The DABC Framework Interface to Readout Hardware

Jörn Adamczewski-Musch, Hans G. Essel, Sergei Linev

Abstract–**The Data Acquisition Backbone Core (DABC) is a new GSI software framework to run a data acquisition with distributed event building on high performance Linux clusters. Experiment specific front-end electronics is to be integrated to the software by means of hardware interface plug-ins like Device and Transport classes. DABC offers elaborate mechanisms for multiprocessing, buffer management, and dataflow throttling. These are transparently available for all implemented plug-ins. Device plug-ins can link a DABC node to remote readout hardware via network connections like Ethernet. Other Device plug-ins can communicate on the Linux device driver level with custom boards directly inserted at the node.**

Besides delivering the data input, a DABC Device can also provide control access to the connected hardware. This functionality can be used for setting up, or monitoring the front-ends from the application via DABC parameters and commands.

An implementation example is a multi purpose PCI Express Optical Receiver (PEXOR) board developed at GSI. This board features an FPGA and 4 optical links and may be used for various front-ends, depending on the FPGA programming. A kernel driver and the DABC Device plug-in for this board have been developed and tested. They are described here with some performance benchmark results.

As another example, DABC is applied for data taking during test beam times of the Compressed Baryonic Matter (CBM) experiment from 2008 to 2010. Here the front-end readout controller boards (ROC) were integrated to the DABC hardware interface, both for an UDP based Ethernet protocol, and for optical connections to a custom PCIe board.

***Index Terms*–Data acquisition, Device driver, Object oriented programming, Software architecture**

I. Introduction

THE Data Acquisition Backbone Core (DABC) is a general purpose data acquisition (DAQ) software framework designed for distributed data processing on Linux clusters with fast switching networks [1],[2]. Unlike other experiment dedicated DAQ systems [3]-[5], DABC is a framework library which provides a base for various different DAQ systems. DABC offers the software infrastructure. The experiment specific software components to handle read out electronics and process the data are implemented in software plug-ins. The development of DABC is furthermore motivated by future experiments' requirements for a "triggerless" DAQ. Here time-stamped data from the frontend hardware have to be transported over a "builder network" to first level event selection nodes. Each selection node may need the full detector information to filter out physics events from the data stream. Hence DABC is intended to maximize network bandwidth and processing performance by means of traffic shaping and thread optimization. The first DABC release was published under the GPL in 2009 [6]. DABC is based on plain C++ and supports Ethernet (IP sockets) and InfiniBand (verbs) networks by default. Moreover, DABC implements file formats and data connections to the established DAQ framework MBS [7]. So it is easy to integrate legacy readout hardware and analysis software.

Manuscript received June 14, 2010, revised April 12, 2011

Jörn Adamczewski-Musch, Hans G. Essel , and Sergei Linev are with Experiment Electronics Department, GSI Helmholtzentrum für Schwerionenforschung, Planckstr 1, 64291 Darmstadt, Germany. The corresponding author is Jörn Adamczewski-Musch (phone: +49-6159-711337, fax: +49-6159-712986, e-mail: J.Adamczewski@gsi.de).

