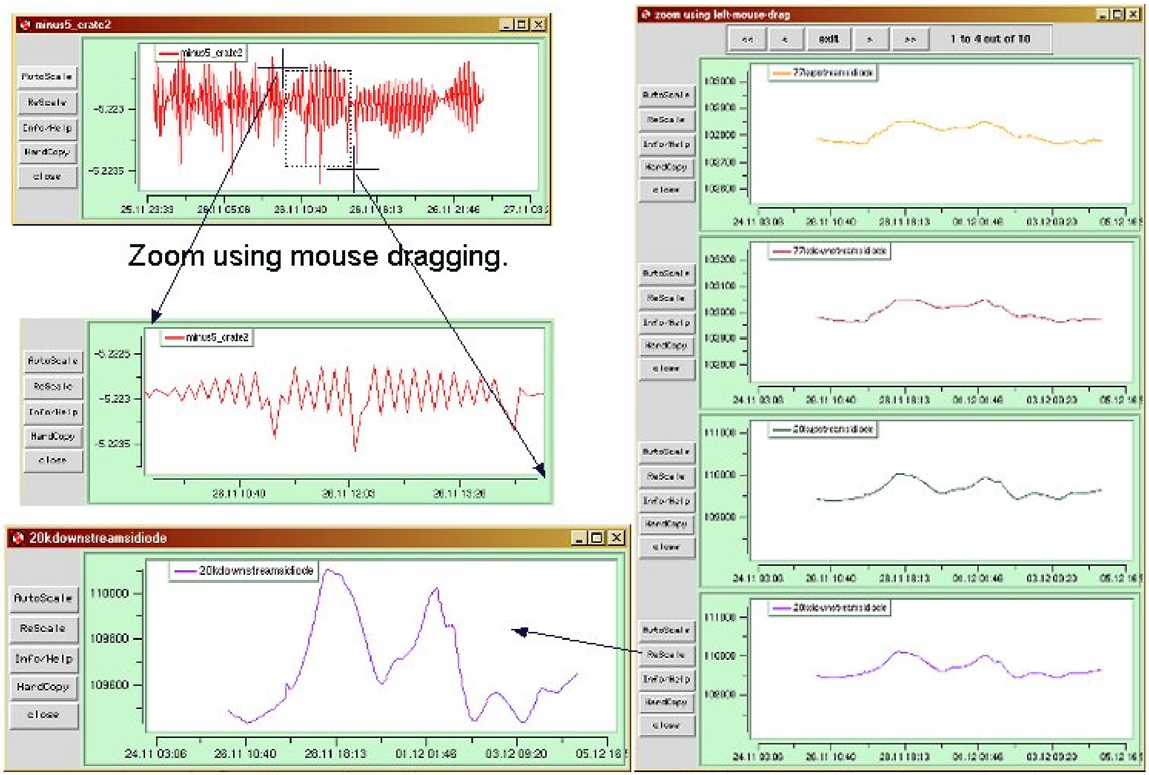
6.17 hvedit task

HV or Slow control Windows application editor.

High Voltage editor, graphical interface to the Slow Control System (NT).

- Arguments

-h	:	help
-h hostname	:	host name
-e exptname	:	experiment name



Zoom using mouse dragging.

Rescale button

Detail-all selection from main window:
- Display 4 graphs per page.
- Page selection on top of window.

Figure 29: *stripchart.tcl: Channel selection with zoom option.*

-D : start program as a daemon

- Usage To control the high voltage system, the program HVEdit can be used under Windows 95/NT. It can be used to set channels, save and load values from disk and print them. The program can be started several times even on different computers. Since they are all linked to the same ODB arrays, the demand and measured values are consistent between them at any time. HVEdit is started from the command line:
- Examples

```
>hvedit
```

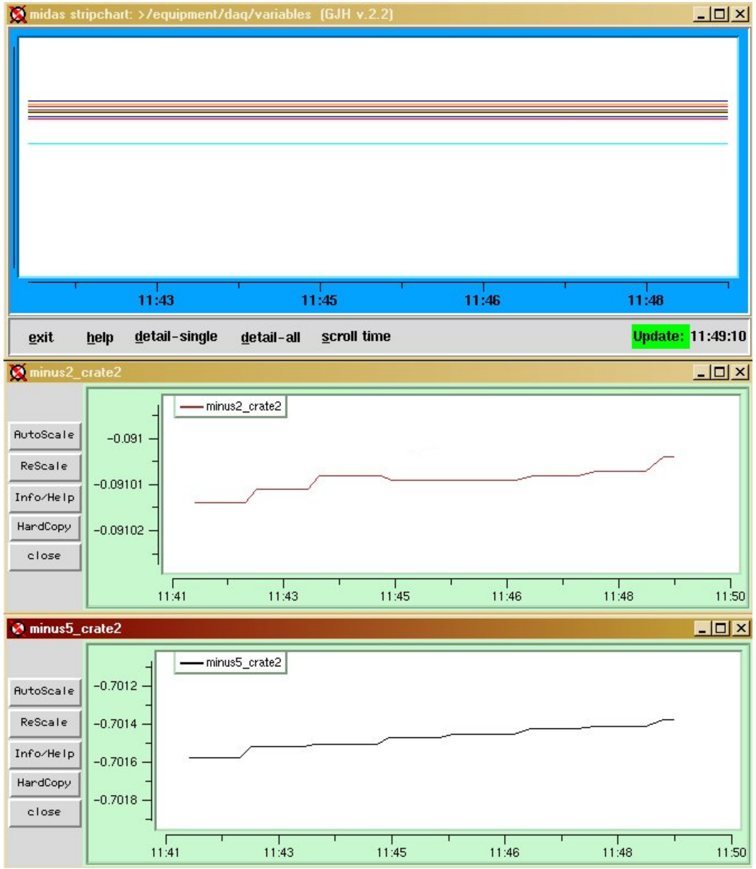


Figure 30: *stripchart.tcl: Online data, running in conjunction with mchart.*

7 New application

List of application examples.

New applications can be build using the examples given under *midas/examples/...*

- midas/examples/lowlevel/produce.c A simple data producer. It simply connects to a buffer on the local machine (not possible under MS-DOS) or to a remote host which has the server program running. An event-id can be selected (e.g. 1 or 2 or so), a host (either enter RETURN for the local host or the IP name for a remote host), an event size (1 Byte up to 64k) and a cache size. The cache is installed on the machine where the buffer is created and limits the access to the buffer by caching the data. This avoids numerous semaphore calls and speeds up data transfer expecially for small event sizes. The producer just generates dummy events until it is interrupted via Ctrl-C.
- midas/examples/lowlevel/consume.c A simple consumer getting data from a local or remote host and checking the data inegrety. First, the first and last word in the events is checked in order to detect overwritten data, then the serial number of the event is checked in order to detect lost events. By specifying different id's, several types of events can be passed through the same buffer although this is not advisable for high rate applications.
- midas/examples/lowlevel/rpc_test.c A simple program to test the RPC layer in MIDAS. It connects to a MIDAS server and executes a test routine on the server.
- midas/examples/lowlevel/rpc_srvr.c, rpc_clnt.c This is a standalone RPC server and client which do not use any midas functionality. This can be used to implement simple RPC client/server programs which have nothing to do with midas. This example won't works on VxWorks.
- midas/examples/basic/minirc.c Mini-Runcontrol program which can be used to start and stop a run. Usually this is done inside odbedit. Provides example for transition request.
- midas/examples/basic/minife.c Mini-Frontend showing the basic concept of a MIDAS frontend. It connects to an experiment and and waits for a run start command via an RPC call. When a run is started, it sends empty events with ID 1. Normally, the mfe.c frontend framework is used to build a frontend. This program is more evolved and provide internal structure of the frontend scheduler with RPC server capability. It is recommended for "expert".
- midas/examples/basic/odb_test.c A simple test program which shows basic ODB operation. It read a value from the online database (the run number), increments it and writes it back to the ODB. Then it opens a "hot link" to that value and enters an infinite loop. Whenever the run number gets changed by someone else (like from ODBEdit), the local function run_number_changes is called automatically.
- midas/examples/basic/analyzer.c Very basic analyzer which requests two event types (trigger event ID 1 and scaler event ID 2). It simply prints a notification when one of the events is received. This program has to becompiles together with mana.c to work properly.

- midas/examples/basic/msgdump.c Displays all messages produced via the cm_msg call in other clients.
- midas/examples/macro/frontend.c Shows the use of macro definition (midas_macro.h) in order to code frontend in a higher level of language.
- midas/examples/miniexp/... This directory contains an example of a MIDAS experiment. It contains an even definition for a trigger event (ID 1) with eight ADC and TDC channels and a scaler event (ID 2) also with eight channels. The frontend program generates both types of events and fills them with data. The data is stored in the events in MIDAS bank format which is very similar to BOS/YBOS banks. The analyzer uses a single user routine to recover these banks from the events and analyzes them, then fills them into histograms and N-tuples. Refer to the *readme.txt* file.
- midas/examples/slowcont/frontend.c This file provide a complete example of a slow control frontend containing 16 channels of High Voltage, 2 general purpose Input signal and 2 general purpose output signal.

8 appendix A: Data format

Names

Midas supports two differents data format so far. A possible new candidate would be the NeXus format, but presently no implementation has been developed.

1. MIDAS format
2. YBOS format

8.1 Midas format

Special formats are used in MIDAS for the event header, banks and when writing to disk or tape. This appendix explains these formats in detail. Each event carries a 16-byte header. The header is generated by the front-end with the bm_compose_event() routine and used by the consumers to distinguish between different events. The header is defined in the EVENT_HEADER structure in *midas.h*. It has following structure:

The event ID describes the type of event. Usually 1 is used for trigger events, 2 for scaler events, 3 for HV events etc. The trigger mask can be used to describe the sub-type of an event. A trigger event can have different trigger sources like "physics event", "calibration event", "clock event". These trigger sources are usually read in by the front-end in a pattern unit. Consumers can request events with a specific trigger mask. The serial number starts at one and is incremented by the front-end for each event. The time stamp is written by the front-end before an event is read out. It uses the time() function which returns the time in seconds since 1.1.1970 00:00:00 UTC. The data size contains the number of bytes that follow the event header. The data area of the event can contain information in any user format, although only certain formats are supported when events are copied to the ODB or written by the logger in ASCII format. Event headers are always kept in the byte ordering of the local machine. If events are sent over the network between computers with different byte ordering, the event header is swapped automatically, but not the event contents.

Bank Format Events in MIDAS format contain "MIDAS banks". A bank is a substructure of an event and can contain one type of data, either a single value or an array of values. Banks have a name of exactly four characters, which are treated, as a bank ID. Banks in an event consist of a global bank header and an individual bank header for each bank. Following picture shows a MIDAS event containing banks:

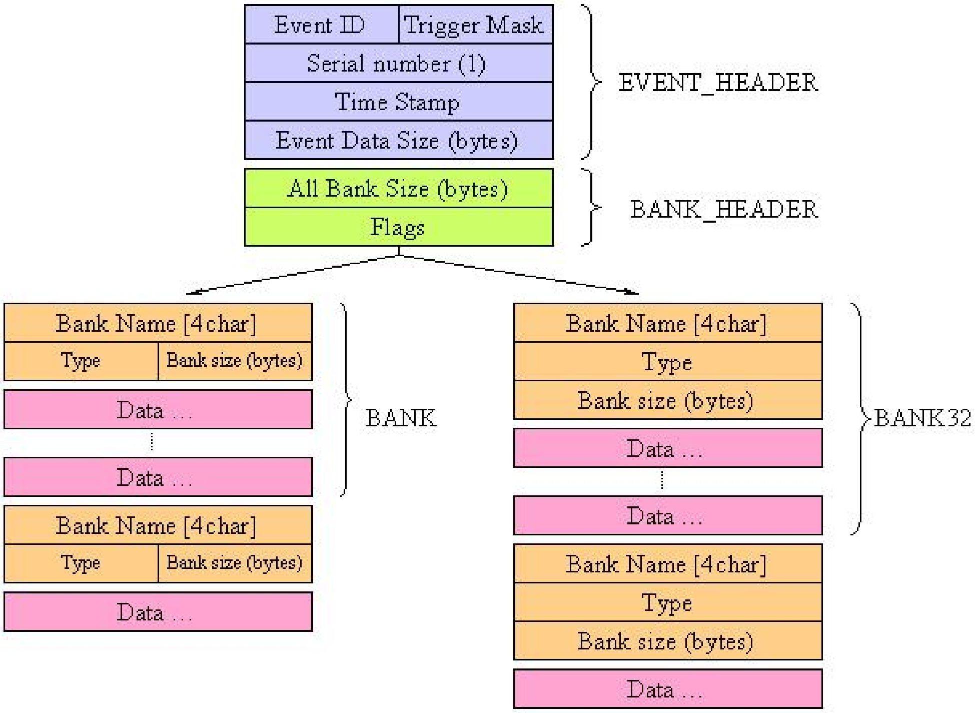


Figure 31: *Event and bank headers with data block.*

The "data size total" is the size in bytes of all bank headers and bank data. Flags are currently not used. The bank header contains four characters as identification, a bank type that is one of the TID_xxx values defined in midas.h, and the data size in bytes. If the byte ordering of the contents of a complete event has to be swapped, the routine bk_swap() can be used.

Tape Format Events are written to disk files without any reformatting. For tapes, a fixed block size is used. The block size TAPE_BUFFER_SIZE is defined in midas.h and usually 32kB. Three special events are produced by the system. A begin-of-run (BOR) and end-of-run (EOR) event is produced which contains an ASCII dump of the ODB in its data area. Their IDs is 0x8000 (BOR) and 0x8001 (EOR). A message event (ID 0x8002) is created if Log messages is enabled in the logger channel setting. The message is contained in the data area as an ASCII string. The BOR event has the number MIDAS_MAGIC (0x494d or 'MI') as the trigger mask and the current run number as the serial number. A tape can therefore be identified as a MIDAS formatted tape. The routine tape_copy() in the utility mtape.c is an example of how to read a tape in MIDAS format.

8.2 YBOS format

As mentioned earlier the YBOS documentation is available at the following URL address: ybos Originally YBOS is a collection of FORTRAN functions which facilitate the manipulation of group of data. It also describes a mode of encoding/storing data in a organized way. YBOS defines specific ways for:

1. Gathering related data (bank structure).
2. Gathering banks structure (logical record).
3. Gathering/Writing/Reading logical record from/to storage device such as disk or tape. (Physical record).

YBOS is organized on a 4-byte alignment structure.

The YBOS library function provides all the tools for manipulation of the above mentioned elements in a independent Operating System like. But the implementation of the YBOS part in Midas does not use any reference to the YBOS library code. Instead only the strict necessary functions have be re-written in C and incorporated into the Midas package. This has been motivated by the fact that only a sub-set of function is essential to the operation of:

- The front-end code: for the composition of the YBOS event (bank structure, logical record).
- The data logger: for writing data to storage device (physical record).

This Midas/YBOS implementation restricts the user to a subset of the YBOS package only for the front-end part. It doesn't prevent him/her to use the full YBOS library for stand alone program accessing data file written by Midas.

The YBOS implementation under Midas has the following restrictions:

- Single leveled bank structures only (no recursive bank allowed).
- Bank structure of the following type: ASCII, BINARY, WORD , DOUBLE WORD, IEEE FLOATING.
- No mixed data type bank structure allowed.

`Logical Record format (Event Format)` In the YBOS terminology a logical record refers to a collection of YBOS bank while in the Midas front-end, it can be referred to as an event. The logical record consists of a logical record length of a 32bit-word size followed by a single or collection of YBOS bank. The logical record length counts the number of double word (32bit word) composing the record without counting itself.

YBOS uses "double word" unit for all length references.

Bank Format The YBOS bank is composed of a bank header 5 double long words followed by the data section which has to end on a 4 bytes boundary.

The bank length parameter corresponds to the size of the data section in double word count + 1. The supported bank type are defined in the /include/ybos.h file see YBOS data types.

Physical Record (Tape/Disk Format) The YBOS physical record structure is based on a fixed block size (8190 double words) composed of a physical record header followed by data from logical records.

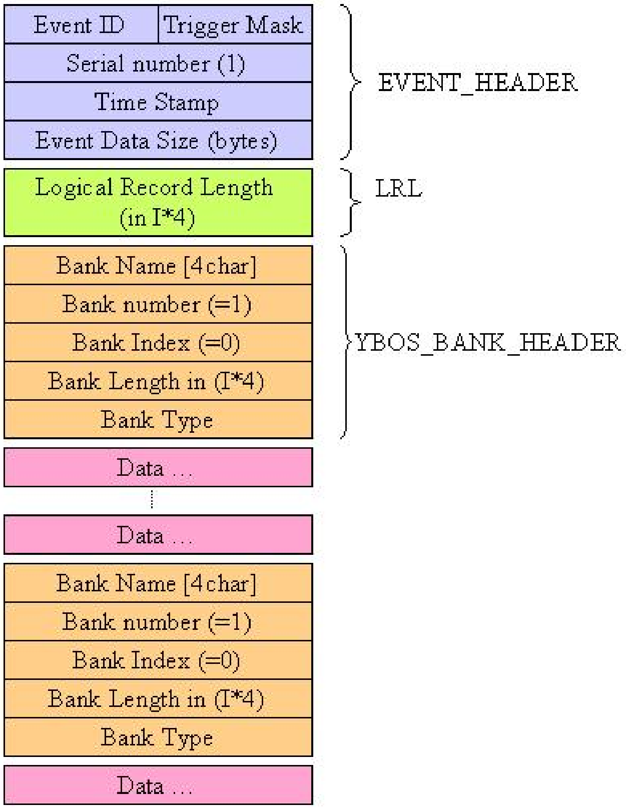


Figure 32: *Ybos Event and bank headers with data block.*

The Offset is computed with the following rules:

1. If the logical record fits completely in the space of the physical record, the offset value in the physical record header will be 4.
2. If the block contains first the left over fragment of the previous event started in the previous block, the offset will be equal to the length of the physical record header + the left over fragment size.
3. If the logical record extent beyond a full block, the offset will be set to -1.
4. The mark of the end of file is defined with a logical record length set to -1.

