Data Acquisition Backbone Core

Programmer Manual

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

Produced: February 12, 2009, Revisions:
Titel: DABC: Programmer Manual

Document	Date	Editor	Revision	Comment
DABC-prog	2009-01-04	Hans G.Essel	1.0.0	First scetch
DABC-prog	2009-01-13	Sergey Linev	1.0.1	Second scetch
DABC-prog	2009-01-13	Jörn Adamczewski-Musch	1.0.1	Second scetch

Contents

Chapter 1

DABC Programmer Manual: Overview

[programmer/prog-overview.tex]

1.1 Introduction

The *DABC* *Programmer Manual* describes the aspects of the Data Acquisition Backbone Core framework that are necessary for programming user extensions. To begin with, this overview chapter explains the software objects and their collaboration, the intended mechanisms for controls and configuration, the dependencies of packages and libraries, and gives a short reference of the most important classes.

The following chapters contain full explanations of the *DABC* interface and service classes, describe the set-up with parameters, and give a reference of the Java GUI plug-in possibilities.

Finally, some implementation examples are treated in detail to illustrate these issues: the adaption of the GSI legacy DAQ system *MBS* within *DABC*; the application of a distributed event builder network (Bnet); the data import via UDP from a readout controller board (ROC); and the use of a PCI express board (*Active Buffer Board*).

1.2 Role and functionality of the objects

1.2.1 Modules

All processing code runs in module objects. There are two general types of modules:
the ***dabc::ModuleSync*** and the ***dabc::ModuleAsync***.

1.2.1.1 Class *dabc::ModuleSync*

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 ("non strictly blocking mode"). In the "strictly blocking mode", the blocking calls do nothing but wait. 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 *MainLoop()* itself catches and handles the exception. On state machine commands (e.g. Halt or Suspend, see section 1.3.1), the blocking calls are also left by exception, thus putting the mainloop thread into a stopped state.

1.2.1.2 Class *dabc::ModuleAsync*

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. Thus each callback function must check the available resources explicitly whenever it is entered.

1.2.2 Commands

A module may register ***dabc::Command*** objects in the constructor and may define command actions by overwriting a virtual command callback method *ExecuteCommand*.

1.2.3 Parameters

A module may register ***dabc::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.

1.2.4 Manager

The modules are organized and controlled by one manager object of class ***dabc::Manager***; this singleton instance is persistent independent of the application's state. One can always access the manager via *dabc::mgr()* function.

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 virtual interface methods must be implemented in subclass of ***dabc::Manager*** that knows the specific controls framework.

The manager receives and **dispatches commands** to the destination modules where they are queued and eventually executed by the modules threads (see section 1.2.1). The manager has an independent manager thread, used for manager commands execution, parameters timeout processing and so on.

1.2.5 Memory and buffers

Data in memory is referred by ***dabc::Buffer*** objects. Allocated memory areas are kept in ***dabc::MemoryPool*** objects.

In general case ***dabc::Buffer*** contains a list of references to scattered memory fragments from memory pool. Typically a buffer references exactly one segment. Buffer may have an empty list of references. In addition, the buffer can be supplied with a custom header.

The auxiliary class ***dabc::Pointer*** offers methods to transparently treat the scattered fragments from the user point of view (concept of "virtual contiguous buffer"). Moreover, the user may also get direct access to each of the fragments.

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 modules communicates with a memory pool via a ***dabc::PoolHandle*** object.

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 contiguos.

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 user 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.

1.2.6 Ports

Buffers are entering and leaving a module through ***dabc::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.

Depending on the module type, there are different possibilities to work with the ports in the processing functions of the module. These are described in section 4.2.5 for ***dabc::ModuleSync*** and section 4.2.6 for ***dabc::ModuleAsync*** respectively.

1.2.7 Transport

Outside the modules the ports are connected to ***dabc::Transport*** objects. On each node, a transport may either transfer buffers between the ports of different modules (local data transport), 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).

1.2.8 Device

A transport belongs to a ***dabc::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 ***dabc::Transport*** implementation shall be able to block (e. g. on socket receive), there can be only one transport for this thread.

A ***dabc::Device*** instance usually represents an I/O component (e. g. network card); there may be more than one ***dabc::Device*** instances 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.

1.2.9 Application

The ***dabc::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 in the different control system states (see section 1.3.1). In contrast to the ***dabc::Manager*** implementation that defines a framework control system (e.g. DIM, EPICS), the subclass of ***dabc::Application*** defines the experiment specific behaviour of the DAQ.

1.3 Controls and configuration

1.3.1 Finite state machine

The running state of the DAQ system is ruled by a *Finite State Machine* [**?**] 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.

The finite state machine itself is not necessarily part of the manager, but may be provided by an external control system. In this case, the manager defines the states, but does not check if a state transition is allowed. However, the *DABC* core system offers a native state machine to be used in the controls implementation; it can be activated in the constructor of the ***dabc::Manager*** subclass by method *InitSM()*.

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.

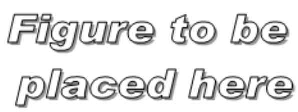


Figure 1.1: The finite state machine as defined by the manager.

There are 5 generic states to treat all set-ups:

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. However, the user may explicitly specify on creation time that an object shall be persistent (e.g. a device may be kept until the end of the process once it had been created).

1.3.2 Commands

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

1.3.3 Parameters for configuration and monitoring

The *Configuration* is done using parameter objects. The manager provides an interface to register parameters to the configuration/control system.

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 monitoring parameters change, the control system is updated by interface methods of the manager and may refresh the GUI representation. Chapter 5 will explain the usage of parameters for configuration in detail.

1.4 Package and library organisation

The complete system consists of different packages. Each package is represented by a subproject of the source code with own namespace. There may be one or more shared libraries for each package. Main packages are as follows:

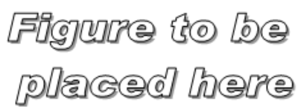


Figure 1.2: The *DABC* packages

1.4.1 Core system

The **Core system** package uses namespace ***dabc::***. It defines all base classes and interfaces, and implements basic functionalities for object organization, memory management, thread control, and event communication. Section 1.5.1 gives a brief overview of the **Core system** classes.

1.4.2 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 ***dabc::Manager***.

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

1.4.3 Plugin packages

Plugin packages may provide special implementations of the core interface classes:
dabc::Device, ***dabc::Transport***, ***dabc::Module***, or ***dabc::Application***. Usually, these classes are made available to the system by means of a corresponding ***dabc::Factory*** that is automatically registered in the ***dabc::Manager*** when loading the plugin library.

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.

1.4.3.1 Bnet package

This package uses namespace ***bnet::***. It 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 (***bnet::WorkerApplication***) and controller nodes (***bnet::ClusterApplication***). These may be used as base classes in further **Application packages**. Section 1.5.2 gives a brief overview of the **Bnet package** classes; chapter 7 describes an example using the Bnet plugins.

1.4.3.2 Transport packages

Depend on the **Core system**, and may depend on external libraries or hardware drivers. Implement ***dabc::Device*** and ***dabc::Transport*** classes for specific data transfer mechanism, e.g. **verbs** or **tcp/ip socket**. May also implement ***dabc::Device*** and ***dabc::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.

1.4.4 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 ***dabc::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 (e.g. subclass of ***bnet::WorkerApplication***). Additionally, they may provide experiment specific ***dabc::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**.

1.4.5 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

1.5 Main Classes

1.5.1 Core system

The most important classes of the *DABC* core system are described in the following.

dabc::Basic : The base class for all objects to be kept in *DABC* collections (e. g. ***dabc::Folder***).

dabc::Command : Represents a command object. A command is identified by its name which it keeps as text string. Additionally, a command object may contain any number of arguments (integer, double, text). These can be set and requested at the command by their names. The available arguments of a special command may be exported to the control system as ***dabc::CommandDefinition*** objects. A command is sent from a ***dabc::CommandClient*** object to a ***dabc::CommandReceiver*** object that executes it in its scope. The result of the command execution may be returned as a reply event to the command client. The manager is the standard command client that distributes the commands to the command receivers (i.e. module , manager, or device). See chapter 3.3 for more details on the command mechanisms.

dabc::Parameter : Parameter object that may be monitored or changed from control system. Any ***dabc::WorkingProcessor*** implementation may register its own parameters. Parameter can be used for configuration of object at creation time (via configuration file), monitoring of object properties in GUI or manipulating of object properties at runtime, changing parameter values via controlling interface. Currently supported parameter types are:

- ***dabc::IntParameter*** - simple integer value
- ***dabc::DoubleParameter*** - simple double value
- ***dabc::StrParameter*** - simple string value
- ***dabc::StateParameter*** - contains state record, e. g. current state of the finite state machine and associated colour for gui representation
- ***dabc::InfoParameter*** - contains info record, e. g. system message and associated properties for gui representation
- ***dabc::RateParameter*** - contains data rate record and associated properties for GUI representation. May be updated in predefined time intervals.
- ***dabc::HistogramParameter*** - contains histogram record and associated properties for GUI representation.

dabc::WorkingThread : An object of this class represents a system thread. The working thread may execute one or several jobs; each job is defined by an instance of ***dabc::WorkingProcessor***. The working thread waits on an event queue (by means of pthread condition) until an event for any associated working processor is received; then the corresponding event action is executed by calling *ProcessEvent()* of the corresponding working processor.

dabc::WorkingProcessor : Represents a runnable job. Each working processor is assigned to one working thread instance; this thread can serve several working processors in parallel. In a special mode a processor can also run its explicit main loop. ***dabc::WorkingProcessor*** is a subclass of ***dabc::CommandReceiver***, i.e. a working processor may receive and execute commands in its scope.

dabc::Module : A processing unit for one "step" of the dataflow. Is subclass of ***dabc::WorkingProcessor***, i. e. the module may be run by an own dedicated thread, or a working thread may execute several modules that are assigned to it. A module has ports as connectors for the incoming and outgoing data flow.

dabc::ModuleSync : Is subclass of ***dabc::Module***; defines interface for a synchronous module that is allowed to block. User must implement virtual method *MainLoop()* that uses a dedicated working thread to run. Method *TakeBuffer()* provides blocking access to a memory pool. Blocking methods *dabc::ModuleSync::Send()* and *dabc::ModuleSync::Receive()* are used from the *MainLoop()* code to send (or receive) buffers over (or from) a ports.

dabc::ModuleAsync : Subclass of ***dabc::Module***; defines interface for an asynchronous module that must never block the execution. Several ***dabc::ModuleAsync*** objects may be assigned to one working thread. User must either re-implement virtual method *ProcessUserEvent()* wich is called whenever **any** event for this module (i.e. this working processor) is processed by the working thread. Or the user may implement callbacks for special events (e.g. *ProcessInputEvent()*, *ProcessOutputEvent()*, *ProcessPoolEvent()*,..) that are invoked when the corresponding event is processed by the working thread. The events are dispatched to these callbacks by the *ProcessUserEvent()* default implementation then. There are no blocking function available in ***dabc::ModuleAsync***; but the user **must** avoid any polling loops, waiting for resources - event processing function must be returned as soon as possible.

dabc::Port : A connection interface between module and transport. From inside the module scope, only the ports are visible to send or receive buffers by reference. Data connections between modules (i.e. transports between the ports of the modules) are set up by the application using methods of ***dabc::Manager*** which specify the full module/port names. For ports on different nodes, commands to establish a connection may be send remotely (via controls layer, e.g. DIM) and handled by the manager of each node.

dabc::Transport : A producing or consuming entity for buffers, which it delivers to (or receives from, resp.) a ***Module*** via the ***Port*** interface. As an example, ***dabc::NetworkTransport*** implements the transport between modules on different nodes.

dabc::Device : Device class used for creation and configuration of transport objects. Is a subclass of ***dabc::WorkingProcessor***. The ***dabc::Transport*** and ***dabc::Device*** base classes have various implementations:

- ***dabc::LocalTransport*** and ***dabc::LocalDevice*** for memory transport within same process
- ***dabc::SocketTransport*** and ***dabc::SocketDevice*** for tcp/ip sockets
- ***verbs::Transport*** and ***verbs::Device*** for InfiniBand **verbs** connection
- ***pci::Transport*** and ***pci::BoardDevice*** for DMA I/O from PCI or PCIe boards

dabc::Manager : Is manager of everything in DABC. There is the only instance of manager in the process scope, availible via *dabc::mgr()* or *dabc::Manager::Instance()* functions. It combines different roles:

1. It is a manager of all ***dabc::Basic*** objects in the process scope. Objects (e. g. modules, devices, parameters) are kept in a folder structure and can be identified by full path name.
2. It defines the interface to the controls system (state machine, remote command communication, parameter export); this is to be implemented in a subclass. The manager handles the command and parameter flow locally and remotely: commands submitted to the local manager are directed to the command receiver where they shall be executed. If any parameter is changed, this is recognized by the manager and optionally forwarded to the associated controls system. Current implementations of manager are:
   - ***dabc::Manager*** provides base manager functionality, can only be used for single-node application without any controlling possibilities.
   - ***dabc::StandaloneManager*** provides simple socket controls connection between several node in multi-node cluster, cannot be used with GUI.

- ***dimc::Manager*** Provides DIM [?] as transport layer for controlling commands. Additionally, parameters may be registered and updated automatically as DIM services. There is a general purpose Java GUI for this implementation.

3. It provides interfaces for user specific plug-ins that define the actual set-up:
   several ***dabc::Factory*** objects to create objects, and one ***dabc::Application*** object to define the state machine transition actions.

dabc::Factory : Factory plug-in for creation of applications, modules, devices, transports and threads.

dabc::Application : Defines user actions on transitions of the finite state machine of the manager. Good place for export of application-wide configuration parameters. May define additional commands.

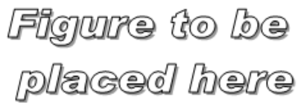


Figure 1.3: Simplified *core* classes diagram

1.5.2 BNET classes

The classes of the Bnet package, providing functionalities of the event builder network.

bnet::ClusterApplication : Subclass of ***dabc::Application*** to run on the cluster controller node of the builder network.

1. It implements the master state machine of the Bnet. The controlling GUI usually sends state machine commmands to the controller node only; the Bnet cluster application works as a command fan-out and state observer of all worker nodes.
2. It controls the traffic scheduling of the data packets between the worker nodes by means of a data flow controller (class ***bnet::GlobalDFCModule***). This controller module communicates with the Bnet sender modules on each worker to let them send their packets synchronized with all other workers.
3. It may handle failures on the worker nodes automatically, e. g. by reconfiguring the data scheduling paths between the workers.

bnet::WorkerApplication : Subclass of ***dabc::Application*** to run on the worker nodes of the builder network.

1. Implements the local state machine callbacks for each worker with respect to the Bnet functionality.
2. It registers parameters to configure the node in the Bnet, and methods to set and check these parameters.
3. Defines factory methods *CreateReadout()*, *CreateCombiner()*, *CreateBuilder()*, *CreateFilter()*, *CreateStorage()* to be implemented in user specific subclass. These methods are used in the worker state machine of the Bnet framework.

bnet::GeneratorModule : Subclass of ***dabc::ModuleSync***. Framework class to fill a buffer from the assigned memory pool with generated (i.e. simulated) data.

1. Method *GeneratePacket(buffer)* is to be implemented in application defined subclass (e.g. ***bnet::MbsGeneratorModule***) and is called frequently in module's *MainLoop()*.
2. Each filled buffer is forwarded to the single output port of the module.

bnet::CombinerModule : Subclass of ***dabc::ModuleSync***. Framework prototype class to format inputs from several readouts to one data frame (e.g. combine an event from subevent readouts on that node).

1. It provides memory pools handles and one input port for each readout connection (either ***bnet::GeneratorModule*** or connection to a readout transport).
2. Creates output port for combined subevents.
3. The formatting functionality is to be implemented in method *MainLoop()* of user defined subclass (e.g. ***bnet::MbsCombinerModule***).

bnet::SenderModule : Subclass of ***dabc::ModuleAsync***. Responsible for sending the subevents data frames to the receiver nodes, according to the network traffic schedule as set by the Bnet cluster plugin.

1. It has **one** input port that gets the event packets (or time sorted frames) from the preceding Bnet combiner module. The input data frames are buffered in the Bnet sender module and analyzed which frame is to be sent to what receiver node. This can be done in a non-synchronized "round-robin" fashion, or time-synchronized after a global traffic schedule as evaluated by the Bnet cluster plugin.
2. Each receiver node is represented by one output port of the Bnet sender module that is connected via a network transport (tcp socket, *InfiniBand verbs*) to an input port of the corresponding Bnet receiver node.

bnet::ReceiverModule : Subclass of ***dabc::ModuleAsync***. Receives the data frames from the Bnet sender modules and sorts together packets, belonging to the same events (or time frames, resp.).

1. It has **one** input port **for each sender node** in the Bnet. The data frames are buffered in the Bnet receiver module until the corresponding frames of all senders have been received; then received frames are send sequentially to the output port.
2. It has **exactly one** output port. This is connected to the ***bnet::BuilderModule*** implementation that performs the actual event building task.

bnet::BuilderModule : Subclass of ***dabc::ModuleSync***. Framework prototype class to select and build a physics event from the data frames of all Bnet senders as received by the receiver module.

1. It has **one** input port connected to the Bnet receiver module. The data frame buffers of all Bnet senders are transferred serially over this port and are then kept as an internal **std::vector** in the Bnet builder module.
2. Method *DoBuildEvent()* is to be implemented in user defined subclass (e. g. ***bnet::MbsBuilderModule***) and is called in module's *MainLoop()* when a set of corresponding buffers is complete.
3. It provides **one** output port that may connect to a Bnet filter module, or a user defined output or storage module, resp.
4. The user has to implement the sending of the tagged events to the output port explicitly in his subclass.

bnet::FilterModule : Subclass of ***dabc::ModuleSync***. Framework prototype class to filter out the incoming physics events according to the experiment's "trigger conditions".

1. Has **one** input port to get buffers with already tagged physics events from the preceding Bnet builder module.
2. Has **one** output port to connect a user defined output or storage module, resp.
3. Method *TestBuffer(buffer)* is to be implemented in user defined subclass (e. g. ***bnet::MbsFilterModule***) and is called in module's *MainLoop()* for each incoming buffer. Method should return `true` if the event is "good" for further processing.
4. Forwards "good" buffers to the output port and discards others.

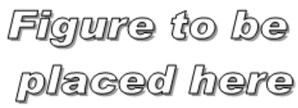


Figure 1.4: Simplified *Bnet* class diagram

Chapter 2

DABC Programmer Manual: Manager

[programmer/prog-manager.tex]

2.1 Introduction

The ***dabc::Manager*** is the central singleton object of the *DABC* framework. It combines a number of different roles, such as:

- objects manager;
- memory pools manager;
- threads manager;
- commands dispatcher;
- run control state manager;
- plug-in manager for factories and application;
- implementation of control and configuration system

Although these functionalities internally could as well be treated in separate classes, ***dabc::Manager*** class defines the common application programmer's interface to access most of these features. Since the manager is a singleton, these methods are available everywhere in the user code by means of the static handle *dabc::Manager::mgr()->*.

The following section 2.2 describes such interface methods to be used by the programmer of the ***Module***, ***Transport***, ***Device***, and ***Application*** classes. In contrast to this, section 2.3 gives a guide how to re-implement the ***Manager*** class itself for a different control and configuration system. This should be seldomly necessary for the common DAQ designer, but is added here as a reference and as useful insight into the *DABC* mechanisms.

2.2 Framework interface

2.2.1 General object management

All objects are organized in a folder structure and can be accessed by the full path name. However, for most purposes it is recommended to rather use higher level ***Manager*** methods to cause some action(e. g. *StartModule()*) than to work directly with the primitive objects.

Module* FindModule(const char* name) : Access to a ***Module*** by name. Returns 0 if module does not exist.

Port* FindPort(const char* name) : Access to a ***Port*** by name. Returns 0 if port does not exist.

Device* FindDevice(const char* name) : Access to a ***Device*** by name. Returns 0 if device does not exist.

Device* FindLocalDevice() : Shortcut to get the "local device" that is responsible for basic transport mechanisms like transport of buffers through the local memory.

Factory* FindFactory(const char* name) : Access to a ***Factory*** by name. Returns 0 if factory does not exist.

WorkingThread* FindThread(const char* name, const char* required_class = 0) : Access to a ***WorkingThread*** by name. The *required_class* string may be specified to check if the working thread implementation matches the client intentions. Returns 0 if thread object does not exist, or if it does not fullfill *required_class*.

Application* GetApp() : Access to the unique ***Application*** Object of this node.

2.2.2 Factory methods

Since all *DABC* objects are provided by ***dabc::Factory*** plug-ins, the application programmer needs to invoke corresponding factory methods to instantiate them. However, the factories themselves should not be accessed by the user code (although the ***Manager*** offers a getter method, see section 2.2.1). Instead, creation and registration of the key objects, like ***Module*** or ***Device***, is done transparently by the ***Manager*** within specific creation methods. These will scan over all existing factories whether the corresponding factory method can provide an object of the requested class name. In this case the object is created, kept in the object manager, and may be addressed by its full name later.

bool CreateModule(const char* classname, const char* modulename, const char* thrdname = 0) : Instantiate a ***Module*** of class *classname* with the object name *modulename*. Optionally, the name of the working thread *thrdname* may be specified that shall run this module. If a thread of this name is already exisiting, it will be also applied for the new module; otherwise, a new thread of the name will be created. If *thrdname* is not defined, *DABC* will use a new module thread automatically with an internal name. Returns true or false depending on the instantiation success.

bool CreateDevice(const char* classname, const char* devname) : Instantiate a ***Device*** of class *classname* with the object name *devname*. Returns true or false depending on the instantiation success.

bool CreateTransport(const char* portname, const char* transportkind, const char* thrdname = 0) : Instantiate a ***Transport*** from the ***Device*** of type *transportkind* (e. g. "PCI-Device") and connect it to the port of full name *portname* (e. g. "Readout1/Input"). Optionally the name of the working thread *thrdname* may be specified that shall run this transport. If a thread of this name is already exisiting, it will be also applied for the new transport; otherwise, a new thread of the name will be created. If *thrdname* is not defined, *DABC* will use a new thread automatically with an internal name. Returns true or false depending on the instantiation success.

FileIO* CreateFileIO(const char* typ, const char* name, int option) : Returns a new ***FileIO*** of type *typ* (e. g. "posix") with name *name*. The *option* value may define the file access option, e. g. ReadOnly, ReadWrite, WriteOnly, Create, and Recreate. Current standard implementation is ***dabc::PosixFileIO*** which is provided in the manager by default. Returns 0 if instantiation of desired type fails.

bool CreateApplication(const char* classname = 0, const char* appthrd = 0) : Instantiate the ***Application*** of class *classname*. Optionally the name *appthrd* of the main application thread may be specified. To be used in the *main()* function of the runtime executable on inititialization time.

