The DABC Programmers Manual

J.Adamczewski-Musch, S.Linev, H.G.Essel

Experiment Electronics Department
GSI Darmstadt

January 20, 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	Jörn Adamczewski-Musch	1.0.1	Second scetch

Contents

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 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); and the data import via UDP from a readout controller board (ROC).

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 synchron 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), 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 asynchron 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 commands 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 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 module where they are queued and eventually executed by the module thread (see section 1.2.1). Additionally, the manager has an independent manager thread. This 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 buffer reference exactly one segment. Buffer may have an empty list of references. In addition, buffer can be supplied with 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. Module communicate with memory pool via ***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 port buffers in the processing functions of the module. These are described in section 2.2.1 for ***dabc::ModuleSync*** and section 2.2.2 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).

Transport object may use configuration parameters from port to which it assigned to.

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()*, respectively, 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.

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 existing modules. Thus, the final connecting is done between Configured and Ready.

Ready : The application is fully configured, but not running (threads 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.

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:

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.

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).

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 stat 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 special mode processor can run its own 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 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 availible in classdabc::ModuleAsync. But user **must** avoid any poolling loops, waiting for availible 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 : Moves buffers by reference between two ports, or between a port and the device which owns the transport, respectively. Implementation may be subclass of ***dabc::WorkingProcessor***.

dabc::Device : Is subclass of ***dabc::WorkingProcessor***. Represents generally an module-external data producing or -consuming entity, e.g. a network connection. Each device may create and manage several transport objects. The ***dabc::Transport*** and ***dabc::Device*** base classes have various implementations:

- ***dabc::LocalTransport*** and ***dabc::LocalDevice*** for memory transport within one 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 : Subclass of ***dabc::WorkingProcessor*** (and thus also of ***dabc::CommandReceiver***) and ***dabc::CommandClient***. The manager is one single instance per process; 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 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::StandaloneManager*** provides simple socket controls connection to send remote commands within a multi node cluster. This is used for the standalone Bnet examples without higher level control system.
   - ***dimc::Manager*** Provides DIM [4] as transport layer for inter-module commands. Additionally, parameters may be registered and updated automatically as DIM services.
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 modules and devices.

dabc::Application : Subclass of ***dabc::WorkingProcessor*** (and therefore ***dabc::CommandReceiver***). Interface for the user specific code. Defines the actions on transitions of the finite state machine of the manager. May export parameters for configuration, and may define additional commands.

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 plugin 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 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::FormaterModule : Subclass of ***dabc::ModuleSync***. Framework 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 and one input port for each readout connection (either ***bnet::GeneratorModule*** or connection to a readout transport).
2. 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 event 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 formatter module (or a special event combiner module, resp.). The input data frames are buffered in the Bnet sender module and analyzed which frame is to be sent to what receiver node at what time. 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 combines corresponding event packets (or time frames, resp.) of the different senders.

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 the combined total frame is send to the output port.
2. It has **exactly one** output port. This is connected to the ***bnet::BuilderModule*** implementation (or a user defined builder module, resp.) that performs the actual event tagging and "first level filtering" due to the experimental physics.

bnet::BuilderModule : Subclass of ***dabc::ModuleSync***. Framework 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. ***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 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. ***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 the tested "good" buffers to the output port and discards all other buffers.

2 DABC Programmer Manual: Plugin interfaces

[programmer/prog-plugin.tex]

2.1 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 extensability 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 maintenable 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.

2.2 Modules

2.2.1 ModuleSync

Simple data processing functionality is most easily 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.

1. The constructor of ***dabc::ModuleSync*** subclass creates all ports, may initialize the pool handles, and may define commands and parameters.
2. The virtual *ExecuteCommand()* method of ***dabc::ModuleSync*** subclass may implement the callbacks of the defined commands.
3. The virtual *MainLoop()* method of ***dabc::ModuleSync*** subclass implements the processing job. This method runs in a dedicated working thread.
4. The user code shall not directly access the memory pools to request new buffers. Instead, it can use ***dabc::ModuleSync*** method *TakeBuffer()* with a ***dabc::PoolHandle*** object as argument. This method provides a blocking access to a memory pool, i. e. the *MainLoop()* thread may block here until a buffer is available.
5. The *MainLoop()* code can send and receive buffers from, and to ports by ***dabc::ModuleSync*** methods *Send(port), buffer)*, and *Receive(port), buffer)*, resp. These methods may also block the *MainLoop()* if an output port queue is full, or if no buffer is available at the input port, resp.

2.2.2 ModuleAsync

Alternatively, a subclass of ***dabc::ModuleAsync*** can be implemented. In contrast to ***dabc::ModuleSync***, there is no main loop function which runs in a dedicated, blockable thread, but some event callbacks are implemented that are only executed if a *DABC* event occurs, e. g. a buffer arrives at the input port; a requested buffer is available from a memory pool, etc. Several ***dabc::ModuleAsync*** objects may be assigned to one working thread, so any event callback function must never block the execution

1. The constructor of ***dabc::ModuleAsync*** subclass creates all ports, may initialize the pool handles, and may define commands and parameters.
2. The virtual *ExecuteCommand()* method of ***dabc::ModuleAsync*** subclass may implement the callbacks of the defined commands.
3. For each kind of *DABC* event, virtual methods may be implemented in ***dabc::ModuleAsync*** subclass that are executed by the working thread when the corresponding event occurs for the module, e .g. :

   void ProcessInputEvent(Port port)* : A new buffer was received at a connected port. The ***Port**** argument gives the reference to the port where the buffer was received.

   void ProcessOutputEvent(Port port)* : A buffer in the queue of an output port was taken by the connected transport. The ***Port**** argument gives the reference to the port where the buffer was send.

   void ProcessConnectEvent(Port port)* : A port has been connected to an external ***dabc::Transport***. The ***Port**** argument gives the reference to the port that was connected. This method can be used for initialization actions on connection.

   void ProcessDisconnectEvent(Port port)* : A port has been disconnected from a ***dabc::Transport***. The ***Port**** argument gives the reference to the port that was disconnected. This method can be used for cleanup actions.

   void ProcessPoolEvent(PoolHandle pool)* : A requested buffer is available at a memory pool. The ***PoolHandle**** argument gives the reference to the memory pool handle which fired the event; this is used to actually take the buffer into the module without blocking execution.

   void ProcessTimerEvent(Timer timer)* : A timer has fired an event. The ***Timer**** argument gives the reference to identify which timer reached its timeout.
4. In baseclass ***dabc::ModuleAsync*** all events are dispatched to the callbacks above by method *ProcessUserEvent(ModuleItem item, uint16_t evid)*. This 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 ***dabc::ModuleAsync*** subclass if one wants to treat all events in one method. The ***uint16_t evid*** argument gives the number of the event type. The component that fired the event may be accessed by the ***ModuleItem**** pointer.
5. To avoid blocking the shared working thread, the user must always check if a resource (e.g. a port, a memory pool) would block before any calls (e.g. *Send()*, *TakeBuffer()*) are invoked on it. All callbacks must **return** in the "I would block" case; on the next time the callback is executed by the framework the user must check the situation again and react appropriately. This might require an own bookkeeping of available resources.

