A Data Acquisition Board Firmware for Pellet Tracking System

Filza Saleem
Abstract

A Data Acquisition Board Firmware for Pellet Tracking System

Filza Saleem

The PANDA experiment requires precise information about the target position in the beam [3]. The target in form of frozen hydrogen 30μm droplets called "pellets" moving at a speed of 60m/s is tracked by a system consisting of 8 line scan CCD cameras sending data to "Camera Link Boards", where the data are merged and sent further to a data acquisition system for storage and subsequent analysis [15].

The aim of the thesis is to replace the current bulky Data Acquisition system with just one board ZYNQ Board constructed by P. Stamatakopoulos, G. Ntounas and P. Marciniewski [16]. In this thesis, the firmware is implemented in the ZYNQ Board having a Xilinx ZYNQ XC7Z030-1FBG484C. The ZYNQ Board receives data delivered by Camera Link Boards over four optical links running at 2Gb/s. The Data Acquisition system serves for data merging and retransmission over 1 Gb/s Ethernet.

For implementing of the Data Acquisition system, the following three steps are followed. In the first step a custom IP is designed in Xilinx Vivado, which is responsible for receiving data from camera link boards, merging the data and storing them in data buffers. The IP also sends push-back signals to camera link boards for stopping the data flow. In the second step DAQ firmware is designed in block diagram using Xilinx Vivado IP Integrator. The purpose of DAQ firmware block design is to send the buffered data to ARM based Processing System (PS) of the ZYNQ via AXI-4 bus. In last step Linux Operating System is installed on the ARM processor for receiving data in ARM processor and sending data to a PC through Ethernet. The Firmware was tested first in the laboratory using an Integrated Logic Analyzer of Xilinx Vivado and then in the real environment of the The Svedberg Laboratory in Uppsala.
Acknowledgement

I would like to thank my supervisor Pawel Marciniewski for his support, patience and guidance throughout my thesis work. It would not have been possible for me to complete my thesis and thesis report without his guidance and support.

Many thanks to my reviewer Leif Gustafsson for giving me the opportunity to be part of this project. I want to thank him for the comments that he gave me on my report which helped me improved my report. I would also like thank him for the knowledge that he gave us during VHDL course during my master’s degree.

Last but not least, I would like to thank my family for their support and trust during my whole studying career. Without them, I would not have reached this point.
Table of Contents

Chapter 1 ............................................................................................................................... 5
  1.1 Introduction .................................................................................................................. 5
  1.2 Experimental Environment ....................................................................................... 6
  1.3 PANDA Experiment .................................................................................................... 7
  1.4 Uppsala pellet target station (UPTS) .......................................................................... 8

Chapter 2 The Data Acquisition for the Pellet Tracking .................................................... 11
  2.1 Current status of the hardware: .................................................................................. 11
  2.2 The Firmware of the Camera Link board ................................................................. 12
  2.3 The Firmware of VME DAQ Board .......................................................................... 13
  2.4 New concept ............................................................................................................... 14

Chapter 3 ZYNQ-Board The Stand-Alone DAQ interface ................................................. 15
  3.1 Hardware concept ...................................................................................................... 15
  3.2 ZYNQ system ............................................................................................................. 16
  3.3 GTX ............................................................................................................................ 17
    3.3.1 8B/10B Encoding Protocol ................................................................................... 17
    3.3.2 K Character .......................................................................................................... 19
    3.3.3 GTX Transmitter .................................................................................................... 19
    3.3.4 GTX Receiver ........................................................................................................ 20

Chapter 4 DAQ evaluation ................................................................................................ 21
  4.1 Tools .......................................................................................................................... 21
  4.2 Implementation of DAQ ............................................................................................. 21
    4.2.1 Input Data Format ............................................................................................... 21
    4.2.2 The Structure of the firmware ............................................................................ 22
  4.3 The data Packaging IP ............................................................................................... 25
  4.4 Creating DAQ firmware with IP Integrator .............................................................. 26
4.4.1 Design for Bare-metal Applications ................................................................. 27
4.4.2 Design for Linux Application ........................................................................... 28

Chapter 5 Linux for ZYNQ-a step by step process .................................................. 31
5.1 Xilinx Zynq-Linux .............................................................................................. 31
5.1.1 Boot Sequence ................................................................................................. 31
5.1.2 Building and Booting Linux ............................................................................. 33
  5.1.2.1 Fetch Sources: ........................................................................................ 34
  5.1.2.2 Creating and Building Device tree ............................................................ 34
  5.1.2.3 Building U-boot ...................................................................................... 36
  5.1.2.4 Building Linux ....................................................................................... 38
  5.1.2.5 Building Ramdisk .................................................................................... 40
  5.1.2.6 Creating BOOT image .......................................................................... 40
  5.1.2.7 Booting from SD card ............................................................................. 42
  5.1.2.8 Creating an Application ......................................................................... 43
  5.1.2.9 Running an Application ......................................................................... 44
5.2 Networking ......................................................................................................... 45

Chapter 6. Tests .......................................................................................................... 46
6.1 Tests with Zed Board .......................................................................................... 46
  6.1.1 Bare Metal .................................................................................................... 46
    6.1.1.1 Hello world .......................................................................................... 46
    6.1.1.2 FIFO Reading ........................................................................................ 47
  6.1.2 Linux ............................................................................................................. 48
    6.1.2.1 Hello world .......................................................................................... 48
    6.1.2.2 Fifo Reading .......................................................................................... 48
6.2 Tests with Zynq-Board ....................................................................................... 49
  6.2.1 Bare Metal .................................................................................................... 50
6.2.1.1 Bare metal Loopback test
................................................................. 50
6.2.1.2 Bare Metal Readout with Experimental Setup
............................................................... 51
6.2.2 Linux
........................................................................................................... 53
6.2.2.1 Loopback with Linux Implementation
........................................................................ 53
6.2.2.2 Linux Implementation Readout with Experimental Setup
.............................................. 53

Chapter 7 Conclusion and future work
...................................................................................... 57

Chapter 8 References
........................................................................................................... 58
Chapter 1.

1.1 Introduction

Data acquisition is the process of gathering real physical data and converting them into digital form and storing data on files. It plays a very important role in experimental physics. Large particle detectors, such as the PANDA, under construction at GSI, Darmstadt, Germany, consist of many detector subsystems, responsible for data streaming. Hence, multiple data sources, such as digitizers, are distributed over a large area of the detector system [3].

The data generated in medium and large experiments such as PANDA are gathered by complex data acquisition systems (DAQ), typically consisting of many heavy and hardly portable devices as well as platform-specific software. Such DAQs are not convenient for testing partial systems or devices under development, especially when the development and tests are shared between different collaborating groups of scientists, where DAQ portability is essential.

