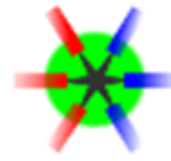# Data Acquisition Backbone Core

User Manual

J.Adamczewski-Musch, S.Linev, H.G.Essel
GSI Darmstadt,
Experiment Electronics Department

Document	Date	Editor	Revision	Comment
DABC-user	2009-03-10	Hans G.Essel	1.0.1	First scetch

Contents

Chapter 1

DABC User Manual: Overview

[user/user-overview.tex]

1.1   Outline of this manual

This *DABC* User Manual contains all information that is necessary to install and use the *DABC* framework.

Chapter 2, page 3 should be useful to understand the most commonly used terms of *DABC*.

Chapter 3, page 11 describes how to install the *DABC* packages on any linux machine, and how to set up the working environment. Additionally, some typical use cases and their configuration files are shown. The following chapters then give more detailed explanations how to operate in different modes with the *DABC* Java GUI:

Chapter 4, page 19 covers the general functionality of the GUI which is common for most applications. Especially, this is mostly sufficient to control a DAQ cluster purely with one or several *DABC* nodes.

Chapter 5, page 29 describes the *DABC* GUI in a mode to control a pure *MBS* data acquisition system without a native *DABC* node.

The application use case for a mixed DAQ cluster, both with *DABC* and *MBS* nodes, is treated in Chapter 6, page 37.

Chapter 7, page 43 describes the use case of a *DABC* builder network (BNET), both with and without using *MBS* .

Finally, Chapter 8, page 45 describes the use case of ROC front-ends.

However, the scope of the *DABC* User Manual does not contain detailed descriptions of the *DABC* framework architecture, the software mechanisms, and the example programs. These subjects are treated thouroughly in the *DABC* Programmer Manual.

1.2   Release Notes

1.2.1   Version 1.0.01 (10. March 2009)

1. Add IP multicast support in SocketTransport.
2. Add IB multicast support in verbs::Transport.

3. Possibility to add user-defined parameters directly in xml file - in Context/User section.
4. If Context/Run/copycfg = true, config file will be copied to working directory of specified node, useful for cluster without common file system.
5. Implement all-to-all and multicast tests in net-test application.
6. Bugfix several minor errors in Verbs plugin.
7. Bugfix: suppress output of scripts running from ssh (caused problems with GUI).
8. Bugfix: GUI: Register DIM service after full instantiation of parameter object.
9. Bugfix: GUI: Histogram drawer had uninitialized field.

1.2.2   Version 1.0.00 (26. February 2009)

These are the features of the first official release:

1. A Data Acquisition framework in C++ language for linux platforms with modular components for dataflow on multiple nodes.
2. Runtime environment with basic services for: threads, event handling, memory management, command execution, configuration, logging, error handling
3. Plug-in mechanism for user defined DAQ applications
4. Plug-in mechanism for a control system. Features a finite state machine logic and parameters for monitoring and configuration. The default implementation is based on the DIM protocol (http://dim.web.cern.ch/dim)
5. Java GUI to operate the standard DIM control system of *DABC*/*MBS*. Fully generic evaluating *DABC* process variables, but extendable by user written components.
6. Contains a sub-framework to set-up distributed event builder networks (BNET)
7. Supports TCP/IP and InfiniBand/verbs networks for data transport
8. Supports formats and readout of GSI's standard DAQ system *MBS* (Multi Branch System). May also write data into *MBS* listmode format, and may emulate MBS socket data servers. Additionally, MBS systems can be controlled by the DABC GUI.

Chapter 2

DABC User Manual: Introduction

[user/user-introduction.tex]

2.1 About *DABC*

The Data Acquisition Backbone Core *DABC* is a Data Acquisition (DAQ) framework with modular components for dataflow on multiple nodes. It provides a C++ runtime environment with all basic services, such as: threads and event handling, memory management, command execution, configuration, logging and error handling. User written DAQ applications can be run within this environment by means of a plug-in mechanism.

DABC contains a sub-framework with additional interfaces to set-up distributed event builder networks. As transport layers for such networks, *tcp/ip* and *InfiniBand/verbs* are supported.

DABC supports by default the data formats and readout connections of GSI's standard DAQ system *MBS* (Multi Branch System). It may also write data files with the *MBS* `*.lmd` format, and it may emulate *MBS* data server sockets, such as *stream* or *transport* servers.

The *DABC* control system features a finite state machine logic and parameters for monitoring and configuration. The current implementation is based on the DIM protocol [2], other implementations could replace this one. A generic Java GUI is provided to operate this standard DIM control system. This GUI may also control *MBS* systems which support the DIM communication. It is extendable by user written components.

2.2 Introduction

The the following sections we give a short introduction to the main components and terms of *DABC*. Figure 2.1 should be helpful.

2.2.1 Modules

All processing code runs in module objects. There are two general types of modules: synchronous and asynchronus. A synchronous module may block for longer time waiting for data and must therefore run in its own computing thread. Asynchronous modules must never block. Therefore several of them may run as a chain in one single thread.
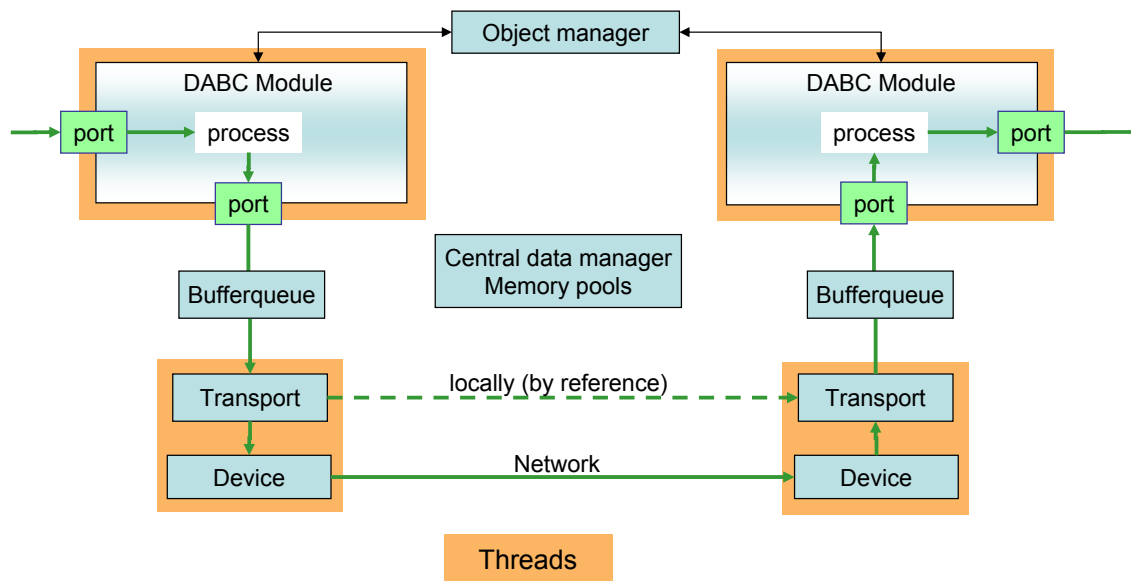
Figure 2.1: Components and data flow.

2.2.1.1   Synchronous module

Each synchronous module is executed by a dedicated working thread. The thread executes a method *MainLoop()* with arbitrary code, which may block the thread. In blocking calls of the framework (resource or port wait), optionally command callbacks may be executed implicitly. A timeout may be set for all blocking calls; this can optionally throw an exception when the time is up. On timeout with exception, either the *MainLoop()* is left and the exception is then handled in the framework thread; or the *Main-Loop()* itself catches and handles the exception. On state machine commands (e.g. Halt or Suspend, see Programmer Manual section **??**), the blocking calls are also left by exception, thus putting the mainloop thread into a stopped state.

2.2.1.2   Asynchronous module

Several asynchronous modules may be run by a shared working thread. The thread processes an event queue and executes appropriate callback functions of the module that is the receiver of the event. Events are fired for data input or output, command execution, and if a requested resource (e.g. memory buffer) is available. **The callback functions must never block the working thread**. Instead, the callback must return if further processing requires to wait for a requested resource. Therefore each callback function must check the available resources explicitly whenever it is entered.

2.2.2   Commands

A module may register ***Command*** objects and may define command actions by overwriting a virtual command callback method *ExecuteCommand*.

2.2.3 Parameters

A module may register *Parameter* objects. Parameters are accessible by name; their values can be monitored and optionally changed by the controls system. Initial parameter values can be set from XML configuration files.

2.2.4 Manager

The modules are organized and controlled by one manager object which is persistent independent of the application's state.

The manager is an object manager that owns and keeps all registered basic objects into a folder structure.

Moreover, the manager defines the interface to the control system. This covers registering, sending, and receiving of commands; registering, updating, unregistering of parameters; error logging and global error handling.

The manager receives and dispatches commands to the destination modules where they are queued and eventually executed by the modules threads (see Programmer Manual section **??**). The manager has an independent manager thread, used for manager commands execution, parameters timeout processing and so on.

2.2.5 Memory and buffers

Data in memory is referred by *Buffer* objects. Allocated memory areas are kept in *MemoryPool* objects. In general case a buffer contains a list of references to scattered memory fragments from memory pool. Typically a buffer references exactly one segment. Buffers may have an empty list of references. In addition, buffers can be supplied with a custom headers.

The buffers are provided by one or several memory pools which preallocate reasonable memory from the operating system. A memory pool may keep several sets, each set for a different configurable memory size.

A new buffer may be requested from a memory pool by size. Depending on the module type and mode, this request may either block until an appropriate buffer is available, or it may return an error value if it can not be fulfilled. The delivered buffer has at least the requested size, but may be larger. A buffer as delivered by the memory pool is contiguous.

Several buffers may refer to the same fragment of memory. Therefore, the memory as owned by the memory pool has a reference counter which is incremented for each buffer that refers to any of the contained fragments. When a consumer frees a buffer object, the reference counters of the referred memory blocks are decremented. If a reference counter becomes zero, the memory is marked as "free" in the memory pool.

2.2.6 Ports

Buffers are entering and leaving a module through *Port* objects. Each port has a buffer queue of configurable length. A module may have several input, output, or bidirectional ports. The ports are owned by the module.

2.2.7   Transport

Outside the modules the ports are connected to ***Transport*** objects. On each node, a transport may either transfer buffers between the ports of different modules (local data transport without copy), or it may connect the module port to a data source or sink (e. g. file i/o, network connection, hardware readout).

In the latter case, it is also possible to connect ports of two modules on different nodes by means of a transport instance of the same kind on each node (e. g. *InfiniBand verbs* transport connecting a sender module on node A with a receiver module on node B via a *verbs* device connection).

2.2.8   Device

A transport belongs to a ***Device*** object of a corresponding type that manages it. Such a device may have one or several transports. The threads that run the transport functionality are created by the device. If the Transport implementation shall be able to block (e. g. on socket receive), there can be only one transport for this thread.

A device object usually represents an I/O component (e. g. network card). There may be several device objects of the same type in an application scope. The device objects are owned by the manager singleton; transport objects are owned and managed by their corresponding device.

A device is persistent independent of the connection state of the transport. In contrast, a transport is created during *connect()* or *open()* and deleted during *disconnect()* or *close()*, respectively.

A device may register parameters and define commands. This is the same functionality as available for modules.

2.2.9   Application

The ***Application*** is a singleton object that represents the running application of the DAQ node (i. e. one per system process). It provides the main configuration parameters and defines the runtime actions for the different control system states (see Programmer Manual section **??**). In contrast to the Manager implementation that defines a framework control system (e.g. DIM, EPICS), the Application defines the experiment specific behaviour of the DAQ.

2.3   Controls and configuration

2.3.1   Finite state machine

The running state of the DAQ system is ruled by a Finite State Machine [4] on each node of the cluster. The manager provides an interface to switch the application state by the external control system. This may be done by calling state change methods of the manager, or by submitting state change commands to the manager (from GUI).

