The DABC Programmers Manual

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

Experiment Electronics Department
GSI Darmstadt

January 15, 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 . . . **5**

1.1 Introduction . . . 5

1.2 Role and functionality of the objects . . . 5

1.2.1 Modules . . . 5

1.2.1.1 Class ***dabc::ModuleSync*** . . . 5

1.2.1.2 Class ***dabc::ModuleAsync*** . . . 5

1.2.2 Commands . . . 6

1.2.3 Parameters . . . 6

1.2.4 Manager . . . 6

1.2.5 Memory and buffers . . . 6

1.2.6 Ports . . . 7

1.2.7 Transport . . . 7

1.2.8 Device . . . 7

1.2.9 Application . . . 8

1.3 Controls and configuration . . . 8

1.3.1 Finite state machine . . . 8

1.3.2 Commands . . . 9

1.3.3 Parameters and configuration . . . 9

1.3.4 Parameters and monitoring . . . 9

1.3.5 Parameter change . . . 9

1.4 Package and library organisation . . . 9

1.4.1 Core system . . . 10

1.4.2 Control and configuration system . . . 10

1.4.3 Plugin packages . . . 10

1.4.3.1 Bnet package . . . 10

1.4.3.2 Transport packages . . . 10

1.4.4 Application packages . . . 10

1.4.5 Distribution contents . . . 11

1.5 Main Classes . . . 11

1.5.1 Core system . . . 11

1.5.2 BNET classes . . . 13

2 DABC Programmer Manual: Plugin interfaces **17**
2.1 Introduction 17
2.2 Modules 17
2.2.1 ModuleSync 17
2.2.2 Special modules 17
2.3 Device and transport 18
2.4 Factories 18
2.5 The DABC Application 18

3 DABC Programmer Manual: Parameters and Setup **21**
3.1 Setting up system 21
3.1.1 Parameter class 21
3.1.2 Use parameter for control 21
3.1.3 Example of parameters usage 22
3.1.4 Configuration parameters 22
3.1.5 Configuration file example 23
3.1.6 Basic syntax 24
3.1.7 Context 24
3.1.8 Run arguments 24
3.1.9 Variables 24
3.1.10 Default values 25
3.1.11 Usage commands for configuration 26

4 DABC Programmer Manual: GUI **29**
4.1 GUI Guide lines 29
4.2 GUI Panels 29
4.2.1 *DABC* launch panel 29
4.2.2 *MBS* launch panel 29
4.2.3 Combined *DABC* and *MBS* launch panel 29
4.2.4 Command panel 29
4.2.5 Parameter selection panel 29
4.2.6 Monitoring panels 29
4.2.7 Parameter table 29
4.2.8 Logging window 29
4.3 GUI save/restore setups 29
4.4 Application specific GUI plug-in 29

4.4.1 Application interfaces . . . 29
4.4.2 Java Interfaces to be implemented by application . . . 30
4.4.2.1 Interface xiUserPanel . . . 30
4.4.2.2 public interface xiUserCommand . . . 30
4.4.2.3 public interface xiUserInfoHandler . . . 30
4.4.3 Java Interfaces provided by xgui . . . 30
4.4.3.1 Interface xiDesktop . . . 30
4.4.3.2 Interface xiDimBrowser . . . 30
4.4.3.3 Interface xiDimCommand . . . 30
4.4.3.4 Interface xiDimParameter . . . 31
4.4.3.5 Interface xiParser . . . 31

5 DABC Programmer Manual: Example MBS 33

5.1 Overview . . . 33
5.2 Events iterators . . . 33
5.3 File I/O . . . 34
5.4 Socket classes . . . 35
5.5 Server transport . . . 36
5.6 Client transport . . . 36
5.7 Events generator example . . . 37
5.8 Local event building . . . 37

6 DABC Programmer Manual: Example Bnet 39

6.1 Example network event building . . . 39

7 DABC Programmer Manual: Example ROC 41

7.1 Overview . . . 41
7.2 Device and transport . . . 41
7.3 Combiner module . . . 42
7.4 Calibration module . . . 43
7.5 Readout application . . . 43
7.6 Factory . . . 43
7.7 Source and compilation . . . 44
7.8 Running ROC application . . . 44

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 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 implicitely ("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 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 in a subclass implementation of ***dabc::Command***.