In 2015 Georgios Ntounas and Panagiotis Stamatakopoulos designed a hand-held DAQ board as a practical part of their master thesis. The device was called ZYNQ Board, since it is based on ZYNQ SoC from Xilinx [4]. The design was later corrected by Pawel Marciniewski and a pilot series of the ZYNQ Board was produced. The modules will be used in small experiments, where among others they will substitute the existing bulky VME-based DAQ for the Pellet Test Station, a relatively little part of the PANDA experiment.

The aim of this thesis project is to design the Data Acquisition firmware for the ZYNQ Board, which will serve for data merging and re-transmission over 1Gbit Ethernet. The firmware should be implemented on a Xilinx ZYNQ XC7Z030-1FBG484C. The ZYNQ Board receives data from a front-end device over four optical links running at 2 Gbit/s and buffers it. The optical data link as well as most of the system are 32-bit word oriented. The buffered data are then sent over an AXI-4 Bus to an ARM-based Processing System of the ZYNQ. In the ARM processor a Linux operating system is used.

The ZYNQ Board delivers a push-back signal stopping the data sources, if the DAQ is unable to read the data at the rate of the source data for a longer time. The push-back in form of a BUSY signal prevents data corruption caused by overfilling the FIFO buffers.

The most difficult issue to be solved in this thesis is to get the ZYNQ Board to operate under an open Linux operating system. The documentation about the procedure of Linux installation is very scares, the implemented SoC device is lacking supporting files and problems are being discussed on internet forums. A part of the thesis contains a step-by-step description of the Linux installation procedure, which is a result of a broad survey and an own research of the author.
1.2 Experimental Environment

In accelerator based physics, experiments are roughly divided into collider experiments and fixed target experiments. In collider based experiments, particles move in opposite direction at very high energy. At a beam crossing point, opposite momenta of colliding particles cancel each other, hence the Centre of mass of mass point is at rest in the laboratory system.

In a fixed target experiment, the particle beam is aimed at a large and stationary target. Such a stationary target might be solid, liquid, or gas. A detector measures the charge, momentum, mass, etc. of the resulting particles. An example of this process is the PANDA (anti-Proton ANnihilations at Darmstadt) experiment where the fixed target material used is hydrogen.

To have hydrogen as target material in hadron physics experiment requires a cryogenic installation. Frozen hydrogen is the solution that provides highest density of the target material, good vacuum conditions and minimum of structural material in the interaction region. The target could be either pellets or a cluster jet. Pellets are small spheres of frozen hydrogen. They are produced at some distance from the interaction region, above the particle detector system, and can travel inside thin pipes. A cluster jet target contains clusters of various number of gas molecules. Among these two solutions pellets provide better luminosity – the rate of interactions is higher because the density of matter in a pellet stream is larger than in a cluster jet, which implies a higher probability of interaction. Also, pellet targets provide better sustained vacuum conditions. Last, but not least, the pellet target provides a possibility of knowing the position of single pellets at any moment [14].

Both of the mentioned target types are planned to be used in the PANDA setup. The pellet-target has been developed for the WASA_{at\_COSY} experiment in Uppsala in 1995 [2] and was in operation in the WASA experiment in Forschungszentrum Julich (FZJ) in Germany until 2015. Right now, the Uppsala group is doing development for the PANDA experiment, for this purpose a copy of the WASA pellet target was built at The Svedberg Laboratory, Uppsala. This target is a standalone system. It is called the Uppsala Pellet Test Station (UPTS) and is used for target development and studies, like the pellet tracking system and Data acquisition firmware presented in this work.
1.3 PANDA Experiment

The PANDA Experiment will be one of the future experiments at the Facility for Antiproton and Ion Research (FAIR) [3]. The facility is currently built on the area of the GSI Helmholtzzentrum für Schwerionenforschung GmbH (GSI) in Darmstadt, Germany. At FAIR the antiproton storage ring HESR which will deliver an antiproton beam with a momentum range of 1.5 GeV/c - 15 GeV/c. The storage antiproton in HESR collide with a fixed target (i.e. hydrogen pellets) inside the PANDA detector. To meet the experimental requirements, the interaction point must be measured with the accuracy expressed in micrometers. The pellets of the size of 30 micrometers and moving in a vertical vacuum pipe at a velocity of 60 m/s will be tracked optically by 4 sets of cameras. Each camera set will consist of 2 lasers illuminating the pellets and 2 CCD line-scan cameras observing the reflected light from the pellets in X-Y coordinates. 2 sets of cameras spaced by 20 cm in height will be located ca 2 m above the anti-proton beam and 2 sets ca 2 m below the beam. The signals from the cameras will be transmitted to a set of Camera Link Interfaces, where the information will be pre-processed. The processing results in form of pellet amplitude, position and time will be transmitted via an optical link to the Data Acquisition Board.

Figure 1.2. Setup of PANDA experiment.
1.4 Uppsala Pellet Target Station (UPTS)

The UPTS is an independent pellet target/generator built for the PANDA detector experiment. The UPTS is almost similar to the WASA with minor changes. A picture of the UPTS is shown in figure 1.3.
Pellets are illuminated with lasers installed and are observed with line scan cameras. Figure 1.4 shows the cameras installed for Pellet detection. The raw data collected by the cameras are sent to a camera link interface for preprocessing, and later sent to the data acquisition board (ZYNQ Board used in this project).

Figure 1.4. Cameras setup for Pellet tracking

Figure 1.5 shows the principle of operation of the pellet tracking system. The red area shows the interaction region where the pellet stream and the accelerator beam overlap. The size of the interaction region is a few millimeters. In order to know the interaction point more precisely, each pellet is tracked to know their position accurately at the time of an interaction. The tracking will be accomplished using lasers and fast line-scan cameras. The cameras, placed at different levels along the pellet stream, will measure pellet’s position and time of passage (figure 1.5). Then, the information from many cameras will be put together to identify and reconstruct the track of each pellet. Information about the pellet position in the interaction region at the time of an interaction will be used in the analysis of the experimental data [14].
Figure 1.5. Principle of operation of a pellet tracking system [14]
Chapter 2. The Data Acquisition for the Pellet Tracking

2.1 Current status of the hardware:

The data from Line Scan Cameras are transmitted to a Camera-Link Board via a serial data communication protocol called camera link.

In 2013 Madhu Sudhanna Rao Thelajala and Geng Xiaoxiu performed their master’s thesis project at Uppsala university called Hydrogen Pellet Detection using Single-line CCD Cameras. This project was done for Camera-Link Board in which Spartan 6 FPGA was used. The design was later modified by Pawel Marciniewski and is now in use at Uppsala Pellet target station.

The camera-link board is programmed to detect pellets and parametrize the pellet events according to an algorithm designed using a VHDL (Hardware Description Language). The processed data on the camera-link board is transmitted to a VME DAQ board by an optical fibre for further processing. The optical-fibre link is connected to a GTP (Gigabit Transceiver Protocol) transceiver of the two FPGA boards. The VME DAQ board has totally 16 optical link ports and it supports reading multiple processed camera data from several camera-link board boards. The VME DAQ board is equipped with a VME (Versa Module Europa) bus to transfer the pellet data to a computer. The data flow scheme of the readout system is shown in Figure 2.1.

