FPGA based Tesla cavity controller and simulator
DOOCS server design, implementation and application

Piotr Rutkowski, Ryszard Romaniuk, Krzysztof Poźniak, Tomasz Jeżyński,
Piotr Pucyk, ELHEP Laboratory, Institute of Electronic Systems (ISE),
Warsaw University of Technology
Stefan Simrock
TESLA, DESY, Hamburg

Keywords: TESLA Free Electron Laser (FEL), C++, Superconducting cavity control, DOOCS, Internal Interface

Contents:
0. Abstract..................................................................................................................1
1. Introduction............................................................................................................1
2. System overview ...................................................................................................2
3. Hardware-software system environment ..............................................................4
4. System operation description ..............................................................................6
5. DOOCS server C++ software – description ........................................................6
6. Classes Detail .......................................................................................................10
6. Code example.......................................................................................................13
7. Results..................................................................................................................14
8. Conclusions..........................................................................................................15
9. References............................................................................................................15

0. Abstract

Compact overview on FPGA based Tesla cavity controller and simulator preliminary version of working system. Major emphasise is put on high level software part of the system. Description of following steps: designing of proposed solution, implementing it and getting results from application introduced.

1. Introduction

Tesla accelerator basing on superconducting technology is being constructed in Deutsches Elektronen Synchrotron in Hamburg. The accelerator is controlled by Low Level Radio Frequency (LLRF) system. LLRF consist of hardware and software mixture. The hardware equipment is commanded by Distributed Objected Oriented Control System (DOOCS)\(^1\). In general DOOCS is expected to serve all hardware components in Tesla experiment. Since 2002 ELHEP group\(^2\) has been trying to build part of DOOCS system confined to operate with ELHEP cavity controller and simulator. This work is a presentation of the software development for FPGA DOOCS server with the respect to existing environment and hardware base developed. Article is an attempt to

\(^1\) Link to DOOCS bibliography
\(^2\) Link to ELHEP web page
describe current stage of work from the big picture to the details. In the first part the
general idea is presented – overall system overview. Second part contains some
programming specifics. Results and conclusions are included.

2. System overview

Systematic approach must be introduced in order to keep software reflecting hardware functionality consistent. Three layers communication model has been developed in ELHEP team (Fig. 1).

- On the top sits the **high level software** layer that interacts with hardware functionality
- Second layer covers special **communication interface** enabling hardware access via the well-defined and efficient mechanism[^3]
- On the lowest level is the hardware with the expected **functionality** on board.

Thanks to Internal Interface hardware design is efficient and well coordinated with the software driving the hardware equipment. Software can be on different environments and for different application purposes.

Fig. 1 Three layer communication model.

Designed DOOCS server driving fpga is in the top layer – high level software. ELHEP DOOCS server must be well coordinated with the existing DOOCS infrastructure i.e. Solaris operating system, recent DOOCS libraries (updated by CVS) and DOOCS C++ classes methodology (DOOCS native classes are the core that developer will inherit his classes from). Designed server is required to be flexible for future development and upgrades. Upgrades will surely occur on this preliminary stage before complete ELHEP fpga solution system is finished. These boundaries conditions put developers on focal point for the whole system commissioning.

DOOCS server is a final product combining all hardware functionality for the end-users: experiment operators, scientists, other system parts clients i.e. FSM[^4] etc.

On the Fig. 2 the three layer model and design is presented. In this approach each layer is separated from another. This picture presents way for change hardware register value from the Graphic User Interface level to the hardware level.

[^3]: Link to the note of dr K. Poźniak
[^4]: Link to FSM web page
The aim of the FPGA DOOCS server is to:

a) provide DOOCS users with the access to fpga hardware (registers, memory)
b) control hardware with respect to the requested and implemented algorithms

In order to achieve this, one must be equipped with possibility to hardware access. Thus software interface must be created. This interface should be flexible enough to deliver access to the fpga via different paths i.e. intermediate Windows server or direct LPT link[5]. Flexible means that changing access method from one to the other should have practically no impact on inner server logic. In order to meet this requirement server must be designed in Objected Oriented Programming style. This server was designed in OOP way, which gives these pros:

a) Using interface classes separates users from the implementation specifics
b) Server development is easier – objects and their relations are easy to see and understand
c) Debug and bug fixing is faster because one knows better what C++ code does
d) Understanding the project by the team members can be much better – diagrams etc. Common description language – UML

First version of the FPGA DOOCS classes has been created. One should be able to crate server fitted to the exact user needs and equipped with users control algorithm. Implemented hardware interface has fpga access by the intermediate Windows server. Works on the direct access via the LPT link from the Solaris environment are investigated and should bring the fruits soon.

---

Fig. 2 Three layer implementation example.

