MBS GUI 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

MBS GUI Manual: Setup

[user/user-setup-mbsgui.tex]

1.1 Installing GUI

1.1.1 Linux at GSI

When working at the GSI linux cluster, the *DABC* framework including Java GUI 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
```

Note the space behind the dot. In this case, please skip this installation section.

However, if working on a separate node outside GSI, it is mandatory to install the *DABC* Java GUI software from scratch. Hence the *DABC* Java GUI distribution is available for download at http://dabc.gsi.de. It is provided as a compressed tarball of sources `dabcgui_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* Java GUI distribution at an appropriate installation directory, e. g. :
   ```
   cd /opt/dabcgui
   tar zxvf dabcgui_v1.0.00.tar.gz
   ```
   This will extract the archive into a subdirectory which is labelled with the current version number like `/opt/dabcgui/dabcgui_v1.0.00`. This is the GUI system directory.
2. Prepare the GUI environment login script: A template for this script can be found as
   ```
   guilogin.sh
   ```
   You need a DIM installation built with the JDIM=yes option. DIMDIR must be set in the script. In addition one may define for convenience
   ```
   alias dimdns=$DIMDIR/linux/dns
   alias dimdid=$DIMDIR/linux/did
   ```
3. 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 `guilogin.sh` in your shell profile.
4. Execute the just modified login script in your shell to set the environment:
   ```
   . guilogin.sh
   ```
   This will set the environment to run the GUI.

1.1.2   Windows

On Windows one needs the xgui.jar file only. The GUI itself can be started from a little BAT file like:

```
set HOST=%COMPUTERNAME%
set USER=%USERNAME%
set DIM_DNS_NAME=
set CLASSPATH=%CLASSPATH%;<dim path>/classes;<gui jar file>
set PATH=%PATH%;<dim path>/bin
java -Xmx200m xgui.xGui -mbs
```

To start GUI from desktop one has to create a Verknuepfung (RMB on file), then change Icon in Eigenschaften (RMB), then drag Verknuepfung to desktop.

1.1.3   Starting the GUI

When the installation is done, or at GSI after `dabclogin`, one more thing is necessary for DIM control and GUI: the DIM name server. It must run on a node accessible from all *MBS* nodes and the node the GUI shall run.

1. 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
   . guilogin.sh
   $DIMDIR/linux/dns &
   $DIMDIR/linux/did &
   ```
   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 the *MBS* and/or the GUI terminates. The DID is useful for inspecting DIM services.
2. Set the DIM name server environment variable in any working shell (e. g. the shell that will start the GUI later):
   ```
   . guilogin.sh
   export DIM_DNS_NODE=nsnode.cluster.domain
   ```
3. Now the *MBS* GUI can be started in such prepared shell by typing `mbs`.

The GUI may run on a machine with no access to the *MBS* working directory, e. g. a windows PC. Therefore the GUI setup files are typically on their own GUI 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 Chapters 2, page 3 and 3.1, page 13.

Chapter 2

MBS GUI Manual: General GUI

[user/user-gui.tex]

2.1 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 3.3.1, page 18 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.

2.2 GUI Panels
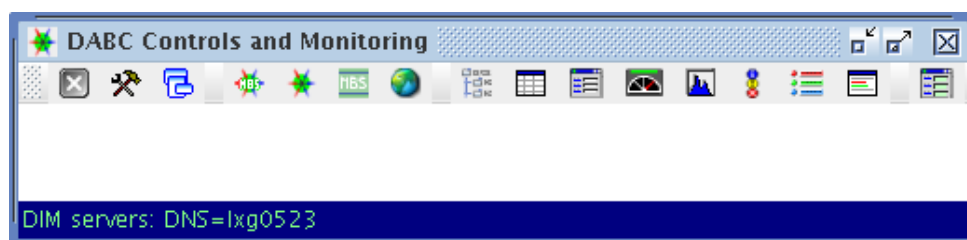
Figure 2.1: Main toolbar buttons.

Fig. 2.1, page 3 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. 2.2, page 4 shows a more typical view of a running *DABC*. In general, all panels (including the GUI
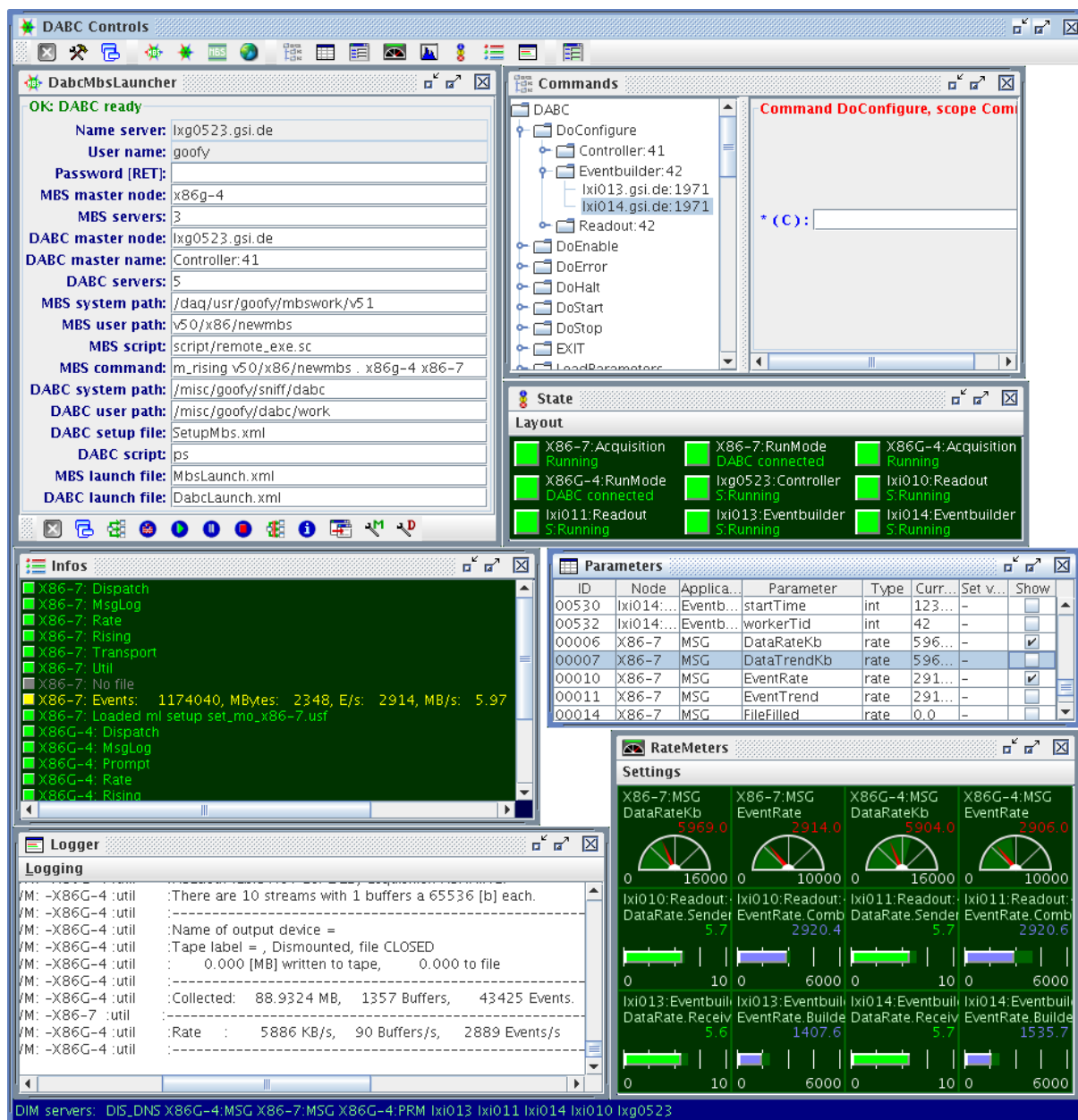
Figure 2.2: More typical full screen view.

itself) can be closed and reopened any time.

2.2.1   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 2.3, page 12) 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, command arguments, parameter selection filters. Details see 2.3, page 12. Note that the content of the control panels must be saved by similar buttons in these panels.

Open *DABC MBS* control panel, see **??**, page **??**.

Open *DABC* control panel, see 2.2.2, page 5.

Open *MBS* control panel, see 3.1.2, page 13.

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 (2.2.6, page 7).