2.2.3 Module manipulation

void StartModule(const char* modulename) : Enables the module of name *modulename* for processing. Depending on the ***Module*** type (synchronous or asynchronous, see section 1.2.1), this will start execution of the *MainLoop()*, or activate processing of the queued events belonging to this module, resp.

void StopModule(const char* modulename) : Disables processing for the module of name *modulename*.

bool StartAllModules(int appid = 0) : Enables processing for all modules with identifier number *appid*. The optional identifier may be set in the *Module* definition to select different kinds of modules here. By default, this method will start all exisiting modules on this node. Returns true of false depending on success.

bool StopAllModules(int appid = 0) : Disables processing for all modules with identifier number *appid*. The optional identifier may be set in the *Module* definition to select different kinds of modules here. By default, this method will stop all exisiting modules on this node. Returns true of false depending on success.

bool DeleteModule(const char* modulename) : Deletes the module of name *modulename*. Returns true of false depending on the deletion success.

bool IsModuleRunning(const char* modulename) : Method returns true if module of name *modulename* is running, i. e. its processing is enabled. If module does not exist or is not active, false is returned.

bool IsAnyModuleRunning() : Method returns false if **no** exisiting module is running anymore. Otherwise returns true.

bool ConnectPorts(const char* port1name, const char* port2name, const char* devname=0) :
Connects module ***Port*** of full name *port1name* with another module ***Port*** of full name *port2name*. A full port name consists of the module name and a local port name, separated by forward slash, e. g. "Readout3/Output", "CombinerModule/Input2". Optionally the ***Device*** type for the connection ***Transport*** may be defined with argument *devname*. By default, the ports are connected with a FIFO-like transport of queued ***Buffer*** references in local memory, as managed by ***dabc::LocalDevice***.

2.2.4 Thread management

bool MakeThreadForModule(Module* m, const char* thrdname = 0) :
Creates a thread for module *m* and assigns module to this thread. If thread name *thrdname* is not specified, module name is used. Returns true of false depending on success.

bool MakeThreadFor(WorkingProcessor* proc, const char* thrdname = 0, unsigned startmode = 0) : Creates thread for working processor *proc* and assigns processor to this thread If thread name *thrdname* is not specified, a default name is used. Value of *startmode* specifies initial run state of the thread (currently, thread is started if $startmode > 0$).

2.2.5 Command submission

bool Submit(Command* cmd) : This method generally submits a command *cmd* for execution. The command is put in the queue of its command receiver working thread and is then asynchronously executed there. The ***Manager*** will either forward the command to its receiver, if such is specified as command parameter; or the ***Manager*** working thread itself will execute the command. Thus method does not block and returns `true` if it accepts the command for execution, otherwise `false`.

Command* LocalCmd(Command* cmd, const char* fullitemname = "") : Prepares the properties of command *cmd* for execution on the local node. The returned *Command** may be directly used for *Submit(cmd)* (e. g. `m.Submit(m.LocalCmd(new Command("Start"),"Generator")));` it may also be collected in a ***dabc::CommandsSet***. The command receiver is defined by the full name *fullitemname* in the object folder structure, e. g. "Modules/ReadoutModule1", or by a unique reduced name, resp.

Command* LocalCmd(Command* cmd, Basic* rcv) : Prepares the properties of command *cmd* for execution on the local node. The returned *Command** may be directly used for *Submit(cmd)* (e. g. `m.Submit(m.LocalCmd(new Command("Start"),generator)));` it may also be collected in a ***dabc::CommandsSet***. The command receiver *rcv* is passed directly by reference.

Command* RemoteCmd(Command* cmd, int nodeid, const char* itemname = "") : Prepares a command *cmd* for execution on a remote node. The returned *Command** may be directly used for *Submit(cmd)* (e. g. `m.Submit(m.RemoteCmd(new Command("Start"),3 ,"Generator")));` it may also be collected in a ***dabc::CommandsSet***. The execution node is specified by the unique *nodeid* in the DAQ cluster. The command receiver on that node is defined by the full name *fullitemname* in the object folder structure, e. g. "Modules/ReadoutModule1", or by a unique reduced name.

Command* RemoteCmd(Command* cmd, const char* nodename, const char* itemname = "") : Prepares a command *cmd* for execution on a remote node. The returned *Command** may be directly used for *Submit(cmd)* (e. g. `m.Submit(m.RemoteCmd(new Command("Start"),"node01","Generator")));` it may also be collected in a ***dabc::CommandsSet***. The execution node is specified by the unique *nodename* as defined by the configuration system (see section 2.3.2). The command receiver on that node is defined by the full name *fullitemname* in the object folder structure, e. g. "Modules/ReadoutModule1", or by a unique reduced name.

bool SubmitLocal(CommandClientBase& cli, Command* cmd, const char* fullitemname = "") : Submits a command *cmd* for execution on the local node. The command receiver is defined by the full name *fullitemname* in the object folder structure, e. g. "Modules/ReadoutModule1". The caller must provide a command client object *cli* that will receive a command reply depending on the execution success. It is recommended to rather prepare a command by means of *LocalCmd()* and then use a plain *Submit()*, if command client functionality is not necessary.

bool SubmitLocal(CommandClientBase& cli, Command* cmd,Basic* rcv) : Submits a command *cmd* for execution on the local node. The command receiver *rcv* is passed directly by reference. The caller must provide a command client object *cli* that will receive a command reply depending on the execution success. It is recommended to rather prepare a command by means of *LocalCmd()* and then use a plain *Submit()*, if command client functionality is not necessary.

bool SubmitRemote (CommandClientBase& cli, Command* cmd, int nodeid, const char* itemname = "") : Submits a command *cmd* for execution on a remote node. The execution node is specified by the unique *nodeid* in the DAQ cluster. The command receiver on that node is defined by the full name *fullitemname* in the object folder structure, e. g. "Modules/ReadoutModule1", or by a unique reduced name. The caller must provide a command client object *cli* that will receive a command reply depending on the execution success. It is recommended to rather prepare a command by means of *RemoteCmd()* and then use a plain *Submit()*, if command client functionality is not necessary.

bool SubmitRemote (
CommandClientBase& cli, Command* cmd, const char* nodename, const char* itemname = "") : Submits a command *cmd* for execution on a remote node. The execution node is specified by the unique *nodename* as defined by the configuration system (see section 2.3.2). The command receiver on that node is defined by the full name *fullitemname* in the object folder structure, e. g. "Modules/ReadoutModule1", or by a unique reduced name. The caller must provide a command client object *cli* that will receive a command reply depending on the execution success. It is recommended to rather prepare a command by means of *RemoteCmd()* and then use a plain *Submit()*, if command client functionality is not necessary.

2.2.6 Memory pool management

bool CreateMemoryPool (
const char* poolname, unsigned buffersize, unsigned numbuffers,
unsigned numincrement, unsigned headersize, unsigned numsegments) : Instantiates a ***MemoryPool*** of name *poolname*, with *numbuffers* buffers of size *buffersize*. If a pool of this name already exists, it may be extended up to the specified buffer number. The *numincrement* value specifies with how many buffers at once the memory pool can optionally be extended on the fly. Optional arguments *headersize* and *numsegments* may define the buffer header size, and the partition of the buffer segments, resp. The ***MemoryPool*** mechanisms are discussed in detail in section 3.1. Method returns true or false depending on success.

MemoryPool* FindPool(const char* name) : Access to memory pool by name *name*. Returns 0 if not found.

bool DeletePool(const char* name) : Delete memory pool of name *name*. Returns true or false depending on success.

2.2.7 Miscellaneous methods

bool CleanupManager(int appid = 0) : Safely deletes all modules, memory pools and devices with specified application id *appid*. The default id 0 effects on all user components. In the end all unused threads are also destroyed.

virtual void DestroyObject(Basic* obj) : Deletes the referenced object *obj* in manager thread. Useful as safe replacement for call "delete this".

void Print() : Displays list of running threads and modules on stdout.

2.3 Control system plug-in

For the common *DABC* usage, the provided standard control and configuration system, featuring DIM protocol [?], XML setup files, and a generic Java GUI, will probably be sufficient. However, if e. g. an experiment control system is already existing and the data acquisition shall be handled with the same means, it might be necessary to adjust *DABC* to another controls and configuration framework. Moreover, future developments may replace the current standard control system by a more powerful, or a more convenient one.

Because of this, the connection between the *DABC* core system and the control system implementation was designed with a clear plug-in interface. Again the ***dabc::Manager*** class plays here a key role.

This section covers all methods and mechanisms for the control system plug-in. As an example, part 2.3.3 describes in detail the standard implementation as delivered with the *DABC* distribution .

2.3.1 Factory

A new control system plug-in is added into *DABC* by means of a ***dabc::Factory*** subclass that defines the method *bool CreateManagerInstance(const char* kind, dabc::Configuration* cfg)*. This method should create the appropriate ***dabc::Manager*** instance and return `true` if the name *kind*, as specified by the runtime environment, matches the implementation. The default *DABC* runtime executable will also pass a configuration object *cfg* read from an XML file which may be passed to the constructor of the ***Manager***.

As it's mandatory for other *DABC* factories, the ***dabc::Factory*** for the manager must be instantiated as global object in the code that implements it. This assures that the factory exists in the system on loading the corresponding library.

2.3.2 Manager

Besides its role as a central singleton to access framework functionalities, the ***dabc::Manager*** is also the interface base class for the control and configuration system that is applied with *DABC* .

2.3.2.1 Virtual methods

The ***dabc::Manager*** defines several virtual methods concerning the *finite state machine*,the registration and subscription of parameters, the command communication in-between nodes, and the management of a DAQ cluster, resp. These methods have to be implemented for differrent kinds of control systems in an appropriate subclass and are described as follows:

Manager(const char* managername, bool usecurrentprocess, Configuration* cfg) : The constructor of the subclass. The recommended parameters are passed from the manager factory (see section 2.3.1) to the baseclass constructor, such as the object name of the manager; optionally a flag indicating to use either the main process or another thread for manager command execution; and an optional configuration object *cfg*.

1. The constructor should initialize the control system implementation.
2. If the default state machine module of the *DABC* core is used, the constructor should invoke method *InitSMmodule()*. Otherwise, the constructor must initialize an external state machine of the control system, following the state and transition names defined as static constants in `dabc/Manager.h`.
3. The constructor must call method *init()* to initialize the base functionalities and parameters. This should be done *after* the control system is ready for handling parameters and commands, and *after* the optional *InitSMmodule()* call.

˜Manager() : The destructor of the subclass. It should cleanup and remove the control system implementation. It must call method *destroy()* at the end.

bool InvokeStateTransition(const char* state_transition_name, Command* cmd) : This should initiate the state transition for the given *state_transition_name*. This **must** be an asynchronous function that does not block the calling thread, possibliy the main manager thread if the state transition is triggered by a command from a remote "master" state machine node. Thus the actual state transition should be performed in a dedicated state-machine thread, calling the synchronous method *DoStateTransition(const char*)* of the base class (see section 2.3.2.2).

Synchronization of the state with the invoking client is done by the passed command object reference *cmd*. This should be used as handle in the static call *dabc::Command::Reply(cmd,true)* when the state transition is completed, or *dabc::Command::Reply(cmd,false)* when the transition has been failed, resp.

Note that base class ***dabc::Manager*** already implements this method for the *DABC* default state machine module which is activated in the manager constructor with *InitSMmodule()*. **It needs a re-implementation only if an external state machine shall be used.**

void ParameterEvent(dabc::Parameter* par, int event) : Is invoked by the framework when any ***Parameter*** is created (argument value $event = 0$), changed ($event = 1$), or destroyed ($event = 2$), resp. Pointer *par* should be used to access parameter name and value for export to the control system.

void CommandRegistration(dabc::Module* m, dabc::CommandDefinition* def, bool reg) : Is invoked by the framework when any module exports (argument *reg* true), or unexports (argument *reg* false) a ***dabc::Command*** to, or from the control system, resp. This allows to invoke such commands via the controls connection from a remote node. The command definition object *def* contains a description of possible command parameters; pointer *m* should be used to access the owning module and get its name. This information may be used to represent the command within the controls implementation.

bool Subscribe(dabc::Parameter* par, int remnode, const char* remname) : This method shall link the value of a local parameter *par* to a remote parameter of name *remname* that exists on node number *remnode* of the DAQ cluster. Control system implementation may use a publisher-subscriber mechanism here to update the local subscription whenever the remote parameter changes its value.

The actual update handler must call method *InvokeChange(const char* val)* of the corresponding local representation *dabc::Parameter* par* then. The new value *val* is passed as (`printf()` style formatted) text representation to the parameter which will change itself appropriately. This decouples the parameter change from the invoking control system callback in a thread-safe manner.

bool Unsubscribe(dabc::Parameter* par) : The subscription of a local parameter *par* to a remote paramter by a formerly called *Subscribe()* is removed from the control system.

bool IsMainManager() : Should return true if this node is the single master controller node of the DAQ cluster. This node will define the master state machine that rules the states of all other nodes. Otherwise (returns false) this node is a simple worker node. The node properties should be taken from the configuration.

bool HasClusterInfo() : Returns true if this node has complete information of the DAQ cluster.

int NumNodes() : Returns the number of all DAQ nodes in the cluster. This may be taken from a configuration database, e. g. an XML file, but may also test the real number of running nodes each time it's called.

int NodeId() const : Returns the unique id number of this node in the DAQ cluster. This should be taken from the cluster configuration.

bool IsNodeActive(int num) : Returns true if DAQ cluster node of id number *num* is currently active, otherwise false. This may allow to check on runtime if some of the configured nodes are not available and should be excluded from the DAQ setup.

const char* GetNodeName(int num) : For each DAQ cluster node of id number *num*, this method must define a unique name representation. The name should represent the node in a human readable way, e. g. by means of URL and a functional node description ("daq01.gsi.de-readout"). It should match the description in the cluster configuration. Note: **This name must match the local name of the manager object on each node**.

bool SendOverCommandChannel(const char* managername, const char* cmddata) :
This method sends a ***dabc::Command*** as a streamed text representation *cmddata* to a remote DAQ cluster node of name *managername*. The *managername* argument must match one of the names defined in *GetNodeName(int num)*. The implementation should use transport mechanisms of the control system to transfer the command string to the remote site (e. g. native control commands that wrap *cmddata*). The receiver of such commands on the target node should call base class method *RecvOverCommandChannel(const char* cmddata)* to forward the command representation to the core system, which will reconstruct and execute the ***dabc::Command*** object.

bool CanSendCmdToManager(const char* mgrname) : Returns true if it is possible to send a remote command to the manager on DAQ cluster node of name *mgrname*, otherwise false. The node name argument must match one of the names defined in *GetNodeName(int num)*. This method may implement to forbid the sending of commands on some nodes.

int ExecuteCommand(dabc::Command* cmd) : This method executes synchronously any *DABC* command that is submitted to this manager itself. It will run in the scope of the manager thread (depending on constructor argument *usecurrentprocess*, this is either the main process thread, or a dedicated manager thread).

It may be re-implemented to add new commands required for the controls implementation. The *DABC* mechanism of methods *SubmitCommand()* and *ExecuteCommand()* may allow to decouple control system callbacks from their execution thread.

2.3.2.2 Baseclass methods

In addition to the virtual methods to be implemented in the manager subclass, there is a number of ***dabc::Manager*** base class methods that should be called from the control system to perform actions of the framework:

bool DoStateTransition(const char* state_transition_cmd) : Performs the state machine transition of name *state_transition_cmd*. This method is synchronous and returns no sooner than the transition actions are completed (true) or an error is detected (false). Note that the real transition actions are still user defined in methods of the ***dabc::Application*** implementation.

bool IsStateTransitionAllowed(const char* state_transition_cmd, bool errout) : Checks if state transition of name *state_transition_cmd* is allowed for the default state machine implementation (which should be reproduced exactly by any external SM implementation) and returns true or false, resp. Argument *errout* may specify if error messages shall be printed to `stdout`.

void RecvOverCommandChannel(const char* cmddata) : Receives a *DABC* command as text stream *cmddata* from a remote node. Usually this function should be called in a receiving callback of the control system communication layer, passing the received command representation to the core system. Here the command object is unstreamed again, forwarded to its receiver and executed.

This is the pendant to virtual method *SendOverCommandChannel()* which should implement the **sending** of a streamed command from the core to a remote manager by transport mechanisms of the control system.

2.3.3 Default implementation for DIM

The *DABC* default controls and configuration system is based on the DIM library [?] and is marked by namespace ***dimc::*** (for "DIM Control"). The main classes are described in the following:

2.3.3.1 *dimc::Manager*

Implements the control system interface of ***dabc::Manager*** as described above.

1. It uses the **default state machine module** of the *DABC* core system. This is activated in the constructor by calling *InitSMmodule()*. Thus virtual method *InvokeStateTransition()* is **not** re-implemented here.
2. It exports a dedicated ***dabc::StatusParameter*** that is synchronized with the value of the core state machine in *ParameterEvent()*. This parameter is required to display the state of the node on the generic Java GUI.
3. It applies the generic ***dabc::Configuration*** for setting up the node properties. The standard executable `dabc_run` will create this object from parsing an XML file.
4. The other interface functionalities use one component of class ***dimc::Registry***.

2.3.3.2 *dimc::Registry*

The main component of the ***dimc::Manager*** that offers service methods really implementing the manager interface. It registers all parameters, commands, and subscriptions; and it defines the allowed access methods for the DIM server itself.

1. The **DIM server** is instantiated in the constructor as ***dimc:Server*** singleton. Methods *StartDIMServer()* and *StopDIMServer()* actually initiate and terminate the service.
2. **Naming of nodes and services:** Method *GetNodeName(int num)* of ***dimc::Manager*** uses *CreateDIMPrefix(num)* of ***dimc::Registry***. This evaluates the unique name for node number *num* from the ***dabc::Configuration*** object: It consists of a global prefix ("DABC"), the configuration *NodeName()*, and the *ContextName()* property of the node id, all separated by forward slashes ("/").
The node name is also taken as prefix for the helper methods *BuildDIMName()* (*ReduceDIMName()*, resp.) that transform local *DABC* parameter and command names into unique DIM names (and back, resp.). Moreover, methods *CreateFullParameterName()* (*ParseFullParameterName()*, resp.) define how the local parameter name itself is composed (decomposed, resp.) from the names of its parent module and its internal variable name. They utilize corresponding static methods of class ***dimc::nameParser*** in a thread-safe way.
3. **Parameter export:** *dimc::Manager::ParameterEvent()* uses methods *RegisterParameter()* (and *UnregisterParameter()*, resp.) to declare (undeclare, resp.) a corresponding DIM service. Here the ***dimc::Registry*** keeps auxiliary objects of class ***dimc::ServiceEntry*** that link the ***DimService*** with

the ***dabc::Parameter*** (see section 2.3.3.4). On parameter change, method *ParameterUpdated()* will initiate an update of the corresponding DIM service.

4. **Control system commands:** Method *DefineDIMCommand(const char* name)* creates and registers simple (char array) ***DimCommand*** objects that may be executed on this node. The ***dimc::Registry*** constructor defines commands for all state machine transitions, such as Configure, Enable, Halt, Start, Stop. Additionally, there are DIM commands for shutting down the node, setting a parameter value, and wrapping a *DABC* command as string representation (*"ManagerCommand"* for *SendOverCommandChannel()*, see section 2.3.2), resp.
Moreover, a *DABC* module may register a ***dabc::Command*** as new control system command on the fly. In this case ***dimc::Manager*** method *CommandRegistration()* will use *RegisterModuleCommand()* of ***dimc::Registry***. This will both define a ***DimCommand***, and publish a corresponding command descriptor as DIM service to announce the command structure to the generic Java GUI. Method *UnRegisterModuleCommand()* may remove command and descriptor service again.
When the DIM server receives a remote command, method *HandleDIMCommand()* checks if this command is registered; then *OnDIMCommand()* will transform the ***DimCommand*** into a ***dabc::Command*** and *Submit()* this to the Manager. The actual command execution will thus happen in re-implemented method *ExecuteCommand()* of ***dimc::Manager***. Thus the command action runs independent of the DIM commandhandler thread.
5. **Parameter subscription:** Method *Subscribe()* (*Unsubscribe()*, resp.) of ***dimc::Manager*** are forwarded to *SubscribeParameter()* (UnsubscribeParameter() , resp.) of ***dimc::Registry***. These implement it by means of the ***DimService*** update mechanism. Subscriptions are kept as vector of ***dimc::DimParameterInfo*** objects (see section 2.3.3.5).
6. **Remote command execution:** Method *SendOverCommandChannel()* of ***dimc::Manager*** is forwarded to *SendDimCommand()* of ***dimc::Registry***. The streamed ***dabc::Command*** is wrapped as text argument into the DIM *ManagerCommand* and send to the destination by node name via *DimClient::sendCommand().*

2.3.3.3 *dimc::Server*

Subclass of DIM class ***DimServer***, implementing command handler, error handler, and exit handlers for client and server exit events.

1. Because most DIM server actions are invoked by static methods of ***DimServer***, it is reasonable to have only one instance of ***dimc::Server***; thus this class is designed as **singleton pattern**. Access and initial creation is provided by method *Instance().* A safe cleanup is granted by *Delete()* (ctors and dtors are private and cannot be invoked directly).
2. The ***dimc::Registry*** is set as "owner" of ***dimc::Server*** by means of a back pointer. All handler methods of the ***DimServer*** are implemented as forward calls to corresponding methods of the ***dimc::Registry*** and treated there, such as:
   - *commandHandler()* to *HandleDIMCommand()*
   - *errorHandler()* to *OnErrorDIMServer()*
   - *clientExitHandler()* to *OnExitDIMClient()*
   - *exitHandler()* to *OnExitDIMServer()*

2.3.3.4 *dimc::ServiceEntry*

This is a container to keep the ***DimService*** together with the corresponding ***dabc::Parameter*** object and some extra properties. The ***dimc::ServiceEntry*** objects are managed by the ***dimc::Registry*** and applied for the *RegisterParameter()* method.

1. For ***std::string*** parameters an internal ***char**** array is used as buffer which is actually exported as DIM service.
2. Method *UpdateBuffer()* updates the DIM service; it optionally may copy the parameter contents to the buffer before.
3. Method *SetValue()* sets the ***dabc::Parameter*** to a new value, as defined by a string expression.

2.3.3.5 *dimc::ParameterInfo*

A subclass of DIM class ***DimStampedInfo*** which subscribes to be informed if a remote DIM service changes its value. The ***dimc::ParameterInfo*** objects are managed by the ***dimc::Registry*** and applied for the *Subscribe()* method.

1. The ***dimc::ParameterInfo*** has a reference to a local ***dabc::Parameter*** object that shall be updated if the subscribed service changes.
2. Depending on the subscribing ***dabc::Parameter*** type (integer, double, string,...), the constructor will instantiate an appropriate ***DimStampedInfo*** type.
3. Method *infoHandler()* of ***DimStampedInfo*** is implemented to update the parameter to the new value by means of an *InvokeChange()* call.

Chapter 3

DABC Programmer Manual: Services

[programmer/prog-services.tex]

3.1 Memory management

3.1.1 Zero-copy approach

The *DABC* framework is based on a dataflow concept: Data buffers are flowing through many components like ***Modules***, ***Transports***, and ***Devices***. If it was required to copy the data content in each step of such transfer chain, this would reduce performance drastically. Therefore *DABC* has a central memory management that provides global memory **Buffers** from a **Memory Pool**. All components use just references to this memory; these can be passed further without copying the content. This technique is called **zero-copy approach** and is fully supported by *DABC* .

3.1.2 Memory pool

The memory in ***dabc::MemoryPool*** is organized in big *blocks* of contiguous virtual memory. Each *block* is divided into memory pieces of the same size, the *subblocks*; the size of each *subblock* is defined as a power of 2 (e. g. 4096 bytes). The ***MemoryPool*** can have several memory *blocks* with different *subblock* sizes.

