Fachhochschule Wiener Neustadt für Wirtschaft und Technik
Fachhochschul-Diplomstudiengänge Technik

Realisation of a fast data storage and transfer system using a FPGA and an USB interface.

Juni 2005 Roman LEITNER

_____ _____

Datum Verfasser

Fachhochschule Wiener Neustadt für Wirtschaft und Technik
Wiener Neustadt
Fachhochschul-Diplomstudiengänge Technik

Titel der Diplomarbeit:

Realisation of a fast data storage and transfer system using a FPGA and an USB interface.

Diplomarbeit

Eingereicht von: Roman Leitner

Vertiefung: Mechatronik/Mikrosystemtechnik

am **Fachhochschul-Diplomstudiengang Präzisions-, System- und Informationstechnik**

Begutachter: Prof. (FH) DI Helmut Frais-Kölbl

Wiener Neustadt: Juni 2005

_____

Ich versichere,

dass ich die Diplomarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe und diese Diplomarbeit bisher weder im In- und Ausland in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

_____ _____

Datum Unterschrift

Kurzzusammenfassung:

Das Beam Loss Monitoring-System des LHCs ist ein komplexes Mess- und Datenaufnahmesystem. Die Aufgabe dieses Systems ist die Stabilität des Teilchenstrahles zu überwachen und ihn bei zu grossen Verlusten abschalten zu lassen. Ionisationskammern ausserhalb der Magnete messen die Teilchenverluste und wandeln diese in ein Stromsignal um. Dieses Signal wird bereits im Tunnel in ein Frequenzsignal umgewandelt und in einem FPGA für den Transport zur Oberfläche über Glasfaserkabel vorbereitet. An der Oberfläche wird die Übertragung auf Fehler überprüft.
Um die Daten zu analysieren und online den Zustand des Teilchenstrahles zu überwachen, müssen diese Daten zu einem Computer übertragen werden. Diese Diplomarbeit befasst sich mit der Übertragung, Speicherung und Darstellung der Daten auf einem Computer. Die Daten werden mit Hilfe einem USB 2.0 Modul übertragen, mittels LabVIEW gespeichert und bei Bedarf angezeigt.

Schlagworte: Beam Loss Monitor, FPGA, USB, FIFO, LabVIEW

Abstract:

The Beam Loss Monitoring-system of the LHC is a complex measurement- and acquisition-system. The task of this system is to monitor the particle-beam and to request a dump in case of too big losses. Ionisation-chambers outside the magnets measure the losses of the beam and change this signal to a current signal. This signal will be converted to a frequency signal and prepared for the transmission to the surface inside a FPGA. On the surface the transmission is checked for errors.
To analyse and observe the status of the particle-beam, this data must be transferred to a computer. This thesis covers the transmission, storage and visualisation of the data on the computer. The data are transmitted by an USB 2.0 module. The storage and visualisation is made in LabVIEW.

Keywords: Beam Loss Monitor, FPGA, USB, FIFO, LabVIEW

Acknowledgements

I would like to thank a couple of people I meet before and during my stay at CERN.

First of all I want thank my supervisor at CERN Bernd Dehning. Thank you for giving me the possibility to spend my practical training there. You had always an open ear for my questions and problems. Thank you for your patience during the correction of my thesis.
I also would like to thank my supervisor at the Fachhochschule Wr.Neustadt Helmut Frais-Kölbl for his support and his help .

A huge *'Thank you!'* to the whole BLM section. It was a pleasure to spend some time with all of you. Virginia Prieto, Laurette Ponce, Barbara Holzer, Gianfranco Ferioli, Jan Koopman, Ewald Effinger, Gianluca Guaglio, Michael Hodgson Claudine Chery, Raymond Tissier and of course Markus Stockner. Especially thanks to Christos Zamantzas and Jonathan Emery who helped me to finish my project and always had a good clue when something did not work. Thanks for the precisely timed coffee breaks, finally i like drinking coffee.

I want to thank the Austrians at CERN for the funny conversations during the lunch breaks and after the working hours. Günter, Martin, Roman, Gerd, Markus(again), Florian, Thomas, Julia, Fadmar, Runar(You are one of us!) and all the others.

Thanks to Wolfgang Wöber and Wolfgang Haindl of the FH who offered me a nice job during my time there. I learned there a lot and i will never forget the barbecues. Thanks to Thomas, Helmut, Werner(!) and of course to Rene and Robert.

