View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by CERN Document Server

The LHCb Hostcard USB Interface

HUSBi

— Project Report —

Final Thesis for Engineering Studies at

IHA – Engineering College of Århus, Denmark

Technical Student: Sune Wolff

Supervisors:

Burkhard Schmidt, CERN – PH/DT2

Leif Munkøe, IHA

Submitted: December 20, 2006

CERN-THESIS-2007-009

20/12/2006

Abstract

The LHCb hostcard USB interface (HUSBi) card is a High Speed USB2.0 communication card used for data interchange and control in high voltage distribution systems for particle detectors at CERN, and particularly in the LHCb experiment. A primary high voltage power supply provides power to the detectors through a network of individual controllers. The HUSBi card is used to link the controllers with host PCs, permitting individual control of all the channels used to supply the detector.

The project includes the development of a Linux kernel device driver for the HUSBi communication card. This driver includes a security level that protects the hardware from manipulation that could damage the expensive detector equipment if the high voltage is incorrectly applied. This is done by monitoring each individual channel, and slowly ramping the voltage to the desired level.

This thesis includes an overview of the project, with focus on the methods and tools used throughout the course of the project. A description of the knowledge gained from the project is also supplied. Finally, suggestions for improvements or further developments are made.

Table of Contents

1. Introduction

Currently, four major experiments are being constructed at CERN - the "European Organization for Nuclear Research". One of these experiments, the LHCb[1] experiment, will use so called Multi-Wire Proportional Chambers (MWPC), in order to monitor a specific type of particles called muons. A large number of high voltage channels are needed in order to supply these MWPC. The channels are supplied by a high voltage distribution system, which distribute the output voltage from a primary power supply to several thousand individual channels. Companies such as CAEN have made a living out of producing such systems. As a more economic alternative, the joint effort of University of Florida (UF) and Saint Petersburg Nuclear Physics Institute (PNPI) has resulted in such a system used for high voltage distribution. Both the UF/PNPI system and a system from CAEN are used for the LHCb experiment, but the system of interest for this thesis is that from UF/PNPI.

1.1. The UF/PNPI High Voltage Distribution System

Figure 1 illustrates how the distribution of the high voltage is made in the UF/PNPI system:

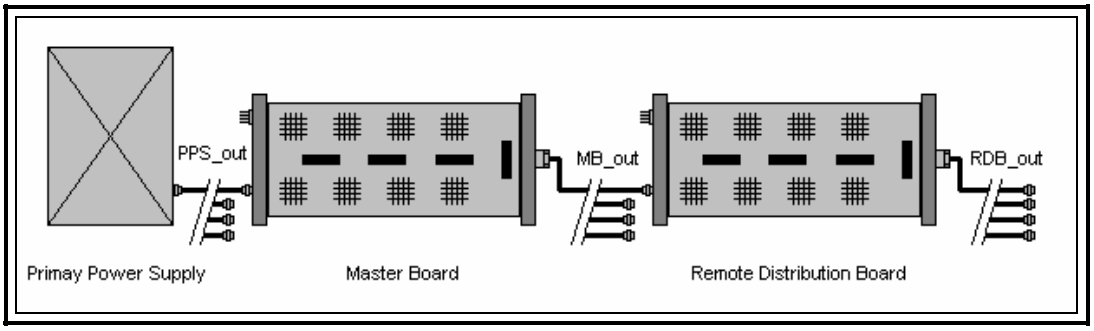

Figure 1: The UF/PNPI high voltage distribution system

[1] The "Large Hadron Collider beauty" experiment

The system consists of a Primary Power Supply (PPS), whose output voltage needs to be distributed to the many channels. This is done in two stages: A Master Board (MB), which has eight output channels, is connected to the output of the PPS. Each of these channels is connected to a Remote Distribution Board (RDB), each of which has 36 channels.

The system also provides both regulation and monitoring of the output voltage of the channels on the different modules. The MB can regulate the output voltage (MB_out) on individual channels between 0V and the input voltage supplied by the PPS (PPS_out). The RDB can set the output voltage (RDB_out) on individual channels to the input voltage supplied by the MB ±1000V, with a maximum of the input voltage (MB_out) and a minimum of 0V. The voltages set, the actual monitored voltage, and many other parameters are being made available to a control system by the means of a communication path:

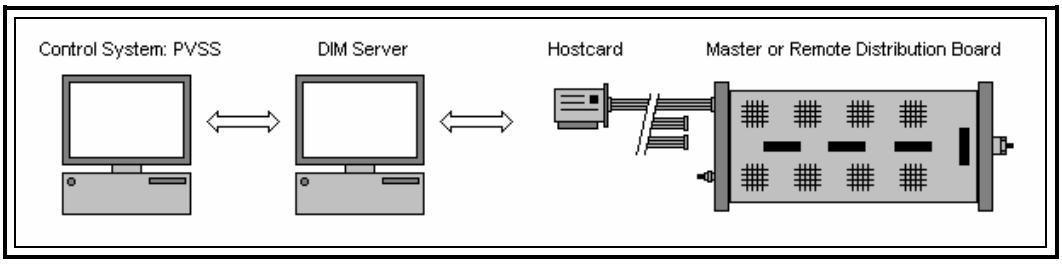

Figure 2: Communication in the UF/PNPI system

Each module (MB or RDB) needs to be connected to a so called hostcard, which is installed in a PC running Linux OS, providing the control functions. The hostcard communicates with a device driver, installed on the PC, over the PCI bus. The PC is also running a DIM [1] server which takes care of the monitoring and adjustment of the channels. This server is controlled from another PC using the program “PVSS” (Prozessvisualisierungs- und Steuerungs-System from ETM/Austria) which is the chosen SCADA (Supervisory Control and Data Acquisition) system at CERN.