Usually a ***MemoryPool*** has a fixed structure: the memory is allocated once and will not change during the complete run. This is the preferrable mode of operation, because any memory allocation may lead to an undefined execution time, or could even cause an error, if the system has too few resources. Nevertheless, one can configure a ***MemoryPool*** to be extendable: The ***MemoryPool*** will allocate new *blocks*, if it has no more memory available to provide a requested ***Buffer***.

Each *subblock* of the ***MemoryPool*** has a 32-bit reference counter which counts how many references to this memory region are in use by the ***Buffers***. This is necessary for book-keeping of available memory, since several ***Buffer*** objects can refer to the same *subblock*.

The user can request a new ***Buffer*** from the ***MemoryPool*** with method *TakeBuffer()*. This method returns a ***dabc::Buffer*** instance with an internal reference to a formerly unused *subblock* of the appropriate size. The reference counter of this *subblock* is incremented then. To release a ***Buffer***, one should call static method *dabc::Buffer::Release()*. This will delete the ***Buffer*** object and decrement again the *subblock* reference counter.

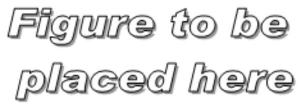


Figure 3.1: Schema of buffers with segmentts and memory pool with blocks and subblocks

3.1.3 Buffer

In the general case, ***dabc::Buffer*** contains a list of *segments* (gather list). Each *segment* (represented by class ***dabc::MemSegment***) refers to a different part of a *subblock* in the ***MemoryPool*** (compare section 3.1.2). The ***dabc::MemSegment*** contains a unique buffer id, the pointer to the *segment* begin, and the size of the *segment*.

Usually, a ***dabc::Buffer*** contains just one segment, which fully covers a complete *subblock* of the ***MemoryPool*** (for instance, when a new ***Buffer*** is requested with method *MemoryPool::TakeBuffer()*). Methods *NumSegments()* and *Segment(unsigned)* provide access to the list of segments. One can also directly access the pointer, and the size of each *segment*, via methods*GetDataLocation(unsigned)*, and *GetDataSize(unsigned)*, respectively. For instance, filling a complete ***Buffer*** with zeros will look like this:

```
#include "dabc/Buffer.h"

void UserModule::ProcessOutputEvent(dabc::Port* port)
{
   dabc::Buffer* buf = Pool()->TakeBuffer(2048);
   memset(buf->GetDataLocation(), 0, buf->GetDataSize());
   Output(0)->Send(buf);
}
```

It is also possible to create a mere reference ***Buffer*** object that uses the same memory of another ***Buffer***, by means of method *Buffer::MakeReference()*. This will deliver the pointer to a new ***dabc::Buffer*** instance with the same list of *segments* as the original instance. It will also increment the reference counter for all used *subblocks* in the ***MemoryPool***.

This method should be used e. g. to send the same data over several ***Ports***: one just makes as many reference ***Buffers*** as required and sends them to all destinations independently, without copying the data. For instance, a simplified version of *dabc::Module::SendToAllOutputs()* will look like:

```
void dabc::Module::SendToAllOutputs(dabc::Buffer* buf)
{
   for(unsigned n=0;n<NumOutputs();n++)
      Output(n)->Send(buf->MakeReference());
   dabc::Buffer::Release(buf);
}
```

The ***dabc::Buffer*** object has a 32-bit type identifier which can be set with method *SetTypeId()*, and can be retrieved with *GetTypeId()*. Its purpose is to identify the type of the buffer content. The value of this identifier is application specific - for instance, the *MBS* plugin defines its own type, which is then used by the transports to distinguish if the buffer contains an *MBS* event format.

Each ***dabc::Buffer*** can be supplied with an additional **header**. This is piece of memory which is allocated and managed by the pool separately from the main payload memory and in generally **should be smaller** than the payload memory. The idea of the buffer header is to add user-specific information to an already existing ***Buffer***, without changing the contained payload data, and even without touching the ***Buffer*** identifier. The header size can be set by *SetHeaderSize()* method; the pointer to the header can be obtained by *GetHeader()* method. The main difference between header memory and payload memory concerns the behaviour when the ***Buffer*** is send via a "zero copy" network transport implementation, like *InfiniBand verbs*: In contrast to the payload data, which will be transferred directly from the ***Buffer*** memory by DMA, the header contents will be explicitely copied first.

3.1.4 Pointer

Class ***dabc::Pointer*** provides a virtual contiguous access to segmented data which is referenced by a ***dabc::Buffer*** object. Using ***dabc::Pointer***, one should not care how many segments are referenced by the ***Buffer***, and how big they are. One can use following methods:

Pointer() or *reset()* initialize or reset the pointer as a reference of a ***dabc::Buffer***, of another ***dabc::Pointer***, or just of a simple memory region
ptr() or *operator()* the current memory pointer
rawsize() size of contiguous system memory from current pointer position
fullsize() size of full memory from current pointer position
shift() shift pointer
copyfrom() set pointed memory content from a ***dabc::Pointer***, or just from a memory region
copyto() copy pointed memory content into specified memory region

Example of pointer usage:

```
#include "dabc/Buffer.h"
#include "dabc/Pointer.h"

void UserModule::ProcessOutputEvent(dabc::Port* port)
{
   if (!Input(0)->CanRecv()) return;
   dabc::Buffer* buf = Input(0)->Recv();
   dabc::Pointer ptr(buf);
   uint32_t v = 0;
   while (ptr.fullsize()>0) {
      ptr.copyfrom(&v, sizeof(v));
      ptr.shift(sizeof(v));
      v++;
   }
   Output(0)->Send(buf);
}
```

3.1.5 Buffer guard

Class ***dabc::BufferGuard*** is the equivalent of a ***LockGuard*** for threads (see section 3.2.3), preventing memory leaks due to unreleased ***Buffers***. It should be used to automatically release a ***Buffer*** whenever the function scope is left, both by returning regularly, and by throwing an exception. One should **explic-**

itly take out the ***Buffer*** from the guard with *BufferGuard::Take()* to avoid such automatic release in a normal situation. A typical usage of ***dabc::BufferGuard*** is shown here:

```
...
dabc::BufferGuard buf = pool->TakeBuffer(2048);
...
port->Send(buf.Take());
...
```

Class ***dabc::ModuleSync*** provides several methods to work directly with ***dabc::BufferGuard*** - this allows to correctly release a ***Buffer*** in case of any exception, which otherwise may not be handled correctly by the user.

3.1.6 Allocation

There are several methods how a ***MemoryPool*** can be created:

- Automatically, when the user tries to access it via a ***PoolHandle*** the first time
- using *dabc::Manager::CreateMemoryPool()* method
- using dabc::CmdCreatePool command

Automatic creation is useful for simple applications with a few modules. In this case the parameters specified by the ***PoolHandle*** (size and number of buffers) are used.

But in many situations it is good to create a memory pool explicitly, setting all its parameters directly, or from a configuration file. Typically, the memory pool is created by the user's ***Application*** class in method *CreateAppModules()*, called by state change command **DoConfigure**. In simple case:

```
bool UserApplication::CreateAppModules()
{
   ...
   dabc::mgr()->CreateMemoryPool("WorkPool", 8192, 100);
   ...
}
```

One can call *CreateMemoryPool()* method several times to create memory *blocks* for different buffer sizes. As alternative, one can create and configure a command object dabc::CmdCreateMemoryPool where all possible settings can be done via following static methods:

AddMem() add configuration for specified buffer size
AddRef() add configuration for number of references and header sizes
AddCfg() set generic configuratio like cleanup timeout or size limit

For instance, one can do the following:

```
bool UserApplication::CreateAppModules()
{
   ...
   dabc::Command* cmd = new dabc::CmdCreateMemoryPool("WorkPool");
   dabc::CmdCreateMemoryPool::AddMem(cmd, 8192, 100);
   dabc::CmdCreateMemoryPool::AddMem(cmd, 2048, 500);
   dabc::CmdCreateMemoryPool::AddRef(cmd, 2048, 2000);
   dabc::CmdCreateMemoryPool::AddCfg(cmd, true); // set fixed layout
   dabc::mgr()->Execute(cmd);
```

```
    ...
}
```

All parameters, configured for the command, can be set up in the configuration file. In this case one should just call `dabc::mgr()->CreateMemoryPool("WorkPool")`.

3.2 Threads organization

Class ***dabc::WorkingThread*** organizes a working loop and performs execution of runnable jobs, represented by ***dabc::WorkingProcessor*** class.

3.2.1 Working loop

The implementation of ***dabc::WorkingThread*** is based on the *pthreads* library.

The main task of ***dabc::WorkingThread*** is to wait for events (using *pthread_cond_wait()* function), and then execute the event callback in the corresponding ***WorkingProcessor***. This functionality is implemented in *dabc::WorkingThread::MainLoop()*.

Usually events are produced by calling *dabc::WorkingProcessor::FireEvent()* method; this method can be invoked from any thread. All events are queued and a *pthread condition* is fired in this case. The thread, waiting for this condition, is woken up, and the next event from the queue will be delivered to the ***WorkingProcessor*** by calling virtual method *dabc::WorkingProcessor::ProcessEvent()*. Here any user-specific code can be implemented.

Another task of ***dabc::WorkingThread*** consists in **timeout handling**. Some ***WorkingProcessors*** may require to be invoked not only by events, but also after specified time intervals.

Method *dabc::WorkingProcessor::ActivateTimeout()* prepares the thread to execute *dabc::WorkingProcessor::ProcessTimeout()* after the specified time interval. This virtual method may also be implemented by the user.

3.2.2 Sockets handling

The POSIX sockets library provides the handling of all socket operations in an event-like manner, using the *select()* function. Such approach was used in ***dabc::SocketThread*** and ***dabc::SocketProcessor*** classes to provide handle several sockets in parallel from a single thread.

With each ***dabc::SocketProcessor*** a socket descriptor is associated which can deliver events like: "can read next portion of data from socket", "sending over socket will not block", "socket is broken", and so on. The main loop of ***dabc::SocketThread*** is modified such, that, instead of waiting for the *pthread condition*, the thread waits for the next event from the sockets.

Handling these events allows to send and receive of data via sockets in a non-blocking manner, i. e. one can run several socket operations in parallel with one thread.

At the same time, ***dabc::SocketThread*** class allows to run normal jobs, implemented with base class ***dabc::WorkingProcessor***. So within a ***SocketThread*** one can mix socket processors (like some ***Transports***) with normal processors (like ***modules***).

A similar approach was used to support the *InfiniBand verbs* API in DABC.

3.2.3 Mutex usage

All methods of ***dabc::WorkingThread*** and ***dabc::WorkingProcessor*** are thread safe (except for those started with underscore "_" symbol). So user code could avoid mutexes completely. But if data is shared between ***Processors*** which run in different threads, one should use mutexes though. Here it is recommended to work with a ***dabc::LockGuard***. This class takes care that the mutex will be unlocked automatically whenever the current function scope is left.

For instance, if one has global static variable associated with a mutex, one should implement a thread-safe setter method like this:

```
#include "dabc/threads.h"

int GlobalVariable = 0;
dabc::Mutex GlobalMutex;

void SetGlobalVariable(int newvalue)
{
   dabc::LockGuard guard(GlobalMutex);
   GlobalVariable = newvalue;
}
```

3.3 Command execution

The idea of command execution is to invoke user-specific code from any part of the system. There are several reasons to prefer a command interface over direct calls of class methods:

- The execution of a command is performed not in the context of the calling thread, but in the thread to which the command receiver object is assigned. This allows to avoid unnecessary mutex locking.
- The execution of a command can be performed synchronous or asynchronous to the calling thread, so one can easily specify a timeout for the command execution.
- A command can be submitted to any object in the system, including objects on remote nodes.
- The code that invokes the command execution does not strongly depend on the code that executes the command: the invoking client library must know a command base class and some common parameter names, but not the implementation of the execution itself. This allows to decouple the required libraries on different nodes.
- A command object can contain an arbitrary number of argument values, and can also be used to return any number of result values.

3.3.1 Command class

Class ***dabc::Command*** is a container for argument and result values. The name of the ***command*** is the main identifier for the command action which is executed in the ***CommandReceiver*** object.

There are a number of methods to set/get command parameters:

Type	Getter	Setter
string	GetStr()	SetStr()
int	GetInt()	SetInt()
unsigned int	GetUInt()	SetUInt()
bool	GetBool()	SetBool()
double	GetDouble()	SetDoble()

In all setter methods the first argument is the name of a command parameter, and the second is the new parameter value of the corresponding type. In all getter methods, the first argument is again the parameter name, and the second is an optional default parameter value. This default value is returned if a parameter of that name is not contained in the ***Command***. To instantiate a command, one should do:

```
...
dabc::Command* cmd = new dabc::Command("UserCommand");
cmd->SetInt("UserArg", 5);
...
```

Usually, the name of a ***Command*** defines the action which will be performed. There are several subclasses of ***dabc::Command*** (for instance, in file `dabc/Manager.h`), but these subclasses are only used to set the command name and command-specific parameters. There is no sense to define some extra methods in the subclass, since ***dabc::Command*** is designed as a mere container for parameters.

With method *ConvertToString()* one can convert a ***Command*** and all contained parameters in a plain string. Method *ReadFromString()* is used to reconstruct the ***Command*** object from a string. This feature is useful to transfer a ***Command*** over a network connection, or store it to a file.

3.3.2 Command receiver

Class ***dabc::CommandReceiver*** provides the interface for all classes which should execute a ***Command***. The main place for user code is virtual method *ExecuteCommand()* which gets a ***Command*** object as argument. A Typical implementation of this method looks like:

```
int UserModule::ExecuteCommand(dabc::Command* cmd)
{
   if (cmd->IsName("UserCommand")) {
      int v = cmd->GetInt("UserArg", 0);
      DOUT1(("Execute UserCommand with argument = %d", v));
      return dabc::cmd_true;
   } else
   if (cmd->IsName("UserGetCommand")) {
      DOUT1(("Execute UserGetCommand without arguments"));
      cmd->SetInt("UserRes", fCounter);
      return dabc::cmd_true;
   }

   return dabc::ModuleAsync::ExecuteCommand(cmd);
}
```

Method *ExecuteCommand* should analyse the command name and perform command-specific actions. It should return dabc::cmd_true if the command has been executed succesfully, or dabc::cmd_false otherwise.

The default implementation of ***dabc::CommandReceiver*** methods performs the command execution in the calling thread. However, most command actions may access resources which are also used by another working thread assigned to the***CommandReceiver*** object. In this case all command execution code had to protect these resources by mutex locks (see section 3.2.3), which would decrease performance. Because of this, class ***dabc::WorkingProcessor*** inherits from ***dabc::CommandReceiver***, and implements several virtual methods (like *IsExecutionThread()*, *IsExecutionThread(Submit())*) which are necessary to deliver and execute a command in the thread context of the assigned ***WorkingThread***. The user **must not reimplement** these methods again in the derived classes. In the *DABC* subclasses of ***dabc::WorkingProcessor***, like ***dabc::Module***, ***dabc::Application***, the custom commands will be executed in the appropriate thread context.

With method *Execute()* of class ***dabc::CommandReceiver*** one can execute a command directly in the receiving object. Here one can specify a ***dabc::Command*** object as argument, or just a command name, if the command has no arguments. Method *Execute()* will block until the command is executed - this is called the **synchronous** mode of command execution. Optionally, one can set a timeout - how long the calling thread will wait until the command is executed.

Method *Execute()* can only check if a command is executed succesfully or not, as it has a boolean return value. There are advanced methods *ExecuteInt()*, and *ExecuteStr()*, which return the result of a command execution as integer, or string value, resp. They will deliver the final value of the command parameter which is specified by name in the second function argument. For example, the result of command "UserGetCommand" execution from the previous example one can obtain like this:

```
...
dabc::Module* m = dabc::mgr()->FindModule("Module1");
int res = m->ExecuteInt("UserGetCommand", "UserRes");
...
```

There is an other way to execute a command - submit the ***Command*** with *Submit()* method (see also section 2.2.5) In this case the command will be executed **asynchronous** to the calling thread, therefore one cannot get any direct information about the result of command execution from the return value of *Submit()*.

3.3.3 Command client

To really work with asynchronous command execution, one should be able to analyse the result of such commands though. This can be done with class ***dabc::CommandClient***. Before submitted for execution, commands should be assigned to a ***dabc::CommandClient*** object. In this case, the ***CommandClient*** will get a callback from the ***Command*** when execution is done, and can react on this callback. One can assign more than one ***Command*** to a ***CommandClient***.

A first use case for the ***CommandClient***: if one needs to execute many commands at once. Using *Execute()* method, all commands can be executed **sequentially** only. By means of the ***CommandClient***, however, one can submit many commands first, and then wait for all of them to be executed. If the associated ***CommandReceivers*** run with different threads, the commands will be executed **in parallel**. For instance:

```
...
dabc::CommandClient cli;
for (unsigned n=0; n<10; n++) {
   dabc::Module* m = dabc::mgr()->FindModule(FORMAT(("Module%u",n)));
   dabc::Command* cmd = new dabc::Command("UserCommand");
   cli.Assign(cmd);
   m->Submit(cmd);
}
bool res = cli.WaitCommands();
...
```

This example submits 10 commands into 10 different modules, and waits at one place until all commands are executed.

Another use case for the ***CommandClient***: it keeps the ***Command*** object after execution and can analyse the contained result values. For instance, all 10 commands from previous example may return several values each. If one instantiates the ***CommandClient*** with `true` as constructor argument, at the end a list of all commands will be available via *ReplyedCmds()* method:

```
...
dabc::CommandClient cli(true);
...
bool res = cli.WaitCommands();
DOUT1(("One has %u commands in replyed queue", cli.ReplyedCmds().Size()));
...
```

One more use case of the command client interface is the ***dabc::CommandsSet***. This class inherits from ***dabc::CommandClientBase***, the abstract base class for all commands clients. It useful if execution of a "master" command should cause the execution of several other commands. For instance, when execution of a command in one module should be distributed to two other modules, one should do:

```
int UserModule::ExecuteCommand(dabc::Command* cmd)
{
   if (cmd->IsName("MasterCommand")) {
      dabc::CommandsSet* set = new dabc::CommandsSet(cmd);

      dabc::Module* m1 = dabc::mgr()->FindModule("Module1");
      m1->Submit(set->Assign(new dabc::Command("UserCommand1"));

      dabc::Module* m2 = dabc::mgr()->FindModule("Module2");
      m2->Submit(set->Assign(new dabc::Command("UserCommand2"));

      dabc::CommandsSet::Completed(set, 10.);

      return dabc::cmd_postponed;
   }

   return dabc::ModuleAsync::ExecuteCommand(cmd);
}
```

Here one creates a ***CommandsSet*** for a "master" command and submits two "slave" commands via the command client argument to two other modules. Method *dabc::CommandsSet::Completed()* is used to inform the framework that all commands are submitted and should be ready within 10 seconds. Return argument `dabc::cmd_postponed` indicates that the master command may not be ready when *ExecuteCommand()* is returned. Therefore ***dabc::CommandsSet*** will take care about the correct reply of the master command, either when all slaves are ready, or when the master command timeout has expired.

Chapter 4

DABC Programmer Manual: Plugins

[programmer/prog-plugin.tex]

4.1 Introduction

A multi purpose DAQ system like *DABC* requires to develop user specific code and adopt this into the general framework. A common object oriented technique to realize such extensibility consists in the definition of base classes as interfaces for dedicated purposes. The programmer may implement subclasses for these interfaces as **Plug-Ins** with the extended functionality that matches the data format, hardware, or other boundary conditions of the data-taking experiment. Moreover, the *DABC* core itself applies such powerful plug-in mechanism to provide generic services in a flexible and maintainable manner.

This chapter gives a brief description of all interface classes for the data acquisition processing itself. This covers the processing **Modules**, the **Transport** and **Device** objects that move data between the DAQ components, and the **Application** that is responsible for the node set-up and run control. A **Factory** pattern is used to introduce new classes to the framework and let them be available by name at runtime.

4.2 Modules

DABC provides ***dabc::Module*** class, which plays role of data processing entity in framework. In this class necessary components like pool handles, ports, parameters, timers are organised. Class ***dabc::Module*** has two subclasses - ***dabc::ModuleSync*** and ***dabc::ModuleAsync***, which provides two different paradigms of data processing: within explicit main loop, and via event processing, respectively. Before we discuss these two kinds of modules, let's consider components which can be used with both types of the module.

4.2.1 Pool handles

Class ***dabc::PoolHandle*** should be used in any module to communicate with ***dabc::MemoryPool***. By creating a pool handle with method *CreatePoolHandle()*, the module declares that it wants to use buffers from the memory pool as specified by name. More than one pool handles can be used in one module. A pool handle can be accessed with method *dabc::Module::FindPool()* via name, or with method *dabc::Module::Pool()* via handle number (started from 0).

If a pool of the given name does not exist, it will be created automatically at the time of the first request. Buffer size and the number of buffers, which are specified in the *CreatePoolHandle()* call, play a role in this case only.

4.2.2 Ports

Class ***dabc::Port*** is the only legal way to transport buffers from/to the module. Class ***dabc::Module*** provides following methods for working with ports:

kind	Create	Count	Access	Search
input	*CreateInput(name, ...)*	*NumInputs()*	*Input(unsigned)*	*InputNumber()*
output	*CreateOutput(name, ...)*	*NumOutputs()*	*Output(unsigned)*	*OutputNumber()*
inp/out	*CreateIOPort(name, ...)*	*NumIOPorts()*	*IOPort(unsigned)*	*IOPortNumber()*

A port usually should be created in the module constructor. As first argument in the creation methods a unique port name should be specified. As second argument, the pool handle should be specified; this defines the memory pool where necessary memory can be fetched for the transports associated with the port. The Length of input or (and) output queue defines how many buffers can be kept in corresponding queue. One also can specify the size of user header, which is expected to be transported over the port - it is important for further transport configurations.

Any kind of port can be found by name with *FindPort()* method. But this is not the fastest way to work with ports, because string search is not very efficient. One better should use in code methods like *NumInputs()* and *Input(unsigned)* (for input ports), where the port id number (i. e. the sequence number of port creation) is used.

Class ***dabc::Port*** provides methods *Send()* and *Recv()* to send or receive buffers. While these are non-blocking methods, one should use *CanSend()* and *CanRecv()* methods before one can call transfer operations.

4.2.3 Parameters and configurations

Parameters are used in module for configuration, controlling and monitoring. More information about parameters handling see in chapter 5.

4.2.4 Commands processing

There is the possibility in *DABC* to execute user-defined commands in a module context. Virtual method *ExecuteCommand()* is called every time when a command is submitted to the module. The command is **always** executed in the module thread, disregarding from which thread the command was submitted. Therefore it is not necessary to protect command execution code against module function code by means of thread locks.

Most actions in *DABC* are performed with help of commands.

Here is an example how command execution can look like:

```
int UserModule::ExecuteCommand(dabc::Command* cmd)
{
   if (cmd->IsName("UserPrint")) {
      DOUT1(("Printout from UserModule"));
      return dabc::cmd_true;
```

```
    }
    return dabc::ModuleSync::ExecuteCommand(cmd);
}
```

This is invoked somewhere in the code of another component:

```
...
dabc::Module* m = dabc::mgr()->FindModule("MyModule");
dabc::Command* cmd = new dabc::Command("UserPrint");
m->Execute(cmd);
// again, but in short form
m->Execute("UserPrint");
...
```

After command execution has finished, method *Execute()* returns true or false, depending on the success. The ***dabc::Command*** object is deleted automatically after execution.

