TESLA Report 2005-05

Software Layer for SIMCON ver. 1.1. FPGA-based TESLA Cavity Control System,

USER'S MANUAL

Waldemar Koprek, Paweł Kaleta, Jarosław Szewiński,
Krzysztof T. Pozniak, Ryszard S. Romaniuk

Institute of Electronic Systems, Warsaw University of Technology, Poland
ELHEP Group

ABSTRACT

The paper describes design and practical realization of low and high level software for laboratory purposes to control FPGA-based LLRF electronic equipment for TESLA. There is presented a universal solution for particular functional devices of the control system with FPGA chips. The paper describes architecture of software layers and programming solutions of hardware communication based on the proprietary Internal Interface (II) technology. Such a solution was used for the Superconducting Cavity Controller and Simulator (SIMCON) for TESLA experiment (Test Facility) in DESY. The examples of the build and tested software blocks were given in this paper.

This documentation is a unity with TESLA Reports published in 2004 by the Elhep and describing the SIMCON hardware, ver.1.0. The paper was written in a form of a User's Manual.

List of changes

The first version of this technical documentation was published as TESLA Report 2004 -10. The initial document was accompanying TESLA-FEL Report 2004-04.

The present document, as compared with 2004-10, was essentially expanded, revised and supplemented with data and functionalities not present in SIMCON ver.1.0. (March 2004).

The next version of software is prepared for the hardware platform SIMCON ver. 2.1. and then ver.3.0. While the versions of SIMCON 1 and 2 use nearly the same hardware platform (single channel), the SIMCON ver. 3 uses a new eight-channel multilayer PCB of VME format.

CONTENTS

1 System overview 3
2 Software layers definition 4
3 Channel layer 6
3.1 LPT channel 7
3.2 VME Channel (PC) 7
3.3 VME Channel (Solaris) 7
4 Internal Interface Engine (IIE) 8
4.1 Dynamic loading IID files 8
4.2 Using IID in communication 9
4.3 IIE Configuration 11
4.4 IIE API 11
5 Application layer 20
5.1 Matlab 20
5.2 Detailed description of using Matlab functions 22
6 IIE as network client (IIE Adapter) 25
6.1 IIE Adapter Configuration 26
6.2 IIE Adapter Communication Protocol 26
6.3 IIE Compatibity 30
7 Booting FPGA 31
8 Example of using MEX-functions 32
9 Acknowledgement 33
10 References 33

1 SYSTEM OVERVIEW

Each of the devices of the cavity controller system, which bases on the FPGA chip has a separate, internal functional structure. The device operation and the form of the dedicated software depends on the FPGA structure. To minimize the influence of the hardware structure on software there raised a necessity to create special system solution which could be used to describe FPGA-based systems. Such a standard of FPGA description was implemented as a proprietary solution called the Internal Interface (II). Due to the II description of the FPGA - based electronic systems it is possible to build universal software which can be used in laboratory and in experiment conditions. The software is nondependent directly on the hardware structure.

On the base of such solutions the specific capability was achieved to develop the software which can be used to dedicated purposes for particular devices. Various graphical user interfaces can be built, basing on the assumed solution, such as control panels, measurements panels or panels for debugging hardware. Implementation description of such solutions can be found in the further parts of this paper.

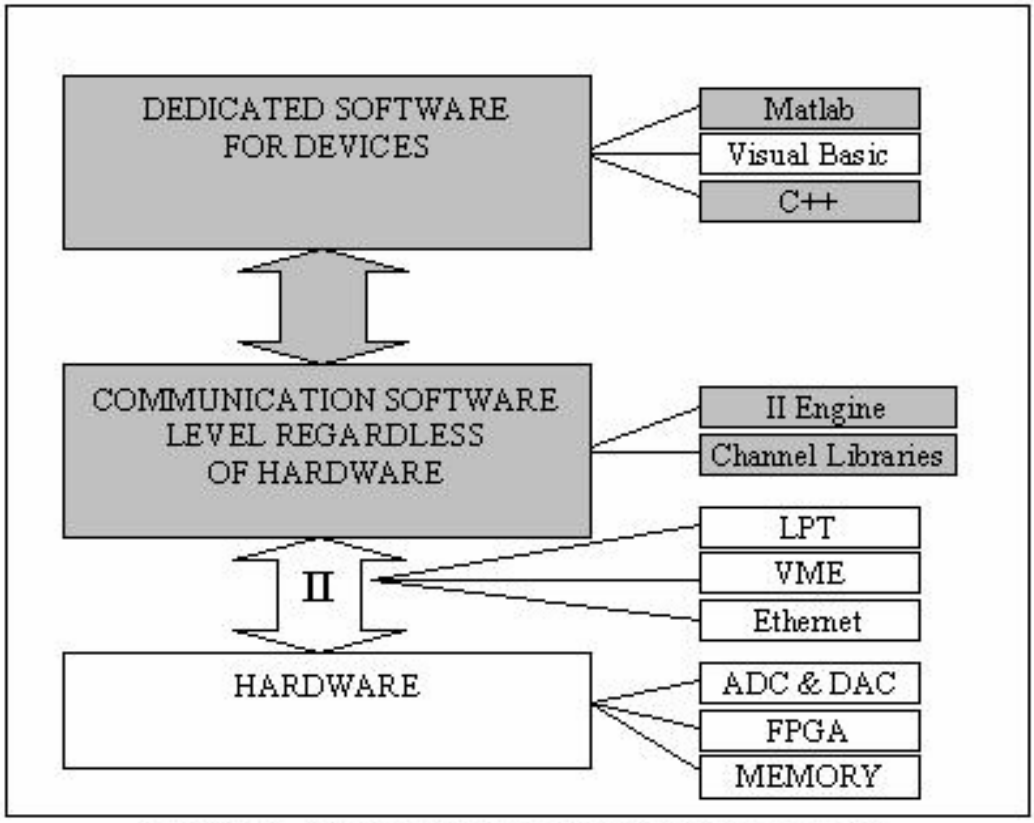

Figure 1. Assumption of software architecutre

The main used technology of the described software environment is the Internal Interface. The II is a proprietary and very efficient method of description of address space. The Internal Interface file with the description (Internal Interface Definition file - IID file) is used to generate the VHDL files that implement this address space in the

FPGA chip. Later, this file is used as the configuration file for the LLRF TESLA cavity control environment.

From the logical point of view, the environment is divided into several layers. The interfaces are defined between the software layers. The various versions of the interfaces are under tests and evaluations. Because of that experimental stage, we can replace one component in the software layer with another one without affecting other parts of the system. This gives us flexibility and possibility to introduce new ideas and new technologies to the project in the future without re-creating the whole project from the very beginning.

The software environment can operate in two modes: local and remote (networked). In the local mode, the client is located on the same machine which has the FPGA hardware connected. In the remote mode, the client is connected to the system over the TCP/IP network. The considerations in the next chapters (2-3) will apply to the local mode, while the remote mode will be discussed in chapter 4.

Currently, the environment has been developed mainly on MSWindows platform (as a laboratory tool-set used to develop FPGA systems), and also some parts of the system have been adapted to the Solaris platform where it works in cooperation with the DOOCS (DOOCS server is a client for the TESLA control environment).

2 SOFTWARE LAYERS DEFINITION

The software system is divided into three layers:

- The Input-Output Layer - This layer is a communicating layer. Its main task is sending and receiving data between the hardware and the rest of the system. The system is designed for the hardware with address-space, thus exchanging the data is developed as a sequence of reading and writing operations into that address-space. The signal part of the system, which implements the I/O layer functionality is called the channel or channel library.
- The Internal Interface Layer - This layer is responsible for serving the information about hardware and enabling communication with it. The Internal Interface Layer imports hardware description which is included in the Internal Interface Description (IID) file. After interpreting that file, the computing of all addresses is performed in the address-space of the hardware. In this way, the Internal Interface Layer has information about all components allocation. A unique name is combined with every component, which in turn is used for the communication with the User Application Layer. This layer is called the IID library.
- The User Application Layer - This layer contains dedicated, user software, which can use previous layers to communicate with the hardware.