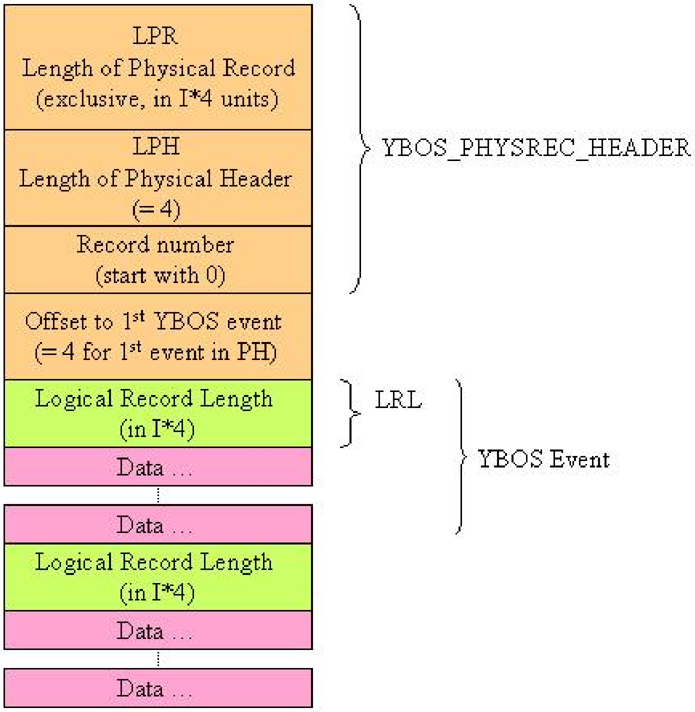


Figure 33: *Ybos Physical record structure with data block.*

9 appendix B: Supported hardware

Names

Drivers included in the driver's directory of the MIDAS distribution support various hardware modules. The driver library is continuously extended to suit the needs of various experiments. The term "any OS" refers to NT, Win9x, Linux, DOS, VxWorks. The VMS is no longer supported.

The following list gives an overview of the current supported hardware:

File	Device Model	Manufacturer	Device Type	Environment
mcstd.h	-	-	Midas CAMAC standard functions	any OS
esone.h/c	-	-	ESONE CAMAC standard functions	any OS
camacnul.c	-	-	NULL CAMAC driver	any OS
camacrpc.c	-	-	Midas RPC CAMAC server	any OS
hyt1331.c	HYT1331	HYTEC Ltd.	PC-CAMAC interface	MSDOS, Linux
kcs2926.c	KS2926/3922	Kinetics Ltd.	PC-CAMAC interface 8bits PC card	NT/Linux
kcs2927.c	KS2927/3922	Kinetics Ltd.	PC-CAMAC interface 16bits PC card	NT/Linux
dsp004.c	DSP	DSP Ltd.	PC-CAMAC interface 8bits PC card	NT/Linux
wecc32.c	CC32	W-ie-n-er	PC-CAMAC interface PCI card	Linux
ps7106.c	PS 7106	Phillips Scientific	CAMAC discriminator	Uses hyt1331.h
lrs2365.c	LRS 2365	LeCroy	CAMAC logic unit	Uses hyt1331.h
lrs1151.c	LRS1151	LeCroy	VME 16 channel scalers	VME/VxWorks
lrs1190.c	LRS1190	LeCroy	VME dual ported memory	VME/VxWorks
vmeio.c	VMEIO	TRIUMF	VME 24bit I/O	VME/VxWorks
lrs1821.c/h	LRS 1821	LeCroy	FASTBUS Segment Manager Interface	MS-DOS
lrs2373.c/h	LRS 2373	LeCroy	Memory Lookup Unit	Uses hyt1331.h
rs232.c/h	Standard PC	Various	RS232 serial port	MS-DOS/Windows NT

Follows the class drivers and corresponding device drivers for the slow control system:

File	Device Model	Manufacturer	Device Type	Environment
cd_hv.h/c	Class driver	-	High Voltage	All
null.c	-	-	dummy driver	All
lrs1440.c	LRS 1440	LeCroy	High voltage system	Uses rs232.c
lrs4032.c	LRS 4032	LeCroy	High voltage system	Uses esone.c
caen170a.c	CAEN 170A	CAEN	High voltage system	Uses esone.c
cd_multi.c/h	Class driver	-	Analog In/Out	All
das1600.c	DAS 1600	Keithley	Analog/Digital In/Out	MS-DOS/Windows NT
dastemp	DAS-TEMP	Keithley	Temperature meas.	MS-DOS/Windows NT
cd_gen.c/h	Class driver	-	Generic I/O	All
ch_acc.c	EPICS channel access	-	Remote channel access	All

9.1 CAMAC drivers

The CAMAC drivers can be used in different configuration and may have special behaviors depending on the type of hardware involved. Below are summurized some remarks about these particular hardware modules.

hyt1331.c This interface uses an ISA board to connect to the crate controller. This card implement a "fast" readout cycle by re-triggering the CAMAC read at the end of the previous one. This feature is unfortunately not reliable when fast processor is used. Wrong returned data can be expected when CPU clocks is above 250MHz. Attempt on "slowing down" the IO through software has not guaranteed perfect result. Contact has been taken with HYTEC in order to see if possible fix can be applied to the interface. First revision of the PC-card PAL has been tested but did not show improvement. CVS version of the hyt1331.c until 1.2 contains "fast readout cycle" and should not be trusted. CVS 1.3 driver revision contains a patch to this problem. In the mean time you can apply your own patch (see Midas FAQs red box, see also Hytec)

hyt1331.c Version >= 1.8.3 This version has been modified for 5331 PCI card support running under the dio task.

khyt1331.c Version >= 1.8.3 A full Linux driver is available for the 5331 PCI card interfacing to the hyt1331. The kernel driver has been written for the Linux kernel 2.4.2, which comes with RedHat 7.1. It could be ported back to the 2.2.x kernel because no special feature of 2.4.x are used, although many data structures and function parameters have changed between 2.2 and 2.4, which makes the porting a bit painful. The driver supports only one 5331 card with up to four CAMAC crates.

kcs292x.c The 2926 is an 8 bit ISA board, while the 2927 is a 16bit ISA board. An equivalent PCI interface (2915) exists but is not yet supported by Midas (See KCS). No support for Windowx yet.

Both cards can be used also through a proper Linux driver *camaclx.c*. This requires to first load a module *camac-kcs292x.o*. This software is available but not part of the Midas distribution yet. Please contact midas@triumf.ca for further information.

wecc32.c The CAMAC crate controller CC32 interface to a PCI card... you will need the proper Linux module... Currently under test. WindowsNT and W95 drivers available but not implemented under Midas. (see CC32)

dsp004.c The dsp004 is an 8 bit ISA board PC interface which connect to the PC6002 CAMAC crate controller. This module is not being manufactured anymore, but somehow several labs still have that controller in use.

ces8210.c The CAMAC crate controller CBD8210 interface is a VME module to give access up to 7 CAMAC crate. This driver can be used only under VxWorks OS. (see CBD8210)

9.2 VME drivers

So far the PC-VME interface supported by Midas is ...

wevmemm.c I've got the board, I'm going to test later this year 2000 (see Wiener).

bt617.c Routines for accessing VME over SBS Bit3 Model 617 interface under Windows NT using the NT device driver Model 983 and under Linux using the vmehb device driver. The VME calls are implemented for the "mvmestd" Midas VME Standard. (see Bit3).

9.3 GPIB drivers

There is no specific GPIB driver part of the Midas package. But GPIB is used at Triumf under WindowsNT for several Slow Control frontends. The basic GPIB DLL library is provided by National Instrument. Please contact midas@triumf.ca for further information.

For GPIB Linux support please refer to The Linux Lab Project

10 appendix C: CAMAC and VME access function call

Names

Midas defines its own set of CAMAC and VME calls in order to unify the different hardware modules that it supports. This interface method permits to be totally hardware as well as OS **independent**. The same user code developed on a system can be used as a template for another application on a different operating system.

While the file **mcstd.h/c** (Midas Camac Standard) provides the interface for the CAMAC access, the file **mvmestc.h/c** (Midas VME Standard) is for the VME access.

An extra CAMAC interface built on the top of **mcstd** provides the ESONE standard CAMAC calls **esone.h/c**.

10.1 Midas CAMAC standard functions

*exportable midas CAMAC functions [***mcstd.h***]*

Names

10.1.1

EXTERNAL INLINE void EXPRT **cam16i** (const int c, const int n, const int a, const int f, WORD* d)

16 bits read.

cam16i() 16 bits input.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (0..7)
	`d`	data read out data

10.1.2

EXTERNAL INLINE void EXPRT **cam24i** (const int c, const int n, const int a, const int f, DWORD* d)

24 bits read.

cam24i() 24 bits input.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (0..7)
	`d`	data read out data

10.1.3

EXTERNAL INLINE void EXPRT **cam8i_q** (const int c, const int n, const int a, const int f, BYTE* d, int* x, int* q)

8 bits read with X, Q response.

cam8i_q() 8 bits input with Q response.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (0..7)
	`d`	data read out data
	`x`	X response (0:failed,1:success)
	`q`	Q resonpse (0:no Q, 1: Q)

10.1.4

EXTERNAL INLINE void EXPRT **cam16i_q** (const int c, const int n, const int a, const int f, WORD* d, int* x, int* q)

16 bits read with X, Q response.

cam16i_q() 16 bits input with Q response.

Return Value:	void	
Parameters:	c	crate number (0..)
	n	station number (0..30)
	a	sub-address (0..15)
	f	function (0..7)
	d	data read out data
	x	X response (0:failed,1:success)
	q	Q resonpse (0:no Q, 1: Q)

10.1.5

EXTERNAL INLINE void EXPRT **cam24i_q** (const int c, const int n, const int a, const int f, DWORD* d, int* x, int* q)

24 bits read with X, Q response.

cam24i_q() 24 bits input with Q response.

Return Value:	void	
Parameters:	c	crate number (0..)
	n	station number (0..30)
	a	sub-address (0..15)
	f	function (0..7)
	d	data read out data
	x	X response (0:failed,1:success)
	q	Q resonpse (0:no Q, 1: Q)

10.1.6

EXTERNAL INLINE void EXPRT **cam16i_r** (const int c, const int n, const int a, const int f, WORD** d, const int r)

Repeat 16 bits read r times.

cam16i_r() Repeat 16 bits input.

Return Value:	void	
Parameters:	c	crate number (0..)
	n	station number (0..30)
	a	sub-address (0..15)
	f	function (0..7)
	d	data read out data
	r	repeat time

10.1.7

EXTERNAL INLINE void EXPRT **cam24i_r** (const int c, const int n, const int a, const int f, DWORD** d, const int r)

Repeat 24 bits read r times.

cam24i_r() Repeat 24 bits input.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (0..7)
	`d`	data read out
	`r`	repeat time

10.1.8

EXTERNAL INLINE void EXPRT **cam8i_rq** (const int c, const int n, const int a, const int f, BYTE** d, const int r)

Repeat 16 bits read r times while Q.

cam8i_rq() Repeat 8 bits input with Q stop.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (0..7)
	`d`	pointer to data read out
	`r`	repeat time

10.1.9

EXTERNAL INLINE void EXPRT **cam16i_rq** (const int c, const int n, const int a, const int f, WORD** d, const int r)

Repeat 16 bits read r times while Q.

cam16i_rq() Repeat 16 bits input with Q stop.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (0..7)
	`d`	pointer to data read out
	`r`	repeat time

10.1.10

EXTERNAL INLINE void EXPRT **cam24i_rq** (const int c, const int n, const int a, const int f, DWORD** d, const int r)

Repeat 24 bits read r times while Q.

cam24i_rq Repeat 24 bits input with Q stop.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (0..7)
	`d`	pointer to data read out
	`r`	repeat time

10.1.11

EXTERNAL INLINE void EXPRT **cam8i_sa** (const int c, const int n, const int a, const int f, BYTE** d, const int r)

Scan read sub-address (8 bit).

cam8i_sa Read the given CAMAC address and increment the sub-address by one. Repeat r times.
Examples:

```
    BYTE pbkdat[4];
    cam8i_sa(crate, 5, 0, 2, &pbkdat, 4);
```

equivalent to :

```
    cam8i(crate, 5, 0, 2, &pbkdat[0]);
    cam8i(crate, 5, 1, 2, &pbkdat[1]);
    cam8i(crate, 5, 2, 2, &pbkdat[2]);
    cam8i(crate, 5, 3, 2, &pbkdat[3]);
```

Return Value: `void`

Parameters:

`c`	crate number (0..)
`n`	station number (0..30)
`a`	sub-address (0..15)
`f`	function (0..7)
`d`	pointer to data read out
`r`	number of consecutive sub-address to read

10.1.12

EXTERNAL INLINE void EXPRT **cam16i_sa** (const int c, const int n, const int a, const int f, WORD** d, const int r)

Scan read sub-address (16 bit).

cam16i_sa Read the given CAMAC address and increment the sub-address by one. Repeat r times.
Examples:

```
    WORD pbkdat[4];
    cam16i_sa(crate, 5, 0, 2, &pbkdat, 4);
```

equivalent to :

```
    cam16i(crate, 5, 0, 2, &pbkdat[0]);
    cam16i(crate, 5, 1, 2, &pbkdat[1]);
    cam16i(crate, 5, 2, 2, &pbkdat[2]);
    cam16i(crate, 5, 3, 2, &pbkdat[3]);
```

Return Value: `void`

Parameters:
`c` crate number (0..)
`n` station number (0..30)
`a` sub-address (0..15)
`f` function (0..7)
`d` pointer to data read out
`r` number of consecutive sub-address to read

10.1.13

EXTERNAL INLINE void EXPRT **cam24i_sa** (const int c, const int n,
const int a, const int f,
DWORD** d, const int r)

Scan read sub-address (24 bit).

cam24i_sa() Read the given CAMAC address and increment the sub-address by one. Repeat r times.
Examples:

```
DWORD pbkdat[8];
cam24i_sa(crate, 5, 0, 2, &pbkdat, 8);
```

equivalent to :

```
cam24i(crate, 5, 0, 2, &pbkdat[0]);
cam24i(crate, 6, 0, 2, &pbkdat[1]);
cam24i(crate, 7, 0, 2, &pbkdat[2]);
cam24i(crate, 8, 0, 2, &pbkdat[3]);
```

Return Value: `void`
Parameters:
`c` crate number (0..)
`n` station number (0..30)
`a` sub-address (0..15)
`f` function (0..7)
`d` pointer to data read out
`r` number of consecutive sub-address to read

10.1.14

EXTERNAL INLINE void EXPRT **cam8i_sn** (const int c, const int n,
const int a, const int f,
BYTE** d, const int r)

Scan read station (8 bit).

cam8i_sn() Read the given CAMAC address and increment the station number by one. Repeat r times.
Examples:

```
BYTE pbkdat[4];
cam8i_sa(crate, 5, 0, 2, &pbkdat, 4);
```

equivalent to :

```
cam8i(crate, 5, 0, 2, &pbkdat[0]);
cam8i(crate, 6, 0, 2, &pbkdat[1]);
cam8i(crate, 7, 0, 2, &pbkdat[2]);
cam8i(crate, 8, 0, 2, &pbkdat[3]);
```

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (0..7)
	`d`	pointer to data read out
	`r`	number of consecutive station to read

10.1.15

EXTERNAL INLINE void EXPRT **cam16i_sn** (const int c, const int n, const int a, const int f, WORD** d, const int r)

Scan read station (16 bit).

cam16i_sn() Read the given CAMAC address and increment the station number by one. Repeat r times.
Examples:

```
WORD pbkdat[4];
cam16i_sa(crate, 5, 0, 2, &pbkdat, 4);
```

equivalent to :

```
cam16i(crate, 5, 0, 2, &pbkdat[0]);
cam16i(crate, 6, 0, 2, &pbkdat[1]);
cam16i(crate, 7, 0, 2, &pbkdat[2]);
cam16i(crate, 8, 0, 2, &pbkdat[3]);
```

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (0..7)
	`d`	pointer to data read out
	`r`	number of consecutive station to read

10.1.16

EXTERNAL INLINE void EXPRT **cam24i_sn** (const int c, const int n, const int a, const int f, DWORD** d, const int r)

Scan read station (24 bit).

cam24i_sn() Read the given CAMAC address and increment the station number by one. Repeat r times.
Examples:

```
DWORD pbkdat[4];
cam24i_sa(crate, 5, 0, 2, &pbkdat, 4);
```

equivalent to :

```
cam24i(crate, 5, 0, 2, &pbkdat[0]);
cam24i(crate, 6, 0, 2, &pbkdat[1]);
cam24i(crate, 7, 0, 2, &pbkdat[2]);
cam24i(crate, 8, 0, 2, &pbkdat[3]);
```

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (0..7)
	`d`	pointer to data read out
	`r`	number of consecutive station to read

10.1.17

EXTERNAL INLINE void EXPRT **cami** (const int c, const int n, const int a, const int f, WORD* d)

Same as cam16i()

Same as cam16i()

10.1.18

EXTERNAL INLINE void EXPRT **cam8o** (const int c, const int n, const int a, const int f, BYTE d)

Write 8 bits data.

cam8o() Write data to given CAMAC address.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (16..31)
	`d`	data to be written to CAMAC

10.1.19

EXTERNAL INLINE void EXPRT **cam16o** (const int c, const int n, const int a, const int f, WORD d)

Write 16 bits data.

cam16o() Write data to given CAMAC address.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (16..31)
	`d`	data to be written to CAMAC

10.1.20

EXTERNAL INLINE void EXPRT **cam24o** (const int c, const int n, const int a, const int f, DWORD d)

Write 24 bits data.

cam24o() Write data to given CAMAC address.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (16..31)
	`d`	data to be written to CAMAC

10.1.21

EXTERNAL INLINE void EXPRT **cam8o_q** (const int c, const int n, const int a, const int f, BYTE d, int* x, int* q)

Write 8 bits data with Q.

cam8o_q() Write data to given CAMAC address with Q response.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (16..31)
	`d`	data to be written to CAMAC
	`x`	X response (0:failed,1:success)
	`q`	Q resonpse (0:no Q, 1: Q)

10.1.22

EXTERNAL INLINE void EXPRT **cam16o_q** (const int c, const int n, const int a, const int f, WORD d, int* x, int* q)

Write 16 bits data with Q.

cam16o_q() Write data to given CAMAC address with Q response.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (16..31)
	`d`	data to be written to CAMAC
	`x`	X response (0:failed,1:success)
	`q`	Q resonpse (0:no Q, 1: Q)

10.1.23

EXTERNAL INLINE void EXPRT **cam24o_q** (const int c, const int n, const int a, const int f, DWORD d, int* x, int* q)

Write 24 bits data with Q.

cam24o_q() Write data to given CAMAC address with Q response.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (16..31)
	`d`	data to be written to CAMAC
	`x`	X response (0:failed,1:success)
	`q`	Q response (0:no Q, 1: Q)

10.1.24

EXTERNAL INLINE void EXPRT **cam8o_r** (const int c, const int n, const int a, const int f, BYTE* d, const int r)

Repeat Write 8 bits data.

cam8o_r() Repeat write data to given CAMAC address r times.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (16..31)
	`d`	data to be written to CAMAC

10.1.25

EXTERNAL INLINE void EXPRT **cam16o_r** (const int c, const int n, const int a, const int f, WORD* d, const int r)

Repeat Write 16 bits data.

cam16o_r() Repeat write data to given CAMAC address r times.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (16..31)
	`d`	data to be written to CAMAC

10.1.26

EXTERNAL INLINE void EXPRT **cam24o_r** (const int c, const int n, const int a, const int f, DWORD* d, const int r)

Repeat Write 24 bits data.

cam24o_r() Repeat write data to given CAMAC address r times.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (16..31)
	`d`	data to be written to CAMAC

10.1.27

EXTERNAL INLINE void EXPRT **camo** (const int c, const int n, const int a, const int f, WORD d)

Same as cam16o()

Same as cam16o()

10.1.28

EXTERNAL INLINE int EXPRT **camc_chk** (const int c)

Crate presence check.

camc_chk() Crate presence check.