1.2. The HUSBi Project

The UF/PNPI system is at present time fully functional and has been successfully tested. The fact that the existing hostcard uses the "old" PCI 5V bus for communication, however, could cause problems in the future. The industry standard PC platform is rapidly moving away from this bus towards high speed serial communication instead. As USB2.0 is being more and more widely used, it was decided to change the interface of the hostcard from PCI to High Speed[2] USB2.0. The abbreviation "HUSBi" is used to describe the project of creating a hostcard USB interface throughout this document.

The HUSBi project is split into two phases: an initial analysis phase called the "Proof of Concept" phase, and the development phase called the "Prototype Hostcard" phase. The objective of the first phase was to measure the achievable data rate of the USB2.0 communication. This result was to be compared with a calculated theoretical minimum required transfer rate for the UF/PNPI system. The result of this comparison would decide how to proceed in the following phase.

In the "Prototype Hostcard" phase, the development of the High Speed USB2.0 interface to the hostcard was implemented. This interface includes a USB2.0 device driver of the Linux PC, as well as hardware interface on the hostcard. The development of the device driver was the biggest challenge by far, and throughout this thesis the focus will be set on this part of the project.

[2] USB2.0 High Speed: 480Mbps

1.3. Purpose of HUSBi

Since the objective of the HUSBi project was to change only the interface of the hostcard, precautions were taken in order to make sure that the remaining parts of the system could be kept unchanged if at all possible. A lot of work was already put into the development of the PVSS controls system, the DIM server, and the production of the modules of course. These existing parts of the system define the boundary conditions of the HUSBi project, since the new USB2.0 device driver interfaces with these parts. The driver also needed to interface to a developer's board[3] which was used instead of the hostcard for the development of HUSBi. The purposes of the project are listed below:

- Change the interface of the hostcard to High Speed USB2.0;
- Keep the remaining parts of the UF/PNPI system unchanged if possible;
- Perform initial analysis of the data rate required by the system, and compare with the achievable data rate for USB2.0 communication;
- Implement changes to the architecture of the device driver, in order to increase the achievable data rate, or to reduce the required data rate;
- The USB2.0 device driver must have the same functionality as the existing PCI driver. This includes the interface to the DIM server along with the internal functionality of the driver; and
- In order to demonstrate the USB communication between the driver and hostcard, simple read and write procedures must be implemented in the FPGA on the developer's board.

As can be seen in the above list, the production of a new hostcard with USB2.0 interface is outside the scope of this project. This would include PCB design along with redesigning the code for the FPGA in order to make use of the new USB interface. With the manpower available, it was impossible to fit these additions into the timescale of the project.

[3] XEM3010 from Opal Kelly (www.opalkelly.com)

1.4. Glossary

The following terms are used throughout this report:

- **Altera:** FPGA manufacturer
- **API:** Application Programming Interface
- **Cypress:** Semiconductor manufacturer: cy7c68013a / cy7c68001
- **DIM:** Distributed Information Management system
- **EEPROM:** Electronically Erasable Programmable Read-Only Memory
- **Endpoint:** Unidirectional data pipe used for USB communication.
- **FIFO:** First In, First Out
- **FPGA:** Field Programmable Gate Array
- **FX2LP:** USB Transceiver cy7c68013a
- **GUI:** Graphic User Interface
- **HDL:** Hardware Description Language
- **ISE:** Integrated Software Environment
- **MB:** Master Board – part of the UF/PNPI system
- **Opal Kelly:** Small company from Oregon USA. Creators of the XEM3010 developer's Board and FrontPanel API
- **OS:** Operating System
- **PCB:** Printed Circuit Board
- **PNPI:** St. Petersburg Nuclear Physics Institute
- **PVSS:** Prozessvisualisierungs- und Steuerungs-System from ETM/Austria
- **RDB:** Remote Distribution Board – part of the UF/PNPI system
- **SCADA:** Supervisory Control and Data Acquisition
- **Spartan:** Low cost medium density FPGA family from Xilinx
- **SX2:** USB Transceiver cy7c68001
- **UF:** University of Florida
- **VHDL:** Very high speed integrated circuit HDL
- **wxWidgets:** Cross platform API for writing GUI applications
- **XEM3010:** Developer's board used for the HUSBi project
- **Xilinx:** FPGA manufacturer

2. Proof of Concept

Since the project description was not very detailed, some initial analysis of both the UF/PNPI system and the USB2.0 protocol was needed in order to create the requirements for HUSBi. It was also decided to implement a small demo project called "HUSBi ComTest", which could be used in order to test different methods to transfer data between the host PC and the hostcard.