To all my colleagues at the FH Wr.Neustadt: I am very happy that i meet you. We had a lot of fun in the lectures and even more afterwards. Thanks to Elvis, Döner, chriscross, Werner, Wogri, Nic, Weissi, Geri, Leini, Raeff, Thomas,

Robert, Roman, Michl, Flo, Lebi, everyone in the dormitory and everyone i forgot here.

To all my other friends from Deutschkreutz and from the HTL: Thank you for your support all the time and for your strong friendship. Pam, Pu, Karin, Blonda, Achi, Nico, Elli, the twins, *die Römer*, Heinzi, Andl, Hans J., Jürgen, Mario, the "Good morning" - *email distribution list* and everybody else. Thank you for giving me something familiar every day.

I want to thank my brother and my mother for being there for me. We are a family and there is nothing we cannot manage.

In the end i want to thank a very special person and i want to dedicate this thesis to her. Kerstin, I know it was a bigger distance than usual, but we both made it as best as we could. Thank you for correcting the same mistake again and again. Thank you for your support.

Contents

1. Introduction

1.1. About CERN

CERN[1] is one of the largest and most influential research institutes in the world. The organization was founded in 1954 to explore what matter is made of and what forces hold it together. Today, CERN has over 120 different research projects. About 2500 staff members, 420 young students and fellows supported by the organization and 5000 visiting physicists, engineers and other specialists out of 40 countries and 371 scientific institutions try to figure out what is going on between the elementary particles. The instruments to investigate these particles are accelerators. There are several of them at CERN (Fig.1.1).

New particles are created by colliding in an accelerator prepared particles. The characteristics of these short living particles are determined in detectors at the interaction points (Fig.1.2).

1.2. The LHC

The LHC[2] is the largest project at CERN and when it is finished it will be the most energetic accelerator in the world which brings two proton beams or two ion beams into head-on collisions. With the ion collisions scientists will be able to recreate the conditions prevailing at the "Big Bang". The first beam will be injected into the LHC in 2007.

The LHC is a cyclic accelerator with a circumference of 27 km and a diameter of 8.6 km. The location of this machine is in the area next to Geneva approximately 100 m below the surface. The LHC-pipe consists of two tubes in which particles will be accelerated in opposite direction.

[1] *le Conseil Europ´een pour Recherche Nucl´eaire*
[2] Large Hadron Collider

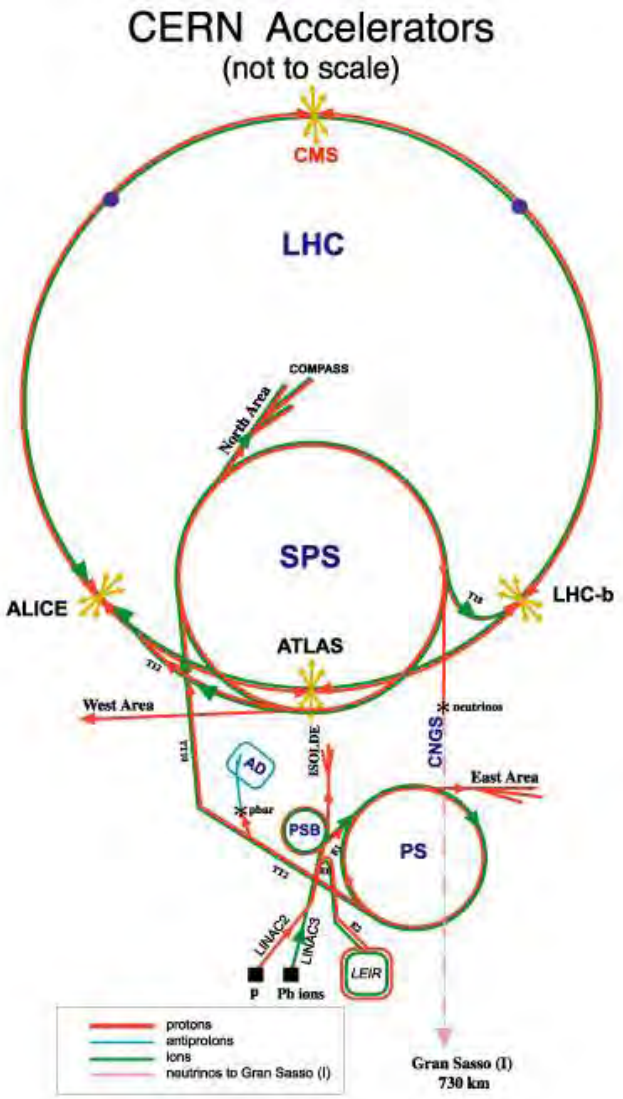


Figure 1.1.: Accelerators at CERN [2]