Figure 2.1. The data flow scheme of the read out system [15]
2.2 The Firmware of the Camera Link board

In Figure 2.2, the firmware overview for the Camera Link board can be seen. In the Deserializer block, the data coming from the four links of the cameras are deserialised and formatted. For deserializing, an IP cores SERDES (SERialization and DESerilization) provided by Xilinx are used. In deserialization the serial data are converted into parallel form and are then stored into a Deserializer_FIFO.

Next step is to find out at which pixel (position of the pixel) the hydrogen pellet passed the camera and also find out the amplitude of the shadow from the pellet. This is done in a two stage automatic pedestal extraction. Every pixel of the camera shows a pedestal amplitude depending on the ambient conditions.
light. One has to correct this effect in order to detect real pellet events efficiently. The two stages conf0 and conf1 of the automatic pedestal correction generate pedestal values for each single pixel and store these values in a block memory. Once the automatic pedestal correction process has been completed, a flag is set to enable the pellet recognition part. This block also contains a timestamp generator for each valid camera line. The recognized pellet events are stored in the corresponding FIFOs. A clocking is implemented for the above part and the slow control part. After this, a GTP (Multi-gigabit Transceiver Protocol) part transmits the pellet events serially using 8b/10b coding technology to the VME DAQ board. Additionally, a slow control part is implemented for four cameras to configure the cameras and to configure the readout. Configuration readout data are stored in the corresponding FIFO. Configuration readout data are also sent to the VME DAQ board by using the GTP part. All the code components explained above are instantiated and connected to each other in the top level entity file called Cam_link_int4.vhd. The top level entity also contains an instantiation of a Xilinx ChipScope instance. The chipscope components are generated using the Xilinx LogiCore generator [15].

2.3 The Firmware of VME DAQ Board

The simplified firmware architecture of the ATLB/Data Concentrator is shown in 2.3. The device consists of two physical boards: a VME Base board and an Optical Transceiver Mezzanine board. The boards are connected using two 120-pin connectors, providing a Local Bus interconnection.

![Figure 2.3. Overview of the firmware for VME DAQ Board](image)

The Optical Transceiver Mezzanine receives data from up to 16 front-end devices over optical links running at 2 Gbit/s and stores the data in individual 16kB Link FIFOs. The optical data links as well as most of the system are 32-bit word oriented. In the Event Builder module, the received data are time sorted, framed and send over the Local Bus to the Data FIFO on the VME Base board. During the process of data receiving and sorting a number of data consistency checks is performed to detect errors in the DAQ earliest possible. Depending on the experiment, different data sorting algorithms are used.
A Slow Control System provides access to on-board data sorting parameters as well as to front-end devices. The ATLB/Data Collector can send control commands to the front-end devices over the optical links, working in full duplex mode.

A Trigger and Synchronization module allows for simultaneous use of multiple ATLBs in the DAQ. This module provides a push-back signal stopping the data sources, if the DAQ is unable to read the data at the rate of the source data for a longer time. The push-back in form of a BUSY signal prevents data corruption caused by overfilling the FIFO buffers.

A Control/Status register allows for a direct VME access to the status of the data FIFO. Polling it is particularly useful for initiation of efficient VME block transfers.

### 2.4 New Concept

In this project the ZYNQ Board will be connected to the camera link board which is connected to cameras. The Pellet information will be received form the camera-link board into the ZYNQ Board for further processing and retransmission over Ethernet. So the aim is to replace the VME DAQ Board with the ZYNQ Board.

![Figure 2.4. Data flow from camera-link board to ZYNQ Board](image)
Chapter 3. ZYNQ-Board The Stand-Alone DAQ interface

3.1 Hardware concept

The heart of the ZYNQ-board system is the XC7Z030-1FBG484C SoC FPGA, which combines an ARM processor along with a Xilinx 7-series FPGA. In order to achieve the desired optical communication, the board features four front-end optical data converters, along with the NIM standard, which provides the means of communication with the front end system [4]. The data are transferred from the board to a PC via Gigabit Ethernet or USB. For the Ethernet communication, Marvell’s 88E1518 Gigabit Ethernet transceiver is used, while the USB communication is being implemented using SMSC’s USB3320 transceiver. For more complex data processing or extended data storage, space is required. 512 MB of RAM memory and a Micro SD slot are implemented. The board also supports a video interface via a ADV7511 HDMI transmitter that is shown in Figure 3.1. Figure 3.2 shows the ZYNQ Board.

![Figure 3.1. block diagram of ZYNQ System.](image1.png)

![Figure 3.2. ZYNQ Board](image2.png)
3.2 ZYNQ System

The most important component of the board is a Zynq-7000 All Programmable System-On-Chip (SoC) from Xilinx, thus it is worth mentioning some related background information about it.

These Integrated Circuits (IC) combine an ARM dual Cortex-A9 processor system with Xilinx 7-series FPGA. Thus, they offer the flexibility of an FPGA along with the processing power of an ASIC. The internal ZYNQ architecture comprises two parts: The Programmable Logic (PL) equivalent to an FPGA and the Processing System (PS), which is the ARM processor [4]. They can be used independently, powering down the part that is not used, or together, exploiting the Advanced eXtensible Interface (AXI) industry standard to interconnect them. The PS includes not only the ARM processor but a collection of processing resources, composing the Application Processing Unit (APU), as well as memory interfaces and peripheral interfaces. In figure 3.3 a block diagram of the PS is presented.

![Figure 3.3. The ZYNQ Processing System](image)

The I/O Banks are divided into PS and PL Banks. PS Banks are groups of I/Os that are connected directly to the processing system, while the respective PL are those groups of pins connected to the FPGA fabric. The PL Banks are further divided into High Range (HR) and High Performance (HP) Banks. The HR are compatible with a wide variety of I/O standards (up to 3.3Volts), while the HP are used for high-speed interfaces. The Zynq-7000 family comprises several general purpose devices, with different features and characteristics. Table 3.1 summarizes some of their basic characteristics of the ZYNQ family members.
Table 3.1. Zynq-7000 family members