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.

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.

The manager is an **object manager** that owns and keeps all registered basic objects into a folder structure. Each object has a direct backpointer to the manager.

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 may execute manager commands for configuration, and all high priority commands like state machine switching that must be executed immediately even if the mainloop of the modules is blocked.

1.2.5 Memory and buffers

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

A buffer contains a list of references to scattered memory fragments, or it may have an empty list of references. 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. Each memory pool is identified by a unique ***dabc::PoolHandle*** object which is used e.g. by the module to communicate with the memory pool.

A new buffer may be requested from a memory pool by size. Depending on the module type and mode, this request may block until an appropriate buffer is available; or it may return an error value if it can not be fulfilled, resp. 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 ports, output ports, 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: For ***dabc::ModuleSync***, function *MainLoop()* may wait for buffers to arrive at one or several input ports, calling the respective *Recv()* methods; this call may block. Processed buffers can be put into output ports by method *Send()*; these calls may also block the mainloop, if the port queue is full since subsequent components do not read the outgoing buffers fast enough (data backpressure mechanism).

In contrast to this, ***dabc::ModuleAsync*** has event callbacks that are executed by the framework when a buffer arrives at an input port (*processInput()*), or when an output port is ready to accept a buffer (*processOutput()*). Alternatively, *ProcessUserEvent()* may be implemented to handle events of all input and output ports in one function.

1.2.7 Transport

Outside the modules the ports are connected to ***dabc::Transport*** objects. On each node, a transport may either transfer buffers between the ports of different modules (local data transport), or it may connect the module port to a data source or sink (e. g. file i/o, network connection, hardware readout).

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

1.2.8 Device

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

A ***dabc::Device*** instance usually represents an IO 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 system process) itself. 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 part of the manager, i.e. the manager does not necessarily check if a state transition is allowed. Instead, the controls framework provides a state machine implementation.

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:

Configure : between Halted and Configured. Core framework automatically creates standard modules and devices. The application plug-in creates application specific objects, requests memory pools, and establishes all local port connections. Core framework instantiates the requested memory pools

Enable : 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.

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

Stop : 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 !

Halt : 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). Execution of these commands is independent of the state machine transitions.

1.3.3 Parameters and configuration

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 external configuration system may set the parameters from a database, or 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 Configure 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.

The current parameters may be stored back to the data base of the configuration system.

1.3.4 Parameters and monitoring

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.

1.3.5 Parameter change

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).

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***.

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 transport, or socket; 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.
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* collection (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 his 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. The use case of a parameter depends on the lifetime of its parent object: The configuration parameters should be created from the application plug-in which is persistent during the process lifetime. Transient monitoring parameters may be created from a device (optionally also persistent, but not yet existing at process startup when configuration database is read!), from a module, or from a transport implementation (if this also inherits from ***dabc::WorkingProcessor***). The latter parameters do only exist if the application is at least in Configured state, and they will be destroyed with their parents when switching to Halted state. Currently supported parameter types are:

- ***dabc::IntParameter*** - simple integer value
- ***dabc::DoubleParameter*** - simple double value
- ***dabc::StrParameter*** - simple test string
- ***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 may run either one working processor, or serve several working processors in parallel. ***dabc::WorkingProcessor*** is a subclass of ***dabc::CommandReceiver***, i.e. a working processor may receive and execute commands in his 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 runs in a dedicated working thread. Method *TakeBuffer()* provides blocking access to a memory pool. The optionally blocking methods *Send(port), buffer)* and *Receive(port), buffer)* are used from the *MainLoop()* code to send (or receive) buffers over (or from) a port of the ***dabc::ModuleSync***.

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. 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 (might require own bookkeeping of available resources).