2.2.3 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()*, other virtual methods (e.g. *DoBuildEvent()*, *TestBuffer()*) may be implemented that are implicitly called by the superclass *MainLoop()*.
2. The special base classes may provide additional methods to be used for data processing.

2.3 Device and transport

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

1. The ***dabc::Device*** subclass constructor may create pool handles and may define commands and parameters.
2. Method *int CreateTransport(Command* cmd, Port* port)* of the ***dabc::Device*** implementation creates the appropriate ***dabc::Transport*** instance and connects it to the given *Port* port*. It is invoked by the framework when this device is connected to a module port, which is usually specified in the application via calls of ***dabc::Manager*** methods. The optional argument *Command* cmd* may pass further parameters from the application to the new transport, encapsulated in a command object.
3. The special transport class ***dabc::DataTransport*** (inherits from ***dabc::Transport***) should be used as base class to implement a user defined transport (e. g. read from a user defined hardware board). This class already provides queues for (optionally) input and output buffers and a data backpressure mechanism. The user subclass must implement virtual methods to perform the data transport which are called implicitely from the framework at the right time:
   unsigned Read_Size() : Should specify the required buffer size to be read from the device, e. g. the DMA memory size. Framework will then allocate a *Buffer* of appropriate size for the *Read_** functions
   unsigned Read_Start(Buffer buf)* : transport gets a *Buffer* buf* to be filled in the read. Actual filling may be done asynchronously to the calling thread, e. g. by DMA. Thus this function may initiate the asynchronous read here and return immeadeately before the read is complete. For synchronous filling of the buffer, this function should do nothing.
   unsigned Read_Complete(Buffer buf)* : Called before the framework transfers the *Buffer* buf* to the connected port. The transport should no sooner return this function than the reading of this buffer (e. g. by DMA) is complete. In case of synchronous reading, the buffer filling is also initiated here. For asynchronous reading, buffer filling has been initiated before in funcunsigned Read_Start(Buffer* buf) and this function just waits for a "buffer complete" state from the filling device.
   void Read_CallBack(unsigned compl_res = DataInput::di_Ok) : this method MUST be called by transport when *Read_Start()* returns *di_CallBack*. It is only way to "restart" event loop in the transport (Sergei, please explain more!)
   double Read_Timeout() : Defines timeout for operation in ms? TO BE EXPLAINED
   bool WriteBuffer(Buffer buf)* : The arriving *Buffer* buf* is synchronously written to the device.
   void ProcessPoolChanged(MemoryPool pool)* : Called when a funcMemoryPool changes the buffer set up. This allows to work directly on the buffers of a memory pool, e. g. to initialize all buffers for DMA when memory pool is created.
4. The virtual *ExecuteCommand()* method of ***dabc::Device*** subclass may implement the callbacks of the defined commands.

2.4 Factories

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

1. The user must define a ***dabc::Factory*** subclass to add own classes to the system.
2. Each factory is instantiated as static singleton when loading the library that defines it.
3. Factories are registered and kept in the global manager. The access to the factories' functionality is done via methods of the manager that scans all known factories to produce the requested object

class.

4. The framework provides several factories for predefined implementations (e. g. ***bnet::SenderModule***, ***verbs::Device***)
5. The user factory may implement such methods:
   Module CreateModule(const char* classname, const char* modulename, Command* cmd)* : Instantiate a ***dabc::Module*** of class *classname*. The object name of the module is taken from *modulename* argument. Optional argument *Command* cmd* may pass further creation parameters from the application to the new module, encapsulated in a command object.
   Device CreateDevice(const char* classname, const char* devname, Command* cmd)* : Instantiate a ***dabc::Device*** of class *classname*. The object name of the device is taken from *devname* argument. Optional argument *Command* cmd* may pass further creation parameters from the application to the new device, encapsulated in a command object.
   Application CreateApplication(const char* classname, Command* cmd)* : Instantiate a ***dabc::Application*** of class *classname*. Optional argument *Command* cmd* may pass further creation parameters from the set-up to the new application, encapsulated in a command object.
   Note that the factory methods for ***dabc::Transport*** objects belong to the corresponding ***dabc::Device*** implementation (see section 2.3)).

2.5 The DABC Application

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

1. The manager has exactly one application object. The user must implement a ***dabc::Application*** subclass.
2. On startup time, the ***dabc::Application*** is instantiated by means of a factory method *CreateApplication(const char* classname, dabc::Command* cmd)*. The text argument *classname* specifies which application subclass is created; this name is taken from a setup parameter, i. e. it may be read from an XML setup file. The framework will search all registered factories for the method which can fullfill to create an application of that name. So the user must provide a ***dabc::Factory*** that defines such method for his/her application implementation.
3. The application may register parameters that define the application's configuration. These parameters can be set at runtime from the configuration and controls system.
4. The user may implement virtual methods *UserConfigure()* , *UserEnable()*, *UserBeforeStart()*, *UserAfterStop()*, *UserHalt()* in his/her ***dabc::Application*** subclass. These methods are executed by the framework state machine before or after the corresponding state transitions to do additional application specific configuration, run control, and clean-up actions. Note: all generic state machine actions (e.g. cleanup of modules and devices, starting and stopping the working processors) are already handled by the framework at the right time and need not to be invoked explicitely here.
5. 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 these and implementing additional virtual methods (e. g. *CreateReadout()*).

3 DABC Programmer Manual: Manager interface

[programmer/prog-manager.tex]

3.1 Introduction

this chapter covers the manager API description

3.2 Framework interface

here description of mostly used methods to be called from application, modules, etc.

3.3 Control system plug-in

3.3.1 Interfaces

3.3.1.1 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* . It 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 3.3.1.2) 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.

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

bool Subscribe(dabc::Parameter* par, int remnode, const char* remname) :

bool Unsubscribe(dabc::Parameter* par) :

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 **??**).

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.

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 (???).

const char* GetNodeName(int num) :

int NodeId() const :

bool IsNodeActive(int num) :

bool IsMainManager() :

bool CanSendCmdToManager(const char* mgrname) :

bool SendOverCommandChannel(const char* managername, const char* cmddata) :

void CommandRegistration(dabc::Module* m, dabc::CommandDefinition* def, bool reg) :

int ExecuteCommand(dabc::Command* cmd) :

In addition to the virtual methods to be implemented in the manager subclass, there is a number of ***dabc::Manager*** baseclass 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`.

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.

3.3.1.2 ManagerFactory

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.