Some of the application states may be propagated to the active components (modules, device objects), e.g. the Running or Ready state which correspond to the activity of the thread. Other states like Halted or Failure do not match a component state; e.g. in Halted state, all modules are deleted and thus do not have an internal state. The granularity of the control system state machine is not finer than the node application.

There are 5 generic states to treat all set-ups:
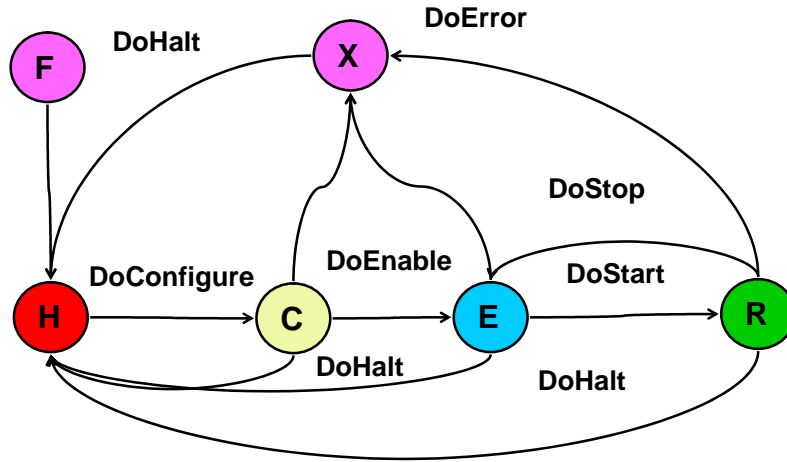
Figure 2.2: The finite state machine as defined by the manager.

Halted : The application is not configured and not running. There are no modules, transports, and devices existing.

Configured : The application is mostly configured, but not running. Modules and devices are created. Local port connections are done. Remote transport connections may be not all fully connected, since some connections require active negotiations between different nodes. Thus, the final connecting is done between Configured and Ready.

Ready : The application is fully configured, but not running (modules are stopped).

Running : The application is fully configured and running.

Failure : This state is reached when there is an error in a state transition function. Note that a run error during the Running state would not lead to Failure, but rather to stop the run in a usual way (to Ready).

The state transitions between the 5 generic states correspond to commands of the control system for each node application:

DoConfigure : between Halted and Configured. The application plug-in creates application specific devices, modules and memory pools. Application typically establishes all local port connections.

DoEnable : between Configured and Ready. The application plug-in may establish the necessary connections between remote ports. The framework checks if all required connections are ready.

DoStart : between Ready and Running. The framework automatically starts all modules, transport and device actions.

DoStop : between Running and Ready. The framework automaticall stops all modules, transport and device actions, i.e. the code is suspended to wait at the next appropriate waiting point (e.g. begin of *MainLoop()*, wait for a requested resource). Note: queued buffers are not flushed or discarded on Stop !

DoHalt : switches states Ready , Running , Configured, or Failure to Halted. The framework automatically deletes all registered objects (transport, device, module) in the correct order.

2.3.2   Commands

The control system may send (user defined) commands to any component (module , device, application). Execution of these commands is independent of the state machine transitions.

2.3.3   Configuration and monitoring

The configuration is done using parameter objects. On application startup time, the configuration system may set the parameters from a configuration file (e.g. XML configuration files). During the application lifetime, the control system may change values of the parameters by command. However, since the set up is changed on DoConfigure time only, it may be forbidden to change true configuration parameters except when the application is Halted.

Otherwise, there would be the possibility of a mismatch between the monitored parameter values and the really running set up. However, the control system may change local parameter objects by command in any state to modify minor system properties independent of the configuration set up (e.g. switching on debug output, change details of processing parameters).

The current parameters may be stored back to the XML file.

Apart from the configuration, the control system may use local parameter objects for monitoring the components. When monitored parameters change, the control system is updated by interface methods of the manager and may refresh the GUI representation. Programmer Manual Chapter **??**, page **??** will explain the usage of parameters for configuration in detail.

2.4   Package and library organisation

The complete system consists of several packages.

2.4.1   Core system

The Core system package defines all base classes and interfaces and implements basic functionalities for object organization, memory management, thread control, and event communication.

2.4.2   Control and configuration system

Depends on the **Core system**. Defines functionality of state machine, command transport, parameter monitoring and modification. Implements the connection of configuration parameters with a database (i.e. a file in the trivial case). Interface to the **Core system** is implemented by subclass of *Manager*.

Note that default implementations of state machine and a configuration file parser are already provided by the **Core system**.

2.4.3   Plug-in packages

Plug-in packages may provide special implementations of the core interface classes:
Device, *Transport*, *Module*, or *Application*. Usually, these classes are made available to the system by means of a corresponding *Factory* that is automatically registered in the *Manager* when loading the plug-in library.
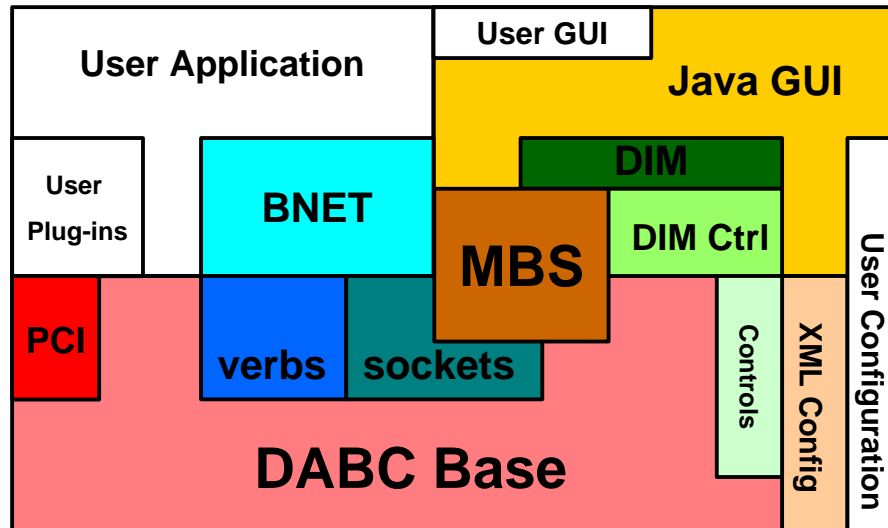
Figure 2.3: Schematic view of the distributed *DABC* components (coloured) and user specific extensions (white)

When installed centrally, the **Plugin packages** are kept in subfolders of the `$DABCSYS/plugins` directory. Alternatively, the **Plugin packages** may be installed in a user directory and linked against the **Core system** installation.

2.4.3.1 Bnet package

This package depends on the **Core system** and implements modules to cover a generic event builder network. It defines interfaces (virtual methods) of the special Bnet modules to implement user specific code in subclasses. The Bnet package provides a factory to create specific Bnet modules by class name. It also provides application classes to define generic functionalities for worker nodes and controller nodes. These may be used as base classes in further **Application packages**.

2.4.3.2 Transport packages

Depend on the **Core system**, and may depend on external libraries or hardware drivers. Implement *Device* and *Transport* classes for specific data transfer mechanism, e.g. **verbs** or **tcp/ip socket**. May also implement *Device* and *Transport* classes for special data input or output. Each transport package provides a factory to create a specific device by class name.

However, the most common transport implementations are put directly to the **Core system**, e.g. local memory, or socket transport; the corresponding factory is part of the **Core system** then.

2.4.4   Application packages

They depend on the **Core system**, and may depend on several **transport packages**, on the **Bnet package**, or other plugin packages. They may also depend on other application packages. Application packages provide the actual implementation of the core interface class *Application* that defines the set-up and behaviour of the DAQ application in different execution states. This may be a subclass of specific existing application. Additionally, they may provide experiment specific *Module* classes.

When installed centrally, the Application packages are kept in subfolders of the `$DABCSYS/applications` directory. Alternatively, an Application package may be installed in a user directory and linked against the **Core system** installation and the required **Plugin packages**.

2.4.5   Distribution contents

The DABC distribution contains the following packages:

Core system : This is plain C++ code and independent of any external framework.
Bnet plugin : Depends on the core system only.
Transport plugins : Network transport for *tcp/ip* sockets and *InfiniBand* verbs. Additionally, transports for GSI *Multi Branch System MBS* connections (socket, filesystem) is provided. Optionally, example transport packages may be installed that illustrate the readout of a *PCIe* board, or data taking via *UDP* from an external readout controller (ROC) board.
Control and configuration system : The general implementation is depending on the DIM framework only. DIM is used as main transport layer for commands and parameter monitoring. On top of DIM, a generic record format for parameters is defined. Each registered command exports a self describing command descriptor parameter as DIM service. Configuration parameters are set from XML setup files and are available as DIM services.
GUI A generic controls GUI using the DIM record and command descriptors is implemented with Java. It may be extendable with user defined components.
Application packages : some example applications, such as:
  - Simple *MBS* event building
  - Bnet with switched *MBS* event building
  - Bnet with random generated events

Chapter 3

DABC User Manual: Setup

[user/user-setup.tex]

3.1 Installing *DABC*

When working at the GSI linux cluster, the *DABC* framework is already installed and will be maintained by people of the gsi EE department. Here *DABC* needs just to be activated from any GSI shell by typing
. dabclogin (dot space). In this case, please skip this installation section and proceed with following section 3.2, page 12 describing the set-up of the user environment.

However, if working on a separate DAQ cluster outside GSI, it is mandatory to install the *DABC* software from scratch. Hence the *DABC* distribution is available for download at http://dabc.gsi.de. It is provided as a compressed tarball of sources dabc_vn.m.ss.tar.gz where n m and ss are version numbers. The following steps describe the recommended installation procedure:

1. **Unpack this *DABC* distribution** at an appropriate installation directory, e. g. :
   ```
   cd /opt/dabc;
   tar zxvf dabc_v1.0.00.tar.gz
   ```
   This will extract the archive into a subdirectory which is labelled with the current version number like /opt/dabc/dabc_v1.0.00. This becomes the future *DABC* system directory.
2. **Prepare the *DABC* environment login script**: A template for this script can be found at
   ```
   scripts/dabclogin.sh
   ```
   - Edit the DABCSYS environment according to your local installation directory. This is done in the following lines:
     ```
     export DABCSYS=/opt/dabc/dabc_1_0.00
     ```
   - Specify correct location of your JAVA installation. This is done in the lines (shown here an example, make sure to get the path where the include directory is located):
     ```
     export JAVA_HOME=/usr/lib/jvm
     ```
   - Copy the script to a location in your global $PATH for later login, e. g. /usr/bin. Alternatively, you may set an *alias* to the full pathname of dabclogin.sh in your shell profile.
3. Execute the just modified login script in your shell to set the environment:
   ```
   . dabclogin.sh
   ```
   This will set the environment for the compilation.
4. Change to the *DABC* installation directory and start the build:
   ```
   cd $DABCSYS
   make
   ```
   This will compile the *DABC* framework and install a suitable version of DIM in a subdirectory of $DABCSYS/dim.

After succesful compilation, the *DABC* framework installation is complete and can be used from any shell after invoking `. dabclogin.sh` The next sections 3.2, page 12 and 3.3, page 13 will describe further steps to set-up the *DABC* working environment for each user.

3.2   Set-up the *DABC* environment

Once the general *DABC* framework is installed on a system, still each user must "activate" the environment and do further preparations to work with it.