The system is developed and encapsulated into the shared libraries with specified functionality. The communication between them is realized throughout the well-described interfaces (API). Every library which implements API can be used in the system and works properly. In this way, the system can be extended with new channels and used in new user applications.

The system concept is independent from the operating system platform. Now it was developed and tested on two platforms: Win32 and Solaris.

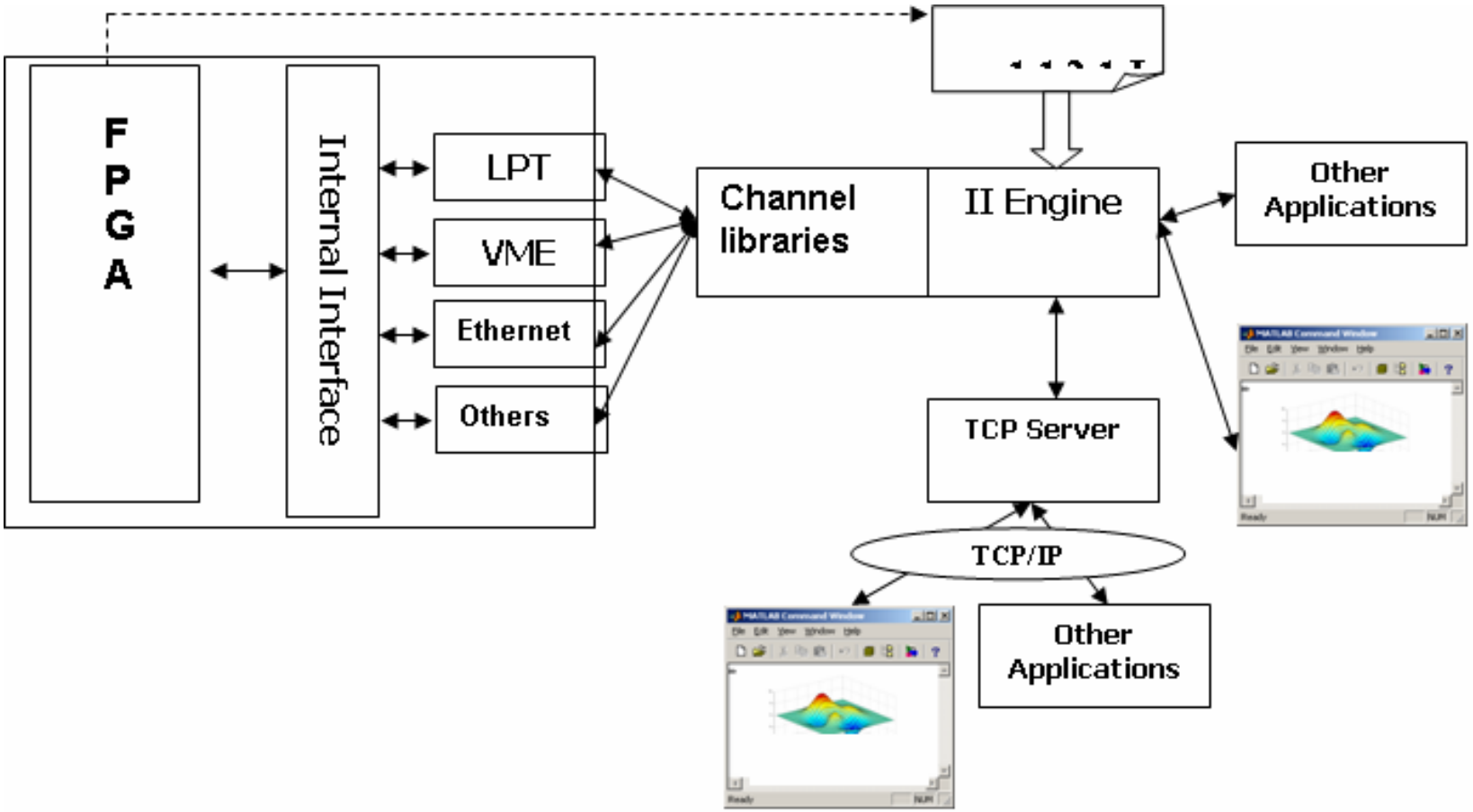

Figure 2. Block diagram of laboratory software for SIMCON

3. CHANNEL LAYER

The Channel is a general software method of communication with the device. Because one single board may contain many FPGA chips, the device is a single address space which can span over one or more FPGA chips.

All channels have the same uniform interface, which is a set of methods that does not dependent on the channel architecture, but describes operations on the higher level of abstraction. For example, there are primitive functions “read” and “write” which let us to operate on the address space.

The other aspect of the channels idea is that the different channels have different properties, for example in the Ethernet the channel data is usually sent as a buffer, but in the LPT interface (using EPP protocol) there is no support for transferring the whole buffers, so data is sent byte after byte. Because of this problem, the channel interface must be general enough to handle all kinds of channels, and not loose their performance. In this case, the channels which are simpler, will emulate the behavior of more complex channels. For example, the LPT channel has implemented a method for reading a single word (which contain 4 bytes). The implementation of buffer read method will simply call method for reading single word many times in the loop.

Each channel has a data bus width – a maximum number of bits which can be transferred simultaneously. For example, the LPT can transfer 8 bits of data during the single read/write operation (not including the control and status lines), while the VME bus can transfer 32 bits of data at once. Unfortunately, each device may have different widths of the address and data busses, the size of these busses is not determined a priori (it may even not be multiplicity of 8). In case of sending operation, each channel must cut data into parts that can fit into the channel bus width. When receiving, the channel is doing the reverse operations to reconstruct data from device’s address space into the computer memory. Also, the hardware must have a functional logic which will reconstruct the transferred data in the FPGA address space, or will prepare data for sending from FPGA to the PC. Example of such solution can be found in [3].

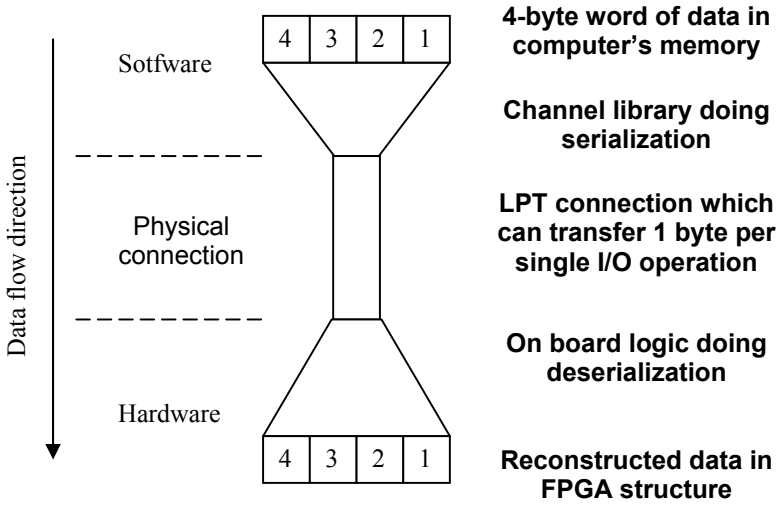

Figure 3. Example of sending data over the LPT channel

The simplest solutions use a single medium (like LPT), which connects the PC with one device. In more complex cases, one machine can control many devices

(multiple address spaces), for example using one Ethernet connection for accessing multiple hardware. Because of that, it was necessary to enable the access to the hardware for multiple client applications, where each client operates on a different device, but on the other hand, each client must have exclusive access to single device. To enable exclusivity, there are used platform depended methods of inter-process synchronization (semaphores, mutexes, critical sections, etc.).