5 http://tesla.desy.de/~elhep/home/prutkows/FPGA_DOOCS.pdf
3. Hardware-software system environment

The following section presents overview on system architecture in operation in real test environment.

![System overview diagram](image)

**Fig. 3 System overview**

**System consist of:**

**Hardware –**
1. VME crate situated in the Linac Hall
2. FPGA development board (in the VME crate)
3. SUN – SunOS 5.8 (in the VME crate)
4. PC – Windows 2000, near the VME crate

**Software –**
1. DOOCS server (on the SUN)
2. Intermediate Windows Application with TCP/IP server (on the PC)
System operation can be divided onto two phases:

1. Fpga register accessing via the intermediate Windows server (abbr. - TCP)
2. Accessing fpga registers directly from Solaris environment (abbr. - LPT)

On Fig. .. is shown FPGA board. The development board was mounted on special designed VME suited panel that can be put into standardised VME crate.

*Description part...*
4. System operation description:

TCP (Fig. 3 Data path “1”)

1. Booting – via the USB link from the PC
2. DOOCS – fpga data exchanging:
   a. Read – Doocs property on <1>(SUN) calls <2>(PC) for the read of the
      according register value. <2> translates this call on the commands
      accessing fpga and retrieving data. <2> sends this data to <1>. <1>
      converts hardware hex value to float within expected range.
   b. Write - Doocs property on <1> converts float value to the hardware
      hex value and calls <2> for the write to the proper register. <2>
      translates this call on the commands accessing fpga and writing data.
      To verify process <2> retrieve data from the register and sends data
      back to <1>. <1> checks the difference between write and read values
      and embedded in server logic reacts accordingly to the user
      expectations (i.e. server shutdown or just writing the fact to the log
      file).

LPT (Fig. 3 Data path “2”)

1. Booting – via the USB link from the PC
2. DOOCS – fpga data exchanging:
   a. Read – Doocs property on <1> calls method responsible for the read of
      the according register value. Within this method commands accessing
      fpga and retrieving data are executed. Conversion of hardware hex
      value to float within expected range is performed at the end.
   b. Write – Doocs property on <1> calls method responsible for the write
      to the according register value. Within this method commands
      accessing fpga writing, retrieving verifying data are executed.
      Conversion of hardware hex value to float within expected range is
      performed at the end.

5. DOOCS server C++ software – description

On Fig. 4 system overview is presented. DOOCS client can access fpga hardware via
special classes (inherited from DOOCS native D classes) that access fpga by TCP/IP
communication with Windows application. Windows application from its part
communicates with fpga via the LPT. TCP/IP communication is hidden in the classes
implementing hardware access interface.
Provided D fpga classes offer log mechanism to log file. Strange server behavior may
be easily reported to the user. Basic exceptions handled are reported to log file
automatically.

Nallatech doesn’t provide drivers on Solaris environment currently – in the future ELHEP team may
try to work out the protocol and boot fpga from SUN. It can be difficult, though. That is why this paper
doesn’t include this as an option. This option needs further investigation.
Fig. 4 System C++ overview

Server is supposed to provide end-user – DOOCS client – with the access to expected fpga registers and memory by more that one single way (i.e. TCP/IP or direct LPT) but in the transparent for DOOCS client manner. It means that from client point of view there is no difference between underlying fpga communication mode implementations (it can be difference in performance – write/read time – of each method, though).

TCP – string communication overview

Communication between D fpga classes and the Windows application is implemented by the string commands and responses exchange. There is a set of commands that Windows server understands and can respond to in a well-defined way. Client connected to Windows server may request writing, reading to the fpga registers and memory. Other commands based on user needs are (or easily can be) provided.
Fig. 5 String communication mechanism

Commands exchanging mechanism description (Fig. 5):
1. Client application sends the request to the Windows application with TCP/IP server running
2. Server receives string command parses it to get information what fpga register read/write
3. Windows application accesses expected hardware register and performs read or write operation
4. According to the result of point above the return string is prepared and sent back to the client application.

String exchange has its pros and cons. The main disadvantage in string communication is amount of data that need to be transferred via the TCP/IP link. Binary design protocol could reduce amount of transmitted data. On the other hand simple string exchange make feasible to connect different application to Windows server commanding fpga hardware. ELHEP members have broadly applied this feature[6][7]. Second reason was the fact that intermediate Windows server is a temporary solution compared to direct communication between DOOCS and fpga via the LPT. “Time-to-market” was the crucial factor for making operation with fpga from DOOCS possible.

FPGA DOOCS server classes - overview

The FPGA DOOCS classes design diagram is presented on the Fig. 6. To make understanding of the diagram easier:
- D classes are filled with blue colour.
- Interface classes are filled with green colour.
- Exception classes are filled with orange colour.