Since the USB2.0 Specification [2] is an enormous and complex document, only a few key elements were analyzed in order to find the most suitable way to transfer the data. USB communication is done through "endpoints" which are unidirectional data pipes. Of the four different types available, HUSBi utilizes Bulk transfers which are commonly used for devices that need to transfer a lot of data, and which does not tolerate any data loss.

A calculation of the required data rate in the UF/PNPI system was completed. It was assumed that each read- or write execution would introduce a delay of a few milliseconds to the system [3], and that all data for one channel will be transferred in a single package. The required data rate has been calculated to 156.7Mbps. This result is based on a worst case scenario, so the actual required data rate might be lower, but it is still a very important boundary condition of the project.

2.1. HUSBi Communication Test

In order to increase development speed of this initial demo project, an XEM3010 developer's board from the company Opal Kelly was used as the hardware platform. It comes with an API to ease the access to the board from the host PC, and another API which is used in order to setup the endpoints in the FPGA on the XEM3010 in a simple way.

The objective of “HUSBi ComTest” was to test the maximum data rate by using different data transfer strategies. The main interest was to test the difference in data rate achieved by sending a curtain amount of data in one package, and sending the same data in several smaller packages. It was also of great interest to test if data could be transferred between the host PC and the FPGA on the XEM3010 without corrupting the data or loosing parts of it.

A user interface has been developed for “HUSBi ComTest”, which allows the user to perform different tests with several configurable parameters. The interface was implemented with the aid of a cross platform GUI library called “wxWidgets”.

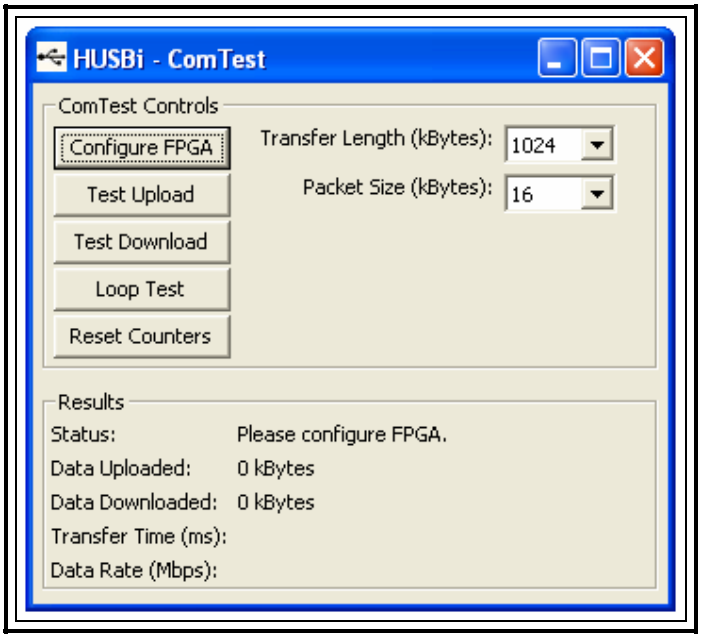

Figure 3: HUSBi ComTest user interface

HUSBi ComTest allows the user to configure the FPGA with a binary file and test the data rate achieved uploading data from the FPGA to the host PC or downloading data the opposite way. The total amount of data used for these tests is user configurable, as is the size of each package. If these two numbers are of equal size, only one package will be transferred.

A loop test can also be performed, where a package of random generated data is send from the host PC to the FPGA and back again. The received data is then compared with

the data originally send off in order to test if the data has been corrupted, or if parts of the data were lost during transmission.

2.2. Experience Gained from HUSBi Communication Test

Developing “HUSBi ComTest” gave a good introduction into the world of High Speed USB2.0 interfaces. By using the API provided with the XEM3010 developer’s board, communication between the host PC and the FPGA proved to be less difficult to implement than expected.

Working with the wxWidgets API was challenging. By using some of the more complex functions available, a sophisticated user interface with a genuine Windows look was created. Although the knowledge gained was not used to any further extent in the HUSBi project, it can still prove to be valuable in the future.

A lot of experience setting up project properties and compiler settings in Visual Studio was acquired from working with the two APIs mentioned above. Especially the wxWidgets API was very complex to link correctly to the project. Valuable experience in working with Visual Studio 2005 was gained by setting up the project for HUSBi ComTest.

On the hardware side, VHDL code was developed for the FPGA on the XEM3010. Even though the design was somewhat simple, a lot of experience was obtained from working with both VHDL language and with development tools from Xilinx such as “ISE WebPack 8.2i”.