Return Value:	`0:Success,` -1:No CAMAC response	
Parameters:	`c`	crate number (0..)

10.1.29

EXTERNAL INLINE void EXPRT **camc** (const int c, const int n, const int a, const int f)

CAMAC command.

camc() CAMAC command (no data).

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (8..15, 24..31)

10.1.30

EXTERNAL INLINE void EXPRT **camc_q** (const int c, const int n, const int a, const int f, int* q)

CAMAC command with Q.

camc_q() CAMAC command with Q response (no data).

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (8..15, 24..31)
	`q`	Q response (0:no Q, 1:Q)

10.1.31

EXTERNAL INLINE void EXPRT **camc_sa** (const int c, const int n, const int a, const int f, const int r)

Scan command on sub-address.

camc_sa() Scan CAMAC command on sub-address.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (8..15, 24..31)
	`r`	number of consecutive sub-address to read

10.1.32

EXTERNAL INLINE void EXPRT **camc_sn** (const int c, const int n, const int a, const int f, const int r)

Scan command on station.

camc_sn() Scan CAMAC command on station.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`f`	function (8..15, 24..31)
	`r`	number of consecutive station to read

10.1.33

EXTERNAL INLINE int EXPRT **cam_init** (void)

CAMAC initilize.

cam_init() Initialize CAMAC for access.

Return Value:	`1:`	success

10.1.34

EXTERNAL INLINE void EXPRT **cam_exit** (void)

Close CAMAC.

cam_exit() Close CAMAC accesss.

10.1.35

EXTERNAL INLINE void EXPRT **cam_inhibit_set** (const int c)

Set Crate inhibit.

cam_inhibit_set() Set Crate inhibit.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)

10.1.36

EXTERNAL INLINE void EXPRT **cam_inhibit_clear** (const int c)

Clear Crate inhibit.

cam_inhibit_clear() Clear Crate inhibit.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)

10.1.37

EXTERNAL INLINE int EXPRT **cam_inhibit_test** (const int c)

Test Crate inhibit.

cam_inhibit_test() Test Crate Inhibit.

Return Value:	`1`	for set, 0 for cleared
Parameters:	`c`	crate number (0..)

10.1.38

EXTERNAL INLINE void EXPRT **cam_crate_clear** (const int c)

Clear Crate.

cam_crate_clear() Issue CLEAR to crate.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)

10.1.39

EXTERNAL INLINE void EXPRT **cam_crate_zinit** (const int c)

Z Crate.

cam_crate_zinit() Issue Z to crate.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)

10.1.40

EXTERNAL INLINE void EXPRT **cam_lam_enable** (const int c, const int n)

Enable LAM (Crate Controller).

cam_lam_enable() Enable LAM generation for given station to the Crate controller. It doesn't enable the LAM of the actual station itself.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	LAM station

10.1.41

EXTERNAL INLINE void EXPRT **cam_lam_disable** (const int c, const int n)

Disable LAM (Crate Controller).

cam_lam_disable() Disable LAM generation for given station to the Crate controller. It doesn't disable the LAM of the actual station itself.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	LAM station

10.1.42

EXTERNAL INLINE void EXPRT **cam_lam_read** (const int c, DWORD* lam)

Read LAM of crate controller.

cam_lam_read() Reads in lam the lam pattern of the entire crate.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`lam`	LAM pattern of the crate

10.1.43

EXTERNAL INLINE void EXPRT **cam_lam_clear** (const int c, const int n)

Clear LAM register (Crate Controller).

cam_lam_clear() Clear the LAM register of the crate controller. It doesn't clear the LAM of the particular station.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`lam`	LAM pattern of the crate
	`n`	LAM station

10.1.44

EXTERNAL INLINE int EXPRT **cam_lam_wait** (int* c, DWORD* n, const int millisec)

Wait for LAM.

cam_lam_wait() Wait for a LAM to occur with a certain timeout. Return crate and station if LAM occurs.

Return Value:	`1`	if LAM occured, 0 else
Parameters:	`c`	crate number (0..)
	`lam`	LAM pattern with a bit set for the station which generated the LAM
	`millisec`	If there is no LAM after this timeout, the routine returns

10.1.45

EXTERNAL INLINE void EXPRT **cam_interrupt_enable** (const int c)

Enable interrupts in crate controller.

cam_interrupt_enable() Enable interrupts in specific crate

Return Value:	`void`	
Parameters:	`c`	crate number (0..)

10.1.46

EXTERNAL INLINE void EXPRT **cam_interrupt_disable** (const int c)

Disable interrupts in crate controller.

cam_interrupt_disable() Disables interrupts in specific crate

Return Value:	`void`	
Parameters:	`c`	crate number (0..)

10.1.47

EXTERNAL INLINE int EXPRT **cam_interrupt_test** (const int c)

Test Crate Interrupt.

cam_interrupt_test() Test Crate Interrupt.

Return Value:	`1`	for set, 0 for cleared
Parameters:	`c`	crate number (0..)

10.1.48

EXTERNAL INLINE void EXPRT **cam_interrupt_attach** (const int c, const int n, void (*isr)(void))

Attach service routine.

cam_interrupt_attach() Attach service routine to LAM of specific crate and station.

Return Value:	`void`	
Parameters:	`c`	crate number (0..)
	`n`	station number

10.1.49

EXTERNAL INLINE void EXPRT **cam_interrupt_detach** (const int c, const int n)

Detach service routine.

cam_interrupt_detach() Detach service routine from LAM.

Return Value: `void`
Parameters:
`c` crate number (0..)
`n` station number

10.2 ESONE CAMAC standard functions

*exportable esone CAMAC functions [***esone.h, esone.c***]*

Names

Not all the functionality of ESONE standard have been fully tested.

10.2.1

INLINE void **ccinit** (void)

CAMAC initialization.

ccinit CAMAC initialization

CAMAC initialization must be called before any other ESONE subroutine call

Return Value: `void`

10.2.2

INLINE int **fccinit** (void)

CAMAC initialization.

fccinit CAMAC initialization with return status

fccinit can be called instead of ccinit to determine if the initialization was successful

Return Value:	`1`	for success, 0 for failure

10.2.3

INLINE void **cdreg** (int* ext, const int b, const int c, const int n, const int a)

External Address register.

cdreg Control Declaration REGister.

Compose an external address from BCNA for later use. Accessing CAMAC through ext could be faster if the external address is memory mapped to the processor (hardware dependent). Some CAMAC controller do not have this option see appendix D: Supported hardware. In this case

Return Value:	`void`	
Parameters:	`ext`	external address
	`b`	branch number (0..7)
	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)

10.2.4

INLINE void **cssa** (const int f, int ext, unsigned short* d, int* q)

16 bit function.

cssa Control Short Operation.

16 bit operation on a given external CAMAC address.

The range of the f is hardware dependent. The number indicated below are for standard ANSI/IEEE Std (758-1979)
Execute cam16i for f<8, cam16o for f>15, camc_q for (f>7 or f>23)

Return Value:	`void`	
Parameters:	`f`	function code (0..31)
	`ext`	external address
	`d`	data word
	`q`	Q response

10.2.5

INLINE void **cfsa** (const int f, const int ext, unsigned long* d, int* q)

24 bit function.

cfsa Control Full Operation.

24 bit operation on a given external CAMAC address.

The range of the f is hardware dependent. The number indicated below are for standard ANSI/IEEE Std (758-1979)
Execute cam24i for f<8, cam24o for f>15, camc_q for (f>7 or f>23)

Return Value:	`void`	
Parameters:	`f`	function code (0..31)
	`ext`	external address
	`d`	data long word
	`q`	Q response

10.2.6

INLINE void **cccc** (const int ext)

Crate Clear.

cccc Control Crate Clear.

Generate Crate Clear function. Execute cam_crate_clear()

Return Value:	`void`	
Parameters:	`ext`	external address

10.2.7

INLINE void **cccz** (const int ext)

Crate Z.

cccz Control Crate Z.

Generate Dataway Initialize. Execute cam_crate_zinit()

Return Value:	`void`	
Parameters:	`ext`	external address

10.2.8

INLINE void **ccci** (const int ext, int l)

Crate I.

ccci Control Crate I.

Set or Clear Dataway Inhibit, Execute cam_inhinit_set() /clear()

Return Value:	`void`	
Parameters:	`ext`	external address
	`l`	action l=0 -> Clear I, l=1 -> Set I

10.2.9

INLINE void **ctci** (const int ext, int* l)

Crate I.

ctci Test Crate I.

Test Crate Inhibit, Execute cam_inhibit_test()

Return Value:	`void`	
Parameters:	`ext`	external address
	`l`	action l=0 -> Clear I, l=1 -> Set I

10.2.10

INLINE void **cccd** (const int ext, int l)

Crate D.

cccd Control Crate D.

Enable or Disable Crate Demand.

Return Value:	`void`	
Parameters:	`ext`	external address
	`l`	action l=0 -> Clear D, l=1 -> Set D

10.2.11

INLINE void **ctcd** (const int ext, int* l)

Test Crate D.

ctcd Control Test Crate D.

Test Crate Demand.

Return Value:	`void`	
Parameters:	`ext`	external address
	`l`	D cleared -> l=0, D set -> l=1

10.2.12

INLINE void **cdlam** (int* lam, const int b, const int c, const int n, const int a, const int inta[2])

Declare LAM.

cdlam Control Declare LAM.

Declare LAM, Identical to cdreg.

Return Value:	`void`	
Parameters:	`lam`	external LAM address
	`b`	branch number (0..7)
	`c`	crate number (0..)
	`n`	station number (0..30)
	`a`	sub-address (0..15)
	`inta[2]`	implementation dependent

10.2.13

INLINE void **ctgl** (const int ext, int* l)

Test GL.

ctgl Control Test Demand Present.

Test the LAM register.

Return Value:	`void`	
Parameters:	`lam`	external LAM register address
	`l`	l !=0 if any LAM is set.

10.2.14

INLINE void **cclm** (const int lam, int l)

dis/enable LAM.

cclm Control Crate LAM.

Enable or Disable LAM. Execute F24 for disable, F26 for enable.

Return Value:	`void`	
Parameters:	`lam`	external address
	`l`	action l=0 -> disable LAM , l=1 -> enable LAM

10.2.15

INLINE void **cclnk** (const int lam, void (*isr)(void))

Link LAM to service procedure

cclnk Link LAM to service procedure

Link a specific service routine to a LAM. Since this routine is executed asynchronously, care must be taken on re-entrancy.

Return Value:	`void`	
Parameters:	`lam`	external address
	`isr`	name of service procedure

10.2.16

INLINE void **cculk** (const int lam)

Detach LAM from service procedure

cculk Unlink LAM from service procedure

Performs complementary operation to cclnk.

Return Value: `void`
Parameters: `lam` external address

10.2.17

INLINE void **ccrgl** (const int lam)

Re-enable LAM

ccrgl Relink LAM

Re-enable LAM in the controller

Return Value: `void`
Parameters: `lam` external address

10.2.18

INLINE void **cclc** (const int lam)

Clear LAM.

cclc Control Clear LAM. Clear the LAM of the station pointer by the lam address.

Return Value: `void`
Parameters: `lam` external address

10.2.19

INLINE void **ctlm** (const int lam, int* l)

Test LAM.

ctlm Test LAM.

Test the LAM of the station pointed by lam. Performs an F8

Return Value:	`void`	
Parameters:	`lam`	external address
	`l`	No LAM-> l=0, LAM present-> l=1

10.2.20

INLINE void **cfga** (int f[], int exta[], int intc[], int qa[], int cb[])

General external address scan function.

cfga Control Full (24bit) word General Action.

Return Value:	`void`	
Parameters:	`f`	function code
	`exta[]`	external address array
	`intc[]`	data array
	`qa[]`	Q response array
	`cb[]`	control block array
	cb[0] : number of function to perform	
	cb[1] : returned number of function performed	

10.2.21

INLINE void **csga** (int f[], int exta[], int intc[], int qa[], int cb[])

General external address scan function.

csga Control (16bit) word General Action.

Return Value:	`void`	
Parameters:	`f`	function code
	`exta[]`	external address array
	`intc[]`	data array
	`qa[]`	Q response array
	`cb[]`	control block array
	cb[0] : number of function to perform	
	cb[1] : returned number of function performed	

10.2.22

INLINE void **cfmad** (int f, int extb[], int intc[], int cb[])

Address scan function.

cfmad Control Full (24bit) Address Q scan.

Scan all sub-address while Q=1 from a0..a15 max from address extb[0] and store corresponding data in intc[]. If Q=0 while A<15 or A=15 then cross station boundary is applied (n-> n+1) and sub-address is reset (a=0). Perform action until either cb[0] action are performed or current external address exceeds extb[1].

implementation of cb[2] for LAM recognition is not implemented.

Return Value:	`void`	
Parameters:	`f`	function code
	`extb[]`	external address array
	extb[0] : first valid external address	
	extb[1] : last valid external address	
	`intc[]`	data array
	`qa[]`	Q response array
	`cb[]`	control block array
	cb[0] : number of function to perform	
	cb[1] : returned number of function performed	

10.2.23

INLINE void **csmad** (int f, int extb[], int intc[], int cb[])

Address scan function.

csmad Control (16bit) Address Q scan.

Scan all sub-address while Q=1 from a0..a15 max from address extb[0] and store corresponding data in intc[]. If Q=0 while A<15 or A=15 then cross station boundary is applied (n-> n+1) and sub-address is reset (a=0). Perform action until either cb[0] action are performed or current external address exceeds extb[1].

implementation of cb[2] for LAM recognition is not implemented.

Return Value:	`void`	
Parameters:	`f`	function code
	`extb[]`	external address array
	extb[0] : first valid external address	
	extb[1] : last valid external address	
	`intc[]`	data array
	`qa[]`	Q response array
	`cb[]`	control block array
	cb[0] : number of function to perform	
	cb[1] : returned number of function performed	

10.2.24

INLINE void **cfubc** (const int f, int ext, int intc[], int cb[])

Repeat function Q-stop.

cfubc Control Full (24bit) Block Repeat with Q-stop.

Execute function f on address ext with data intc[] while Q.

Return Value:	`void`	
Parameters:	`f`	function code
	`ext`	external address array
	`intc[]`	data array
	`cb[]`	control block array
	cb[0] : number of function to perform	
	cb[1] : returned number of function performed	

10.2.25

INLINE void **csubc** (const int f, int ext, int intc[], int cb[])

Repeat function Q-stop.

csubc Control (16bit) Block Repeat with Q-stop.

Execute function f on address ext with data intc[] while Q.

Return Value:	`void`	
Parameters:	`f`	function code
	`ext`	external address array
	`intc[]`	data array
	`cb[]`	control block array
	cb[0] : number of function to perform	
	cb[1] : returned number of function performed	

10.2.26

INLINE void **cfubr** (const int f, int ext, int intc[], int cb[])

Repeat function.

cfubr Repeat Mode Block Transfer (24bit).

Execute function f on address ext with data intc[] if Q. If noQ keep current intc[] data. Repeat cb[0] times.

Return Value:	`void`	
Parameters:	`f`	function code
	`ext`	external address array
	`intc[]`	data array
	`cb[]`	control block array
	cb[0] : number of function to perform	
	cb[1] : returned number of function performed	

10.2.27

INLINE void **csubr** (const int f, int ext, int intc[], int cb[])

Repeat function.

csubr Repeat Mode Block Transfer (16bit).

Execute function f on address ext with data intc[] if Q. If noQ keep current intc[] data. Repeat cb[0] times.

Return Value:	`void`	
Parameters:	`f`	function code
	`ext`	external address array
	`intc[]`	data array
	`cb[]`	control block array
	cb[0] : number of function to perform	
	cb[1] : returned number of function performed	

10.3 Midas VME standard functions

*exportable midas VME functions [***mvmestd.h***]*

This interface is brand new and not yet documented. But the for your information this code will implement basic VME access functions such as device open/close, map/unmap, read/write.