<table>
<thead>
<tr>
<th>Processor</th>
<th>Dual core ARM Cortex-A9 with NEON and FPU extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. processor</td>
<td>800MHz</td>
</tr>
<tr>
<td>clock frequency</td>
<td>1GHz</td>
</tr>
<tr>
<td>Programmable Logic</td>
<td>Artix-7</td>
</tr>
<tr>
<td>No. of Flip-Flops</td>
<td>35,200</td>
</tr>
<tr>
<td>No. of 6-input LUTs</td>
<td>17,000</td>
</tr>
<tr>
<td>No. of 36Kb Block, RAMs</td>
<td>60</td>
</tr>
<tr>
<td>No. of DSP48 slices (18x25 bit)</td>
<td>80</td>
</tr>
<tr>
<td>No. of SelectIO, Input/Output, Blocks</td>
<td>HR: 100,HP: 0 HR: 150,HP: 0 HR: 200,HP: 0 HR: 100,HP: 150 HR: 212,HP: 150 HR: 250,HP: 150</td>
</tr>
<tr>
<td>No. of PCIe Express Blocks</td>
<td>-</td>
</tr>
<tr>
<td>No. of serial transceivers</td>
<td>-</td>
</tr>
<tr>
<td>Serial transceivers maximum rate</td>
<td>- 6.25Gb/s - 6.6Gb/s, 12.5Gb/s 6.6Gb/s, 12.5Gb/s 10.3Gb/s</td>
</tr>
</tbody>
</table>

3.3 GTX

In this project, the GTX (Gigabit transceiver) transceiver module is used for communication between the camera-link board and the ZYNQ Board. GTX is a high speed serial communication technology. The GTX is a SerDes (Serializer/Deserializer) capable of supporting a line rate from 500 Mb/s to 12.5 Gb/s and is a power-efficient transceiver.

3.3.1 8B/10B Encoding Protocol

In order to transmit data over the high speed serial interface it is required to encode the data before transmission and decode it after receiving to minimize the error during transmission. The GTX IP core provides the option to select the 8B/10B encoding scheme. It is used to achieve DC-balance and bounded disparity to allow reasonable clock recovery. In this scheme two bits are added per byte which means 32 bits are converted into 40 bits.

For 8B/10B encoding every byte (8-bits) data are divided into two groups, one of them has 3-bits and the other has 5-bits. The 3-bits group is encoded into 4-bits and the 5-bits group is encoded into 6-bits. The order of transmission is that 4 bits are sent before the 6 bits. The figure 3.4 clarifies the concept of 8B/10B encoding.

As can be seen in Figure 3.4 that two additional bits are added called “running disparity” or “RD” to ensure that the number of 1 and 0 transmitted are same. To achieve this, the encoder always calculates the difference between the number of 1s transmitted or received and the number of 0s transmitted or received, and at the end of each character transmitted or received, makes the difference either +1 or −1. The Figure 3.5 shows the wizard of the GTX core where 8B/10B encoding is selected.
Figure 3.4. 8B/10B encoding scheme

Figure 3.5. GTX transceiver wizard showing 8B/10B encoding selection
3.3.2 K Character

The 8B/10B scheme uses some special character for control. This special character is called k character. The k character selected is a comma that is “BC” in hex. TXCHARISK port is used to indicate if data on TXDATA are K characters or regular data. TXCHARISK is high for a byte if is a k character.

As can be seen in the figure 3.6 that the transmission is 4 bytes and an idle character is defined as x"000000BC", where “BC” is coded as “K” character. This is also a skip/add sequence used in the elastic buffer for synchronization of the transmitter and receiver.

![Figure 3.6. GTX transceiver wizard showing Idle character [5]](image)

3.3.3 GTX Transmitter

The GTX transmitter block diagram is shown in figure 3.7. The GTX transmitter consists of two main parts: Physical medium attachment layer (PMA) and physical coding sublayer (PCS). The PMA is used for data serializing and transmission and the PCS is used for data coding and adjusting. Data can be transmitted through the GTX by writing it to TXDATA port on the positive edge of TXUSRCLK2. The width of the TXDATA is selected to be 32 bits. The data after writing to the TXDATA are encoded according to 8B/10B encoding scheme and adjusted internally by the GTX IP core and is transmitted out through TXN_OUT and TXP_OUT [5].
3.3.4 GTX Receiver

GTX receiver is shown in blocks in figure 3.8. Receiver like transmitter has two parts PMA and PCS. Data are received serially through differential ports RXN_IN and RXP_IN into the PMA where it is then converted into parallel by the SIPO (serial input parallel output) block. Then in the PCS the polarity of the RXN_IN and RXP_IN are checked and if by accidently swapped are corrected. After that as the 8B/10B encoding scheme was used in the transmitter, so the data are decoded and finally received into the FPGA in RXDATA port which is selected to be 32 bits wide [5].

Figure 3.8. GTX receiver block diagram
Chapter 4. DAQ Evaluation

4.1 Tools

Tools used for this project are Xilinx Vivado and Xilinx Software Development Kit (SDK).

Xilinx Vivado provides the opportunity to create a project in VHDL or Verilog, synthesize it, implement it and then a bit stream can be generated. The PL (Programmable Logic) can be programmed with the generated bit stream. Xilinx Vivado also provides tools for creating a new IP from the existing project. The IP integrator of the Xilinx Vivado let us create the block diagram project. The Logic analyzer is an IP provided by Xilinx Vivado can be used to analyze the internal signals of the design.

The Xilinx Software Development Kit (SDK) let us create Bare-metal application and an application for Linux that is it let us create applications for non OS (Operating System as well as system that runs Linux).

4.2 Implementation of DAQ

Implementing the DAQ three main steps were followed

1. PL part of the project was created in Xilinx Vivado where the project was implemented in VHDL code. The created project was then packed in an IP.
2. The DAQ firmware was created in a block diagram using Xilinx Vivado IP integrator where the IP created in step 1 is used.
3. The project created in step 2 is exported to SDK where an application is created for the PS of the ZYNQ board. The application is created as a Bare-metal or Linux Application depending on if there is Linux running in PS or not.

4.2.1 Input Data Format

Cameras are connected to the Camera Link Interface, which evaluates and sends the pellet information to the ZYNQ Board over optical links. The links are using 32-bit data words, divided into 22 bits data payload data, 8-bit control data and 2 bit data validity flags, see figure. This allows for control of the hardware settings in real time.

Data transfers are separated by idle sequence words (x”000000BC”), where the last character (x”BC”) is a comma character K28.5.
When the information sent from the camera link board is not an idle sequence, the “Valid data bit acknowledge” is “1” instead of “0”. The lower 8 bits make “BD” instead of “BC” as shown below.

Single Pellet information is contained in five words. The following table shows the order of the words received.

The camera link board can also be configured to send the raw data. The structure of the raw data are that first Pellet information is received in the above format followed by raw data. Raw data start with “FE” as shown below.

### 4.2.2 The Structure of the Firmware

The ZYNQ Board has four optical links available but a single link is used in this project which makes it simple to handle the incoming data. The ZYNQ Board is connected to the Camera-Link Board and received in the Pellet information via GTX. The GTX IP core is provided by the Xilinx Vivado used in this project with the line rate of 2GB per second and a reference clock at 125 MHz.