Open parameter table (2.2.7, page 8).

Open parameter selection panel (2.2.7.1, page 9).

Open rate meter panel (2.2.8, page 9).

Open histogram panel (2.2.8, page 9).

Open state panel (2.2.8, page 9).

Open info panel (2.2.8, page 9).

Open log panel (2.2.8, page 9).

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.

2.2.2   *DABC* control panel

The standard *DABC* control panel is shown in 2.3, page 6. As mentioned already some applications may provide their own control panels like the *MBS* applications (see section 3.1.2, page 13). 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 2.3, page 12).

```xml
<?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" />
```
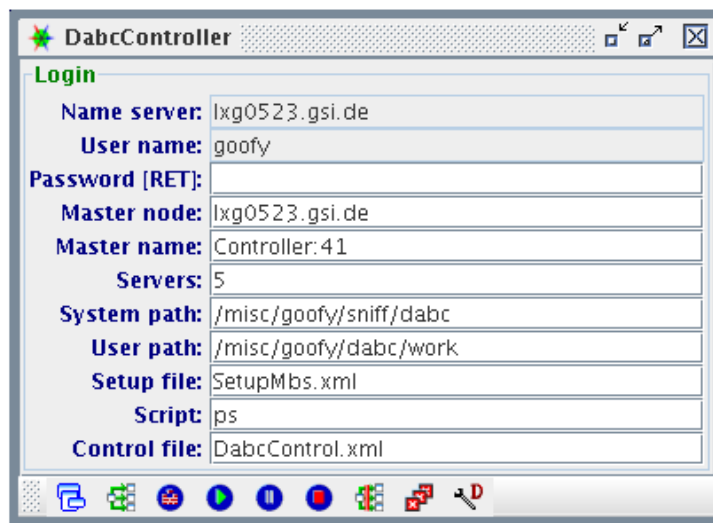
Figure 2.3: DABC controller panel.

```
<DabcSetup prompt="DABC setup file" value="SetupDabc.xml" />
<DabcScript prompt="DABC Script" value="ps" />
<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.

2.2.2.1   *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 2.3, page 12).

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 2.2.3, page 7). 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.

ssh shell script execution on master node.

2.2.3 Action in progress
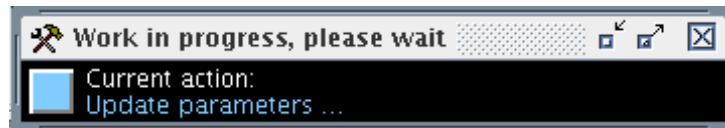
Figure 2.4: Launching progress.

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. 2.4, page 7 pops up. Please wait until the popup disappears.

2.2.4 *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 3.1.2, page 13.

2.2.5 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 **??**, page **??**.

2.2.6 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)
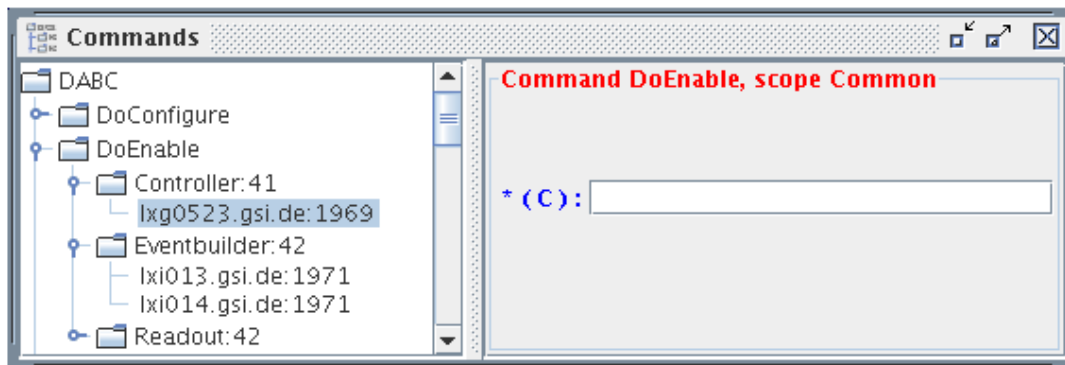
Figure 2.5: Command panel.

Example: DABC/lxg0523/Controller:41/doEnable. Fig. 2.5, page 8 shows on the left side the command tree. The tree is built from name, application, nodes. Double click (or RETURN) on a treenode 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.

2.2.7 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 2.6: Parameter table.

in Fig. 2.6, page 8. 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 2.3, page 12).
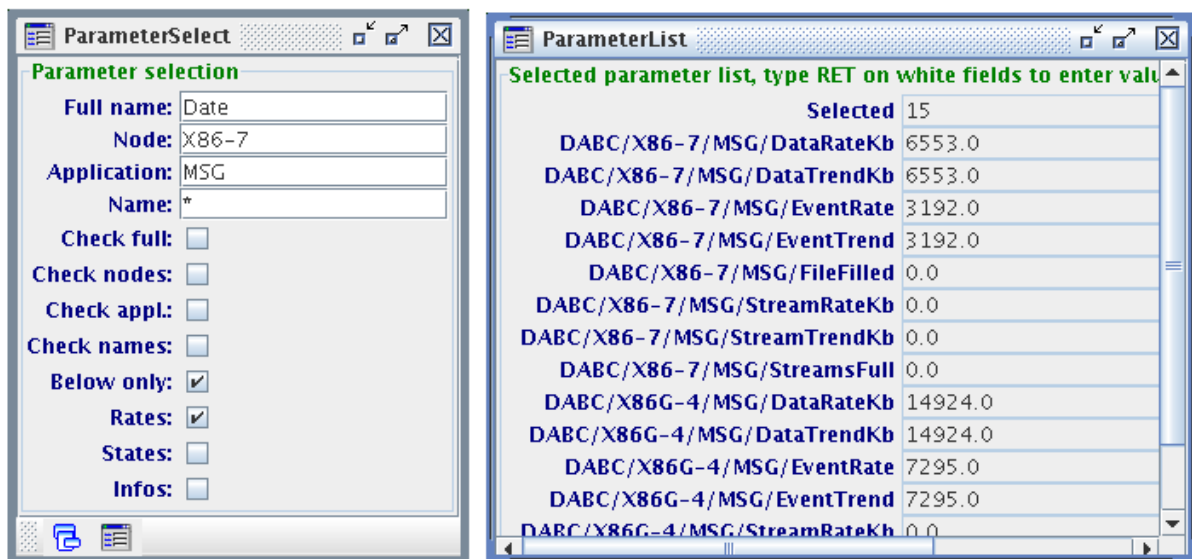
Figure 2.7: Parameter selection panel and selected parameter list.

2.2.7.1 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. 2.7, page 9. Text substrings for each of the four name fields can be specified as well as a selection of record types. Values can be saved (see 2.3, page 12). With the check boxes the filter function for each of these can (de)activated. The parameter list at the right window in Fig. 2.7, page 9 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.

2.2.8 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

2.2.8.1 States

States are records having a number for severity (0 to 4), a color, and a brief state description (see Fig. 2.8, page 10). 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.
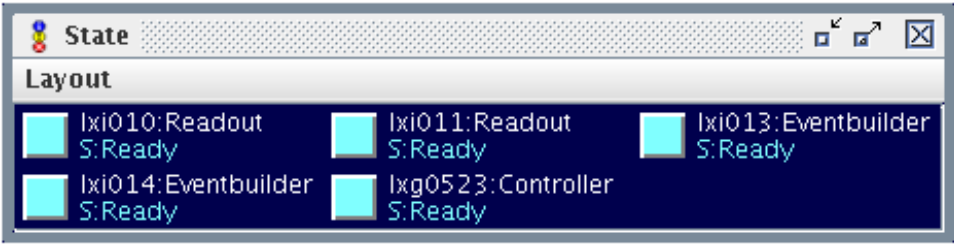
Figure 2.8: States.
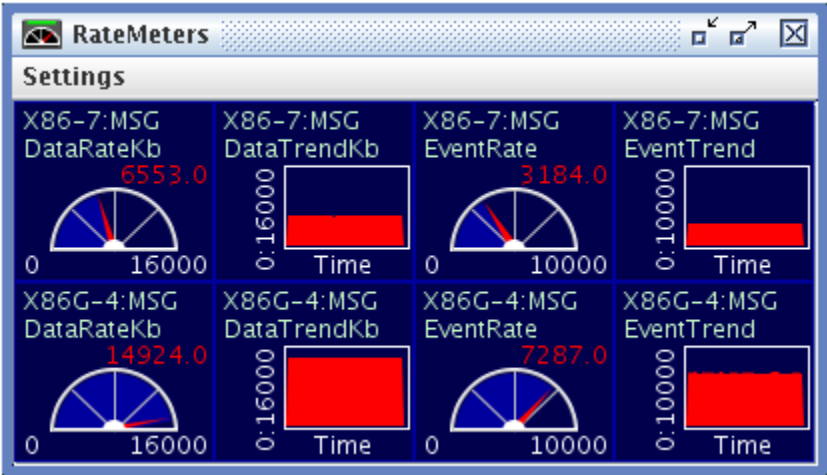
Figure 2.9: Rates.

2.2.8.2   Rate meters

All rate meters are displayed in the meter panel, Fig.2.9, page 10. Meters can be removed in the parameter table (See Fig. 2.6, page 8) with the Show buttons like the other graphical parameters. Saving the setup, the visibility will be preserved.
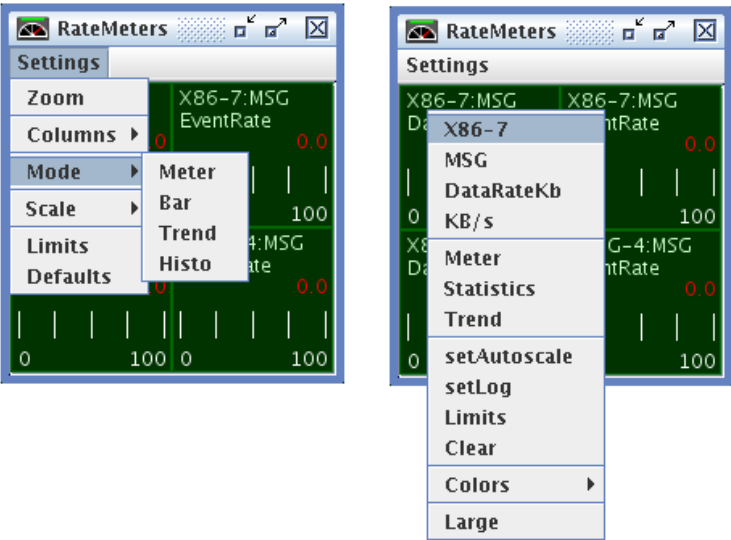
Figure 2.10: Steering menus.

On the left side in Fig. 2.10, page 10 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 2.3, page 12).

2.2.8.3 Histograms

Histogram panels are handled in pretty much the same way as the rate meters.   All histograms are
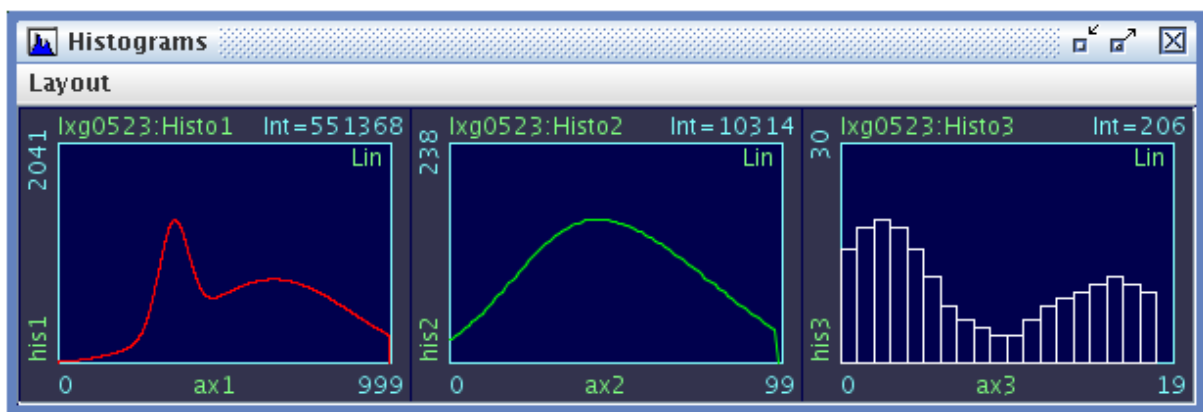
Figure 2.11: Histograms.

displayed in the histogram panel, Fig.2.11, page 11. Histograms can have arbitrary size set in Layout menu.

2.2.8.4 Information
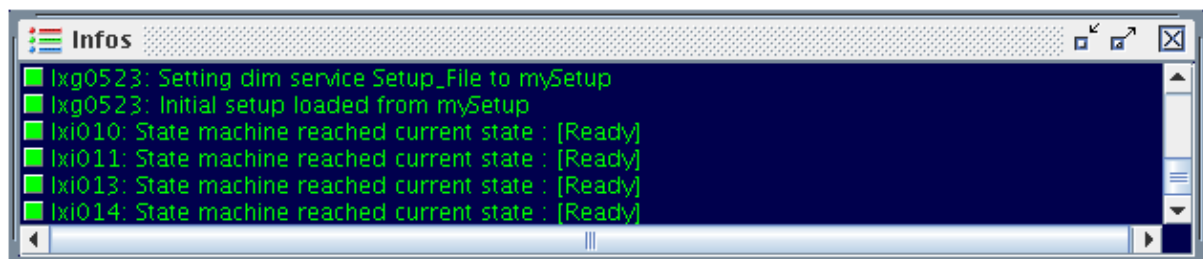
Figure 2.12: Info.

Information records mainly display one line of text with a color (see Fig. 2.12, page 11).

2.2.8.5 Logging window

Fig. 2.13, page 12 show the logging window.
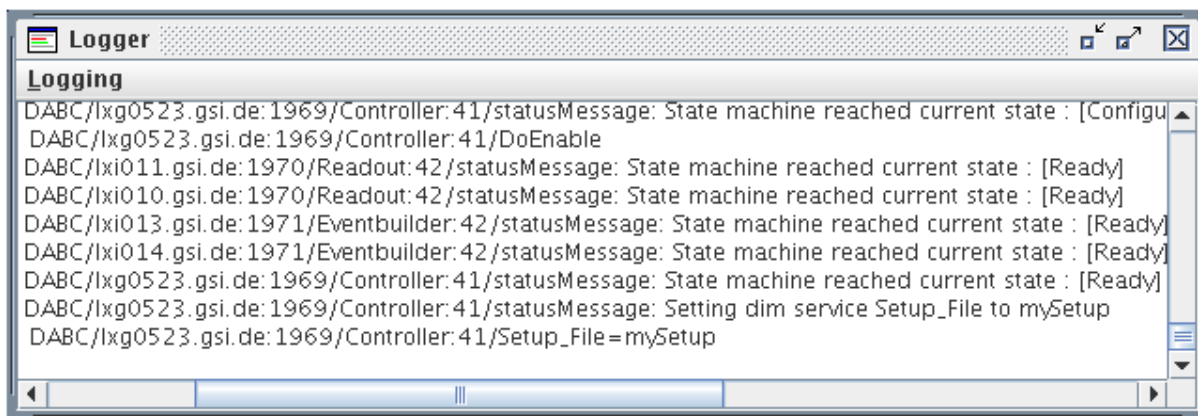
Figure 2.13: Logging.

2.3   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.
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 3

MBS GUI Manual: *MBS* GUI

[user/user-gui-mbs.tex]

3.1 *MBS* event building

3.1.1 *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 **??**, page **??**). 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.
Except

- The node running the GUI must get granted `rsh` access at least to the *MBS* node where the prompter or dispatcher shall run. This means that the node name and user name of the GUI node must be in the `.rhosts` file in the Lynx home directory
- The node name of the Lynx node itself also must be in `.rhosts`.
- 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 3.3.1, page 18).

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.

3.1.2 *MBS* control panel

Fig. 3.1, page 14 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 2.3, page 12). 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>
```