1. Execute the *DABC* login script in a linux shell to set the environment. At GSI linux installation, this is done by
   ```
   . dabclogin
   ```
   For the user installation as described in above section 3.1, page 11, by default the script is named
   ```
   . dabclogin.sh
   ```
   The login script will already enable the *DABC* framework for compilation of user written components. Additionally, the general executable `dabc_run` now provides the *DABC* runtime environment and may be started directly for simple "batch mode" applications on a single node.
   However, further preparations are necessary if *DABC* shall be used with DIM control system and GUI.
2. Open a dedicated shell on the machine that shall provide the DIM name server, e. g.
   ```
   ssh nsnode.cluster.domain
   export DIM_DNS_NODE=nsnode.cluster.domain
   . dabclogin.sh
   dimDns &
   dimDid &
   ```
   to launch the DIM name server. This is done **once** at the beginning of the DAQ setup; usually the DIM name server needs not to be shut down when *DABC* applications terminate. The DID is useful for inspecting DIM services.
3. Set the DIM name server environment variable in any *DABC* working shell (e. g. the shell that will start the dabc gui later):
   ```
   . dabclogin.sh
   export DIM_DNS_NODE=nsnode.cluster.domain
   ```
4. Now the *DABC* GUI can be started in such prepared shell by typing `dabc`, (or `mbs` for a plain *MBS* gui, resp.). See below in gui section.

To operate a *DABC* application one should create a dedicated working directory to keep all relevant files:

- Setup files for *DABC* (XML).
- Log files (text).

The following section 3.3, page 13 gives a general description of the setup file syntax.

The GUI may run on a machine with no access to the *DABC* working directory, e. g. a windows PC. Therefore the GUI setup files may use a different working directory, containing:

- Data files for startup panels (XML).
- Configuration files for GUI (XML).

These configuration files for the GUI are described in more detail in Chapter 4, page 19.

Of course both setups, for the *DABC* application and the GUI, can be put into one working directory if the GUI has access to it.

3.3   *DABC* **setup file**

The setup file is an XML file in a *DABC*-specific format, which contains values for some or all configuration parameters of the system.

3.3.1   Setup file example

Let's consider this simple but functional configuration file:

```
<?xml version="1.0"?>
<dabc version="1">
  <Context host="localhost" name="Generator">
    <Run>
      <lib value="libDabcMbs.so"/>
      <func value="InitMbsGenerator"/>
    </Run>
    <Module name="Generator">
      <Port name="Output">
        <OutputQueueSize value="5"/>
        <MbsServerPort value="6000"/>
      </Port>
    </Module>
  </Context>
</dabc>
```

This is an example XML file for an MBS generator, which produces MBS events and provides them to an *MBS transport* server. This use case is described further in section 8.1, page 45.

Other examples of *DABC* setup files can be found in the sections 6.1, page 37, 7.1, page 43, and 7.2, page 44 of this manual.

3.3.2   Basic syntax

A *DABC* configuration file should always contain <dabc> as root node. Inside the <dabc> node one or several <Context> nodes should exists. Each <Context> node represents the *application context* which runs as independent executable. Optionally the <dabc> node can have <Variables> and <Defaults> nodes, which are described further in the following sections 3.3.5, page 14 and 3.3.6, page 15.

3.3.3   Context

A <Context> node can have two optional attributes:

"host"   host name, where executable should run, default is "localhost"
"name"   application (manager), default is the host name.

Inside a <Context> node configuration parameters for modules, devices, memory pools are contained. In the example file one sees several parameters for the output port of the generator module.

3.3.4   Run arguments

Usually a <Context> node has a <Run> subnode, where the user may define different parameters, relevant for running the *DABC* executable:

lib name of a library which should be loaded. Several libraries can be specified.

func name of a function which should be called to create modules. This is an alternative to instantiating a subclass of *dabc::Application* (compare section **??**, page **??**)

runfunc function name to run some sequence of operations (start, stop, reconfigure) over application. Useful for batch mode

port ssh port number of remote host

user account name to be used for ssh (login without password should be possible)

init init script, which should be called before dabc application starts

test test script, which is called when test sequence is run by run.sh script

timeout ssh timeout

debugger argument to run with a debugger. Value should be like "gdb -x run.txt –args", where file run.txt should contain commands "r bt q".

workdir directory where *DABC* executable should start

debuglevel level of debug output on console, default 1

logfile filename for log output, default none

loglevel level of log output to file, default 2

DIM_DNS_NODE node name of DIM dns server, used by DIM controls implementation

DIM_DNS_PORT port number of DIM dns server, used by DIM controls implementation

cpuinfo instantiate *dabc::CpuInfoModule* to show CPU and memory usage information. Value must be >= 0. If 0, only two parameters are created, if 15 - several ratemeters will be created.

parslevel level of pars visibility for control system, default 1

3.3.5   Variables

In the root node <dabc> one can insert a <Variables> node which may contain definitions of one or several variables. Once defined, such variables can be used in any place of the configuration file to set parameter values. In this case the syntax to set a parameter is:

```
<ParameterName value="${VariableName}"/>
```

It is allowed to define a variable as a combination of text with another variable, but neither arithmetic nor string operations are supported.

Using variables, one can modify the example in the following way:

```
<?xml version="1.0"?>
<dabc version="1">
  <Variables>
    <myname value="Generator"/>
    <myport value="6010"/>
  </Variables>
  <Context name="Mgr${myname}">
    <Run>
      <lib value="libDabcMbs.so"/>
      <func value="InitMbsGenerator"/>
    </Run>
    <Module name="${myname}">
```

```
        <SubeventSize value="32"/>
        <Port name="Output">
           <OutputQueueSize value="5"/>
           <MbsServerPort value="${myport}"/>
        </Port>
     </Module>
   </Context>
</dabc>
```

Here context name and module name are set via `myname` variable, and mbs server socket port is set via `myport` variable.

There are several variables which are predefined by the configuration system:

- DABCSYS - top directory of *DABC* installation
- DABCUSERDIR - user-specified directory
- DABCWORKDIR - current working directory
- DABCNUMNODES - number of <Context> nodes in configuration files
- DABCNODEID - sequence number of current <Context> node in configuration file

Any shell environment variable is also available as variable in the configuration file to set parameter values.

3.3.6   Default values

There are situations when one needs to set the same value to several similar parameters, for instance the same queue length for all output ports in the module. One possible way is to use syntax as described above. The disadvantage of such approach is that one must expand the XML file to set each queue length explicitly from the appropriate variable; so in case of a big number of ports the file will be very long and confusing to the user.

Another possibility to set several parameters at once consists in **wildcard rules** using "*" or "?" symbols. These can be defined in a <Defaults> node:

```
<?xml version="1.0"?>
<dabc version="1">
  <Variables>
    <myname value="Generator"/>
    <myport value="6010"/>
  </Variables>

  <Context name="Mgr${myname}">
    <Run>
      <lib value="libDabcMbs.so"/>
      <func value="InitMbsGenerator"/>
    </Run>
    <Module name="${myname}">
        <SubeventSize value="32"/>
        <Port name="Output">
           <MbsServerPort value="${myport}"/>
        </Port>
    </Module>
  </Context>
  <Defaults>
    <Module name="*">
```

```
        <Port name="Output*">
            <OutputQueueSize value="5"/>
        </Port>
    </Module>
  </Defaults>
</dabc>
```

In this example for all ports which names begin with the string "Output", and which belong to any module, the output queue length will be 5. A wildcard rule of this form will be applied for all contexts of the configuration file, i. e. by such rule we set the output queue length for all modules on all nodes. This allows to configure a big multi-node cluster with a compact XML file.

Another possibility to set default value for some parameters - create parameter with the same name in parent object. Here word **create** is crutial - one should use *CreateParInt()* method in module constructor - it is not enough just put additional tag in xml file. For instance, one can create parameter "MbsServerPort" in generator module and than MBS server transport, created for output port, will use that value for as default server port number.

3.4   Installation of additional plug-ins

Apart from the *DABC* base package, there may be additional plug-in packages for specific use cases. Generally, these plug-in packages may consist of a **plugins** part and an **applications** part. The *plugins* part offers a library containing new components (like ***Devices***, ***Transports***, or ***Modules***). The *applications* part mostly contains the XML setup files to use these new components in the *DABC* runtime environment; however, it may contain an additonal library defining the *DABC* ***Application*** class.

As an example, we may consider a plug-in package for reading out data from specific PCIe hardware like the Active Buffer Board *ABB* [3]. This package is separately available for download at http://dabc.gsi.de and described in detail in chapter **??**, page **??** of the *DABC* programmer's manual.

There are principally two different ways to install such separate plug-in packages: Either within the general DABCSYS directory as part of the central *DABC* installation, as described in following section 3.4.1, page 16. Or at an independent location in a user directory, as described in section 3.4.2, page 17.

3.4.1   Add plug-in packages to $DABCSYS

This is the recommended way to install a plug-in package if this package should be provided for all users of the *DABC* installation. A typical scenario would be that an experimental group owns dedicated DAQ machines with system manager priviliges. In this case, the plugin-package may be installed under the same account as the central *DABC* installation (probably, but not necessarily even the root account). The new plug-in package should be directly installed in the $DABCSYS directory then, with the following steps:

1. Download the plug-in package tarball, e. g. `abb1.tar.gz`
2. Call the `dabclogin.sh` script of the *DABC* installation (see section user-env)
3. Copy the downloaded tarball to the $DABCSYS directory and unpack it there:
   ```
   cp abb1.tar.gz $DABCSYS
   cd $DABCSYS
   tar zxvf abb1.tar.gz
   ```
   This will extract the new components into the appropriate `plugins` and `applications` folders below $DABCSYS.

4. Build the new components with the top Makefile of `$DABCSYS`:
   `make`

5. To work with the new components, the configuration script(s) of the *applications* part should be copied to the personal workspace of each user (see section 3.3, page 13). For the *ABB* example, this is found at
   `$DABCSYS/applications/bnet-test/SetupBnetIB-ABB.xml`

3.4.2 Plug-in packages in user directory

This is the case when *DABC* is installed centrally at the fileserver of an institute, and several experimental groups shall use different plug-ins. It is also the recommended way if several users want to modify the source code of a plug-in library independently without affecting the general installation.

The new plug-in package should be installed in a user directory then, with the following steps:

1. Download the plug-in package tarball, e. g. `abb1.tar.gz`
2. Create a directory to contain your additional *DABC* plugin packages:
   `mkdir $HOME/mydabcpackages`
3. Call the `dabclogin.sh` script of the *DABC* installation (see section user-env)
4. Copy the downloaded tarball to the `$DABCSYS` directory and unpack it there:
   `cp abb1.tar.gz $HOME/mydabcpackages`
   `cd $HOME/mydabcpackages`
   `tar zxvf abb1.tar.gz`
   This will extract the new components into the appropriate `plugins` and `applications` folders below the working directory.
5. To build the *plugins* part, change to the appropriate package plugin directory and invoke the local Makefile, e. g. for the *ABB* example:
   `cd $HOME/mydabcpackages/plugins/abb`
   `make`
   This will create the corresponding plug-in library in a subfolder denoted by the computer architecture, e. g. :
   `$HOME/mydabcpackages/plugins/abb/x86_64/lib/libDabcAbb.so`
6. For some plug-ins, there may be also small test executables with different Makefiles in subfolder `test`. These can be optionally build and executed independent of the *DABC* runtime environment.
7. The *DABC* working directory for the new plug-in will be located in subfolder
   `applications/plugin-name`
   For the *ABB* example, the application will set up a builder network with optional Active Buffer Board readouts, so this is at
   `$HOME/mydabcpackages/applications/bnet-test`
   As in this example, there may be an additional library to be build containing the actual ***Application*** class. This is done by invoking the Makefile within the directory:
   `cd $HOME/mydabcpackages/applications/bnet-test`
   `make`
   Here the application library is produced directly on top of the working directory:
   `$HOME/mydabcpackages/applications/bnet-test/libBnetTest.so`
8. The actual locations of the newly build libraries (plugins, and optionally applications part) has to be edited in the <lib> tag of the corresponding *DABC* setup-file (here: `SetupBnetIB-ABB.xml`). The default set-up examples in the plug-in packages assume that the library is located at `$DABCSYS/lib`, as it is in the alternative installation case as described in section 3.4.1, page 16.

Chapter 4

DABC User Manual: GUI

[user/user-gui.tex]

4.1 GUI Guide lines

The current *DABC* GUI is written in Java using the DIM software as communication layer. The standard part of the GUI described here may be extended by application specific parts. How to add such extensions is described in the programmer's manual. Typically they are started as prompter panels via buttons in the main GUI menu.