In the module constructor, one can register a command for the control system by means of a corresponding ***dabc::CommandDefinition*** object. In this case the command and its arguments are known remotely and can be invoked from a controls GUI:

```
UserModule::UserModule(const char* name) : dabc::ModuleSync(name)
{
    ...
    dabc::CommandDefinition* def = NewCmdDef("UserPrint");
    def->AddArgument("Level", dabc::argInt, false); // optional argument
    def->Register(true);
}
```

4.2.5 ModuleSync

Data processing functionality in a most intuitive way can be implemented by subclassing the ***dabc::ModuleSync*** base class, which defines the interface for a synchronous module that is allowed to block its dedicated execution thread.

This class provides a number of methods which will block until the expected action can be performed.

Method	Description
Recv()	Receive buffer from specified input port
Send()	Send buffers over output port
RecvFromAny()	Receive buffer from any of specified port
WaitInput()	Waits until required number of buffers is queued in input port
TakeBuffer()	Get buffer of specified size from memory pool
WaitConnect()	Waits until port is connected

In all these methods a timeout value as last argument can be specified. Method *SetTmoutExcept()* defines if a ***dabc::TimeoutException*** exception is thrown when the timeout is expired. By default, these blocking methods just return false in case of timeout.

Data processing should be implemented in *MainLoop()* method. It usually contains a **while()** loop where *ModuleWorking()* method is used to check if execution of module code shall be continued. This method will also execute the queued commands, if *synchronous command execution* was specified before by method *SetSyncCommands()*. By default, a command can be executed in any place of the code.

Let's consider a simple example of a module which has one input and two output ports, and delivers buffers from input to one or another output sequentially. Implementation of such class will look like:

```
#include "dabc/ModuleSync.h"

class RepeaterSync : public dabc::ModuleSync {
public:
   RepeaterSync(const char* name) : dabc::ModuleSync(name)
   {
      CreatePoolHandle("Pool", 2048, 1);
      CreateInput("Input", Pool(), 5);
      CreateOutput("Output0", Pool(), 5);
      CreateOutput("Output1", Pool(), 5);
   }

   virtual void MainLoop()
   {
      unsigned cnt(0);
      while (ModuleWorking()) {
         dabc::Buffer* buf = Recv(Input());
         if (cnt++ % 2 == 0) Send(Output(0), buf);
                        else Send(Output(1), buf);
   }
};
```

In constructor one sees creation of pool handle and input and output ports. Method *MainLoop()* has a simple `while()` loop, that receives a buffer from the input and then sends it alternatingly to the first or the second output.

4.2.6 ModuleAsync

In contrast to data processing in ***dabc::ModuleSync*** main loop, class ***dabc::ModuleAsync*** provides a number of callbacks routines which are executed only if dedicated *DABC* events occur. For instance, when any input port gets new buffer, virtual method *ProcessInputEvent()* will be called. User should reimplement this method to react on the event.

The main advantage of such approach is that the thread is not blocked and several modules ***dabc::ModuleAsync*** can run within same working thread. At the same time, using such programming technique may require additional bookkeeping, as it is not allowed to block the callback routine while waiting for some resource to be available.

Class ***dabc::ModuleSync*** provides number of methods for handling different events:

Method	Description
ProcessInputEvent()	new buffer in input queue, it can be read with *port->Recv()*
ProcessOutputEvent()	new space in output queue is available, one can use *port->Send()*
ProcessConnectEvent()	port is connected to transport
ProcessDisconnectEvent()	port was disconnected from transport
ProcessPoolEvent()	requested buffer can be read with *handle->TakeRequestedBuffer()*
ProcessTimerEvent()	timer has fired an event

By reimplementing one or several of these methods one can react on corresponding events.

Actually, all events are dispatched to the methods mentioned above by method *ProcessUserEvent()*. This method is called by the working thread whenever **any** event for this module shall be processed. However, this virtual method may also directly be re-implemented in the user subclass if one wants to treat all events centrally. As arguments one gets the pointer to the relevant component (port, timer, ...) and a number describing the event type (dabc::evntInput, dabc::evntOutput, ...)

Class ***dabc::ModuleAsync*** has no methods which can block. Nevertheless the user should avoid any kind of polling loops, waiting for some other resource (buffer, output queue and so on) - the callbacks should **return** as soon as possible. In such situation, processing must be continued in another callback that is invoked when the required resource is available. This might require an own bookkeeping of such situations (kind of state transition logic).

Let's consider as an example the same repeater module, but implemented as asynchronous module:

```
#include "dabc/ModuleAsync.h"
#include "dabc/Port.h"

class RepeaterAsync : public dabc::ModuleAsync {
   unsigned   fCnt;
public:
   RepeaterAsync(const char* name) : dabc::ModuleAsync(name)
   {
      CreatePoolHandle("Pool", 2048, 1);
      CreateInput("Input", Pool(), 5);
      CreateOutput("Output0", Pool(), 5);
      CreateOutput("Output1", Pool(), 5);
      fCnt = 0;
   }

   virtual void ProcessInputEvent(dabc::Port* port)
   {
      while (Input()->CanRecv() && Output(fCnt % 2)->CanSend()) {
         dabc::Buffer* buf = Input()->Recv();
         Output(fCnt++ % 2)->Send(buf);
      }
   }

   virtual void ProcessOutputEvent(dabc::Port* port)
   {
      while (Input()->CanRecv() && Output(fCnt % 2)->CanSend()) {
         dabc::Buffer* buf = Input()->Recv();
         Output(fCnt++ % 2)->Send(buf);
      }
   }
};
```

The constructor of this module has absolutely the same components as in previous example. One should add *fCnt* member to count direction for output of next buffer. Value of *fCnt* in some sense defines current state of the module. Instead of the `MainLoop()` one can see two virtual methods for input and output event processing. In each methods one sees same code, with while loop inside. In the loop one checks that input and current output are ready and retransmit buffer. When any port (input or output) has no more possibility to transmit data, method will be returned.

One needs a while() loop here because not every input event and not every output events leads to buffer transports. If input queue is empty (*CanRecv()* returns false), or output queue is full (*CanSend()* returns false), one cannot transfer a buffer from input to output; thus the callback must be returned. But when the event processing routine is called the next time, one should tranfer several buffers at once. Since methods *Send()* and *Recv()* cannot block, such while() loop will not block either. But in any case one should avoid such **wrong** code:

```
   virtual void ProcessInputEvent(dabc::Port* port)
```

```
{
   // this kind of waiting is WRONG!!!
   while(!Output(fCnt % 2)->CanSend()) usleep(10);

   dabc::Buffer* buf = Input()->Recv();
   Output(fCnt++ % 2)->Send(buf);
}
```

Here the while() loop can wait for an infinite time until the output port will accept a new buffer, and during this time the complete thread will be blocked.

As both processing methods are the same in the example, one can implement central *ProcessUserEvent()* method instead:

```
virtual void ProcessUserEvent(dabc::ModuleItem*, uint16_t)
{
   while (Input()->CanRecv() && Output(fCnt % 2)->CanSend()) {
      dabc::Buffer* buf = Input()->Recv();
      Output(fCnt++ % 2)->Send(buf);
   }
}
```

To introduce time-dependent actions in ***dabc::ModuleAsync***, one should use timers. Timer objects can be created with method *CreateTimer()*. It delivers a timer event with specified intervals, which can be processed in *ProcessTimerEvent()* method.

One can modify the previous example to display the number of transported buffers every 5 seconds.

```
RepeaterAsync(const char* name) : dabc::ModuleAsync(name)
{
   ...
   CreateTimer("Timer1", 5.);
}

virtual void ProcessUserEvent(dabc::ModuleItem* item, uint16_t evnt)
{
   ...
   if (evnt == dabc::evntTimeout) DOUT1(("Buffers count = %d", fCnt));
}
```

4.2.7 Special modules

For special set ups (e.g. Bnet), the framework provides ***dabc::Module*** subclasses with generic functionality (e.g. ***bnet::BuilderModule***, ***bnet::FilterModule***). In this case, the user specific parts like data formats are implemented by subclassing these special module classes.

1. Instead of implementing *MainLoop()* (or *ProcessUserEvent()*, resp.) other virtual methods (e.g. *DoBuildEvent()*, *TestBuffer()*) may be implemented that are implicitly called by the superclass *MainLoop()* (or by the appropriate event callbacks, resp.).
2. The special base classes may provide additional methods to be used for data processing.

4.3 Device and transport

All data transport functionality is implemented by subclassing ***dabc::Device*** and ***dabc::Transport*** base classes.

4.3.1 Transport

Actual transport of ***Buffers*** from/to a ***Port*** is done by a ***dabc::Transport*** implementation. During connection time each module port gets the pointer to a transport object which provides a number of methods for buffer transfer. As the ***Transport*** object typically runs in another thread than the module, the transmission of a buffer does not happen immediately when calling *dabc::Port::Send()* or *dabc::Port::Recv()* methods, but the buffer is at first kept in a queue which must be provided by the ***Transport*** implementation.

4.3.2 Device

Class ***dabc::Device*** usually (but not always) represents some physical device (like a network or a PCIe card) and has the role of a management unit for the ***Transports*** which belong to that device. The ***Device*** is always the owner of its ***Transport*** objects, i. .e it creates, keeps, and deletes them.

A ***Device*** is typically created in the user application by:

```
...
dabc::mgr()->CreateDevice("roc::Device", "ROC");
...
```

Later one can find this device with *dabc::Manager::FindDevice()* method.

Each ***dabc::Device*** implementation should define the virtual method *CreateTransport()* such, that an appropriate ***Transport*** instance is created and connected to the specified ***Port***. This factory method is invoked by the framework when the device is connected to a module port. This is usually specified in the user application by calls of
`dabc::mgr()->CreateTransport()`, or `dabc::mgr()->ConnectPorts()`, resp.

Similar to the ***Module*** functionality, the ***Device*** class may export configuration ***Parameters***. It may also define ***Commands*** which are handled by extending the virtual method *ExecuteCommand()* with a device specific implementation.

4.3.3 Local transport

dabc::LocalTransport implements the connection between two "local" ports, i. e. the ports are on the same node with a common memory address space. It organizes a queue which is shared between both connected ports, and performs the movement of ***dabc::Buffer*** pointer through this queue. If corresponding modules run in the same thread, ***LocalTransport*** works without any mutex locking.

To manage the ***LocalTransport***, the ***dabc::Manager*** always has instance of ***dabc::LocalDevice*** class. It can be accessed via *dabc::mgr()->FindLocalDevice()* call.

To connect two local ports, one should call:

```
...
dabc::mgr()->ConnectPorts("Module1/Output", "Module2/Input");
...
```

4.3.4 Network transport

This is a kind of ***Transport*** which is used to connect ***Ports*** on different nodes. Abstract base class ***dabc::NetworkTransport*** introduces such kind of functionality: This transport is locally connected to one port only, and all buffer transfer is done via network connections with the remote node.

For the moment *DABC* has two implementations of network transports: for socket and *InfiniBand verbs*. To use ***NetworkTransport*** on the nodes, one should follow a two step strategy. At the first step, on all nodes the necessary ***Devices*** and ***modules*** should be created:

```
...
dabc::mgr()->CreateDevice(dabc::typeSocketDevice, "UserDev");
dabc::mgr()->CreateModule("UserModule", "MyModule");
...
```

Then during the second step, on the "master" node (where dabc::mgr()->IsMainManager() is true, see section 2.3.2, page 28) one should call:

```
...
dabc::mgr()->ConnectPorts("Node0$MyModule/Input",
                          "Node1$MyModule/Output", "UserDev");
...
```

Such call starts an elaborated sequence: At first a server socket will be opened by ***Device*** "UserDev" on node "Node0"; then ***Device*** "UserDev" on "Node1" will try to connect to that server socket; finally, on both nodes appropriate ***NetworkTransports*** will be created, using these negotiated sockets, and connected to the ports "$MyModule/Input", and "$MyModule/Output", resp.

Exactly for this kind of actions the *DABC* state machine has two transition commands "DoConfigure" and "DoEnable". Accordingly, class ***dabc::Application*** has two methods *CreateAppModules()* and *ConnectAppModules()* (see 4.4).

4.3.5 Data transport

In general, to implement a user-specific transport one should subclass from ***dabc::Transport***. But this requires a deeper knowledge about the *DABC* mechanisms: how threads are working, how one should organize input/output queues, how the transport should request data from a memory pool, and which initialization commands are used by the framework. To simplify transport development and provide all basic services class ***dabc::DataTransport*** was introduced.

For a **data input** the user should implement the following virtual methods :

Read_Size() : Should return the required buffer size to read next portion of data from the data source. For instance, many file formats have a header before each portion of data, describing the payload size that follows. This method then should be used to read such header. Method can also return following values:
dabc::di_EndOfStream - end of stream, normal close of the input
dabc::di_Repeat - nothing to read now, call again as soon as possible
dabc::di_RepeatTimeout - nothing to read now, try again after timeout
dabc::di_Error - error, close transport

Read_Timeout() : Defines timeout (in seconds) for operation like *Read_Size()*

Read_Start() : Starts reading of buffer. Should return:
dabc::di_Ok - normal case, call of *Read_Complete()* will follow

dabc::di_Error - error, skip buffer, starts again from *Read_Size()*
dabc::di_CallBack - asynchronous readout, user should call *Read_CallBack()*
If di_CallBack returned, processing of this transport is suspended until user calls *Read_CallBack()* method, providing the result of reading: di_Ok or di_Error. This mode is only possible if the device driver has its own thread (or DMA engine, resp.) that can perform the readout and then can call *DABC* methods. The big advantage of such mode: the data transport thread is not blocked by waiting for a result from the device, therefore several ***DataTransports*** can share the same thread.

Read_Complete() : Finish reading of the buffer. Can return:
dabc::di_Ok - normal, buffer will be delivered to port
dabc::di_Error - error, close transport
dabc::di_EndOfStream - end of stream, normal close of the transport
dabc::di_SkipBuffer - normal, but buffer will not be delivered to the port
dabc::di_Repeat - not ready, call again as soon as possible
dabc::di_RepeatTimeout - not ready, call again after timeout
In the simple case, actual reading of data is directly performed in this method. Otherwise one may wait here until another thread or a DMA transfer, initiated before by *Read_Start()*, has filled the buffer. In this case one should be carefull and not block thread forever - it is better to return with dabc::di_Repeat, so the thread can continue its event loop and handle other workers.

For **data output**, the user should just implement virtual method *WriteBuffer()* .

In some cases user may redefine *ProcessPoolChanged()* which is called when memory pool changes its layout - new buffers were allocated or released. It may be required for DMA operations, where each buffer from a memory pool should be initialised once before it can be used for data transport.

It is not always necessary to create a user-specific ***Device*** for a user written ***DataTransport***, since the standard ***LocalDevice*** can be used if it is only required as owner for the transport objects. In this case, the factory method *CreateTransport()* should be provided already in the user ***Factory*** (see section 4.5).

However, some user implementations of ***DataTransport*** may require services of a corresponding ***Device*** though. In this case, the user should implement a ***Device*** that provides the factory method *CreateTransport()* (see section 4.3.2). This can instantiate the ***DataTransport*** with a back reference to the responsible ***Device***.

4.3.6 Input/output objects

Besides the ***Transports***, *DABC* provides an interface for implementing a simple input/output by means of base classes ***dabc::DataInput*** and ***dabc::DataOutput***. The interface is similar to that of ***dabc::DataTransport***, but these classes are not depending on any other components (threads, devices, etc.), and therefore can be applied without the *DABC* data flow engine. The only feature which is not supported by ***dabc::DataInput*** is the CallBack mode.

In addition, methods *dabc::DataInput::Read_Init()* and *dabc::DataOutput::Write_Init()* can be implemented to get configuration parameters from the port object to which the i/o object is assigned to.

A typical use case of input/output objects is the file I/O. For instance, `.lmd` file handling is implemented using these classes.

To instantiate such classes, user should inplement factories methods *CreateDataInput()* and *CreateDataOutput()* (see 4.5).

4.4 The *DABC* Application

The specific application controlling code is defined in the ***dabc::Application***.

On startup time, the ***dabc::Application*** is instantiated by means of a factory method *CreateApplication()*. As argument the factories get the application class name, provided from the configuration file. Thus, to use his/her application implementation, the user must provide a ***dabc::Factory*** that defines such method.

The manager has exactly one application object - the name of this object is always "App". The application singleton can be accessed from everywhere via *dabc::mgr()->GetApp()* call.

The application may register parameters that define the application's configuration. These parameters can be set at runtime from the configuration file or by controls system.

The application class implements the user-specific actions during the state machine transitions. The application has virtual method *DoStateTransition()* which is called from the state machine during state change. As argument, name of state transition command is delivered. There are the following state machine commands:

dabc::Manager::stcmdDoConfigure - creates all necessary application components: devices, modules, memory pools
dabc::Manager::stcmdDoEnable - connects local and (or) remote nodes together (if necessary)
dabc::Manager::stcmdDoStart - starts execution of user modules
dabc::Manager::stcmdDoStop - stop execution of user modules
dabc::Manager::stcmdDoHalt - destroy all components, created during configure
dabc::Manager::stcmdDoError - react on error, which happened during other commands

Class ***dabc::Application*** already has default implementation for *DoStateTransition()* method, where some virtual methods are called:

CreateAppModules() - creates all necessary application components
ConnectAppModules() - activity to connect with remote nodes or
IsAppModulesConnected() - check if connection is already performed
BeforeAppModulesStarted() - optional activity before modules are started
AfterAppModulesStopped() - optional activity after modules are stopped
BeforeAppModulesDestroyed() - optional call before modules are destroyed

Actually, for a single-node application it is enough to implement *CreateAppModules()*, since all other methods have meaningfull implementation for that case.

For special DAQ topologies (e.g. Bnet), the framework offers implementations of the ***dabc::Application*** containing the generic functionality (e. g. ***bnet::WorkerApplication***, ***bnet::ClusterApplication***). In this case, the user specific parts are implemented by subclassing and implementing additional virtual methods (e. g. *CreateReadout()*).

4.5 Factories

The creation of the application specific objects is done by ***dabc::Factory*** subclasses.

The user must define a ***dabc::Factory*** subclass to add own classes to the system. The user factory should already be instantiated as global stack object in its class implementation code - this will create the factory immediately after the user library has been loaded. On creation time, a factory is registered automatically to the ***dabc::Manager*** instance.

The user factory may implement such methods:

CreateModule() : Instantiate a ***dabc::Module*** of specified class.
CreateDevice() : Instantiate a ***dabc::Device*** of specified class.
CreateThread() : Instantiate a ***dabc::WorkingThread*** of specified class.
CreateApplication() : Instantiate a ***dabc::Application*** of specified class.
CreateTransport() : Instantiate a ***dabc::Transport*** of specified class. This method is used when transport does not requires specific device functionality (like ***dabc::DataTransport***). Typically transport objects created by the ***dabc::Device*** methods.
CreateDataInput() : Instantiate a ***dabc::DataInput*** of specified type. Initialisation of object will be done by *Read_Init()* call.
CreateDataOutput() : Instantiate a ***dabc::DataOutput*** of specified type. Initialisation of object will be done by *Write_Init()* call.

Since all factories are registered and kept in the global *DABC* manager, all methods mentioned here have equivalent methods in class ***dabc::Manager***. The manager simply iterates over all factories and executes the appropriate factory method until an object of the requested class is created. For instance, to create a module, one should do:

```
...
dabc::mgr()->CreateModule("mbs::GeneratorModule", "Generator");
...
```

Invocation of these methods in manager is implemented via corresponding commands (for instance, CmdCreateModule for module creation). These command classes should be used directly, if one wants to deliver extra configuration parameters to the object's constructor (most factories methods gets this command as optional argument). For instance:

```
...
dabc::Command* cmd = new dabc::CmdCreateModule("mbs::GeneratorModule",
                                               "Generator");
cmd->SetInt("NumSubevents", 5);
cmd->SetInt("SubeventSize", 64);
dabc::mgr()->Execute(cmd);
...
```

The *DABC* framework provides several factories for predefined implementations (e. g. ***bnet::SenderModule***, ***verbs::Device***)

Chapter 5

DABC Programmer Manual: Setup

[programmer/prog-setup.tex]

5.1 Parameter class

Configuration and status information of objects can be represented by the ***Parameter*** class. Any object derived from ***WorkingProcessor*** class (e. g. ***Application***, ***Device***, ***Module***, and ***Port***) can have a list of parameters assigned to it.

There are a number of ***WorkingProcessor*** methods to create parameter objects of different kinds and access their values. These are shown in the following table:

Type	Class	Create	Getter	Setter
string	StrParameter	CreateParStr()	GetParStr()	SetParStr ()
double	DoubleParameter	CreateParDouble()	GetParDouble()	SetParDouble()
int	IntParameter	CreateParInt()	GetParInt()	SetParInt()
bool	StrParameter	CreateParBool()	GetParBool()	SetParBool()

The *GetPar...()* methods will internally create a new ***Parameter*** of the specified name if it does not exist before. For any type of parameter the *GetParStr()* and *SetParStr()* methods can be used which will deliver the parameter value as text string expression.

As one can see, to represent a boolean value a string parameter is used. If text of string is "true" (in lower case), the boolean value is recognized as `true`, otherwise as `false`.

It is recommended to use these ***WorkingProcessor*** methods to create parameters and access their values; but one can also use *FindPar()* method to find any parameter object and use its methods directly.

5.2 Use parameter for control

One advantage of the *DABC* parameter objects is that parameter values can be observed and changed by a control system.

When a parameter value is changed in the program by a *SetPar...* method, the control system is informed and represents such change in an appropriate GUI element. On the other hand, if the user modifies a parameter value in the GUI, the value of the parameter object will be changed and the corresponding parent object (***Module***, ***Device***) gets a callback via virtual method *ParameterChanged()*. By implementing a suitable reaction in this call, one could reconfigure or adjust the running program on the fly.

A parameter object may be "fixed" via *Parameter::SetFixed()* method. This disables to change the parameter value, both from the program and the control/configuration system side. Only when the "fixed" flag is reset to false, the parameter can be modified again.

Not all parameters objects should be visible to the control system. Each parameter has a **visibility level** which is assigned to the parameter instance when it is created. A parameter is known to the control system only if its visibility level is smaller than the system visibility level configured with *Manager::ParsVisibility()*. The default parameter visiblity level can be set via *WorkingProcessor::SetParDflts()* function before the parameters objects are created.

5.3 Example of parameters usage

Let's consider an example of a module which uses parameters:

```
class UserModule : public dabc::ModuleAsync {
   public:
      UserModule(const char* name, dabc::Command* cmd = 0) :
         dabc::ModuleAsync(name, cmd)
      {
         CreateParBool("Output", true);
         CreateParInt("Counter", 0);
         CreateTimer("Timer", 1.0, false);
      }

      virtual void ProcessTimerEvent(dabc::Timer*)
      {
         SetParInt("Counter", GetParInt("Counter")+1);
         if (GetParBool("Output"))
            DOUT1(("Counter = %d", GetParInt("Counter")));
      }
};
```

In the module constructor two parameters are created - boolean and integer, and a timer with 1 s period. When the module is started, the value of integer parameter "Counter" will be changed every second. If boolean parameter "Output" is set to true, the counter value will be displayed on debug output.