At 4 experimental areas[3] (Atlas, Alice, CMS, TOTEM and LHC-b) the collisions of the two beams are created with an energy of 14 TeV. Each experiment has its own specialised features and functions. One of the targets of the LHC is to find the particle which is responsible for the mass. The Standart Model says, that each one of the 4 elementary forces[4] is carried by a particle. It is interesting to note that the first studied force, the gravity, is still mysteries. The existence of all other forces and their sources is already prooved. The carrier of the these

[3]also called *interaction points*
[4]gravity , electromagnetic , strong and weak force.

forces are the *photon*, the *gluon* and the *W and Z boson*.

To generalise the Standart Model an additional particle was introduced, named after the introducer Higgs : the *Higgs*-particle. The existence of this particle is not verified yet, so nobody knows if the Standart Model is correct. To verify the theoretical construct and the existence of this particle, the LHC is build. With the help of this powerful cyclic accelerator it is possible to scan a wide range of beam-energy up to 7 TeV. The Higgs-Boson mass is estimated between 100 GeV and 200 GeV[5]. When the particle is found, a new linear accelerator(LINAC) for this special energy level will be build at CERN to examine the characteristics of the particle.

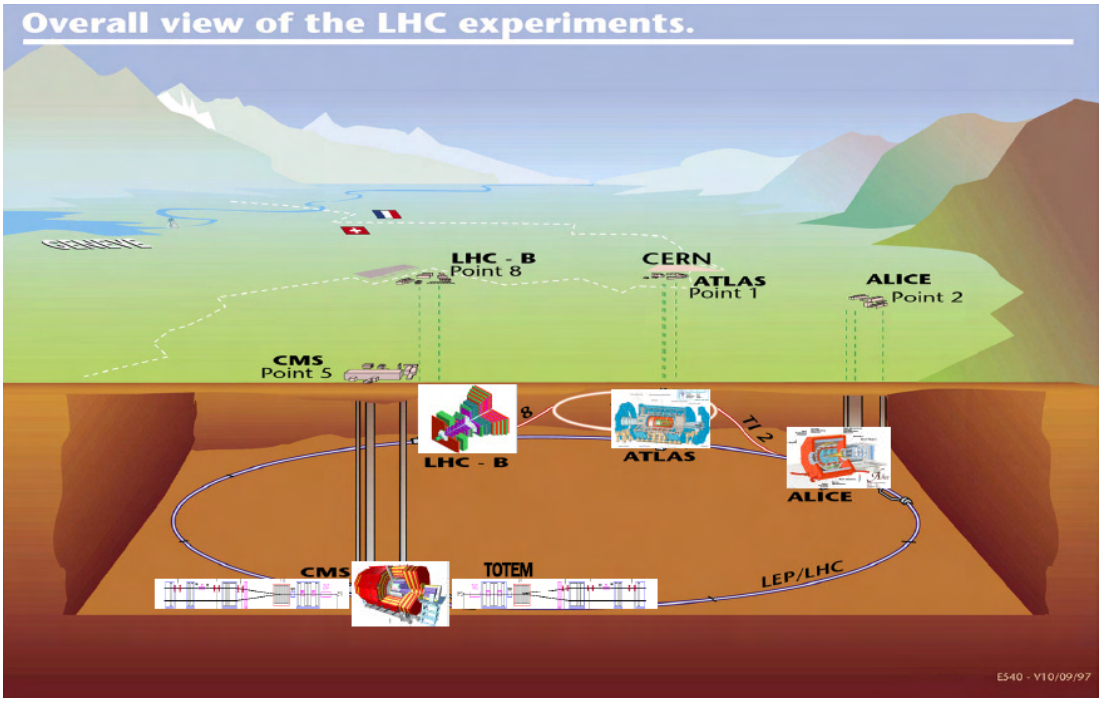


Figure 1.2.: Aerial view at the LHC Accelerator with the experiments : ATLAS, CMS,TOTEM, LHC-b and ALICE [2]