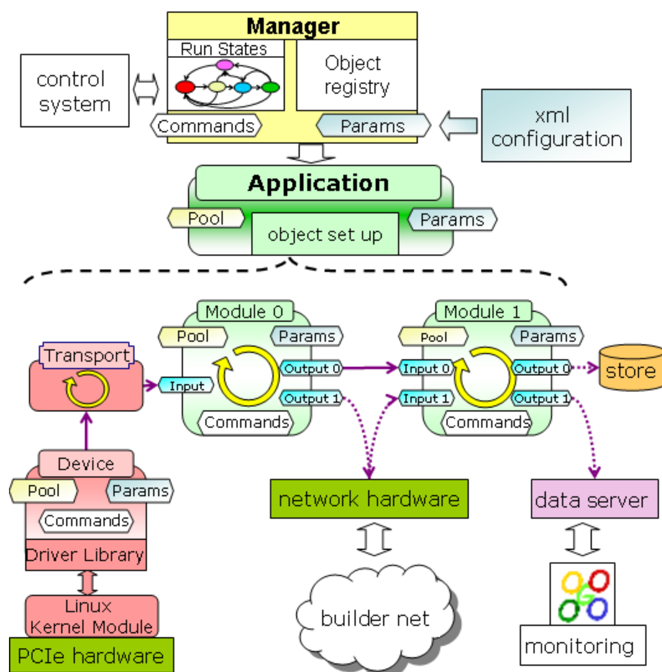

Fig. 1. Example of software objects in one DABC node: The *Manager* is the configuration and control system instance. The *Application* plug-in defines the user-specific set up. Data processing is done in *Modules* that are connected with "Input" and "Output" *Ports*. A *Device* object is responsible for several *Transports* of the same kind and can implement specific hardware I/O. All objects share memory pools for data *Buffers* ("Pool") and can read and export control parameters ("Params").

The DABC framework features a runtime environment with transparent mechanisms for multithreading, memory management, command and event dispatching, and configuration and controls. This was recently presented in detail [8]. Concerning the software architecture, DABC applies common design patterns [9]. For example, user-specific parts of the DAQ system are implemented as plug-ins, i.e. specializations of base classes for dedicated purposes. These plug-ins are integrated to the framework by means of the *Abstract Factory* pattern, with user defined *Factory Methods* for initialization [9].

In the following we will describe the integration of custom readout hardware in DABC. Let's consider a single node of the DAQ cluster. A typical situation is shown in Fig.1. The *Manager* is the central instance. It registers all objects, defines the run state with a finite state machine, evaluates the XML configuration file, and is the gateway to the control system implementation. The *Manager* realizes the *Singleton* design pattern [9]. This ensures that there is only one *Manager* object, and that it is accessible from everywhere in the framework. The user specific set-up is configured by the *Application* plug-in as registered to the *Manager* when loading the user application library. It implements the initialization method, defining which kind of processing objects are created, and how these are connected.

The DABC concept of data flow consists of *Module* and *Device* software objects which are linked at configurable *Port* connections by buffering *Transport* instances. The *Modules* run the actual data processing, either in separate threads, or in a thread shared with other processing entities. The *Devices* are responsible for the *Transport* connections to or from a *Module*, or between *Modules*.

Usually any hardware device is implemented in DABC by a *Device* class with a corresponding *Transport* class. The DABC framework provides these implementations for standard hardware, e.g. Network Interface Controllers, which are encapsulated in libraries like *OFED verbs* [10], or the Linux *sockets*. For custom readout hardware, developed for a specific experiment's set-up, these classes must be implemented by the user. Additionally, specific processing of data from this hardware is to be put into at least one user *Module*. In the following, we will show the recommended class interfaces for such purpose.

II. Hardware Interface Classes

In Fig. 2 a simplified Unified Modeling Language (UML) class diagram illustrates the relations between the key classes of the hardware interface. Class inheritance is shown by green solid arrows from subclass to base class; associations between classes by black dashed arrows with object multiplicities (e.g. one *Device* can handle many *Transports*, "1 to *").

All base classes of the DABC plug-in interface, such as *Device*, *DataTransport*, and *Module*, are subclasses of *WorkingProcessor* which is assigned to a *WorkingThread*. This class represents a single thread of the operating system. *WorkingProcessor* actions to run in the thread can be triggered by an external event (e.g. "new buffer in queue"), including command execution. This mechanism, however, is transparent to the user, since the hardware interface classes just require a couple of functions (class methods) to be defined in its user implementations. The DABC framework will take care that these functions are executed at the "right time" within the preconfigured multithreading environment. Additionally, any *WorkingProcessor* instance has service functions to define and access *Parameter* containers which can be set from the DABC configuration system at runtime. These are to be used for the specific set up of the user hardware and readout functions. The details of the interface classes are discussed in the next sections.