To achieve a flexible uniform interface for all channels, we have implemented them as plugins. Every channel module is a dynamic link library (DLL on MS Windows platform and Solaris environment). This solution gives the possibility to change the way of communication , by simply changing the channel file, without recompiling other parts of the system. The higher levels of system will not even notify this change, because the channels are transparent for them.

3.1. LPT channel

This channel uses the PC's parallel port to communicate with the hardware using EPP protocol. The EPP protocol defines two modes of transmission - address and data – so it is easy to use EPP protocol to cover address space of the FPGA system. In this case, the accesses to the Internal Interface are encapsulated in the EPP protocol (II -> EPP).

On the other hand, the hardware must contain the on-board logic, which translates EPP interface to the board internal bus interface (Internal Interface).

The main usage of this channel is to operate with the hardware, which can not be connected to the VME bus, or in standalone mode when the VME-board is not placed in the bus. The LPT channel is implemented only on Win32 platform as a dynamic link library named **eppii.dll.**

3.2. VME Channel (PC)

This channel was made to enable the user to operate on the boards placed in the VME bus, without the SUN-VME Controller, using a PC class computer and proprietary designed EPP-VME Controller. The PC communicates with the controller over the parallel port (LPT) using the EPP protocol. In this case, the accesses to the internal bus are encapsulated in the EPP-VME control codes, and those codes are encapsulated in the EPP protocol (II -> EPPVME -> EPP)

This channel is implemented only on Win32 platform as a dynamic link library named **vmeii.dll**

3.3. VME Channel (Solaris)

The VME channel on the Solaris platform is implemented as a shared object (dynamic library) which operates in the user-space, and uses the kernel mode driver which controls the VME bus. The reason for creating this library was to serve the channel ("lower") interface to the VME bus for the **libxiid.so** (middle layer engine), which was ported from MS Windows.

4. INTERNAL INTERFACE ENGINE (IIE)

The Internal Interface (II) was developed to automate the local communication interface. It works at the hardware side (VHDL) and the software side (C/C++). The basis of that project is to describe the FPGA I/O area (bits, registers, etc.) using well-known syntax in the Internal Interface Description (IID) file. The IIE was developed to serve II functionality in the software layer of the system.

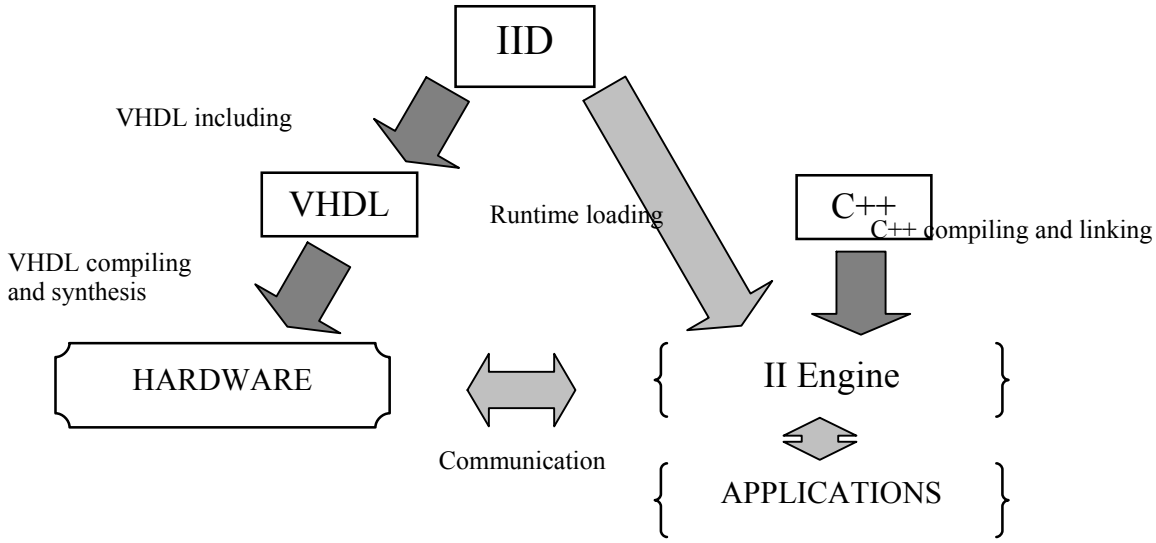

Figure 4. Method of working of the Internal Interface Engine

The working idea of the IIE is different when compared with primary solution of the II ([1]). The IID is always loaded in runtime regime into the system. The IID must be interpreted and used to create the memory-map of the hardware. After that the IIE can serve information from the IID in communication with the hardware. All operations which have been mentioned are described below.

4.1. Dynamic loading IID files

First task of the IIE is runtime loading of the IID files. It makes the process absolutely independent from the software and the changes in the IID files. This is very important in the hardware developing process. This task is released by the IID interpreter.
The IID file is a text file with C-like syntax (syntax of IID is fully described in [1]). There are used predefined types, variables and functions, so it requires the full syntax analyze. The complete interpreting IID process is shown in figure 5.

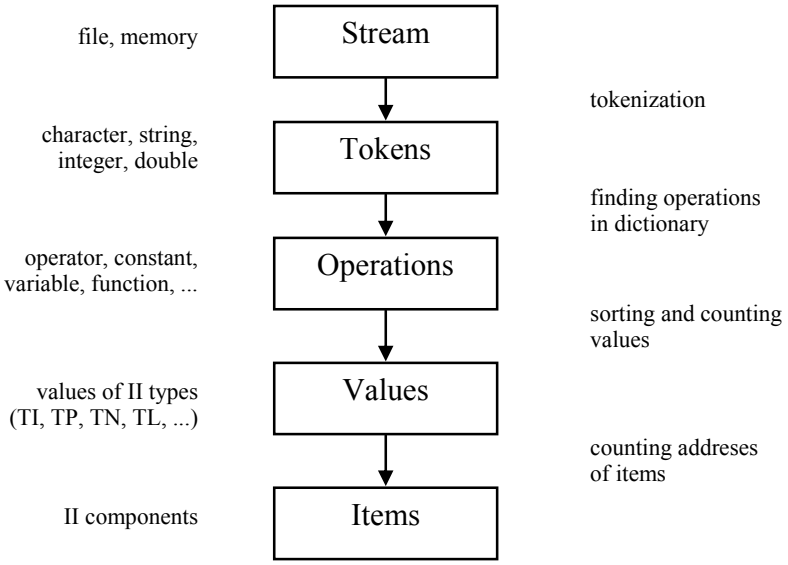

Figure 5. Process of interpreting IID

The IID is taken from the input data stream. In most cases this is a text file. The first step of text analyzing is splitting the text to the elementary phrases. In C notation there are numbers, characters and string of characters. After that, the phrases are compared with the interpreter dictionary to their meaning is found and then translated into operations. The next step of the analysis is sorting and evaluation of all operations. The sorting process secures operations priorities and it is working in agreement with the Reverse Polish Notation (RPN). After evaluating the operations all the values are taken. From these values, the final structures are created. The prepared data is used to create the memory-map of the hardware.

Due to the IID, the system has information about the hardware components and theirs functional collocation. To compute the hardware collocation (address-space) the IID has to contain three constants with predefined names. There are:

- IICFG_ADDR_BASE - constant of TVL type which describes logical hardware base address, which is meant as the base address of the device in the FPGA without the channel,
- IICFG_ADDR_WIDTH - constant of TVL type which describes length of the hardware address bus (in bits),
- IICFG_DATA_WIDTH - constant of TVL type which describes length of the hardware data bus (in bits).