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 plug-in by methods of ***dabc::Manager*** specifying 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
- ***dabc::VerbsTransport*** and ***dabc::VerbsDevice*** for InfiniBand **verbs** connection
- ***dabc::PCITransport*** and ***dabc::PCIBoardDevice*** for DMA I/O from PCI or PCIe boards

dabc::Manager : Subclass of ***dabc::WorkingProcessor*** (and by that 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.
   - ***dabc::xd::Manager*** for use in the XDAQ framework. Provides DIM as transport layer for inter-module commands. Additionally, parameters may be registered and updated automatically as DIM services. This manager is used in the XDAQ Bnet example from the dabc::xd::Node. [[DabcXdaq][More details of this implementation are described here.]]
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::ClusterPlugin : Subclass of ***dabc::ApplicationPlugin*** 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::WorkerPlugin : Subclass of ***dabc::ApplicationPlugin*** 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. Currently the user has to implement the sending of the tagged events to the output port explicitly in his subclass. **Should be done automatically in** *MainLoop()* **later?**

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

The data processing functionality is usually implemented by subclassing the ***dabc::ModuleSync*** base class.

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.
4. The user code shall not directly access the memory pools to request new buffers. Instead, it can use methods of ***dabc::Module*** with a ***dabc::PoolHandle*** object as argument. These methods may block the *MainLoop()*.
5. The user code can send and receive buffers from and to ports by methods of ***dabc::ModuleSync***. These methods may block the *MainLoop()*.

2.2.2 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. The virtual *ExecuteCommand()* method of ***dabc::Device*** subclass may implement the callbacks of the defined commands.
3. Factory method *CreateTransport()* of ***dabc::Device*** subclass defines which transport instance is created by the framework whenever this device shall be connected to the port of a module.
4. The virtual *Send()* and *Recv()* methods of the ***dabc::Transport*** subclass must implement the actual transport from and to a connected port, respectively.
5. To implement a simple user defined transport (e.g. read from a special socket protocol), there is another base class ***dabc::DataTransport*** (subclass of***dabc::Transport***). This class already provides queues for (optionally) input and output buffers and a data backpressure mechanism. Instead of *Send* and *Recv()*, the user subclass must implement special virtual methods, e.g. *ReadBegin()*, *ReadComplete(buffer)* to request the next buffer of a given size from the base class, and to peform filling this buffer from the associated user device, respectively. These methods are called implicitely from the framework at the right time.

2.4 Factories

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

1. All factories are registered and kept in the global manager. The access to the factories' functionality is done via methods of the manager.
2. All factories are instantiated as static singletons when loading the libraries that defines them. The user need not to create them explicitely.
3. The ***dabc::Factory*** subclasses must implement methods *CreateModule(classname)*, and *CreateDevice(classname)* , to instantiate modules or devices, respectively. The user must subclass the ***dabc::Factory*** to add own classes to the system. **Note:** the factory method for transport objects is not in this factory, but in the corresponding ***dabc::Device*** subclass.
4. The manager can keep more than one factory and scans all factories to produce the requested object class.
5. The framework provides several factories for predefined implementations (e.g. ***bnet::SenderModule***, ***dabc::VerbsDevice***).

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 provide a ***dabc::Application*** subclass.
2. The ***dabc::Application*** is instantiated and registered on startup time by a global *InitPlugins()* function. This function is declared and executed in the framework, but defined in the user library and thus knows the user specific plug-in classes. This trick is necessary to decouple the executable main function (e.g. χ*DAQ*executive) from the user specific part.
3. The user may implement virtual methods *UserConfigure()* , *UserEnable()*, *UserBeforeStart()*, *User-*

AfterStop(), *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.

4. The application may register parameters that define the application's configuration. These parameters can be set at runtime from the configuration and controls system.
5. The methods of the application should use application factories to create modules and devices by name string. However, for convenience the user may implement factory methods *CreateModule()* and *CreateDevice()* straight in his/her ***dabc::Application*** subclass.
6. 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: Parameters and Setup

[programmer/prog-setup.tex]

3.1 Setting up system

3.1.1 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.

3.1.2 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.

3.1.3 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.

3.1.4 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.

3.1.5 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.

3.1.6 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.

3.1.7 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.

3.1.8 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

3.1.9 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.

3.1.10 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.

3.1.11 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.

4 DABC Programmer Manual: GUI

[programmer/prog-gui.tex]

4.1 GUI Guide lines

The *DABC*GUI is written in Java. 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.