The standard part builds a set of panels (windows) according the parameters the DIM servers offer. Only services from one single DIM name server (node name specified as shell variable DIM_DNS_NODE) defining a name space can be processed. See 5.3.1, page 34 for preparations.

The GUI needs no file access to the *DABC* working directory. However, user must have ssh (or rsh) access to the *DABC* (or *MBS*) master node. Currently the GUI must run under the same account as the *DABC*. In monitoring mode (no commands) the GUI may run under different account. Master node must have remote access to all worker nodes. The user's ssh settings must enable remote access without prompts.

The layout of the GUI can be adjusted to individual needs. It is strongly recommended to save these settings to see the same layout after a restart of the GUI. The GUI can be restarted any time. *DABC* and *MBS* systems continue without GUI.

4.2 GUI Panels
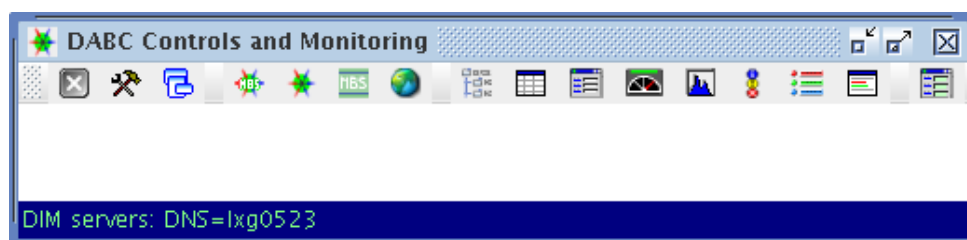
Figure 4.1: Main toolbar buttons.

Fig. 4.1, page 19 shows the main menu of *DABC* (minimal view). The GUI as it comes up is divided

in three major parts: one sees on top a toolbar with icon buttons. Most of these open other windows. The dark line at the bottom shows a list of active DIM servers. The other windows are placed in the white middle pane. The functions of the buttons and the invoked panels is described in the next sections. Depending on the application some buttons may be not seen, additional ones may show up. If one does not work with *MBS* plug-ins the control panels for *MBS* are of cause not useful.

Fig. 4.2, page 20 shows a more typical view of a running *DABC*. In general, all panels (including the
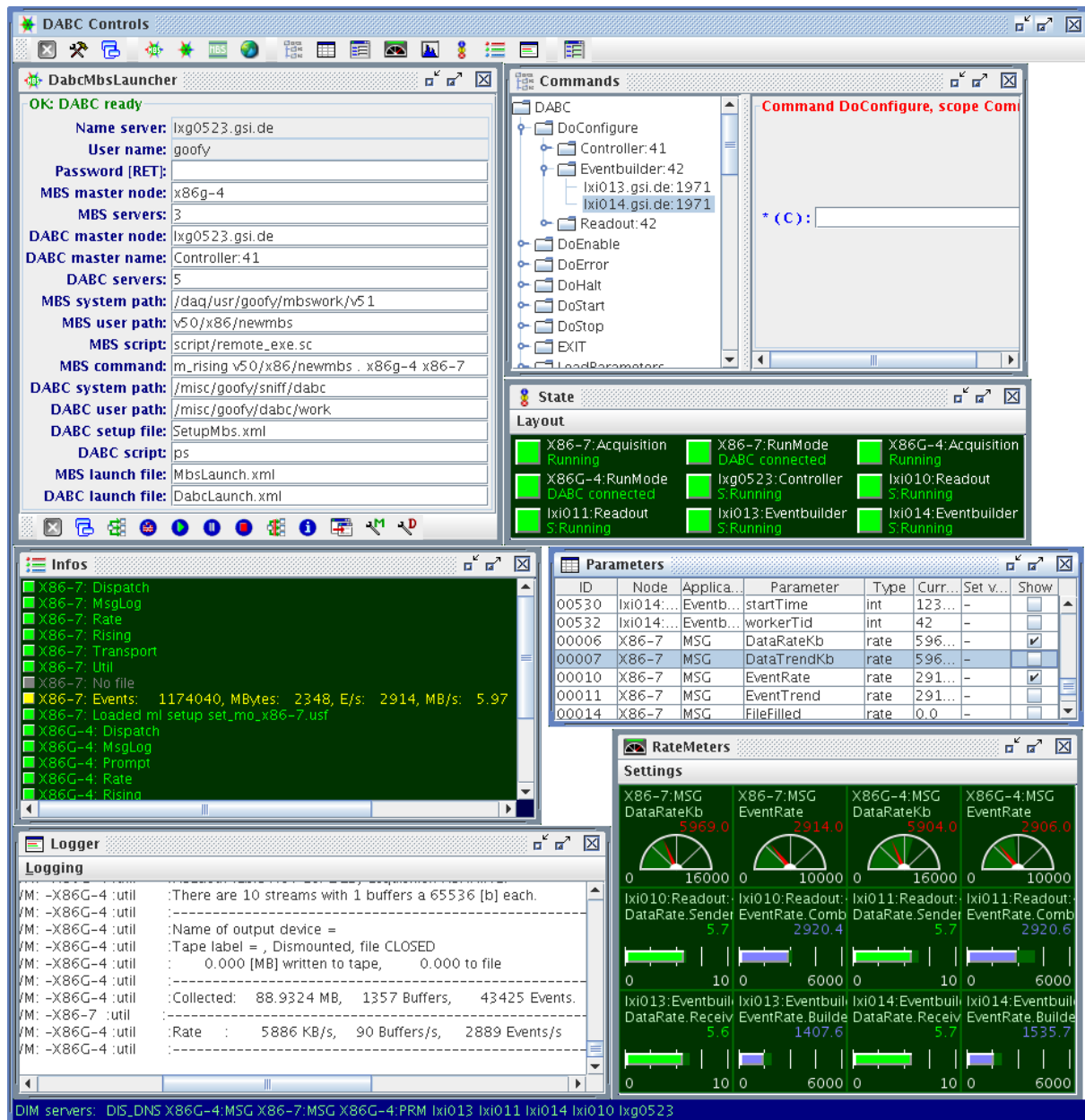
Figure 4.2: More typical full screen view.

GUI itself) can be closed and reopened any time.

4.2.1 Main *DABC* GUI buttons

Quit GUI. Will prompt (RET will quit). The *DABC* will continue to run. The GUI may be started

anywhere again. In case you saved the layout (recommended, see 4.3, page 27) and you start the GUI from the same directory it will look pretty much the same as you left it.

Test, shell script

Save settings: window layout, record attributes, parameter selection filters. Details see 4.3, page 27. Note that the content of the control panels must be saved by similar buttons in these panels.

Open *DABC MBS* control panel, see 6.1.3, page 39.

Open *DABC* control panel, see 4.2.2, page 21.

Open *MBS* control panel, see 5.1.2, page 29.

Refresh. All parameters and commands are removed. Rebuild DIM service list from DIM name server. Parameters and Commands are sorted alphabetically by name. All panels are updated. In normal operation there is no need to refresh manually.

Open command panel (4.2.6, page 23).

Open parameter table (4.2.7, page 24).

Open parameter selection panel (4.2.7.1, page 24).

Open rate meter panel (4.2.8, page 25).

Open histogram panel (4.2.8, page 25).

Open state panel (4.2.8, page 25).

Open info panel (4.2.8, page 25).

Open log panel (4.2.8, page 25).

Eventually one might see additional icons from application panels (this one is only an example).

The three control panels (*DABC*, *MBS*, combined *DABC* and *MBS*) are used depending on the application to be controlled. Eventually an application provides additional specific control panels.

4.2.2 *DABC* control panel

The standard *DABC* control panel is shown in 4.3, page 22. As mentioned already some applications may provide their own control panels like the *MBS* applications (see section 5.1.2, page 29). But most of the buttons are very common. From left to right they startup a system, configure it, start data taking, pause data taking, stop tasks, shut down. At the very left we see a save button, at the right a shell execution button. Values are read from file `DabcControl.xml` (default, may be saved/restored to/from other file, see 4.3, page 27).

```
<?xml version="1.0" encoding="utf-8"?>
<DabcLaunch>
<DabcMaster prompt="DABC Master" value="node.xxx.de" />
<DabcName prompt="DABC Name" value="Controller:41" />
<DabcUserPath prompt="DABC user path" value="myWorkDir" />
<DabcSystemPath prompt="DABC system path" value="/dabc" />
<DabcSetup prompt="DABC setup file" value="SetupDabc.xml" />
<DabcScript prompt="DABC Script" value="ps" />
```

Figure 4.3: DABC controller panel.

```
<DabcServers prompt="%Number of needed DIM servers%" value="5" />
</DabcLaunch>
```

DabcMaster: Node where the master controller shall be started. Can be one of the worker nodes.
DabcName: A unique name inside *DABC* of the system.
DabcUserPath: User working directory. The GUI does not need to have access to the filesystem.
DabcSystemPath: Path where the *DABC* is installed.
DabcSetup: Setup file name.
DabcScript: Command to be executed in an ssh at the master node.
DabcServers: Number of workers and controllers. This information is minimum for the GUI to know
when all *DABC* nodes are up. The GUI waits until this number of DIM servers is up and running.
Note that this number must be consistent with the *DABC* setup file used.

The name server name is translated from shell environment variable DIM_DNS_NODE, the user name
from shell environment variable USER. Password can be chosen when the first remote shell script is
executed (which itself is protected by user password). All following commands then need this password.

4.2.2.1 *DABC* controller buttons

Save panel settings to the file Control file. If you choose a name different from the default you must
set a shell variable to it to get the values from that file (see 4.3, page 27).

Startup all tasks. Executes a *DABC* script `dabcstartup.sc` via `ssh` on the master node under
user name. Then it waits until the number of DIM servers expected are announced. A progress panel
pops up during that time (see 4.2.3, page 23). When the servers are up the main GUI Update is triggered
building all panels from scratch according the parameters offered by the servers.

Configure. Executes state transition command Configure on master node and waits for the transition.
All plug-in components are created. Then execute Enable. Waits until all workers go into Ready state.
Now the *DABC* is ready to run. Triggers the main GUI Update.

Start acquisition. Executes Start command. All components go into running state Running.

Pause acquisition. Executes Stop command. All components go into standby state Ready.

■ Halt acquisition. Executes Halt command. This closes all plug-ins. States go into Halted. Next must be shut down or configure.

▦ Exit all processes by EXIT commands. After 2 seconds trigger the main GUI Update.

▨ Shut down all processes on all nodes by script. This is the hard shut down.

D ssh shell script execution on master node.

4.2.3   Action in progress
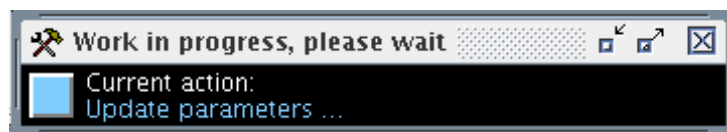
Figure 4.4: Launching progress.

When starting up, configure or shut down the GUI has to wait until the front-ends have completed the action. During that time a progress window similar to the one shown in Fig. 4.4, page 23 pops up. Please wait until the popup disappears.

4.2.4   *MBS* control panel

To control and monitor a stand-alone *MBS* system a dedicated control panel is provided by the *MBS* application. This panel is described in the *MBS* section 5.1.2, page 29.