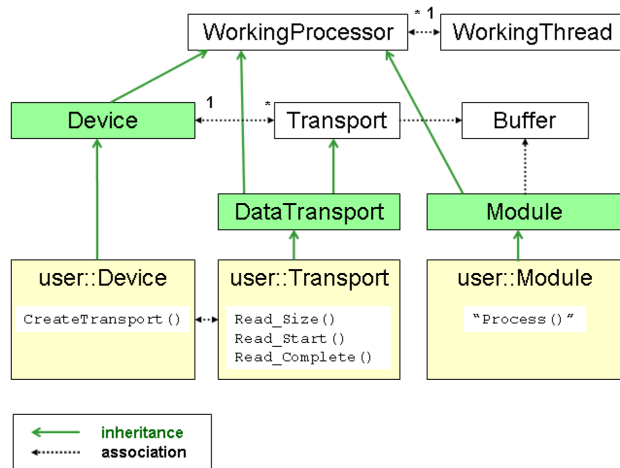

Fig. 2. Simplified UML diagram of the DABC hardware interface classes; the relevant virtual methods are shown in the user classes. The *Device* base class can manage several *Transport* instances. Each of these is a user implementation of the *DataTransport* interface which actually fills data from the hardware into *Buffer* objects provided by the framework. The *Module* connected to the *DataTransport* will receive the *Buffer* contents and can process them. *Device*, *DataTransport*, and *Module* are all subclasses of *WorkingProcessor*, hence their actions can be run by independent threads (*WorkingThread* class), or they can partially share one thread.

A. The Device Class

The *Device* class contains generic framework functionality to manage data transports to or from any hardware device. It keeps and owns a collection of *Transport* objects that are created for such kind of *Device.* The actual *Device* implementation for specific hardware is done in a subclass, e.g. *user::Device* (see Fig. 2). This may use existing driver libraries, or may be based on low level system calls to a Linux kernel module. The *user::Device* must provide a factory method *CreateTransport()* to define the corresponding *user::Transport* implementation. It is called from the framework when a *Module* is connected to this *Device.* Therefore the *Device* class also realizes the *Abstract Factory* design pattern [9] for the *Transport*.

B. The DataTransport Class

While the *Device* object represents the complete hardware entity, the actual data transfer to a *Module* must be implemented in a *Transport* subclass. Because the abstract base class *Transport* is a rather low level interface, DABC provides the advanced class *DataTransport* as recommended base class for all custom *Transport* implementations. It defines a simple interface of virtual methods for reading data:

1. *unsigned Read_Size()* is called before payload data are read from the device. It must return the size of the next expected data packet. This size may be read here from a header in advance of the actual data stream, or from a dedicated device register. If such is not possible, the maximum expected data size is to be returned. From this information the framework will provide the *Buffer* to fill.
2. *Read_Start(Buffer*)* is called at the begin of the data transfer from the device and may optionally initiate the transfer (e.g. a DMA to the buffer). The target buffer to fill is passed as an argument. This function must not block, since it is meant for asynchronous transfers: The previously queued *Buffers* can be processed during the active DMA of the current *Buffer*. If implemented for synchronous transfers, however, this function does nothing but return an "OK" value.
3. *Read_Complete(Buffer*)* is called when the transfer shall be finished. The target buffer to fill is passed as an argument. This function must block until the transfer from the device to the buffer is complete, since the framework will use this buffer further in the connected *Module* right after the function returns. For synchronous transfers, all functionality is implemented here. Depending on the return value, the framework may retry the transfer, or skip the buffer. This can be used to read out data in a polling mode, or to handle device errors, respectively.

C. The Module Class

Device and *Transport* implementations handle the data transfer from the readout hardware into the framework buffers. Processing the data within these buffers should be done in subsequent *Module* implementations. For a testing node, at least one "Readout Module" (*user::Module* in Fig. 2) should be provided that can connect to the *Transport* and check the data received from the hardware. It may also save the data for further processing, or export data for online monitoring to a data server. Here the *Module* could utilize the supported MBS data formats and socket connections [7]. With these it is easy to apply existing monitoring tools like Go4 [11] for analysis and visualization.

III. Configuration and Controls

A. The Node Manager

Each node of the DAQ cluster is controlled by the DABC *Manager* singleton (see Fig. 1). It registers and owns all software objects. It defines and operates a finite state machine for the run states of the node, e.g. "Halted", "Configured", "Running", "Stopped". Furthermore, the *Manager* is the gateway to the control and configuration system that can switch the run states, can dispatch external commands, and can set and monitor parameter values.

