PCIExpress Communication Layer for ATCA-based Linear Accelerator Control System

Tomasz Kucharski, Adam Piotrowski, Dariusz Makowski, Grzegorz Jabłoński
Department of Microelectronic and Computer Science
Technical University of Łódź
ul. Wólczańska 221/223
90-924 Łódź, Poland

***Abstract*—PCIExpress architecture is widely used communication bus designed, among other things, for industrial application. Additionally, according to PICMG 3.4 specification it is part of an ATCA architecture. For that reason PCIExpress was used as communication interface for data transmission between ATCA carrier boards and AMC modules for the new control system for XFEL linear accelerator. In this paper authors present general overview of this system, describe communication protocols designed to exchange data with external user application and show results of performance test.**

***Index Terms*—PCIExpress Bus, Advanced Telecommunications Computing Architecture, Advanced Mezzanine Card, TCP/IP Server**

I. Introduction

Digital LLRF (Low Level Radio Frequency) control system for superconducting cavities requires information about large number of signals and parameters which are either directly measured or calculated based on physical input signals. The cavity probe signal is converted from 1.3 GHz to the intermediate frequency. Signal is digitized and field vector is calculated. The resulting field vector of each cavities is multiplied by a rotation matrix to calibrate amplitude and phase. The vector sum of 32 cavity fields is subtracted from the setpoint table and the resulting error signal provides a feedback signal to the vector modulator controlling the indicated wave. A feedforward is used to correct repetitive errors. Beam current information is utilized to modify feedforward table to correct additional beam current varies. To reduce cavity detuning errors the cavity detuning is calculated from forward power, reflected power and the prob signal [1]. LLRF control system is connected with other accelerator systems with significant number of digital and analog signals [2]. Therefore, to enhance the availability and reliability of new control system for XFEL (X-ray Free-Electron Laser) linear accelerator, ATCA (Advanced Telecommunications Computing Architecture) [3] standard was chosen as a base for the entire device. Distributed version of LLRF control system currently developed in DESY (Deutsches Elektronen-Synchrotron) is composed of four custom-designed ATCA carrier boards with three AMC (Advanced Mezzanine Card) slots and several additional components like Gigabit Ethernet Switch, CPU board and external PCIExpress (PCIe) Root Complex Board. The PCIe (Peripheral Component Interconnect-Express) available on Fabric Interface is used for transmission of non time-critical data between ATCA blade and AMC modules [4]. PCIe bus is not accessible from outside of the ATCA crate, therefore additional software communication system was created to give access to PCIe from external applications. To control behavior of PCIe devices system was supply with Ethernet-based interface. To provide PCIe-to-Ethernet bridge service there was a need to write PCIe driver for Linux kernel, socket-based application in user space and external communication library for end-user programs.

II. System Overview

A. Communication Levels

A block diagram of PCIe-based communication subsystem with emphasized hardware and software interactions is presented in Fig. 1. The carrier board has high-performance microprocessor with Linux operating system (OS) on board. From software point of view each of the AMC module connected to PCIe bus is mapped directly to the blade address space, therefore it is accessible from the OS level. The PCIe-to-Gigabit Ethernet bridge is a server application, designed to perform communication between external programs connected to the network and PCIe devices transparent for users. Several low-level drivers were designed to exchange data between the bridge application and hardware components connected to the PCIe bus. The drivers are responsible for data formatting,