4.2. Using IID in communication

The second task of the IIE is to enable the communication between the hardware and the external application. Using the information, which is extracted from the IID, the IIE enables other applications to communicate with the hardware on the functional level, instead of the hardware level. From the external application point of view, getting data from the hardware reduces to the process of fetching of these data

by the unique name (mnemonic) instead by the physical address. The IIE serves full mnemonics' list, which can be used by the application.

The IIE manages all operations on the hardware components. Every I/O operation contains sub-operations:

- Translation of unique name of the hardware component into the hardware (physical) address – This sub-operation uses the memory-map which is created during the loading of the IID process. Because of structure of the memory-map, the searching process is released in a logarithmic time scale.
- Prevention operations on component without permission – the II enables reducing access to every hardware component. There is a possibility to deny the read or write operation.
- Prevention operations on component with different hardware address – This situation is possible in programming errors in user algorithms. For example, the IIE will return the error while write operation is performed, when there isn't enough space for the value in the component.
- Recalling input-output operation in case of hardware errors – In some cases, the hardware can return error while the communication process is performed. The IIE recalls last communication operation 16 times, until it will return error to the user application.

The system works on three types of data which are derived from the II. There are: word, bit-set and area. In the simplest case (where one word component is related with one hardware address) to set or get a single word component, only one I/O operation is needed. With the area type, the number of I/O operation is rising to the number of its registers. Bits is the most complex type to work with, because it is implemented as part of some register. To get the value of one bit-set component system has to read the register, mask useless bits and shift it to get proper value. In setting operation there is needed one read and one write operation, because the rest of the register must be unchanged before it will be written. The summary of sequence of the I/O operation is showed in table below:

operation \ type	word	bits	area
Setting	write	read, write	write x number
Getting	read	read	read x number

Table 1. Number of I/O operation while communication processes

Every input-output operation can be called multiple in two reasons:

1. There is a hardware error and the system tries to resolve the problem itself. If the hardware returns error after 16 tries, the system will return the error to the user application.
2. IICFG_DATA_WIDTH is set by smaller value that the hardware word has bits. The II can split the components and put them into more than one address, so there are more operations needed.

The number of the I/O operations can be different in one more case and it results

from setting bits operation. Because many bits can be placed in one register, there is possibility to write more than one bits component as single I/O operation. The IIE serves in this case and it is called the merging bits operation. When the merging mode is enabled, the IIE stores the results of the bits setting operations in a buffer, which can be used to write and read on demand. While the merging process (enabling, setting bits and writing on demand) there are called only one reading and one writing operation and it is independent from the number of setting bits operations.

4.3. The IIE Configuration

The IIE is encapsulated in shared dynamic library which is called xiid.dll. It is configurable by two text files:

- source.txt - To configure system to work with hardware the IID is needed. The IIE uses this text file to find the IID file or files. The IID can be written into a single file or more files. In the first case, the configuration file contains one line with a path to the IID file. The path can be full or relative (current directory is configuration file directory). In the second case, the 'source.txt' file contains more than one line. The IID is interpreted in order of entries (from the first line) of the configuration file. The interpreting session is common for all the IID files. This means that the IID, which is included in many files, is treated as IID which would be included in one merged file. Thus, every declaration in one file can be used by the other.
- channel.txt – IIE can serve II functionality independently from the channel, but it must use some channel for low level communication with the hardware. The software, which releases this function must be dynamic library and implemented API of IIE. To connect it to the IIE the 'channel.txt' file must contain a path to the channel library.

4.4. IIE API

init

The init function initiates and configures IIE.

```
int init();
```

Remarks

The init function is used to initiate and configure IIE according to information from configuration files 'source.txt' and 'channel.txt'.
From 'source.txt' there is taken IID files' paths in order of entries. Every file is being interpreted by IIE parser. When errors don't exist and all data is taken, IIE parser is removed from memory. After that memory-map of hardware is computing.
From 'channel.txt' there is taken path to channel library. It used to find, load library in the system and initiate channel to work.

Return values

If no error occurs, init returns zero. Otherwise, a non-zero value is returned and internal error register is set. Error register can be read by get_last_error function.

destroy
The destroy function ends work with IIE library.

```
int destroy();
```

Remarks
The destroy function is used to release previously allocated memory and close channel library.

Returns values
Function always returns zero.

get_status
The get_status function returns status of last channel operation.

```
const unsigned int get_status();
```

Remarks
The get_status function is used to read status of last channel operation. Status is set by channel only and not masked.

Return values
Function always returns status register. When the 7th bit of status register is set that means communication error and error register is set by *IIERR_DEVICE_UKNOWN_PROTOCOL* value. When the 6th bit of status register is set that means checksum error and error register is set by *IIERR_DEVICE_BAD_CHECKSUM* value. Error register can be read by get_last_error function.

get_last_error
The get_last_error function returns identificator of last occurred error.

```
const unsigned int get_last_error();
```

Remarks
The get_last_error function is used to read identificator of last occurred error.

Return values
Function always returns error register. No return value is reserved to indicate an error.

Error codes
IIERR_OPERATION_SUCCESS
Last operation has finished without any error.
IIERR_DEVICE_NOT_LOADED
Channel library can not be load or initiate.
IIERR_DEVICE_BAD_CHECKSUM
There has been error of bad checksum while I/O operation.

IIERR_DEVICE_UKNOWN_PROTOCOL
There has been error of communication protocol while I/O operation.
IIERR_DEVICE_ITEM_NOT_FOUND
II component doesn't exist in IIE storage.
IIERR_DEVICE_ACCESS_DENIED
I/O operation on II component is deny.
IIERR_DEVICE_TYPE_MISMATCH
Operation for type of selected II component is forbidden.
IIERR_DEVICE_OUT_OF_BOUNDS
Value is bigger than available space in II component.
IIERR_DEVICE_CONFIG_FAILED
IID doesn't contain all IICFG constants.
IIERR_CONFIG_SOURCE_NOT_FOUND
Configuration file 'source.txt' hasn't been found.
IIERR_CONFIG_CHANNEL_NOT_FOUND
Configuration file 'channel.txt' hasn't been found.
IIERR_PARSING_ERROR
Interpreting IID has been corrupted because of parse errors.

get_addr_width
The get_addr_width function returns length of address bus of hardware.

```
const unsigned long get_addr_width();
```

Remarks
The get_addr_width function is used to read value which describes length of address bus of hardware in bits units.

Return values
Function returns length of address bus of hardware. No return value is reserved to indicate an error.

get_data_width
The get_data_width function returns length of data bus of hardware.

```
const unsigned long get_data_width();
```

Remarks
The get_data_width function is used to read value which describes length of data bus of hardware in bits units.

Return values
Function returns length of data bus of hardware. No return value is reserved to indicate an error.

get_item_id
The get_item_id function converts unique name of II component to its identificator.

```
const unsigned int get_item_id(
     const char *name
);
```

Parameters
name
Null-terminated string which represented unique name of II component.

Remarks
The get_item_id function is used to convert unique name of II component to its indentificator.

Returns values
If no error occurs, get_item_id returns identificator of II component. Otherwise, a maximum value which can be represented by unsigned int type is returned. In case error function set error register. Error register can be read by get_last_error function.

get_items_count
The get_items_count function returns number of all II components stored by IIE.

```
const unsigned int get_items_count();
```

Remarks
The get_item_count function is used to read number of all II components stored by IIE.