4.2.5   Combined *DABC* and *MBS* control panel

To control and monitor *MBS* front-ends with *DABC* event builders a dedicated control panel is provided by the *MBS* application. This panel is described in the *MBS* section 6.1.3, page 39.
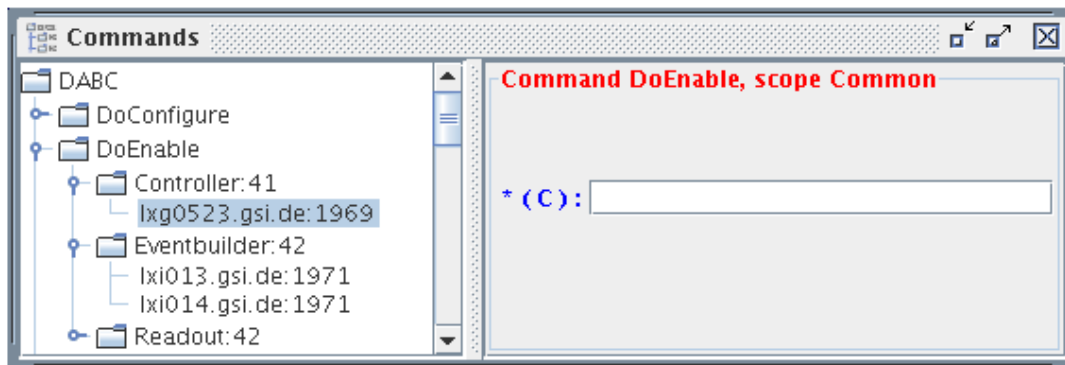
4.2.6   Command panel

The control system of *DABC* and/or the application specific plug-ins can define commands. These commands are encoded as DIM services including a full description of arguments. Therefore the GUI can build up at runtime a command tree and provide the proper forms for each command. Commands are executed in all components of *DABC*.

The *DABC* naming convention for commands and parameters defines four main name fields separated by slashes:

1. DIM server name space (example: DABC)
2. Node (example: lxg0523)
3. Application (example: Controller:41)
4. Name (example: doEnable)

Example: DABC/lxg0523/Controller:41/doEnable.   Fig. 4.5, page 24 shows on the left side the command tree. The tree is built from name, application, nodes. Double click (or RETURN) on a treenode

Figure 4.5: Command panel.

executes the command on all treenodes below. A click on a command opens at the right side the argument panel. Entering argument values and RETURN executes the command. In the example shown in the figure double click on doEnable would execute that command on three nodes. Double click on Eventbuilder would execute only on two nodes.

4.2.7 Parameter table

DABC parameters are DIM services as the commands. The naming convention is the same. The server providing parameters can be make them (no)visible and (un)changable. *DABC* defines some special parameter types having a data structure and a specific interpretation like a rate parameter having a value, limits, a color, and a graphic presentation. A rate parameter is assumed to be changed and updated regularly. The GUI displays these special parameters in dedicated panels. Parameters are used in all components of *DABC*. The central place for all parameters in the GUI is the parameter table as shwon


Figure 4.6: Parameter table.

in Fig. 4.6, page 24. The parameter table holds all parameters which are marked by the provider to be visible. The parameter values can be changed in the Set value column if no minus sign is there in which case the provider does not grant modification. The buttons in the Show column indicate if the parameter is shown in some graphics panel. It can be removed from or added to this panel by the buttons. The table can be ordered by columns (click on column header). The column width can be adjusted and is saved/restored by main save button (see 4.3, page 27).

4.2.7.1 Parameter selection

To get a more selective view on the parameters one can specify filters in the panel shown at the left side of Fig. 4.7, page 25. Text substrings for each of the four name fields can be specified as well as a selection
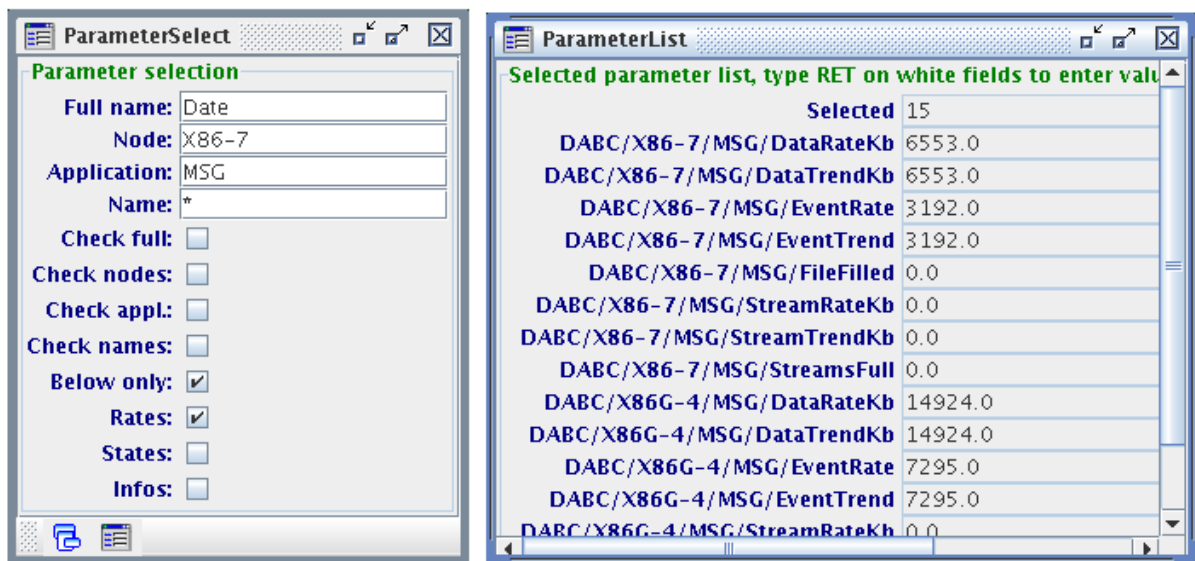
Figure 4.7: Parameter selection panel and selected parameter list.

of record types. Values can be saved (see 4.3, page 27). With the check boxes the filter function for each of these can (de)activated. The parameter list at the right window in Fig. 4.7, page 25 shows only the parameters matching all filters.

If the data field is white the parameter can be changed. This cannot be done in place because the parameter might be updated in the mean time. Instead press RETURN in the field. A prompter will pop up to enter the value.

4.2.8 Monitoring panels

As already mentioned the *DABC* provides definitions of special purpose DIM parameters. These *Records* can be recognized by the GUI and are handled in appropriate way. Currently there are

○ States
○ Rates
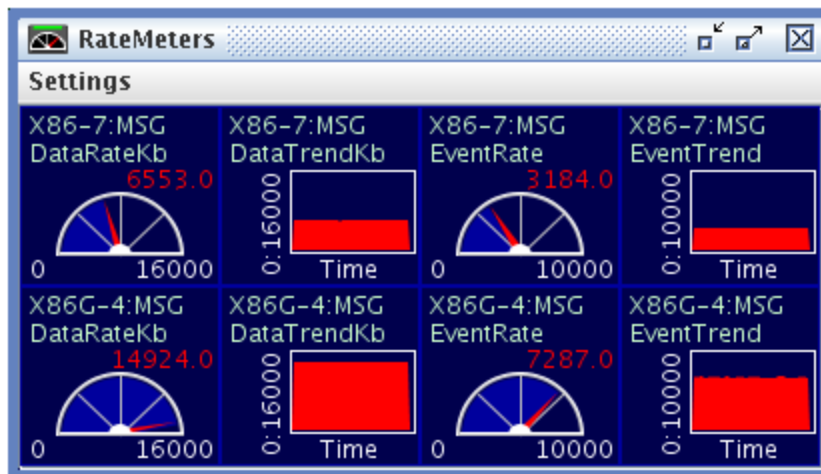○ Histograms
○ Infos

4.2.8.1 States
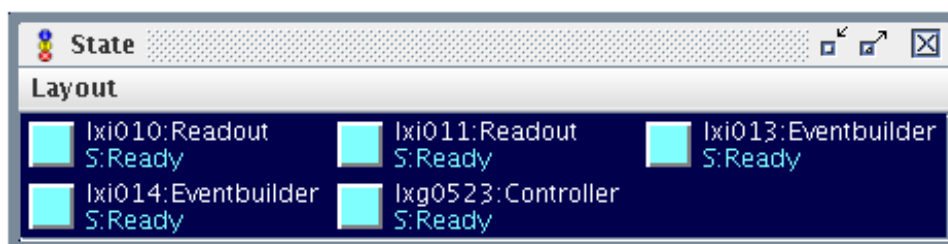
Figure 4.8: States.

States are records having a number for severity (0 to 4), a color, and a brief state description (see Fig. 4.8, page 25). Of cause the states of the *DABC* state machine are shown as states. Application plug-ins
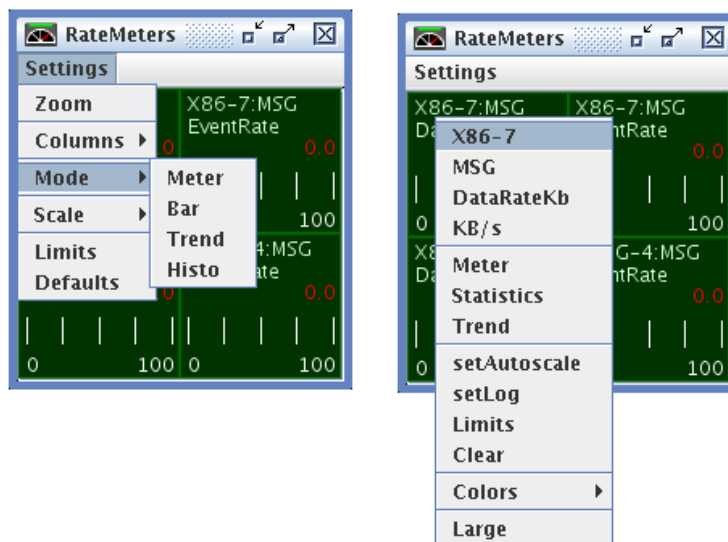
may use this kind of records also for other information.

4.2.8.2   Rate meters

Figure 4.9: Rates.

All rate meters are displayed in the meter panel, Fig.4.9, page 26. Meters can be removed in the parameter table (See Fig. 4.6, page 24) with the Show buttons like the other graphical parameters. Saving the setup, the visibility will be preserved.


Figure 4.10: Steering menus.

On the left side in Fig. 4.10, page 26 the Settings menu is shown. It affects all items in the panel. One can Zoom (toggle between large and normal view), change the number of columns, change the display mode, toggle Autoscale, and set limits (applied to all meters).

Besides that each individual item can be adjusted by right mouse button. The context menu is shown on the right. All changes done individually are changing the defaults! The global changes can be overwritten by these defaults. All settings are saved with the setup and restored on GUI startup (see 4.3, page 27).
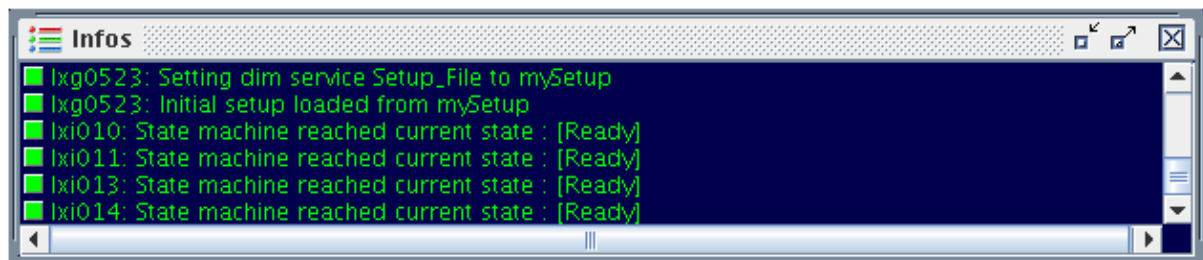
4.2.8.3   Histograms

Histogram panels are handled in pretty much the same way as the rate meters.   All histograms are
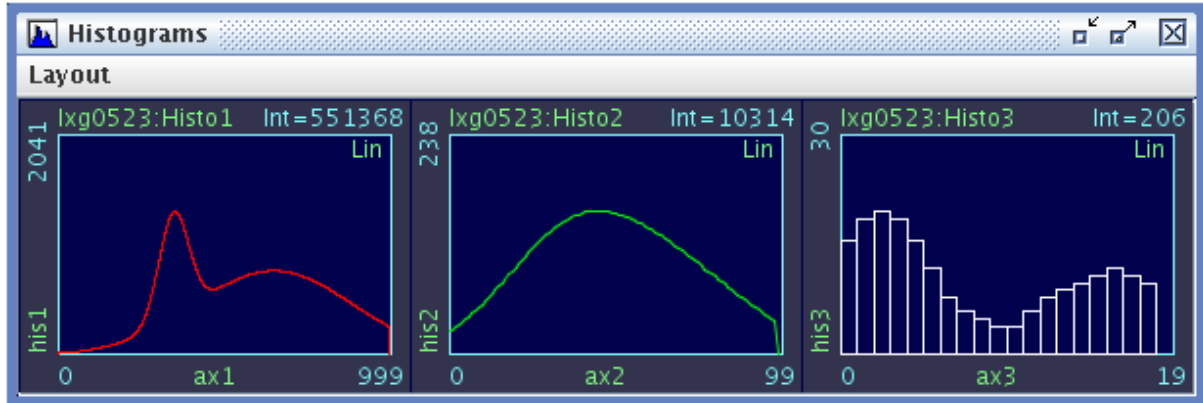
Figure 4.11: Histograms.

displayed in the histogram panel, Fig.4.11, page 27. Histograms can have arbitrary size set in Layout menu.