3.3.2 Default implementation for DIM

The *DABC* default controls and configuration system is based on the DIM library [4] and is marked by namespace ***dimc::*** (for "DIM Control"). It features the DIM publisher-subscriber mechanism with *DABC* process records and uses state machine module and XML parser as provided by the *DABC* core system. The main classes are described in the following:

dabc::Manager : implements the control system interface of ***dabc::Manager*** as described above. Main methods are:

1. Implements SendOverCommandChannel() to transport core commands over the dabc::xd DIM layer.
2. Implements ParameterEvent(): Interface method to register and update dabc parameter to infospace and DIM. If parameter is created, its contstructor calls this method. Depending on parameter type it is registered using Add...Record() methods of Registry. Vice versa, on parameter deletion it is removed via RemoveRecord(). If module parameter changes value, this is forwarded using Update...Record().
3. Implements CommandRegistration() : Interface method to register or unregister a module command as a DIM command and corresponding command descriptor service. Uses internally dabc::xd::Registry::RegisterModuleCommand() and ...UnRegisterModuleCommand() methods.
4. Implements ModuleException() to react to a dabc::exception if thrown in a module. So far, we only do error output here. TODO: concept of exception handling in modules, threads and control system!

5. Implements methods for the daq cluster management: GetNodeName() , NodeId() to map the unique node id number as used in the core with an arbitrary node name of the configuration/control system; NodeId() delivers the id of the current node. NumNodes() returns the number of all configured cluster nodes as known from the set up. These methods are just forwards to corresponding interface methods of Node.
6. Implements Subscribe() and Unsubscribe() for parameters. These methods are just forwards to corresponding interface methods Registry::SubscribeParameter() , and ..:UnsubscribeParameter() , resp.
7. Implements InvokeStateTransition(). This method is forwarded to corresponding interface method of Node.

4 DABC Programmer Manual: Parameters and Setup

[programmer/prog-setup.tex]

4.1 Parameter class

Configuration and status information of objects can be represented by ***Parameter*** class. Any objects, derived from ***WorkingProcessor*** class, can has a list of parameters, assigned to it - for instance, ***Application***, ***Device***, ***Module***, ***Port*** classes.

There are number of class ***WorkingProcessor*** methods to create parameter objects of different kinds and access their values. Full list one can see in 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

As one can see, to store boolean parameter, string is used. If value of string is "true" (in lower case), boolean value of such parameter recognized as true, otherwise false.

For any type of parameter GetParStr/SetParStr methods can be used.

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

4.2 Use parameter for control

The main advantage to use parameter objects is that parameter values can be observed and changed using controlling system.

At any time, when parameter value changed by program via SetPar... methods, control system informed and represents such change in appropriate GUI element. At the same time, if user modifies parameter value in the GUI, value of parameter object will be changed and correspondent parent object (module, device) get callback via virtual method ParameterChanged(). Implementing correct reaction on this call, one can provide possibility to reconfigure/adjust running code on the fly.

Parameter value may be fixed via Parameter::SetFixed() method. This blocks possibility to change parameter value from both program and user side. Only when fixed flag set again to false, parameter value can be changed.

Not all parameters objects should be visible to control system. There is so-called visibility level of parameter, which is assigned to parameter when its instance is created. Only if visibility level smaller than current level (configured in Manager::ParsVisibility()), parameter will be "seen" by control system. Such level is configured via WorkingProcessor::SetParDflts() function before parameters objects are created.

4.3 Example of parameters usage

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

Using control system, value of 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")));
      }
```

From the performance reasons one should avoid usage of parameter getter/setter methods (like GetParBool() or SetParInt()) inside loop, which executed many times. Main aim of parameter object is to provide connection to control system and in other situations normal class members should be used.

4.4 Configuration parameters

Another use of parameter object is its usage for objects configuration. When one creates object like module or device, one often need to deliver one or several configurations parameters to constructor such as required number of input ports or server socket port number.

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

GetCfgStr string

GetCfgDouble double
GetCfgInt integer
GetCfgBool boolean

All these methods has following arguments: name of configuration parameter, default value of configuration parameter [optional] and pointer on ***Command*** object [optional]. Lets 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 period of timer is set via configuration parameter "Period". How its value will be defined?

First of all, will be checked if parameter with given name exists in the command. If not, appropriate entry in configuration file will be searched. If configuration file also does not contains such parameter, default value will be used.

4.5 Configuration file example

Configuration file is xml file in dabc-specific format, which contains value of some or all configuration parameters of the system.

Lets consider simple but functional example of configuration file:

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

This is an example of xml file for mbs generator, which produces mbs events and provides them to mbs transport server. To run that example, just "run.sh test.xml" should be executed in the shell. Other application (dabc or Go4) can connect to that server and read generated mbs events.

4.6 Basic syntax

DABC configuration file should always contain <dabc> as root node. Inside <dabc> node one or several <Context> nodes should exists. Each <Context> node represents independent application context, which runs as independent executable. Optionally <dabc> node can has <Variables> and <Defaults> nodes, which are described further.

4.7 Context

<Context> node can has two optional attributes:

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

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

4.8 Run arguments

Usually <Context> node has <Run> subnode, where user defines different parameters, relevant for running dabc application:

lib library name, which should be loaded. Several libraries names can be specified.
func function name which should be called to create modules. It is alternative to writing subclass of ***Application*** and instantiating it.
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 debugger. Value should be like "gdb -x run.txt –args", where file run.txt should contain commands "r bt q".
workdir directory where dabc application 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 control
DIM_DNS_PORT port number of dim dns server, used by dim control

4.9 Variables

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

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

It is allowed to combine variable with text or other variable, but neither arithmetic nor string operations are supported.

Using variables, one can modify example in following way:

```
<?xml version="1.0"?>
<dabc version="1">
  <Variables>
    <myname value="Generator"/>
    <myport value="6010"/>
  </Variables>
  <Context name="Mgr${myname}">
    <Run>
      <lib value="libDabcMbs.so"/>
      <func value="StartMbsGenerator"/>
    </Run>
    <Module name="${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 defined by 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 environment variable can also be used as variable to set parameter values.

4.10 Default values

There are situations, when one need to set same value to several similar parameters, for instance same output queue length for all output ports in the module. One possible way is to use variables (as described before) and set parameter value via variable. Disadvantage of such approach that one should expand xml files and in case of big number of ports xml file will be very long and unreadable.