Figure 3.1: MBS controller.

```
<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" />
</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.

3.1.2.1 *MBS* controller buttons

Save panel settings, see 2.3, page 12.

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 2.2.3, page 7).

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.

3.1.3 *MBS* command panel
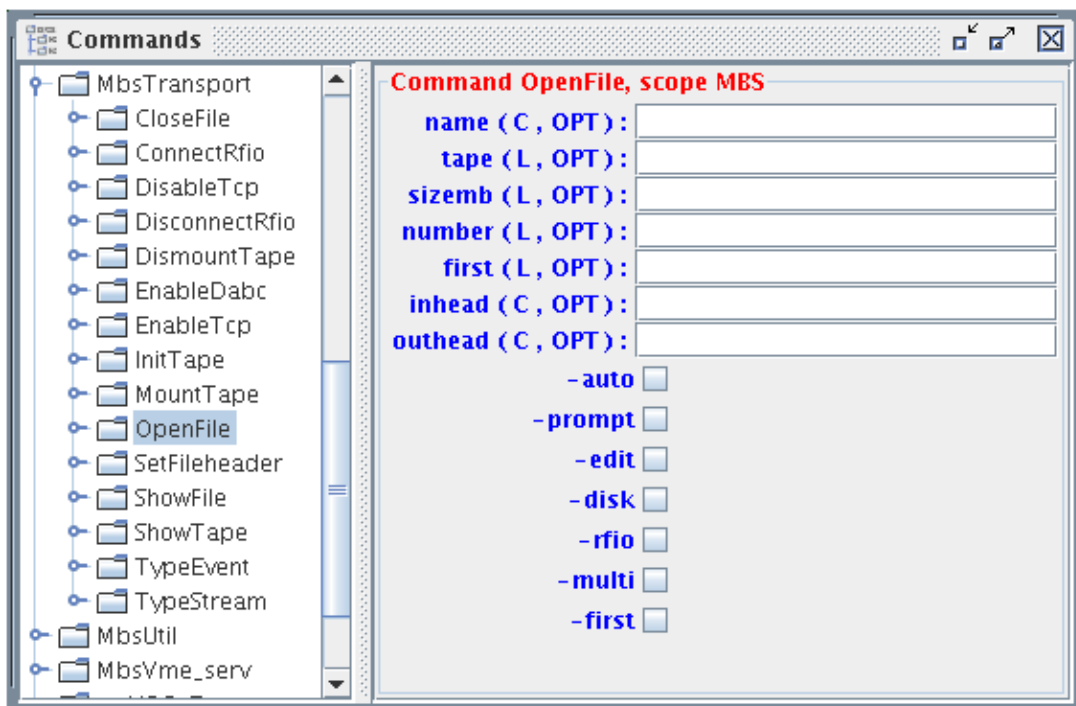
Figure 3.2: Command panel.

Fig. 3.2, page 15 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. 3.4, page 16). 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. 3.3, page 16 shows that only dispatcher and prompter are up and therefore only their commands are seen. Fig. 3.4, page 16 shows in addition the commands of util and transport after configuration.
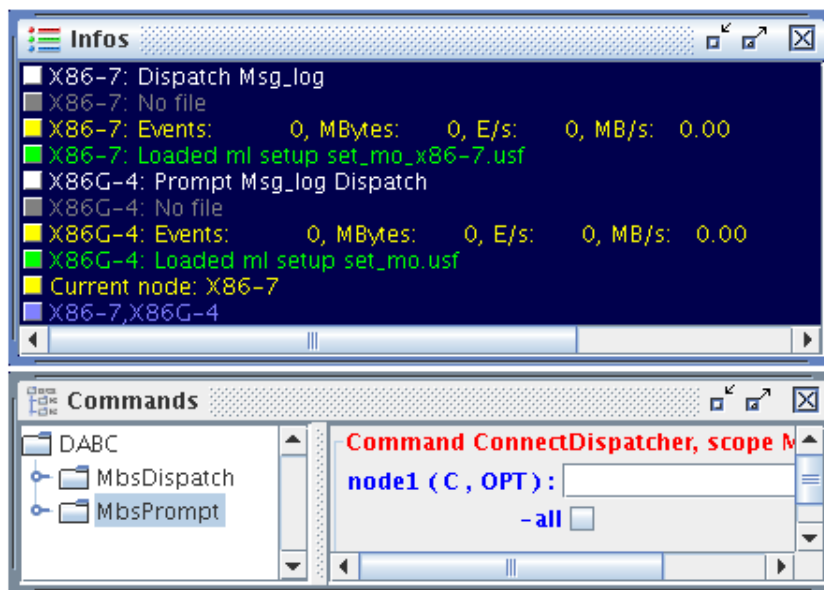
Figure 3.3: Info and command panel.
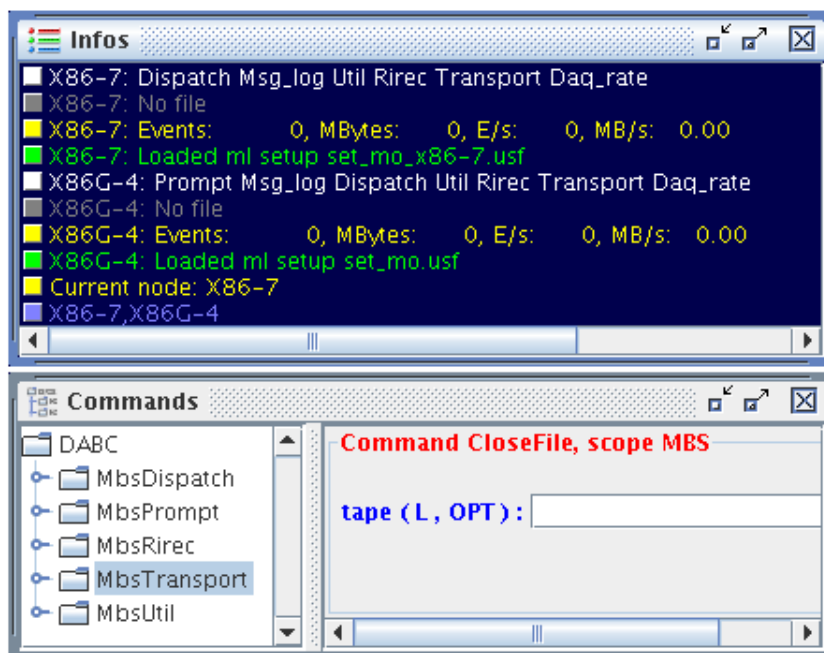
Figure 3.4: Info and command panel.

3.2 *MBS* DIM parameters

3.2.1 *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

3.2.2 *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

3.2.3 *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.

3.2.4 *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).

3.2.5 *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

3.2.6 *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

3.2.7   *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.

3.3   Working directories

3.3.1   *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 4

MBS GUI Manual: *MBS* Application GUI

[programmer/prog-gui.tex]

4.1 GUI Guide lines

The *DABC* GUI is written in Java. In the following we refer to it as a whole as *xGUI*. It uses the DIM Java package to register the DIM services provided by the *DABC* DIM servers. It is generic in that it builds most of the panels from the services available. Thus it can control and monitor any system running DIM servers conforming to rules described in the following. According the description above it does the following:

- Get list of commands and parameters and create objects for each.
- Put parameters in a table.
- Put commands in a command tree.
- Create graphics panels for rate meters, states, histograms, and infos.

4.2 DIM Usage

DIM is a light weight communication protocoll based on publish/subscribe mechanism. Servers publish named services (commands or parameters) to a DIM name server. Clients can subscribe such services by name. They then get the values of the services subscribed from the server providing it. Whenever a server updates a service, all subscribed clients get the new value. Clients can also execute commands on the server side.

DIM provides the possibility to specify parameters and command arguments as primitives (I or L,X,C,F,D) or structures. The structures are described in a format string which can be retrieved by the clients (for parameters and commands) and servers (for commands):