Returns values
Function returns number of all II components. No return value is reserved to indicate an error.

get_item
The get_item function get full information about II component.

```
int get_item(
     const unsigned int id,
     struct iid_item_t * const item
);
```

Parameters
id
Identificator of II component.
item
Pointer to structure where information will be placed.

Remarks
The get_item function is used to read all information about II component. This data is stored into structure type iid_item_t.

```
     struct iid_item_t
```

```
{
     char name[256];
     unsigned int id;
     unsigned int pid;
     unsigned char type;
     unsigned char wrtype;
     unsigned char rdtype;
     unsigned long wrpos;
     unsigned long rdpos;
     unsigned long width;
     unsigned long number;
     unsigned long addrpos;
     unsigned long addrlen;
};
```

Returns values
If no error occurs, function returns non-zero value. Otherwise, a zero value is returned and error register is set. Error register can be read by get_last_error function.

set_word
The set_word function set new value into II word component.

```
int set_word(
     const unsigned int id,
     const unsigned long data
);
```

Parameters
id
Identificator of II word component.
data
A value to set.

Remarks
The set_word function is used to set value into II word component. Function call write operation from channel library. Number of these calls can be variable. For more details see section 'Using IID in communication'. Before setting function makes tests for correction of II component type and possibility to write.

Returns values
If no error occurs, function returns non-zero value. Otherwise, a zero value is returned, error register and status register are set. Error register can be read by get_last_error function. Status register can be read by get_status function.

get_word
The get_word function get stored value from II word component.

```
int get_word(
     const unsigned int id,
     unsigned long * const data
);
```

Parameters

id

Identificator of II word component.

data

A pointer to buffer for value.

Remarks

The get_word function is used to get value from II word component. Function call read operation from channel library. Number of these calls can be variable. For more details see section 'Using IID in communication'. Before getting function makes tests for correction of II component type and possibility to write.

Returns values

If no error occurs, function returns non-zero value. Otherwise, a zero value is returned, error register and status register are set. Error register can be read by get_last_error function. Status register can be read by get_status function.

set_bits

The set_bits function set new value into II bits component.

```
int set_bits(
     const unsigned int id,
     const unsigned long data
);
```

Parameters

id

Identificator of II bits component.

data

A value to set.

Remarks

The set_bits function is used to set value into II bits component. Function call read and write operation from channel library. Number of these calls can be variable. For more details see section 'Using IID in communication'. Before setting function makes tests for: correction of II component type and possibility to write.

Returns values

If no error occurs, function returns non-zero value. Otherwise, a zero value is returned, error register and status register are set. Error register can be read by get_last_error function. Status register can be read by get_status function.

merge_bits

The merge_bits function merge new value into II bits component.

```
int merge_bits(
     const unsigned int id,
     const unsigned long data
);
```

Parameters

id

Identificator of II bits component.

data

A value to set.

Remarks

The merge_bits function is used to set value into II bits component when more than one II bits component can be set by one I/O operation. Function doesn't use any I/O operation, but stores data into merging buffer until set_merged_bits will be called. Merging operation can be failed when to bits component aren't located in one register. For more details see section 'Using IID in communication'. Before merging function makes tests for: correction of II component type and possibility to write.

Returns values

If no error occurs, function returns non-zero value. Otherwise, a zero value is returned, error register is set. Error register can be read by get_last_error function.

set_merged_bits

The set_merged_bits function set new value into II bits component in merging mode.

```
int set_merged_bits();
```

Remarks

The set_merged_bits function is used to set value into II bits component after merge_bits function calls. Function uses value from merging buffer to set new value. Function call read and write operation from channel library. Number of these calls can be variable. For more details see section 'Using IID in communication'.

Returns values

If no error occurs, function returns non-zero value. Otherwise, a zero value is returned, error register and status register are set. Error register can be read by get_last_error function. Status register can be read by get_status function.

get_bits

The get_bits function get value from II bits component.

```
int get_bits(
     const unsigned int id,
     unsigned long * const data
);
```

Parameters

id

Identificator of II bits component.

data

A pointer to buffer for value.

Remarks

The get_bits function is used to get value from II bits component. Function call read operation from channel library. Number of these calls can be variable. For more details see section 'Using IID in communication'. Before getting function makes tests for: correction of II component type and possibility to write.

Returns values

If no error occurs, function returns non-zero value. Otherwise, a zero value is returned, error register and status register are set. Error register can be read by get_last_error function. Status register can be read by get_status function.

set_area

The set_area function set new values into II area component.

```
int set_area(
     const unsigned int id,
     const unsigned long *data,
     const unsigned long num,
     const unsigned long off
);
```

Parameters

id

Identificator of II area component.

data

A pointer to array of values to set.

num

Number of values to set.

off

Offset from the beginning of II area component.

Remarks

The set_area function is used to set value into II area component. There is possibility to set all space in component or only part of it. Function call read and write operation from channel library. Number of these calls can be variable. For more details see section 'Using IID in communication'. Before setting function makes tests for: correction of II component type and possibility to write.

Returns values

If no error occurs, function returns non-zero value. Otherwise, a zero value is returned, error register and status register are set. Error register can be read by

get_last_error function. Status register can be read by get_status function.

get_area
The get_area function get values from II bits component.

```
int get_area(
     const unsigned int id,
     unsigned long * const data,
     const unsigned long num,
     const unsigned long off
);
```

Parameters
id
Identificator of II area component.
data
A pointer to array for values.
num
Number of values to get.
off
Offset from the beginning of II area component.

Remarks
The get_area function is used to get values from II area component. Function call read operation from channel library. Number of these calls can be variable. For more details see section 'Using IID in communication'. Before getting function makes tests for: correction of II component type and possibility to write.

Returns values
If no error occurs, function returns non-zero value. Otherwise, a zero value is returned, error register and status register are set. Error register can be read by get_last_error function. Status register can be read by get_status function.

5. APPLICATION LAYER

The application layer contains user applications, which need to access the FPGA hardware. These applications are called clients of the system. Any application which uses “upper” interface may become a client of the system. This “upper” interface is realized as a set of functions exported from the shared library. In short, to enable a custom application use the FPGA access environment, a programmer needs only to load a shared library and locate exported functions.
Currently, in most cases the Matlab is used as a client of the system.

5.1. The Matlab

Because a lot of work is invested in the modeling of the Tesla SC Cavity in Matlab, there was a strong need for the comparison of the results of the modeling with the data measured on the real cavity. Because of that, there have been written tools for the Matlab, which enable the user to operate on the external hardware. These tools have been implemented as MEX’es (Matlab Executable’s). They appear as the Matlab functions (like M.-files).