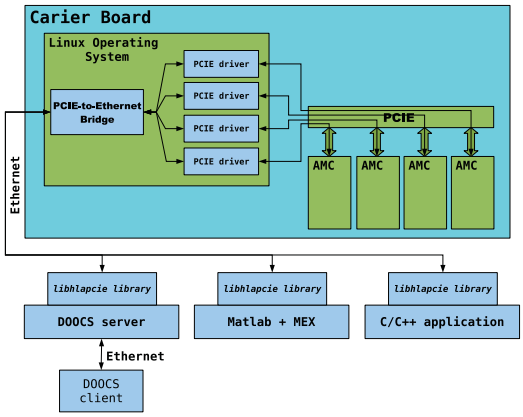

Fig. 1. Overview of PCIe-based communication system

interrupts handling and reading from or writing data to memory locations. An appropriate procedures were implemented for various types of devices. The communication between external client applications and the server bridge is performed using a High Level Application PCIe library (libhlapcie). The library is responsible for data encapsulation, device or register addresses mapping and data format conversion. At the beginning of data transmission, the client application sets the address of the requested PCIe register and, if possible, establishes communication. A transmission frame is sent to the bridge application. An additional communication protocol was introduced to unify data frame format and addressing convention. The PCIe-to-Gigabit Ethernet bridge, based on the contents of the received frame, exchanges information with an appropriate device and sends answers back. The High Level Application library together with the bridge server application constitutes intermediate level of the PCIe communication software subsystem, that can be used with applications like DOOCS servers, Matlab scripts or C/C++ standalone programs i.e. to visualize or control behavior of the hardware devices connected to the PCIe bus. Following sections of this paper presents detailed description of the components of communication subsystem.

B. Device Addressing

Communication system supports three-level addressing mode. The correct address of register available in PCIe address space contains three elements: name of the ATCA carrier board, the name of the requested device and the name of the register in the PCIe namespace. The communication library converts the name of the ATCA carrier board to IP address and the name of the register to the offset relative to the PCIe base address. In the next step PCIe-to-Ethernet bridge, based on the name of device, select appropriate driver and perform data exchange with hardware. Complete address evaluation process is presented in Fig. 2.

III. PCIExpress Device Driver

A. Recognizing Devices

Linux OS can handle any device provided that the manufacturer supplied right drivers. AMC modules were developed by DMCS, so the driver had to be written from the scratch. Every PCIe device introduces itself to the operating system with its device and vendor identification numbers (ID). If we register in the kernel driver that claims to work with a certain vendor and device ID, when device is plugged in, kernel links the device with that driver. Driver is written as a kernel module. It can be easily added and removed at any time without the need of recompiling the kernel or even restarting the system (using insmod and rmmod shell commands). To link driver with specific device or specific group of devices, *pci_device_id* structure must be appropriately filled and registered in the system. VENDOR_ID and DEVICE_ID are numbers that are hard-coded in the PCIe device. PCI_DEVICE is a macro that creates architecture-independent data structure. When the handling list is ready, driver has to export it to the kernel devices table.

```
struct pci_device_id pci_ids[]={
    {PCI_DEVICE(VENDOR_ID, DEVICE_ID)},
    {0,}
};
```

```
MODULE_DEVICE_TABLE(pci, pci_ids);
```

PCIe device registers are seen in OS as memory areas mapped to system addressing space starting from values written to device configuration registers called Base Address Registers (BARs). Number of available memory regions depends on the device configuration. When device is installed the driver has to map physical memory assigned to the device to logical address accessible to driver. This operation is performed by execution of *request_mem_region* and *ioremap* kernels functions. The former allocates physical memory region and the latter maps it to an appropriate virtual address.

B. Accessing Devices

To supply basic functionality, driver has to have implemented appropriate functions to open and close device driver, read and write data from/to hardware and move current position of file pointer, respectively open, close, write, read and seek. Reading and writing is simply reading and writing data from the representing memory using low-level functions. Additionally that functions must transfer data between user space and kernel space. During read operation driver reads 32-bit with *ioread32* and copy it in a local buffer. When the buffer is full, the data is moved to the user space with a *copy_to_user* function. The procedure of writing is similar - data from user space is copied to kernel space with *copy_from_user* function and next it is written to device buffer with *iowrite32*. I/O functions described above have to be registered in system, therefore the driver has to fill structure *file_operations* with pointers to correct write/read/open/close/ioct/seek functions.

```
file_operations pcie_test_fops = {
    .owner   = THIS_MODULE,
    .llseek = llseek_pcie,
    .open    = open_pcie,
    .release = release_pcie,
    .read    = read_pcie,
    .write   = write_pcie,
    .ioctl   = ioctl_pcie,
};
```

The structure is linked to the kernel. When user wants to make an action on the PCIe device, correct function is selected and proceeded.