```
T:s;T:s;T:s ...
```

Thus a client can generically access parameter structures, but without semantical interpretation. In addition to the data and format string one longword called `quality` is sent.

4.2.1 *DABC* DIM naming conventions

When the number and kind of services of DIM servers often change it would be very convenient if a generic GUI would show all available services without further programming. It would be also very nice if standard graphical elements would be used to display certain parameters like rate meters. If we have many services it would be convenient to have a naming convention which allows to build tree structures on the GUI.

Naming conventions for generic *xGUI* (line breaks for better reading):

```
   /servernamespace
   /nodename[:nodeID]
   /[[applicationnamespace::]applicationname:]applicationID
   /[TYPE.module.]name
Example:
   /DABC/lx05/Control/RunState
```

We recommend to forbid spaces in any name fields. Dots should not be used except in names (last field). The generic *xGUI* can handle only services from one server name space (defined by DIM_DNS_NODE). For *DABC* and *MBS* this servernamespace is set to DABC.

4.2.2 *DABC* DIM records

For generic GUIs we need something similar to the EPICS records. This means to define structures which can be identified. How shall they be indentified? One possibility would be to prefix a type to the parameter name, i.e. `rate:DataRate`. Another to use the quality longword. This longword can be set by the server. One could mask the bytes of this longword for different information:

```
   mode (MSB)| visibility | type | status (LSB)
   mode: not used
   visibility: Bit wise (can be ORed)
     HIDDEN    = all zero
     VISIBLE   = 1  appears in parameter table
     MONITOR   = 2  in table, graphics shown automatically
                    if type is STATE, RATE or HISTOGRAM
     CHANGABLE = 4  in table, can be modified
     IMPORTANT = 8  in table also if GUI has a "minimal" view.
   type: (exclusive)
     PLAIN     = 0
     GENERIC   = 1
     STATE     = 2
     RATE      = 3
     HISTOGRAM = 4
     MODULE    = 5
     PORT      = 6
     DEVICE    = 7
     QUEUE     = 8
     COMMANDDESC= 9
     INFO      = 10
   status: (exclusive)
     NOTSPEC     = 0
     SUCCESS     = 1
     INFORMATION = 2
     WARNING     = 3
     ERROR       = 4
     FATAL       = 5
```

Then we could provide at the client side objects for handling and visualization of such records.

4.2.2.1   Record ID=0: Plain

Scalar data item of atomic type

4.2.2.2 Record ID=1: Generic self describing

For these one would need one structure per number of arguments. Therefore the generic type would be rather realized by a more flexible text format, like XML. This means the DIM service has a string as argument which must be parsed to get the values.

XML schema char, similar to command descriptor.
Format: C

4.2.2.3 Record ID=2: State

severity int, 0=Success, 1=warning, 2=error, 3=fatal)
color char, (Red, Green, Blue, Cyan, Magenta, Yellow)
state char, name of state
Format: L:1;C:16;C:16

4.2.2.4 Record ID=3: Rate

value float
displaymode int, (arc, bar, statistics, trend)
lower limit float
upper limit float
lower alarm float
upper alarm float
color char, (Red, Green, Blue, Cyan, Magenta, Yellow)
alarm color char, (Red, Green, Blue, Cyan, Magenta, Yellow)
units char
Format: F:1;L:1;F:1;F:1;F:1;F:1;C:16;C:16;C

4.2.2.5 Record ID=4: Histogram

Structure must be allocated including the data field witch may be integer or double.

channels int
lower limit float
upper limit float
axis lettering char
content lettering char
color char, (White, Red, Green, Blue, Cyan, Magenta, Yellow)
first data channel int
Format: L:1;F:1;F:1;C:32;C:32;C:16;I(or D)

4.2.2.6 Record ID=10: Info

verbose int, (0=Plain text, 1=Node:text)
color char, (Red, Green, Blue, Cyan, Magenta, Yellow)
text char, line of text
Format: L:1;C:16;C:128

4.2.2.7 Record ID=9: Command descriptor

This is an invisible parameter describing a command argument list. The service name must be correlated with the command name, e.g. by trailing underscore.

description char, XMl string describing arguments

Format: C

The descriptor string could be XML specifying the argument name, type, required and description. Question if default value should be given here for optional arguments. Example:

```
<?xml version="1.0" encoding="utf-8"?>
<command name="com1" scope="public" content="default">
<argument name="arg1" type="F" value="1.0" required="req"/>
<argument name="arg2" type="I" value="2" required="opt"/>
<argument name="arg3" type="C" value="def3" required="req"/>
<argument name="arg4" type="boolean" value="" required="opt"/>
</command>
```

The command definition can be used by the *xGUI* to build input panels for commands. The `scope` can be used to classify commands, `content` should be set to default if argument values are default, values if argument values have been changed.

4.2.2.8 Commands

Commands have one string argument only. This leaves the arguments to semantic definitions in string format. To implement a minimal security, the first 14 characters of the argument string should be an encrypted password (13 characters by crypt plus space). The arguments are passed as string. A command structure could look like:

password char[14]
argument char, string
Format: C

The argument string has the same XML as the command description. Thus, the same parser can be used to encode/decode the description (parameter) and the command. An alternate format is the *MBS* style format `argument=value` where boolean arguments are given by `-argument` if argument is true.

4.2.2.9 Setting parameters

If a parameter should be changable from the *xGUI*, there must be a command for that. A fixed command SetParameter must be defined on the server for that. Argument is a string of form `name=value`. In the parameter table of the *xGUI* one field can be provided to enter a new value and the command SetParameter is used to set the new value.

4.2.3 Application servers

Any application which can implement DIM services can be controlled by the generic *xGUI* if it follows the protocol described above. The first application was *DABC*, the second one *MBS*.

4.2.4 *DABC* GUI usage of DIM

The service names follow a structured syntax as described above. The name fields are used to build trees (for commands). Using the DIM quality longword (delivered by the server together with each update) simple aggregated data services (records) are defined. Currently the records
STATE, RATE, HISTOGRAM, COMMANDDESC and INFO.
are used. When the *xGUI* receives the first update of a service (immediately after subscribing) it can determine the record type and handle the record in an appropriate way. The COMMANDDESC record is an XML string describing a command. The name of a descriptor record must be the name of the command it describes followed by an underscore.

4.3 GUI global layout

The top window of the *xGUI* is a **JFrame**. Inside that is a **JPanel** which contains on top a **JToolBar** (all the main buttons), in the middle a **JDesktopPane** (main viewing area), and at the bottom a **JTextArea** (One line text for server list). All other windows are inside (added to) the desktop as **JInternalFrame**s. Typically such a frame contains again a **JPanel**. Inside that panel various different layouts can be used like **JSplitPane**, or a **Jtree** in a **JScrollPane**. In fact, **xInternalFrame**, a subclass of **JInternalFrame** is used. It can contain exactly one panel, has a mechanism to store and restore its size and position, and implements the callback functions for resizing and closing.

Inside the internal frames two types of panels are often used: prompter panels and graphics panels.

4.3.1 Prompter panels

Prompter panels can be implemented subclassing class **xPanelPrompt**. Example: *DABC* launch panel. The layout is in rows. A row can be a prompter line (**JLabel** label and **JTextField** input field), a text button **JButton**, or a **JLabel** label and **JCheckBox**. At the bottom there is a **JToolBar** where buttons with icons can be placed. The prompter class must implement the **ActionListener**, ie. provide the *actionPerformed* function which is the central call back function for all elements.

4.3.2 Graphics panels

Graphics panels are provided by class **xPanelGraphics**. The layout is as a matrix with columns and rows. All items to be added must be **JPanel**s and implement **xiPanelItem** (see below). The items are added line by line. The number of items per line (columns) is a parameter. All items must have the same size. Currently no menu bar is supported.

4.4 GUI Panels

Brief description of panels implemented in the *xGUI*.

4.4.1 *DABC* launch panel

xPanelDabc extending **xPanelPrompt**.
Form to enter all information needed to startup *DABC* tasks and buttons to execute standard commands. The values of the form (internally stored in **xFormDabc** extending of **xForm**) can be saved to an XML file and are restored from it. File name is either `DabcLaunch.xml` or translation of DABC_LAUNCH_DABC, respectively.

4.4.2 *MBS* launch panel

xPanelMbs extending **xPanelPrompt**.
Form to enter all information needed to startup *MBS* tasks and buttons to execute standard commands. The values of the form (internally stored in **xFormMbs** extending of **xForm**) can be saved to an XML file and are restored from it. File name is either `MbsLaunch.xml` or translation of DABC_LAUNCH_MBS, respectively.

4.4.3 Combined *DABC* and *MBS* launch panel

xPanelDabcMbs extending **xPanelPrompt**.
It is a combination of both, *DABC* and *MBS* launch panel.

4.4.4 Parameter table

xPanelParameter extending *JPanel*.
Is rebuilt from scratch by *xDesktop* whenever the DIM service list has been updated.
The panel gets the list of parameters (*xDimParameter*) from the DIM browser (*xDimBrowser*). It builds a table from all visible parameters. It creates a list of command descriptors (*xXmlParser*).

4.4.5 Parameter selection panel

xPanelSelect extending *xPanelPrompt*.
This form can be used to specify various filters on parameter attributes. Parameters matching the filters are shown in a separate frame. Values are updated on DIM update and can be modified interactively.

4.4.6 Command panel

xPanelCommand extending *JPanel*.
Is rebuilt from scratch by *xDesktop* whenever the DIM service list has been updated.
This panel is split into a right and a left part. On the left, there is the command tree, on the right the argument prompter panel for the currently selected command. The panel gets the list of commands (*xDimCommand*) from the DIM browser (*xDimBrowser*). The list of command descriptors (*xXmlParser*) is copied in *xDesktop* from *xPanelParameter* to *xPanelCommand* and the *xXmlParser* objects are added to the *xDimCommand* objects they belong to.

4.4.7 Monitoring panels

These panels are very similar to *xPanelGraphics* but have additional functionality. **TODO:** In the future, *xPanelGraphics* should be extended to provide all that functionality, or at least serves as base class.

xPanelMeter: *JPanel*, for rate meters (*xMeter*)
xPanelState: *JPanel*, for states (*xState*)
xPanelInfo: *JPanel*, for infos (*xInfo*)
xPanelHisto: *JPanel*, for histograms (*xHisto*)

The monitoring panels contain special graphics objects:

4.4.7.1 *xMeter*

Displays a changing value between limits as rate meter, bar, histogram or trend. With the right mouse a context menu is popped up where one can switch between these modes. One also can change the limits, autoscale mode (limits are adjusted dynamically), and the color.

4.4.7.2 *xRate*

Displays a changing value between limits as bar. Very compact with full name.

4.4.7.3 *xState*

Displays a severity as colored box together with a brief text line.

4.4.7.4 *xHisto*

Displays a histogram.

4.4.7.5   *xInfo*

Displays a colored text line.

4.4.8   Logging window

xPanelLogger extending *JPanel*.
Central window to write messages.

4.5   GUI save/restore setups

There are several setups which can be stored in XML files and are retrieved when the *xGUI* is started again.

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.

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`.

4.5.1   Record attributes

File `Records.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<Record>
<Meter name="DABC/X86-7/MSG/DataRateKb"
       visible="true"
       mode="0"
       auto="false"
       log="false"
       low="00000000.0"
       up="00016000.0"
       color="Red"/>
</Record>
```