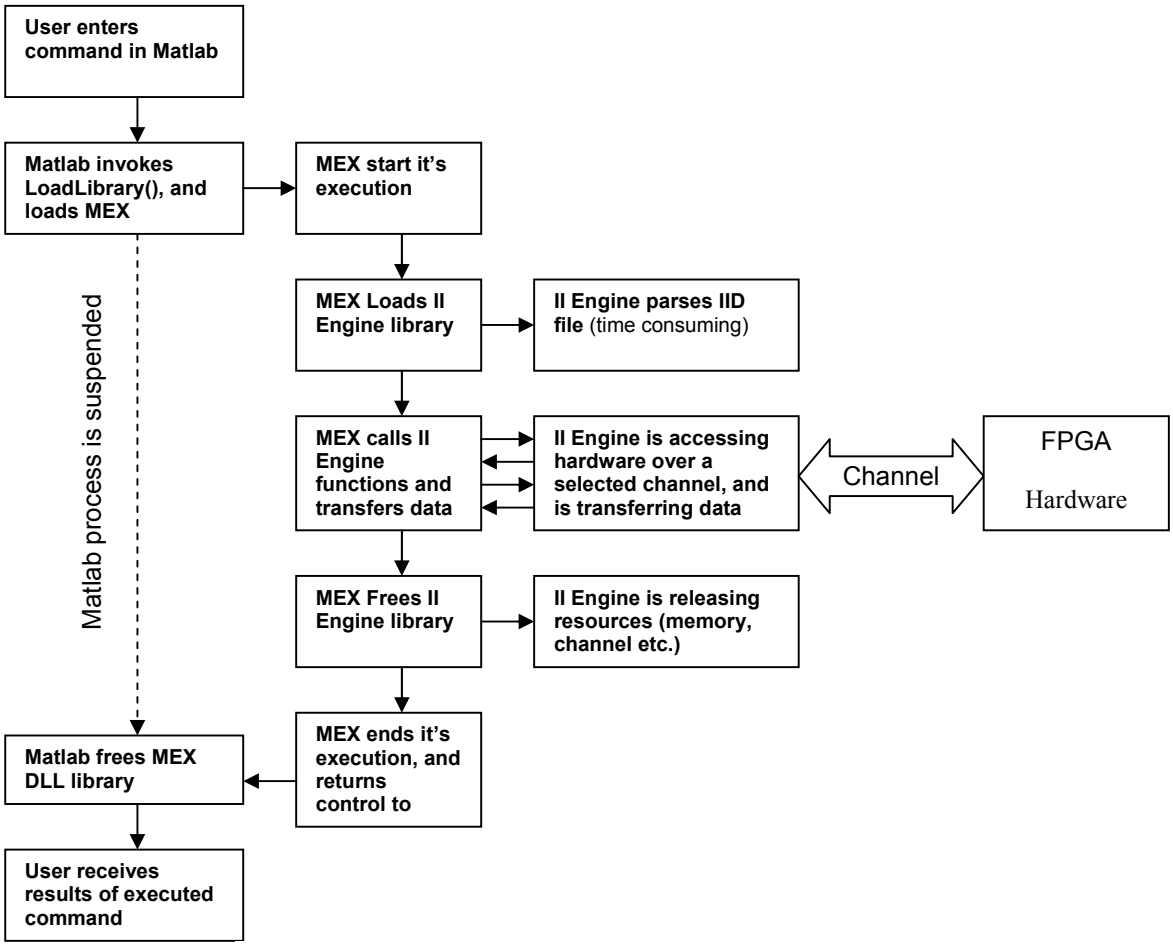

Figure 6. Typical lifecycle of MEX in this project

Because all MEX-files are implemented as the dynamic link libraries, it is a problem to keep the internal state of MEX-functions between separate calls to the same MEX functions. After the MEX function finishes its execution, all the internal data (which was stored on the call stack) is lost. In practice, it means that each call to the MEX function causes the middle layer library (libxiid.so) to perform full initialization (including IID files parsing), which is time consuming.

The typical life cycle of the MEX is presented on figure 3.

Unfortunately, the Matlab uses the same thread to handle the user input and the MEX execution, because of that Matlab process is suspended (window is blocked) while MEX is being executed. The problem is when the II Engine is loaded, it parses IID file to translate names (mnemonics) of the FPGA registers and memory areas into physical addresses. This operation is time consuming, it may take even few seconds (depending on the CPU speed). This is extremely uncomfortable for the user when reading or writing hundreds of registers in the FPGA.

To walk around this problem the following technique was used, there are two special MEX files, both of them do nothing, except that first (ii_lock) loads II Engine (unbalanced call to LoadLibrary(), without calling FreeLibrary() before exit), and second (ii_unlock) releases II Engine (unbalanced call to FreeLibrary()).
When using this method, MEX files are not loading the whole II Engine each time, but they only attach to preloaded library witch has calculated addresses of all registers in the computer memory.

The technique described above is possible, because the calls to LoadLibrary() and FreeLibrary() are cumulative; FreeLibrary must be called as many times as LoadLibrary was, the other way the system will keep the library in memory as long as number of calls to FreeLibrary() is less than number of call to LoadLibrary().
DLL libraries are mapped into address space of the calling process, the DLL_PROCESS_ATTACH event is notified only during the first call of the LoadLibrary for calling process, and DLL_PROCESS_DETACH event is notified only during the last call of the FreeLibrary for the calling process.

In the paragraph above, as the examples were used Win32 API functions, but all techniques described above are available on Unix-like system using dynamic linking interface (dlfcn.h). There is only one difference, calls to dlopen are not cumulative, after many calls to dlopen, first call to dlclose will unload the library. Because of that, dlclose should not be called in any MEX files except ii_unlock (described below).

The following Matlab MEX functions have been created to communicate with the hardware:

- ii_get_word
- ii_set_word
- ii_get_bits
- ii_set_bits
- ii_get_area
- ii_set_area
- ii_get_items

- ii_lock
- ii_unlock

5.2. Detailed description of using Matlab functions

[ ] = ii_lock()

parameters: none
return values: none

This function is usually called first, before all other *ii_** functions. It preloads xiid and channel libraries. At this point, those libraries perform initialization (which may be time consuming), so other ii_* functions attach to loaded and initialized libraries.
Using this function is not obligatory, but in such a case, each call to “ii” function will perform full initialization, including parsing IID files and preparing channel (if provided). These operations are usually time consuming and make working with the system uncomfortable, so using this function is recommended.

[ ] = ii_unlock()

parameters: none
return values: none

This is complementary function to the ii_lock, it frees libraries preloaded by ii_lock. This function should by called when the user has finished operating on the hardware to let operating system remove shared libraries from the memory.

If Matlab Application is going to be closed, it is not obligatory to call ii_unlock, because all dynamic libraries will be unloaded any way.

items = ii_get_items()

parameters: none
return values: items – vector of structures, each structure contain information about one element of the Internal Interface.

Fields:

name - string that describes name of the element – mnemonic
type - string that describes the type of the element, it can be one of the following values: „Page”, ”Area”, ”Word”, ”Vector”, ”Bits”, ”Unknown”
width - uint32 value that describes size of the element (in bits)
number - uint32 value that describes number of sub-elements

This provides ability to determine the structure of FPGA system (description is taken from IID file).

Exaples:
>> items = ii_get_items(); - read the structures
>> items(1).name - name of the first element

>> items(3).type - type of the third element
>> items.name - names of all II elements

word = ii_get_word(name)

parameters:

name – name of the requested item (string)

return values:

word – uint32 value containing requested data

This function returns value of the word identified by the parameter “name”.

Example:
x = ii_get_word(‘USER_REG1’);

[ ] = ii_set_word(name,word)

parameters:

name – name of the requested item (string)

word – uint32 value with new data for requested word

return values: none

This function assigns new value to the word identified by the parameter “name”.

Example:
ii_set_word(‘USER_REG1’,uint32(123));

bits = ii_get_bits(name)

parameters:

name – name of requested item

return values:

bits – uint32 value of the bits identified by the parameter name.

This function returns value of the bits identified by the parameter “name”.

Example:
x = ii_get_bits(‘BITS_5’);

[ ] = ii_set_bits(name,bits)

parameters:

name – name of requested item

bits – uint32 value with new data for requested bits

return values: none

This function assigns new value to the word identified by the parameter “name”.

Example:
ii_set_bits(‘BITS_5’,uint32(123));

area = ii_get_area(name,start,size)

parameters:

name – name of requested item

start – offset calculated from the beginning of the area (type: uint32)

size – requested number of words to read

returned values:

area – vector of uint32 words read from the specified range (described by “start” and “size” parameters) of requested area.

This function returns a vector of values read from the area identified by the “name” parameter. Returned values are taken from the range specified by the parameters “start” (offset) and “size”(number).