11 appendix D: Computer Busy Logic

A "computer busy logic" has to be implemented for a front-end to work properly. The reason for this is that some ADC modules can be re-triggered. If they receive more than one gate pulse before being read out, they accumulate the input charge that leads to wrong results. Therefore only one gate pulse should be sent to the ADC's, additional pulses must be blocked before the event is read out by the front-end. This operation is usually performed by a latch module, which is set by the trigger signal and reset by the computer after it has read out the event:

The output of this latch is shaped (limited in its pulse with to match the ADC gate width) and distributed to the ADC's. This scheme has two problems. The computer generates the reset signal, usually by two CAMAC output functions to a CAMAC IO unit. Therefore the duration of the pulse is a couple of ms. There is a non-negligible probability that during the reset pulse there is another hardware trigger. If this happens and both inputs of the latch are active, its function is undefined. Usually it generates several output pulses that lead to wrong ADC values. The second problem lies in the fact that the latch can be just reset when a trigger input is active. This can happen since trigger signals usually have a width of a few tens of nanoseconds. In this case the latch output signal does not carry the timing of the trigger signal, but the timing of the reset signal. The wrong timing of the output can lead to false ADC and TDC signals. To overcome this problem, a more elaborate scheme is necessary. One possible solution is the use of a latch module with edge-sensitive input and veto input. At PSI, the module "D. TRIGGER / DT102" can be used. The veto input is also connected to the computer:

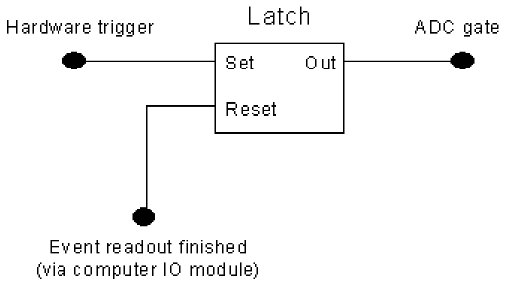


Figure 34: *Latched trigger layout.*

To reset this latch, following bit sequence is applied to the computer output (signals are displayed active low):

The active veto signal during the reset pulse avoids that the latch can receive a "set" and a "reset" simultaneously. The edge sensitive input ensures that the latch can only trigger on a leading edge of a trigger signal, not on the removing of the veto signal. This ensures that the timing of the trigger is always carried at the ADC/TDC gate signal.

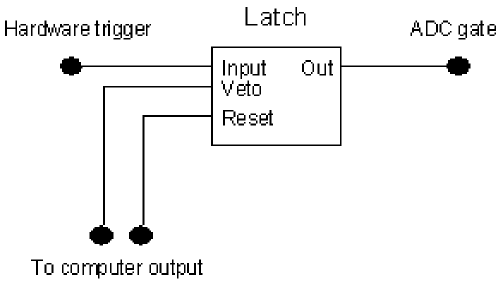


Figure 35: *Improved Latched trigger layout.*

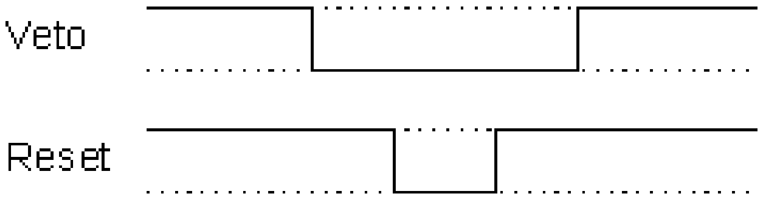


Figure 36: *Veto Timing.*

12 appendix E: Midas libraries

Programming information, environment, macros, functions.

Names

The midas libraries are composed of function calls subdivided into several main categories:

- *bk_xxx(...)* Midas bank manipulation
- *bm_xxx(...)* buffer management calls
- *cm_xxx(...)* common system calls
- *db_xxx(...)* database managemnt calls
- *el_xxx(...)* Electronic Log calls
- *hs_xxx(...)* History manipulation calls
- *ss_xxx(...)* system calls
- *ybk_xxx(...)* YBOS bank manipulation

12.1 Environment variables

Midas uses a couple of environment variables to facilitate application startup.

MIDAS_EXPTAB This variable specify the location of the **exptab** file containing the predefined midas experiment. The default location are: OS_UNIX: /etc, / , OS_WINNT: \system32, \system.

MIDAS_SERVER_HOST This variable predefines the names of the host on which the Midas experiment shared memories are residing. It is needed when connection to a remote experiment is requested. This variable is valid for Unix as well as Windows OS.

MIDAS_EXPT_NAME This variable predefines the name of the experiment to connect by default. It prevents the requested application to ask for the experiment name when multiple experiments are available on the host or to add the -e <expt_name> argument to the application command. This variable is valid for Unix as well as Windows OS.

MIDAS_DIR This variable predefines the LOCAL directory path where the shared memories for the experiment are located. It supersede the host_name and the expt_name as well as the MIDAS_SERVER_HOST and MIDAS_EXPT_NAME as a given directory path can only refer to a single experiment.

MCHART_DIR This variable is ... for later... This variable is valid only under Linux as the -D is not supported under WindowsXX

12.2 State Codes

These number will be apparent in the ODB under the "**/Runinfo/State**".

- 1 *STATE_STOPPED*
- 2 *STATE_PAUSED*
- 3 *STATE_RUNNING*

12.3 Transition Codes

These number will be apparent in the ODB under the **"/Runinfo/Requested transition"**

- 1 *TR_START*
- 2 *TR_STOP*
- 4 *TR_PAUSE*
- 8 *TR_RESUME*

12.4 Midas Data Types

Midas defined its own data type for OS compatibility. It is suggested to use them in order to insure a proper compilation when moving code from one OS to another. *float* and *double* retain OS definition.

- **BYTE** unsigned char
- **WORD** unsigned short int (16bits word)
- **DWORD** unsigned 32bits word
- **INT** signed 32bits word
- **BOOL** OS dependent.

When defining a data type either in the frontend code for bank definition or in user code to define ODB variables, Midas requires the use of its own data type declaration. The list below shows the main Type IDentification to be used (refers to midas.h for complete listing):

- **TID_BYTE** unsigned byte 0 255
- **TID_SBYTE** signed BYTE -128 127
- **TID_CHAR** single character 0 255
- **TID_WORD** two BYTE 0 65535
- **TID_SHORT** signed WORD -32768 32767
- **TID_DWORD** four bytes 0 2**32-1
- **TID_INT** signed DWORD -2**31 2**31-1
- **TID_BOOL** four bytes bool 0 1
- **TID_FLOAT** four bytes float format

- **TID_DOUBLE** eight bytes float format

12.5 Midas bank examples

There are several examples under the Midas source code that you can check. Please have a look at

- Frontend code *midas/examples/experiment/frontend.c* etc...
- Backend code *midas/examples/experiment/analyzer.c* etc...

12.6 YBOS Bank Types

YBOS defines several type but all types should be 4 bytes aligned. Distinction of signed and unsigned is not done. When mixing MIDAS and YBOS in the frontend for RO_ODB see The Equipment structure make sure the bank types are compatible.

- **I1_BKTYPE** Bank of Bytes
- **I2_BKTYPE** Bank of 2 bytes data
- **I4_BKTYPE** Bank of 4 bytes data
- **F4_BKTYPE** Bank of float data
- **D8_BKTYPE** Bank of double data
- **A1_BKTYPE** Bank of ASCII char

12.7 YBOS bank examples

Basic examples using YBOS banks are available in the midas tree under examples/ybosexpt.

Frontend code Example 1, 2 shows the bank creation with some CAMAC acquisition.

```
-------- example 1 -------- Simple 16 bits bank construction

void read_cft (DWORD *pevent)
{
  DWORD *pbkdat, slot;

  ybk_create((DWORD *)pevent, "TDCP", I2_BKTYPE, &pbkdat);
  for (slot=FIRST_CFT;slot<=LAST_CFT;slot++)
    {
      cami(3,slot,1,6,(WORD *)pbkdat);
      ((WORD *)pbkdat)++;
      cam16i_rq(3,slot,0,4,(WORD **)&pbkdat,16);
    }
  ybk_close((DWORD *)pevent, I2_BKTYPE, pbkdat);
  return;
}
-------- example 2 -------- Simple 32bit bank construction
{
  DWORD *pbkdat;

  ybk_create((DWORD *)pevent, "TICS", I4_BKTYPE, &pbkdat);
  camo(2,22,0,17,ZERO);
  cam24i_r(2,22,0,0,(DWORD **) &pbkdat,10);
  cam24i_r(2,22,0,0,(DWORD **) &pbkdat,10);
  cam24i_r(2,22,0,0,(DWORD **) &pbkdat,10);
  cam24i_r(2,22,0,0,(DWORD **) &pbkdat,10);
  cam24i_r(2,22,0,0,(DWORD **) &pbkdat,9);
  ybk_close((DWORD *)pevent, I4_BKTYPE, pbkdat);
  return 0;
}
```

Example 3 shows a creation of an EVID bank containg a duplicate of the midas header. As the Midas header is stripped out of the event when data are logger, it is necessary to compose such event to retain event information for off-line analysis. Uses of predefined macros (see Midas Library) are available in order to extract from a pre-composed Midas event the internal header fields i.e. Event ID, Trigger mask, Serial number, Time stamp. In this EVID bank we added the current run number which is retrieve by the frontend at the begin of a run.

```
-------- example 3 -------- Full equipment readout function

INT read_cum_scaler_event(char *pevent, INT off)
{
  INT i;
```

```
       DWORD *pbkdat, *pbktop, *podbvar;

       ybk_init((DWORD *) pevent);

       // collect user hardware SCALER data
       ybk_create((DWORD *)pevent, "EVID", I4_BKTYPE, (DWORD *)(&pbkdat));
       *(pbkdat)++ = gbl_tgt_counter++;                      // event counter
       *((WORD *)pbkdat) = EVENT_ID(pevent);     ((WORD *)pbkdat)++;
       *((WORD *)pbkdat) = TRIGGER_MASK(pevent); ((WORD *)pbkdat)++;
       *(pbkdat)++ = SERIAL_NUMBER(pevent);
       *(pbkdat)++ = TIME_STAMP(pevent);
       *(pbkdat)++ = gbl_run_number;                         // run number
       ybk_close((DWORD *)pevent, pbkdat);

       // BEGIN OF CUMULATIVE SCALER EVENT
       ybk_create((DWORD *)pevent, "CUSC", I4_BKTYPE, (DWORD *)(&pbkdat));
       for (i=0 ; i<NSCALERS ; i++){
         *pbkdat++ = scaler[i].cuval[0];
         *pbkdat++ = scaler[i].cuval[1];
       }

       ybk_close(DWORD *)pevent, I4_BKTYPE, pbkdat);
       // END OF CUMULATIVE SCALER EVENT

       // event in bytes for Midas
       return (ybk_size ((DWORD *)pevent));
     }
```

Backend code If the data logging is done through YBOS format (see ODB /Logger Tree Format) the events on the storage media will have been stripped from the MIDAS header used for transfering the event from the frontend to the backend. This means the logger data format is a "TRUE" YBOS format. Uses of standard YBOS library is then possible.

```
--- Example of YBOS bank extraction ----

void process_event(HNDLE hBuf, HNDLE request_id, EVENT_HEADER *pheader, void *pevent)
{
    INT status;
    DWORD *plrl, *pybk, *pdata, bklen, bktyp;
    char  banklist[YB_STRING_BANKLIST_MAX];

    // pointer to data section
    plrl = (DWORD *)       pevent;

    // Swap event
    yb_any_event_swap(FORMAT_YBOS,plrl);

    // bank name given through argument list
    if ((status = ybk_find (plrl, sbank_name, &bklen, &bktyp, (void *)&pybk)) == YB_SUCCESS)
    {
      // given bank found in list
      status = ybk_list (plrl, banklist);
      printf("#banks:%i Bank list:-%s-\n",status,banklist);
      printf("Bank:%s - Length (I*4):%i - Type:%i - pBk:0x%p\n",sbank_name, bklen, bktyp, pybk);
```

```
   // check id EVID found in event for id and msk selection
   if ((status = ybk_find (plrl, "EVID", &bklen, &bktyp, (void *)&pybk)) == YB_SUCCESS)
   {
    pdata = (DWORD *)((YBOS_BANK_HEADER *)pybk + 1);
...
   }

  // iterate through the event
  pybk = NULL;
  while ((bklen = ybk_iterate(plrl, &pybk, (void *)&pdata))
         && (pybk != NULL))
   printf("bank length in 4 bytes unit: %d\n",bklen);

  }
  else
  {
   status = ybk_list (plrl, banklist);
   printf("Bank -%s- not found (%i) in ",sbank_name, status);
   printf("#banks:%i Bank list:-%s-\n",status,banklist);
  }
  ...
  ... ...
}
```

12.8

Midas Library

exportable midas functions through inclusion of midas.h

Names

12.8.1

#define **LAM_SOURCE** (c, s)

MACRO Code LAM register with crate and station.

LAM_SOURCE Code the LAM crate and LAM station into a bitwise register.

Parameters:		
	c	Crate number
	s	Slot number

12.8.2

#define **LAM_STATION** (s)

MACRO Code LAM Station.

LAM_STATION Code the Station number bitwise for the LAM source.

Parameters:		
	s	Slot number

12.8.3

#define **LAM_SOURCE_CRATE** (c)

MACRO Convert coded Crate.

LAM_SOURCE_CRATE Convert the coded LAM crate to Crate number.

Parameters: c coded crate

12.8.4

#define **LAM_SOURCE_STATION** (s)

MACRO Convert coded Sattion.

LAM_SOURCE_STATION Convert the coded LAM station to Station number.

Parameters: s Slot number

12.8.5

#define **TRIGGER_MASK** (e)

MACRO Trigger mask.

TRIGGER_MASK Extract or set the trigger mask field pointed by the argument.

Parameters: e pointer to the midas event (pevent)

12.8.6

#define **EVENT_ID** (e)

MACRO event ID.

EVENT_ID Extract or set the event ID field pointed by the argument.

Parameters: e pointer to the midas event (pevent)

12.8.7

#define **SERIAL_NUMBER** (e)

MACRO serial number.

SERIAL_NUMBER Extract or set/reset the serial number field pointed by the argument.

Parameters: `e` pointer to the midas event (pevent)

12.8.8

#define **TIME_STAMP** (e)

MACRO Time stamp.

TIME_STAMP Extract or set/reset the time stamp field pointed by the argument.

Parameters: `e` pointer to the midas event (pevent)

12.8.9

INT **cm_set_msg_print()** (INT system_mask, INT user_mask, int (*func)(const char*))

Set message

Description: Set message masks. When a message is generated by calling cm_msg, it can got to two destinatinons. First a user defined callback routine and second to the "SYSMSG" buffer. A user defined callback receives all messages which satisfy the user_mask.

Remarks:

Example:

```
int message_print(const char *msg)
{
  char str[160];

  memset(str, ' ', 159);
  str[159] = 0;
  if (msg[0] == '[')
    msg = strchr(msg, ']')+2;
  memcpy(str, msg, strlen(msg));
```

```
    ss_printf(0, 20, str);
    return 0;
  }

  ...
    cm_set_msg_print(MT_ALL, MT_ALL, message_print);
  ...
```

Return Value:	`CM_SUCCESS`	
Parameters:	`system_mask`	Bit masks for MERROR, MINFO etc. to send system messages.
	`user_mask`	Bit masks for MERROR, MINFO etc. to send messages to the user callback.
	`func`	Function which receives all printout. By setting "puts",messages are just printed to the screen.

12.8.10

INT **cm_msg()** (INT message_type, char* filename, INT line, const char* routine, const char* format, ...)

Returns MIDAS environment variables.

Description: This routine can be called whenever an internal error occurs or an informative message is produced. Different message types can be enabled or disabled by setting the type bits via cm_set_msg_print.

Remarks: Do not add the "

n" escape carriage control at the end of the formated line as it is already added by the client on the receiving side.

Example:

```
...
  cm_msg(MINFO, "my program", "This is a information message only);
  cm_msg(MERROR, "my program", "This is an error message with status:%d", my_status);
  cm_msg(MTALK, "my_program", My program is Done!");
  ...
```

Return Value:	`CM_SUCCESS`	
Parameters:	`message_type`	See Message Macros.
	`filename`	Name of source file where error occured
	`line`	Line number where error occured
	`routine`	Routine name.
	`format`	message to printout, ... Parameters like for printf()

12.8.11

INT **cm_msg1()** (INT message_type, char* filename, INT line, const char* facility, const char* routine, const char* format, ...)

Redirect messages to a prive log file.

Description: This routine is similar to cm_msg(). It differs from cm_msg() only by the logging destination being a file given through the argument list i.e:**facility**.

Remarks: Do not add the "
n" escape carriage control at the end of the formated line as it is already added by the client on the receiving side.

Example: The first arg in the following example uses the predefined macro MINFO which handles automatically the first 3 arguments of the function (see Message Macros.

```
    ...
    cm_msg1(MINFO, "my_log_file", "my_program"," My message status:%d", status);
    ...
//----- File my_log_file.log
Thu Nov  8 17:59:28 2001 [my_program] My message status:1
```

Return Value:	`CM_SUCCESS`	
Parameters:	`message_type`	See Message Macros.
	`filename`	Name of source file where error occured
	`line`	Line number where error occured
	`facility`	Logging file name
	`routine`	Routine name
	`format`	message to printout, ... Parameters like for printf()

12.8.12

INT **cm_msg_register()** (void (*func)(HNDLE, HNDLE, EVENT_HEADER*, void*))

Register a message dispatch function.

Description: Register a dispatch function for receiving system messages.