4.2.8.4   Information

Figure 4.12: Info.

Information records mainly display one line of text with a color (see Fig. 4.12, page 27).

4.2.8.5   Logging window

Fig. 4.13, page 28 show the logging window.

4.3   GUI save/restore setups

There are several setups which can be stored in XML files and are retrieved when the *xGUI* is started again. The file names can be specified by shell variables.

DABC_CONTROL_DABC : Values of *DABC* control panel. Saved by button in panel.
     Default `DabcControl.xml`. Filename in panel itself.
DABC_CONTROL_MBS : Values of *MBS* control panel. Saved by button in panel.
     Default `MbsControl.xml`. Filename in panel itself.
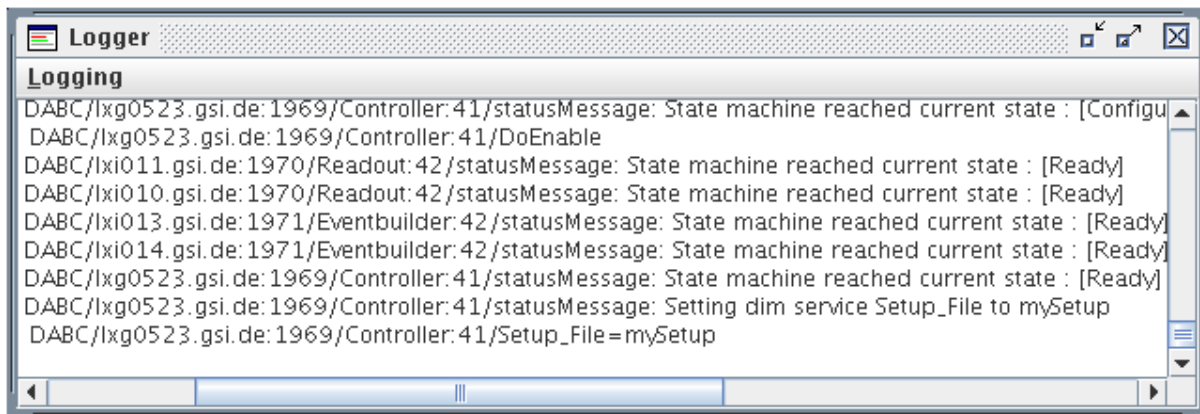
Figure 4.13: Logging.

DABC_RECORD_ATTRIBUTES : Attributes of records. Saved by main save button.
    Default `Records.xml`.
DABC_PARAMETER_FILTER : Values of parameter filter panel. Saved by main save button.
    Default `Selection.xml`.
DABC_GUI_LAYOUT : Layout of frames. Saved by main save button.
    Default `Layout.xml`.

Chapter 5

DABC User Manual: *MBS* GUI

[user/user-gui-mbs.tex]

5.1 *MBS* event building

5.1.1 *MBS* setup

Any *MBS* system can be controlled by the *DABC* GUI. It can run in two operation modes: with *MBS* event builder or *DABC* event builder (see 6.1, page 37). The first case means a standard *MBS* system.

To control a standard *MBS* nothing has to be done by the user on the *MBS* side. The node running the GUI must get granted `rsh` access at least to the *MBS* node where the prompter shall run. **Note**, however that in the user's *MBS* startup file (typically `startup.scom`) the `m_daq_rate` task must be started as last task (this is probably the case already). This task calculates the rates. The GUI waits for this task after execution of the startup file. Because *MBS* has no states there is no other way to know when the startup has finished. Of cause, the *MBS* itself must have been built with the DIM option (since version v5.1). Central log file is written as usual. Optionally one can provide a text file with specifications which parameters shall be published by DIM (see 5.3.1, page 34).

For the standard *MBS* control one needs no *DABC* installation. The GUI jar file is sufficient. DIM must be installed. See installation guide on the download page.

5.1.2 *MBS* control panel

Fig. 5.1, page 30 shows the panel to be used to control a standard *MBS*. The values are restored from file `MbsControl.xml` (default, may be saved to other file, see 4.3, page 27). The file `MbsControl.xml` can be created easily in the GUI itself by filling the input fields of the control panel and save.

```
<?xml version="1.0" encoding="utf-8"?>
<MbsLaunch>
<MbsMaster prompt="MBS Master" value="node-xx" />
<MbsUserPath prompt="MBS User path" value="myMbsDir" />
<MbsSystemPath prompt="MBS system path" value="/mbs/v51" />
<MbsStartup prompt="MBS startup" value="startup.scom"/>
<MbsShutdown prompt="MBS shutdown" value="shutdown.scom"/>
<MbsCommand prompt="Script command" value="whatever command" />
<MbsServers prompt="%Number of needed DIM servers%" value="3" />
```

Figure 5.1: MBS controller.

```
</MbsLaunch>
```

MbsMaster : Lynx node where the *MBS* prompter is started.

MbsUserPath : *MBS* user working directory. The GUI need not to have access to that filesystem.

MbsSystemPath : Path on Lynx where the *MBS* is installed. GUI needs no access to this path.

MbsStartup : The user specific *MBS* startup command procedure, typically `startup.scom`, located on user path.

MbsShutdown : The user specific *MBS* shutdown command procedure, typically `shutdown.scom`, located on user path.

MbsCommand : With RET an *MBS* command in executed (on current node). The shell script button executes this string as `rsh` command on master node.

MbsServers : Number of nodes plus prompter. This information is minimum for the GUI to know when all *MBS* nodes are up. The GUI waits until this number of DIM servers is up and running.

That file can be created from within the GUI in the *MBS* controller panel. Enter all values necessary, and store them.

5.1.2.1 *MBS* controller buttons

Save panel settings, see 4.3, page 27.

Execute script `prmstartup.sc` at master node. Starts prompter, dispatchers and message loggers and waits until they are up. Trigger the main Update. A progress panel pops up during that time (see 4.2.3, page 23).

Execute script `dimstartup.sc` at master node. Starts dispatcher and message logger for single node *MBS*. Trigger the main Update.

Configure. Execute user's *MBS* startup procedure in prompter (dispatcher). Wait for all `m_daq_rate` tasks are running. Trigger the main Update.

Start acquisition. Execute Start acquisition. Wait for all acquisition states go into Running.

Pause acquisition. Execute Stop acquisition. Wait for all acquisition states go into Stopped.

Halt acquisition. Execute user's *MBS* shutdown procedure in prompter. Prompter, dispatcher and message loggers should still be running.

Shut down all. Execute script `prmshutdown.sc` at master node. After 2 seconds trigger the main Update.

Show acquisition. Output in log panel.

Shell script executes command on master node.

5.1.3 *MBS* command panel
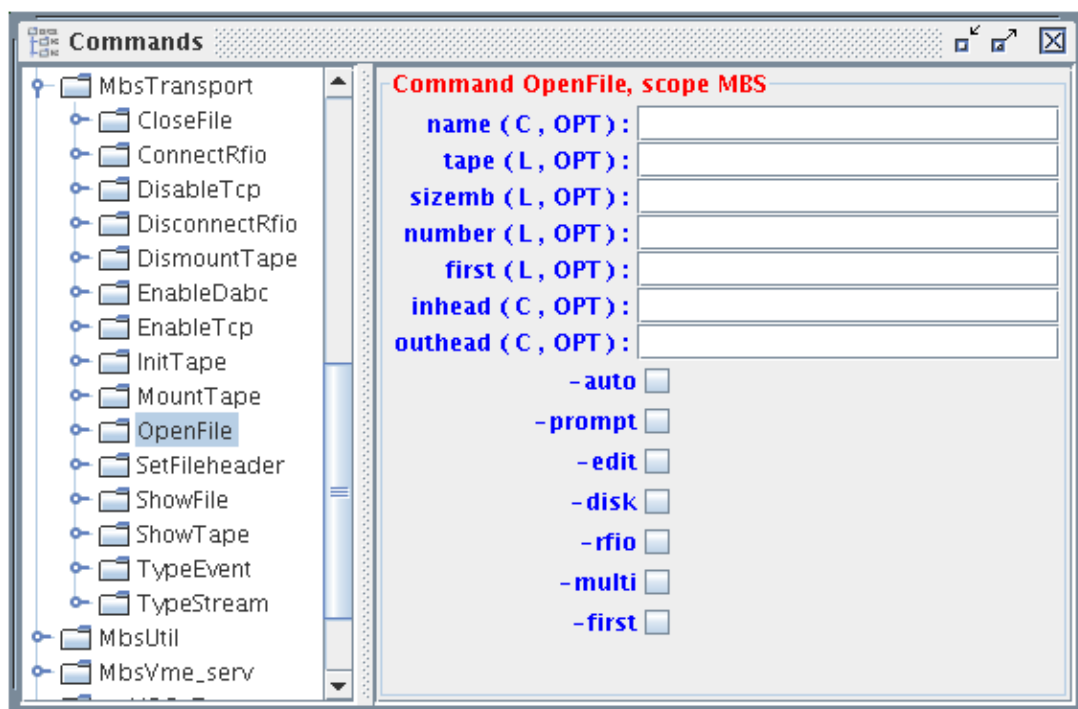
Figure 5.2: Command panel.

Fig. 5.2, page 31 shows on the left side the command tree. Double click (or RETURN) on a command executes the command. The top tree level is the executing *MBS* task, below that are the commands, and the master node (prompter node) is the only node below each command. However, command is sent to the prompter node, but executed on the current node which is displayed in the info panel (see Fig. 5.4, page 32). Click on a command opens at the right side the argument panel. Entering argument values and RETURN executes the command.