B. The Application Plug-in

The actual set up of each node in the DAQ cluster is done by writing an *Application* subclass as a plug-in for the *Manager*. At least one virtual method *CreateAppModules()* must be implemented here to define the *Modules*, *Devices* and *Transport* connections that should be created at initialization time. Additionally, several other virtual methods can be overwritten to modify the framework default behavior, up to the details of the DAQ state transition functions.

C. Configuration Parameters

The configuration is handled by DABC *Parameters* that are registered in the *Application* constructor and are accessible by name in the methods of the user implemented classes. The parameter values can be loaded from a configuration file at each start of the DABC process. The XML syntax of the configuration file allows techniques like wildcard expressions to easily set up similar *Modules* and *Devices* on multiple nodes with one configuration file.

Moreover, values of the registered *Parameters* can be controlled and monitored at runtime by means of the control system associated with the node *Manager*. Currently DABC supports a control system protocol based on the *Distributed Information Management* (DIM) system [12] and provides a generic Java GUI for visualization of node states and parameter values [13]. Interfacing other control frameworks such as the *Experimental Physics and Industrial Control System* (EPICS) [14] is currently under construction.

IV. Hardware Examples and Performance

A. The PEXOR Board

Our first example is a PCIe board for reading data from front-end hardware via optical links. The *PciExpress Optical Receiver* board (PEXOR) was developed at GSI [15]. It features a Lattice *Field Programmable Gate Array* (FPGA) with serial Gigabit transceiver interfaces (SERDES) to connect PCI Express x1 or x4 (i.e. one or four lanes), and four 2Gbit "small form-factor pluggable" (SFP) optical transceivers. The optical protocol to read out a chain of GSI front-end boards has been implemented [16]. The hardware has previously been tested with drivers and software on Lynx OS, developed for the DAQ system MBS [7].

1) Linux Driver

To use this hardware with DABC, a Linux device driver has been developed. The driver operations for *read()* and *write()* implement *Programmed Input/Output* (PIO) to the PEXOR on-board memory. Operation *mmap()* allocates kernel buffers for *Direct Memory Access* (DMA) operations and maps these to user space addresses. The *pexor* kernel module manages these buffers with separate lists for free buffers, for already filled buffers, and for buffers currently in use by the client programs. Further functionality is available by *ioctl()* operations, such as

- initiating a DMA transfer,
- retrieving and releasing DMA buffers,
- communicating with the front-end boards at the SFP connections,
- accessing any register in the address spaces of the board's PCIe memory, or of the SFP slaves, respectively.

The interrupt handler is prepared for DMA completion and data trigger interrupts which may be used with future versions of the FPGA firmware.

Together with the kernel module, the PEXOR driver package provides a plain C++ library with higher level functionality. It features thread safe classes representing different PEXOR board instances (i.e. hardware versions). These classes offer methods to work transparently with the driver from user space. Furthermore, the library defines wrapper classes for the kernel DMA buffers and their memory pool, and some auxiliary classes for logging and performance benchmarks.

2) DABC Integration

The DABC plug-ins for the PEXOR board directly use this library via a *Device* class, encapsulating the driver functionality as a *Façade* design pattern [9]. The *Transport* class implements methods *Read_Size()* , *Read_Start()* , and *Read_Complete()*, as introduced in section II.B.

For performance tests, data concentrator front-end boards (EXPLODER) [16] were connected to the PEXOR. The *Device* class was implemented with a test mode to write random data into the front-end memories at initialization time. This avoids the need of connecting a "real" data source. Then, for each cycle of the running acquisition, the *Transport* initiates the transfers of this data from the SFPs. The data is collected from all EXPLODER boards connected to one SFP in a token-ring protocol technique, performed by the on-board FPGAs. The token transfer can be handled either asynchronously, *Read_Start()* requests the data block, and *Read_Complete()* receives the filled DMA buffer. Alternatively, the transfer can be handled synchronously, *Read_Complete()* is both requesting the block and receiving the buffer. If more than one SFP chain is connected, the token block read is done in a "parallel" mode. At first, token blocks are requested from all connected SFP chains, and then the token buffer of each channel is polled for completion and further transferred via DMA to the host. Since the SFP channels work independently in the PEXOR FPGA, the token data is read in parallel into PEXOR memory. However, the subsequent DMA transfers to the PC host are serialized over the PCIe bus. This limits the maximum achievable bandwidth.