1.3. The Beam Loss Monitoring System

Particle accelerators have particle losses like every other machine[6]. These particles deposit partially their energy in the super-conducting coils. The result is

[5]Using Einstein's famous formula $E = mc^2$ the mass of a particle is proportional to its energy.
[6]Losses : Particles are leaving their trajectory and hit the superconducting magnets.

a local heating of the superconducting coils. When many particles deposit their energy in the coils of the magnets, the temperature of the magnets will raise. (fig.1.3)

The coils of the LHC magnets become superconducting at about 4.5 K . By cooling the superconducting magnets down to 1.8 K, the fluid helium becomes *super-fluid*. This means that the helium has almost no viscosity and provides a greater heat transfer capacity.

When the temperature raises the resistance of the copper coils will increase. Because a current through higher resistance causes a bigger loss of power in the conducting material, the coils will heat up until the magnet fill fall into normal conducting mode. This event is called a *quench*.

$$P \Uparrow when\ R \uparrow,\ because\ of\ P = I^2 * R \tag{1.1}$$

At this time the magnet field is not strong enough to bend the particle beam and the beam will fly into the coils of the magnets. When the beam hit the helium it will expand immediately like an explosion and might damage the magnet.

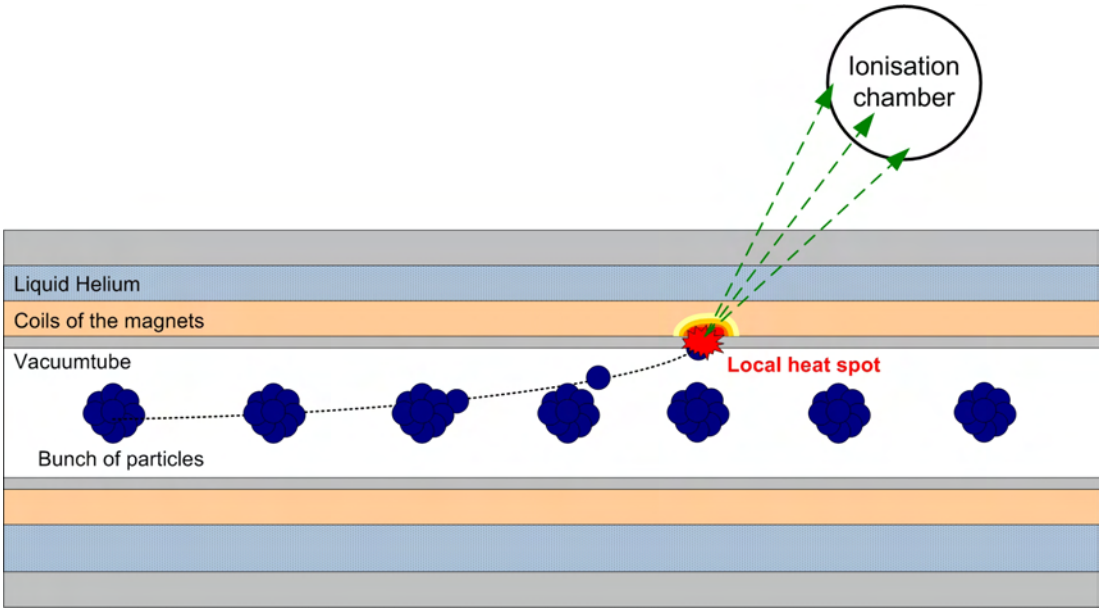


Figure 1.3.: Schematic view of the loss of particles from the circulating beam. When particles hit the wall of the vaccumtube, they deposit their energy in the metal, which causes a local heat spot.

If a magnet is damaged or even destroyed by a *quench*, it must be dismantled

and repaired. This causes high costs and long repair times.

To avoid a *quench* the loss of particles must be observed. Before a *quench* occurs, the particle beam needs to be *dumped*. To *dump* the beam means to kick the beam out of the accelerator ring into a huge block of solid carbon to protect the magnets from damage.

There are many so called machine protection systems foreseen at the LHC. One of them is the Beam Loss Monitoring System. Its task is to protect all magnets of the LHC from damage due to lost particles. To have a seamless monitoring of the beam, the BLM-system can observes losses at 3500 locations along the whole LHC ring.