Only the *MBS* commands of the running tasks are shown. Fig. 5.3, page 32 shows that only dispatcher and prompter are up and therefore only their commands are seen. Fig. 5.4, page 32 shows in addition the commands of util and transport after configuration.
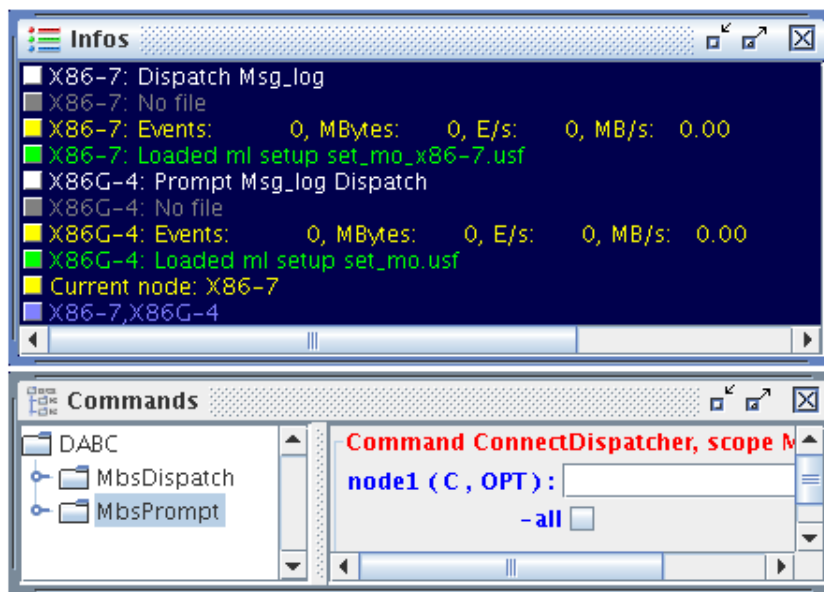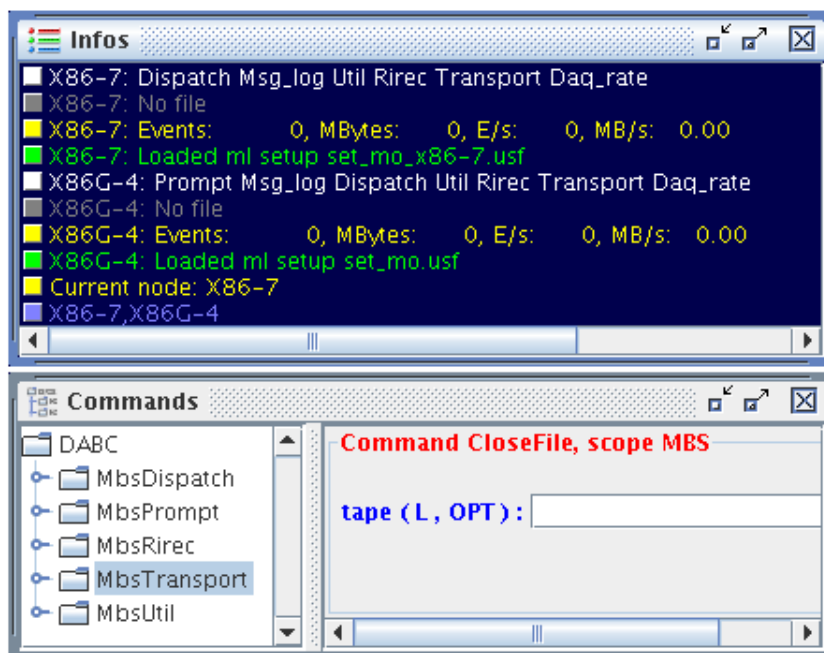
Figure 5.3: Info and command panel.

Figure 5.4: Info and command panel.

5.2 *MBS* **DIM parameters**

5.2.1 *MBS* **states**

Acquisition/State Running | Stopped
BuildingMode/State Delayed | Immediate
EventBuilding/State Working | Suspended
FileOpen/State File open | File closed
RunMode/State DABC connected | MBS to DABC | Transport client | MBS standalone
SpillOn/State Spill ON | Spill OFF

TriggerMode/State Master | Slave

5.2.2   *MBS* rates

MSG/DataRateKb KByte/s
MSG/DataTrendKb KBytes/s as trend
MSG/EventRate Events/s
MSG/EventTrend Events/s as trend
MSG/EvSizeRateB Event size sample in bytes
MSG/EvSizeTrendB Event size sample in bytes
MSG/StreamRateKb Stream server Kbyte/s
MSG/StreamTrendKb Stream server Kbyte/s as trend
MSG/FileFilled File filled in percent
MSG/StreamsFull Number of full streams in percent
MSG/TriggerRate Trigger/s of readout tasks
MSG/TriggernnRate (nn=01...15) Trigger/s type nn of readout tasks

5.2.3   *MBS* histograms

Shown in histo window.

MSG/TrigCountHis Histogram with 16 channels for counts of trigger types (0 = total) as seen by the
readout task.
MSG/TrigRateHis Histogram with 16 channels for count rates of trigger types (0 = total) as seen by
the readout task.

5.2.4   *MBS* infos

Shown in info window.

MSG/eFile Name of file.
MSG/ePerform Events, MBytes, Events/s and MBytes/s.
MSG/eSetup Name of setup file loaded.
PRM/Current Current command execution node (master node only).
PRM/NodeList List of nodes (master node only).

5.2.5   *MBS* tasks

Task list is shown in info window (name slightly different):

Dispatch Msg_Log Read_Meb Collector Transport Event_Serv Util Read_Cam Esone_Serv Stream_Serv
Histogram Prompt Rate SMI Sender Receiver Asynch_Receiver Rising Time_Order Vme_Serv

5.2.6   *MBS* text

MSG/GuiNode Node where GUI runs
MSG/Date Date as written in file header
MSG/Run Run ID as written in file header
MSG/Experiment Experiment as written in file header
MSG/User Lynx user name as written in file header

MSG/Platform CPU platform

5.2.7 *MBS* numbers

MSG/BufferSize
MSG/Buffers collected so far.
MSG/Events collected so far.
MSG/FileMbytes written in file.
MSG/FlushTime
MSG/MBytes collected so far.
MSG/StreamKeep
MSG/StreamMbytes
MSG/StreamScale
MSG/StreamSync
MSG/UserVal_nn (nn=00...15) These values can be set in the user readout function.
MSG/TriggernnCount (nn=01...15) Trigger counts type nn of readout tasks.

5.3   Working directories

5.3.1 *MBS* configuration of DIM

Optional text file `dimsetup` in the *MBS* working directory specifies which rate meters, histograms or states shall appear in the GUI. Upper limits of the rate meters can be specified. This file can be copied from `$MBSROOT/set/dimsetup`. Only the parameters which are in this file are optional.

Note, that a file name of an open lmd file is only displayed when either FileOpen or FileFilled is selected for this node.

```
## This file controls the rate meter and state appearance.
## File name must be dimsetup and in the MBS working directory.
## The value numbers are the maximum values for rate meters
## Colons only if value is specified!
## Node names must be uppercase, * wildcards all

##========= All nodes:
##---- Rates:
* EventRate      : 10000.
#* EventTrend    : 10000.
* DataRateKb     : 16000.
#* DataTrendKb   : 16000.
#* StreamRateKb  : 16000.
#* StreamTrendKb : 16000.
#* EvSizeRateB   : 128.
#* EvSizeTrendB  : 128.
# ++ File filling status in percent, typically only on one node (transport)
#* FileFilled   :   100.
#* StreamsFull  :   100.
#* TriggerRate : 10000.
# ++ Trigger rates for the individual triggers: 01...15
#* Trigger01Rate : 10000.

##---- States:
```

```
# ++ Delayed or immediate event building:
* BuildingMode
# ++ Current eventbuilding running or suspended:
* EventBuilding
# ++ Shows spill signal:
#* SpillOn
# ++ Shows if file is open, typically only on one node (transport)
#* FileOpen
# ++ Show trigger master
#* TriggerMode

##---- User integers from daqst, 00...15
# can be set by f_ut_set_daqst_user(index,value);
#* UserVal_00
#* TriggerCount
# ++ Trigger counts for the individual triggers: 01...15
#* Trigger01Count

##---- Histograms
#* TrigCountHis
#* TrigRateHis

##======== Node XXX (uppercase)
#XXX EventRate   : 10000.
#XXX DataRateKb  : 16000.
#XXX FileOpen
#XXX FileFilled  :   100.
#XXX SpillOn
#XXX EventTrend  : 10000.
#XXX DataTrendKb : 16000.
#XXX TriggerMode
```

Chapter 6

DABC User Manual: *DABC* Application *MBS*

[user/user-app-mbs.tex]

6.1 *MBS* event building with *DABC*

In this case one *DABC* node reads data from several *MBS* nodes via *Transport* socket connections, and combines them into one *MBS* output event.

To run *MBS* front-ends with *DABC* nodes as event builders some modifications of the *MBS* setup files must be done. For the *DABC* side setup files must be provided.

6.1.1 *MBS* setup

When we want to use *DABC* nodes as event builders, we need a different setup on the *MBS* side. We assume that we have more than one *MBS* node. Such a multi-node system is controlled by an *MBS* prompter running on one node.

- The setup has to be changed such that all nodes run as if they are stand alone (this is done typically by setting COL_MODE to 0 in the usf setup file). That means that each node must run the Readout - Collector - Transport - Daq_rate chain. The *DABC* event builders connect to the transports.
- The *MBS* buffer size should be set to the stream size and the number of buffers per stream must be set to one.

6.1.2 *DABC* setup

On the *DABC* user working directory we need configuration files.

Summary of parameters:

MbsFileName File name for list mode data file (LMD). Overwritten by command.
MbsFileSizeLimit File closes when size is reached, and new file opens.
BufferSize Should match *MBS* buffer size.
MbsServerKind Transport | Stream.
MbsServerPort Port number transport (6000).
MbsServerName *MBS* node of transport.
NumInputs Number of *MBS* channels for one combiner.
DoFile Provide output file.
DoServer Provide server.

These parameters are used to configure an optional event generator:

NumSubevents
FirstProcId
SubeventSize
Go4Random

The following example configuration file `$DABCSYS/applications/mbs/Combiner.xml` shows how to configure one combiner module reading from two *MBS* transport servers. A simple setup looks like this:

```xml
<?xml version="1.0"?>
<dabc version="1">
  <Context host="localhost" name="MbsEb">
    <Run>
      <lib value="libDabcMbs.so"/>
      <func value="StartMbsCombiner"/>
    </Run>
    <Module name="Combiner">
        <NumInputs value="2"/>
        <DoFile value="false"/>
        <DoServer value="true"/>
        <BufferSize value="16384"/>
        <Port name="Input0">
           <MbsServerKind value="Transport"/>
           <MbsServerName value="X86-xx"/>
           <MbsServerPort value="6000"/>
        </Port>
        <Port name="Input1">
           <MbsServerKind value="Transport"/>
           <MbsServerName value="X86-yy"/>
           <MbsServerPort value="6000"/>
        </Port>
        <Port name="FileOutput">
           <OutputQueueSize value="5"/>
           <MbsFileName value="combiner.lmd"/>
           <MbsFileSizeLimit value="128"/>
         </Port>
        <Port name="ServerOutput">
           <MbsServerKind value="Stream"/>
         </Port>
    </Module>
  </Context>
</dabc>
```

We have one node (Context) with a simple run function *StartMbsCombiner()* that uses a single Module to do the event combination from two input Ports. The node names and other parameters of the external *MBS* connections are specified in the MbsServerName properties of these ports. Of course the *MBS* setup must match these definitions.

There are two output Ports in parallel here: A FileOutput that writes into a `*.lmd` file as specified in the property MbsFileName; and a ServerOutput that offers a standard *MBS* stream server for a monitoring program. A full description is in Programmer Manual section **??**, page **??**.

Now we can use the combined controller panel to startup *MBS* and *DABC*.

Figure 6.1: Combined DABC and MBS controller.

### 6.1.3   Combined *DABC* and *MBS* control panel

This panel shown in Fig. 6.1, page 39 is simply a superposition of the single ones. Here the Context name of the *DABC* node and the *DABC* setup file name must be specified. Number of *DABC* servers is one.

#### 6.1.3.1   Combined *DABC* and *MBS* controller buttons

Save panel settings, see 4.3, page 27.

Execute script `dabcstartup.sc` at *DABC* master node. Starts DIM servers. Execute script `prmstartup.sc` at *MBS* master node. Starts prompter, dispatchers and message loggers. Waits for all components (Sum of DIM servers) are running. A progress panel pops up during that time (see 4.2.3, page 23). If all components are up trigger the main Update.

Configure. Execute user's *MBS* startup procedure in prompter. Waits for all *MBS* `Daq_rate` tasks are running. Executes state transition command Configure on *DABC* master node and wait for the transition. All plug-in components are created. Then execute Enable. If all components are up trigger the main Update.

Start *MBS* acquisition, wait for all acquisition states Running, then execute *DABC* Start command. All components go into running state Running.

Pause acquisition. Execute *MBS* stop acquisition, wait for all acquisition states Stopped. Execute *DABC* Stop command. All components go into standby state Ready.