Another possibility to set several parameters at once - create <dabc/Defaults> node and specify cast rule, using "*" or "?" symbols like that:

```
<?xml version="1.0"?>
```

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

  <Context name="Mgr${myname}">
    <Run>
      <lib value="libDabcMbs.so"/>
      <func value="StartMbsGenerator"/>
    </Run>
    <Module name="${myname}">
       <SubeventSize value="32"/>
       <Port name="Output">
          <MbsServerPort value="${myport}"/>
       </Port>
    </Module>
  </Context>
  <Defualts>
    <Module name="*">
       <Port name="Output*">
          <OutputQueueSize value="5"/>
       </Port>
    </Module>
  </Defualts>
</dabc>
```

In this case for all ports, which names are started with string "Output" from any module, output queue length will be 5.

In form, as it is specified in example, such multicast rule will be applied for all contexts from configuration file means by such rule we set output queue length for all modules on all nodes. This allow us to create compact xml files for big multi-nodes configuration.

4.11 Usage commands for configuration

Lets consider possibility to configure module, using ***Command*** class.

This is required when object (like mnodule) should be created with fixed parameters disregard of values specified in config file.

In our example one can modify StartMbsGenerator() function in following way:

```
extern "C" void StartMbsGenerator()
{
    dabc::Command* cmd = new dabc::CmdCreateModule("mbs::GeneratorModule", "Genera
    cmd->SetInt("SubeventSize", 128);
    if (!dabc::mgr()->Execute(cmd)) {
       EOUT(("Cannot create generator module"));
```

```
        exit(1);
    }

    ...
}
```

Here one add additional arguments to CmdCreateModule, which set mbs subevent size to 128. After that any modification of <SubeventSize> entry in config file take no effects.

5 DABC Programmer Manual: GUI

[programmer/prog-gui.tex]

5.1 GUI Guide lines

The *DABC* GUI is written in Java `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

5.2 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.

5.2.1 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 servernamespace. For *DABC* and *MBS* this servernamespace is set to DABC.

5.2.2 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 STATUS, 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.

5.2.2.1 Record ID=0: Plain

Scalar data item of atomic type

5.2.2.2 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

5.2.2.3 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

5.2.2.4 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

5.2.2.5 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)

5.2.2.6 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

5.2.2.7 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.

5.2.2.8 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. 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.

5.2.3 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*.

5.2.4 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.

5.3 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.

5.3.1 Prompter panels

Prompter panels can be implemented subclassing class ***xPanelPrompt***. The layout is in rows. A row can be a prompter line (label and ***JTextField*** field), a text button, or a 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.

5.3.2 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.

5.4 GUI Panels

Brief description of panels implemented in the *xGUI*.

5.4.1 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.

5.4.2 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.

5.4.3 Combined DABC and MBS launch panel

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

5.4.4 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.

5.4.5 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***).

5.4.6 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.

5.4.7 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:

5.4.7.1 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.

5.4.7.2 xState

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

5.4.7.3 xHisto

Displays a histogram.

5.4.7.4 xInfo

Displays a colored text line.

5.4.8 Logging window

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

5.5 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`.
DABC_LAUNCH_MBS : Values of *MBS* launch panel. Saved by button in panel.
Default `MbsLaunch.xml`.
DABC_RECORD_ATTRIBUTES : Attributes of rate and histogram panels. 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`.

5.5.1 Records

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>
```

5.5.2 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>
```

5.5.3 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>
```

5.5.4 DABC launch panel values

File `DabcLaunch.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<DabcLaunch>
```

```
<Labels
DabcMaster="DABC Master"
DabcName="DABC Name"
DabcUserPath="DABC user path"
DabcSystemPath="DABC system path"
DabcSetup="DABC setup file"
DabcScript="DABC Script"
DabcServers="%Number of needed DIM servers%"
/>
<Fields
DabcMaster="lxg0523.gsi.de"
DabcName="Controller:41"
DabcUserPath="/misc/goofy/dabc/work"
DabcSystemPath="/misc/goofy/sniff/dabc"
DabcSetup="SetupMbs.xml"
DabcScript="ps"
DabcServers="5"
/>
</DabcLaunch>
```

5.5.5 MBS launch panel

File `MbsLaunch.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<MbsLaunch>
<Labels
MbsMaster="MBS Master"
MbsUserPath="MBS User path"
MbsSystemPath="MBS system path"
MbsScript="MBS Script"
MbsCommand="Script command"
MbsServers="%Number of needed DIM servers%"
/>
<Fields
MbsMaster="x86g-4"
MbsUserPath="v50/x86/newmbs"
MbsSystemPath="/daq/usr/goofy/mbswork/v51"
MbsScript="script/remote_exe.sc"
MbsCommand="m_rising v50/x86/newmbs . x86g-4 x86-7"
MbsServers="3"
/>
</MbsLaunch>
```

5.6 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.
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`.

5.6.1 Java Interfaces to be implemented by application

5.6.1.1 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.

- `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!

5.6.1.2 public 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.

5.6.1.3 public 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.

TODO: instead of *getName* the name could given as argument directly in *setInfoHandler*.

5.6.2 Java Interfaces provided by GUI

5.6.2.1 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.

5.6.2.2 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)`

5.6.2.3 Interface xiDimCommand

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

5.6.2.4 Interface xiDimParameter

- `double getDoubleValue()`
- `float getFloatValue()`
- `int getIntValue()`
- `long getLongValue()`
- `xiParser getParserInfo()`
- `String getValue()`
- `xRecordMeter getMeter()`
- `xRecordState getState()`
- `xRecordInfo getInfo()`
- `xiParser getParserInfo()`
- `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.

5.6.2.5 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()

5.6.3 Other interfaces

5.6.3.1 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.

Exampele:

```
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);}
```

5.7 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).

5.7.1 *xDimBrowser*

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

xDimBrowser(...) : Constructor. Arguments: references to the graphics panels ***xPanelMeter***, ***xPanelState***, ***xPanelInfo*** and ***xPanelHisto***. There are protected functions to get then the references to these panels.

protected initServices(String wildcard) : Get list of available services from DIM name server DIM_DNS_NODE. Create vectors of alphabetically ordered parameters (***xDimParameter***) and commands (***xDimCommand***) and their interfaces, respectively. The references of the graphics panels are passed to the parameter objects.

addInfoHandler(xiDimParameter p, xiUserInfoHandler ih) : Interface function to add an additional info handler to a parameter. The *infoHandler* function of this handler is called at the end of the *infoHandler* function of ***xDimParameter***.

removeInfoHandler(xiDimParameter p, xiUserInfoHandler ih) : Interface function to remove an info handler added before.

protected Vector<xDimParameter> getParameterList() :

protected Vector<xDimCommand> getCommandList() :

Vector<xiDimParameter> getParameters() : From outside one gets only references to the interfaces.

Vector<xiDimCommand> getCommands() : From outside one gets only references to the interfaces.

protected releaseServices(boolean cleanup) : Removes all external handlers of the parameters. Sets all parameters to inactive. This means that in the *infoHandler*s no more graphical activity is performed. It `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.

:

5.7.2 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.

5.7.2.1 *xPanelParameter*

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

1. Get reference to list of parameters (from browser).
2. Set in all parameters the table index to -1 (*infoHandler*s will no longer update table fields.