4.2 GUI Panels

4.2.1 *DABC* launch panel

4.2.2 *MBS* launch panel

4.2.3 Combined *DABC* and *MBS* launch panel

4.2.4 Command panel

4.2.5 Parameter selection panel

4.2.6 Monitoring panels

4.2.7 Parameter table

4.2.8 Logging window

4.3 GUI save/restore setups

4.4 Application specific GUI plug-in

Besides the generic part of the GUI it might be useful to have specific user panels as well, integrated in the generic GUI. This is done by implementation of a xPanelPrompt derived class. The class name can be passed as argument to the java command starting the GUI. The class must implement some interfaces.

4.4.1 Application interfaces

A user may implement these interfaces in his own menues. He can connect his own call back functions to parameters, and a command function to be called when a command shall be executed. He may create his own panels for display using the graphical primitives like rate meters.

4.4.2 Java Interfaces to be implemented by application

4.4.2.1 Interface xiUserPanel

`public abstract void init(xiDesktop desktop, ActionListener actionlistener);`
Called by xgui after instantiation. The desktop can be used to add frames (see below).
`public String getHeader();` Must return a header/name text after instantiation.
`public String getToolTip();` Must return a tooltip text after instantiation.
`public ImageIcon getIcon();` Must return an icon after instantiation.
`public xiUserCommand getUserCommand();` Must return an object implementing xiUserCommand, or null. See below.
`public void setDimServices(xiDimBrowser browser);` 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 can be registered for each selected parameter. Then the infoHandler method is called for each parameter update.
`public void releaseDimServices();` All local references to commands or parameters must be cleared!

4.4.2.2 public interface xiUserCommand

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

4.4.2.3 public interface xiUserInfoHandler

```
public void infoHandler(xiDimParameter p);
```

4.4.3 Java Interfaces provided by xgui

4.4.3.1 Interface xiDesktop

```
public void addDesktop(JInternalFrame frame, String name);
```

4.4.3.2 Interface xiDimBrowser

```
public xiDimParameter[] getParameters();
public xiDimCommand[] getCommands();
public void setInfoHandler(xiDimParameter parameter,
                           xiUserInfoHandler infohandler);
public void sleep(int s);
```

4.4.3.3 Interface xiDimCommand

```
public void exec(String command);
public xiParser getParserInfo();
```

4.4.3.4 Interface xiDimParameter

```
public xRecordMeter getMeter();
public xRecordState getState();
public xRecordInfo getInfo();
public xiParser getParserInfo();
```

4.4.3.5 Interface xiParser

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

5 DABC Programmer Manual: Example MBS

[programmer/prog-exa-mbs.tex]

5.1 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.

5.2 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;
   }
}
```

5.3 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 required for cases like socket or InfiniBand transports, but maybe 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 only 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. Following parameters are supported:

- 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.

5.4 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 operation (creation, connection, sending, receiving and error handling) in form of event processing.

Class ***dabc::SocketThread*** organises event loop, produced by sockets, assigned with ***dabc::SocketProcessor***. Processing of these events done by virtual methods of class ***dabc::SocketProcessor***, which has several subclasses for different kinds of sockets:

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

Usage of socket processor classes requires that thread, to which such processors assigned should be of type ***dabc::SocketThread***. At the same time normal processors like ***dabc::ModuleAsync*** can work together with ***dabc::SocketThread***. 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:

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

5.5 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 of 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:

- MbsServerKind - kind of mbs server ("Transport" or "Stream")
- MbsServerPort - port number for socket connection

In previous section there is example of creation server transport with default parameters. To instantiate MBS server tranport with user-defined parameters, following code should be used:

```
...
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::TransportServer) + 5
dabc::mgr()->Execute(cmd);
...
```

5.6 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:

- MbsServerKind - kind of mbs server ("Transport" or "Stream") to connect to
- MbsServerName - host name where mbs server runs
- MbsServerPort - 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::TransportServer)
dabc::mgr()->Execute(cmd);
...
```

5.7 Events generator example

5.8 Local event building

6 DABC Programmer Manual: Example Bnet

[programmer/prog-exa-bnet.tex]

6.1 Example network event building

7 DABC Programmer Manual: Example ROC

[programmer/prog-exa-roc.tex]

7.1 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

7.2 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::Compltete BufferReading()
{
   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*.

7.3 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.

7.4 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]

7.5 Readout application

The main aim of ***roc::ReadoutApplication*** class is configure and run application, which combines read-out 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.

7.6 Factory

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

7.7 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).

7.8 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>
```