2.3. Proof of Concept Results

Several tests were made in order to examine the achievable data rates for both upload and download. In these tests, 32MB of random generated data was transferred between the host PC and the FPGA on the XEM3010 developer's board. Different package sizes were used, in order to test the variation in achievable data rate. The results of these tests can be seen below:

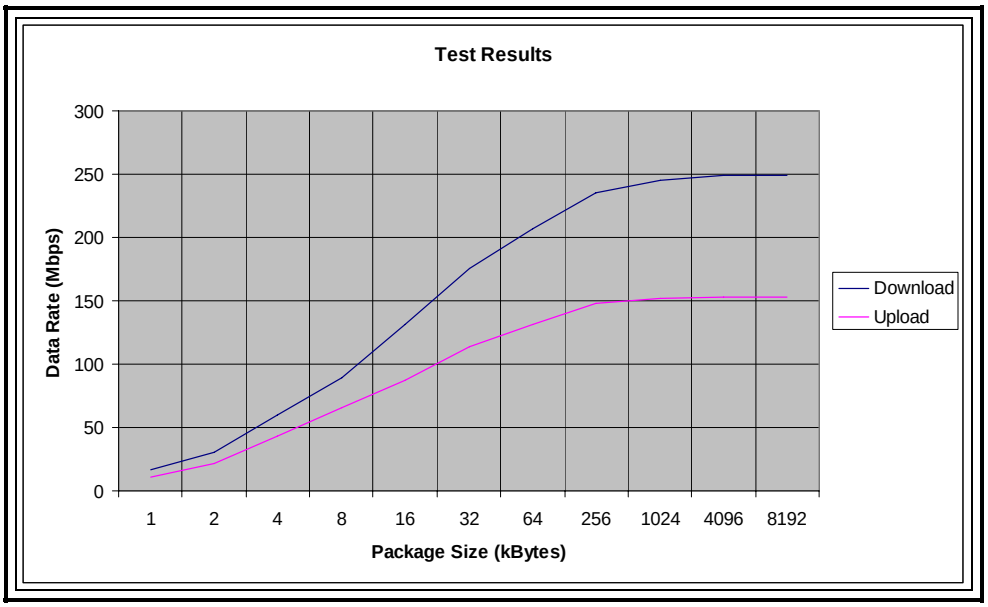

Figure 4: Test results from HUSBi ComTest

When the package size was increased the data rate also increased. This was true for both the upload and the download tests. In order to achieve the highest possible data rate, the number of transferred packages should be minimized. Optimally, all data should be send in one large package.

Another of the objectives for the "Proof of Concept" phase was to attain data rates of a minimum of 156.7Mbps. When downloading data into the FPGA this is easily achieved, but uploading data to the host PC this is not true. The difference in achieved upload and download data rate is tested and confirmed by Opal Kelly [4]. It was concluded that changes had to be made in order to reduce the amount of data which was read. This would introduce a much needed safety margin between needed and achieved upload data rate.

3. HUSBi Linux USB Device Driver

The main focus of the HUSBi project was to develop a Linux USB device driver for the "Prototype Hostcard". As the UF/PNPI system requires that the parameters of the hardware modules are read with 20ms intervals, a very reliable timer was needed. Only drivers running in the kernel of the operating system have 100% reliable timers available [5], so it was decided to implement the HUSBi USB device driver as a kernel module instead of a normal user application. This choice introduced some restriction on which files and libraries the driver could be linked to. This, unfortunately, meant that the APIs used in "HUSBi ComTest" could not be used.

3.1. Initial Analysis

Since no documentation was available on the hostcard PCI driver, and only very little on the DIM server [6], a lot of research and analysis was required prior to the development of the new driver.

3.1.1. Driver in Kernel Space

Developing a driver running in the kernel of the operating system is a complex task. Compared to normal user applications, none of the usual debugging tools are available, and if the kernel module is not running perfectly the entire OS will crash, and a reboot is needed. Knowledge of kernel driver development has mainly been obtained from the book: "Linux Device Drivers" [7], which includes in-depth explanation of all aspects of Linux kernel drivers, as well as small examples throughout the book.

In order to enhance error tracking of the system, a serial link has been set up between the Linux PC and the HyperTerminal of a standard Windows PC. All kernel messages are then displayed in the HyperTerminal, and are available even after the kernel of the Linux PC has crashed. This was an excellent addition to the development environment, which reduced the time spent debugging the system.

3.1.2. Driver Interface

It is important to have a firm understanding of how the different function in the driver interacts, and how the driver interfaces to the DIM server and the hostcard. The figure below illustrates this, where the items shaded dark grey are the hardware specific parts of the system:

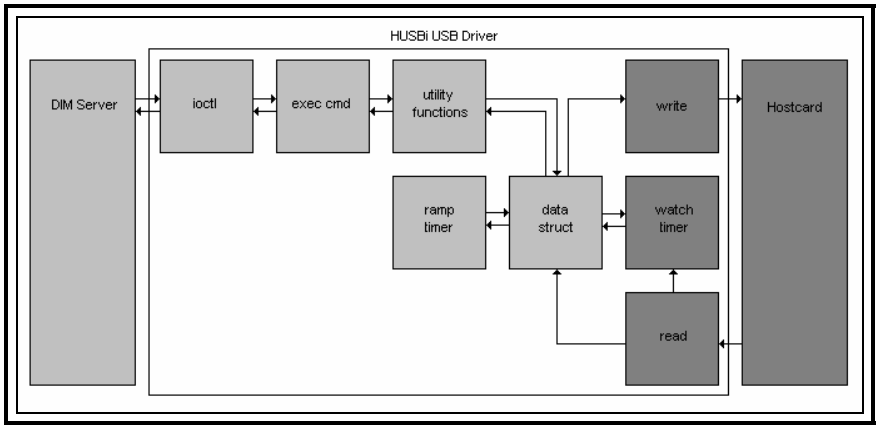

Figure 5: HUSBi USB device driver interface

3.2. Design Process

The main reasons behind the high data rate required for the UF/PNPI system are the many read and write operations used, each of which introduces a small delay. So in order to reduce the data rate, a new architectural principle is introduced in the HUSBi driver. Instead of reading the data of each channel, one at a time, all the data of all modules and channels attached to the hostcard is read at once, and is stored in an internal data structure on the host PC. The functions in the driver which used to access the hostcard directly, in order to read the values of the channels, will now find the data needed in this data structure instead. The result of this, is that only a single read operation is needed each 20ms. A new calculation was made which took these changes into account, resulting in a required data rate of only 2.84Mbps. This only includes the most important data, which is used to make sure that the detector equipment is not damaged. In addition, all the data for all channels needs to be transferred, but only every second as the DIM server only poll the driver very slowly (poll rate between 1 and 10 seconds). The data rate of 2.84Mbps is easily obtainable, and a huge safety margin between the achievable and required data rate is established in the system.

3.3. Implementation of the HUSBi Device Driver

In order to make the task more manageable, the implementation was divided into two separate parts. The first part is a simulator driver covering all the lightly shaded parts of Figure 5 including the interface towards the DIM server. This is followed by adding the hardware interface, which is shaded in a darker gray in the figure, for the final version of the driver. By implementing the driver in two steps, the interface towards the DIM server could be tested without any hardware attached, and only when this interface was working as intended the interface towards the hostcard was added.

3.3.1. Simulator Driver

Since the simulator driver does not serve any real hardware device, it is created as a *char* driver which is a suitable class for most simple drivers [7]. This type of driver can be used to serve a data structure instead of a physical device, which is exactly what is needed for the simulator driver. A lot of effort was put into ensuring that every possible command from the DIM server was executed correctly by the driver. When the server writes a value to the hostcard, the driver stores this value in the data structure instead. In a similar way, the driver will read a value in the data structure, whenever the DIM server requires an updated value.

3.3.2. HUSBi Device Driver

For the final version of the HUSBi driver, hardware support was added. The driver class was changed from *char* to *usb*, which involved quite a few changes to how the driver registers itself in the kernel. In the source tree of Linux 2.6.x a skeleton of a USB driver[4] can be found, which was used as inspiration for the HUSBi driver. This sample driver shows examples of implementation of registration of the driver, open/close function, read/write functions, removal of the driver, and much more. The combination of this sample driver, and the book “Linux Device Drivers” [7] made it possible to implement the HUSBi USB Device Driver, with a fully functional interface to the hostcard.

[4] Can be found in the source tree: /usr/src/linux2.6.x/drivers/usb/usb-skeleton.c

3.4. Experience Gained from the HUSBi Device Driver

A lot of experience has been gained from developing a USB kernel module. It is a task which people are normally advised to avoid if possible, as debugging a driver running in the kernel is very complicated. If the module crashes the kernel, the developer will often be left without any trace of what caused the crash. This section summarizes the valuable knowledge obtained from the work on the HUSBi Device Driver.

3.4.1. Kernel Driver

It quickly became obvious that developing a kernel module is something completely different than a normal user application. Even a simple task, such as allocating memory for the data structure, requires several instructions.

The biggest difference by far is the fact that the kernel module cannot be linked to any external libraries or header files. This means that only functions that are actually part of the kernel itself[5] may be used in kernel modules. The number of available functions increases with each new version of the Linux kernel, but it is still negligible compared to the amount of functions, libraries, and APIs available to a user application. This means that a lot of micromanagement and low level code is needed for a kernel module, opposed by the user application which just makes use of a function in an API.

A small example will illustrate this difference clearly: In order to write an array of data into the FPGA in "HUSBi ComTest", the function WriteToPipeOut() was called once, and the API took care of the rest. In the HUSBi driver, the hvcard_write() function is no less than 50 lines of code.

The fact that new versions of the Linux kernel are occasionally released is both good and bad. As explained above, the new versions introduce additional functions which can be used for kernel module development. Unfortunately the names of functions change from time to time and infrequently used functions are removed all together. This means that sample code found in textbooks or on the Internet can be somewhat

[5] These files can be found in the "include/linux" directory in the Linux kernel source tree

useless, as the functions used either has had their name changed or has been removed from the kernel source tree.