Because in the current implementation the kernel DMA buffers of the driver are not yet registered to the DABC memory pool, the contents of the received buffer are copied to the DABC target buffer, optionally formatted with headers of the MBS format [7]. A readout *Module* is connected to the *Transport* and provides these data for online monitoring with Go4 [11]. Here the data integrity is checked by unpacking and histograming the submemory contents of the token chain.

3) Performance Measurements

The above test runs a non triggered read-out in a polling loop, where every single *Read_Complete()* gets a full data buffer. Therefore it shows up the maximum possible data rate for this plug-in implementation. Fig. 3 illustrates the measured bandwidth values vs. packet size for different numbers of SFP chains, each equipped with one EXPLODER board.

The plain DMA bandwidth between PEXOR board and PC memory was measured with the driver library to be <550 Mbytes/s (for a maximum of 64 kbytes transfer size of each SFP buffer, PCIe x4 configuration), defining the upper limit for any DAQ utilizing this hardware.

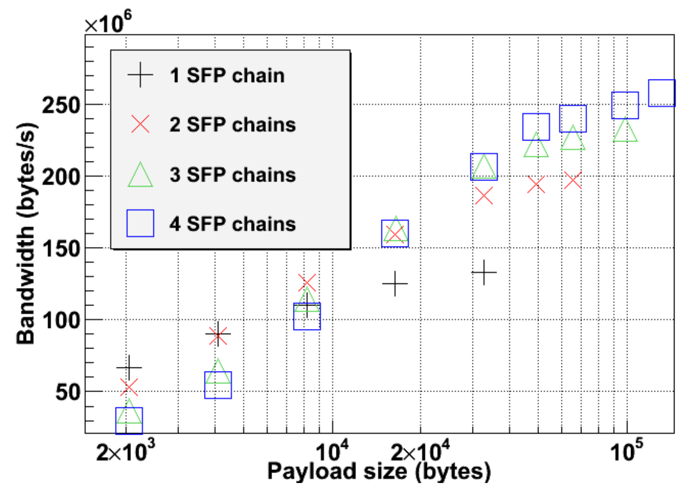

Fig. 3. PEXOR readout bandwidth vs. full data payload size: DABC parallel token block request with subsequent DMA for different numbers of connected SFP chains (1 EXPLODER front-end per chain). Read out of randomly generated data in polling mode without trigger.

The achieved DABC token block readout speed of <250 Mbytes/s (parallel read out, asynchronous mode, 4 connected data chains) is limited by the serialization of SFP, DMA, and memory copy. Since each SFP channel currently has only one token receive buffer in the PEXOR memory, the DMA transfer can not start before each token block read is complete; additionally, after receiving the DMA buffers in the host program, they are copied to DABC user buffers within the same thread. The resulting payload bandwidth B to the host memory is the *harmonic mean* H of all subsequent bandwidth components B_i divided by the number n of such components, as reminded in (1):

$$B = \frac{H}{n} = \left(\sum_{i=0}^{n} \frac{1}{B_i} \right)^{-1} \quad (1)$$

Here the three subsequent bandwidth components B_i are the sum of all SFP bandwidths, the DMA bandwidth, and the memory copy speed from DMA buffer to DABC user buffer. For a 4-SFP-channels' total bandwidth of 800 Mbytes/s (with 2 GBit SFPs), a DMA bandwidth of 550 Mbytes/s, and the measured *memcpy* speed of 2 Gbytes/s, one can expect a theoretical readout bandwidth of 280 Mbytes/s for this implementation, fairly in agreement with the measurements.