3. Scan through all parameters and check if any quality is still -1 which would mean that the type is undefined. That is repeated two times with 2 seconds delay to give the DIM servers the chance to update all parameters. If still any quality is -1 this is an error.
4. Restore record attributes of meters and histograms from XML file.
5. Cleanup graphics panels.
6. Create new table.
7. Add parameters to table by calling function *xDimParameter.addRow*. This function also creates graphical presentations of the parameters (e.g. ***xMeter***) and add them to the appropriate graphics panels (e.g. ***xPanelMeter***) if needed.
8. Builds list of command descriptors (***xXmlParser***).
9. Add table to its panel.

5.7.2.2 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).

5.7.3 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 and call *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 *setCommandDescriptors*. The descriptors are provided as parameters. The descriptor list is generated by ***xPanelParameter***.
7. Update all graphics panels (*updateAll*). All graphics items had been added; Panels are reformatted.
8. Call *init* and *setDimServices* of all application panels. Pass ***xiUserCommand*** object from first application panel object to ***xPanelCommand***.
9. Create the internal frames to display all panels which shall be visible.

5.7.4 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 *browser.releaseServices*.
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 *browser.initServices*.
5. Create ***xPanelParameter***.
6. Create ***xPanelCommand***.
7. Update all graphics panels (*updateAll*). All graphics items had been added; Panels are reformatted.
8. Call *setDimServices* of all application panels. Pass ***xiUserCommand*** object from first application panel object to ***xPanelCommand***.
9. Call browser *enableServices* function.
10. Call *setCommandDescriptors*.
11. Update the internal frames of parameters and commands.

6 DABC Programmer Manual: Example MBS

[programmer/prog-exa-mbs.tex]

6.1 Overview

MBS (Multi Branch System) is 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 standard DABC distribution. All sources can be found in $DABCSYS/plugin/mbs directory. All these sources compiled into library libDabcMbs.so, which is placed in $DABCSYS/lib.

6.2 Events iterators

MBS defines own event/subevent structures. To access such events data, number of structures are defined in "mbs/LmdTypeDefs.h" and "mbs/MbsTypeDefs.h". In first file structure ***mbs::Header*** is defined, which is just container for arbitrary raw data. Such container is used to store/read data from LMD files. In "mbs/MbsTypeDefs.h" file 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 buffer (type mbt_MbsEvents), where several subsequent MBS events (there is no buffer header in front!). To iterate over all events in such buffer class ***mbs::ReadIterator*** was designed (defined in "mbs/Iterator.h"). It provides possibility to iterate (access) over all events in buffer in 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*** is developed to fill number of MBS events into ***dabc::Buffer***. Way to use this iterator illustrated by 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;
   }
}
```

6.3 File I/O

LMD file format used in MBS. There is class ***mbs::LmdFile***, which provide C++ interface for reading/writing such lmd file.

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

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

Class ***mbs::LmdOutput*** inherits ***dabc::DataOutput*** and provides possibility to save MBS events, placed in ***dabc::Buffer*** objects, in LMD file. In addition to ***mbs::LmdFile*** functionality, it allows to create multiple files when file size limit is exceeded. 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 provide them over input ports into 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, that user can instantiate these classes using plugin meachanism of DABC.

Here is an example, how output file for 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);
...
```

Here one first creates module and than configure (via command) type of output transport and its parameters.

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

```
...
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 create module with 3 inputs and than for each input port lmd file transport is created.

6.4 Socket classes

All communication with MBS servers performed via socket. DABC has number of class for socket handling, included in base package (libDabcBase.so). Main idea of these classes is to handle socket operations (creation, connection, sending, receiving and error handling) in form of event processing.

Class ***dabc::SocketThread*** organises event loop, produced by sockets. Each system socket assigned with instance of ***dabc::SocketProcessor*** class. Processing of socket events 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 ***dabc::SocketThread*** together with other kind of processors like module classes, but not wise-versa (one cannot use socket processors inside other thread types). Therefore, it is possible to run module and all its socket transports in one single thread if socket thread for such module created in advance (see MBS generator example).

6.5 Server transport

Class ***mbs::ServerTransport*** was developed to provide MBS servers functionality in DABC. Using this class one can emulate work of MBS transport server and MBS stream server. This is also good example for usage of ***dabc::SocketProcessor*** classes.

Implementation of ***mbs::ServerTransport*** based on generic class ***dabc::Transport*** and internally uses two kinds of sockets: socket for handling connection and I/O socket for sending data.

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 xml file like here:

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

Than, to create such transport, following code should be executed:

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

Another possibility to specify these parameters - use dabc::CmdCreateTransport:

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

6.6 Client transport

Class ***mbs::ClientTransport*** allows connect DABC with MBS. For the moment 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, following pease 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::DefualtServerPort(mbs::StreamServer) + 5);
  dabc::mgr()->Execute(cmd);
  ...
```

6.7 Events generator

Class ***mbs::GeneratorModule*** is an example of simple module design, which just fills buffers with random MBS events and provides them to the output. Schematically implementation of module can be shown as:

```
#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 module constructor pool handle create to declare that module requires memory pool with 10 buffers of defined size. Buffer size here taken as configuration parameter. When output port is created, pool handle and default queue size is specified.

The only virtual method which should be implemented for generator module is *ProcessOutputEvent*. This function call every time when free space in port output queue is appears. It means, when module starts, it immediately gets N times (size of output queue, here 5) this call while there is N empty places in the queue. The only action here is take new buffer from memory pool, fill it with rundom events and send to output.

Real class ***mbs::GeneratorModule*** included in 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 is also *StartMbsGenerator* function to instantiate and run generator module. It demonstrates, how thread of type ***dabc::SocketThread*** can be created and used by both module and transport object.

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

To run generator module with all default parameters, simple xml file should be used:

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

One can also specify all module and transport parameters directly:

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

There is example $DABCSYS/application/mbs/GeneratorTest.xml file, which demonstrate usage of generator module.

6.8 MBS event building

Class ***mbs::CombinerModule*** provides possibility 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

Module can has two outputs: for file storage (port name FileOutput) and for providing data further over MBS server (port name ServerOutput).

There is function *StartMbsCombiner*, which initialized combiner module and starts data taking. Example configuration file $DABCSYS/applications/mbs/Combiner.xml shows how to configure combiner from three MBS transport servers:

```
<?xml version="1.0"?>
<dabc version="1">
  <Context host="localhost" name="Combiner">
    <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>
```

7 DABC Programmer Manual: Example Bnet

[programmer/prog-exa-bnet.tex]

7.1 Overview

In complex experiments there are lot of front-end systems, which runs in parallel. They takes data and marks them with trigger information or just with time stamps. To be able analyze such data, all portions belonging to the same event (or time stamp), should be combined in one processing unit. Such task usually called event building.

To support event building functionality in DABC, special sub-framework called BNET (building network) was introduced. It's main aim - simplify implementation of experiment-specific event building, distributed over several nodes.

Typical event building network contains several **readout** nodes, which are connected to some number of data sources each. In readout node one reads data from data sources and combines together data parts, which logically belongs together (so-called subevent building). In case of trigerred system one combines together data, which has same trigger number. In case of time-stamped data one combines together data which belongs to the same time interval. While one have several readout nodes, to build complete event, one should bring together all data, which belongs to same trigger (or time interval) into the same **builder** node. One typically has not single but several builder nodes, therefore full connectivity between all readouts and all builder nodes should be introduced. Once all subevents delivered to the same builder node, one can build complete event and store it on the file or tape.

BNET framework defines required functional units (modules), which should be used in application, and provides implementation of several important components. BNET also defines topology of these functional units, which can be customized up to definite level.

7.2 Controller application

Event building task usually distributed over several nodes, which should be controlled. Therefore in BNET all nodes classified by their functionality on two kinds: controller and workers. Workers perform all kinds of data transport and analysis codes while controller configures and steers all workers.

Functionality of controller is implemented in ***bnet::ClusterApplication*** class. Via controlling interface cluster controller distributes commands, coming from operator, to all workers, observes status of all workers and reconfigures them automatically when errors are detected.

Functionality of ***bnet::ClusterApplication*** based on state-machine logic of DABC. It means, that all actions performed during state changing command, implemented in virtual *DoStateTransition* method. State transition on cluster controller requires, that appropriate state transition performed on all worker nodes.

Technically it means, that command, which is executed on cluster controller only than can be completed, when state transition commands on all workers are completed. This is implemented with use 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 in real work.

7.3 Worker application

Basic functionality of worker implemented in ***bnet::WorkerApplication*** class. Its main aim - by commands from cluster controller instantiate all necessary modules, configure and connect them together.

Main functionality of ***bnet::WorkerApplication*** class is implemented in virtual *CreateAppModules* method, which is called during transition from Halted to Configured state. In this method all local modules are instantiated and configured. Some of these modules should be experiment specific, therefore class ***bnet::WorkerApplication*** provide number of virtual methods, where experiment-specific components should be created:

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

Class ***bnet::WorkerApplication*** has following 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

To implement experiment-specific BNET application, user should first of all create its own application class, which inherits from ***bnet::WorkerApplication*** and implement mentioned above virtual methods. If required, one can also add more parameters.

Class ***bnet::WorkerApplication*** has also number of 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.

7.4 Combiner module

Combiner module merges together several data sources and produces subevents packets. Subevent here means that data from all sources, which corresponds to the same event (or time frame) should be placed in same ***dabc::Buffer*** object. This buffer object should has header with unique identifier of type `bnet::EventId` - 64-bit unsigned integer.

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

Subevent identifier should has increasing number without gap. When no data for specific identifier is available, empty buffer with no data and correct header should be delivered to output.

There is ***bnet::CombinerModule*** class, which provides prototype of combiner module. It uses following parameters:

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. While class ***bnet::WorkerApplication*** already has parameter with such name, its value will be directly used for model configuration.

When implementing experiment-specific combiner class, one should either derive it's from ***bnet::CombinerModule*** class or start its implementation from scratch. One can add more experiment-specific parameters to the module.

7.5 Network topology

Topology (connectivity) of network between workers defined by parameters IsSender and IsReceiver of ***bnet::WorkerApplication*** class. By setting these parameters one can configure role of each worker:

- 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 from both parameters has true value.

Cluster controller during configuration establish connections between workers so, that each sender module connected with all receiver modules. This guarantees, that each receiever can get necessary data from all data sources and perform event building.

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

Subevents buffers, produced by combiner module, will be delivered to sender module. Based on event identifier, buffer will be send to specific receiver, where event with such id will be build. For now simple round-robin schedule is used by BNET, but in next DABC versions one or several others data transfer schedules will be implemented. Idea of BNET framework is that such improvments should be possible without changing of user code.

7.6 Event builder module

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

To build experiment-specific builder module, one can derive it from ***bnet::BuilderModule*** class or start it from scratch. Event builder module has one input and one output. Over input module gets N buffers with subevents for same event identifier. Over output module should deliver buffer with build events.

When user inherits its builder module from ***bnet::BuilderModule*** class, it is enough to implement virtual *DoBuildEvent* method, which gets as argument list from N buffers with subevents. Format of output buffer is completely user-defined. One allowed to fill several events into the same output buffer if necessary.

7.7 Filter module

This is optional component of BNET for situation, when build events should be filtered before they are stored. To implement such filter, one can derived it from ***bnet::FilterModule*** and reimplement virtual method *TestBuffer*. As alternative, filterring can be implemented directly in the event builder module.

7.8 BNET test application

This is test application, which is designed for testing different asspects of BNET without necessity to have real data sources. Complete source code and configurations examples can be found in $DABCSYS/applications/bnet-test directory.

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, configuration with 4 worker nodes with sender and receiver functionality each showin in SetupBnet.xml example:

```
<?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 same configurations parameters, which can be found in <Defaults> section. In section <Context name="Worker*"> one sees, that IsGenerator, IsSender and IsReceiver parameters are set to true. This defines so-called all-to-all topology, when each node can communicates with all other nodes including itself. Parameter NumReadouts=4 means that there are 4 inputs on each combiner resulting in 16 data sources over complete system.

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

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

7.9 BNET for MBS application

This is ready-ro-use implementation of distributed event building for MBS. 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 cluster application parameter NumEventsCombine defines how many events should be bundled togther in one buffer. It is crutial that transport buffer size should be big enough for such number of subevents. Cluster parameter NumEventsCombine during initialisation copied to each worker parameter CfgEventsCombine, which finally used in ***bnet::MbsCombinerModule*** during initialisation:

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

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

Class ***bnet::MbsBuilderModule*** builds mbs events from delivered by receiver module buffers. It also use parameter CfgEventsCombine from application to be sure how many real MBS events contained in input buffers.

Application class ***bnet::MbsWorkerApplication*** implements several methods to correctly instantiate combiner and builder modules. It also implement virtual method *CreateReadout*, where approprite input transport for combiner module should be created. In case of mbs there are three possibilities:

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

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

There is example "SetupBnetMbs.xml" file, which contains running example for MBS event building of with 2 readout nodes, connected with 2 generators each and 2 event builder nodes. Configuration file can be customised via variables definition 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 node where cluster controller will runs, node1 and node3 used as readout nodes, node2 and node4 as builder nodes. On all four worker nodes one mbs generator application will be started. To run application, just type `run.sh SetupBnetMbs.xml`.