The signals that enter the GTX can be seen in figure 4.1. Q0_CLK1_GTREFCLK_PAD_N_IN and Q0_CLK1_GTREFCLK_PAD_P_IN are the clock signals going in the GTX. There is another clock signal DRPCLK_IN going into the GTX. Some SFP control signal can also be seen in the figure 4.1. TXN_OUT and TXP_OUT are differential signals for serially transmitting data from GTX. The data are received in the GTX serially through RXP_IN and RXN_IN which is then converted to parallel 32 bits and is received in “gt0_rxdata_i” vector.

In order to filter out the pellet data from the idle sequence and to store the pellet information in a FIFO, another component named “DATABUFF” is declared and used in the top
VHDL file. The data after received in the “gt0_rxdata_i” vector is fed into the “DATABUFF” component in a “DATA_IN” vector which is 32 bits wide as can be seen in figure 4.1.

![Figure 4.1. Overview of Firmware (On PL)](image)

The data in the “DATABUFF” is then filtered out and checked if the data received is a comma character or is the pellet information. Figure 4.2 shows a four state Finite State Machine (FSM) which is used for detecting the comma character in the incoming data.

The FSM starts initially at state “00” and move to next state that is “01” if the DATA_IN is x"000000BC" and then move to next if again the data received is a comma. As more than one state is used for the comma character in order to make sure that comma is not sent further. Then from state “10” FSM moves to state “11” if the DATA_IN is not x"000000BC" and also trans_n is not ‘1’.

It is clear from the data format that the bits DATA_IN [0] and DATA_IN [1] are the flags showing that the pellet is found and slow control status. trans_n is a signal used for detecting that the incoming signal is not x"000000BC" and the bits DATA_IN (1 downto 0) are not zeros.

After that the data are written into a 32 bit wide and 2048 deep FIFO. The FIFO is provided as an IP core from Xilinx. Wr_clk is 50MHZ (GTX clk). Write enable wr_en is enabled whenever the FIFO is not full and also the data received is not a comma character and the Pellet found signal bit is ‘1’. The FIFO is then read in the Linux OS where the data are to be sent through Ethernet.
The FIFO also has an almost_full signal which is used for stopping the camera interface from sending more data. When the almost_full signal is asserted then “0x00001001” is sent out.

Figure 4.1 shows the signals that are connected to the PS. FIFO_DATA_OUT, fifo_Aempty and fifo_RD_EN are connected to PS via the axi_gpio. The FIFO_DATA_OUT is 32 bits wide and transfers Pellet information to the PS. The Fifo_Aempty signal notify the PS when there is no data available to be read. The Fifo_RD_EN signal is used to enable the reading of the Pellet information.

As can be seen in figure 4.1 that fifo_RD_EN signal is controlled by the processor. The signal is controlled by PS axi_gpio. It was observed that the axi_gpio could not read continuous stream of data, it was skipping values in between. So to overcome this the read enable (fifo_RD_EN) has to be enabled and then disabled before the next value to read. The FSM in figure 4.3 shows how this is implemented. fifo_RD_EN is the signal that is controlled from the PS and rd_en_delay is actually enabling the FIFO reading.

The FIFO also has an almost_full signal which is used for stopping the camera interface from sending more data. When the almost_full signal is asserted then “0x00001001” is sent out.

Figure 4.2. FSM for filtering data from idle sequence

Figure 4.3. FSM for FIFO read enable
4.3 The Data Packaging IP

The project created in section 4.2 is then packed and an IP is created which is further used for accessing the buffered Pellet information in the PS of the ZYNQ Board. Figure 4.4 shows how an IP is created from the current project. Figure 4.5 shows the created IP which is further used.

*Figure 4.4. IP creation wizard*
The IP integrator of Xilinx Vivado is used for creating the whole design of the DAQ firmware in block diagram, as the IP integrator lets us use block diagram to configure IP that is related to the PL and the Zynq-7000 AP SoC device processor (PS). It sets the PL information in an XML file, INIT files, which are then used by the SDK to create and configure Board Support Package (BSP) libraries, infer compiler options, define JTAG settings, and automate other operations that require information about the hardware [6].

Figure 4.6 shows the flow which is followed for creating a project in the IP integrator.

**4.4 Creating DAQ Firmware with IP Integrator**

![Diagram](Image)

*Figure 4.5. Pellet IP*

*Figure 4.6. Flow of the project designed using IP integrator*
4.4.1 Design for Bare-metal Applications

Bare-metal Applications were created before creating Linux bootable image and Linux applications for the zed-board and Zynq Board in order to learn communication of data between PL and PS through AXI4 bus.

Zed board – defined board BSP:

When a Zedboard Evaluation kit is selected while creating a project, it provides additional information like I/O peripheral configuration, pull-up register information etc to the Xilinx Vivado. That is why zed board was used to for learning before using the Zynq Board.

As the Zedboard does not have optical links so for testing communication between the PL and the PS, An IP was created which had a FIFO. A loopback was running and writing some predefined values in the FIFO. Same signals were coming out of the IP and controlled in the PS.

ZYNQ Board

When a project is created in Xilinx Vivado and an xc7zc030fbg484 zynq chip is selected, it does not provide extra information as was provided in case of zedboard. So, I/O peripheral has to be configured in this case, which can be done by double clicking the ZYNQ7 Processing System IP. Figure 4.7 shows window opened for customizing ZYNQ7 PS IP where uart1, GPIO and SD0 are selected. It is also important to select the right DDR memory type.

![Figure 4.7. Window for customizing zynq PS IP](image)

27
After selecting configurations, the design of the DAQ firmware is created for the ZYNQ Board. The IP created in section 4.3 is used.

4.4.2 Design for Linux Application

The same designs were used for the Zed board and ZYNQ Board as were created in the previous sections.

The Linux boots from an SD card as was mentioned previously that ZYNQ Board has to be configured, so it is important to disable the pullup registers for the SD card. Ethernet 0 is also selected because it is required for transferring data from PS to computer.

Figure 4.10 shows the DAQ firmware design created for the ZYNQ Board.

![Figure 4.8. Wizard for customizing PS: disabling pullup registers](image)
Figure 4.9. Wizard for customizing PS: selecting DDR memory
Figure 4.10. DAQ firmware design for Zynq Board for Linux OS
Chapter 5. Linux for ZYNQ-a step by step process

Xilinx provides the following distributions of Linux that can run on Xilinx devices.

- **PetaLinux**: PetaLinux is a Linux embedded operating system which includes a Linux kernel, as well as a complete configuration, build, emulate and deploy tools for Xilinx devices (Zynq SoC, MicroBlaze soft microprocessor). PetaLinux comprises of three constituent parts [11]:
  1) A fully customizable embedded Linux OS for Xilinx devices.
  2) Pre-built binary images (which are bootable out-of-the-box).
  3) PetaLinux SDK

- **Xillybus – Xillinux**: Xillinux is a desktop distribution of Linux that will allow you to run a full graphical desktop environment on the Zedboard; a keyboard and mouse can be attached via the USB On-The-Go port of the Zedboard, while a monitor can be connected to the provided VGA port. As well as providing a full Linux distribution, Xillybus also supply a development kit for interaction between the Linux host and peripherals running on the programmable logic. This is in the form of FIFOs on the logic side, and native Linux file operations on the host side. The Linux distribution, which is based upon the popular desktop distribution of Ubuntu 12.04 Long Term Support (LTS), and the development kit are available for free download from the Xillybus website [11].