7 Links to notes using step operation
Fig. 5 FPGA DOOCS classes diagram
6. Classes Detail

Below there is a description of all classes used in server designed. Classes are in the alphabetic order with comments. Some of descriptions may be trivial but they are included for instruction and documentation purposes.

### Class D_float

```java
class D_float

DOOCS native D class. It enables writing and reading float value from the DOOCS server
```

### Field Summary

<table>
<thead>
<tr>
<th>Private</th>
<th>Fpga_access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ifpga_access</td>
<td></td>
</tr>
<tr>
<td>private lu2_str</td>
<td>u2_str</td>
</tr>
</tbody>
</table>

### Class D_float_fpga

```java
class D_float_fpga

D_float

|-- D_float_fpga
```

**Extends:**

D_float, Tlog_name

Extended DOOCS D class with fpga access via the IFpga_acess interface.
In order to allow log warnings and exceptional situations this class inherits from Tlog_name also

### Field Summary

<table>
<thead>
<tr>
<th>Private</th>
<th>Fpga_access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ifpga_access</td>
<td></td>
</tr>
</tbody>
</table>

### Class D_name

```java
class D_name

DOOCS native D class. It enables writing and reading char* value from the DOOCS server
```

### Class D_name_fpga

```java
class D_name_fpga

D_name

|-- D_name_fpga
```

**Extends:**

D_name, Tlog_name

Extended DOOCS D class with fpga access via the IFpga_acess interface.
In order to allow log warnings and exceptional situations this class inherits from Tlog_name also

### Field Summary

<table>
<thead>
<tr>
<th>Private</th>
<th>Fpga_access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ifpga_access</td>
<td></td>
</tr>
<tr>
<td>private lu2_str</td>
<td>u2_str</td>
</tr>
</tbody>
</table>

### Method Summary

```java
public

fill_spectrum_buf(Tbuf_values)
```

Additional method to basic D_spectrum methods' set. Gets values from the fpga hardware and writes to D_class spectrum storage

### Class Efpga

```java
class Efpga

Exception class.
Exceptions generated by the classes implementing Ifpga_access.
```
abstract class Ifpga_access

Interface class to the fpga hardware. It enables all methods required to access fpga registers and memory. Interface has the least method list needed - in order to keep class' methods easily understandable. This interface is expected to be implemented as:
1. TCP/IP intermediate Windows server.
2. LPT interface
3. Ethernet (connection from SUN to PCs on llrf board)

Interface should report abnormal operation as exception from well defined set of exceptions - derived from E_fpga

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>public abstract void BootFile(std::string )</td>
</tr>
<tr>
<td>public abstract std::string Read()</td>
</tr>
<tr>
<td>public abstract Tbuf_values ReadBuf()</td>
</tr>
<tr>
<td>public abstract void Write(std::string )</td>
</tr>
<tr>
<td>public abstract void WriteBuf(Tbuf_values )</td>
</tr>
<tr>
<td>public abstract std::string WriteRd(std::string )</td>
</tr>
</tbody>
</table>

abstract class Iu2_str

Conversion between float value and hex value (in u2 code) in string form. This conversion is needed by the hardware access interface - Ifpga_interface communicates with underlaying fpga in hex format.

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>public abstract long double GetFloat(std::string )</td>
</tr>
<tr>
<td>public abstract std::string GetString(long double )</td>
</tr>
</tbody>
</table>
**Class Tfpga_clnt**

class Tfpga_clnt

TCP/IP connection basic handling (open, close).
One object of this class is expected to handle connection for all objects of Tfpga_tcp.

**Class Tfpga_tbuf**

Ifpga_access

| +--Tfpga_tcp
|   | +--Tfpga_tbuf

class Tfpga_tbuf

Extends: Tfpga_tcp

Full Ifpga_access implementation. Access to memory. Methods related to fpga registers throws exceptions (they are not implemented in this class)

**Class Tfpga_tcp**

Ifpga_access

| +--Tfpga_tcp

class Tfpga_tcp

Extends: Ifpga_access

First implementation of the Ifpga_access - via the TCP/IP intermediate Windows server. Basic methods to access fpga. Needs to be inherited from and to implement main Ifpga_access interface methods. It is not a full interface - it contains workhorse methods for basic operation in TCP/IP communication.