8 DABC Programmer Manual: Example ROC

[programmer/prog-exa-roc.tex]

8.1 Overview

CBM readout controller (ROC) is FPGA-based board, which is aimed to readout nXYTER chip and transport data over Ethernet to PC. Software package ROClib provides basic functionality to readout data from ROC.

To support usage of ROC in DABC, following classes were designed:

- ***roc::Device*** device class, wrapper for SysCoreController
- ***roc::Transport*** access to SysCoreBoard functionality
- ***roc::CombinerModule*** module to combine data from several ROCs in single output
- ***roc::CalibrationModule*** module to calibrate time scale in ROC data
- ***roc::ReadoutApplication*** application to perform readout from ROC boards
- ***roc::Factory*** factory class to organize plugin

8.2 Device and transport

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

Each instance of ***roc::Transport*** has a pointer to ***SysCoreBoard*** object, over which data taking from specific ROC is performed. Implementation of class ***roc::Transport*** based on ***dabc::DataTransport*** class. Class ***dabc::DataTransport*** uses event loop mechanism and does not requires explicit thread. This feature allows to run several instances of such transports in the same thread. In ROC case all ***roc::Transport*** instances uses thread of ***roc::Device***. Lets try to describe how class ***roc::Transport*** is working.

In the beginning of event loop (when module starts) *roc::Transport::StartTransport* is called, which is used to call *SysCoreBoard::startDaq* to start data taking. After that event loop consist from subsequent calls of *Read_Size*, *Read_Start* and *Read_Complete* functions, derived from ***dabc::DataTransport*** class.

Aim of *Read_Size* method is to define size of next buffer, required for data reading. In case of ***roc::Transport*** this size is fixed and defined by configuration parameter:

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

When system deliver buffer of requested size, *Read_Start* function is called to start reading of that buffer from data source. ***SysCoreBoard*** internally has its own buffer, therefore call *SysCoreBoard::requestData* either inform that required number of messages already received or one should wait. Waiting in thread logic will mean that thread should be blocked and cannot be used by other transport. Therefore, another approach is used - ROClib will provide call back of virtual *SysCoreControl::DataCallBack* method when required amount of data is there. Implementation of method is looks like:

```
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;
}
```

In case when data already exists in internal buffer of ***SysCoreBoard*** object, dabc::DataInput::di_Ok is returned and than immediately *Read_Complete* will be called, which finally fill output 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;
}
```

Returned from function *Read_Start* value dabc::DataInput::di_CallBack indicates that event loop should be suspended. When required amount of data received by ROClib, it produces call of *SysCoreControl::DataCallBack* method, which is reimplemented in ***roc::Device*** class and calls following method of ***roc::Transport***:

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

While required amount of data is received, one only retrieves it with the same *Read_Complete* method and reactivate event loop by calling *Read_CallBack*.

8.3 Combiner module

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

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 16386]
- NumOutputs - number of outputs [default 2]

As output MBS events are provided. Each MBS event contain ROC messages between two sync markers. For each ROC separate MBS event allocated. Field iSubcrate contains ROC id.

8.4 Calibration module

Class ***roc::CalibrationModule*** perform calibration of time scale for all ROCs and merging all messages in single data stream. As output, MBS event with single subevent is produced.

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 16386]
- NumOutputs - number of outputs [default 2]

8.5 Readout application

The main aim of ***roc::ReadoutApplication*** class is configure and run application, which combines readout of data from several ROCs, store it into lmd file and optionally create mbs stream server to observe data from remote Go4 GUI. 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 combined data
- CalibrFile - name of lmd file to store calibrated data
- MbsFileSizeLimit - maximum size of each file, in Mb

Three calibration modes are supported:

- DoCalibr=0 - Only CombinerModule is instantiated, which produces kind of ROC "raw" data
- DoCalibr=1 - Both CombinerModule and CalibrationModule are instantiated
- DoCalibr=2 - Only CalibrationModule, used to convert raw data from lmd files into 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. Last mode is special case, when RawFile specifies not output but input file for the calibration module.

8.6 Factory

Factory class ***roc::Factory*** implements several methods to create ROC-specific application, device and modules.

8.7 Source and compilation

Source code for all classes can be found in $DABCSYS/plugins/roc directory. Compiled library libDabcKnut.so should be found in $DABCSYS/lib directory. If one need to modify some code in this library, one should copy sources in user directory and call "make" in this directory. In this case library can be found in directory like $ARCH/lib, where $ARCH is current CPU architecture (for instance, i686).

8.8 Running ROC application

To run readout application, approprite xml configuration file should be created. There are two examples of configuration files in $DABCSYS/applications/roc. In Readout.xml one founds example of 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>
```

While this is single-node application, to run it, dabc_run executable can be used: `dabc_run Readout.xml`. dabc_run executable will load specified libraries, create application, configure it and switch system in running mode.