Using a control system, the value of the boolean parameter can be changed. To detect and react on such change, one should implement following method:

```
virtual void ParameterChanged(dabc::Parameter* par)
{
   if (par->IsName("Output"))
   DOUT1(("Output flag changed to %s", DBOOL(GetParBool("Output")));
}
```

For performance reasons one should avoid to use parameter getter/setter methods (like *GetParBool()* or *SetParInt()*) inside a loop being executed many times. The main purpose of a parameter object is to provide a connection to the control and configuration system. In other situations simple class members should be used.

5.4 Configuration parameters

Another use case of parameters consists in the object configuration. When one creates an object, like a module or a device, it is often necessary to deliver one or several configuration values to the constructor, e. g. the required number of input ports, or a server socket port number.

For such situation configuration parameter are defined. These parameters should be created and set in the object constructor with following methods only:

GetCfgStr string
GetCfgDouble double
GetCfgInt integer
GetCfgBool boolean

All these methods have following arguments: the parameter name, a default value [optional], and a pointer to a ***Command*** object [optional]. Let's add one configuration parameter to our module constructor:

```
UserModule(const char* name, dabc::Command* cmd = 0) :
        dabc::ModuleAsync(name, cmd)
{
        CreateParBool("Output", true);
        CreateParInt("Counter", 0);
        double period = GetCfgDouble("Period", 1.0, cmd);
        CreateTimer("Timer", period, false);
}
```

Here the period of the timer is set via configuration parameter "Period". How will its value be defined? First of all, it will be checked if a parameter of that name exists in command *cmd*. If not, the appropriate entry will be searched in the configuration file. If the configuration file also does not contain such parameter, the specified default value 1.0 will be used.

5.5 Configuration file example

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

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. To run that example, just "run.sh test.xml" should be executed in a shell. Other applications (*DABC* or *Go4*) can connect to that server and read generated mbs events.

5.6 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 5.9 and 5.10

5.7 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.

5.8 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 4.4)
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.

5.9 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.

5.10 Default values

There are situations when one needs to set the same value to several similar parameters, for instance the same output queue length for all output ports in the module. One possible way is to use variables, as described above. The disadvantage of such approach is that one must expand the XML file to set each queue length explicitely from the 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.

5.11 Usage of commands for configuration

Let's consider the possibility to configure a module by means of the ***Command*** class. Here the use case is that an object (like a module) should be created with fixed parameters, ignoring the values specified in the configuration file.

In our example one can modify *InitMbsGenerator()* function in the following way:

```
extern "C" void InitMbsGenerator()
{
  dabc::Command* cmd = new dabc::CmdCreateModule("mbs::GeneratorModule",
                                                  "Generator");
  cmd->SetInt("SubeventSize", 128);
  if (!dabc::mgr()->Execute(cmd)) {
```

```
      EOUT(("Cannot create generator module"));
      exit(1);
   }

     ...
}
```

Here one adds an additional parameter of name "SubeventSize" to the ***CmdCreateModule*** object, which will set the MBS subevent size to 128. The generator module constructor will get the parameter value via method *GetCfgInt()*, as described in section 5.4. Since the parameters of the passed *cmd* object will override all other settings here, the value of the corresponding <SubeventSize> entry in the configuration file has no effect.

Chapter 6

DABC Programmer Manual: Example *MBS*

[programmer/prog-exa-mbs.tex]

6.1 Overview

MBS (Multi Branch System) is the standard DAQ system of GSI. Support of MBS in *DABC* includes several components:

- type definitions for different MBS structures
- iterator classes for reading/creating MBS event/subevent data
- support of new `.lmd` file format
- ***mbs::ClientTransport*** for connecting to MBS servers
- ***mbs::ServerTransport*** to "emulate" running MBS servers
- ***mbs::CombinerModule*** for performing mbs events building
- ***mbs::GeneratorModule*** for generating random mbs events

This plugin is part of the standard *DABC* distribution. All sources can be found in the `$DABCSYS/plugin/mbs` directory. All these sources are compiled into library `libDabcMbs.so` which is located in `$DABCSYS/lib`.

6.2 Event iterators

The MBS system has native event and subevent formats. To access such event data, several structures are introduced in `mbs/LmdTypeDefs.h` and `mbs/MbsTypeDefs.h`. In the first file, structure ***mbs::Header*** is defined which is just a container for arbitrary raw data. Such container is used to store data to, or to read data from `.lmd` files, resp. In file `mbs/MbsTypeDefs.h` the following structures are defined:

- ***mbs::EventHeader*** - MBS event header of 10-1 type
- ***mbs::SubeventHeader*** - MBS subevent header of 10-1 type

DABC operates with buffers of type mbt_MbsEvents that contain several subsequent MBS events; there is no buffer header in front here. To iterate over all events in such buffer and to access them, class ***mbs::ReadIterator*** is provided (as defined in `mbs/Iterator.h`). This is done in the following way:

```
#include "mbs/Iterator.h"
```

```
void Print(dabc::Buffer* buf)
{
   mbs::ReadIterator iter(buf);
   while (iter.NextEvent()) {
      DOUT1(("Event %u size %u",
              iter.evnt()->EventNumber(),
              iter.evnt()->FullSize()));
      while (iter.NextSubEvent()) {
         DOUT1(("Subevent crate %u procid %u size %u",
                iter.subevnt()->iSubcrate,
                iter.subevnt()->iProcId,
                iter.subevnt()->FullSize()));
      }
   }
}
```

Another class ***mbs::WriteIterator*** fills MBS events into ***dabc::Buffer***. This is illustrated by the following code:

```
#include "mbs/Iterator.h"

void Fill(dabc::Buffer* buf)
{
   mbs::WriteIterator iter(buf);
   unsigned evntid = 0;
   while (iter.NewEvent(evntid++)) {
      for (unsigned subcnt = 0; subcnt < 3; subcnt++) {
         if (!iter.NewSubevent(28, 0, subcnt)) return;
         // fill raw data iter.rawdata() here
         memset(iter.rawdata(), 0, 28);
         iter.FinishSubEvent(28);
      }
      if (!iter.FinishEvent()) return;
   }
}
```

Method *NewEvent()* will create and event header of the event id *evntid* at the current iterator position in the buffer. In a similar way, method *NewSubevent()* will put a subevent header of the specified arguments there. Access to the data pointer after the last event or subevent header is done by *iter.rawdata()* method. Finally, *FinishSubevent()* and *FinishEvent()* will complete the subevent, or event definition, resp.

6.3 File I/O

MBS uses the `.lmd` ("List Mode Data") file format. Class ***mbs::LmdFile*** provides a C++ interface for reading and writing such files.

To use ***mbs::LmdFile*** as input, or output transport of the module, classes ***mbs::LmdInput*** and ***mbs::LmdOutput*** were developed, resp.

In general, to provide user-specific input/output capability over a port, one should implement the complete ***dabc::Transport*** interface, which includes event handling, queue organization, and a complex initialization sequence. All this is necessary for cases like socket or *InfiniBand* transports, but too complicated for simple cases as file I/O. Therefore, a special kind of transport ***dabc::DataIOTransport*** was developed, which handles most of such complex tasks and reduces the implementation effort to the relatively simple ***dabc::DataInput*** and ***dabc::DataOutput*** interfaces.

Class ***mbs::LmdOutput*** inherits ***dabc::DataOutput*** and implements to save MBS events, if contained in ***dabc::Buffer*** objects, into an `.lmd` file. In addition to ***mbs::LmdFile*** functionality, it allows to create multiple files when a file size limit is exceeded. The class has following parameters:

- MbsFileName - name of lmd file (including .lmd extension)
- MbsFileSizeLimit - size limit (in Mb) of single file, 0 - no limit

Class ***mbs::LmdInput*** inherits ***dabc::DataInput*** and allows to read MBS events from `.lmd` file(s) and to provide them over input ports into a module. It has following parameters:

- MbsFileName - name of lmd file (multicast symbols * and ? supported)
- BufferSize - buffer size to read data

CreateDataInput() and *CreateDataOutput()* methods were implemented in ***mbs::Factory*** class such, that a user can instantiate these classes via the *DABC* plugin mechanism.

Here is an example how the output file for a generator module can be configured:

```
...
dabc::mgr()->CreateModule("mbs::GeneratorModule", "Generator");
dabc::Command* cmd =
   new dabc::CmdCreateTransport("Generator/Output", mbs::typeLmdOutput);
cmd->SetStr(mbs::xmlFileName, "output.lmd");
cmd->SetInt(mbs::xmlSizeLimit, 100);
dabc::mgr()->Execute(cmd);
...
```

At first the module is created; then the type of output transport and its parameters are set via command.

Another example shows how several input files can be configured for a combiner module:

```
...
dabc::Command* cmd =
   new dabc::CmdCreateModule("mbs::CombinerModule", "Combiner");
cmd->SetInt(dabc::xmlNumInputs, 3);
dabc::mgr()->Execute(cmd);

for (unisgned n=0;n++;n<3) {
   cmd = new dabc::CmdCreateTransport(
      FORMAT(("Combiner/Input%u",n)), mbs::typeLmdInput);
   cmd->SetStr(mbs::xmlFileName, FORMAT(("input%u_*.lmd",n)));
   dabc::mgr()->Execute(cmd);
}
...
```

In this example one creates a module with 3 inputs and connects each input port with an `*.lmd` file transport.

6.4 Socket classes

All communication with the MBS nodes is performed via tcp sockets. The *DABC* base package `libDabcBase.so` implements a number of classes for general socket handling. The main idea of these classes is to handle socket operations (creation, connection, sending, receiving, and error handling) by means of event processing callbacks.

Class ***dabc::SocketThread*** organises the event loop that handles such event signals produced by a socket. Each system socket is assigned to an instance of ***dabc::SocketProcessor***. The actual socket event processing is then done in virtual methods of class ***dabc::SocketProcessor*** which has several subclasses for different kinds of sockets:

- - ***dabc::SocketServerProcessor*** - server socket for connection
- - ***dabc::SocketClientProcessor*** - client socket for connection
- - ***dabc::SocketIOProcessor*** - send/recv handling

One can use a ***dabc::SocketThread*** together with other kind of processors like module classes, but not vice-versa: one cannot use socket processors inside other thread types. Therefore, it is possible to run module with all socket transports in one single thread, if the socket thread for such module is created in advance (see MBS generator example in section 6.7).

6.5 Server transport

Class ***mbs::ServerTransport*** provides the functionalities of an *MBS transport server* or and *MBS stream server* in DABC . This is also a good example of the ***dabc::SocketProcessor*** classes.

mbs::ServerTransport is based on the generic class ***dabc::Transport*** and uses internally two kinds of sockets: one socket for the connection handling, and another "I/O" socket for sending data.

The server transport has following parameters:

Name	Type	Dflt	Description
MbsServerKind	str	Transport	kind of mbs server: "Transport" or "Stream"
MbsServerPort	int	6000	server port number for socket connection

These parameters can be set in the XML configuration file like this:

```
...
  <Module name="Generator">
    <Port name="Output">
      <MbsServerKind value="Transport"/>
      <MbsServerPort value="16020"/>
    </Port>
  </Module>
...
```

To create such transport and connect it to the module's output port, the following code should be executed:

```
  ...
  dabc::mgr()->CreateTransport("Generator/Output",
                    mbs::typeServerTransport, "GeneratorThrd");
  ...
```

Another possibility to specify these parameters consists in the command dabc::CmdCreateTransport which may wrap such values:

```
  ...
```

```
dabc::Command* cmd = new dabc::CmdCreateTransport("Generator/Output",
                         mbs::typeServerTransport, "MbsTransThrd");
cmd->SetStr(mbs::xmlServerKind, mbs::ServerKindToStr(mbs::StreamServer));
cmd->SetInt(mbs::xmlServerPort, mbs::DefaultServerPort(mbs::StreamServer) + 5);
dabc::mgr()->Execute(cmd);
...
```

6.6 Client transport

Class ***mbs::ClientTransport*** allows to connect *DABC* with MBS. At the moment the MBS *transport* and *stream* servers are supported.

Client transport has following parameters:

Name	Type	Dflt	Description
MbsServerKind	str	Transport	kind of mbs server: "Transport" or "Stream"
MbsServerName	str	localhost	host name where mbs server runs
MbsServerPort	int	6000	server port number for socket connection

To create client connection, the following piece of code should be used:

```
...
dabc::Command* cmd = new dabc::CmdCreateTransport("Combiner/Input0",
                         mbs::typeClientTransport, "MbsTransThrd");
cmd->SetStr(mbs::xmlServerKind, mbs::ServerKindToStr(mbs::StreamServer));
cmd->SetStr(mbs::xmlServerName, "lxi010.gsi.de");
cmd->SetInt(mbs::xmlServerPort, mbs::DefaultServerPort(mbs::StreamServer) + 5);
dabc::mgr()->Execute(cmd);
...
```

6.7 Event generator

Class ***mbs::GeneratorModule*** is an example of a simple module which just fills buffers with random MBS events, and provides them to the output port. Schematically the implementation of this module looks like this:

```
#include "dabc/ModuleAsync.h"

class GeneratorModule : public dabc::ModuleAsync {
protected:
   dabc::PoolHandle*        fPool;
   dabc::BufferSize_t       fBufferSize;
public:
   GeneratorModule(const char* name, dabc::Command* cmd = 0) :
      dabc::ModuleAsync(name, cmd)
   {
      ...
      fBufferSize = GetCfgInt(dabc::xmlBufferSize, 16384, cmd);
      fPool = CreatePoolHandle("Pool", fBufferSize, 10);
      CreateOutput("Output", fPool, 5);
   }
```

```
   virtual void ProcessOutputEvent(dabc::Port* port)
   {
      dabc::Buffer* buf = fPool->TakeBuffer(fBufferSize);
      FillRandomBuffer(buf);
      port->Send(buf);
   }
};
```

In the module constructor a pool handle is created, declaring a required memory pool with 10 buffers of defined size *fBufferSize*. The buffer size is taken from a configuration parameter with name *dabc::xmlBufferSize* (this string constant is predefined as "BufferSize"). When the output port is created, this pool handle and a default queue size is specified.

The only virtual method implemented for generator module is *ProcessOutputEvent()*. This function is called every time when a free buffer slot appears in the port output queue. Thus, when the module starts processing, this call will be immediately executed *N* times (size of output queue, here 5), because there are *N* empty entries in the queue. The only action here is to take a new buffer from the memory pool, fill it with random events and send it to output port.

The actual ***mbs::GeneratorModule*** is part of the `libDabcMbs.so` library and has following parameters:

Name	Type	Dflt	Description
NumSubevents	int	2	number of subevents in generated event
FirstProcId	int	0	value of procid field of first subevent
SubeventSize	int	32	size of rawdata in subevent
Go4Random	bool	true	is raw data filled with random value
BufferSize	int	16384	server port number for socket connection

There following function *InitMbsGenerator()* instantiates the generator module. It also demonstrates how a thread of type ***dabc::SocketThread*** can be created and used by both a module and a transport object.

```
extern "C" void InitMbsGenerator()
{
  dabc::mgr()->CreateThread("GenerThrd", dabc::typeSocketThread);
  dabc::mgr()->CreateModule("mbs::GeneratorModule", "Generator", "GenerThrd");
  dabc::mgr()->CreateTransport("Generator/Output", mbs::typeServerTransport, "GenerThrd");
}
```

To run the generator module with all default parameters, this simple XML file is sufficient:

```
<?xml version="1.0"?>
<dabc version="1">
  <Context host="lxi009" name="Server">
    <Run>
      <lib value="libDabcMbs.so"/>
      <func value="InitMbsGenerator"/>
    </Run>
  </Context>
</dabc>
```

Besides one may specify all module and transport parameters explicitely here:

```
<?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>
```

Example file `$DABCSYS/application/mbs/GeneratorTest.xml` demonstrates the usage of a generator module.

6.8 *MBS* event building

Class ***mbs::CombinerModule*** allows to combine events from several running MBS systems. It has following parameters:

Name	Type	Dflt	Description
BufferSize	int	16384	buffer size of output data
NumInputs	int	2	number of mbs data sources
DoFile	bool	false	create LMD file store for combined events
DoServer	bool	false	create MBS server to provide data further

The module implements two optional output ports: for file storage (port name "FileOutput"), and for providing data further over an MBS server (port name "ServerOutput").

Function *StartMbsCombiner()* initializes the combiner module and starts data taking. The following example configuration file `$DABCSYS/applications/mbs/Combiner.xml` shows how to configure a combiner module reading from three MBS transport servers:

```
<?xml version="1.0"?>
<dabc version="1">
  <Context host="localhost" name="Worker">
    <Run>
      <lib value="libDabcMbs.so"/>
      <func value="StartMbsCombiner"/>
      <logfile value="combiner.log"/>
    </Run>
    <Module name="Combiner">
       <NumInputs value="3"/>
       <DoFile value="false"/>
       <DoServer value="true"/>
```

```
        <BufferSize value="16384"/>
        <Port name="Input0">
           <InputQueueSize value="5"/>
           <MbsServerKind value="Transport"/>
           <MbsServerName value="lxi009"/>
           <MbsServerPort value="6000"/>
        </Port>
        <Port name="Input1">
           <InputQueueSize value="5"/>
           <MbsServerKind value="Transport"/>
           <MbsServerName value="lxi010"/>
           <MbsServerPort value="6000"/>
        </Port>
        <Port name="Input2">
           <InputQueueSize value="5"/>
           <MbsServerKind value="Transport"/>
           <MbsServerName value="lxi011"/>
           <MbsServerPort value="6000"/>
        </Port>
        <Port name="FileOutput">
           <OutputQueueSize value="5"/>
           <MbsFileName value="combiner.lmd"/>
           <MbsFileSizeLimit value="128"/>
         </Port>
        <Port name="ServerOutput">
           <OutputQueueSize value="5"/>
           <MbsServerKind value="Stream"/>
         </Port>
      </Module>
   </Context>
</dabc>
```

6.9 *MBS* upgrade for *DABC*

This section is rather for the *MBS* programmer than for application programmers. To have minimal changes, we use standard collector and transport. Two changes:

6.9.1 Increased buffer size support

This is done in a completely compatible way. For the following see data structures 6.9.4.3, page 74 and 6.9.4.2, page 74. The only problem is the 16 bit `i_used` field in the old buffer header `s_bufhe` structure (new `iUsed`) keeping the number of 16 bit data words (behind buffer header). The other 16 bits are used for event spanning. With a new rule we store this number in 32 bit field `l_free[2]` , now `iUsedWords`. Only if old `l_dlen`, now `iMaxWords` is less equal `MAX__DLEN` defined as `(32K-sizeof(bufhe))/2` this number is also stored in `i_used`(`iUsed`) as before. Modifications have to be made in all *MBS* modules accessing `i_used`. Modules outside *MBS* can be modified on demand to support large buffers. Current buffers still can be handled without change.
When *MBS* writes large buffer files only the used part of the file header is written. Number of 16 bit words behind buffer header structure is stored in `filhe_used`. Event API `f_evt` is updated to handle large buffers on input. Note: By setup the number of buffers per stream can be set to one. This suppresses event spanning. Large buffers can be used by standard *MBS*.

6.9.2 Variable sized buffers

As a second step variable sized buffers are implemented. They get a new type 100. The allocated buffers are still fixed length as before. However, the *MBS* transport would write only the used part of the buffers to clients. Processing these buffers a module must first read the header, then get the used size from `iUsedWords` (old `l_free[2]`) and read the rest. Modules outside *MBS* must be modified to process such buffers. In *MBS*, after stream buffers are created, buffer types are set to 100 by a new command enable dabc in transport. This command also sets the transport synchronous mode. In this mode transport processes streams only if a client is connected.

6.9.3 New LMD file format

With *DABC* as event builder for *MBS* there is no need to write files in *MBS*. This gives more freedom to design a new file format. This format will be written by *DABC* and read by `fLmd` functions (get event). The format is quite simpler than the old one, because it has no buffer structure causing so much complications by event spanning. The data elements itself, mainly the events, remain unchanged.

A file has a file header as before, but with a fixed size part and a variable part (size `iUsedWords`).

Behind the header follow data elements with `sMbsHeader` headers (length, type, subtype) allowing to identify and process or skip them. Elements must be sized in 4 byte units. Besides event data, time stamps may be inserted from the original *MBS* formatted buffers to preserve this time information. Writing/reading such a file is very straight forward. The file header contains the number of data elements (`iElements`) and the maximum size of elements (`iMaxWords`). This information is collected throughout the file writing and written on close into the file header. The file header is an `sMbsFileHeader` structure.

The file size is no longer restricted to 2GB. Optionally an element index is written at the end of the file. This allows for random access of elements in the file through this index table. The table itself has 32-bit values for the element offsets (in 32-bit). It can therefore address offsets up to 16GB in the file. If larger files are needed, the table can be created with 64-bit values giving unlimited addressing.

Note: This file format needs the `rewind` file function because the file header must be rewritten to store `iMaxWords`, `iElements`, and optionally the offset of the index table. This function is currently not implemented in the `RFIO` package, but will be done.

6.9.4 *MBS* data structures

All structures are defined independent on endianess. When bytes must be swapped, always 4 bytes are swapped. Fields 8 bytes long must be handled separately. Smaller items must be accessed by mask&shift. This makes code independent of endian.