- **Xilinx Zynq-Linux**: Zynq-Linux is an open source OS freely available from Xilinx. It is based on the 3.0 Linux kernel from kernel.org and includes a number of additions from Xilinx, such as a BSP and specific device drivers. Support is also provided for SMP, allowing the kernel to use both CPUs, and can also be configured to use a single CPU [11]. Open source Linux is used in this project.

5.1 Xilinx Zynq-Linux

5.1.1 Boot Sequence

When the zynq device powers on, the FSBL (first stage boot loader) initializes the PS (Processing System) of the zynq device by running a function called ps7_init. It then programs the FPGA that is PL (programmable logic) with the bit stream and hands over the program counter to U-boot.

U-Boot is a bootloader that is capable of loading the Linux kernel, root file system image and device tree from the flash, SD card and even the Ethernet via bootp, DHCP and TFTP.
The device tree which is loaded by the U-boot contains the specification and the addresses of different blocks of the hardware. Linux kernel uses device tree data for three major purposes [7]:

1) **platform identification**: The device tree is used by the kernel for identification of the specific device used.

2) **runtime configuration**: Device tree is used for passing configuration data like kernel parameters and location of the initrd image (root file system) during run time. Most of this data are contained in the chosen node (which is in the device tree) which look like this:

```plaintext
chosen {
    bootargs = "console=ttyPS0,115200 root=/dev/ram rw earlyprintk";
    stdout-path = "serial0:115200n8";
};
```

The bootargs property contains the kernel arguments.

3) **device population**: After the board has been identified, and after the early configuration data has been parsed, then kernel initialization can proceed. At some point in this process, the device tree and some other functions are called for setting up the device like setting up the interrupt handling etc.

Linux kernel will then mount the root file system

Root file system is the environment which contains all the applications. Two formats are used for root file system
- initrd (ramdisk.image.gz)
- initramfs (initramfs.cpio.gz)

initrd is used in this project.

Then the Linux drivers are loaded and inserted into the kernel. So the Linux is then ready for user application to be run on it.

The block diagram of the boot sequence of the Linux is shown in figure 5.1.
5.1.2 Building and Booting Linux

Figure 5.2 shows the block diagram of building and booting Linux on Zynq UltraScale+ MPSoC & Zynq-7000 AP SoC [8].
5.1.2.1 Fetch Sources:

Fetch sources describe how to obtain the sources from internet resources that are required for building U-boot, Linux, Device tree and Ramdisk(root file system). Linux and U-boot can be obtained through git by executing the following commands [8].

```
git clone https://github.com/Xilinx/linux-xlnx.git
git clone https://github.com/Xilinx/u-boot-xlnx.git
git clone https://github.com/Xilinx/device-tree-xlnx.git
```

The above commands will create new directory for each source and download in it.

Ramdisk for the arm system can be downloaded directly from the following link: http://www.wiki.xilinx.com/Build+and+Modify+a+Rootfs

5.1.2.2 Creating and Building Device tree

Zedboard:

All the device trees which are provided by xilinx and are downloaded by in the process of fetch sources are available in the directory `linux-xlnx/arch/arm/boot/dts`. The device tree for zed-board is available with the name “zynq-zed.dts”. As in this device tree no boot arguments are given, so before creating the binary of this device tree, the following boot arguments were added in the choose node

```
bootargs = "console=ttyPS0,115200 root=/dev/ram rw earlyprintk";
```

For creating the device tree for zed-board the following command is run on the terminal in the linux-xlnx directory. The binary file will be created in the directory linux-xlnx/arch/arm/boot/dts with name “zynq-zed.dtb”.

```
make ARCH=arm zynq-zed.dtb
```

![Figure 5.3. Compiling device tree for Zed Board](image)

ZYNQ Board

As the device tree is not provided by the Xilinx for every ZYNQ device, so it needs to be created for the ZYNQ Board. For creating the device tree, one needs to export the project created in Xilinx Vivado to the SDK and launch SDK. Before creating the device tree, the repository device-tree-xlnx
which is already downloaded in the fetch sources is added first. A repository in SDK can be added as
follow: [8].

SDK Menu: Xilinx Tools → Repositories → New... (<select device-tree-xlnx>) → OK

Figure 5.4 shows how to add a repository. After the repository is added a device tree can be created. For
device tree creation:

→ Finish

A new window of BSP settings will appear where the bootargs section is filled with
“console=ttyPS0, 115200 root=/dev/ram rw earlyprintk”. After the process is complete the device
tree will be created with the name “system.dts” as shown in the figure 5.5.

![Figure 5.4. Adding repository for creating device tree](image.png)
Now the device tree can be compiled and a binary can be created by running the following command on the terminal in the same directory where the “system.dts” resides.

```
dtc -O dtb -o system.dtb -b 0 system.dts
```

### 5.1.2.3 Building U-boot

Since all the compilations are done for a different architecture (ZYBQ devices) so the environment variable called CROSS_COMPILER needs to be set before building the U-boot. The CROSS_COMPILER can be set by executing the following command.

```
export CROSS_COMPILE=arm-ixilinx-linux-gnueabi-
source <Xilinx Tools installation directory>/SDK/2015.4/settings64.sh
```

The cross compiler tool that is arm-ixilinx-linux-gnueabi- is part of the Xilinx tools and is installed with it while installing Xilinx Vivado.

After the cross compiler is set, U-boot can be compiled by going into the u-boot-xlnx directory which was created while fetching sources and telling the makefile the configuration of the zynq chip for which the u-boot needs to be compiled. For Zed board.

```
make zynq_zed_config
```

Figure 5.6 shows the screenshot of writing the configurations.
Figure 5.6. Writing the Zedboard configurations

In the u-boot-xlnx/configs directory are all the available configurations but the configuration for ZYNQ Board is not available there. So the configuration file was created with the name “zynq_zc7030_defconfig” by copying the content of the “zynq_zc706_defconfig” and replaced the device tree with the one which was created for ZYNQ Board. Now in order to make the device tree available for these configurations copy the device tree and its binary (system.dts and system.dtb) in the u-boot-xlnx/arch/arm/dts directory.

Now the configurations can be written for ZYNQ Board as

```
make zynq_zc7030_config
```

After the configuration is written then u-boot for both ZedBoard and ZYNQ Board can be compiled as follow.

```
make
```

After the build process is done the u-boot image is created. Figure 5.7 shows the screenshot of the terminal after the u-boot is created. The u-boot is created in the same directory. The u-boot image should be renamed from “u-boot” to “u-boot.elf” because extension is required for creating boot.bin image. The Boot.bin image creation is explained later. The u-boot image can be seen in the figure 5.8.