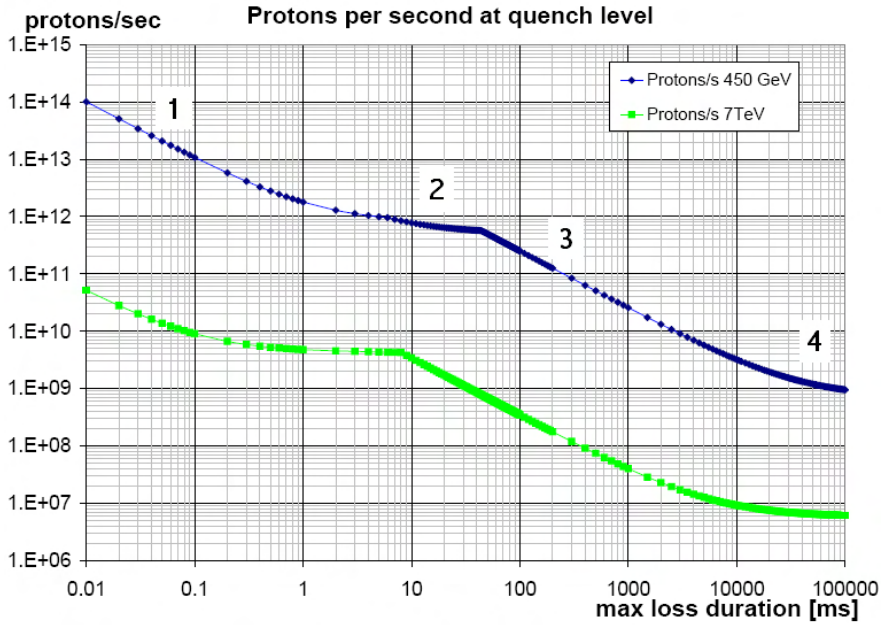


Figure 1.4.: Quench-levels of the LHC [3]

To avoid a *quench* it is necessary to know what is the amount of lost particles, until the magnet will quench. Using the Figure1.4 it is possible to define the *quench*-levels of the LHC. These curves are equal to the cooling capacity of the LHC. The losses of the LHC must be below the curves.

To protect the machine from a *Quench* it is important to record all losses over a long period, because also a low loss can cause a *Quench* if it has a long duration.

For short losses (1) the heat capacity of the cable is the relevant quantity. If the duration of the loss takes longer (2) the heat is delivered from the cable to

the cooling element, fluid helium. In the third area (3) all the heat capacity of the HE is filled up. For long losses (4) the helium transports the heat away.

1.3.1. Ionisation chambers

The ionisation chambers are the detectors for the BLM. When a particle hits the coils of the magnet, it will deposit its energy in the coils and also generate secondary particles . These secondary shower particles ionising the detector gas in the ionisation chamber . Due to the high supply voltage of the chamber both charges will drift towards opposite electrodes.

There are three prototypes of chambers.

Type A: Parallel Plate Chamber This design consists of 30 parallel aluminium plates (cathodes and anodes). The distance between the plates is 5 mm.

Type B: 2-Coaxial Chamber The cathode of this type is of coaxial shape with a diameter of 1 cm. The coaxial anode has a diameter of 3.8 cm and is concentric to the cathode.

Type C: 3-Coaxial Chamber This kind of chamber has an additional concentric anode inside the cathode.

All types of ionisations chambers are mounted inside a stainless steel tube, which is filled with gas (either N_2 or Ar-CO_2) at a pressure of 1.1 bar.

Type A is already in use as BLMs in the SPS and other accelerators at CERN. Also in the LHC parallel chambers will be used, because the signal development is faster, the E-field is constant between the plates and the ionisable volume is better defined.

About 150 electron ion pairs are created per cm path. The traversing charged particle rate is in such a way converted into an electrical current. The dynamic range of the BLMs is between 1 pA and 1 mA.

The chambers are located at the quadrupole magnets because the beam is reaching the largest size at this location and is formed down by these magnets. The probability to lose at this location particles is highest. The impact of beam protons on the superconducting magnets has been simulated to find out how where and how many BLMs are needed. To detect the beam losses outside of the cryostat three ionisation chambers per beam are necessary.