6.9.4.1 Connect to *MBS* transport

Structure used to talk between client and transport server. Client connects to server (*MBS*) and reads this structure first. Structure maps the `sMbsTransportInfo` info buffer.

```
typedef struct{
  uint32_t iEndian;       // byte order. Set to 1 by sender
  uint32_t iMaxBytes;     // maximum buffer size
  uint32_t iBuffers;      // buffers per stream (should be 1 for DABC mode)
  uint32_t iStreams;      // number of streams (=0 for DABC mode)
```

```
} sMbsTransportInfo;
```

6.9.4.2 Buffer header

Buffer header, maps `s_bufhe`, some fields used in different way. The main difference is the usage of `iUsedWords` for the data length.

```
typedef struct{
  uint32_t iMaxWords;    // compatible with s_bufhe (total buffer size - header)
  uint32_t iType;        // compatible with s_bufhe, low=type (=100), high=subtype
  uint32_t iUsed;        // not used for iMaxWords>MAX__DLEN (16360), low 16bits only
  uint32_t iBuffer;      // compatible with s_bufhe
  uint32_t iElements;    // compatible with s_bufhe
  uint32_t iTemp;        // Used volatile
  uint32_t iTimeSpecSec; // compatible with s_bufhe (2*32bit) (struct timespec)
  uint32_t iTimeSpecNanoSec; // compatible with s_bufhe (2*32bit) (struct timespec)
  uint32_t iEndian;      // compatible with s_bufhe free[0]
  uint32_t iWrittenEndian; // LMD__ENDIAN_BIG, LMD__ENDIAN_LITTLE, LMD__ENDIAN_UNKNOWN
  uint32_t iUsedWords;   // total words without header, free[2]
  uint32_t iFree3;       // free[3]
} sMbsBufferHeader;
```

6.9.4.3 File header

File header, maps `s_bufhe`, some fields used in different way.

```
typedef struct{
  uint32_t iMaxWords;    // Size of largest element in file
  uint32_t iType;        // compatible with s_bufhe, low=type (=100), high=subtype
  lmdoff_t iTableOffset; // optional offset to element index table in file
  uint32_t iElements;    // Number of elements in file
  uint32_t iOffsetSize;  // Offset size, 4 or 8 [bytes]
  uint32_t iTimeSpecSec; // compatible with s_bufhe (2*32bit) (struct timespec)
  uint32_t iTimeSpecNanoSec; // compatible with s_bufhe (2*32bit) (struct timespec)
  uint32_t iEndian;      // compatible with s_bufhe free[0]
  uint32_t iWrittenEndian; // LMD__ENDIAN_BIG, LMD__ENDIAN_LITTLE, LMD__ENDIAN_UNKNOWN
  uint32_t iUsedWords;   // total words following header, free[2]
  uint32_t iFree3;       // free[3]
} sMbsFileHeader;
```

6.9.4.4 Data element structures

- Time stamp
  ```
  typedef struct{
    uint32_t iMaxWords;
    uint32_t iType;
    uint32_t iTimeSpecSec;
    uint32_t iTimeSpecNanoSec;
  } sMbsTimeStamp;
  ```
- Common data item header
  ```
  typedef struct{
    uint32_t iWords;       // following data words
    uint32_t iType;        // compatible with s_ve10_1, low=type (=10), high=subtype
  } sMbsHeader;
  ```

- *MBS* event header (type 10,1)
```
typedef struct{
  uint32_t iWords;       // data words + 4
  uint32_t iType;        // compatible with s_ve10_1, low=type (=10), high=subtype
  uint32_t iTrigger;
  uint32_t iEventNumber;
} sMbsEventHeader;
```
- *MBS* subevent header
```
typedef struct{
  uint32_t iWords;       // data words + 2
  uint32_t iType;        // compatible with s_ves10_1, low=type (=10), high=subtype
  uint32_t iSubeventID;  // 2 low bytes=procid, next byte=subcrate, high byte control
} sMbsSubeventHeader;
```

6.9.4.5 Some fixed numbers

```
#define LMD__TYPE_FILE_HEADER_101_1   0x00010065
#define LMD__TYPE_EVENT_HEADER_10_1   0x0001000a
#define LMD__TYPE_FILE_INDEX_101_2    0x00020065
#define LMD__TYPE_BUFFER_HEADER_10_1  0x0001000a
#define LMD__TYPE_BUFFER_HEADER_100_1 0x00010064
#define LMD__TYPE_TIME_STAMP_11_1     0x0001000b
#define LMD__INDEX          1
#define LMD__OVERWRITE      1
#define LMD__LARGE_FILE     1
#define LMD__BUFFER         1
#define LMD__NO_INDEX       0
#define LMD__NO_OVERWRITE   0
#define LMD__NO_LARGE_FILE  0
#define LMD__NO_BUFFER      0
#define LMD__NO_VERBOSE     0
#define LMD__VERBOSE        1
#define LMD__ENDIAN_BIG     2
#define LMD__ENDIAN_LITTLE  1
#define LMD__ENDIAN_UNKNOWN  0
```

6.9.5 *MBS* update for DIM control

6.9.5.1 New or modified files

New files:

f_dim_server.c, f_dim_server.h : all DIM functions.
dimstartup.sc, dimshutdown.sc : for single node
prmstartup.sc, prmshutdown.sc, dimremote_exe.sc : for multi node, propagate DIM_DNS_NODE
m_launch.c : fork programs to appear without path in ps output.
m_cmd2xml.c : Generate xml command description file from output of show command

Modified:

alias.com : add launch alias
m_prompt.c
m_dispatch.c
m_msg_log.c

f_ifa.c, f_ifa.h
f_mg_msg_output.c
f_mg_msg_thread.c : in /mbs/v51 has argument which is not specified in call!
f_pr_reset.c : kill processes in defined order.
f_stccomm.c : Socket created in stc_createserver will be shut down and closed by stc_close.
m_wait_for.c : Dont wait for zombies.
remote_exe.sc : use m_launch to run program.
Makefile : modules using DIM must include DIM path, all programs must link DIM library.

6.9.5.2 f_stccomm

On the server side `stc_createserver` fills structure `s_tcpcomm` with socket number (`s_tcpcomm.sock_rw`). `stc_acceptclient` returnes socket number of connection. This socket must be closed by `stc_discclient(socket)`. `stc_close` shuts down and closes socket `s_tcpcomm.socket`, not `s_tcpcomm.sock_rw`.

Therefore a server side `stc_close` did not close the server socket. This has been changed in that `s_tcpcomm.socket` is now equal `s_tcpcomm.sock_rw`. When a server called `stc_close` no more accept is possible as opposed to the current behavior.

On the client side a `stc_connectserver` returns the socket as well as setting `s_tcpcomm.socket`. Therefore in this case `stc_close` was always shutting down and closing the socket, whereas `stc_discclient(socket)` closes the socket.

6.9.5.3 *MBS* launcher

```
launch <program> [<program path>] [. <program args>]
```

If no program path is given, MBSROOT bin directory is assumed. Note that in the launched program environment PWD is the path from where `m_launch` was called, whereas the current path is the one of the program. Therefore program must `chdir(getenv("PWD"))` to work on the expected directory.

6.9.5.4 *MBS* DIM commands and parameters

The parameters and commands follow the *DABC* naming conventions. The DIM command MbsCommand is generic. The argument string is any *MBS* command. All `.scom` files are provided as DIM commands. New program `cmd2xml` generates an xml file with the *DABC* formatted description of all MBS commands.

6.9.5.5 DIM control modes

MBS can be controlled through DIM in two modes: single and multimode. In single mode the dispatcher is a DIM command server, the message logger a DIM parameter server for messages and status information (selected from DAQ status). In multinode mode the prompter is the DIM command server and the message loggers are status servers. The master message logger also is the DIM message server.

The table 6.1, page 77 shows an overview of the different operation modes.

Mode	Interactive	DIM GUI	Remote GUI
Single	Dispatcher:TTY	DIM command	-
	Logger: TTY,file	DIM status+message, file	-
Multi	Dispatcher:TCP	TCP	TCP
	Prompter:TTY	DIM command	TCP
	MasterLogger:TTY+file	file	file
	Msg server	DIM status+message	Msg server
	TCP inputs	TCP inputs	TCP inputs
	SlaveLogger:TCP	TCP,DIM status	TCP

Table 6.1: *MBS* operation modes.

6.9.5.6 Single node mode

There are two new scripts to start and shutdown a single node MBS:
`dimstartup.sc` and `dimshutdown.sc`.
These are called by `rsh` from the GUI node. Arguments are the path of MBSROOT and the user working path. For starting the DIM server also the DIM name server node is passed.

```
dimstartup.sc $MBSROOT $PWD $DIM_DNS_NODE
```

launches `m_dispatch -dim` after waiting for all 60xx sockets closed.

```
dimshutdown.sc $MBSROOT $PWD
```

calls `m_remote reset -l`. When dispatcher is started with -dim message logger is started with argv[1] = dim (otherwise task). Then the DIM commands are defined and dispatcher goes into `pause()` loop (needed with non threaded DIM version for keep alive signals).

When message logger is started with argv[1] = dim it creates the DIM parameters and starts a thread to update these every second. One DIM parameter is used for the messages and updated when a message arrives. Messages from local tasks are received in a thread (`f_mg_msg_thread`) and either sent to master message logger (when this one is slave) or processed by `f_mg_msg_output`. This function updates the DIM message parameter when in DIM mode, sends message to connected remote message client or prints it if not, and writes log file.

6.9.5.7 Multi node mode

In multi node mode the MBS nodes are controlled through one master node where a prompter is running. The prompter can be started (and stopped) interactively, or from a remote node by script. There are two new scripts to start and shutdown a multi node MBS from a remote node (GUI):
`prmstartup.sc` and `prmshutdown.sc`.
These are called by `rsh` from the GUI node. Arguments are the path of MBSROOT and the user working path. For starting the DIM server also the DIM name server node is passed.

```
prmstartup.sc $MBSROOT $PWD $DIM_DNS_NODE $REMOTE_NODE
```

launches `m_prompt -dim -r <remotenode>` after waiting for all 60xx sockets closed.

```
prmshutdown.sc $MBSROOT $PWD
```

calls `m_remote reset -l task=m_prompt`. With `m_wait_for -task m_prompt` it waits for prompter to be stopped, then calls `m_remote reset` (all nodes from `node_list.txt`). For the message logger modes see table 6.2, page 78. When prompter is started with -dim it starts the master message logger with argv[1] = masterdim (otherwise = master). When prompter is started with the -r <remote argument>, the remote node name is passed as argv[2] to the message logger.

All dispatchers are started in the MBS prompter by `f_ifa_init(NodeList,DimNameServer)` function. It calls per node from `NodeList f_ifa_remote` which starts the dispatcher in function `f_ifa_rsh_proc_start` by script `dimremote_exe.sc` (DIM mode) or `remote_exe.sc` (normal mode). When prompter is started with -dim then the DIM command server is started. Otherwise it starts TCP server on port 6006 (if started with the -r <remotenode> option) waiting for connection of GUI client and reading commands, or reading commands from terminal.

```
dimremote_exe.sc $MBSROOT $PWD $DIM_DNS_NODE $PROMPTER_NODE
```

launches `m_dispatch -dim -<prompternode>`. The dispatchers start their message logger with the same argument (prompter node) and optional argv[2] = slavedim. On the prompter node, however, the message logger should already run (started by prompter).

Because started with argv[2] = -<prompternode> the dispatchers then start a TCP server waiting on port 6004 for connection of prompter. When prompter connects, dispatcher sends process id. Then waits for commands. Command completion is sent back. Prompter may terminate, start again and connect.

The master logger starts a server in a thread waiting on port 6005 for connections of message logger slaves. For each slave a new thread is started waiting for messages of the slave. These threads are protected by mutex. Only one thread can write into logfile. However, slaves do not write logfile. If prompter was started from remote, master message logger starts server on port 6007 waiting for remote message client to connect. If remote node is specified, `pth_rem_serv` thread waits for connection of a message

argv	TCP	Slave	DIM	remote
none	0	0	-	-
task	0	0	-	-
master	1	0	-	pth_server→pth_links
dim	0	0	msg,status,pth_dim_serv	
masterdim	1	0	msg,status,pth_dim_serv	pth_server→pth_links
masternode	1	1	-	connect masternode
masterdim any	1	0	msg,status,pth_dim_serv	pth_server→pth_links
masternode slavedim	1	1	status,pth_dim_serv	connect masternode
master remotenode	1	0	msg,status,pth_dim_serv	pth_rem_serv,pth_server →pth_links

Table 6.2: *MBS* Message logger modes.

client. After connection global `l_tcp_chan_rem` is set and `f_mg_msg_output` send messages to that socket. As TCP master the `pth_server` thread waits for connections of message slaves. After connection starts `pth_links` thread to read messages and process in `f_mg_msg_output` (mutex locked). As TCP slave connect to <masternode>, set global `l_tcp_chan`. In DIM mode create services and start `pth_dim_serv` to update every second. In all cases `f_mg_msg_thread` is called where in slave mode messages are sent to the master, otherwise processed by `f_mg_msg_output`.

6.9.5.8 MBS controlled by DIM

Graphics on Eigene Dateien/experiments/mbs

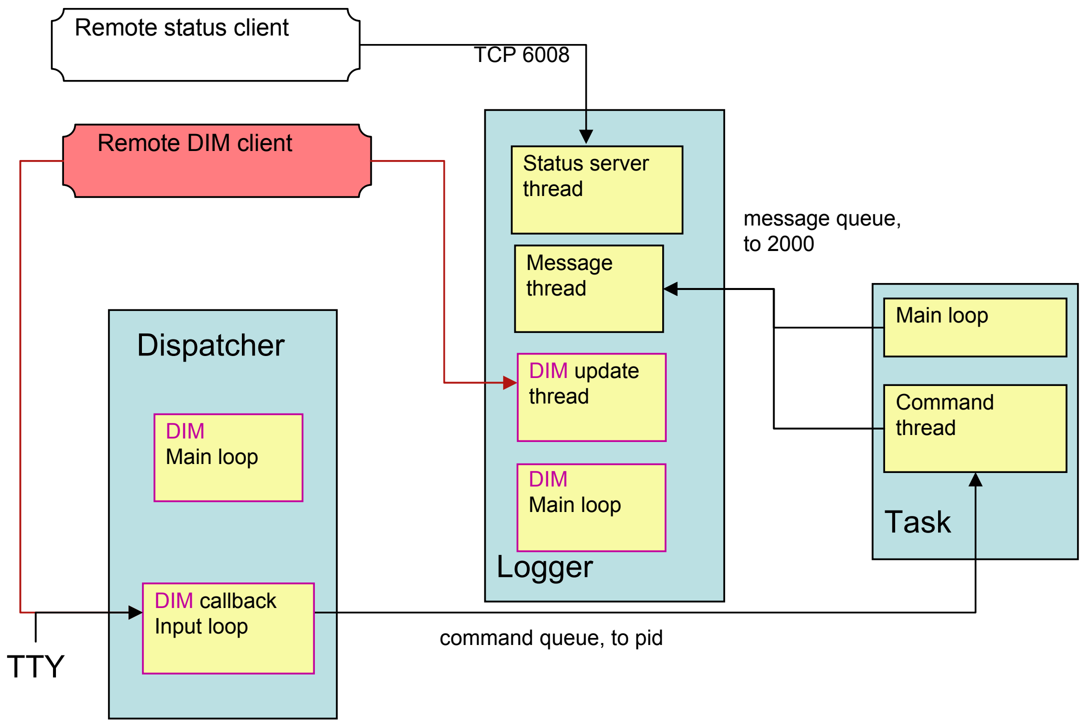
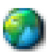


Figure 6.1: Single node *MBS* controlled by DIM.

6.10 List of icons

 browser Browser.

 comicon Command.

 conndsp Connect single MBS

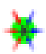
 connprm Connect MBS prompter

 control Test, shell script

 controlmbs *DABC* shell script

 controldabc *MBS* shell script

 dabcconfig Configure

 dabcicon *DABC* launcher

 dabcmbsicon *DABC MBS* launcher

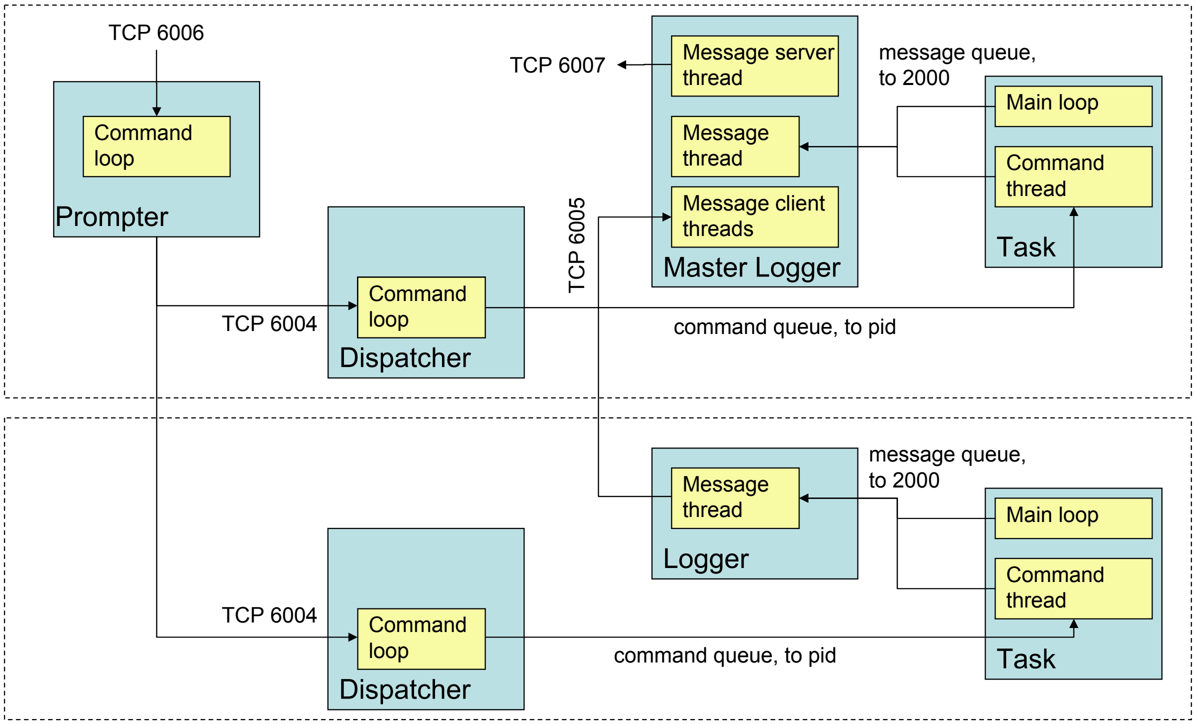


Figure 6.2: Multi node *MBS* controlled by TCP.

dabcstart Start acquisition

dabcstop Hold acquisition

disconn Shut down all

exitall Exit all

fileclose Close

filesave Save

histowin Histogram panel

info Show acquisition

infowin Info panel

logwin Log panel

mbsconfig Configure

mbsicon *MBS* launcher

mbsstart Start acquisition

mbsstop Stop acquisition

meterwin Rate meter panel

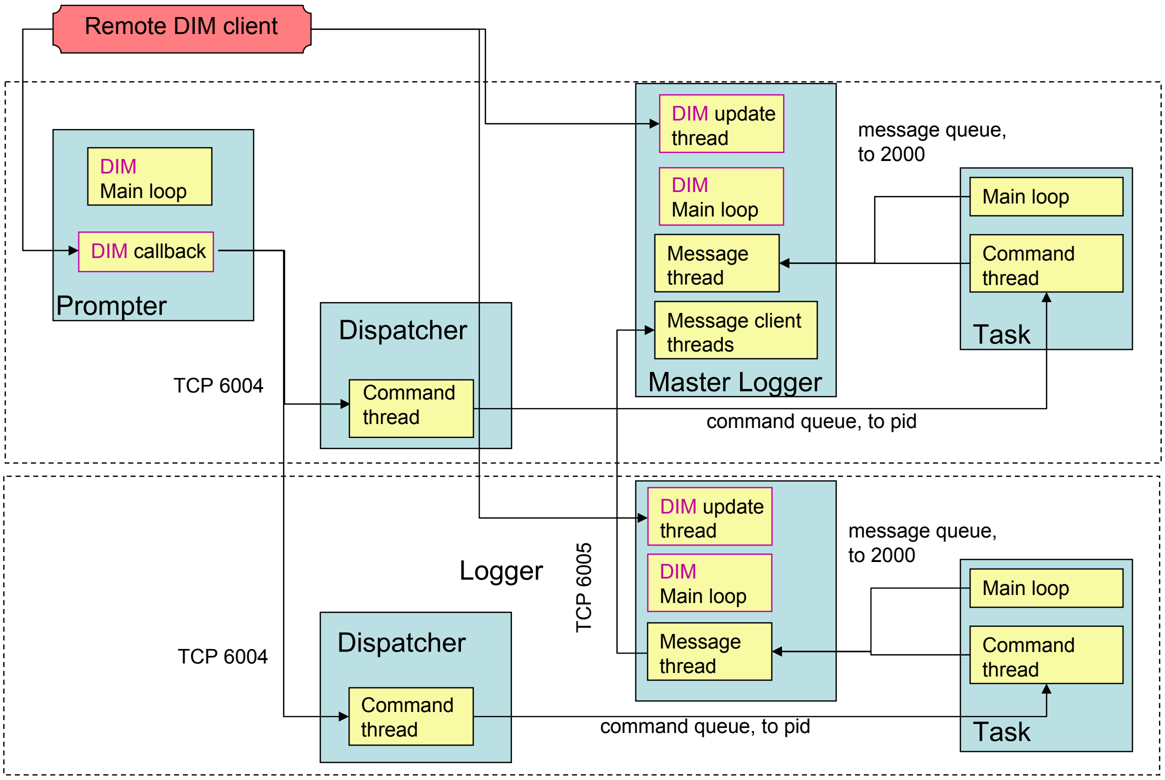


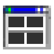
Figure 6.3: Multi node *MBS* controlled by DIM.

 paramwin Parameter table

 rshmbx *MBS* remote shell script

 savewin Save settings

 statewin State panel

 usergraphics Graphics panel

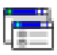
 usericon Parameter selection panel

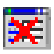
 usericonblue User panel

 usericonred User panel

 usericongreen User panel

 usericonyellow User panel

 user Windows

 windowclose Windows close

 windowblue Graphics window

 windowred Graphics window

 windowgreen Graphics window

Chapter 7

DABC Programmer Manual: Example Bnet

[programmer/prog-exa-bnet.tex]

7.1 Overview

Complex experiments feature a lot of front-end systems running in parallel. These take data and mark them with trigger information, or just with time stamps. To completely analyze such data, all portions belonging to the same event (or time stamp), must be combined in one processing unit. Such task is usually called "event building".

To support event building functionality in DABC , a special sub-framework called BNET ("Building NETwork") was introduced. Its main purpose is to simplify the experiment-specific implementation of such event building, distributed over several network nodes.

A typical event building network contains several **readout** nodes, which are each connected to several data sources. A readout node reads data from its data sources and combines together data parts which logically belong together; this is called "subevent building". In case of a triggered system it combines together data with the same trigger number; in case of time-stamped data it combines together data which belongs to the same time interval. Because there are several readout nodes, building a complete event requires to bring together all data of the same trigger number (or time interval, resp.) into the same **builder** node. Typically the system has not only a single builder node, but several of them; so full connectivity between all readouts and all builder nodes is necessary. Once all corresponding subevents have been delivered to the same builder node, the complete event may be build and eventually stored on disk or tape.

For such use case, the BNET framework defines a programming interface to implement the functional units (i. e. applications and modules), and it already provides several important components. BNET also defines the topology of these functional units which can be customized up to a definite level.

7.2 Controller application

The event building task is usually distributed over several nodes which must be controlled and synchronized. Therefore in BNET all nodes are classified by their functionality in two kinds: **controller** node, and **worker** nodes. Workers perform all data transport, and run the (sub-)event building code. The controller configures and steers all workers.

The controller node is implemented as ***bnet::ClusterApplication*** class. Via the control system interface the cluster controller distributes commands from the operator GUI to all workers. It observes the state of all workers, and may reconfigure them automatically if errors are detected.

The functionality of ***bnet::ClusterApplication*** is based on state-machine logic of *DABC* . All actions are performed during the execution of a state changing command, implemented in virtual method *DoStateTransition()*. A state transition on the cluster controller node means that the appropriate state transition is performed on all worker nodes. Technically speaking: a state machine command which is executed on the cluster controller is only completed if the state transition commands on all workers are successfully completed. This is implemented by means of class ***dabc::CommandsSet*** (see method *StartClusterSMCommand()* for details).

Class ***bnet::ClusterApplication*** has following configuration parameters:

Name	Type	Dflt	Description
NetDevice	str	dabc::SocketDevice	device class for network connections
NumEventsCombine	int	1	number of events (time frames) combined together
TransportBuffer	int	8192	size of buffer used for data transport cluster wide

Class ***bnet::ClusterApplication*** is fully functional and can be used as is for a real cluster set-up.

7.3 Worker application

The basic functionality of a worker is implemented in ***bnet::WorkerApplication*** class. Its main purpose is to instantiate, configure, connect, and run all working modules, triggered by commands from the cluster controller.