So one very important lesson learned, is always to make sure that the needed function is still present in the source tree. Additionally, it is a very good idea to read through the source code of the function, in order to confirm that the functionality has not been altered.

3.4.2. Debugging Tools

One of the biggest challenges faced with when developing a kernel module is error tracking and debugging the source code in general. Even the smallest error in the execution of the driver, will cause the entire kernel of the operating system to break down. A kernel crash is accompanied by a huge number of error messages, which are all logged in a file in the kernel. Unfortunately this log is erased then the kernel is restarted, which means that the original cause of the break will not be visible to the developer.

In order to work around this problem, a serial link was set up between the Linux host PC and a Windows PC. All kernel messages were copied to the HyperTerminal of the second PC. Since the HyperTerminal stores several of the latest messages received, the developer can examine all the kernel messages receives since the break down, in order to analyze which error caused the crash.

This addition debugging tool proved to be absolutely invaluable. During the development phase of the HUSBi device driver the kernel crashed a lot of times, and without the possibility to examine what caused the error, there would have been no way to properly debug the driver.

3.5. HUSBi Device Driver Results

The two different versions of the HUSBi device driver were tested separately. The simulator driver was tested on its own in order to make sure that the interface between the driver and the DIM server was working as intended. The final version was tested with the remaining system in the acceptance test, so most of the result mentioned here is from the test of the simulator driver.

3.5.1. DIM Server Interface

The interface between the driver and the DIM server was tested with the following test setup:

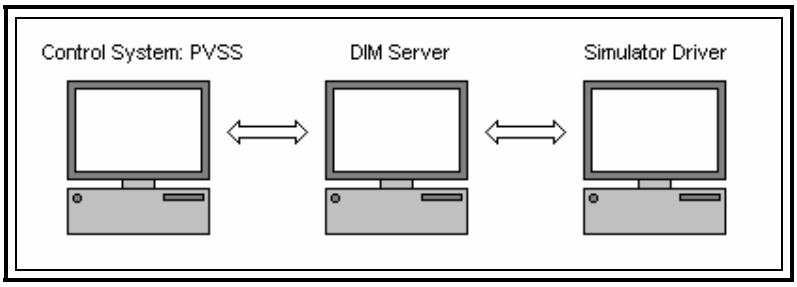

Figure 6: Simulator driver test setup

An entire system, including the simulated hostcard with a Master Board and Remote Distribution Board attached, was set up in the control system. All tests performed on the system had the desired outcome. The DIM server located the simulated modules, and all commands send to the driver were executed in a satisfactory manner.

3.5.2. Hostcard Functionality

Using the same test setup the functionality of the driver was also tested. This included automatic ramping the output voltage of channels, by the use of a kernel timer. It was also confirmed that the output voltage could not be set on a channel which was not actually turned on, as well as other similar tests. Of all the tests performed, none produced negative results, so it was concluded that the functionality of the old PCI hostcard driver has been successfully implemented in the HUSBi device driver.

3.6. Suggestions for Improvements

Even though all the functionality which is needed for the UF/PNPI system has been implemented, there is still room for improvements. This upgrading of the driver will mostly be kernel related though, as any changes to the functionality will require changes to the DIM server as well, which should be avoided unless absolutely necessary.

A common problem in kernel modules is concurrency problems, when different functions try to access the same data at the same time. This can be avoided by introducing the use of semaphores and mutexes (mutual exclusion) [7]. Data can be locked by a function by using these operations, and only unlocked once the function is finished using the data. Any other function which tries to access a piece of locked data will have to wait until the data is unlocked. This addition would make sure that no *race conditions* would occur when different functions access the data in the data structure.

One thing that has not yet been tested is what happens if the driver tries to allocate memory for the full data structure of one hostcard with 16 Remote Distribution Boards attached. The function which is used to allocate memory in the HUSBi driver is called kmalloc(), which is usually used for small amounts of memory. If larger quantity of memory must be allocated, it is usually better to use a page oriented technique. The size of a memory page differs from system to system, and is set by the variable *PAGE_SIZE* in the kernel. The size of a page can be anything between 4KB and 64KB, and a maximum of 1024 or 2048 pages (system depended) can be allocated. If the HUSBi driver ever should encounter problems using kmalloc(), memory allocation must be done on a “per page basis”.

4. USB Interface for the HUSBi Hostcard

In order to demonstrate that the HUSBi device driver can communicate with a hostcard over the USB2.0 bus, the interface had to be implemented on a hardware platform as well. This phase of the project includes hardware design for an FPGA, like the one which is on the existing hostcard. Even the newest and fastest FPGAs on the marked do not run at high enough clock frequency to receive a serial bit stream of 480Mbps which High Speed USB2.0 devices must support. For this reason, an external USB transceiver is used in order to de-serialize the received bit stream.