IV. PCIe-to-Ethernet Bridge

A. Introduction

PCIe-to-Ethernet bridge is an advanced TCP/IP server working as a daemon under control of Linux operating system

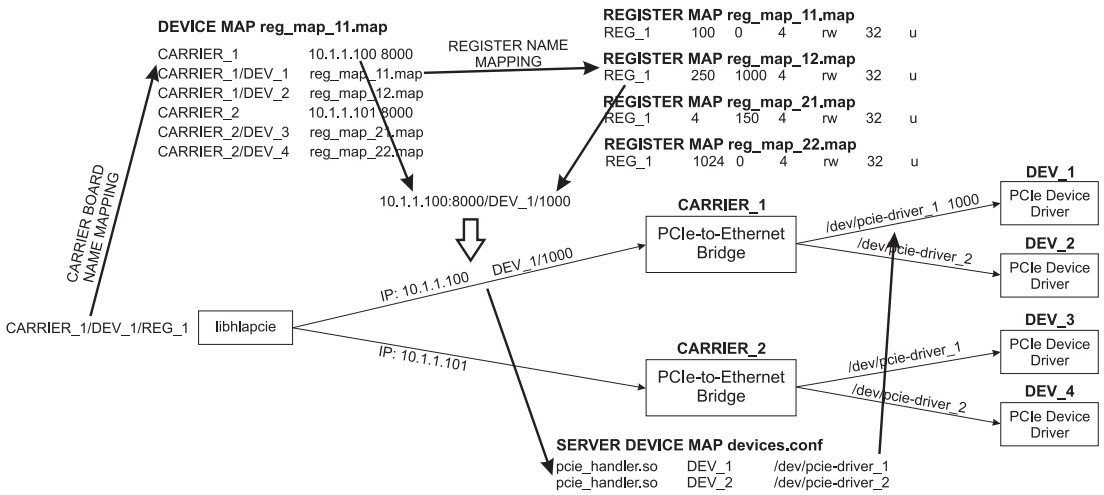

Fig. 2. Address evaluation in PCIe communication system

installed on carrier board. Application waits for requests from external clients and based on contents of received frame, it selects the correct device and establishes communication. Response to command: data in the case of read, write confirmation in the case of write with acknowledge or error frame is sent back to the client. It is worth to mention, that in the server application there is a distinction between the real kernel device name and a common name distributed on Ethernet. Users do not have to bother if a physical device, they are trying to access has changed its name. Server application converts public names to correct linux devices, according to the configuration files. There is a list of structures that stores devices data e.g. local filename that represents the device, name representing the device on the Ethernet, stream descriptor etc. The application has a multi-threaded nature. Main thread initializes all configuration, set up Ethernet features and waits for connections. When a client establishes data transfer, a new thread is created. That thread exists is the system until the second side closes connection.

B. Communication Protocol

To ensure reliable functionality new high level communication protocol was developed. Data exchange between clients and server is performed with the use of frame composed of two elements: fixed-length header and variable-length payload with contents depends on command type. Format of communication frame is presented in Fig 3. Fixed-length part of the frame contains general information about command, like:

- frame type - identify current type of the frame
- request identification number - number that combines request with appropriate answer. Initialized by the random value at the beginning of data transition
- device name - string that identifies target device. Maximum sixteen chars
- payload length - size of variable-length part of the frame

Protocol supports four type of commands:

- Read data from selected device

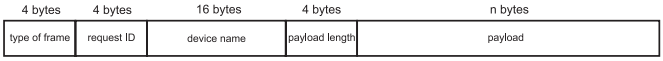

Fig. 3. Communication frames used during data exchange between libhlapcie and PCIe-to-Ethernet server

- Write data to selected device with acknowledge
- Write data to selected device without acknowledge
- List available PCIe devices

Detailed information about structure of frames corresponding to each of the available command are presented in Fig. 4.

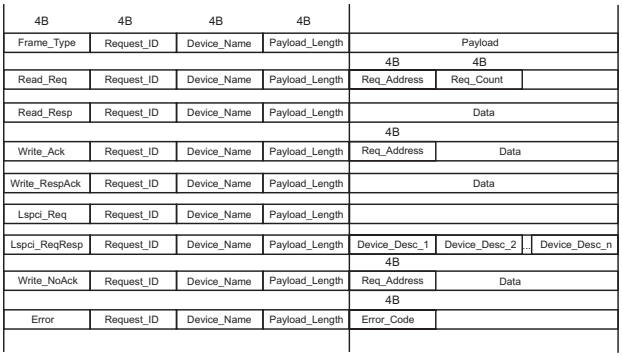

Fig. 4. Available communication frames

C. Configuration Files

The assumption was to create as universal application as it is possible. The easiest way to make it fully customized is to set a bunch of configuration files. The application uses two input files one with basic settings regarding program itself and the other describing devices that application operate on. The aim of the server is to make every device mounted on the machine available on the Ethernet for reading and writing. It can be any PCIe device, but also fake-device like i.e. local files. For each type of device there is a different driver. Far-end user do not have to know what type of device he is

dealing with. The decision of choosing the right driver must me made locally by the server application. For proper working, aware user has to prepare configuration file that state the device type. That function enables adding any device without need of recompiling the whole program. When application is starting, it looks for configuration files and parse them. If some variables are not set by user, the default values, listed in TABLE I, are used.