Main functionality of ***bnet::WorkerApplication*** is implemented in virtual method *CreateAppModules()* which is called during transition from Halted to Configured state. In this method all local modules are instantiated and configured. Some of these modules depend on the actual experiment, therefore class ***bnet::WorkerApplication*** provides a number of virtual methods to create experiment-specific components:

- *CreateCombiner()* - create a module combining several data sources to produce a subevent
- *CreateBuilder()* - create a module which combines N subevents to a complete event
- *CreateFilter()* - optional filter module to filter out events
- *CreateReadout()* - creates a readout transport connected to a data source
- *CreateStorage()* - creates a storage transport to store data on disk/tape

The user must define his/her own application class which inherits from ***bnet::WorkerApplication***, implementing these virtual methods.

Class ***bnet::WorkerApplication*** has the following "public" configuration parameters:

Name	Type	Dflt	Description
IsGenerator	bool	false	use generators instead of data sources
IsSender	bool	false	is sender module is created (readout functionality)
IsReceiver	bool	false	is receiver module is created (event builder functionality)
IsFilter	bool	false	is filter module is created (event builder should be true)
NumReadouts	int	1	number of data inputs
Inpit0Cfg	str		string parameter to configure input 0 - user specific
Inpit1Cfg	str		string parameter to configure input 1 and so on - user specific
StoragePar	str		string parameter to configure storage - user specific
ReadoutBuffer	int	2048	buffer size, used for readout
ReadoutPoolSize	int	4MB	size of memory pool for readout
TransportPoolSize	int	16MB	size of memory pool for data transport
EventBuffer	int	32768	buffer size, used for event building
EventPoolSize	int	4MB	size of memory pool for event building

There are also number of "private" parameters which are not seen by control system and cannot be configured via XML file:

Name	Type	Dflt	Description
CfgNodeID	int		node id (starts from 1 for workers)
CfgNumNodes	int		number of nodes in configuration
CfgSendMask	str		string in form of "xxox" defines which nodes are sender "x" or not "o"
CfgRecvMask	str		string in form of "xxox" defines which nodes are sender "x" or not "o"
CfgClusterMgr	str		name of cluster controller node
CfgNetDevice	str		name of configured network device, same as cluster param NetDevice
CfgEventsCombine	int		number of events combined together, same as cluster param NumEventsCombin
CfgReadoutPool	str		name of memory pool, used for readout ("ReadoutPool" or "TransportPool")
CfgConnected	bool		true when local configuration of application completed

These parameters are set during initialization phase. Some of them like CfgEventsCombine should be used by modules for it's configuration.

If required, the user subclass of ***bnet::WorkerApplication*** may define additional configuration parameters.

7.4 Combiner module

The combiner module merges together several data sources and produces subevent packets. Here a "subevent" means that data from all sources which belong to the same event (or time frame, resp.) are put into the same ***dabc::Buffer*** object. This buffer object should have a header with a unique identifier of type `bnet::EventId`; this is a 64-bit unsigned integer.

```
...
dabc::Buffer* buf = fPool->TakeBuffer(bufsize);
buf->SetHeaderSize(sizeof(bnet::EventId));
 *((bnet::EventId*) buf()->GetHeader()) = evid++;
...
```

The subevent identifier number should be subsequently increasing without a gap. When no data for the current identifier is available, an empty buffer with no data and correct header must be delivered to the output.

Class ***bnet::CombinerModule*** provides the prototype of a combiner module. It uses the following single parameter:

Name	Type	Dflt	Description
NumReadouts	int	1	number of data inputs

Actually, parameter NumReadouts may not be defined in the configuration of the module itself. Since class ***bnet::WorkerApplication*** already has a parameter of such name, its value will be directly used for the module configuration.

When implementing an experiment-specific combiner class, one should either derive it from ***bnet::CombinerModule*** class, or start "from scratch" by subclassing ***dabc::ModuleSync*** or ***dabc::ModuleAsync***. One may add more experiment-specific parameters to the module.

7.5 Network topology

The connection topology of the event building network is defined by parameters IsSender and IsReceiver of ***bnet::WorkerApplication***. These parameters configure the roles of each worker node:

- collector of data from data source(s) and sender to event builder
- receiever of data from collectors and builder of complete events
- both functions at the same application

It is required that at least one of both parameters has a true value. During configuration, the cluster controller establishes the connections between the workers such, that each sender module is connected with all receiver modules. This guarantees that each receiever node gets data from all sources, necessary to perform the full event building.

The two classes ***bnet::SenderModule***, and ***bnet::ReceiverModule***, implement the functionality of data sender, and data receiver, respectively. These classes are instantiated by ***bnet::WorkerApplication*** and should not be modified by the user.

The subevents buffers, as produced by the combiner module, are delivered to the sender module. Based on the event identifier, the buffer is send to that specific receiver where the event with such id will be build. For now a simple round-robin schedule is used by BNET, but in next *DABC* versions one or several other data transfer schedules will be implemented. One idea of the BNET framework is that such improvements are possible without changing the user application code.

7.6 Event builder module

The task of the receiver module is to collect all buffers of the same event identifier and deliver them at once to the event builder module.

To define an experiment-specific builder module, one can either derive it from ***bnet::BuilderModule*** class, or implement it "from scratch" by subclassing ***dabc::ModuleSync*** or ***dabc::ModuleAsync***. The event builder module has one input and one output. Over the input port it gets N buffers with subevents for the same event identifier. Over the output port it should deliver one buffer with build events.

When the user inherits his/her builder module from ***bnet::BuilderModule***, it is enough to implement the virtual *DoBuildEvent()* method, which gets as argument a list of N buffers with subevents. The format of the output buffer is completely user-defined. It is allowed to fill several events into the same output buffer if necessary.

7.7 Filter module

This is an optional component of BNET if build events shall be filtered before they are stored. To implement such filter, one can derive it from ***bnet::FilterModule*** and reimplement virtual method *TestBuffer()*. As an alternative, filtering can be implemented directly in the event builder module.

7.8 BNET test application

This application may test different asspects of a BNET without the necessity to have real data sources. The complete source code and configuration examples can be found in `$DABCSYS/applications/bnet-test` directory.

The example contains following classes:

- ***bnet::TestGeneratorModule***
- ***bnet::TestCombinerModule***
- ***bnet::TestBuilderModule***
- ***bnet::TestFilterModule***
- ***bnet::TestWorkerApplication***
- ***bnet::TestFactory***

There are several examples of configuration files. For instance, the configuration of 4 worker nodes with sender and receiver functionality each is shown in `SetupBnet.xml`:

```
<?xml version="1.0"?>
<dabc version="1">
  <Context host="lxi008" name="Controller:41">
    <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="libBnetTest.so"/>
       </Run>
       <Application class="bnet::TestWorker">
         <IsGenerator value="true"/>
         <IsSender value="true"/>
         <IsReceiver value="true"/>
         <NumReadouts value="4"/>
      </Application>
    </Context>
```

```
  </Defaults>
</dabc>
```

Here one can see cluster controller apllication in the beginning, configured to use ***dabc::SocketDevice*** for workers connections. And there are four workers with the same configurations parameters which can be found in <Defaults> section. In section <Context name="Worker*"> one sees, that IsGenerator, IsSender and IsReceiver parameters are all set to true. This defines the so-called "all-to-all" topology, i. e. each node communicates with all other nodes including itself. Parameter NumReadouts=4 means that there are 4 inputs on each combiner, resulting in 16 data sources for the complete system.

To run this example, one should specify correct host names for all contexts and start it with `run.sh SetupBnet.xml` command.

This can be used as template for developing a user-specific application. One can change functionality of combiner and builder modules, and provide a real readout instead of the generator module.

7.9 BNET for MBS application

This is a ready-ro-use implementation of distributed event building for MBS. The source code can be found in `$DABCSYS/applications/bnet-mbs`. It contains following classes:

- ***bnet::MbsCombinerModule***
- ***bnet::MbsBuilderModule***
- ***bnet::MbsWorkerApplication***
- ***bnet::MbsFactory***

Class ***bnet::MbsCombinerModule*** combines together events with the same event identifier from all inputs. In the cluster application parameter NumEventsCombine defines how many events should be bundled together in one buffer. It is crucial that transport buffer size is big enough for such number of subevents. During initialisation, cluster parameter NumEventsCombine is copied to each worker parameter CfgEventsCombine, which is finally used in ***bnet::MbsCombinerModule***:

```
bnet::MbsCombinerModule::MbsCombinerModule(...
   ...
   fCfgEventsCombine = GetCfgInt(CfgEventsCombine, 1, cmd);
   ...
```

For the moment ***bnet::MbsCombinerModule*** skips an event, if it is not present on all local data inputs.

Class ***bnet::MbsBuilderModule*** builds MBS events from the buffers as delivered by the receiver module. It also takes application parameter CfgEventsCombine to tell how many real MBS events are contained in the input buffers.

Application class ***bnet::MbsWorkerApplication*** implements several methods to correctly instantiate combiner and builder modules. It also implements virtual method *CreateReadout()*, where the input transport for the combiner module is created. In case of MBS there are three possibilities:

1. when IsGenerator=true module ***mbs::GeneratorModule*** connected as data input
2. when the appropriate readout parameter (like Input0Cfg for the first input) is a filename with ".lmd" suffix, the specified file will be used as data input
3. otherwise, the value of readout parameter (like Input0Cfg) will be used as MBS server name for connecting of ***mbs::ClientTransport*** to the appropriate data input

In virtual method *CreateStorage()* an output `.lmd` file will be created, if parameter StoragePar value is not empty.

Example file `SetupBnetMbs.xml` shows the configuration for an MBS event building with 2 readout nodes, connected with 2 generators each and 2 event builder nodes. This configuration file can be customised via `<Variables>` definitions in the beginning:

```
<?xml version="1.0"?>
<dabc version="1">
  <Variables>
     <node0 value="lxi008"/>
     <node1 value="lxi009"/>
     <node2 value="lxi010"/>
     <node3 value="lxi011"/>
     <node4 value="lxi012"/>
     <custport value="16015"/>
  </Variables>
  ...
</dabc>
```

Here node0 specifies the node where the cluster controller will run, node1 and node3 are used as readout nodes, node2 and node4 as builder nodes. On all four worker nodes one MBS generator application will be started. To run the application, just type `run.sh SetupBnetMbs.xml`.

Chapter 8

DABC Programmer Manual: Example ROC

[programmer/prog-exa-roc.tex]

8.1 Overview

The CBM ReadOut Controller (*ROC*) is an FPGA-based board to configure and read out the *nXYTER* chip [**?**], and to transport the acquired data over Ethernet to a PC. The software package *ROClib* provides the basic functionality to work with such *ROC*.

To support the usage of *ROC* in *DABC* , the following classes were implemented:

- ***roc::Device*** device class, wrapper for the ***SysCoreController*** class of *ROClib*
- ***roc::Transport*** corresponding transport, with access to the functionality of ***SysCoreBoard*** class of *ROClib*
- ***roc::CombinerModule*** module to combine data from several *ROCs* into a single output port
- ***roc::CalibrationModule*** module to calibrate the time scale in *ROC* data
- ***roc::ReadoutApplication*** application to perform readout from *ROC* boards
- ***roc::Factory*** factory class to organize these plugins

8.2 Device and transport

The *ROC* device class ***roc::Device*** inherits from two classes: ***dabc::Device*** and ***SysCoreControl***, where ***SysCoreControl*** provides simultaneous access to several *ROC* boards. Usually the instance of a device class corresponds to one physical device or board, but here the device object is rather used as central collection of ***SysCoreBoard*** objects, and as thread provider.

Each instance of ***roc::Transport*** has a pointer to a ***SysCoreBoard*** object which handles data taking from a specific *ROC*. The implementation of ***roc::Transport*** is based on the ***dabc::DataTransport*** class (see section 4.3.5) which runs as ***WorkingProcessor*** with an asynchronous event handling mechanism, so it does not require a dedicated thread. This feature allows to process several instances of such transports in the same thread. In the *ROC* case, all ***roc::Transport*** instances use the thread of ***roc::Device***.

Let's have a look how ***roc::Transport*** is working: When the connected module starts, method *StartTransport()* is called, which will invoke *SysCoreBoard::startDaq()* to start data taking. After that, the buffer filling loop consists in subsequent calls of *Read_Size()*, *Read_Start()* and *Read_Complete()* functions, implementing the interface of ***dabc::DataTransport*** (see section 4.3.5).

Method *Read_Size()* defines the size of the next buffer, required for data reading. In case of ***roc::Transport*** this size is fixed and is taken from a configuration parameter:

```
unsigned roc::Transport::Read_Size()
{
   return fBufferSize;
}
```

When the system has delivered a buffer of the requested size, function *Read_Start()* is called to start reading of that buffer from the data source:

```
unsigned roc::Transport::Read_Start(dabc::Buffer* buf)
{
   int req = fxBoard->requestData(fReqNumMsgs);
   if (req==2) return dabc::DataInput::di_CallBack;
   if (req==1) return dabc::DataInput::di_Ok;
   return dabc::DataInput::di_Error;
}
```

The ***SysCoreBoard*** (accessed by pointer `fxBoard`) keeps internally own buffers of the received UDP messages. The call *SysCoreBoard::requestData()* informs by return value either that the required number of messages is already received, or that the caller should wait (i. e. poll this method here until it returns that all data is ready). However, waiting would mean that the working thread was blocked and could not run the other transport instances. Therefore, another approach is used: The *ROClib* will call back the virtual method *SysCoreControl::DataCallBack()* when the required amount of data is there. This method is implemented for ***roc::Transport*** to complete filling the current ***dabc::Buffer***.

If data already exists in the internal buffers of ***SysCoreBoard***, the value dabc::DataInput::di_Ok is returned; then the ***DataTransport*** framework will immediately call *Read_Complete()* which finally fills the *DABC* buffer:

```
unsigned roc::Transport::Read_Complete(dabc::Buffer* buf)
{
   unsigned fullsz = buf->GetDataSize();
   if (!fxBoard->fillData((char*) buf->GetDataLocation(), fullsz))
      return dabc::DataInput::di_SkipBuffer;
   if (fullsz==0)
      return dabc::DataInput::di_SkipBuffer;
   buf->SetTypeId(roc::rbt_RawRocData);
   buf->SetDataSize(fullsz);
   return dabc::DataInput::di_Ok;
}
```

The return value dabc::DataInput::di_CallBack of function *Read_Start()* indicates that processing of this transport should be suspended, because the requested amount of data is not ready yet. When all this data has been received by the *ROClib*, it will invoke method *SysCoreControl::DataCallBack()* which is reimplemented in subclass ***roc::Device***, simply forwarding to the following method of ***roc::Transport***:

```
void roc::Transport::CompleteBufferReading()
{
   unsigned res = Read_Complete(fCurrentBuf);
   Read_CallBack(res);
}
```

As the required amount of data is ready now, one only retrieves it to the current buffer with the same *Read_Complete()* method, and reactivates the processing of this transport instance by calling *Read_CallBack()*.

8.3 Combiner module

Class ***roc::CombinerModule*** combines data from several ROC boards in one MBS event. It also performs sorting of data according the timestamp, resolves the "last epoch" bits, and fixes several coding errors (class ***SysCoreSorter*** is used for this).

The module has following configuration parameters:

- NumRocs - number of *ROC* boards, connected to combiner [default 1]
- BufferSize - size of buffer, used to read data from *ROCs* [default 16384]
- NumOutputs - number of outputs [default 2]

As output *MBS* events are provided. Each *MBS* event contains *ROC* messages between two *sync markers*. For each *ROC* a separate *MBS* subevent is allocated; field *iSubcrate* of the subevent header contains the *ROC* id.

8.4 Calibration module

Class ***roc::CalibrationModule*** performs the calibration of the time scale for all *ROCs* and merges all messages into a single data stream. As output, an *MBS* event with a single subevent is produced.

The module has following configuration parameters:

- NumRocs - number of ROC boards, which should be provided in MBS event [default 2]
- BufferSize - size of buffer, used to produce output data [default 16384]
- NumOutputs - number of outputs [default 2]

8.5 Readout application

The main aim of ***roc::ReadoutApplication*** class is to configure and run the application which combines the data readouts from several *ROC*s. It can store the data into a `.lmd` file, and it may provide *MBS* *stream* or *transport* servers for online monitoring, e. g. with a remote *Go4* analysis. It has following configuration parameters:

- NumRocs - number of ROC boards
- RocIp0, RocIp1, RocIp2, ... - addresses (IP or nodname) of ROC boards
- DoCalibr - defines calibration mode (see further)
- BufferSize - size of buffer
- NumBuffers - number of buffers
- MbsServerKind - kind of MBS server ("None", "Stream", "Transport")
- RawFile - name of `.lmd` file to store "raw" combined data (after ***CombinerModule***)

- CalibrFile - name of `.lmd` file to store "calibrated" data (after ***CalibrationModule***)
- MbsFileSizeLimit - maximum size of each file, in Mb. If the written data would exceed this size, a new output file is automatically opened with a sequence number appended to the file name.

Three calibration modes are supported:

- DoCalibr=0 - Only the ***CombinerModule*** is instantiated, which produces a kind of *ROC* "raw" data
- DoCalibr=1 - Both ***CombinerModule*** and ***CalibrationModule*** are instantiated
- DoCalibr=2 - Only the ***CalibrationModule*** is instantiated. This is used to convert "raw" data read from `.lmd` files into the "calibrated" format.

In all modes output in form of raw or (and) calibrated data can be stored in `.lmd` file(s), defined by RawFile and CalibdFile parameters respectively. The last mode is a special case, since RawFile does not specify the output, but the input file for the calibration module.

8.6 Factory

Factory class ***roc::Factory*** implements several methods to create the *ROC*-specific application, device and modules, as described in section 4.5.

8.7 Source and compilation

The source code of all classes can be found in `$DABCSYS/plugins/roc` directory. Compiled library `libDabcKnut.so` is in directory `$DABCSYS/lib`. If one needs to modify some code in this library, one should copy the sources to a user directory and call "make" in this directory. In this case the library is build into a subdirectory, named like `$ARCH/lib`, where `$ARCH` is the current CPU architecture (for instance, "i686").

8.8 Running the *ROC* application

To run the readout application, an approprite XML configuration file is required. There are two examples of configuration files in `$DABCSYS/applications/roc`.

File `Readout.xml` configures the readout from 3 ROCs:

```
<?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>
```

Because this is a single-node application, it 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.

File `Calibr.xml` shows the special case of a configuration to convert "raw" data into "calibrated" data without running any real DAQ:

```
<?xml version="1.0"?>
<dabc version="1">
<Context name="Calibr">
  <Run>
    <lib value="libDabcMbs.so"/>
    <lib value="libDabcKnut.so"/>
    <logfile value="Calibr.log"/>
  </Run>
  <Application class="roc::Readout">
    <DoCalibr value="2"/>
    <NumRocs value="3"/>
    <BufferSize value="65536"/>
    <NumBuffers value="100"/>
    <RawFile value="/d/cbm06/cbmdata/SEP08/raw/run028/run028*.lmd"/>
    <MbsServerKind value="Stream"/>
    <CalibrFile value="testcal.lmd"/>
    <MbsFileSizeLimit value="110"/>
  </Application>
 </Context>
</dabc>
```

Here the "raw" data is read from the files matching the name wildcard pattern as defined in the `<RawFile>` tag. Note that this example will read subsequently all data of run "028" which possibly was saved into several files with subsequent numbers appended to their names, due to the `<MbsFileSizeLimit>` mechanism as described above. The "calibrated" data is written as usual to the output file as specified in the `<CalibrFile>` tag.

Chapter 9

DABC Programmer Manual: Example PCI

[programmer/prog-exa-pci.tex]

9.1 Overview

Reading data streams from a PCI board into the PC is a common use case for data acquisiton systems. In *DABC* one can implement access to such boards by means of special ***Device*** and ***Transport*** classes that communicate with the appropriate linux device driver. The ***Device*** represents the board and may do the hardware set-up at Configure time, using dedicated ***dabc::Parameters***. The ***Transport*** may fill its data buffers via board DMA, and pass the ***Buffers*** to the connected readout ***Module***.

This example treats the **Active Buffer Board** (*Active Buffer Board*) [?] , a PCI express (PCIe) board with a *Virtex 4* FPGA and optical connectors to receive data from the experiment frontend hardware. It is developed for the CBM experiment [?] by the *Institut f. Technische Informatik* at Mannheim University. The board developers deliver a kernel module as linux device driver, and the ***mprace::*** C++ library to work with the board from user space.

Since this driver software may also be applied for other PCIe boards, the corresponding *DABC* classes ***pci::Device*** and ***pci::Transport*** are rather generic, using namespace ***pci::***. The special properties of the *Active Buffer Board*board are then implemented in a ***pci::BoardDevice*** subclass and in further classes with namespace ***abb::***.

Besides some simple test executables that read from and write to the *Active Buffer Board*on a single machine, there is an example of a ***bnet::WorkerApplication*** that applies the *Active Buffer Board*classes for the readout module.

9.2 PCI Device and Transport

9.2.1 pci::BoardDevice

Subclass of ***dabc::Device***. Adapter to the the ***mprace::Board*** functionality, i.e. the generic PCIe.

1. It implements the **Transport factory method** *CreateTransport()*. This will create a ***pci::Transport*** and assign a dedicated working thread for each transport. The *DABC* framework will use this

method to establish the data connection of a ***Port*** with the PCI device.

2. It defines a **plug-in point** for an abstract board component: The device functionalities may require driver implementations that are more board specific. Because of this, the ***mprace::*** library provides base class ***Board*** with some virtual methods to work on the driver. This is applied here as handle to the actual ***Board*** implementation (e. g. a ***mprace:ABB***) that must be instantiated in the constructor of the subclass. Note that all functionalites require a real ***Board*** implementation, thus it is not possible to instantiate a mere ***pci::BoardDevice*** without subclassing it!
3. It adds **Device specific commands** ***CommandSetPCIReadRegion*** and ***CommandSetPCIWriteRegion*** that define the regions in the PCI address space for reading or writing data, resp. Method *ExecuteCommand(Command*)* is extended to handle such commands.
4. It manages the **scatter-gather mapping** of userspace ***dabc::Buffer*** objects for the DMA engine. These are taken from a regular DABCmemory pool and are each mapped to a ***mprace::DMABuffer*** representation. The DMA mapping is done in method *MapDMABuffer()* which gets the reference to the ***dabc::MemoryPool**** that is used for the ***pci::Transport***. This is required at ***Device*** initialization time; the mapping must be refreshed on the fly if the memory pool changes though.
   Method *GetDMABuffer(dabc::Buffer*)* will deliver for each ***dabc::Buffer**** of the mapped memory pool the corresponding ***mprace::DMABuffer**** object to be used in the underlying ***mprace::*** library. These are associated by the ***dabc::Buffer*** id number which defines the index in the ***std::vector*** keeping the ***mprace::DMABuffer**** handles.
   Method *DoDeviceCleanup()* is implemented for a proper cleanup of the mapped DMA buffers when the ***Device*** is removed by the framework.
5. **Reading data from the board:** Method *ReadPCI(dabc::Buffer*)* implements reading one buffer from PCI, using the BAR, the PCI start address, and the read size, as specified before. These read parameters may be either set by method *SetReadBuffer(unsigned int bar, unsigned int address, unsigned int length)*, or by submitting the corresponding command ***CommandSetPCIReadRegion*** to the ***pci::Device***
   If DMA mode (defined in the constructor) **is not** enabled , this will just use PIO to fill the specified ***dabc::Buffer*** from the PCI address range. If DMA mode **is** enabled, it will perform DMA into the user space ***dabc::Buffer****; this must be taken from a memory pool that was mapped before by means of *MapDMABuffer()*. This is a synchronous call that will initiate the DMA transfer and block until it is complete.
   For asynchronous DMA (double buffering of ***dabc::DataTransport***) following virtual methods are provided: Method *ReadPCIStart(dabc::Buffer*)* may start the asynchronous filling of one mapped ***Buffer*** from the configured PCI board addresses. It should not wait for the completion of the data transfer, but return immeadiately without blocking after triggering the DMA. In contrast to this, *ReadPCIComplete(dabc::Buffer*)* must wait until the DMA transfer into the specified ***Buffer*** is completely finished. So the ***pci::Transport*** will initiate DMA by *ReadPCIStart()* and check for DMA completion by *ReadPCIComplete()*. A subclass of ***pci::BoardDevice*** may re-implement these methods with board specific functionalities.
6. **Writing data to the board:** Method *WritePCI(dabc::Buffer*)* implements writing data from a DABC buffer to the PCI address space, using the BAR, the PCI start address, and the write size, as specified before. These write parameters may be either set by method *SetWriteBuffer(unsigned int bar, unsigned int address, unsigned int length)*, or by submitting the corresponding command ***CommandSetPCIWriteRegion*** to the ***pci::Device***. If DMA mode (defined in the constructor) **is not** enabled , this will just use PIO to transfer the specified ***dabc::Buffer*** to the PCI addresses.
   If DMA mode **is** enabled, it will perform DMA from the user space ***dabc::Buffer****; this must be taken from a memory pool that was mapped before by means of *MapDMABuffer()*. This call will initiate the DMA transfer and block until it is complete. Currently there is no asynchronous implementation for data output to PCI, since this is a rare use case for a DAQ system.