Figure 5.7. U-boot image created
In addition to the u-boot, another image called “mkimage” is also created in the tools directory. Mkimage is used for other tasks so it is important to make the mkimage available for those tasks. This can be done by adding the path of mkimag to $PATH environment variable [8].

```
cd tools
export PATH=`pwd`:PATH
```

### 5.1.2.4 Building Linux

Before creating the image, the kernel needs to be configured. Some default configurations are available in the directory `linux-xlnx/arch/arm/configs` with the name “xilinx_zynq_defconfig”. Default configurations can be set going into the `linux-xlnx` directory and performing the following command. Figure 5.9 shows the configurations set after performing the following commands [8].

```
make ARCH=arm xilinx_zynq_defconfig
```
If some changes in the kernel default configurations are required, it can be done by manually configuring the kernel.

```make
make ARCH=arm menuconfig
```

![Figure 5.10. Window for menuconfig](image)

After the configuration is complete the kernel image can be produced as follow.

```make
make ARCH=arm UIMAGE_LOADADDR=0x8000 uImage
```

After the process is complete, uImage will be created in the directory linux-xlnx/arch/arm/boot. Figure 5.11 shows the screen shot taken after the ulmage was created.

![Figure 5.11. ulmage created after building kernel](image)
5.1.2.5 Building Ramdisk

The uramdisk.image.gz is created by running the following command on the terminal in the same directory where the arm_ramdisk.image.gz lies.

```
  mkimage -A arm -T ramdisk -C gzip -d arm_ramdisk.image.gz uramdisk.image.gz
```

![Figure 5.12. Building root file system](image)

5.1.2.6 Creating BOOT image

After the Vivado project for blinking the led for the zed board block design is complete and bit generation is successful, the hardware design is exported to the SDK (software development kit) which is further used in creation of the FSBL. The project can be exported to SDK as

file → export → export hardware .

Bitstream was selected with it. Then the SDK was launched as

file → launch sdk.

As FSBL (First Stage Boot Loader) is a template which is available in the SDK. So After creating new project in SDK the FSBL template is selected. While creating the FSBL the operating system platform should the “standalone” and the hardware platform should be the one which was imported from Vivado. Figures 5.13 and 5.14 show the creation of FSBL.
Figure 5.13. Wizard for creating new project in SDK

Figure 5.14. Selecting FSBL template
The boot image can be created either by right clicking on FSBL and selecting created boot image or by clicking on Xilinx tools and then selecting create boot image. In creating the boot image the FSBL.elf, bitstream and u-boot.elf are selected from their respective locations. The FSBL.elf should be added as a bootloader and the rest of the files are added as datafile. Figure 5.13 shows the creation of boot image.

![Figure 5.13. Wizard for creating BOOT.bin](image)

### 5.1.2.7 Booting from SD card

In order to boot from the SD card, the jumpers of the zed-board should be configured. Figure 5.16 shows how the jumpers are set on the zed board. After configuring the jumpers copy the images (BOOT.bin, ulimage, dvicetree and uramdisk.image.gz) into the SD card and place it in the zed board. The zedboard is turned on and is connected through the UART as shown in figure 5.17 where all the booting messages can be seen. Figure 5.18 shows the screenshot after the Linux is booted up.

![Figure 5.16. Zed Board jumper settings](image)
5.1.2.8 Creating an Application

A new project is created in SDK in which the Linux operating system is selected. As the axi-gpio IP is used in the vivado project, so for accessing the IP in Linux from user space two ways can be used [9]

- Direct access to device registers via /dev/mem
- User Space I/O (UIO) framework.
The first method is used in this project for accessing the IP for blinking the led on zed board. In this method an IP is accessed from Linux by opening `/dev/mem` and using `mmap()` to map a view of the device’s physical address space into our process’s virtual address space. Figure 5.19 shows the addresses of IPs in the hdf file which is created while importing the hardware to the SDK. The project is then saved and built. As a result the output file is created with the extension of “.elf”.

```c
/* Open /dev/mem file */
fd = open("/dev/mem", O_RDWR);
if (fd < 1) {
    perror(argv[0]);
    return -1;
}
/* mmap the device into memory */
page_addr = (gpio_addr & (~(page_size - 1)));
page_offset = gpio_addr - page_addr;
ptr = mmap(NULL, page_size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, page_addr);
```

![Address Map for processor ps7_cortexa9_0](image)

*Figure 5.19. hdf file showing addresses of IPs used in project*

### 5.1.2.9 Running an Application

For running an application on the Linux, the executable file which was created as a result of creating and building the project in SDK needs to be transferred to the Linux implementation from the system where the executable file resides. The following ways can be used for adding a file to Linux.

- FTP file transfer
- UART file transfer
- SSH file transfer
- Copying file into the root file system image

SSH file transfer is used in this project. For this method Ethernet connection is made between the zed-board and the local host (PC). IP addresses are assigned to both the Zed-board and the PC. The IP address: 192.168.1.11 and netmask: 255.255.255.0 is assigned to PC whereas IP address: 192.168.1.10 and same netmask is assigned to the Zed-board and ZYNQ Board. Figure 5.20 shows the screen shot of assigning an IP address.

![Figure 5.20. Assigning IP address to Zedboard](image)

After the connection is created, SSH is used for transferring files.

```
scp filza@192.168.1.11:<path to the executable file> <destination directory >
```

If the current working directory is already the destination directory on the terminal, then use dot(.) instead of destination directory. The destination directory here in this project is “/tmp/”.

After the file is transferred successfully, the executable file can be run from the directory where file is transferred.

## 5.2 Networking

In order to achieve the aim of transferring the data from Linux OS running on ZYNQ Board through Ethernet to a PC, socket programming is used. “A Socket is an end point of communication between two systems on a network. To be a bit precise, a socket is a combination of IP address and port on one system. So on each system a socket exists for a process interacting with the socket on other system over the network” [11].

TCP/IP model is used in this project. In TCP/IP model, one system would be client and other would be server. Client would send the request to the server for the connection and the server would accept the request. After the connection is established, both the systems can send and receive messages to each other. Server could have more than one client. Figure 5.21 explains all the steps that are needed for client-server communication.
Chapter 6. Tests

6.1 Tests with Zed Board

The tests and their results that were carried out for the zedboard are explained below.

6.1.1 Bare Metal

Projects like Hello world, blinking LED and FIFO Reading were created for zedboard. Bare-metal applications were created for such projects as it does not require any Operating System (OS).