TABLE I
DEFAULT SERVER APPLICATION PARAMETERS

Parameter name	*Default value*
SERVER_PORT	8000
ERVER_ERROR_DIARY	error_diary.txt
SERVER_DYNAMIC_LIBRARY_FOLDER	./dyn_libs/

D. Dynamic Libraries

Another step to achieve universality is the usage of the dynamic libraries for handling devices. Every type of device may need a different way of treating. Writing and reading data can be strictly imposed by the device driver. Application does not have implemented read/write functions but loads those from external libraries. Advantage of that solution is the easiness for creation of the new devices simulated by a software. That can be either perfect sinus wave generator or application that connects to the network and returns wanted data from external servers. The process of adding new functionality consists of writing dedicated library and locating it in the ”libs” folder. When starting, server application scans that folder and dynamically loads new libraries. This action does not require any extra activities from the user. Dynamic libraries are loaded with *dlopen* function. The pointers to read/write procedures are stored in the list of structures for describing each device separately. When there is a need to work on a device, correct structure is selected and used functions that are defined to handle specific type of request.

E. Dynamic Configuration Reloading

Adding new devices to server does not enforce restarting it. The application has the ability to reload configuration file online. Normally, it would mean stopping all active threads, close the established connections and then closing the application. All currently handled users would be disconnected. That would cause some problems in the client applications. Server has got implemented mechanisms that allows to keep all connections alive and reload all structures without restart of application. To enforce server to perform reload, user has to send USR1 signal. Application run once can work forever without the need of restarting and still serves the ability of adding new devices to the system. Function responsible for reload is linked with the SIGUSR1 callback procedure registered in the server. The *dev_reload* closes all devices and frees structures describing them. The file parser is executed and new data is stored in the configuration structures. Next, program searches for proper libraries that can handle available devices and open them.

F. PCIExpress Self-test

The application is not only a bridge between PCIe and Ethernet but has also some diagnosis interface. While working on Linux with PCIe devices it is good to know what physical buses and slots are currently in use. In regular Linux OS there is the *lspci* command that lists all active cards. The server application is equipped with in-built *lspci* program. Far-end client can ask for that information at any time. Data containing PCIe devices mounted on the local machine is sent during the normal connection session. There is a dedicated frame with a proper format that is understandable for the both transfer sides.

V. HIGH LEVEL APPLICATION PCIEXPRESS LIBRARY

High Level Application PCIe Library (libhlapcie) is a library designed to perform communication between end user applications like Matlab MEX, DOOCS servers or C/C++ standalone application and external PCIe-to-Ethernet servers.

Two type of Application Programming Interface (API) was created, the first one called Low Level API (LLAPI) is designed for applications like DOOCS servers and C/C++ standalone programs and, the second one called High Level API (HLAPI) is designed for Matlab MEX.

A. Low Level API

In LLAPI user is obligated to perform manual management of library. He is responsible for such actions like:

- library initialization by execution of *void lpcie::init* function,
- creation and initialization of *ConnectionSettingsHolder* object for each connection with server by execution of *void lpcie::openDevice* function,
- closing connection by execution of *void lpcie::closeDevice* function.

An example application that presets the use of LLAPI is shown in Listing 1.

B. High Level API

In HLAPI library automatically create and manege *ConnectionSettingsHolder* object, open connection with external server, read or write data and close connection at the end of data transmission. In this approach it is easier to perform communication with PCIe-to-Ethernet server, but efficiency of data transfer will be less then in the case of LLAPI. HLAPI was designed as interface for Matlab MEX extensions. An example application that presets the use of HLAPI is shown in Listing 2.