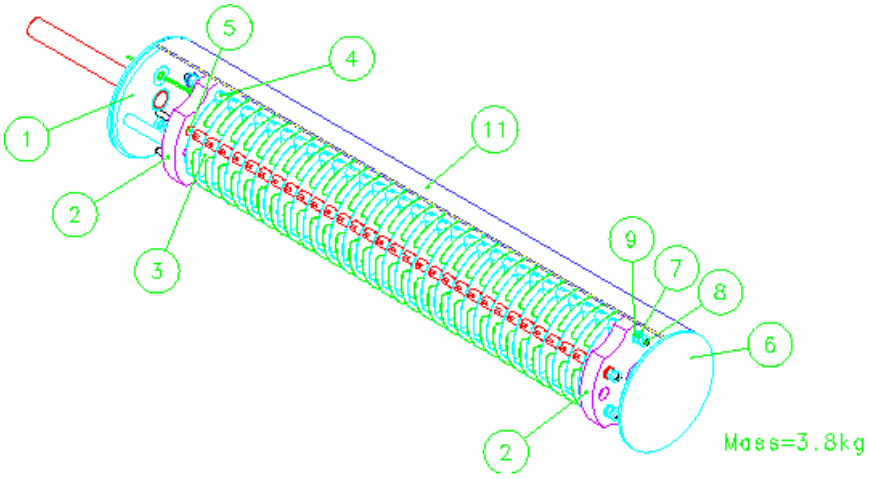


Figure 1.5.: Ionisation chambers: Type A, Parallel Plate Chamber [2]

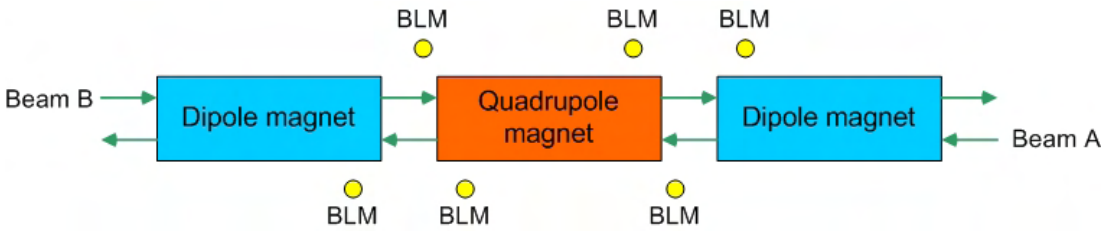


Figure 1.6.: Position of the ionisation chambers next to the magnets.

1.3.2. The readout-electronic for the Beam Loss Monitors

To convert the current signal from the ionisation chamber to a digital signal, a CFC[7] converter is used [5]. This Converter consists of an integrator, a threshold comparator and an current source. When the ionisation chamber delivers a current, it will be stored in the integrator as a voltage. If the current is constant, the voltage will fall linear because of the negative integration of the Operational Amplifier. When the voltage reaches a certain value, the threshold comparator will force the current source to empty the integrator using an negative current. This part of the BLM-System was developed during a previous thesis[5] by Werner Friesenbichler.

The output of the CFC is a sawtooth-shaped voltage signal. The amount of particles which flys through the ionisation-chamber is proportional to the output frequency of the converter. To measure this frequency the CFC pulses

[7] current to frequency

are counted. The higher the amount of particles, the higher the counter value.

Because the counter can only count full pulses the system has a certain error up to almost one count. To prevent this error and to increase the resolution of the system the value of the output-voltage of the CFC will also measured by an ADC in the same moment when the CFC counts are read.

1.3.3. Transmitting the data from the tunnel to the surface

To transmit the data of each chamber separately a huge amount of equipment would be necessary. The BLM uses a so called tunnel card to digitalises and to multiplex the data of up to 8 ionisation chambers into one data package (frame). In the FPGA the counted CFC pulses and the ADC data of the 8 chambers becomes multiplexed to one frame of data. To assign the data later to an unique position in the LHC tunnel, each data-frame also has some additional bits for these informations. The whole data-frame is shown in figure 1.7. It consists of 16 startbits, the card and the frame identify number, 64 bit of counter data (8 x 8 bit), 96 bits of ADC data (8 x 12 bit), 16 status bits and the CRC data.