6.1.1.1 Hello world

The first and basic step was to test if I can get a hello world. Figure 6.1 shows the hello world result.
6.1.1.2 FIFO Reading

For FIFO reading as explained a FIFO IP was created first which was then read in the PS through GPIO. A two dimensional array was declared having 15 values which are 32 bits wide. These values were written into the FIFO and then read out. Figure 6.2 shows the screenshot of the terminal in SDK where FIFO was read and displayed.

type RomType is array(0 to 15) of std_logic_vector(31 downto 0);
--signal ROM : RomType :=
constant my_Rom : ROMType := (
  0 => x"A00000bc",
  1 => x"A00000bc",
  2 => x"a00000bc",
  3 => x"a0000000",
  4 => x"a0000011",
  5 => x"a0000022",
  6 => x"a0000033",
  7 => x"a0000044",
  8 => x"a0000055",
  9 => x"a0000066",
 10 => x"a0000077",
 11 => x"a0000088",
 12 => x"a0000099",
 13 => x"a0000100",
 14 => x"a0000111",
 15 => x"a0000122");
6.1.2 **Linux**

After the Linux OS was booted in the Zedboard, the above projects were also tested in Linux OS.

6.1.2.1 **Hello world**

![Hello world run in Linux on Zedboard](image)

6.1.2.2 **FIFO Reading**

Figure 6.4 shows the screen shot when the FIFO was read in the Linux OS using Zedboard.
Figure 6.4. FIFO Read in Linux on Zedboard

6.2 Tests with ZYNQ-Board

Multiple tests were performed on the Zynq Board as explained below.
6.2.1 Bare Metal

Bare metal applications were used to test the DAQ firmware.

6.2.1.1 Bare metal Loopback test

Predefined values are looped back to test the DAQ design. Figure 6.5 shows the chip scope result where 32 bits’ data are transmitted through a GTX transceiver. It also shows the TXCHARISK as high when the data transmitted is a k character (x"000000BC") as explained in section 4.1.1.2.

![Figure 6.5. Chip scope result showing data transmission through GTX](image1.png)

Figure 6.5. Chip scope result showing data transmission through GTX

Figure 6.6 shows the chip scope result when the transmitted data are received through GTX.

![Figure 6.6. Chip scope result showing data receiving through GTX](image2.png)

Figure 6.6. Chip scope result showing data receiving through GTX
The data after receiving through GTX are filtered and stored in FIFO as explained in section 5.2. The data from FIFO are then read. Figure 6.7 shows the FIFO_DATA_out which are the data read from FIFO. Figure 6.8 shows the same data are displayed after they are transferred to PS through AXI.

Figure 6.7. Chip Scope showing FIFO_DATA_out(FIFO read)

Figure shows the FIFO data read in PS and displayed in the terminal in SDK.

6.2.1.2 Bare Metal Readout with Experimental Setup

Figure 6.9 shows the experimental setup where the project was tested. In this experimental setup five fish lines are used. These fish lines are illuminated by the lasers and if the position of the camera exactly faces the illuminated fish lines, it captures the fish line. The signals from the camera are very similar to the real pellet events.
**Figure 6.9. Experimental setup using fish lines**

Two cameras are connected to the camera link board. These cameras send Pellet information to it. The camera link board then sends that information to the ZYNQ board.

Figure 6.10 shows the chip scope result taken when the test was performed. As it can be seen the pellet found data are sent in five words where each starts with ff0000bd.

**Figure 6.10. Chip scope result showing Pellet information received in Zynq Board from camera interface**

Figure 6.11 shows the data received from two cameras separated by “000000bc”.

**Figure 6.11. Chipscope result shows data received from two cameras**

The camera interface can also send raw data when selected. Raw data format was chosen for one camera. Figure 6.12 and figure 6.13 shows the raw data received from one camera.

**Figure 6.12. Chip scope result shows the received raw data**
6.2.2 **Linux**

The ultimate aim was to use Linux OS for DAQ firmware, so the Linux applications were created and the DAQ was tested.

### 6.2.2.1 Loopback with Linux Implementation

Figure 6.14 shows data displayed on two terminals. Left terminal shows the Linux OS running on Zynq Board where the data are looped back. After loopback data are send through Ethernet to a PC where Linux OS is running. The terminal on the right displays the data received through Ethernet.

![Figure 6.14. Loop back on Zynq Board having DAQ firmware and Linux OS](image)

### 6.2.2.2 Linux Implementation Readout with Experimental Setup

The project was finally tested by connecting the ZYNQ Board with the camera interface through fiber optics and the data was received in Linux OS from where the data was sent through Ethernet to PC. The data received in the PC was stored in a text file instead of displayed on terminal. Figure 6.15 shows the chip scope picture when the data was received from the camera link board into the PL before it is sent for further processing. As can be seen each pellet information starts with “ff0000bd”.

![Figure 6.15](image)
Figure 6.15. Data received from camera Link Board (chip scope picture)

Figure 6.16 show the data received from two cameras are separated by “000000bc”.

Figure 6.16. Data received from two cameras (chip scope picture)

Figure 6.17 shows the data saved on a text file when sent to PC. As it can be seen in the figure that “x000000BC” is filtered out.
Figure 6.17. Pellet data in text file

The camera link board is then configured to send raw data. Figure 6.18 shows the chart drawn from the raw data received in a text file.

Figure 6.18. Chart showing pellet in raw data
Figure 6.19. Pellet raw data in a text file

The data are received from cameras at a speed of 300k word/s and the speed at which the data travels via DAQ firmware is 200k words/s.
Chapter 7. Conclusion and Future Work

The aim of this work was to prove a concept of a small portable DAQ device, which can collect experimental data over optical links and send them to a PC via the Ethernet. The DAQ device should feature an embedded system, running an open Linux, which allows for flexible configuration of DAQ parameters.

The milestones of this work, such as creating a GTX interface for optical links, constructing a simplified DAQ firmware, configuring the embedded system, compilation and installation of the Linux as well as writing the simplified DAQ software for both the embedded system and the PC were achieved.

Given the broad scope of this work and a limited time, it was decided to keep the minimum profile of each of the milestones (only one optical link, limited DAQ control) and instead concentrating on achieving the full path of the data collection from the data source to the disk. In the end, the data sent by a camera on one end were successfully stored on a disk in a PC.

Further improvements (out of the scope of this work) would include extending the optical interfaces count to 4, porting of an event building/DAQ control firmware from the VME DAQ board, implementing an internal DMA data transfer to increase the data rate and writing the DAQ/control software for the PC.
Chapter 8. References


[11] The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC


[14] PHD research “Development and Applications of Tracking of Pellet Streams” by Andrzej Pyszniaik


Abbreviation List

DAQ: data acquisition systems
GTP: Gigabit Transceiver Protocol
VME: Versa Module Europa
GTX: Gigabit transceiver
SerDes: Serializer/Deserializer
PMA: Physical medium attachment layer
PCS: Physical coding sublayer
SDK: Software Development Kit
OS: operating System
PL: Programmable Logic
PS: Processing System
FSM: Finite State Machine
BSP: Board Support Package
AXI: Advance eXtensible Interface