C. Library Configuration

To secure correct operation of communication library, two configuration files must be delivered. The first one, presented below, is PCIe-to-Ethernet server configuration file and contains two types of information:

- mapping between the name of the carrier board and server IP address and port name,

```
#include "libhlapcie.h"
#include "libuserhlapcie.h"

#define PDEV    "BOARD1/DEV_1/REG_1"

int main(int argc, char *argv[])
{
  int * bf;
  int val = 1;
  ConnectionSettingsHolder csh;
  string dev = lpcie::getDevice(PDEV);
  string carrier = lpcie::getCarrier(PDEV);
  string reg = lpcie::getReg(PDEV);
  try{
    lpcie::init("servers.map");
    lpcie::openDevice(&csh, carrier);
    lpcie::writeMem(&csh, dev, reg, &val, 1);
    bf = lpcie::readMem(&csh, dev, reg, 0, 1);
    delete[] bf;
    lpcie::closeDevice(&csh);
  }catch (lpcie::_exception &e){
    lpcie::closeDevice(&csh);
    return (int)e.getErrorCode();
  }
  return 0;
}
```

Listing 1. Example of libhlapcie Low Level Application Programming Interface

```
#include "libhlapcie.h"
#include "libuserhlapcie.h"

#define PDEV    "BOARD1/DEV_1/REG_1"

int main(int argc, char *argv[])
{
  int * bf;
  int val = 1234;
  try{
    lpcie::writeMem(PDEV, &val, 1);
    bf = lpcie::readMem(PDEV, 0, 1);
    delete[] testBuffer;
    } catch (lpcie::_exception &e){
      return (int)e.getErrorCode();
    }
    return 0;
}
```

Listing 2. Example of libhlapcie High Level Application Programming Interface

- mapping between the name of the device and name of the device map configuration file

```
BOARD1        131.169.132.130 8000
BOARD1/DEV_1 MPC8568MDS.map
```

The second one, presented below, is a device map configuration file and contains detailed information about each of available device registers. The meaning of columns is: component name, number of elements, offset of address, size of elements, access rights, effective bits, type of variable respectively.

```
# Generated by: iigen
# Data bus width:       32

REG_1     1     16384     4     rw     32     u
REG_2     1     16388     4     rw     32     u
FREQ      1     16392     4     rw     32     u
```

VI. Tests

A. Machine Test Stand

First test of PCIe Communication Layer was performed in machine test stand located in Extension Hall in FLASH accelerator in DESY. System was equipped with ADLINK CPU-9600 with 2 AMC bays, and Linux operating system on board. PCIe devices were mapped directly to blade address space therefore there where accessible from operating system level.

B. Piezo Control System

PCIe Communication Layer was used in system for compensation of superconducting cavities detuning using piezoelectric actuators, presented in details in [5].

VII. Conclusions

Developed PCIe-Ethernet layer is a comprehensive communication system with wide range of functionality. It can be used either by advanced users using low-level functions or just hardware developers to test their work using simple interface. Matlab MEX files enable easy visualization of received data. The system is extremely fast. 1G or even 10G Ethernet as a long distance connection and PCIe x8 locally on machine makes the communication very effective. TCP/IP protocol ensures user that the data is reliable and no frame is lost. The system is prepared for the future development. Adding new functions is as easy as writing new libraries to the TCP server. The system was tested and is ready to be implemented.

Acknowledgement

The research leading to these results has received funding from the European Commission under the EuCARD FP7 Research Infrastructures grant agreement no. 227579 and Polish National Science Council Grant 642/N-TESLA-XFEL/09/2010/0.

The authors are a scholarship holders of project entitled "Innovative education ..." supported by European Social Fund.

References

[1] http://mskpc14.desy.de/wiki/index.php/LLRF, "MSK LLRF Wiki - information pages."

[2] D. Makowski and W. Koprek and T. Jezynski and A. Piotrowski and G. Jablonski and W. Jalmuzna and P. Pucyk and S. Simrock , "Interfaces and communication protocols in ATCA-based LLRF control systems," *Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC) 2008, Dresden, Niemcy, ATCA Workshop*, 2008.

[3] PICMG, "AdvancedTCA Base Specification. PICMG 3.0," Tech. Rep., Jan. 2003.

[4] D. Makowski and A. Piotrowski and A. Napieralski , "Universal communication module based on AMC standard," *Mixed Design of Integrated Circuits and Systems (MIXDES) 2008, Pozna, Polska*, 2008.

[5] K. Przygoda and A. Piotrowski and G. Jablonski and D. Makowski and T. Pozniak and A. Napieralski, "ATCA-based control system for compensation of superconducting cavities detuning using piezoelectric actuators," *Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC) 2008, Dresden, Niemcy, ATCA Workshop*, 2008.