4.5.2   Parameter filter

File `Selection.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<Selection>
<Full contains="Date" filter="false" />
<Node contains="X86-7" filter="false" />
<Application contains="MSG" filter="false" />
<Name contains="*" filter="false" />
<Records Only="true"  Rates="true"  States="false"  Infos="false" />
</Selection>
```

4.5.3   Windows layout

File `Layout.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<Layout>
<WindowLayout>
<Main shape="357,53,857,953" columns="0" show="true"/>
<Command shape="0,230,650,200" columns="0" show="false"/>
<Parameter shape="20,259,578,386" columns="0" show="false"/>
<Logger shape="0,650,680,150" columns="0" show="false"/>
<Meter shape="463,13,413,236" columns="4" show="false"/>
<State shape="85,504,313,206" columns="2" show="false"/>
<Info shape="521,482,613,217" columns="1" show="false"/>
<Histogram shape="124,508,613,206" columns="3" show="false"/>
<DabcLauncher shape="0,0,100,100" columns="0" show="false"/>
<MbsLauncher shape="50,14,404,272" columns="0" show="false"/>
<DabcMbsLauncher shape="0,0,430,424" columns="0" show="false"/>
<ParameterSelect shape="300,0,271,326" columns="0" show="true"/>
<ParameterList shape="13,364,810,426" columns="1" show="true"/>
</WindowLayout>
<TableLayout>
<Parameter width="74,74,74,74,74,74,74,74" />
</TableLayout>
</Layout>
```

4.5.4   *DABC* launch panel values

File `DabcLaunch.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<DabcLaunch>
<DabcMaster prompt="DABC Master" value="node.xxxx.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" />
<DabcServers prompt="%Number of needed DIM servers%" value="5" />
</DabcLaunch>
```

4.5.5   *MBS* launch panel values

File `MbsLaunch.xml`