Improvements are planned for future PEXOR versions by double buffering at each SFP channel, which will allow DMA transfers during token readout. Moreover, integrating the PEXOR DMA buffers into the DABC memory pool will decouple DMA receiving and memory copying into separate DABC threads. This will speed up the overall bandwidth on a multi processor CPU.

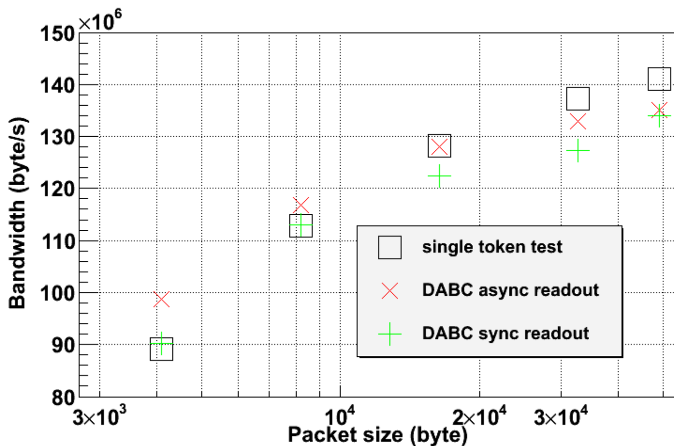

Fig. 4. PEXOR "token block mode" bandwidth comparison vs. packet sizes for a chain of 2 front-end boards at 1 SFP link: request of a single token using the driver library; DABC readout with asynchronous, and synchronous token request mode, respectively.

The asynchronous DABC readout mode turned out to improve performance by about 5% over the synchronous mode (see Fig. 4). In comparison with a single token request of the driver library, the entire DABC framework overhead reduced the token readout speed by only 3%. For small buffer sizes, asynchronous DABC readout is even better, since the data transfer time of about 40 µs is in the order of the Linux context switching time, which will show up if the system schedules out the permanently polling process. For large buffers the DABC bandwidth is mainly affected by the *memcpy* between driver DMA buffer and DABC user buffer. With the harmonic mean estimations as explained above, one can expect for one SFP channel a maximum readout bandwidth of 137 Mbytes/s with memory copy (DABC), and 147 Mbytes/s without (single token test). The MBS data server functionality may also impose a slight penalty in comparison with the plain test.

B. The CBM Readout Controller (ROC)

Another example of a DABC hardware interface is access to the readout controller board (ROC) used for detector tests in the future Compressed Baryonic Matter (*CBM*) experiment [17]. Data from ROC can be delivered to PC either via Ethernet, or via optical fiber connection.

For optical readout a PCIe receiver board (AVNET) is used which has similar functionality as the PEXOR board. The main difference to EXPLODER/PEXOR readout is that the CBM readout controller sends data packets without an explicit request, as required by EXPLODER. The data received by the AVNET board is accumulated in a large FIFO and transported via DMA requests to DABC. The implementation of the *Transport* class is based on *DataTransport* and is very similar to the PEXOR case. The maximum data rate achieved with the ROC pattern generator is about 210 Mbytes/s.

Alternatively, a UDP-based custom protocol has been implemented for communication with ROC via Ethernet. Because UDP does not guarantee packets delivery, a retransmission of lost packets is supported by our protocol. For the implementation of socket-based protocols DABC provides several base classes which allow the developer to work with sockets in an "event triggered" mode. These classes declare virtual methods which are called when a socket has data for reading, or when the user program can send a new portion of data via the socket. The ROC UDP *Transport* implements such classes and can handle connections with several ROCs in a single thread. A data rate of about 12 Mbytes/s per ROC was achieved – this corresponds to the maximum for 100 Mbit Ethernet.

Both optic- and Ethernet-based transports provide similar interfaces to the higher-level code. So the user can set up heterogeneous systems with many ROCs, read out by one or several DABC applications. Such systems were successfully employed for data taking with the CBM test beam at CERN in November 2010, and at COSY (FZ Jülich) in December 2010, respectively [18],[19].

V. Conclusions

The DABC framework provides simple and generic interface classes for integrating a large variety of hardware devices for data I/O. These plug-in points were proven to work with the CBM test beam hardware, and with a general purpose PCIe optical receiver board (PEXOR), respectively. Further development work is going on to add more functionality to both implementation cases.