Example:

Variable x will contain first 15 elements of area named “AREA_1”

```
x = ii_get_area('AREA_1',uint32(0),uint32(15));
```

[ ] = ii_set_area(name,start,size,area)

parameters:

name – name of requested item

start – offset calculated from the beginning of the area (type: uint32)

size – requested number of words to read

area – an uint32 vector of new values to write to the area

returned values: none

This functions stores values from the parameter “area” in the area specified by “name” in location specified by parameters “start” and “size”.

Example:

```
X = uint32(ones(1,15));
ii_set_area('AREA_1',uint32(0),uint32(15),x);
```

6. The IIE AS NETWORK CLIENT (IIE ADAPTER)

The IIE is designed to work on one platform, but it can work in network as well. The environment can operate in two modes: local and remote. In the local mode, the client is located on the same machine which has the FPGA hardware connected, in remote mode the client is connected to the system over TCP/IP network.

All previous considerations were made in context of the local mode, to enable remote mode, it is necessary to make two steps:

- On the system which has the hardware connected, the user has to stop the current Client Application to and start a TCP server as a client, typically execute ii_unlock in Matlab (if it is running and has been connected to the hardware), and execute the start.bat file from server files.
- On the system which will be used by the operator, the user must place (replace) special libxiid.so library which is called “lightweight xiid”. This library is different from the normal xiid, because it does not parse IID files and does not communicate with hardware directly, but it sends all client request to the server, and brings back the responses from server to client.

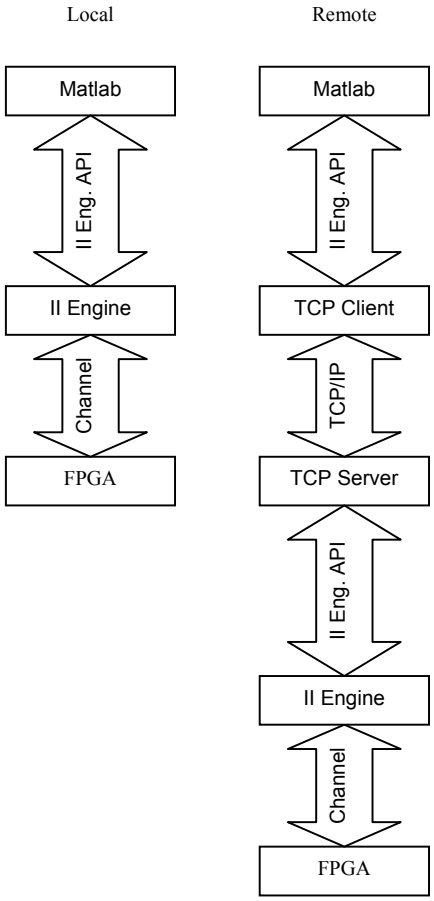

Figure 7. Local and remote configuration

TCP/IP communication is transparent for the all parts of system except the server and lightweight xiid, so any application or Matlab script can operate in both configurations without any modifications. Any hardware that is controlled by the system can be operated remotely.

Because of constant interface of IIE, working in network is served by additional software layer. It is developed in server-client architecture. Server is treated as user application for IIE and client is treated as IIE Adapter for user application on the remote computer. IIE and IIE Adapter have the same interface so it is fully transparent for user application.

Communication protocol uses commands. They describe functions which must be call by server. All available commands are showed below:

CMD_GET_ITEM_ID = 0,
CMD_SET_WORD = 1,
CMD_GET_WORD = 2,
CMD_SET_BITS = 3,
CMD_GET_BITS = 4,
CMD_SET_AREA = 5,
CMD_GET_AREA = 6,
CMD_GET_ITEMS_COUNT = 7,

CMD_GET_ITEM = 8,
CMD_MERGE_BITS = 9,
CMD_SET_MERGED_BITS = 10

Single communication operation uses one request and one response frame. Each of frame contains command byte as the first byte in frame. When request and response frames have the same command it means that they describe the same operation.

6.1. IIE Adapter Configuration

IIE Adapter is encapsulated in shared dynamic library which is called xiid.dll (for compatibility reasons). It is configurable by two text files:

- host.txt - IIE Adapter uses this text file to get information about host. It can be IP address or full name of server.
- port.txt – IIE Adapter uses this text file to get information about port's number in host to communicate with.

6.2. IIE Adapter Communication Protocol

get_item_id

request frame:
CMD_GET_ITEM_ID, char length, char name[]

Parameters:
Length - Length of name in bytes.
name
Name of II component.

response frame:
CMD_GET_ITEM_ID, char status {, unsigned long id }

Parameters:
status
If no error occurs, status equals zero value. Otherwise, non-zero value is set.
id
If no error occurs, id equals identificator of II component. Otherwise, this field is omitted.

get_items_count

request frame:
CMD_GET_ITEM_COUNT

response frame:
CMD_GET_ITEM_COUNT, char status {, unsigned long count }

Parameters:
status
If no error occurs, status equals zero value. Otherwise, non-zero value is set.
count
If no error occurs, count equals number of II components. Otherwise, this field is omitted.

get_item

request frame:
CMD_GET_ITEM, unsigned long id

Parameters:
id
Identificator of II component.

response frame:
CMD_GET_ITEM, char status {, struct iid_item_t item }

Parameters:
status
If no error occurs, status equals zero value. Otherwise, non-zero value is set.
item
If no error occurs, item equals information about II component. Otherwise, this field is omitted.

set_word

request frame:
CMD_SET_WORD, unsigned long id, unsigned long data

Parameters:
id
Identifier of II word component.
data
A value to set.

response frame:
CMD_SET_WORD, char status

Parameters:
status
If no error occurs, status equals zero value. Otherwise, non-zero value is set.

get_word

request frame:

CMD_GET_WORD, unsigned long id

Parameters:
id
Identificator of II word component.

response frame:
CMD_GET_WORD, char status {, unsigned long data }

Parameters:
status
If no error occurs, status equals zero value. Otherwise, non-zero value is set.
data
If no error occurs, data equals value from II word component. Otherwise, this field is omitted.

set_bits

request frame:
CMD_SET_BITS, unsigned long id, unsigned long data

Parameters:
id
Identificator of II bits component.
data
A value to set.

response frame:
CMD_SET_BITS, char status

Parameters:
status
If no error occurs, status equals zero value. Otherwise, non-zero value is set.

merge_bits

request frame:
CMD_MERGE_BITS, unsigned long id, unsigned long data

Parameters:
id
Identificator of II bits component.
data
A value to merge.

response frame:
CMD_MERGE_BITS, char status

Parameters:
status
If no error occurs, status equals zero value. Otherwise, non-zero value is set.

set_merged_bits

request frame:
CMD_SET_MERGED_BITS

response frame:
CMD_SET_MERGED_BITS, char status

Parameters:
status
If no error occurs, status equals zero value. Otherwise, non-zero value is set.

get_bits

request frame:
CMD_GET_BITS, unsigned long id

Parameters:
id
Identificator of II bits component.

response frame:
CMD_GET_BITS, char status {, unsigned long data }

Parameters:
status
If no error occurs, status equals zero value. Otherwise, non-zero value is set.
data
If no error occurs, data equals value from II bits component. Otherwise, this field is omitted.

set_area

request frame:
CMD_SET_AREA, unsigned long id, unsigned long off, unsigned long num, char data[]

Parameters:
id
Identificator of II area component.
off
Offset from the beginning of II area component.
num
Number of bytes to set.

data
Array of bytes to set.

response frame:
CMD_SET_AREA, char status

Parameters:
status
If no error occurs, status equals zero value. Otherwise, non-zero value is set.

get_area

request frame:
CMD_GET_AREA, unsigned long id, unsigned long off, unsigned long num

Parameters:
id
Identificator of II area component.
off
Offset from the beginning of II area component.
num
Number of bytes to get.

response frame:
CMD_GET_AREA, char status {, unsigned long num, char data[] }

Parameters:
status
If no error occurs, status equals zero value. Otherwise, non-zero value is set.
num
If no error occurs, num equals number of bytes which has got from II area component. Otherwise, this field is omitted.
data
If no error occurs, data equals value from II word component. Otherwise, this field is omitted.