**Field Summary**

| private Tfpga_clnt | Fpga_clnt |
| protected std::string | Name |

**Method Summary**

| protected std::string |Recv() |
| protected void |Recv0() |
| protected Tbuf_values |Recvs() |
| protected void |Send(std::string ) |

**Class Tfpga_treg**

Ifpga_access

| +--Tfpga_tcp
|   | +--Tfpga_treg

class Tfpga_treg

**Extends:**

Tfpga_tcp

Full Ifpga_access implementation. Access to hardware register. Methods related to fpga buffers throws exceptions (they are not implemented in this class)

**Class Tlog_name**

class Tlog_name

Log writing class. Enable logging to specified file. It automatically adds name and date in each entry written to file

**Method Summary**

| public void | WriteLog() |

**Class Tu2_str**

Iu2_str

| +--Tu2_str

class Tu2_str

**Extends:**

Iu2_str

Implementation of Iu2_str. Conversion of hex code to float within expected range.
6. Code example

Basic operation with classes described is presented below. Creating two objects of classes accessing fpga register - named “SI” - and fpga memory named - “VOUT_I_B” - is shown.

Implementation part:

```c
FILE *log_file_; // log file handler
Tu2_str* U2_str; // float->hex conversion
Tfpga_clnt* Fpga_clnt; // TCP/IP conn handling
Tfpga_treg* Fpga_treg_SI; // fpga register access
D_name_fpga* Name_fpga_SI; // reg access - D class
Tfpga_treg* Fpga_treg_SQ; // fpga register access
D_float_fpga* Float_fpga_SQ; // reg access - D class
Tu2_str* U2_str_dac; // float->hex conversion
Tfpga_tbuf* Fpga_tbuf_voutI; // fpga memory access
D_spectrum_fpga* Spectrum_buf; // memory access - D class
```

Initialisation part:

```c
//opening fpga log file: fpga_server.log
if ((log_file_ = fopen("fpga_server.log","a+")) == NULL)
{
    fprintf(stderr,"Cannot open log file fpga_server.log!");
    exit(-1);
}
// conversion float -> hex for 18 bits
U2_str = new Tu2_str(-1,1,18);

// TCP/IP connection handling
Fpga_clnt = new Tfpga_clnt("131.169.149.195",10024);

// operations to deliver D class for the fpga register
Fpga_treg_SI = new Tfpga_treg("SI",Fpga_clnt);
Name_fpga_SI = new D_name_fpga("TEST.SI text",
     dynamic_cast<Ifpga_access*>(Fpga_treg_SI),log_file_);

// operations to deliver D class for the fpga register with // float -> hex conversion
Fpga_treg_SQ = new Tfpga_treg("SQ",Fpga_clnt);
Float_fpga_SQ = new D_float_fpga("TEST.SQ float",
     dynamic_cast<Ifpga_access*>(Fpga_treg_SQ),U2_str,log_file_);

// operations to deliver D class for the fpga memory with // float -> hex conversion
U2_str_dac = new Tu2_str(-1,1,14);
Fpga_tbuf_voutI = new Tfpga_tbuf("VOUT_I_B",Fpga_clnt,1000);
Spectrum_buf = new D_spectrum_fpga("OUT_I buffer",
     1000,this,Fpga_tbuf_voutI,U2_str_dac,log_file_);
```
7. Results

First versions of ELHEP DOOCS servers for have been developed.

Cavity Simulator

On the Fig. 6 below the result of cavity simulator ELHEP DOOCS server is presented. RPC_TEST client application is connected to this server. Server provides user with access to the simulator parameters embedded into fpga registers. One can easily change desirable cavity simulator parameter within DOOCS system environment.

![Cavity simulator server example](image)

**Access to fpga registers**

**Manipulation on cavity simulator parameters possible**

**Parameters of the cavity simulator model [TC model]**

Fig. 6 Cavity simulator server example.

Cavity Controller

Next figure – Fig. 7 – visualizes fpga readout from one chosen buffer write and read operation. On controller output I there is buffer implemented that one can access for write and read operation. In the first step – on DOOCS server startup – example envelope is written to the buffer. For the illustration purposes the shape of the signal written is sinus. When running server updates periodically DOOCS property reflecting fpga buffer content.

![Cavity controller server example](image)

Fig. 7 Controller server example.
Next figure – Fig. 8 – presents result of server operation with real signal from real cavity.

![Fig. 8 Real signal operation example](image)

**8. Conclusions**

First ELHEP DOOCS servers have been created and preliminary test have been accomplished. Based on this experience the try to enclose server development in a logical stream within existing DOOCS environment was successfully demonstrated. Layer based design – high-level software, communication layer and last but not least - hardware layer - gives the following pros:

- Design of each layer is independent from the other (with established interfaces between layers)
- Task are divided on experts in each area
- Development can be performed for each layer independently
- Responsibility for each level development is well defined

Server design is an iteration process. Knowledge and understanding gathered from first implementation can lend a hand in upgrading software to precisely fit user needs with the respect to manpower available. System performance and reliability tests are in undergoing stage currently.

Future system functionalities – as accelerator exceptions handling – will be provided within ELHEP system or appropriate interfaces to the other parts of Tesla experiment control system (i.e. FSM [4]) will be put up.

**9. References**

[3] link do noty dr K. Poźniaka