9.2.2 pci::Transport

This class handles the connection between the ***Port*** of a module and the PCI device. It is created in *CreateTransport()* of ***pci::BoardDevice*** when the user application calls the corresponding ***Manager*** method with the names of the port and the device instances, e. g.
`dabc::mgr()->CreateTransport("ReadoutModule/Input", "AbbDevice3");`

It extends the base class ***dabc::DataTransport*** which already provides generic ***Buffer*** queues with a data backpressure mechanism, both for input and output direction. Because this class is also a ***WorkingProcessor***, each ***pci::Transport*** object has a dedicated thread that runs the IO actions. The following virtual methods of ***dabc::DataTransport*** were implemented:

unsigned Read_Size() : Returns the size in byte of the next buffer that is to be read from board. Uses the current readout length as set for the ***pci::BoardDevice*** with *SetReadBuffer()*, or ***CommandSetPCIReadRegion***, resp.

unsigned Read_Start(dabc::Buffer* buf) : This initiates the reading into buffer *buf* and returns without waiting for completion. The functionality is forwarded to *ReadPCIStart()* of ***pci::BoardDevice***. When *Read_Start()* returns, the transport thread can already push the **previously** filled DMA buffer to the connected ***Port***, which may wake up the waiting thread of its ***Module*** for further processing. Thus base class ***dabc::DataTransport*** implicitly provides a double-buffering mechanism here.

unsigned Read_Complete(dabc::Buffer* buf) : Will wait until filling the buffer *buf* from a DMA read operation is completed. The DMA either must have been started asynchronously by a previous *Read_Start()* call; or it must be started synchronously here. This method is used by the base class for synchronization between transport thread and DMA engine of the PCI board. The functionality is forwarded to *ReadPCIComplete()* of ***pci::BoardDevice***.

bool WriteBuffer(dabc::Buffer* buf) : Write content of *buf* to the PCI region as set for the ***BoardDevice*** with *SetWriteBuffer()*, or ***CommandSetPCIWriteRegion***, resp. This is a pure synchronous method, i. e. it will start the DMA transfer and return no sooner than it's completed. The functionality is forwarded to *WritePCI()* of ***pci::BoardDevice***.

void ProcessPoolChanged(dabc::MemoryPool* pool) : Is called by the framework whenever the memory pool associated with the transport instance changes, e. g. at transport connection time, pool expansion, etc. It calls *MapDMABuffers()* of ***pci::BoardDevice*** to rebuild the scatter-gather mappings for each buffer of the pool.

9.3 Active Buffer Board implementation

9.3.1 abb::Device

This subclass of ***pci::BoardDevice*** adds some functionality that is rather specific to the *Active Buffer Board* hardware and the test environment.

1. The constructor instantiates the ***mprace::Board*** component for the *Active Buffer Board* functionalities. Additionally, a DMA engine component ***mprace::DMAEngineWG*** is applied for all DMA specific actions.
2. It implements the actual asynchronous DMA by overriding methods *ReadPCIStart()* and *ReadPCIComplete()*. The base class ***pci::BoardDevice*** can provide synchronous DMA only, because the generic ***mprace::Board*** interface does not cover asynchronous features. These are handled by the ***DMAEngineWG*** component.

3. The constructor uses several **configuration parameters**:
```
unsigned int devicenum = GetCfgInt(ABB_PAR_BOARDNUM, 0, cmd);
unsigned int bar = GetCfgInt(ABB_PAR_BAR, 1, cmd);
unsigned int addr = GetCfgInt(ABB_PAR_ADDRESS, (0x8000 >> 2), cmd);
unsigned int size = GetCfgInt(ABB_PAR_LENGTH, 8192, cmd);
```

   The parameter names are handled by string definitions in `abb/Factory.h`:
```
#define ABB_PAR_BOARDNUM    "ABB_BoardNumber"
#define ABB_PAR_BAR         "ABB_ReadoutBAR"
#define ABB_PAR_ADDRESS     "ABB_ReadoutAddress"
#define ABB_PAR_LENGTH      "ABB_ReadoutLength"
```

   The *GetCfgInt()* will look for a parameter of the specified name already existing in the system, e. g. if the ***Application*** object has defined such. If not, a ***dabc::Parameter*** of that name will be created and exported to the control system. If the configuration file specifies a value for this parameter, it will be set; otherwise, the default value (second argument of *GetCfgInt()*) is set.
   If the constructor gets a command object *cmd* as argument containing a parameter of the specified name, this command's parameter value will override all other values for this parameter defined elsewhere in the system. The user may pass such a *cmd* to the ***abb::Device*** either as third argument of the manager factory method *CreateDevice()*; or by means of a ***dabc::CmdCreateDevice*** object which is invoked by *Execute()* of the manager. This is useful if the device is to be tested without any configuration or control system, as shown in the examples of section 9.4.
4. It provides **pseudo event data** for the Bnet test example in the received DMA buffers: Method *ReadPCI()* is extended to copy an event header of the Bnet format (i.e. incrementing event count and unique id) into each output Buffer after the base class *ReadPCI()* is complete. This workaround is necessary since the *Active Buffer Board*data itself does not contain any information in the test setup. Additionally, method *DoDeviceCleanup()* will reset the event counters at the end of each DAQ run.

9.3.2 abb::ReadoutModule

Subclass of ***dabc::ModuleAsync***; generic implementation of a readout module to use the ***BoardDevice***.

1. It creates the memory pool which is used for DMA buffers in the ***pci::BoardDevice***; this pool is propagated to the device via the ***pci::Transport*** when module is connected, since device will use the pool associated with the connection port.
2. Module runs either in standalone mode (one input port, no output) for testing; or in regular mode (one input port, one output port)
3. *ProcessUserEvent()* defines the module action for any *DABC* events, e. g. input port has new buffer. In standalone mode, the received buffer is just released. In regular mode, buffer is send to the output port.
4. It has a ***dabc::Ratemeter*** object which is updated for each packet arriving in *ProcessUserEvent()*. The average data throughput rate is then printed out to the terminal on stopping the module in *AfterModuleStop()*. Alternatively, by means of method *CreateRateParameter()* it also defines a rate parameter "DMAReadout" that is linked to the input port "Input" and may export the current data rate to the control system.

9.3.3 abb::WriterModule

Subclass of ***dabc::ModuleSync***; generic implementation of a writer module to use the ***BoardDevice***.

1. Creates the memory pool which is used for DMA buffers in the ***pci::BoardDevice***; this pool is propagated to the device via the ***pci::Transport*** when module is connected, since device will use the pool associated with the connection port.
2. Module runs either in standalone mode (one output port, no input) for testing; or in regular mode (one input port, one output port)
3. *MainLoop()* defines the module action. In standalone mode, a new buffer is taken from the memory pool and send to the output port. In regular mode, the send buffer is taken from the input port.
4. It has a ***dabc::Ratemeter*** object which is updated for each packet arriving in *MainLoop()*. The average data throughput rate is then printed out to the terminal on stopping the module in *AfterModuleStop()*. Alternatively, by means of method *CreateRateParameter()* it also defines a rate parameter "DMAWriter" that is linked to the input port "Output" and may export the current data rate to the control system.

9.3.4 abb::Factory

A subclass of ***dabc::Factory*** to plug in the *Active Buffer Board* classes:

1. Implements *CreateDevice()* for the ***abb::Device***. The third argument of this factory method is a ***dabc::Command*** that may contain optional setup parameters of the device.
2. Implements *CreateModule()* for the ***abb::ReadoutModule*** and the ***abb::WriterModule***. Third argument of this factory method is a ***dabc::Command***, containing optional setup parameters of the module.
3. The factory is created automatically as static (singleton) instance on loading the `libDabcAbb.so`.

9.4 Simple read and write tests

The functionality of the *Active Buffer Board* can be tested with several simple executables which are provided in the `test` subfolder of the `abb` plugin package.

9.4.1 DMA Read from the board

The example code `abb_test_read.cxx` shows in a simple *main()* function how to utilize the ***abb::*** classes . for a plain readout with DMA.

1. It applies the ***dabc::StandaloneManager*** as most simple ***Manager*** implementation.
```
int nodeid=0; // this node id
int numnodes=1; // number of nodes in cluster
...
dabc::StandaloneManager manager(0, nodeid, numnodes);
```
2. The ***abb::Device*** is created by means of a command ***CmdCreateDevice*** which is passed to the manager. The command wraps also some initial parameters for the device which are then evaluated in method *CreateDevice()* of ***abb::Factory***:
```
#define READADDRESS  (0x8000 >> 2)
#define READSIZE 16*1024
...
```

```
std::string devname="ABB";
dabc::Command* dcom= new dabc::CmdCreateDevice("abb::Device",
                                                devname.c_str());
// arguments: (class name, device name)
// set additional parameters for abb device here:
dcom->SetInt(ABB_PAR_BOARDNUM, BOARD_NUM);
dcom->SetInt(ABB_PAR_BAR, 1);
dcom->SetInt(ABB_PAR_ADDRESS, READADDRESS);
dcom->SetInt(ABB_PAR_LENGTH, readsize);
res=manager.Execute(dcom);
DOUT1(("CreateDevice = %s", DBOOL(res)));
```

Here the parameter names (e. g. `ABB_PAR_ADDRESS`) use the string definitions as set in `abb/Factory.h`. The parameter values are defined locally (e. g. `READADDRESS`); however, the DMA transfer size `readsize` may be set by the executables's first command line parameter. Boolean variable `res` contains the result of the command execution (true or false) which is printed as debut output to the terminal with the `DOUT1()` macro.

3. It creates a ***abb::ReadoutModule*** by means of a command ***CmdCreateModule*** which is passed to the manager. The command wraps also some initial parameters for the module which are then evaluated in method *CreateModule()* of ***abb::Factory***:

```
cmd = new dabc::CmdCreateModule("abb::ReadoutModule",
                                "ABB_Readout",
                                "ReadoutThread");
// arguments: (class name, module name, thread name)
cmd->SetInt(ABB_COMPAR_BUFSIZE, readsize);
cmd->SetInt(ABB_COMPAR_STALONE,1);
cmd->SetInt(ABB_COMPAR_QLENGTH, 10);
cmd->SetStr(ABB_COMPAR_POOL,"ABB-standalone-pool");
res=manager.Execute(cmd);
DOUT1(("Create ABB readout module = %s", DBOOL(res)));
```

Again the parameter names (e. g. `ABB_COMPAR_QLENGTH`) use common string definitions as set in `abb/Factory.h`, such as: the size of the memory pool buffers `ABB_COMPAR_BUFSIZE` which is set to the required DMA transfer size`readsize`; the standalone run mode of the module `ABB_COMPAR_STALONE`; the port queue length `ABB_COMPAR_QLENGTH`; the name of the module's memory pool `ABB_COMPAR_POOL`, which also.

4. The transport connection between the input port of the readout module and the ***abb::Device*** is established by a direct method call of the manager:

```
res = manager.CreateTransport("ABB_Readout/Input", devname.c_str());
DOUT1(("Connected module to ABB device = %s", DBOOL(res)));
```

The manager will find the *Active Buffer Board* device instance by the string `devname` and use its factory method *CreateTransport()* to instantiate a ***pci::Transport*** that will be connected to the port of name "ABB_Readout/Input".

5. The readout module processing is started by name with a manager method:

```
manager.StartModule("ABB_Readout");
DOUT1(("Started readout module...."));
```

Then the main process waits for 5 seconds while the *DABC* threads and the board DMA performs the data transfer. The module is stopped again then.

```
sleep(5);
manager.StopModule("ABB_Readout");
DOUT1(("Stopped readout module."));
```

After the module has stopped, its internal ***dabc::Ratemeter*** will print some average data rate values

to the terminal. Finally, all objects are destroyed and the manager is cleaning up the process before the program ends:
`manager.CleanupManager();`

9.4.2 DMA Write to the board

The example code `abb_test_write.cxx` shows in a simple *main()* function how to utilize the ***abb::*** classes to write data from the PC to the *Active Buffer Board* with DMA . The code is very similar to the read example as described in the above section 9.4.1:

1. It applies the ***dabc::StandaloneManager*** as most simple ***Manager*** implementation.
2. The ***abb::Device*** is created by means of a command ***CmdCreateDevice*** which is passed to the manager. The command contains the initial parameters for the device. The DMA transfer size `readsize` may be set by the executables's first command line parameter (see section 9.4.1 for code example).
3. It creates a ***abb::WriterModule*** by means of a command ***CmdCreateModule*** which is passed to the manager. The command contains the initial parameters for the module which are then evaluated in method *CreateModule()* of ***abb::Factory***:
```
cmd = new dabc::CmdCreateModule("abb::WriterModule",
                                "ABB_Sender",
                                "WriterThread");
cmd->SetInt(ABB_COMPAR_BUFSIZE, readsize);
cmd->SetInt(ABB_COMPAR_STALONE,1);
cmd->SetInt(ABB_COMPAR_QLENGTH, 10);
cmd->SetStr(ABB_COMPAR_POOL,"ABB-standalone-pool");
cmd->SetStr(ABB_PAR_DEVICE,devname.c_str());
res = manager.Execute(cmd);
DOUT1(("Create ABB writer module = %s", DBOOL(res)));
```
   Again the parameter names are expressd by common string definitions as set in `abb/Factory.h`.
4. The transport connection between the output port of the writer module and the ***abb::Device*** is established by a direct method call of the manager:
```
res = manager.CreateTransport("ABB_Sender/Output", devname.c_str());
DOUT1(("Connected module to ABB device = %s", DBOOL(res)));
```
5. The writer module's processing is started with a manager method:
   `manager.StartModule("ABB_Sender")`. The main process waits 5 seconds while the *DABC* threads and the board DMA perform the data transfer. The module is stopped again then. After the module has stopped, its internal ***dabc::Ratemeter*** will print some average data rate values to the terminal. Finally, the manager is cleaning up all objects and the program terminates.

9.4.3 Simultaneous DMA Read and Write

The example code `abb_test.cxx` shows in a simple *main()* function how to utilize the ***abb::*** classes to write data from the PC to the *Active Buffer Board* in one DMA channel, and simultaneously read data back from the board with another DMA . channel.

It applies the ***abb::Device*** both with a ***abb::WriterModule*** and a ***abb::ReadoutModule*** that run in different threads. So the code is a merger of the above examples 9.4.1 and 9.4.2:

1. It applies the ***dabc::StandaloneManager*** as most simple ***Manager*** implementation.
2. The ***abb::Device*** is created by means of a command ***CmdCreateDevice*** which is passed to the manager. The command contains the initial parameters for the device. The DMA transfer size

`readsize` (same for both directions) may be set by the executables's first command line parameter (see section 9.4.1 for code example).

3. It creates a ***abb::ReadoutModule*** by means of a command ***CmdCreateModule*** which is passed to the manager (see section 9.4.1 for code example).
4. It creates a ***abb::WriterModule*** by means of a command ***CmdCreateModule*** which is passed to the manager (see section 9.4.2 for code example).
5. The transport connections of the ***abb::Device*** both with the input port of the reader module, and the output port of the writer module are established by invoking method *CreateTransport()* of the manager (see sections 9.4.1 and 9.4.2 for comments on the code)
6. Both modules are started with `manager.StartModule("")`. The main process sleeps for 60 seconds during the DMA transfer, then it stops both modules again. After the modules have stopped, their internal ***dabc::Ratemeter*** instances will print some average data rate values to the terminal. Finally the manager is cleaned up and the program ends.

9.5 Active Buffer Board with Bnet application

The DAQ builder network (Bnet) example as described in section 7.8 may optionally utilize the *Active Buffer Board* as input for the Readout module. This is provided in class ***bnet::TestWorkerApplication*** which implements the ***bnet::WorkerApplication***: The Bnet factory method

```
bool bnet::TestWorkerApplication::CreateReadout(const char* portname,
                                                int portnumber)
```

will instantiate an ***abb::Device*** if the configuration parameter for the portnumber *p* ("Input*p*Cfg", as delivered by *ReadoutPar(p)*) is set to "ABB". This ***abb::Device*** is connected directly to the input port of the standard Bnet combiner module, as specified by the *portname* argument of the method:

```
if(ReadoutPar(portnumber) == "ABB") {
       const char* abbdevname = "ABBDevice";
       fABBActive = dabc::mgr()->CreateDevice("abb::Device",
                                              abbdevname);
       res = dabc::mgr()->CreateTransport(portname, abbdevname);
       if (!res) EOUT(("Cannot create ABB transport"));
   }
```

Note that the ***abb::ReadoutModule*** is **not used** here; this is applied for the simple examples only, see section 9.4).

Any other value of *ReadoutPar(p)* will apply the ***TestGeneratorModule*** of the standard Bnet example.

The parameters for the *Active Buffer Board* can be set in the XML configuration file, using the names as defined in `abb/Factory.h`. This may look as follows:

```
...
<Context host="node01" name="Worker2:42">
  <Run>
   <lib value="${DABCSYS}/lib/libpcidriver.so"/>
   <lib value="${DABCSYS}/lib/libmprace.so"/>
```

```
    <lib value="${DABCSYS}/lib/libDabcAbb.so"/>
    <lib value="libBnetTest.so"/>
  </Run>
  <Application class="bnet::TestWorker">
    <NumReadouts value="1"/>
    <Input0Cfg value="ABB"/>
  </Application>
  <Device class="abb::Device">
    <ABB_BoardNumber value="0"/>
    <ABB_ReadoutBAR value="1"/>
    <ABB_ReadoutAddress value="8192"/>
    <ABB_ReadoutLength value="16384"/>
</Context>
...
```

Note that the ABB plugin library `libDabcAbb.so` must be loaded to instantiate the ***abb::Factory*** and apply its classes on the node.

Chapter 10

DABC Programmer Manual: GUI

[programmer/prog-gui.tex]

10.1 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.

10.2 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.

10.2.1 *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.

10.2.2 *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.

10.2.2.1 Record ID=0: Plain

Scalar data item of atomic type

10.2.2.2 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

10.2.2.3 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

10.2.2.4 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

10.2.2.5 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)

10.2.2.6 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

10.2.2.7 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.

10.2.2.8 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.

10.2.2.9 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.

10.2.3 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*.

10.2.4 *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.

10.3 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 implenents the callback functions for resizing and closing.

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

10.3.1 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.

10.3.2 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.

10.4 GUI Panels

Brief description of panels implemented in the *xGUI*.

10.4.1 *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.

10.4.2 *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.

10.4.3 Combined *DABC* and *MBS* launch panel

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

10.4.4 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***).

10.4.5 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.

10.4.6 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.

10.4.7 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:

10.4.7.1 *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.

10.4.7.2 *xRate*

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

10.4.7.3 *xState*

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

10.4.7.4 *xHisto*

Displays a histogram.

10.4.7.5 *xInfo*

Displays a colored text line.

10.4.8 Logging window

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

10.5 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_LAUNCH_DABC : Values of *DABC* launch panel. Saved by button in panel.
Default `DabcLaunch.xml`. Filename in panel itself.

DABC_LAUNCH_MBS : Values of *MBS* launch panel. Saved by button in panel.
  Default `MbsLaunch.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`.

10.5.1 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>
```

10.5.2 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>
```

10.5.3 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>
```

10.5.4 *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>
```

10.5.5 *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>
```

10.6 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).

10.6.1 *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 *infoHandlers* 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.

:

10.6.2 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.

10.6.2.1 *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 (*infoHandlers* 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.

10.6.2.2 *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).

10.6.3 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.

10.6.4 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.

10.7 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`.

10.7.1 Java Interfaces to be implemented by application

10.7.1.1 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!

10.7.1.2 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.

10.7.1.3 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.

10.7.2 Java Interfaces provided by GUI

10.7.2.1 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.

10.7.2.2 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)`

10.7.2.3 Interface *xiDimCommand*

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

10.7.2.4 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.

10.7.2.5 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()`

10.7.3 Other interfaces

10.7.3.1 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);}
```

10.7.4 Example

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

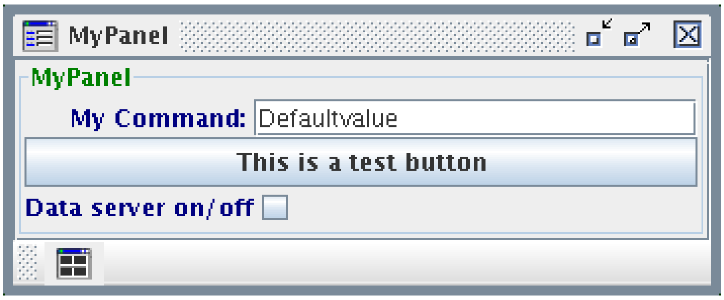
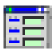


Figure 10.1: Mini panel.

must look 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);
}}
}
```

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()){
```

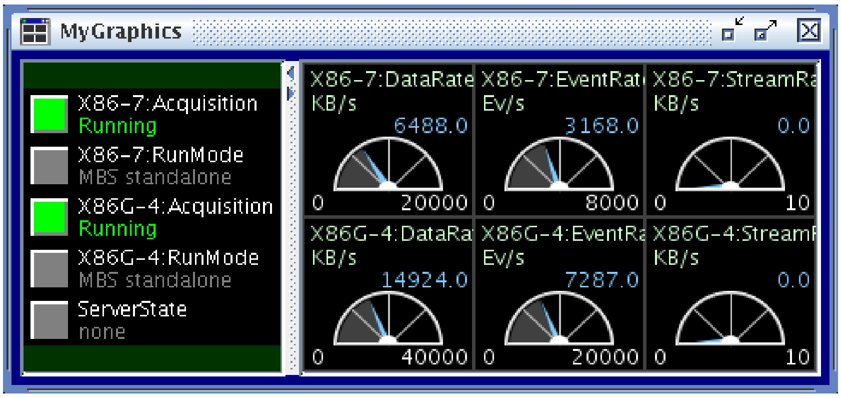


Figure 10.2: Ministates.

```
      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);
}
}
```

10.7.5 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.

Index