6.3. IIE Compatibility

IIE is developed on Win32 and Solaris platform. There have not been noticed any differences in work on these platform.

7. BOOTING FPGA CHIP

Each FPGA chip must be configured to realize specified function before it can be used.
In general, there may be two sources of configuration data:

- on-board EEPROM memory; the chip is configured automatically when the power is switched on.
- External source (computer system), connected to the board using specific interface, for example Altera provides ByteBlaster and BitBlaster cables, and Xilinx provides Cable III and Cable IV.

In this system, JTAG protocol is used to load the FPGA configuration, but the hardware interface is not specified, it may be any channel configured in the system (currently LPT or VME).

Each board has at least one FPGA chip which is configured from the on-board EEPROM, this chip is responsible for the communication interface (channel). Other FPGA chips are loaded though this chip. The example topology is in the figure below:

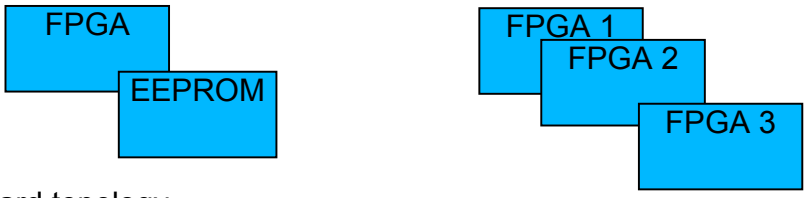

Fig. 8. FPGA board topology

All FPGA chips are visible in the same (channel) address space, but right after the board is switched on only the communication FPGA is accessible. All other chips must be loaded over communication with FPGA, this chip has special entries, it's address space – sets of bits - which represent the JTAG lines of other FPGA chips. Each configured FPGA chip is automatically visible in the channels address space beside the channel of FPGA entries.

At this moment the system accepts configuration data in JAM format, this format is native for Altera's Integrated Development Environment (Quartus). Xilinx tools exports configuration data to the SVF format, but Altera has provided a SVF to JAM converter.

JAM-Loading application (called Jambo) is based on the JAM-Player source codes published by Altera. Technically this is just another client of the system – it simply reads and writes JTAG bits from Channel FPGA's address space. Currently Jambo has been tested and is available on Win32 and Solaris environments.

8. EXAMPLE OF USING MEX-FUNCTIONS

In this section there is shown an example of using MEX-functions. Using the GUI tool in the Matlab, there was built a simple widow to show the possible way of using these MEX-functions.

This window shows how the IID file was parsed and how the registers of the FPGA device looks like. The window consists of two lists. The left list includes all of bits which are found in the TESLA SIMCON and they are accessible for the software. The second one presents all words. In this window there was used IIEngine function, which returns structure composed of bits, words and memory areas. Each element of structure was read from IID file. Thanks of that, we have got the map of the SIMCON elements, which can be write or can be read.

This window allows to read every element of the SIMCON, that value appears in BIT_VALUE or WORD fields. Functionality of this window is especially useful during testing of a new device. In an easy way, the behaviour of the new device can be observed or the developer can perform control operation step by step by putting the values in the appropriate registers.

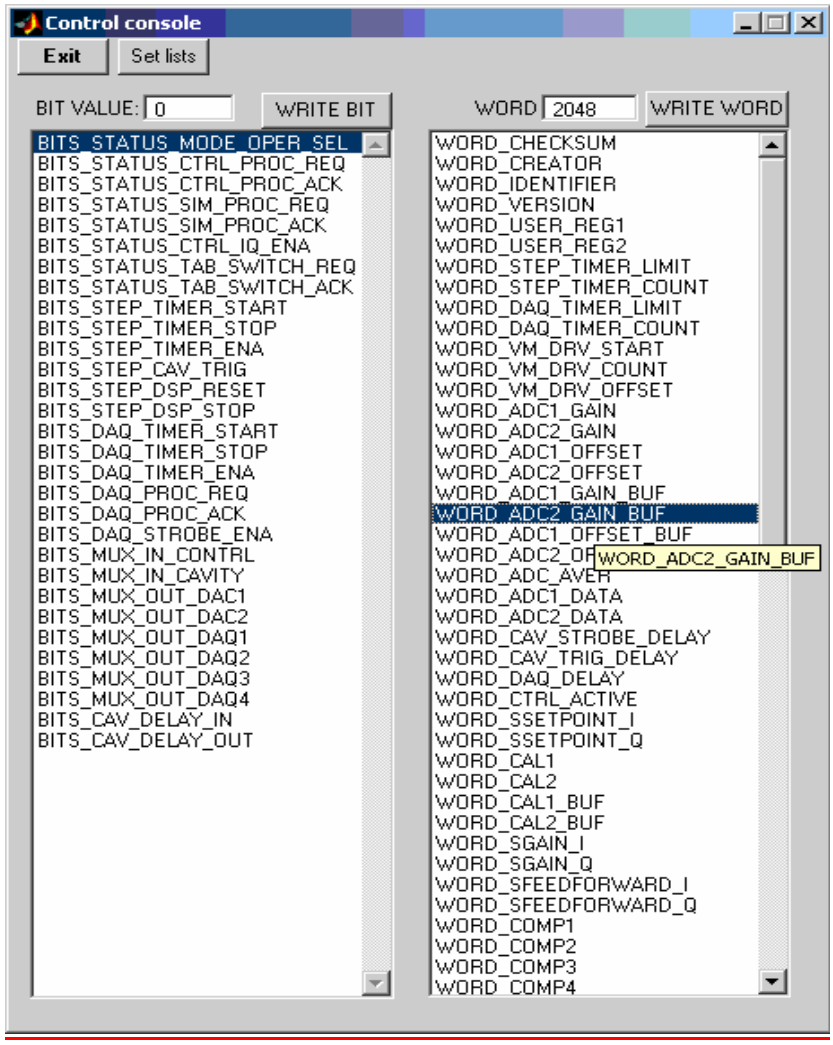

Figure 9. Test Panel

9. ACKNOWLEDGEMENT

We acknowledge the support of the European Community Research Infrastructure Activity under the FP6 "Structuring the European Research Area" program (CARE, contract number RII3-CT-2003-506395)

10. REFERENCES

[1] Poźniak K., Pietrusiński M.: “Internal Interface standard specification (version 1.0, February 2002)”

[2] Poźniak K. T., Czarski T., Rutkowski P., Romaniuk R. S.: „DSP integrated parameterized FPGA based Cavity Simulator & Controller for TESLA Test Facility SIMCON version 1.0 rev. 1, 02.2004”

[3] P.Rutkowski, R.Romaniuk, K.T.Pozniak, T.Jezynski, P.Pucyk, M.Pietrusinski, S.Simrock: “FPGA Based TESLA Cavity SIMCON DOOCS Server Design, Implementation and Application”, TESLA Technical Note, 2003-32

[4] K.T.Poźniak, M.Bartoszek M.Pietrusiński: “Internal Interface for RPC Muon Trigger electronics at CMS experiment”, Proceedings of SPIE, Photonics Applications II In Astronomy, Communications, Industry and High Energy Physics Experiments, Vol. 5484, 2004

[5] http://www.mathworks.com/ [Matlab Homepage]