```
<?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" />
<MbsScript prompt="MBS Script" value="script/remote_exe.sc" />
<MbsCommand prompt="Script command" value="whatever command" />
<MbsServers prompt="%Number of needed DIM servers%" value="3" />
</MbsLaunch>
```

4.6 DIM update mechanism

To get informed when a DIM parameter has been updated a DIM client has to register to it. In a Java DIM client this is done by instantiating a subclass of ***DimInfo***. In *xGUI* this is ***xDimParameter*** implementing callback function *infoHandler*. After registration the callback function is called once immediately. In *infoHandler* one can use getter functions to get the quality, the format string, and the value(s).

4.6.1 *xDimBrowser*

The central object handling the available lists of DIM parameters and commands is the ***xDimBrowser***. It provides the functions:

xDimBrowser(...) : Constructor. Arguments: references to the graphics panels ***xPanelMeter***, ***xPanelState***, ***xPanelInfo*** and ***xPanelHisto***. There are protected functions to get then the references to these panels.
protected initServices(String wildcard) : Get list of available services from DIM name server DIM_DNS_NODE. Create vectors of alphabetically ordered parameters (***xDimParameter***) and commands (***xDimCommand***) and their interfaces, respectively. The references of the graphics panels are passed to the parameter objects.
addInfoHandler(xiDimParameter p, xiUserInfoHandler ih) : Interface function to add an additional info handler to a parameter. The *infoHandler* function of this handler is called at the end of the *infoHandler* function of ***xDimParameter***.
removeInfoHandler(xiDimParameter p, xiUserInfoHandler ih) : Interface function to remove an info handler added before.
protected Vector<xDimParameter> getParameterList() :
protected Vector<xDimCommand> getCommandList() :
Vector<xiDimParameter> getParameters() : From outside one gets only references to the interfaces.
Vector<xiDimCommand> getCommands() : From outside one gets only references to the interfaces.
protected releaseServices(boolean cleanup) : Removes all external handlers of the parameters. Sets all parameters to inactive. This means that in the *infoHandler*s no more graphical activity is performed. If `cleanup` is `true` all parameters release their service and are set to inactive. Then the parameter vector is cleared. Then the command vector is cleared. Note that the objects themselfes are removed only by next garbage collection.
protected enableServices() : All parameters are set to active.
 :

4.6.2 Getting parameters and commands

Once the parameter and command objects have been created by the browser, it is up to the ***xPanelParameter*** and ***xPanelCommand*** object, respectively, to manage them. These two objects are created new each time an update occurs.

4.6.2.1 *xPanelParameter*

Extends ***JPanel***. It has references to the browser and all graphics panels. It owns the parameter table (***JTable***). In the constructor the following steps are performed:

1. Get reference to list of parameters (from browser).
2. Set in all parameters the table index to -1 (*infoHandler*s will no longer update table fields).
3. Scan through all parameters and check if any quality is still -1 which would mean that the type is undefined. That is repeated two times with 2 seconds delay to give the DIM servers the chance to update all parameters. If still any quality is -1 this is an error.
4. Restore record attributes of meters and histograms from XML file.
5. *cleanup* graphics panels.
6. Create new table.
7. Add parameters to table by calling function *xDimParameter.addRow*. This function also creates graphical presentations of the parameters (e.g. ***xMeter***) and add them to the appropriate graphics panels (e.g.

xPanelMeter) if needed.
8. Builds list of command descriptors (*xXmlParser*).
9. Add table to its panel.
10. *updateAll* graphics panels.

4.6.2.2   *xPanelCommand*

Extends **JPanel**. It has references to the browser. It owns the command tree (**JTree**). In the constructor the following steps are performed:

1. Get reference to list of commands (from browser).
2. Create from that list a command tree to be shown on left side in window.
3. Create arguments panel for the right side. When a command is selected and an XML descriptor is available, the arguments are shown as prompter panel.
4. Call back functions for command execution.

Function *setCommandDescriptors* is called from **xDesktop** to build the command descriptor list.
Function *setUserCommand* is called from **xDesktop** to specify a **xiUserCommand** object which provides a function *getArgumentStyleXml* which is used to determine how the command string has to be formatted (either like the command XML description or like the MBS style).

4.6.3   Startup sequence

The build up sequence during the GUI start is done in the **xDesktop**. Sequence on startup:

1. Create application panels and graphics panels.
2. Create browser **xDimBrowser** and call its *initServices*.
3. Create prompter panels.
4. Create **xPanelParameter**.
5. Call browser *enableServices* function. Now all parameters (DIM clients) should already operate.
6. Create **xPanelCommand** and call its *setCommandDescriptors*. The descriptors are provided as parameters. The descriptor list is generated by **xPanelParameter**.
7. Call *init* and *setDimServices* of all application panels. Pass **xiUserCommand** object from first application panel object to **xPanelCommand**.
8. Create the internal frames to display all panels which shall be visible.

4.6.4   Update sequence

The update sequence is either triggered by a menu button interactively, or invoked in callback functions of prompter panels after changes of the DIM services. The update is done in *actionPerformed* of **xDesktop**, command Update. Sequence on update:

1. Call *releaseDimServices* of all application and prompter panels.
2. Call *xDimBrowser.releaseServices* which deactivates all parameters and removes all application handlers.
3. Discard the parameter and command panel and call Java garbage collector. At this point no more references to parameters or commands should exist and all objects can be removed.
4. Call *xDimBrowser.initServices*.
5. Create **xPanelParameter**.
6. Create **xPanelCommand**.
7. Call *setDimServices* of all application panels. Pass **xiUserCommand** object from first application panel object to **xPanelCommand**.
8. Call *xDimBrowser.enableServices*.
9. Call *xPanelCommand.setCommandDescriptors*.
10. Update the internal frames of parameters and commands.

4.7   Application specific GUI plug-in

Besides the generic part of the *xGUI* it might be useful to have application specific panels as well, integrated in the generic *xGUI*. This is done by implementing subclasses of ***xPanelPrompt***. The class name (only one) can be passed as argument to the java command starting the *xGUI* or by setting variable DABC_APPLICATION_PANELS being a comma separated list of class names. Variable is ignored if class name is given as argument. The classes must implement some interfaces:

xiUserPanel : needed by *xGUI*.
xiUserInfoHandler : needed to register to DIM services. This could be a separate class.
xiUserCommand : optional to specify command formats.

One can connect call back functions to parameters, get a list of available commands, create his own panels for display using the graphical primitives like rate meters. Optional ***xiUserCommand*** provides a function to be called in the *xGUI* (***xPanelCommand***) when a command shall be executed. This function steers if the command arguments have to be encoded in XML style or argument list style.

There is for convenience another subclass of ***xInternalFrame*** and ***JInternalFrame*** for easy formatting from one to four panels (***JPanel*** or ***xPanelGraphics***) inside, ***xInternalCompound***.

Examples of such application panel can be found on directory `application`.

4.7.1   Java Interfaces to be implemented by application

4.7.1.1   Interface *xiUserPanel*

○ `abstract void init(xiDesktop d, ActionListener a)`
   Called by *xGUI* after instantiation. The desktop can be used to add frames (see below).
○ `String getHeader();`
   Must return a header/name text after instantiation.
○ `String getToolTip();`
   Must return a tooltip text after instantiation.
○ `ImageIcon getIcon();`
   Must return an icon after instantiation.
○ `xLayout checkLayout();`
   Must return the panel layout after initialization.
○ `xiUserCommand getUserCommand();`
   Must return an object implementing ***xiUserCommand***, or null. See below.
○ `void setDimServices(xiDimBrowser b);`
   Called by *xGUI* whenever the DIM services had been changed. The browser provides the command and parameter list (see below). One can select and store references to commands or parameters. A ***xiUserInfoHandler*** object can be registered for each selected parameter. Then the *infoHandler* method of this object is called for each parameter update.
○ `void releaseDimServices();`
   All local references to commands or parameters must be cleared!

4.7.1.2   Interface *xiUserCommand*

○ `boolean getArgumentStyleXml(String scope, String command);`
   Return true if command shall be composed as XML string, false if *MBS* style string. `Scope` is specified in the XML command descriptor, `command` is the full command name.

4.7.1.3   Interface *xiUserInfoHandler*

○ `void infoHandler(xiDimParameter p, int handlerID)`
   An object implementing this interface can be added to each parameter as call back handler. This is done by the browser function *setInfoHandler*, see below. Function *infoHandler* is then called in the callback of the parameter.

○ `String getName()`
  Called by *xDimParameter* to get a uniquie name of this handler. Must return a name of the handler to distinguish from other handlers.

4.7.2   Java Interfaces provided by GUI

4.7.2.1   Interface *xiDesktop*

○ `void addFrame(JInternalFrame f)`
  Adds a frame to desktop if a frame with same title does not exist.
○ `void addFrame(JInternalFrame frame, boolean manage)`
  Adds a frame to desktop if a frame with same title does not exist.
○ `boolean findFrame(String title)`
  Checks if a frame exists on the desktop.
○ `void removeFrame(String title)`
  Remove (dispose) a frame from the desktop and list of managed frames.
○ `void setFrameSelected(String title, boolean select)`
  Switch a frames selection state (setSelected).
○ `void toFront(String title)`
  Set frames to front.

4.7.2.2   Interface *xiDimBrowser*

○ `Vector<xiDimParameter> getParameters()`
  Typically called in *setDimServices* to get list of available parameters. Only selected parameters may be registered to.
○ `Vector<xiDimCommand> getCommands()`
  Typically called in *setDimServices* to get list of available commands.
○ `void setInfoHandler(xiDimParameter p, xiUserInfoHandler h)`
  Typically called in application function *setDimServices* to register a call back handler (mostly `this`) to a parameter.
○ `void removeInfoHandler(xiDimParameter p, xiUserInfoHandler h)`
  Typically called in application function *releaseDimServices* to remove a call back handler of a parameter.
○ `void sleep(int s)`

4.7.2.3   Interface *xiDimCommand*

○ `void exec(String command)`
○ `xiParser getParserInfo()`

4.7.2.4   Interface *xiDimParameter*

○ `double getDoubleValue()`
○ `float getFloatValue()`
○ `int getIntValue()`
○ `long getLongValue()`
○ `String getValue()`
○ `xRecordMeter getMeter()`
○ `xRecordState getState()`
○ `xRecordInfo getInfo()`
○ `xiParser getParserInfo()`
○ `boolean parameterActive()`
○ `boolean setParameter(String value)`
  Builds and executes a DIM command *SetParameter name=vale* where name is the name part of the full DIM name string.

4.7.2.5   Interface *xiParser*

○ String getDns()
○ String getNode()
○ String getNodeName()
○ String getNodeID()
○ String getApplicationFull()
○ String getApplication()
○ String getApplicationName()
○ String getApplicationID()
○ String getName()
○ String getNameSpace()
○ String[] getItems()
○ String getFull()
○ String getFull(boolean build)
○ String getCommand()
○ String getCommand(boolean build)
○ int getType()
○ int getState()
○ int getVisibility()
○ int getMode()
○ int getQuality()
○ int getNofTypes()
○ int[] getTypeSizes()
○ String[] getTypeList()
○ String getFormat()
○ boolean isNotSpecified()
○ boolean isSuccess()
○ boolean isInformation()
○ boolean isWarning()
○ boolean isError()
○ boolean isFatal()
○ boolean isAtomic()
○ boolean isGeneric()
○ boolean isState()
○ boolean isInfo()
○ boolean isRate()
○ boolean isHistogram()
○ boolean isCommandDescriptor()
○ boolean isHidden()
○ boolean isVisible()
○ boolean isMonitor()
○ boolean isChangable()
○ boolean isImportant()
○ boolean isLogging()
○ boolean isArray()
○ boolean isFloat()
○ boolean isDouble()
○ boolean isInt()
○ boolean isLong()
○ boolean isChar()
○ boolean isStruct()

4.7.3   Other interfaces

4.7.3.1   Interface *xiPanelItem*

Interface to be implemented for objects to be placed onto **xPanelGraphics**. The elementary graphics objects of *xGUI* all have implemented this interface. Example **xMeter**, **xState**, **xHisto**.

○ Dimension getDimension()
○ int getID()
○ String getName()
○ JPanel getPanel()
○ Point getPosition()
○ void setActionListener(ActionListener a)
○ void setID(int id)
   Set internal ID.
○ void setSizeXY()
   Sets the preferred size of item to internal vale.
○ void setSizeXY(Dimension d)
   Sets the preferred size of item to specified dimension.

Example:

```
public void setActionListener(ActionListener a){action=a;}
public JPanel getPanel() {return this;}
public String getName(){return sHead;}
public void setID(int i){iID=i;}
public int getID(){return iID;}
public Point getPosition(){return new Point(getX(),getY());};
public Dimension getDimension(){return new Dimension(ix,iy);};
public void setSizeXY(){setPreferredSize(new Dimension(ix,iy));}
public void setSizeXY(Dimension dd){setPreferredSize(dd);}
```

4.7.4   Example

Example of a minimalistic application panel. Full running code in **MiniPanel**.   That is how the class must look

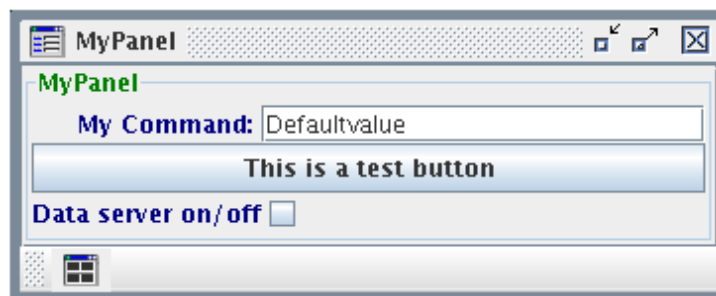**Figure 4.1:** Mini panel.

like:

```
public class MiniPanel  extends xPanelPrompt
                    implements  xiUserPanel,
                                    ActionListener
```

The constructor must not have arguments! Icon, name and tooltip have to be passed by getter function to the caller (the GUI desktop). Layout is mandatory. Declarations have been masked out in the code snippets. There are some

icons one could use for the prompter panels:

 usericonblue usericonred usericongreen usericonyellow

```
public MiniPanel(){
   super("MyPanel");
   menuIcon=xSet.getIcon("icons/usericongreen.png");
   name=new String("MyPanel");
   tooltip=new String("Launch my panel");
   layout = xSet.getLayout(name);
   if(layout == null)
   layout=xSet.createLayout(name,new Point(100,200), new Dimension(100,75),1,true);
}
```

The simple functions to be implemented for the interface *xiUserPanel* (we do not provide a command formatting function) are:

```
public String getToolTip(){return tooltip;}
public String getHeader(){return name;}
public ImageIcon getIcon(){return menuIcon;}
public xLayout checkLayout(){return layout;}
public xiUserCommand getUserCommand(){return null;}
```

The *init* is called once after constructor. Here we have to setup all panels. We have in the main panel three lines: one text prompt, a text button, and a check box. At the bottom we have one icon button which would open the display frame. There are some icons one could use for that:

 windowblue windowred windowgreen

```
public void init(xiDesktop desktop, ActionListener al){
desk=desktop; // save
prompt=addPrompt("My Command: ","Defaultvalue","prompt",20,this);
addTextButton("This is a test button","button","Tool tip, whatever it does",this);
check=addCheckBox("Data server on/off","check",this);
graphIcon   = xSet.getIcon("icons/windowgreen.png");
addButton("Display","Display info",graphIcon,this);
state = new xState("ServerState", xState.XSIZE,xState.YSIZE);
stapan=new xPanelGraphics(new Dimension(160,50),1); // one column of states
metpan=new xPanelGraphics(new Dimension(410,14),1); // one columns of meters
franame=new String("MyGraphics");
fralayout = xSet.getLayout(franame);
if(fralayout == null)
fralayout=xSet.createLayout(franame,new Point(400,400), new Dimension(100,75),1,true);
frame=new xInternalCompound(franame,graphIcon,0,fralayout,xSet.blueD());
}
```

Here we have the callback function for the interactive elements, the text prompt, the button, the checker, and the icon:

```
private void print(String s){
System.out.println(s);
}
public void actionPerformed(ActionEvent e) {
String cmd=e.getActionCommand();
if ("prompt".equals(cmd)) {
  print(cmd+":"+prompt.getText()+" "+check.isSelected());
```

```
} else if ("button".equals(cmd)) {
  print(cmd+":"+prompt.getText()+" "+check.isSelected());
} else if ("check".equals(cmd)) {
  print("Data server "+check.isSelected());
  if(check.isSelected()){
    if(param != null)param.setParameter("0");
    state.redraw(0,"Green","Active",true);
  } else {
    if(param != null)param.setParameter("1");
    state.redraw(0,"Gray","Dead",true);
}} else if ("Display".equals(cmd)) {
  if(!desk.findFrame(franame)){
    frame=new xInternalCompound(franame,graphIcon,0,fralayout,xSet.blueD());
    frame.rebuild(stapan, metpan);
    desk.addFrame(frame);
}}
}
```
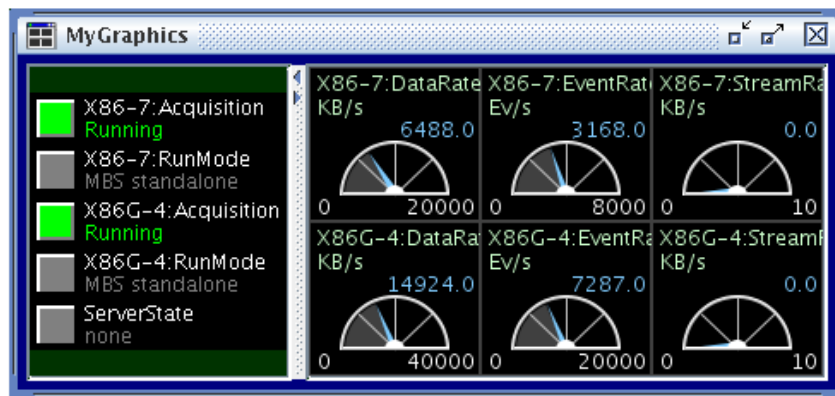

Figure 4.2: Ministates.

With the checker we toggle the *xState* state ServerState in screen shot). The *xiDimParameter* param to be toggled we will find in the next. To get access to DIM parameters we must implement *setDimServices*. We suggest that there is a parameter *Setup_File* which has a string value. The *myInfoHandler* class is described next.

```
public void setDimServices(xiDimBrowser browser){
Vector<xiDimParameter> vipar=browser.getParameters();
for(int i=0;i<vipar.size();i++){
  xiParser p=vipar.get(i).getParserInfo();
  String pname=new String(p.getNode()+":"+p.getName());
  if(p.isRate()){
      xMeter meter=new xMeter(xMeter.ARC,
        pname,0.0,10.0,xMeter.XSIZE,xMeter.YSIZE,xSet.blueL());
      meter.setLettering(p.getNode(),p.getName(),
        vipar.get(i).getMeter().getUnits(),"");
      metpan.addGraphics(meter,false);
      browser.addInfoHandler(vipar.get(i),
        new myInfoHandler(pname,meter,null));
  } else if(p.isState()){
      xState state=new xState(pname,xState.XSIZE,xState.YSIZE);
      stapan.addGraphics(state,false);
      browser.addInfoHandler(vipar.get(i),
        new myInfoHandler(pname,null,state));
  } else if(p.getFull().indexOf("Setup_File")>0) param=vipar.get(i);
```

```
} // end list of parameters
stapan.addGraphics(state,false);
stapan.updateAll();
metpan.updateAll();
if(frame != null) frame.rebuild(stapan, metpan);
```

All references or allocated objects from *setDimServices* we have to free in *releaseDimServices*:

```
public void releaseDimServices(){
    metpan.cleanup();
    stapan.cleanup();
    param=null;
}
```

We provide a little extra class implementing **xiUserHandler** function *infoHandler*. Each parameter we want to monitor gets its own handler instance which has direct access to our graphics panels.

```
private class myInfoHandler implements xiUserInfoHandler{
private myParameter(String Name, xMeter Meter, xState State){
name = new String(Name); // store
meter=Meter; // store
state=State; // store
}
public String getName(){return name;}
public void infoHandler(xiDimParameter P){
if(meter != null) meter.redraw(
   P.getMeter().getValue(),
   true, true);
if(state != null) state.redraw(
   P.getState().getSeverity(),
   P.getState().getColor(),
   P.getState().getValue(),
   true);
}
}
```

4.7.5   Store/restore layout

It is absolutely necessary to save and restore window layouts to be able to see the GUI after restart as before. This is done through **xLayout** objects which are managed centrally. They keep information about frame position, size, visibility, and the number of columns in graphics panels. All existing layouts are stored with the save setup button, and restored on startup.

References

Index