Remarks:

Example: Excerpt from mlxspeaker.c

```
//----- receive_message
void receive_message(HNDLE hBuf, HNDLE id, EVENT_HEADER *header, void *message)
{
  char str[256], *pc, *sp;
  // print message
  printf("%s\n", (char *)(message));

  printf("evID:%x Mask:%x Serial:%i Size:%d\n"
                ,header->event_id
                ,header->trigger_mask
                ,header->serial_number
                ,header->data_size);
  pc = strchr((char *)(message),']')+2;
  ...
  // skip none talking message
  if (header->trigger_mask == MT_TALK ||
      header->trigger_mask == MT_USER)
   ...
}

int main(int argc, char *argv[])
{
  ...

  // now connect to server
  status = cm_connect_experiment(host_name, exp_name, "Speaker", NULL);
  if (status != CM_SUCCESS)
    return 1;

  // Register callback for messages
  cm_msg_register(receive_message);
  ...
}
```

Return Value: `CM_SUCCESS` or bm_open_buffer and bm_request_event return status

Parameters: `func` Dispatch function.

12.8.13

INT **cm_get_environment()** (char* host_name, char* exp_name)

Returns MIDAS environment variables.

Description: Returns MIDAS environment variables.

Remarks: This function can be used to evaluate the standard MIDAS environment variables before connecting to an experiment (see Environment variables). The usual way is that the host name and experiment name are first derived from the environment variables MIDAS_SERVER_HOST and MIDAS_EXPT_NAME. They can then be superseded by command line parameters with -h and -e flags.

Example:

```
#include <stdio.h>
#include <midas.h>
main(int argc, char *argv[])
{
INT  status, i;
char host_name[256],exp_name[32];

 // get default values from environment
 cm_get_environment(host_name, exp_name);

 // parse command line parameters
 for (i=1 ; i<argc ; i++)
  {
  if (argv[i][0] == '-')
   {
   if (i+1 >= argc || argv[i+1][0] == '-')
    goto usage;
   if (argv[i][1] == 'e')
    strcpy(exp_name, argv[++i]);
   else if (argv[i][1] == 'h')
    strcpy(host_name, argv[++i]);
   else
    {
usage:
    printf("usage: test [-h Hostname] [-e Experiment]\n\n");
    return 1;
    }
   }
  }
 status = cm_connect_experiment(host_name, exp_name, "Test", NULL);
 if (status != CM_SUCCESS)
  return 1;
  ...do anyting...
 cm_disconnect_experiment();
}
```

Return Value:	CM_SUCCESS	
Parameters:	host_name	Contents of MIDAS_SERVER_HOST environment variable.
	exp_name	Contents of MIDAS_EXPT_NAME environment variable.

12.8.14

INT **cm_connect_experiment()** (char* host_name, char* exp_name, char* client_name, void (*func)(char*))

Connects to a MIDAS experiment.

Description: This function connects to an existing MIDAS experiment. This must be the first call in a MIDAS application. It opens three TCP connection to the remote host (one for RPC calls, one to send events and one for hot-link notifications from the remote host) and writes client information into the ODB under /System/Clients.

Remarks: All MIDAS applications should evaluate the MIDAS_SERVER_HOST and MIDAS_EXPT_NAME environment variables as defaults to the host name and experiment name (see Environment variables). For that purpose, the function cm_get_environment() should be called prior to cm_connect_experiment(). If command line parameters -h and -e are used, the evaluation should be done between cm_get_environment() and cm_connect_experiment(). The function cm_disconnect_experiment() must be called before a MIDAS application exits.

Example:

```
#include <stdio.h>
#include <midas.h>
main(int argc, char *argv[])
{
INT  status, i;
char host_name[256],exp_name[32];

  // get default values from environment
  cm_get_environment(host_name, exp_name);

  // parse command line parameters
  for (i=1 ; i<argc ; i++)
    {
    if (argv[i][0] == ’-’)
      {
      if (i+1 >= argc || argv[i+1][0] == ’-’)
        goto usage;
      if (argv[i][1] == ’e’)
        strcpy(exp_name, argv[++i]);
      else if (argv[i][1] == ’h’)
        strcpy(host_name, argv[++i]);
      else
        {
usage:
        printf("usage: test [-h Hostname] [-e Experiment]\n\n");
        return 1;
        }
      }
    }
  status = cm_connect_experiment(host_name, exp_name, "Test", NULL);
  if (status != CM_SUCCESS)
```

```
      return 1;
    ...do operations...
    cm_disconnect_experiment();
  }
```

Return Value: CM_SUCCESS,

CM_VERSION_MISMATCH MIDAS library version different on local and remote compu

Parameters:

`host_name`	Specifies host to connect to. Must be a valid IP host name.The string can be empty ("") if to connect to the local computer.
`exp_name`	Specifies the experiment to connect to.If this string is empty, the number of defined experiments in exptab is checked.If only one experiment is defined, the function automatically connects to thisone. If more than one experiment is defined, a list is presented and the usercan interactively select one experiment.
`client_name`	Client name of the calling program as it can be seen byothers (like the scl command in ODBEdit).
`func`	Callback function to read in a password if security hasbeen enabled. In all command line applications this function is NULL whichinvokes an internal ss_gets() function to read in a password.In windows environments (MS Windows, X Windows) a function can be supplied toopen a dialog box and read in the password. The argument of this function mustbe the returned password.

12.8.15

INT **cm_disconnect_experiment()** (void)

Disconnect from a MIDAS experiment.

Description: Disconnect from a MIDAS experiment.

Remarks: Should be the last call to a MIDAS library function in an application before it exits. This function removes the client information from the ODB, disconnects all TCP connections and frees all internal allocated memory. See cm_connect_experiment for example.

Return Value: CM_SUCCESS

12.8.16

INT **cm_get_experiment_database()** (HNDLE* hDB, HNDLE* hKeyClient)

Get the handle to the ODB

Description: Get the handle to the ODB from the currently connected experiment.

Remarks: This function returns the handle of the online database (ODB) which can be used in future db_xxx() calls. The hkeyclient key handle can be used to access the client information in the ODB. If the client key handle is not needed, the parameter can be NULL.

Example:
```
HNDLE hDB, hkeyclient;
char  name[32];
int   size;
db_get_experiment_database(&hdb, &hkeyclient);
size = sizeof(name);
db_get_value(hdb, hkeyclient, "Name", name, &size, TID_STRING);
printf("My name is %s\n", name);
```

Return Value:	`CM_SUCCESS`	
Parameters:	`hDB`	Database handle.
	`hKeyClient`	Handle for key where search starts, zero for root.
	`key_name`	Name of key to search, can contain directories.

12.8.17

INT **cm_register_transition** (INT transition, INT (*func)(INT, char*))

Registers a callback function for run transitions.

Description: Registers a callback function for run transitions.

Remarks: This function internally registers the transition callback function and publishes its request for transition notification by writing the transition bit to /System/Clients/<pid>/Transition Mask. Other clients making a transition scan the transition masks of all clients and call their transition callbacks via RPC.

Clients can register for transitions (Start/Stop/Pause/Resume) or for notifications before or after a transition occurs (Pre-start/Post-start/Pre-stop/Post-stop). The logger for example opens the logging files on pre-start and closes them on post-stop.

The callback function returns CM_SUCCESS if it can perform the transition or a value larger than one in case of error. An error string can be copied into the error variable.

Example: The callback function will be called on transitions from inside the cm_yield() function which therefore must be contained in the main program loop.

```
INT start(INT run_number, char *error)
{
  if (<not ok>)
    {
    strcpy(error, "Cannot start because ...");
    return 2;
    }
  printf("Starting run %d\n", run_number);
  return CM_SUCCESS;
}
main()
{
  ...
  cm_register_transition(TR_START, start);
  do
    {
    status = cm_yield(1000);
    } while (status != RPC_SHUTDOWN &&
             status != SS_ABORT);
  ...
}
```

Return Value:	`CM_SUCCESS`	
Parameters:	`transition`	Transition to register for. Can be TR_PRESTART, TR_START,TR_POSTSTART, TR_PRSTOP, TR_STOP, TR_POSTSTOP, TR_PAUSE or TR_RESUME.
	`func`	Callback function.

12.8.18

INT **bm_open_buffer()** (char* buffer_name, INT buffer_size, INT* buffer_handle)

open an event buffer.

Description: Open an event buffer.

Remarks: Two default buffers are created by the system. The "SYSTEM" buffer is used to exchange events and the "SYSMSG" buffer is used to exchange system messages. The name and size of the event buffers is defined in midas.h as EVENT_BUFFER_NAME and EVENT_BUFFER_SIZE. Following example opens the "SYSTEM" buffer, requests events with ID 1 and enters a main loop. Events are then received in process_event()

Example:

```
#include <stdio.h>
#include "midas.h"
void process_event(HNDLE hbuf, HNDLE request_id,
           EVENT_HEADER *pheader, void *pevent)
{
  printf("Received event #%d\r",
  pheader->serial_number);
}
main()
{
  INT status, request_id;
  HNDLE hbuf;
  status = cm_connect_experiment("pc810", "Sample", "Simple Analyzer", NULL);
  if (status != CM_SUCCESS)
  return 1;
  bm_open_buffer(EVENT_BUFFER_NAME, EVENT_BUFFER_SIZE, &hbuf);
  bm_request_event(hbuf, 1, TRIGGER_ALL, GET_ALL, request_id, process_event);

  do
  {
   status = cm_yield(1000);
  } while (status != RPC_SHUTDOWN && status != SS_ABORT);
  cm_disconnect_experiment();
  return 0;
}
```

Return Value:

BM_SUCCESS,
BM_NO_SHM Shared memory cannot be created
BM_NO_MUTEX Mutex cannot be created
BM_NO_MEMORY Not enough memory to create buffer descriptor
BM_MEMSIZE_MISMATCH Buffer size conflicts with an existing buffer ofdifferent size
BM_INVALID_PARAM Invalid parameter

Parameters:

`buffer_name`	Name of buffer
`buffer_size`	Size of buffer in bytes
`buffer_handle`	Buffer handle returned by function

12.8.19

INT **bm_close_buffer()** (INT buffer_handle)

close event buffer.

Description: Closes an event buffer previously opened with bm_open_buffer().

Return Value: BM_SUCCESS, BM_INVALID_HANDLE

Parameters:

`buffer_handle`	buffer handle

12.8.20

INT **bm_set_cache_size()** (INT buffer_handle, INT read_size, INT write_size)

Turns on/off caching for reading and writing to a buffer.

Description: Modifies buffer cache size.

Remarks: Without a buffer cache, events are copied to/from the shared memory event by event.

To protect processed from accessing the shared memory simultaneously, semaphores are used. Since semaphore operations are CPU consuming (typically 50-100us) this can slow down the data transfer especially for small events. By using a cache the number of semaphore operations is reduced dramatically. Instead writing directly to the shared memory, the events are copied to a local cache buffer. When this buffer is full, it is copied to the shared memory in one operation. The same technique can be used when receiving events.

The drawback of this method is that the events have to be copied twice, once to the cache and once from the cache to the shared memory. Therefore it can happen that the usage of a cache even slows down data throughput on a given environment (computer type, OS type, event size). The cache size has therefore be optimized manually to maximize data throughput.

Return Value:	`BM_SUCCESS,`	BM_INVALID_HANDLE, BM_NO_MEMORY, BM_INVALID_PARAM
Parameters:	`buffer_handle`	buffer handle obtained via bm_open_buffer()
	`read_size`	cache size for reading events in bytes, zero for no cache
	`write_size`	cache size for writing events in bytes, zero for no cache

12.8.21

INT **bm_compose_event()** (EVENT_HEADER* event_header, short int event_id, short int trigger_mask, DWORD size, DWORD serial)

compose the Midas event header.

Description: Compose a Midas event header.

Remarks: An event header can usually be set-up manually or through this routine. If the data size of the event is not known when the header is composed, it can be set later with event_header->data-size = <...> Following structure is created at the beginning of an event

```
typedef struct {
 short int     event_id;
 short int     trigger_mask;
 DWORD         serial_number;
 DWORD         time_stamp;
 DWORD         data_size;
} EVENT_HEADER;
```

Example:

```
      char event[1000];
 bm_compose_event((EVENT_HEADER *)event, 1, 0, 100, 1);
 *(event+sizeof(EVENT_HEADER)) = <...>
```

Return Value: `BM_SUCCESS`

Parameters:

`event_header`	pointer to the event header
`event_id`	event ID of the event
`trigger_mask`	trigger mask of the event
`size`	size if the data part of the event in bytes
`serial`	serial number

12.8.22

INT **bm_request_event()** (HNDLE buffer_handle, short int event_id, short int trigger_mask, INT sampling_type, HNDLE* request_id, void (*func)(HNDLE, HNDLE, EVENT_HEADER*, void*))

event request.

Description: Place an event request based on certain characteristics.

Remarks: Multiple event requests can be placed for each buffer, which are later identified by their request ID. They can contain different callback routines. Example see bm_open_buffer and bm_receive_event

Example:

Return Value: `BM_SUCCESS,`
BM_NO_MEMORY too many requests. The valueMAX_EVENT_REQUESTS in midas.h

Parameters:		
	`buffer_handle`	buffer handle obtained via bm_open_buffer()
	`event_id`	event ID for requested events. Use EVENTID_ALLto receive events with any ID.
	`trigger_mask`	trigger mask for requested events.The requested events must have at least one bit in itstrigger mask common with the requested trigger mask. Use TRIGGER_ALL toreceive events with any trigger mask.
	`sampling_type`	specifies how many events to receive.A value of GET_ALL receives all events whichmatch the specified event ID and trigger mask. If the events are consumed slowerthan produced, the producer is automatically slowed down. A value of GET_SOMEreceives as much events as possible without slowing down the producer. GET_ALL istypically used by the logger, while GET_SOME is typically used by analyzers.
	`request_id`	request ID returned by the function.This ID is passed to the callback routine and mustbe used in the bm_delete_request() routine.
	`func`	allback routine which gets called when an event of thespecified type is received.

12.8.23

INT **bm_delete_request()** (INT request_id)

delete event request.

Description: Deletes an event request previously done with bm_request_event().

Remarks: When an event request gets deleted, events of that requested type are not received any more. When a buffer is closed via bm_close_buffer(), all event requests from that buffer are deleted automatically

Example:

Return Value: `BM_SUCCESS,` BM_INVALID_HANDLE

Parameters: `request_id` request identifier given by bm_request_event()

12.8.24

INT **bm_send_event()** (INT buffer_handle, void* source, INT buf_size, INT async_flag) *send event to buffer.*

Description: Sends an event to a buffer.

Remarks: This function check if the buffer has enough space for the event, then copies the event to the buffer in shared memory. If clients have requests for the event, they are notified via an UDP packet.

Example:

```
char event[1000];
// create event with ID 1, trigger mask 0, size 100 bytes and serial number 1
bm_compose_event((EVENT_HEADER *) event, 1, 0, 100, 1);

// set first byte of event
*(event+sizeof(EVENT_HEADER)) = <...>
#include <stdio.h>
#include "midas.h"
main()
{
 INT status, i;
 HNDLE hbuf;
 char event[1000];
 status = cm_connect_experiment("", "Sample", "Producer", NULL);
 if (status != CM_SUCCESS)
 return 1;
 bm_open_buffer(EVENT_BUFFER_NAME, EVENT_BUFFER_SIZE, &hbuf);

 // create event with ID 1, trigger mask 0, size 100 bytes and serial number 1
 bm_compose_event((EVENT_HEADER *) event, 1, 0, 100, 1);

 // set event data
 for (i=0 ; i<100 ; i++)
 *(event+sizeof(EVENT_HEADER)+i) = i;
 // send event
 bm_send_event(hbuf, event, 100+sizeof(EVENT_HEADER), SYNC);
 cm_disconnect_experiment();
 return 0;
}
```

Return Value: `BM_SUCCESS,`

BM_ASYNC_RETURN Routine called with async_flag == TRUE andbuffer has not enou
BM_NO_MEMORY Event is too large for network buffer or event buffer.One has to incre

Parameters:

`buffer_handle`	Buffer handle obtained via bm_open_buffer()
`source`	Address of event buffer
`buf_size`	Size of event including event header in bytes
`async_flag`	Synchronous/asynchronous flag. If FALSE, the functionblocks if the buffer has not enough free space to receive the event.If TRUE, the function returns immediately with avalue of BM_ASYNC_RETURN without writing the event to the buffer

12.8.25

INT **bm_flush_cache()** (INT buffer_handle, INT async_flag)

empty write cache.

Description: Empty write cache.

Remarks: This function should be used if events in the write cache should be visible to the consumers immediately. It should be called at the end of each run, otherwise events could be kept in the write buffer and will flow to the data of the next run.

Example:

Return Value: `BM_SUCCESS,`
BM_ASYNC_RETURN Routine called with async_flag == TRUEand buffer has not enou
BM_NO_MEMORY Event is too large for network buffer or event buffer.One has to incre

Parameters:
`buffer_handle` Buffer handle obtained via bm_open_buffer()
`async_flag` Synchronous/asynchronous flag.If FALSE, the function blocks if the buffer has notenough free space to receive the full cache. If TRUE, the function returnsimmediately with a value of BM_ASYNC_RETURN without writing the cache.

12.8.26

INT **bm_receive_event()** (INT buffer_handle, void* destination, INT* buf_size, INT async_flag)

receive event from buffer.

Description: Receives events directly.

Remarks: This function is an alternative way to receive events without a main loop. It can be used in analysis systems which actively receive events, rather than using callbacks. A analysis package could for example contain its own command line interface. A command like "receive 1000 events" could make it necessary to call bm_receive_event() 1000 times in a row to receive these events and then return back to the command line prompt.