4.1. Choice of USB Transceiver

The choice of the right USB transceiver chip proved to be very difficult, but equally important. The initial choice was to use the FX2LP [8], which was on the XEM3010 developer's board used for "HUSBi ComTest". This device is a very complex chip, which includes on-chip memory, a Serial Interface Engine (SIE), and an 8-bit microcontroller on top of the actual USB transceiver. There is a huge selection of different setups [9], which can be used in order to change the way the FX2LP handles incoming data, but it quickly became apparent that this chip was a bit to complex for this project.

Another option was found: a demo board using an older USB transceiver called SX2 [10]. This chip is basically the same transceiver, as the one inside the FX2LP, but without all the additional on-chip components. It acts like a FIFO with the USB to one side, and a 16bit data bus to the other. Using simple control signals, the FIFO can be filled with data, and read or written once the FIFO is full. The FPGA on the demo board was from Altera, and not Xilinx as the one on the existing hostcard. By using a high level design language, it should be possible to export the hardware design between different FPGA types.

4.2. The Design Process

In order to ensure that the HUSBi device driver will recognize the SX2 demo board, a successful *enumeration* process between the device and host PC must be carried out. The datasheet for the USB transceiver [10] describes this process in fine details, along with any additional setup of the device, which needs to be configured before USB communication can be established.

It was decided to limit the hardware design to support only a simple read request, as this would be sufficient to fulfill the objective of demonstrating the USB communication. If the SX2 receives a read request from the host PC, a counter will start feeding data into a buffer. Once this buffer is filled, the SX2 will send the data to the requester.

4.3. VHDL Implementation

The code for the FPGA on the SX2 demo board is implemented as a big state machine, written in VHDL in order to ensure portability to other FPGAs. This was an important goal in order to avoid limiting the choice of components used in the future PCB design for the HUSBi card.

A colleague assisted in the programming of the Altera FPGA, as he had the required software available as well as great experience in working with the SX2.

4.4. Experience Gained

During the comparison of available USB transceivers, an important lesson was learnt: the most complex device available is not always the optimal solution. As it turned out, using the SX2, which is the more simple option, introduced just the right functionality needed in order to implement simple USB2.0 communication. The more complex FX2LP can be configured in a large number of ways, in order to optimize performance for a specific system. But for a simple USB link, the SX2 has more than adequate functionality.

The VHDL implemented on the SX2 demo board, is a lot more complicated than the code used in “HUSBi ComTest”. With the more complex code, it is important to remember that processes can run in parallel inside the FPGA. This is true due to the fact that the code is implemented at gate level in the FPGA, meaning that different tasks can run completely independently of each other. This can be a big advantage, but can also cause a lot of problems when being used to the more normal sequential execution of code. It was a great experience to work with a skilled VHDL designer as Paschalis Vichoudis, and a lot of knowledge has been gained from him.

4.5. Results

The enumeration process between the host PC and the SX2 demo board is working as intended, meaning that the HUSBi device driver is automatically loaded when the board is connected to the PC, and unloaded again when the board is disconnected. In "USB language" this type of device is called a *hotplug device*. This adds a lot of flexibility to the use of the device, and could for example mean that no keyboard is necessary for the host PC once the HUSBi device driver is installed.

The read request from the host PC is successfully supported by the SX2 demo board. This demand comes from user space, from the DIM server for example. A connection between the host PC and the SX2 are established, and the read request is send to the FPGA. The read data is made available in the FIFO of the SX2, and the host PC reads the data. Finally, the data is send back to user space by the HUSBi driver.

4.6. Suggestions for Improvements and Further Development

Since the focus of the project is set on the development of the HUSBi device driver, a lot of work needs to be done on the hardware side, before a new hostcard with High Speed USB2.0 interface can be produced.

In order to reduce the amount of data which needs to be transferred between the host PC and the hostcard each 20ms, only the utmost necessary data will be transferred. The result of the calculation of required data rate in section 3.2 is based on only having to receive the most important data. Since the DIM server will have to access data regarding the hostcards, the entire data structure will have to be updated from time to time. Since the DIM server supports very slow controls (poll rate can be set between 1s and 10s), the update of the full data structure will not have to happen very often.

Since both the SX2 and FX2LP has four user configurable endpoints, it would be optimal to set up two endpoints (one IN and one OUT) to be used for transferring the smaller status structure each 20ms. The two remaining endpoints (again one IN and one OUT) can be set up in order to transfer the full data structure each second. This way, the required data rate is still kept low, and all the data will periodically be updated within the timing required by the system.

For the PCB design, several suggestions have been made. The new hostcard will need an FPGA, an EEPROM, and a USB transceiver as the most important components. The leading FPGA manufacturers are Xilinx and Altera, and both companies provide medium density, low cost FPGAs, which is sufficient for the new hostcard. The final choice must be made on the basis of what is available, and maybe a preference based on the experience of the developer.

The choice of EEPROM will be dictated by the choice of FPGA. The datasheet of each type of FPGA provides a list of EEPROMs which the FPGA supports, so it will just be a matter of choosing one which is big enough for the configuration file for the FPGA.

The final choice of USB transceiver will be very difficult. The choice will depend on a lot of things, most importantly if the time is available to develop the firmware for the microcontroller on the FX2LP, which controls how the USB communication is carried out. The obvious choice would be to use the SX2, but it needs to be considered if any of the configuration possibilities of the FX2LP will be missed.