Halt acquisition. Executes *DABC* Halt command. This closes all plug-ins. States go into Halted. Execute

user's *MBS* shutdown procedure in prompter. Prompter, dispatcher and message loggers should still be running. Next must be shut down or configure. After two seconds trigger the main Update.

Shut down all. Execute EXIT command on all *DABC* nodes. Execute script `prmshutdown.sc` at *MBS* master node. After two seconds trigger the main Update.

MBS Show acquisition. Output in log panel.

Shell script for *MBS* master node.

Shell script for *DABC* master node.

6.2   *MBS* and *DABC* with Bnet

The following example configuration file `$DABCSYS/applications/bnet-mbs/SetupBnetMbs.xml` shows how to configure two *DABC* nodes reading from two *MBS* transport servers and two event builder nodes. Another node is used as controller.

The example setup file shows two techniques: first the use of XML variables which are set at the beginning, and can then be referenced, second the specification of default values for parameters of contexts or modules.

```xml
<?xml version="1.0"?>
<dabc version="1">
  <!-- Enter the values for specific setup -->
  <Variables>
     <ctrl value="lxg0523"/>
     <mbs1 value="r3g-100"/>
     <mbs2 value="r3g-101"/>
     <read1 value="lx1001"/>
     <read2 value="lx1002"/>
     <eb1 value="lx1003"/>
     <eb2 value="lx1004"/>
     <bufsize value="65536"/>
     <custport value="6000"/>
  </Variables>
  <Context host="${ctrl}" name="Controller">
    <Run>
      <lib value="${DABCSYS}/lib/libDabcBnet.so"/>
      <runfunc value="RunTestBnet"/>
    </Run>
    <Application class="bnet::Cluster">
       <NetDevice value="dabc::SocketDevice"/>
       <CtrlBuffer value="2048"/>
       <TransportBuffer value="${bufsize}"/>
       <NumEventsCombine value="1"/>
    </Application>
  </Context>
  <Context host="${read1}" name="Read1">
    <Application class="bnet::MbsWorker">
       <NumReadouts value="1"/>
       <Input0Cfg value="${mbs1}"/>
    </Application>
  </Context>
  <Context host="${read2}" name="Read2">
    <Application class="bnet::MbsWorker">
       <NumReadouts value="1"/>
```

```
        <Input0Cfg value="${mbs2}"/>
    </Application>
</Context>
<Context host="${eb1}" name="Build1"/>
<Context host="${eb2}" name="Build2"/>
<Defaults>
    <Context name="*">
      <Run>
        <logfile value="${Context}.log"/>
        <loglevel value="1"/>
        <cpuinfo value="1"/>
      </Run>
      <Module name="*">
        <Ratemeter name="Data*" lower="0" upper="20"/>
        <Ratemeter name="Event*" lower="0" upper="20000"/>
      </Module>
    </Context>
    <Context name="Read*">
      <Run>
        <lib value="libDabcBnet.so"/>
        <lib value="libDabcMbs.so"/>
        <lib value="libBnetMbs.so"/>
      </Run>
      <Application class="bnet::MbsWorker">
        <IsSender value="true"/>
        <ReadoutBuffer value="${bufsize}"/>
      </Application>
      <Module name="Combiner">
        <Port name="Input*">
          <MbsServerPort value="${custport}"/>
          <InputQueueLength value="20"/>
        </Port>
      </Module>
    </Context>
    <Context name="Build*">
      <Run>
        <lib value="libDabcBnet.so"/>
        <lib value="libDabcMbs.so"/>
        <lib value="libBnetMbs.so"/>
      </Run>
      <Application class="bnet::MbsWorker">
        <IsReceiver value="true"/>
        <IsFilter value="false"/>
        <EventBuffer value="${bufsize}"/>
      </Application>
    </Context>
</Defaults>
</dabc>
```

With the same setup of the two MBS nodes as before we can run this example. In the DABC control panel we only have to change the number of DABC servers (5), and the name of the setup file.

Chapter 7

DABC User Manual: *DABC* Application Bnet

[user/user-app-bnet.tex]

7.1   *DABC* eventbuilder network (BNET)

The full functionality of *DABC* is shown in the case that the DAQ uses an event building network (BNET), transferring the partial data from *n* readout nodes to *m* event building nodes, such that each event builder can work on the full detector data. This scenario is discussed in detail in chapter **??**, page **??** of the *DABC* programmer's manual. Appropriate configuration files can be found at `$DABCSYS/applications/bnet-test` directory. An example setup file `SetupBnet.xml` may look like this:

```xml
 <?xml version="1.0"?>
<dabc version="1">
  <Context host="localhost" name="Controller:41">
    <Run>
      <runfunc value="RunTestBnet"/>
    </Run>
    <Application class="bnet::Cluster">
       <NetDevice value="dabc::SocketDevice"/>
    </Application>
  </Context>
  <Context host="lxi009" name="Worker1:42"/>
  <Context host="lxi010" name="Worker2:42"/>
  <Context host="lxi011" name="Worker3:42"/>
  <Context host="lxi012" name="Worker4:42"/>
  <Defaults>
    <Context name="*">
      <Run>
        <logfile value="test${DABCNODEID}.log"/>
        <loglevel value="1"/>
        <lib value="libDabcBnet.so"/>
      </Run>
    </Context>
    <Context name="*Worker*">
      <Run>
        <lib value="${DABCSYS}/applications/bnet-test/libBnetTest.so"/>
```

```
      </Run>
      <Application class="bnet::TestWorker">
        <IsGenerator value="true"/>
        <IsSender value="true"/>
        <IsReceiver value="true"/>
        <NumReadouts value="4"/>
      </Application>
    </Context>
  </Defaults>
</dabc>
```

The setup of such BNET contains several <Context> nodes. Generally, the BNET has two types of nodes:

- One "Controller" node that has a master controller functionality, implemented in the <Application> of class "bnet::Cluster". The controller node must be specified at the *DABC* GUI setup to receive the direct cluster control commands, e. g. state machine transitions commands. In the *DABC* BNET framework, the controller also keeps a general parameter <NetDevice> for the data connection device of the entire DAQ cluster; this can be "dabc::SocketDevice" for tcp/ip, or "verbs::Device" for an *InfiniBand* cluster.
- Several "Worker" nodes of an experiment specific <Application>. They may be configured for different jobs in the BNET; this example provides an *Application* class "bnet::TestWorker" with some boolean parameters to define the functionality.

Note the usage of wildcards "*" in the <Context> names to define properties that should be valid for all nodes matching the pattern, e. g. the libraries to load, or the common application setup for all worker nodes. Here there are 4 workers which all produce random event data (enabled in <IsGenerator>), and all send their data to all others (enabled in <IsSender>). In parallel, they all receive data from the other workers to build the complete event (enabled in <IsReceiver>).

Such BNET setup is best started by means of the *DABC* GUI. The name of the controller <Context> node and the setup file name must be specified in the control panel of the GUI (see section 4.2.2, page 21). Then all nodes can be started just by the "Launch" button   . The configuration and run control of the nodes is done by the state machine buttons of the control panel.

7.2   *DABC* eventbuilder network (BNET) with *MBS*

A more realistic example of a BNET uses data which is read from *n* external *MBS* nodes, each connected to one *DABC* readout node, and transferred to *m* *DABC* eventbuilder nodes. Example file $DABCSYS/applications/bnet-mbs/SetupBnetMbs.xml shows the configuration for an *MBS* event building with 2 *DABC* readout nodes, connected with 2 *MBS* nodes each (simulated by *DABC* generator modules here), and 2 *DABC* event builder nodes. A detailed description of this setup is given in section **??**, page **??** of the *DABC* programmer manual. The usage of such configuration is similar to the BNET example as described above in section 7.1, page 43: The list of <Context> nodes (or the corresponding <Variables>, resp.) must be edited for the actual node names. Additionally the names of the *MBS* nodes for readout should be specified. Then the BNET setup may be launched and controlled by the *DABC* GUI.

Chapter 8

DABC User Manual: Application ROC

[user/user-app-roc.tex]

8.1 *DABC* as *MBS* data server

The use case here is that a single *DABC* node should provide data in the *MBS* event format on a server socket to be used by external analysis and monitoring programs like Go4 [1]. The event data can be simulated by a generator module. A practical case is to read data from any front-ends and format it like *MBS* events. This method is used by the ROC readout.

For the random event generator, such set-up looks like this:

```
<?xml version="1.0"?>
<dabc version="1">
  <Context host="lxi009" name="Server">
    <Run>
      <lib value="libDabcMbs.so"/>
      <func value="InitMbsGenerator"/>
    </Run>
    <Module name="Generator">
       <NumSubevents value="3"/>
       <FirstProcId value="77"/>
       <SubeventSize value="128"/>
       <Go4Random value="false"/>
       <BufferSize value="16384"/>
       <Port name="Output">
          <OutputQueueSize value="5"/>
          <MbsServerKind value="Stream"/>
          <MbsServerPort value="6006"/>
        </Port>
     </Module>
  </Context>
</dabc>
```

There is only one Context node, specified by the nodename, with one simple C function *InitMbsGenerator()* to run, and with one Module that produces the event data as specified in its parameters. The data server is specified by parameters of the Output Port: The tag MbsServerKind can be Stream or Transport to emulate either variant of the standard *MBS* server sockets. A complete description of this example can be found in Programmer Manual section **??**, page **??**. The setup files for standard *MBS* use cases can be found in directory

```
$DABCSYS/applications/mbs
```

8.2   ROC event building

A more practical use case is to prepare data as *MBS* events that was read by *DABC* from external front-end
hardware. This is shown with the setup-file for the readout controller ROC example (see the full description of this
example in Programmer Manual chapter **??**, page **??**):

```xml
<?xml version="1.0"?>
<dabc version="1">
<Context name="Readout">
  <Run>
    <lib value="libDabcMbs.so"/>
    <lib value="libDabcKnut.so"/>
    <logfile value="Readout.log"/>
  </Run>
  <Application class="roc::Readout">
    <DoCalibr value="0"/>
    <NumRocs value="3"/>
    <RocIp0 value="cbmtest01"/>
    <RocIp1 value="cbmtest02"/>
    <RocIp2 value="cbmtest04"/>
    <BufferSize value="65536"/>
    <NumBuffers value="100"/>
    <TransportWindow value="30"/>
    <RawFile value="run090.lmd"/>
    <MbsServerKind value="Stream"/>
    <MbsFileSizeLimit value="110"/>
  </Application>
 </Context>
</dabc>
```

Here the parameters are defined for the <Application> instance "roc::Readout" that controls the readout of 3 *ROC*
nodes via UDP, and combines the data into one *MBS* event by means of some internal *Modules*. Hence there is no
simple run function as before, the *DABC* runtime environment will call appropriate methods of the *Application* to
configure and run the set-up. Note that in this case the *MBS* data is not only provided to a stream server as defined
in <MbsServerKind>, but is also written to a *.lmd (list mode data) file which can be specified in application
parameter <RawFile>.

Both single node examples above do not require to be launched from the *DABC* GUI (although this is possible and
may be useful to monitor the data rates and actual parameters). They can be started directly from a shell by calling
the standard *dabc_run* executable with the configuration file name as argument: dabc_run Readout.xml.
This executable will load the specified libraries, create the application, configure it, and switch the system in the
Running state.

References

[1] Jörn Adamczewski-Musch, Hans Georg Essel, and Sergei Linev. The go4 system homepage, http://go4.gsi.de.

[2] Clara Gaspar. Dim - distributed information management system http://dim.web.cern.ch/dim/, 2008.

[3] Andreas Kugel, Wenxue Gao, and Guillermo Marcus. The active buffer board online documentation, http://cbm-wiki.gsi.de/cgi-bin/view/DAQ/ActiveBufferBoardV1, 2008.

[4] The Wikipedia. Finite state machine: http://en.wikipedia.org/wiki/State_machine, 2009.

Index