Example: The according bm_request_event() call contains NULL as the callback routine to indicate that bm_receive_event() is called to receive events.

```
#include <stdio.h>
#include "midas.h"
void process_event(EVENT_HEADER *pheader)
{
 printf("Received event #%d\r",
 pheader->serial_number);
}
main()
{
INT status, request_id;
HNDLE hbuf;
char event_buffer[1000];
status = cm_connect_experiment("", "Sample",
"Simple Analyzer", NULL);
if (status != CM_SUCCESS)
 return 1;
bm_open_buffer(EVENT_BUFFER_NAME, EVENT_BUFFER_SIZE, &hbuf);
bm_request_event(hbuf, 1, TRIGGER_ALL, GET_ALL, request_id, NULL);

do
{
 size = sizeof(event_buffer);
 status = bm_receive_event(hbuf, event_buffer, &size, ASYNC);
if (status == CM_SUCCESS)
 process_event((EVENT_HEADER *) event_buffer);
 <...do something else...>
 status = cm_yield(0);
} while (status != RPC_SHUTDOWN &&
status != SS_ABORT);
cm_disconnect_experiment();
return 0;
}
```

Return Value: `BM_SUCCESS`,
BM_TRUNCATED The event is larger than the destination buffer and wastherefore trun
BM_ASYNC_RETURN No event available

Parameters:

`buffer_handle` buffer handle

`destination` destination address where event is written to

`buf_size` size of destination buffer on input, size of event plusheader on return.

`async_flag` Synchronous/asynchronous flag. If FALSE, the functionblocks if no event is available. If TRUE, the function returns immediatelywith a value of BM_ASYNC_RETURN without receiving any event.

12.8.27

INT **bm_empty_buffers()** ()

empty event buffer.

Description: Clears event buffer and cache.

Remarks: If an event buffer is large and a consumer is slow in analyzing events, events are usually received some time after they are produced. This effect is even more experienced if a read cache is used (via bm_set_cache_size()). When changes to the hardware are made in the experience, the consumer will then still analyze old events before any new event which reflects the hardware change. Users can be fooled by looking at histograms which reflect the hardware change many seconds after they have been made.

To overcome this potential problem, the analyzer can call bm_empty_buffers() just after the hardware change has been made which skips all old events contained in event buffers and read caches. Technically this is done by forwarding the read pointer of the client. No events are really deleted, they are still visible to other clients like the logger.

Note that the front-end also contains write buffers which can delay the delivery of events. The standard front-end framework mfe.c reduces this effect by flushing all buffers once every second.

Example:

Return Value:	`BM_SUCCESS`
Parameters:	`void`

12.8.28

void **bk_init()** (void* event)

Initialize an event.

Description: Initializes an event for Midas banks structure.

Remarks: Before banks can be created in an event, bk_init() has to be called first.

Return Value:	`void`	
Parameters:	`event`	pointer to the area of event

12.8.29

void **bk_init32()** (void* event)

Initialize an event (> 32KBytes).

Description: Initializes an event for Midas banks structure for large. bank size (> 32KBytes)

Remarks: Before banks can be created in an event, bk_init32() has to be called first.

Return Value:	`void`	
Parameters:	`event`	pointer to the area of event

12.8.30

INT **bk_size()** (void* event)

compute event size.

Description: Returns the size of an event containing banks.

Remarks: The total size of an event is the value returned by bk_size() plus the size of the event header (sizeof(EVENT_HEADER)).

Return Value:	`number`	of bytes contained in data area of event
Parameters:	`event`	pointer to the area of event

12.8.31

void **bk_create()** (void* event, char* name, WORD type, void* pdata)

Create a bank.

Description: Create a Midas bank.

Remarks: The data pointer pdata must be used as an address to fill a bank. It is incremented with every value written to the bank and finally points to a location just after the last byte of the bank. It is then passed to the function bk_close() to finish the bank creation.

Example:
```
INT *pdata;
bk_init(pevent);
bk_create(pevent, "ADC0", TID_INT, &pdata);
*pdata++ = 123
*pdata++ = 456
bk_close(pevent, pdata);
```

Return Value: `void`

Parameters:

`event`	pointer to the data area
`name`	of the bank, must be exactly 4 charaters
`type`	type of bank, one of the Midas Data Types values defined inmidas.h
`pdata`	pointer to the data area of the newly created bank

12.8.32

INT **bk_close()** (void* event, void* pdata)

Close bank.

Description: Close the Midas bank priviously created by bk_create.

Remarks: The data pointer pdata must be obtained by bk_create() and used as an address to fill a bank. It is incremented with every value written to the bank and finally points to a location just after the last byte of the bank. It is then passed to bk_close() to finish the bank creation

Return Value: `number` of bytes contained in bank

Parameters:

`event`	pointer to current composed event
`pdata`	pointer to the data

12.8.33

INT **bk_locate()** (void* event, char* name, void* pdata)

loate a bank in event.

Description: Locates a MIDAS bank of given name inside an event.

Remarks:

Example:

Return Value:	`number`	of values inside the bank
Parameters:	`event`	pointer to current composed event
	`name`	bank name to look for
	`pdata`	pointer to data area of bank, NULL if bank not found

12.8.34

INT **bk_iterate()** (void* event, BANK** pbk, void* pdata)

Retrieve banks pointer from current event.

Description: Iterates through banks inside an event.

Remarks: The function can be used to enumerate all banks of an event. The returned pointer to the bank header has following structure:

```
typedef struct {
char   name[4];
WORD   type;
WORD   data_size;
} BANK;
```

where type is a TID_xxx value and data_size the size of the bank in bytes.

Example:

```
     BANK *pbk;
INT  size;
void *pdata;
char name[5];
pbk = NULL;
do
{
 size = bk_iterate(event, &pbk, &pdata);
 if (pbk == NULL)
  break;
 *((DWORD *)name) = *((DWORD *)(pbk)->name);
 name[4] = 0;
 printf("bank %s found\n", name);
} while(TRUE);
```

Return Value: `Size` of bank in bytes

Parameters:

`event` Pointer to data area of event.

`pbk` pointer to the bank header, must be NULL for the first call tothis function.

`pdata` Pointer to the bank header, must be NULL for the firstcall to this function

12.8.35

INT **bk_swap()** (void* event, BOOL force)

Swap the content of an event.

Description: Swaps bytes from little endian to big endian or vice versa for a whole event.

Remarks: An event contains a flag which is set by bk_init() to identify the endian format of an event. If force is FALSE, this flag is evaluated and the event is only swapped if it is in the "wrong" format for this system. An event can be swapped to the "wrong" format on purpose for example by a front-end which wants to produce events in a "right" format for a back-end analyzer which has different byte ordering.

Return Value: `1==event` has been swap, 0==event has not been swapped.

Parameters:

`event` pointer to data area of event

`force` If TRUE, the event is always swapped, if FALSE, the eventis only swapped if it is in the wrong format.

12.8.36

INT **db_delete_key()** (HNDLE hDB, HNDLE hKey, BOOL follow_links)

Delete ODB key.

Description: Delete a subtree in a database starting from a key (including this key).

Remarks:

Example: ...

```
status = db_find_link(hDB, 0, str, &hkey);
if (status != DB_SUCCESS)
{
  cm_msg(MINFO,"my_delete"," "Cannot find key %s", str);
```

```
      return;
    }

    status = db_delete_key(hDB, hkey, FALSE);
    if (status != DB_SUCCESS)
    {
      cm_msg(MERROR,"my_delete"," "Cannot delete key %s", str);
      return;
    }
  ...
```

Return Value: DB_SUCCESS, DB_INVALID_HANDLE, DB_NO_ACCESS, DB_OPEN_RECORD

Parameters:

hDB	ODB handle obtained via cm_get_experiment_database().
Handle	for key where search starts, zero for root.
follow_links	Follow links when TRUE.

12.8.37

INT **db_find_key()** (HNDLE hDB, HNDLE hKey, char* key_name, HNDLE* subhKey)

Retrieve key handle from key name.

Description: Returns key handle for a key with a specific name.

Remarks: Keys can be accessed by their name including the directory or by a handle. A key handle is an internal offset to the shared memory where the ODB lives and allows a much faster access to a key than via its name. The function db_find_key() must be used to convert a key name to a handle. Most other database functions use this key handle in various operations.

Example:

```
HNDLE hkey, hsubkey;
// use full name, start from root
db_find_key(hDB, 0, "/Runinfo/Run number", &hkey);
// start from subdirectory
db_find_key(hDB, 0, "/Runinfo", &hkey);
db_find_key(hdb, hkey, "Run number", &hsubkey);
```

Return Value: DB_SUCCESS, DB_INVALID_HANDLE, DB_NO_ACCESS, DB_NO_KEY

Parameters:		
	hDB	ODB handle obtained via cm_get_experiment_database().
	hKey	Handle for key where search starts, zero for root.
	key_name	Name of key to search, can contain directories.
	subhKey	Returned handle of key, zero if key cannot be found.

12.8.38

INT **db_set_value()** (HNDLE hDB, HNDLE hKeyRoot, char* key_name, void* data, INT data_size, INT num_values, DWORD type)

Sets key data in ODB.

Description: Set value of a single key.

Remarks: The function sets a single value or a whole array to a ODB key. Since the data buffer is of type void, no type checking can be performed by the compiler. Therefore the type has to be explicitly supplied, which is checked against the type stored in the ODB. key_name can contain the full path of a key (like: "/Equipment/Trigger/Settings/Level1") while hkey is zero which refers to the root, or hkey can refer to a sub-directory (like /Equipment/Trigger) and key_name is interpreted relative to that directory like "Settings/Level1".

Example:
```
INT level1;
db_get_value(hDB, 0, "/Equipment/Trigger/Settings/Level1",
                          &level1, sizeof(level1), TID_INT);
```

Return Value:	DB_SUCCESS,	DB_INVALID_HANDLE, DB_NO_ACCESS, DB_TYPE_MISMATCH
Parameters:	hDB	ODB handle obtained via cm_get_experiment_database().
	hKeyRoot	Handle for key where search starts, zero for root.
	key_name	Name of key to search, can contain directories.
	data	Address of data.
	data_size	Size of data (in bytes).
	num_values	Number of data elements.
	type	Type of key, one of TID_xxx (see Midas Data Types)

12.8.39

INT **db_get_value()** (HNDLE hDB, HNDLE hKeyRoot, char* key_name, void* data, INT* buf_size, DWORD type)

Returns key data from the ODB.

Description: Get value of a single key.

Remarks: The function returns single values or whole arrays which are contained in an ODB key. Since the data buffer is of type void, no type checking can be performed by the compiler. Therefore the type has to be explicitly supplied, which is checked against the type stored in the ODB. key_name can contain the full path of a key (like: "/Equipment/Trigger/Settings/Level1") while hkey is zero which refers to the root, or hkey can refer to a sub-directory (like: /Equipment/Trigger) and key_name is interpreted relative to that directory like "Settings/Level1".

Example:

```
INT level1, size;
size = sizeof(level1);
db_get_value(hDB, 0, "/Equipment/Trigger/Settings/Level1",
                                 &level1, &size, TID_INT);
```

Return Value: DB_SUCCESS, DB_INVALID_HANDLE, DB_NO_ACCESS, DB_TYPE_MISMATCH,DB_TRUNCATED, DB_NO_KEY

Parameters:

hDB	ODB handle obtained via cm_get_experiment_database().
hKeyRoot	Handle for key where search starts, zero for root.
key_name	Name of key to search, can contain directories.
data	Address of data.
buf_size	Maximum buffer size on input, number of written bytes on return.
type	Type of key, one of TID_xxx (see Midas Data Types)

12.8.40

INT **db_enum_key()** (HNDLE hDB, HNDLE hKey, INT index, HNDLE* subkey_handle)

Enumerates keys in a ODB directory.

Description: Enumerate subkeys from a key, follow links.

Remarks: hkey must correspond to a valid ODB directory. The index is usually incremented in a loop until the last key is reached. Information about the sub-keys can be obtained with db_get_key(). If a returned key is of type TID_KEY, it contains itself sub-keys. To scan a whole ODB sub-tree, the function db_scan_tree() can be used.

Example:

```
INT    i;
HNDLE hkey, hsubkey;
KEY   key;
db_find_key(hdb, 0, "/Runinfo", &hkey);
for (i=0 ; ; i++)
{
 db_enum_key(hdb, hkey, i, &hsubkey);
 if (!hSubkey)
  break; // end of list reached
 // print key name
 db_get_key(hdb, hkey, &key);
 printf("%s\n", key.name);
}
```

Return Value: `DB_SUCCESS`, DB_INVALID_HANDLE, DB_NO_MORE_SUBKEYS

Parameters:

`hDB`	ODB handle obtained via cm_get_experiment_database().
`hKey`	Handle for key where search starts, zero for root.
`index`	Subkey index, sould be initially 0, then-incremented in each call until subhKey becomes zero and the functionreturns DB_NO_MORE_SUBKEYS
`subkey_handle`	Handle of subkey which can be used indb_get_key and db_get_data

12.8.41

INT **db_get_key()** (HNDLE hDB, HNDLE hKey, KEY* key)

Returns information about an ODB key.

Description: Get key structure from a handle.

Remarks: The KEY structure has following format:

```
typedef struct {
  DWORD         type;                 // TID_xxx type
  INT           num_values;           // number of values
  char          name[NAME_LENGTH];    // name of variable
  INT           data;                 // Address of variable (offset)
  INT          total_size;          // Total size of data block
  INT          item_size;           // Size of single data item
  WORD         access_mode;         // Access mode
  WORD         lock_mode;           // Lock mode
  WORD         exclusive_client;    // Index of client in excl. mode
  WORD         notify_count;        // Notify counter
  INT          next_key;            // Address of next key
  INT          parent_keylist;      // keylist to which this key belongs
  INT          last_written;        // Time of last write action
} KEY;
```

Most of these values are used for internal purposes, the values which are of public interest are type, num_values, and name. For keys which contain a single value, num_values equals to one and total_size equals to item_size. For keys which contain an array of strings (TID_STRING), item_size equals to the length of one string.

Example:

```
KEY   key;
HNDLE hkey;
db_find_key(hDB, 0, "/Runinfo/Run number", &hkey);
db_get_key(hDB, hkey, &key);
printf("The run number is of type %s\n", rpc_tid_name(key.type));
```

Return Value: DB_SUCCESS, DB_INVALID_HANDLE

Parameters:

hDB	ODB handle obtained via cm_get_experiment_database().
hKey	Handle for key where search starts, zero for root.
key	Pointer to KEY stucture.

12.8.42

INT **db_get_data()** (HNDLE hDB, HNDLE hKey, void* data, INT* buf_size, DWORD type)

Returns data from a key.

Description: Get key data from a handle

Remarks: The function returns single values or whole arrays which are contained in an ODB key. Since the data buffer is of type void, no type checking can be performed by the compiler. Therefore the type has to be explicitly supplied, which is checked against the type stored in the ODB.

Example:

```
HNLDE hkey;
INT   run_number, size;
// get key handle for run number
db_find_key(hDB, 0, "/Runinfo/Run number", &hkey);
```

```
    // return run number
    size = sizeof(run_number);
    db_get_data(hDB, hkey, &run_number, &size,TID_INT);
```

Return Value: DB_SUCCESS, DB_INVALID_HANDLE, DB_TRUNCATED, DB_TYPE_MISMATCH

Parameters:

`hDB`	ODB handle obtained via cm_get_experiment_database().
`hKey`	Handle for key where search starts, zero for root.
`buf_size`	Size of data buffer.
`type`	Type of key, one of TID_xxx (see Midas Data Types).

12.8.43

INT **db_get_data_index()** (HNDLE hDB, HNDLE hKey, void* data, INT* buf_size, INT index, DWORD type)

Get single element of data from an array handle.

Description: returns a single value of keys containing arrays of values.

Remarks: The function returns a single value of keys containing arrays of values.

Example:

Return Value: DB_SUCCESS, DB_INVALID_HANDLE, DB_TRUNCATED, DB_OUT_OF_RANGE

Parameters:

`hDB`	ODB handle obtained via cm_get_experiment_database().
`hKey`	Handle for key where search starts, zero for root.
`data`	Size of data buffer.
`index`	Index of array [0..n-1].
`type`	Type of key, one of TID_xxx (see Midas Data Types).

12.8.44

INT **db_set_data()** (HNDLE hDB, HNDLE hKey, void* data, INT buf_size, INT num_values, DWORD type)

Sets data of a key.

Description: Set key data from a handle. Adjust number of values if previous data has different size.

Remarks:

Example:

```
HNLDE hkey;
INT   run_number;
// get key handle for run number
db_find_key(hDB, 0, "/Runinfo/Run number", &hkey);
// set run number
db_set_data(hDB, hkey, &run_number, sizeof(run_number),TID_INT);
```

Return Value:	DB_SUCCESS,	DB_INVALID_HANDLE, DB_TRUNCATED
Parameters:	hDB	ODB handle obtained via cm_get_experiment_database().
	hKey	Handle for key where search starts, zero for root.
	data	Buffer from which data gets copied to.
	buf_size	Size of data buffer.
	num_values	Number of data values (for arrays).
	type	Type of key, one of TID_xxx (see Midas Data Types).

12.8.45

INT **db_set_data_index()** (HNDLE hDB, HNDLE hKey, void* data, INT data_size, INT index, DWORD type)

Set individual values of a key array.

Description: Set key data for a key which contains an array of values.

Remarks: This function sets individual values of a key containing an array. If the index is larger than the array size, the array is extended and the intermediate values are set to zero.

Example:

Return Value:	DB_SUCCESS,	DB_INVALID_HANDLE, DB_NO_ACCESS, DB_TYPE_MISMATCH
Parameters:	hDB	ODB handle obtained via cm_get_experiment_database().
	hKey	Handle for key where search starts, zero for root.
	data	Pointer to single value of data.
	data_size	
	index	Size of single data element.
	type	Type of key, one of TID_xxx (see Midas Data Types).

12.8.46

INT **db_load()** (HNDLE hDB, HNDLE hKeyRoot, char* filename, BOOL bRemote)

Loads ODB entries from an ASCII file.

Description: Load a branch of a database from an .ODB file.

Remarks: This function is used by the ODBEdit command load. For a description of the ASCII format, see db_copy(). Data can be loaded relative to the root of the ODB (hkey equal zero) or relative to a certain key.