If it is decided to use the FX2LP as the USB transceiver, a very good option would be to incorporate the XEM3010 developer's board into the new PCB design. The fact that the board can be used as a piggy-back board[6], will ease the task of the new PSB design tremendously as the FPGA, EEPROM, and USB transceiver hardware already is on the XEM3010.

Finally, there have been suggestions as to the final layout of the hostcard. In order to implement as much functionality as possible, the following proposal has been made:

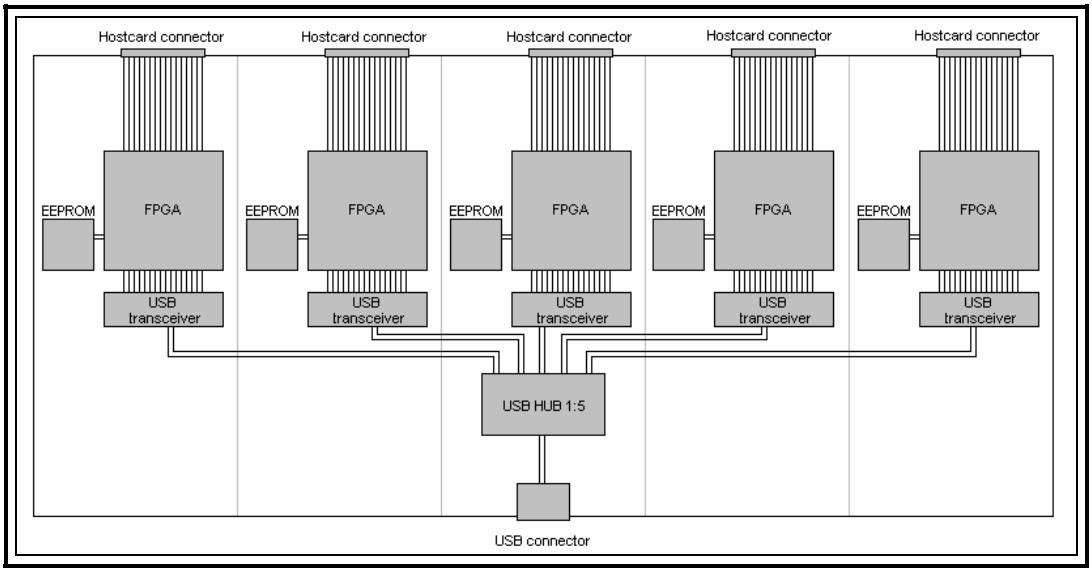

Figure 7: Suggestion of final layout of HUSBi hostcard

This suggestion includes five hostcards in a single module, and uses an on-board USB HUB in order to enable communication to all the FPGAs from just a single USB connection to the host PC.

[6] A small printed circuit board, that plugs into another board in order to enhance its capabilities.

5. Conclusions

The HUSBi project was undertaken in order to change the interface of an already existing communication card, used in a high voltage distribution system, from PCI to High Speed USB2.0. This was done in order to prepare the system for the future, where multiple PCI slots will not be standard installations in PCs. The changes made in order to upgrade the interface include a USB Linux kernel device driver, and USN interface on an FPGA.

A high priority was to ensure that the number of changes which needed to be made to the existing system was kept to an absolute minimum. This was achieved by keeping the naming convention of both functions and commands used in the device driver unchanged. Some major architectural changes have been made to the structure and data flow of the driver, in order to ensure that the communication with the attached hardware could be carried out without encountering any timing problems. The USB interface to the FPGA was created by the use of an external USB transceiver along with some VHDL code for the FPGA.

The interface towards the software elements of the existing system works according to the objectives of the project, and it has been demonstrated that the USB communication between the driver and hardware is working as well. So, all the basic requirements of the project has been met, even though there is still a lot of work to be done before a complete new communication card can be manufactured.

The development of the HUSBi card and its associated software has been very challenging as a lot of so far unknown subjects were covered throughout the course of the project. This also made certain that the work was very interesting.

References

[1] Presentation on DIM; provided on the CD

[2] Compaq, HP et al. *Universal Serial Bus Specification*, Revision 2.0, 2000

[3] Private conversation with W. Bialas, 2006

[4] Opal Kelly Incorporated, *FrontPanel™ User's Manual*, Revision 20060703, 2006

[5] Private conversation with A. Madorsky, 2006

[6] Magnus Lieng, *A short documentation of the UF PNPI DIM Server*, 2006

[7] Allesandro Rubini et al, *Linux Device Drivers*, Third Edition, 2005

[8] Cypress Semiconductor Corporation, *EZ-USB FX2LP™ USB Microcontroller*, Revision 38-08032, 2006

[9] Cypress Semiconductor, *EZ-USB technical Reference Manual*, Version 1.4, 2006

[10] Cypress Semiconductor Corporation, *EZ-USB SX2™ High-Speed USB Interface Device*, Revision 38-08013, 2004