Acknowledgment

We thank Jan Hoffman, Nikolaus Kurz, Shizu Minami, and Wolfgang Ott of GSI Experiment Electronics Department for help and discussions during development of the PEXOR Linux device driver and the DABC plug-in.

References

[1] J. Adamczewski, H. G. Essel, N. Kurz, and S. Linev, "Data Acquisition Backbone Core DABC", *IEEE Trans. Nuclear Science* vol.55, no.1, Feb. 2008, pp 251-255.

[2] J. Adamczewski, H. G. Essel, N. Kurz, and S. Linev, "Data Acquisition Backbone Core DABC", 2008 *J. Phys.: Conf. Ser.* 119 022002 (8pp).

[3] F. Alessio *et al.*,"The LHCb Readout System and Real-Time Event Management", *IEEE Trans. Nuclear Science* vol.57, no.2, Apr. 2010, pp 663-668.

[4] G. Bauer *et al.*, "CMS DAQ Event Builder Based on Gigabit Ethernet", *IEEE Trans. Nuclear Science* vol.55, no.1, Feb. 2008, pp 198-202

[5] H. P. Beck *et al.*, "Performance of the Final Event Builder for the ATLAS experiment", *IEEE Trans. Nuclear Science* vol.55, no.1, Feb. 2008, pp 176-181.

[6] J. Adamczewski-Musch, H. G. Essel, N. Kurz, and S. Linev, "Data Acquisition Backbone Core DABC Release v1.0", 2010, *J. Phys.: Conf. Ser.* , to be published.

[7] H.G. Essel, J. Hoffmann, N. Kurz, and W. Ott, "The general purpose data acquisition system MBS", *IEEE Trans. Nuclear Science* vol.47, no.2, Apr. 2000, pp 337-339.

[8] J. Adamczewski-Musch, H. G. Essel, N. Kurz, and S. Linev, "Data Flow Engine in DAQ Backbone DABC", *IEEE Trans. Nuclear Science* vol.57, no.2, Apr. 2010, pp 614-617.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 14th ed., Reading, MA: Addison Wesley Longman, 1998

[10] Open Fabrics Alliance (2011, Apr.). OFED overview. [Online]. Available: http://www.openfabrics.org/jofa/index.php?option=com_content&view=article&id=3&Itemid=119

[11] J. Adamczewski, M. Al-Turany, D. Bertini, H.G. Essel, and S. Linev, "Go4 online monitoring" , *IEEE Trans. Nuclear Science* vol.51, no.3, Jun. 2004, pp 565-570.

[12] C. Gaspar (2011, Apr.). Distributed Information Management System DIM. CERN. [Online]. Available: http://dim.web.cern.ch/dim

[13] J. Adamczewski-Musch, H.G. Essel, and S. Linev, "A DIM Based Communication Protocol to Build Generic Control Clients", presented at the 17th IEEE Real-Time Conference, Lisboa 2010, Paper PCM-13

[14] Argonne National Laboratory (2011, Apr.). Experimental Physics and Industrial Control System. [Online]. Available: http://www.aps.anl.gov/epics/

[15] J. Hoffmann, N. Kurz, S. Minami, W. Ott, and S. Voltz, "PCI-express Optical Receiver", GSI scientific report 2008, p 258.

[16] S. Minami, J. Hoffmann, N. Kurz, and W. Ott, "Design and Implementation of a Data Transfer Protocol via Optical Fibre", presented at the 17th IEEE Real-Time Conference, Lisboa 2010, Paper PDAQ-31

[17] S. Linev, J. Adamczewski-Musch, and H. G. Essel "Usage of DABC in software development for CBM DAQ ", CBM progress report 2009, p 56.

[18] J.M. Heuser *et al*., "Test of prototype modules of the CBM Silicon Tracking System in a proton beam at COSY", GSI scientific report 2010, submitted for publication

[19] S. Linev, J. Adamczewski-Musch, and J. Frühauf, "DABC as data acquisition framework for CBM", GSI scientific report 2010, submitted for publication