Example:

Return Value:	DB_SUCCESS, DB_INVALID_HANDLE, DB_FILE_ERROR	
Parameters:	`hDB`	ODB handle obtained via cm_get_experiment_database().
	`hKeyRoot`	Handle for key where search starts, zero for root.
	`filename`	Filename of .ODB file.
	`bRemote`	If TRUE, the file is loaded by the server process on theback-end, if FALSE, it is loaded from the current process

12.8.47

INT **db_copy()** (HNDLE hDB, HNDLE hKey, char* buffer, INT* buffer_size, char* path)

Copies part of the ODB into an ASCII string.

Description: Copy an ODB subtree in ASCII format to a buffer

Remarks: This function converts the binary ODB contents to an ASCII. The function db_paste() can be used to convert the ASCII representation back to binary ODB contents. The functions db_load() and db_save() internally use db_copy() and db_paste(). This function converts the binary ODB contents to an ASCII representation of the form:

- for single value:

```
[ODB path]
key name = type : value
```

- for strings:

```
key name = STRING : [size] string contents
```

- for arrayes (type can be BYTE, SBYTE, CHAR, WORD, SHORT, DWORD, INT, BOOL, FLOAT, DOUBLE, STRING or LINK):

```
key name = type[size] :
[0] value0
[1] value1
[2] value2
...
```

Example:

Return Value:	DB_SUCCESS,	DB_TRUNCATED, DB_NO_MEMORY
Parameters:	hDB	ODB handle obtained via cm_get_experiment_database().
	hKey	Handle for key where search starts, zero for root.
	buffer	ASCII buffer which receives ODB contents.
	buffer_size	Size of buffer, returns remaining space in buffer.
	path	Internal use only, must be empty ("").

12.8.48

INT **db_paste()** (HNDLE hDB, HNDLE hKeyRoot, char* buffer)

Pastes values into the ODB from an ASCII string.

Description: Copy an ODB subtree in ASCII format from a buffer

Remarks:

Example:

Return Value:	DB_SUCCESS,	DB_TRUNCATED, DB_NO_MEMORY
Parameters:	hDB	ODB handle obtained via cm_get_experiment_database().
	hKeyRoot	Handle for key where search starts, zero for root.
	buffer	NULL-terminated buffer

12.8.49

INT **db_save()** (HNDLE hDB, HNDLE hKey, char* filename, BOOL bRemote)

Save ODB entries to an ASCII file.

Description: Save a branch of a database to an .ODB file

Remarks: This function is used by the ODBEdit command save. For a description of the ASCII format, see db_copy(). Data of the whole ODB can be saved (hkey equal zero) or only a sub-tree.

Example:

Return Value:	`DB_SUCCESS`, DB_FILE_ERROR	
Parameters:	`hDB`	ODB handle obtained via cm_get_experiment_database().
	`hKey`	Handle for key where search starts, zero for root.
	`filename`	Filename of .ODB file.
	`bRemote`	Flag for saving database on remote server.

12.8.50

INT **db_sprintf()** (char* string, void* data, INT data_size, INT index, DWORD type)

Convert an ODB entry to a string.

Description: Convert a database value to a string according to its type.

Remarks: This function is a convenient way to convert a binary ODB value into a string depending on its type if is not known at compile time. If it is known, the normal sprintf() function can be used.

Example:

```
...
for (j=0 ; j<key.num_values ; j++)
{
  db_sprintf(pbuf, pdata, key.item_size, j, key.type);
  strcat(pbuf, "\n");
}
...
```

Return Value:	DB_SUCCESS	
Parameters:	`string`	output ASCII string of data.
	`data`	Value data.
	`data_size`	Size of single data element.
	`index`	Index for array data.
	`type`	Type of key, one of TID_xxx (see Midas Data Types).

12.8.51

INT **db_get_record_size()** (HNDLE hDB, HNDLE hKey, INT align, INT* buf_size)

Get record size.

Description: Calculates the size of a record.

Remarks:

Example:

Return Value:	DB_SUCCESS,	DB_INVALID_HANDLE, DB_TYPE_MISMATCH,DB_STRUCT_SIZE_MISMATCH, DB_NO_KEY
Parameters:	`hDB`	ODB handle obtained via cm_get_experiment_database().
	`hKey`	Handle for key where search starts, zero for root.
	`align`	Byte alignment calculated by the stub andpassed to the rpc side to align data according to local machine. Must be zerowhen called from user level
	`buf_size`	Size of record structure

12.8.52

INT **db_get_record()** (HNDLE hDB, HNDLE hKey, void* data, INT* buf_size, INT align)

Copies an ODB sub-tree to a local C structure.

Description: Copy a set of keys to local memory.

Remarks: An ODB sub-tree can be mapped to a C structure automatically via a hot-link using the function db_open_record() or manually with this function. Problems might occur if the ODB sub-tree contains values which don't match the C structure. Although the structure size is checked against the sub-tree size, no checking can be done if the type and order of the values in the structure are the same than those in the ODB sub-tree. Therefore it is recommended to use the function db_create_record() before db_get_record() is used which ensures that both are equivalent.

Example:
```
struct {
  INT level1;
  INT level2;
} trigger_settings;
char *trigger_settings_str =
"[Settings]\n\
level1 = INT : 0\n\
level2 = INT : 0";

main()
{
  HNDLE hDB, hkey;
  INT   size;
  ...
  cm_get_experiment_database(&hDB, NULL);
  db_create_record(hDB, 0, "/Equipment/Trigger", trigger_settings_str);
  db_find_key(hDB, 0, "/Equipment/Trigger/Settings", &hkey);
  size = sizeof(trigger_settings);
  db_get_record(hDB, hkey, &trigger_settings, &size, 0);
  ...
}
```

Return Value: `DB_SUCCESS`, DB_INVALID_HANDLE, DB_STRUCT_SIZE_MISMATCH

Parameters:

`hDB`	ODB handle obtained via cm_get_experiment_database().
`hKey`	Handle for key where search starts, zero for root.
`buf_size`	Size of data structure, must be obtained via sizeof(RECORD-NAME).
`align`	Byte alignment calculated by the stub andpassed to the rpc side to align data according to local machine. Must be zerowhen called from user level.

12.8.53

INT **db_set_record()** (HNDLE hDB, HNDLE hKey, void* data, INT buf_size, INT align)

Copies a local C structure to a ODB sub-tree.

Description: Copy a set of keys from local memory to the database.

Remarks: An ODB sub-tree can be mapped to a C structure automatically via a hot-link using the function db_open_record() or manually with this function. Problems might occur if the ODB sub-tree contains values which don't match the C structure. Although the structure size is checked against the sub-tree size, no checking can be done if the type and order of the values in the structure are the same than those in the ODB sub-tree. Therefore it is recommended to use the function db_create_record() before using this function.

Example: ...

```
    memset(&lazyst,0,size);
    if (db_find_key(hDB, pLch->hKey, "Statistics",&hKeyst) == DB_SUCCESS)
      status = db_set_record(hDB, hKeyst, &lazyst, size, 0);
    else
      cm_msg(MERROR,"task","record %s/statistics not found", pLch->name)
  ...
```

Return Value:	DB_SUCCESS,	DB_INVALID_HANDLE, DB_TYPE_MISMATCH, DB_STRUCT_SIZE_MISMATCH
Parameters:	hDB	ODB handle obtained via cm_get_experiment_database().
	hKey	Handle for key where search starts, zero for root.
	data	Pointer where data is stored.
	buf_size	Size of data structure, must be obtained via sizeof(RECORD-NAME).
	align	Byte alignment calculated by the stub andpassed to the rpc side to align data according to local machine. Must be zerowhen called from user level.

12.8.54

INT **db_create_record()** (HNDLE hDB, HNDLE hKey, char* key_name, char* init_str)

Creates an ODB sub-tree from an ASCII representation.

Description: Create a record. If a part of the record exists alreay, merge it with the init_str (use values from the init_str only when they are not in the existing record).

Remarks: This functions creates a ODB sub-tree according to an ASCII representation of that tree. See db_copy() for a description. It can be used to create a sub-tree which exactly matches a C structure. The sub-tree can then later mapped to the C structure ("hot-link") via the function db_open_record().

If a sub-tree exists already which exactly matches the ASCII representation, it is not modified. If part of the tree exists, it is merged with the ASCII representation where the ODB values have priority, only values not present in the ODB are created with the default values of the ASCII representation. It is therefore recommended that before creating an ODB hot-link the function db_create_record() is called to insure that the ODB tree and the C structure contain exactly the same values in the same order.

Following example creates a record under /Equipment/Trigger/Settings, opens a hot-link between that record and a local C structure trigger_settings and registers a callback function trigger_update() which gets called each time the record is changed.

Example:

```
struct {
  INT level1;
  INT level2;
} trigger_settings;
char *trigger_settings_str =
"[Settings]\n\
level1 = INT : 0\n\
level2 = INT : 0";
void trigger_update(INT hDB, INT hkey, void *info)
{
  printf("New levels: %d %d\n",
    trigger_settings.level1,
    trigger_settings.level2);
}
main()
{
  HNDLE hDB, hkey;
  char[128] info;
  ...
  cm_get_experiment_database(&hDB, NULL);
  db_create_record(hDB, 0, "/Equipment/Trigger", trigger_settings_str);
  db_find_key(hDB, 0,"/Equipment/Trigger/Settings", &hkey);
  db_open_record(hDB, hkey, &trigger_settings,
    sizeof(trigger_settings), MODE_READ, trigger_update, info);
  ...
}
```

Return Value: `DB_SUCCESS`, DB_INVALID_HANDLE, DB_FULL, DB_NO_ACCESS, DB_OPEN_RECORD

Parameters:

`hDB` ODB handle obtained via cm_get_experiment_database().

`hKey` Handle for key where search starts, zero for root.

`key_name` Name of key to search, can contain directories.

`init_str` Initialization string in the format of the db_copy/db_save functions.

12.8.55

INT **db_open_record()** (HNDLE hDB, HNDLE hKey, void* ptr, INT rec_size, WORD access_mode, void (*dispatcher)(INT, INT, void*), void* info)

Creates a hot-link between an ODB sub-tree and a C structure.

Description: Open a record. Create a local copy and maintain an automatic update.

Remarks: This function opens a hot-link between an ODB sub-tree and a local structure. The sub-tree is copied to the structure automatically every time it is modified by someone else. Additionally, a callback function can be declared which is called after the structure has been updated. The callback function receives the database handle and the key handle as parameters.

Problems might occur if the ODB sub-tree contains values which don't match the C structure. Although the structure size is checked against the sub-tree size, no checking can be done if the type and order of the values in the structure are the same than those in the ODB sub-tree. Therefore it is recommended to use the function db_create_record() before db_open_record() is used which ensures that both are equivalent.

The access mode might either be MODE_READ or MODE_WRITE. In read mode, the ODB sub-tree is automatically copied to the local structure when modified by other clients. In write mode, the local structure is copied to the ODB sub-tree if it has been modified locally. This update has to be manually scheduled by calling db_send_changed_records() periodically in the main loop. The system keeps a copy of the local structure to determine if its contents has been changed.

If MODE_ALLOC is or'ed with the access mode, the memory for the structure is allocated internally. The structure pointer must contain a pointer to a pointer to the structure. The internal memory is released when db_close_record() is called.

Example: To open a record in write mode.

```
struct {
  INT level1;
  INT level2;
} trigger_settings;
char *trigger_settings_str =
"[Settings]\n\
level1 = INT : 0\n\
level2 = INT : 0";
main()
{
  HNDLE hDB, hkey, i=0;
  ...
  cm_get_experiment_database(&hDB, NULL);
  db_create_record(hDB, 0, "/Equipment/Trigger", trigger_settings_str);
  db_find_key(hDB, 0,"/Equipment/Trigger/Settings", &hkey);
  db_open_record(hDB, hkey, &trigger_settings, sizeof(trigger_settings)
                  , MODE_WRITE, NULL);
    do
    {
      trigger_settings.level1 = i++;
      db_send_changed_records()
      status = cm_yield(1000);
    } while (status != RPC_SHUTDOWN && status != SS_ABORT);
    ...
  }
```

Return Value:	DB_SUCCESS,	DB_INVALID_HANDLE, DB_NO_MEMORY, DB_NO_ACCESS, DB_STRUCT_SIZE_MISMATCH
Parameters:	hDB	ODB handle obtained via cm_get_experiment_database().
	hKey	Handle for key where search starts, zero for root.
	ptr	If access_mode includes MODE_ALLOC:Address of pointer which points to therecord data after the callif access_mode includes not MODE_ALLOC:Address of recordif ptr==NULL, only the dispatcher is called.
	access_mode	Mode for opening record, either MODE_READ orMODE_WRITE. May be or'ed with MODE_ALLOC tolet db_open_record allocate the memory forthe record.
	dispatcher	Function which gets called when record is updated.Theargument list composed of: HNDLE hDB, HNDLE hKey, void *info
	info	Additional info passed to the dispatcher.

12.8.56

INT **db_close_record()** (HNDLE hDB, HNDLE hKey)

Close open record.

Description: Close a record previously opend with db_open_record.

Remarks:

Example:

Return Value:	DB_SUCCESS,	DB_INVALID_HANDLE
Parameters:	hDB	ODB handle obtained via cm_get_experiment_database().
	hKey	Handle for key where search starts, zero for root.

12.8.57

INT **db_send_changed_records()** ()

update ODB from local open records.

Description: Send all records to the ODB which were changed locally since the last call to this function.

Remarks: This function is valid if used in conjunction with db_open_record() under the condition the record is open as MODE_WRITE access code.

Example: Full example dbchange.c which can be compiled as follow
gcc -DOS_LINUX -I/midas/include -o dbchange dbchange.c /midas/linux/lib/libmidas.a -lutil

```
//------- dbchange.c
#include "midas.h"
#include "msystem.h"

typedef struct {
    INT    my_number;
    float   my_rate;
} MY_STATISTICS;

MY_STATISTICS myrec;

#define MY_STATISTICS(_name) char *_name[] = {\
"My Number = INT : 0",\
"My Rate = FLOAT : 0",\
"",\
NULL }

HNDLE hDB, hKey;

// Main
int main(unsigned int argc,char **argv)
{
  char      host_name[HOST_NAME_LENGTH];
  char      expt_name[HOST_NAME_LENGTH];
  INT       lastnumber, status, msg;
  BOOL      debug=FALSE;
  char      i, ch;
  DWORD     update_time, mainlast_time;
  MY_STATISTICS (my_stat);

  // set default
  host_name[0] = 0;
  expt_name[0] = 0;
```

```
// get default
cm_get_environment (host_name, expt_name);

// get parameters
for (i=1 ; i<argc ; i++)
{
  if (argv[i][0] == ’-’ && argv[i][1] == ’d’)
    debug = TRUE;
  else if (argv[i][0] == ’-’)
  {
    if (i+1 >= argc || argv[i+1][0] == ’-’)
      goto usage;
    if (strncmp(argv[i],"-e",2) == 0)
      strcpy(expt_name, argv[++i]);
    else if (strncmp(argv[i],"-h",2)==0)
      strcpy(host_name, argv[++i]);
  }
  else
  {
 usage:
    printf("usage: dbchange [-h <Hostname>] [-e <Experiment>]\n");
    return 0;
  }
}

// connect to experiment
status = cm_connect_experiment(host_name, expt_name, "dbchange", 0);
if (status != CM_SUCCESS)
  return 1;

// Connect to DB
cm_get_experiment_database(&hDB, &hKey);

// Create a default structure in ODB
db_create_record(hDB, 0, "My statistics", strcomb(my_stat));

// Retrieve key for that strucutre in ODB
if (db_find_key(hDB, 0, "My statistics", &hKey) != DB_SUCCESS)
{
  cm_msg(MERROR, "mychange", "cannot find My statistics");
  goto error;
}

// Hot link this structure in Write mode
status = db_open_record(hDB, hKey, &myrec
, sizeof(MY_STATISTICS), MODE_WRITE, NULL, NULL);
if (status != DB_SUCCESS)
{
  cm_msg(MERROR, "mychange", "cannot open My statistics record");
  goto error;
}

// initialize ss_getchar()
ss_getchar(0);
```

```
    // Main loop
    do
    {
      // Update local structure
      if ((ss_millitime() - update_time) > 100)
      {
        myrec.my_number += 1;
        if (myrec.my_number - lastnumber) {
  myrec.my_rate = 1000.f * (float) (myrec.my_number - lastnumber)
    / (float) (ss_millitime() - update_time);
        }
        update_time = ss_millitime();
        lastnumber = myrec.my_number;
      }

      // Publish local structure to ODB (db_send_changed_record)
      if ((ss_millitime() - mainlast_time) > 5000)
      {
        db_send_changed_records();                    // <------- Call
        mainlast_time = ss_millitime();
      }

      // Check for keyboard interaction
      ch = 0;
      while (ss_kbhit())
      {
        ch = ss_getchar(0);
        if (ch == -1)
  ch = getchar();
        if ((char) ch == '!')
  break;
      }
      msg = cm_yield(20);
    } while (msg != RPC_SHUTDOWN && msg != SS_ABORT && ch != '!');

   error:
    cm_disconnect_experiment();
    return 1;
  }
  //-------- EOF dbchange.c
```

Return Value: DB_SUCCESS

12.9 MIDAS Macros

Message Macros, Acquisition. Exportable MACROs through midas.h, msystem.h or ybos.h.

Names

Several group of MACROs are available for simplifying user job on setting or getting Midas information. They are also listed in the Midas Library. All of them are defined in the **midas.h, ybos.h** header files.

Message Macros To be used with cm_msg.

)AQ Event/LAM Macros To be used in the frontend/analyzer code.

12.9.1

DAQ Macros

LAM and Event header manipulation

CAMAC LAM manipulation These Macros are used in the frontend code to interact with the LAM register. Usualy the CAMAC Crate Controler has the feature to register one bit per slot and be able to present this register to the user. It may even have the option to mask off this register to allow to set a "general" LAM register containing either "1" (At least one LAM from the masked LAM is set) or "0" (no LAM set from the maksed LAM register).

The *poll_event()* uses this feature and return a variable which contains bit-wise the current LAM register of the Crate Controller.