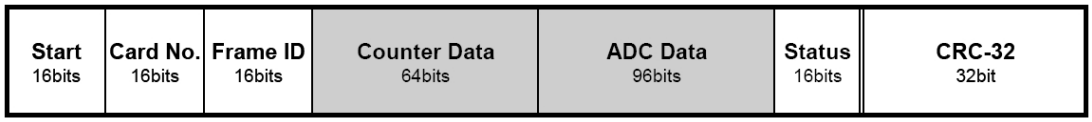


Figure 1.7.: The full data frame of the BLM-system. [1]

Then the data will be transmitted twice via optical link to the surface using an optical link, the GOH-board(fig.1.8). This board consists of the *GOL*[8] and a laser diode. The GOL is a radiation tolerant high-speed transmitter (800 Mbps) with a 16 bit-wide input and a CRC checksum generation. This ASIC was developed at the microelectronics group at CERN.

Because of redundancy reasons the data is transmitted over two separated fibres. On the mezzanine-card(fig.1.9) both optical signals are converted to an electrical signal and passed to the FPGA-board. To check transmission errors the CRC[9] values were calculated in the FPGA and compared with the CRC-value from the tunnel.

[8]Gigabit optical link
[9]Cyclic redundancy check

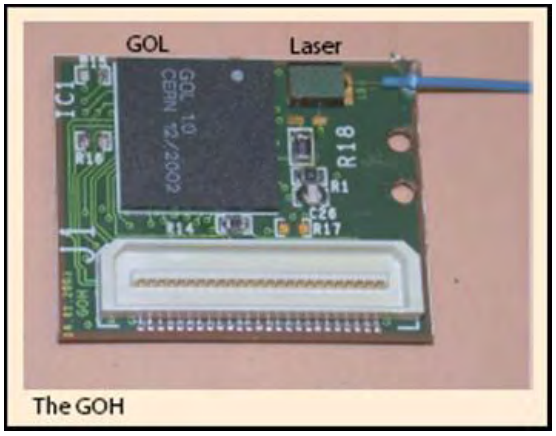


Figure 1.8.: The GOH board, containing the GOL and a Laser diode. [2]

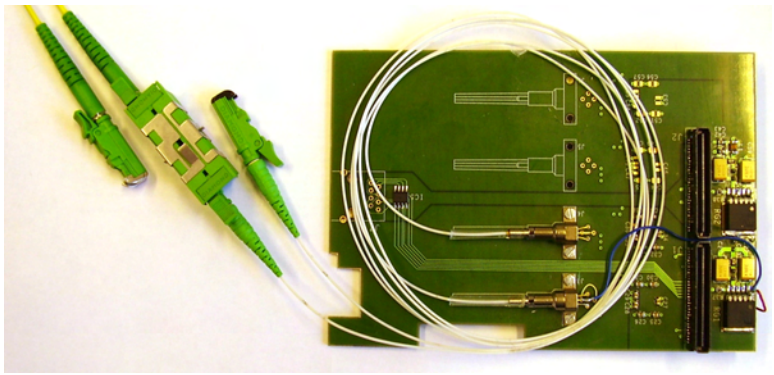


Figure 1.9.: The mezzanine card.

If there was an transmission error on one fibre the data of the other fibre is chosen for further treatment (table 1.1). If both are wrong the beam must be dumped because the monitoring system is not working properly.

Table 1.1.: Signal selection- and Dump-table. 1 = Correct, 0 = Error. [1]

CRC32 check		**Comparison of 4Byte**	**Output**	**Remarks**
A	**B**	**CRCs**		
0	0	0	**Dump**	Both signals have error
0	0	1	**Dump**	S/W trigger (CRCgenerate or check wrong)
0	1	0	**Signal B**	S/W trigger (error at CRC detected)
0	1	1	**Signal B**	S/W trigger (error at data part)
1	0	0	**Signal A**	S/W trigger (error at CRC detected)
1	0	1	**Signal A**	S/W trigger (error at data part)
1	1	0	**Dump**	S/W trigger (one of the counters has error)
1	1	1	**Signal A**	**By default (both signals are correct)**

2. Overview of the system

An abstract overview of the whole BLM testsystem is shown in figure 2.1.

The data from up to eight ionisation chambers are collected by the tunnel card and converted to an optical signal using the GOH-board. On the surface the data is converted back to an electrical signal on the mezzanine card. In the FPGA the transmission is checked for errors.(CRC)

The task of this project is to receive the data after the CRC-check and forward it to a PC for the storage.