In Calibr.xml shown special case of configuration to convert raw data into calibrated files 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>
```

9 DABC Programmer Manual: Example PCI

[programmer/prog-exa-pci.tex]

9.1 Example PCI event building

The Active Buffer Board (*Active Buffer Board*) as example of a PCI device in *DABC*. The *Active Buffer Board*is developed by Institut f. Technische Informatik at Uni Mannheim.

9.2 DABC class description and functionality

9.2.1 dabc::PCIBoardDevice

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

- Keeps vector of ***mprace::DMABuffers*** (mapped user memory taken from regular *DABC*memory pool. Method *GetDMABuffer* of ***dabc::Buffer**** delivers the mapped DMA buffer which corresponds to a given memory pool buffer; these are associated by the buffer id number used as index of the DMA buffer vector. The (re-)mapping is done on the fly if needed, calling *MapDMABuffer* of ***dabc::MemoryPool****.
- Method *ReadPCI* of ***dabc::Buffer**** implements reading one buffer from PCI, with regard to parameters *fReadAddress*, *fReadBAR*, *fReadLength*, resp. Performs DMA into a regular pool buffer (user memory) which is mapped using *GetDMABuffer* of ***dabc::Buffer****. Currently DMA is still initiated by the host PC, i.e. ABB will start transfer by the call of *ReadDMA()* that blocks until DMA is complete. For asynchronous DMA (double buffering of ***dabc::DataTransport***) following methods *ReadPCIStart()* and *ReadPCIComplete()* are provided. Device subclass may re-implement this method, adding board specific stuff.
- Method *ReadPCIStart* of ***dabc::Buffer**** implements start reading one buffer from PCI, with regard to parameters *fReadAddress*, *fReadBAR*, *fReadLength*, resp. Performs DMA into a regular pool buffer (user memory) which is mapped using *GetDMABuffer* of ***dabc::Buffer****. This call does not block until DMA is complete, but returns immediately! Device subclass may re-implement this method, adding board specific stuff.
- Method *ReadPCIComplete* of ***dabc::Buffer**** does a blocking wait until the current DMA buffer is finished with read from PCI. DMA has to be started using *ReadPCIStart()* before; method returns error if there was no DMA going on. Device subclass may re-implement this method, adding board specific stuff.
- Implements *CreateTransport()* as framework factory method to create a ***dabc::PCITransport*** and assign own working thread for each transport.
- Implements several specific commands which are handled in *ExecuteCommand(Command*)*. Currently these commands define DMA address regions for reading and writing.
- Implements *DoDeviceCleanup()* for cleaning the mapped dma buffers.

9.2.2 dabc::PCITransport

Subclass of ***dabc::DataTransport***, providing the connection to a module's input port.

- Transport queues and backpressure mechanism is provided by base class. The IO actions are executed in the thread assigned to the base class ***dabc::WorkingProcessor***.
- Implements *ProcessPoolChanged(**dabc::MemoryPool***)*. This method is called whenever the memory pool associated with the transport changes, e.g. by recreation, expansion, etc. Uses *PCIBoardDevice::MapDMABuffers(**dabc::MemoryPool***)* to rebuild the indexed ***mprace::DMABuffers*** for each buffer of the pool.
- Implements *ReadBegin()*, returning the size of the next buffer that should be read from board. Uses current values as set in the ***dabc::PCIBoardDevice***
- Implements *ReadPrepare(**dabc::Buffer***)* which initiates the actual reading of the buffer passed as argument. Calls *PCIBoardDevice::ReadPCIStart(**dabc::Buffer***)*. Note that on return of *ReadPrepare()*, the previously filled DMA buffer is already forwarded to the connected port (which wakes up thread of the readout module), i.e. ***dabc::DataTransport*** implicitly provides a double-buffering mechanism here.
- Implements *ReadComplete(**dabc::Buffer***)* which returns when the actual buffer has finished being filled from DMA read operation. Calls *PCIBoardDevice::ReadPCIComplete(**dabc::Buffer***)*.

9.2.3 dabc::AbbDevice

Subclass of ***dabc::PCIBoardDevice***.

- Re-Implements *ReadPCI()*: NOTE: To work with the bnet test example, this will copy an event header of the Bnet format (i.e. incrementing event count and unique id) into each output Buffer after *dabc::PCIBoardDevice::ReadPCI()* is complete. Later CBM event format must be written by ABB and implemented in bnet for ABB.
- Implements *DoDeviceCleanup()* to reset event counter for the bnet test example.

9.2.4 dabc::AbbReadoutModule

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

- Creates the memory pool which is used for DMA buffers in the ***PCIBoardDevice***; this pool is propagated to the device via the ***PCITransport*** when module is connected, since device will use the pool associated with the connection port.
- Module runs either in standalone mode (one input port, no output) for testing; or in regular mode (one input port, one output port)
- *ProcessUserEvent()* defines the module action for any dabc events, e.g. input port has new buffer. In standalone mode, received buffer is just released. In regular mode, buffer is send to the output port.

9.2.5 dabc::AbbWriterModule

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

- Creates the memory pool which is used for DMA buffers in the ***PCIBoardDevice***; this pool is propagated to the device via the ***PCITransport*** when module is connected, since device will use the pool associated with the connection port.
- Module runs either in standalone mode (one output port, no input) for testing; or in regular mode (one input port, one output port)
- *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.

9.2.6 dabc::AbbFactory

Subclass of ***dabc::ApplicationFactory***. Generic plugin to utilize ABB classes in a set up:

- Implements *CreateDevice()* for the ***AbbDevice***. Third argument of this factory method is ***dabc::command***, containing initial setup parameters of the device.
- Implements *CreateModule()* for the ***AbbReadoutModule*** and the ***AbbWriterModule***. Third argument of this factory method is ***dabc::command***, containing initial setup parameters of the module.
- Factory is created automatically as static (singleton) instance on loading the `libDabcAbb.so`.

See Class Diagram of dabc PCI classes.

9.3 ABB classes with bnet test example

The ***TestWorkerPlugin*** of the bnet/test example was modified to optionally integrate ABB.

9.3.1 bnet::TestWorkerPlugin

Subclass of ***bnet::WorkerPlugin***. This example has possibililty to utilize ABB classes in a bnet-set up.

- Registers parameters to configure the ABB. Note: The Parameter names are defined as string constants in `AbbFactory.h`. These same names are used by ***AbbFactory*** methods to get the appropriate command parameters!
- Implements *CreateReadout()* with optional ***AbbDevice***. May either use the ***AbbDevice***, or still use the ***TestGeneratorModule*** of the example. This is switched by configuration parameter via the *ReadoutPar()* value (either "ABB", or anything else).Connects ***AbbDevice*** directly with the input port of the combiner module; we do not use ***dabc::AbbReadoutModule*** here (this is applied for standalone example `abb_test` only).

References

[1] CBM collaboration, “CBM Experiment: Technical Status Report”, Januar 2005

[2] CMS collaboration, http://cmsinfo.cern.ch/outreach/, “CMS Outreach”, CERN 2006

[3] The Experimental Physics and Industrial Control System website, http://www.aps.anl.gov/epics/index.php, Argonne National Laboratory 2006

[4] C. Gaspar et al., "DIM - Distributed Information Management System" , http://dim.web.cern.ch/dim/, CERN May 2006

[5] Y. Liu and P. Sinha, “A Survey Of Generic Architectures For Dependable Systems”, IEEE Canadian Review, Spring 2003

[6] The National Instruments Labview web site, http://www.ni.com/labview/, National Instruments Corporation 2006

[7] L. Orsini and J. Gutleber, “The XDAQ Wiki Main Page” http://xdaqwiki.cern.ch/index.php, CERN 2006

[8] L. Orsini and J. Gutleber, “I2O Messaging” http://xdaqwiki.cern.ch/index.php/I2O_Messaging , CERN 2006

[9] L. Orsini and J. Gutleber, “XDAQ Monitor application” http://xdaqwiki.cern.ch/index.php/Monitor_CGI_interface , CERN 2005

[10] L. Orsini and J. Gutleber, http://xdaqwiki.cern.ch/index.php/Configuration_schema “XDAQ XML configuration schema”, CERN 2006

[11] D. Stenberg et al., The curl and libcurl web site, http://curl.haxx.se/, HAXX HB 2006

[12] SystemC website, http://www.systemc.org/

[13] The W3C Consortium, “SOAP Version 1.2 Part 1: Messaging Framework”, http://www.w3.org/TR/soap12-part1, W3C Recommendation, 24 June 2003

[14] K. Whisnant, R.K. Iyer, Z. Kalbarczyk, and P. Jones, “The Effects of an ARMOR-based SIFT Environment on the Performance and Dependability of User Applications”, University of Illinois, 2006

[15] The Wikipedia, “Finite State Machine”, http://en.wikipedia.org/wiki/State_machine, Wikipedia 2006

Index