- LAM_SOURCE
- LAM_STATION
- LAM_SOURCE_CRATE (→ 12.8.1, *page* 197)
- LAM_SOURCE_STATION (→ 12.8.1, *page* 197)

BYTE swap manipulation These Macros can be used in the backend analyzer when *little-endian/big-endian* are mixed in the event.

- WORD_SWAP
- DWORD_SWAP
- QWORD_SWAP

MIDAS Event Header manipulation Every event travelling through the Midas system has a "Event Header" containing the minimum information required to identify its content. The size of the header has been kept as small as possible in order to minimize its impact on the data rate as well as on the data storage requirment. The following macros permit to read or override the content of the event header as long as the argument of the macro refers to the top of the Midas event (pevent). This argument is available in the frontend code in any of the user readout function (pevent). It is also available in the user analyzer code which retrieve the event and provide directly access to the event header (pheader) and to the user part of the event (pevent). Sub-function using pevent would then be able to get back the the header through the use of the macros.

amples/experiment/adccalib.c

```
INT adc_calib(EVENT_HEADER *pheader, void *pevent)
{
INT    i, n_adc;
WORD   *pdata;
float  *cadc;

  // look for ADC0 bank, return if not present
  n_adc = bk_locate(pevent, "ADC0", &pdata);
  if (n_adc == 0 || n_adc > N_ADC)
    return 1;

  // create calibrated ADC bank
  bk_create(pevent, "CADC", TID_FLOAT, &cadc);
  ...
```

amples/experiment/frontend.c

```
INT read_trigger_event(char *pevent, INT off)
{
WORD *pdata, a;
INT  q, timeout;

  // init bank structure
  bk_init(pevent);
  ...
```

nent from running experiment

```
INT read_ge_event(char *pevent, INT offset)
{
  static WORD *pdata;
  INT i, x, q;
  WORD temp;

  // Change the time stamp in millisecond for the Super event
  TIME_STAMP(pevent) = ss_millitime();

  bk_init(pevent);
  bk_create(pevent, "GERM", TID_WORD, &pdata);
  ...
```

nent from running experiment

```
  ...
  lam = *((DWORD *)pevent);

  if (lam & LAM_STATION(JW_N))
  {
    ...
    // compose event header
    TRIGGER_MASK(pevent) = JW_MASK;
    EVENT_ID(pevent)     = JW_ID;
    SERIAL_NUMBER(pevent)= eq->serial_number++;
    // read MCS event
    size = read_mcs_event(pevent);
    // Correct serial in case event is empty
    if (size == 0)
      SERIAL_NUMBER(pevent) = eq->serial_number--;
    ...
  }
  ...
```

- TRIGGER_MASK
- EVENT_ID
- SERIAL_NUMBER
- TIME_STAMP

12.9.2

Message Macros

Message macros for cm_msg()

These Macros compact the 3 first arguments of the cm_msg() call. It replaces the type of message, the routine name and the line number in the C-code. See example in cm_msg().

MERROR For error (MT_ERROR, __FILE__, __LINE__)

MINFO For info (MT_INFO, __FILE__, __LINE__)

MDEBUG For debug (MT_DEBUG, __FILE__, __LINE__)

MUSER Produced by interactive user (MT_USER, __FILE__, __LINE__)

MLOG Info message which is only logged (MT_LOG, __FILE__, __LINE__)

MTALK Info message for speech system (MT_TALK, __FILE__, __LINE__)

MCALL Info message for telephone call (MT_CALL, __FILE__, __LINE__)

The Message codes are:

```
#define MT_ERROR           (1<<0)
#define MT_INFO            (1<<1)
#define MT_DEBUG           (1<<2)
#define MT_USER            (1<<3)
#define MT_LOG             (1<<4)
#define MT_TALK            (1<<5)
#define MT_CALL            (1<<6)
#define MT_ALL              0xFF
```

12.10

YBOS library

exportable ybos functions through inclusion of ybos.h

Names

12.10.1

#define **EVID bank**

EVID bank description with available macro's.

As soon as the Midas header is striped out from the event, the YBOS remaining data has lost the event synchonization unless included by the user. It is therefore necessary to have a YBOS bank duplicating this information usually done in the FE by creating a "EVID" bank filled with the Midas info and other user information.

Unfortunately the format of this EVID is flexible and I couldn't force user to use a default structure. For this reason, I'm introducing a preprocessor flag for selecting such format.

Omitting the declaration of the pre-processor flag the EVID_TRINAT is taken by default see appendix F: Midas build options and consideration.

Special macros are avaialbe to retrieve this information based on the EVID content and the type of EVID structure.

OS_EVID_EVENT_ID(e) Extract the Event ID.

ID_TRIGGER_MASK(e) Extract the Trigger mask.

YBOS_EVID_SERIAL(e) Extract the Serial number.

ne YBOS_EVID_TIME(e) Extract the time stamp.

EVID_RUN_NUMBER(e) Extract the run number.

OS_EVID_EVENT_NB(e) Extract the event counter.

The Macro parameter should point to the first data of the EVID bank.

```
// check if EVID is present if so display its content
if ((status = ybk_find (pybos, "EVID", &bklen, &bktyp, (void *)&pybk)) == YB_SUCCESS)
{
  pdata = (DWORD *)((YBOS_BANK_HEADER *)pybk + 1);
  pevent->event_id      = YBOS_EVID_EVENT_ID(pdata);
  pevent->trigger_mask  = YBOS_EVID_TRIGGER_MASK(pdata);
  pevent->serial_number = YBOS_EVID_SERIAL(pdata);
  pevent->time_stamp    = YBOS_EVID_TIME(pdata);
  pevent->data_size     = pybk->length;
}
```

The current type of EVID bank are:

EVID_TRINAT Specific for Trinat experiment.

```
ybk_create((DWORD *)pevent, "EVID", I4_BKTYPE, (DWORD *)(&pbkdat));
*((WORD *)pbkdat) = EVENT_ID(pevent);     ((WORD *)pbkdat)++;
*((WORD *)pbkdat) = TRIGGER_MASK(pevent); ((WORD *)pbkdat)++;
*(pbkdat)++ = SERIAL_NUMBER(pevent);
*(pbkdat)++ = TIME_STAMP(pevent);
*(pbkdat)++ = gbl_run_number;                // run number
```

EVID_CHAOS Specific to CHAOS experiment.

```
need code here.
```

EVID_TWIST Specific to Twist Experiment (Triumf).

```
ybk_create((DWORD *)pevent, "EVID", I4_BKTYPE, &pbkdat);
*((WORD *)pbkdat) = EVENT_ID(pevent);     ((WORD *)pbkdat)++;
*((WORD *)pbkdat) = TRIGGER_MASK(pevent); ((WORD *)pbkdat)++;
*(pbkdat)++ = SERIAL_NUMBER(pevent);
*(pbkdat)++ = TIME_STAMP(pevent);
*(pbkdat)++ = gbl_run_number;               // run number
*(pbkdat)++ = *((DWORD *)frontend_name);    // frontend name
ybk_close((DWORD *)pevent, pbkdat);
```

Parameters: `e` pointer to the first data of the bank.

12.10.2

INT **bk_list()** (BANK_HEADER* pmbh, char* bklist)

fill a string will all the bank names in the event.

Description: extract the MIDAS bank name listing of an event.

Remarks: The bklist should be dimensioned with YB_STRING_BANKLIST_MAX which correspond to a max of YB_BANKLIST_MAX (midas.h:32) banks.

Return Value: `number` of bank found in this event.

Parameters:
`pmbh` pointer to the bank header (pheader+1).
`bklist` returned ASCII string, has to be booked with YB_STRING_BANKLIST_MAX.

12.10.3

void **ybk_init** (DWORD* plrl)

Initialize an event.

ybk_init()

Description: Initializes an event for YBOS banks structure.

Remarks: Before banks can be created in an event, ybk_init() has to be called first. See YBOS bank examples.

Return Value: `void`

Parameters: `plrl` pointer to the first DWORD of the event area of event

12.10.4

void **ybk_create** (DWORD* plrl, char* bkname, DWORD bktype, void* pbkdat)

Create a YBOS bank.

ybk_create()

Description: Define the following memory area to be a YBOS bank with the given attribute. See YBOS bank examples.

Remarks: Before banks can be created in an event, ybk_init(). has to be called first. YBOS does not support mixed bank type. i.e: all the data are expected to be of the same type. YBOS is a 4 bytes bank aligned structure. Padding is performed at the closing of the bank (see ybk_close) with values of 0x0f or/and 0x0ffb. See YBOS bank examples.

Return Value:	`void`	
Parameters:	`pevent`	pointer to the first DWORD of the event area.
	`bkname`	name to be assigned to the breated bank (max 4 char)
	`bktype`	YBOS Bank Types of the values for the entire created bank.
	`pbkdat`	return pointer to the first empty data location.

12.10.5

INT **ybk_close** (DWORD* plrl, void* pbkdat)

Close YBOS bank.

ybk_close()

Description: Close the YBOS bank previously created by ybk_create.

Remarks: The data pointer pdata must be obtained by ybk_create() (→ 12.10.4, *page* 245) and used as an address to fill a bank. It is incremented with every value written to the bank and finally points to a location just after the last byte of the bank. It is then passed to ybk_close() to finish the bank creation. YBOS is a 4 bytes bank aligned structure. Padding is performed at the closing of the bank with values of 0x0f or/and 0x0ffb. See YBOS bank examples.

Return Value:	`number`	of bytes contained in bank.
Parameters:	`plrl`	pointer to current composed event.
	`pbkdata`	pointer to the current data.

12.10.6

INT **ybk_size** (DWORD* plrl)

compute event size in bytes.

ybk_size()

Description: Returns the size in bytes of the event composed of YBOS bank(s).

Remarks:

Return Value:	`number`	of bytes contained in data area of the event
Parameters:	`plrl`	pointer to the area of event

12.10.7

INT **ybk_list** (DWORD* plrl, char* bklist)

Extract the bank list of an event composed of YBOS banks.

ybk_list()

Description: Returns the size in bytes of the event composed of YBOS bank(s).

Remarks: The bklist has to be a predefined string of max size of YB_STRING_BANKLIST_MAX.

Return Value:	`number`	of banks found in this event.
Parameters:	`plrl`	pointer to the area of event
	`bklist`	Filled character string of the YBOS bank names found in the event.

12.10.8

INT **ybk_find** (DWORD* plrl, char* bkname, DWORD* bklen, DWORD* bktype, void** pbk)

Find bank in event.

ybk_find()

Description: Find the requested bank and return the infirmation if the bank as well as the pointer to the top of the data section.

Remarks:

Return Value:	YB_SUCCESS,	YB_BANK_NOT_FOUND, YB_WRONG_BANK_TYPE
Parameters:	plrl	pointer to the area of event.
	bkname	name of the bank to be located.
	bklen	returned length in 4bytes unit of the bank.
	bktype	returned bank type.
	pbkdata	pointer to the first data of the found bank.

12.10.9

INT **ybk_locate** (DWORD* plrl, char* bkname, void* pdata)

Locate bank in event.

ybk_locate()

Description: Locate the requested bank and return the pointer to the top of the data section.

Remarks:

Return Value:	Number	of DWORD in bank or YB_BANK_NOT_FOUND, YB_WRONG_BANK_TYPE (<0)
Parameters:	plrl	pointer to the area of event
	bkname	name of the bank to be located.
	pdata	pointer to the first data of the located bank.

12.10.10

INT **ybk_iterate** (DWORD* plrl, YBOS_BANK_HEADER** pybkh, void** pdata)

Find bank in event.

ybk_iterate()

Description: Returns the bank header pointer and data pointer of the given bank name.

Remarks:

Return Value:

`data` length in 4 bytes unit. return -1 if no more bank found.

Parameters:

`plrl` pointer to the area of event.
`bkname` name of the bank to be located.
`pybkh` pointer to the YBOS bank header.
`pdata` pointer to the first data of the current bank.

12.10.11

INT **yb_any_file_rclose** (INT data_fmt)

13 appendix F: Midas build options and consideration

There are several pre-compiler flags which can be of some use.

OS selection ”OS_xxxxx” The current OS support is done through fix flag established in the general **Makefile**. Currently the OS supported are: **OS_OSF1**, **OS_ULTRIX**, **OS_FREEBSD**, **OS_LINUX** , **OS_SOLARIS**. For **OS_IRIX** please contact Pierre. The OS_VMS is not included in the Makefile as it requires a particular one and since several years now the VMS support has been dropped.

```
OSFLAGS = -DOS_LINUX ...
```

user flag ”USERFLAGS” In case you need more switch during compilation you can use the variable **USERFLAGS** included in the **OSFLAGS**.

```
 make USERFLAGS=-static linux/bin/mstat
```

”MIDAS_PREF_FLAGS’ This flag is for internal global Makefile preference. Included in the **OSFLAGS**.

```
MIDAS_PREF_FLAGS  = -DYBOS_VERSION_3_3 -DEVID_TWIST
```

”SPECIFIC_OS_PRG” This flag is for internal Makefile preference. Used in particular for additional applications build based on the OS selection. In the example below **mlxspeaker** and **dio** are built only under OS_LINUX.

```
SPECIFIC_OS_PRG = $(BIN_DIR)/mlxspeaker $(BIN_DIR)/dio
```

rt ”INCLUDE_FTPLIB” Application such as the **mlogger**, **lazylogger** can use the ftp channel for data transfer.

pport ”INCLUDE_ZLIB” The applications **lazylogger**, **mdump** can be built with **zlib.a** in order to gain direct access to the data files with the extension *mid.gz* or *ybs.gz* (module ybos.c). Building the application static prevent to rebuild the whole midas package. The analyzer **mana** has this option already enabled.

```
 make USERFLAGS=-DINCLUDE_ZLIB linux/lib/ybos.o
 make USERFLAGS=-static linux/bin/mdump
```

n ”YBOS_VERSION_3_3” The default built for ybos support is version 4.0. If lower version is required include **YBOS_VERSION_3_3** during compilation of the ybos.c

```
 make USERFLAGS=-DYBOS_VERSION_3_3 linux/lib/ybos.o
```

static built By default the midas applications are built against the dynamic library **libmidas.so**. In case static application is required, use the flag **-static** during linking.

```
 make USERFLAGS=-static linux/bin/mstat
```

”DM_DUAL_THREAD” **Valid only under VxWorks**. This flag enable the dual thread task when running the frontend code under VxWorks. The main function calls are the dm_xxxx in midas.c (Refer to Pierre for more information).

SE_EVENT_CHANNEL” To be used in conjunction with the **DM_DUAL_THREAD**.

USE_INT In **mfe.c**. Enable the use of interrupt mechanism (refer to Stefan or Pierre for further information).

14 appendix G: Frequently Asked Questions

Names

Feel free to ask questions to one of us (Stefan, Pierre).

14.1 Why the CAMAC frontend generate a core dump (linux)?

If you're not using a Linux driver for the CAMAC access, you need to start the CAMAC frontend application through the task launcher first. See dio task or mcnaf task.

This task laucher will grant you access permission to the IO port mapped to your CAMAC interface.

14.2 Where does Midas log file resides?

As soon as any midas application is started, a file **midas.log** is produce. The location of this file depends on the setup of the experiment.

- if **exptab** is present and contains the experiment name with the corresponding directory, this is where the file **midas.log** will reside.
- if the midas logger **mlogger task** is running the **midas.log** will be in the directory pointed by the "Data Dir" key under the /logger key in the ODB tree.
- Otherwise the file **midas.log** will be created in the current directory in which the Midas application is started.

14.3 How do I protected my experiment from being controlled by aliases?

- Every experiment may have a dedicated password for accessing the experiment from the web browser. This is setup through the ODBedit program with the command **webpass**. This will create a **Security** tree under **/Experiment** with a new key **Web Password** with the encrypted word. By default Midas allows Full Read Access to all the Midas Web pages. Only when modification of a Midas field the web password will be requested. The password is stared as a cookie in the target web client for 24 hours See ODB /Experiment Tree.
- Other options of protection are described in ODB /Experiment Tree which gives to dedicated hosts access to ODB or dedicated programs.

14.4 Can I compose my own experimental web page?

- Only under 1.8.3 though. You can create your own html code using your favorite HMTL editor. By including custom Midas Tags, you will have access to any field in the ODB of your experiment as well as the standard button for start/stop and page switch. See Utilities mhttpd task Custom page.

14.5 How do I prevent user to modify ODB values while the run is in progress?

- By creating the particular **/Experiment/Lock when running/** ODB tree, you can include symbolic links to any odb field which needs to be set to **Read Only** field while the run state is on. See ODB /Experiment Tree.

14.6 Is there a way to invoke my own scripts from the web?

- Yes, by creating the ODB tree **/Script** every entry in that tree will be available on the Web status page with the name of the key. Each key entry is then composed with a list of ODB field (or links) starting with the executable command followed by as many arguments as you wish to be passed to the script. See ODB /Script Tree.

14.7 I've seen the ODB prompt displaying the run state, how do you do that?

- Modify the **/System/prompt** field. The "S" is the trick.

```
Fri> odb -e bnmr1 -h isdaq01
[host:expt:Stopped]/cd /System/
[host:expt:Stopped]/Systemls
Clients
Client Notify                  0
```

```
Prompt                          [%h:%e:%S]%p
Tmp
[host:expt:Stopped]/System
[host:expt:Stopped]/Systemset prompt [%h:%e:%S]%p>
[host:expt:Stopped]/System>ls
Clients
Client Notify                   0
Prompt                          [%h:%e:%S]%p>
Tmp
[host:expt:Stopped]/System>set Prompt [%h:%e:%s]%p>
[host:expt:S]/System>set Prompt [%h:%e:%S]%p>
[host:expt:Stopped]/System>
```

14.8

I've setup the alarm on one parameter in ODB but I can't make it trigger?

- The alarm scheme works only under **ONLINE"**. See ODB /RunInfo Tree for **Online Mode**. This flag may have been turned off due to analysis replay using this ODB. Set this key back to 1 to get the alarm to work again.

14.9

How do I ...

- ...