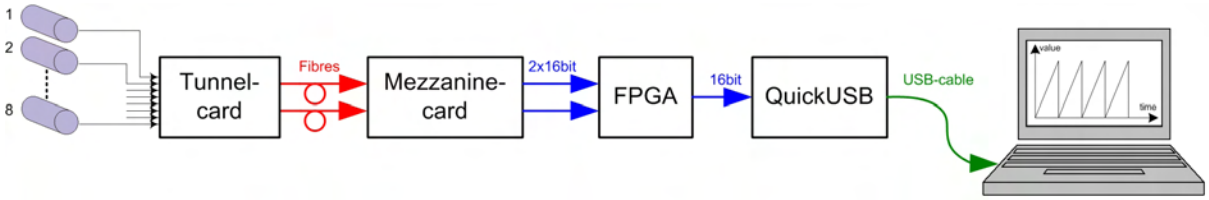


Figure 2.1.: Overview of the test system.

2.1. The data of the BLM-system

There are two versions of this project depending on the tests. In one version the whole data from the BeamLossMonitors will be transferred, in the other the data will be reduced on the chip, and only average values will be transferred.

2.1.1. The full data frame

The *raw* data consists of 256 bits which are delivered every 40 μsec on a 16 bit wide bus. So the data is a frame of 16 x 16 bit with an update rate of 40 μsec.

The calculated[1] data-rate is:

[1]Conversion: 1Mbit = 1000kbit = 1000000kbit

$$datarate_{full} = \frac{amount\ of\ data}{time-period} = \frac{256bit}{40\mu sec} = 6.4Mbit/sec$$

2.1.2. The reduced data frame

For most operations it is not necessary to transfer the raw data of the system. When reducing the amount of data it is important that no necessary information is lost. To reduce the data on the FPGA running sums of the data are calculated. The term '*Running sum*' means that all counter values are summed up for a defined time. Every time when the running sum is refreshed, the actually value will be added and the oldest value will be subtracted from the running sum.

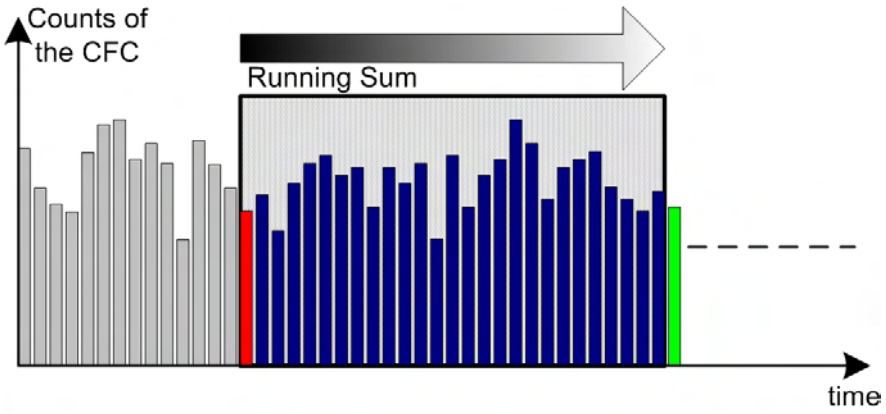


Figure 2.2.: Running sums: When the running sum is refreshed, it the newest counter value(*green*) will be added and the oldest (*red*) subtracted.

In table 2.1.2 the different lengths of the running sums are displayed. The first running sum will sum-up the counter values of the last 40 μs, the second running sum will do the same for 80 μs, and so on. The last running sum will sum-up the data for about 84 seconds.

To observe the limits of the system (fig.2.3, green and blue lines), the running sums are used to fit the curve of the allowed counts of the system.(fig.2.3, green and blue dots)

The size of this reduced data is 256 words[2]. These data, consisting of the maximum values of the running sums, has a much lower update rate than the *raw* data from the BLMs.

[2] 1 word = 16 bit

Table 2.1.: Table of length of the running sums.

sum	range	refreshing
	[ms]	[ms]
1	0.04	0.04
2	0.08	0.04
3	0.32	0.04
4	0.64	0.04
5	2.56	0.08
6	10.24	0.08
7	81.92	2.56
8	327.68	2.56
9	1310.72	81.92
10	5242.88	81.92
11	20971.52	1310.72
12	83886.08	1310.72

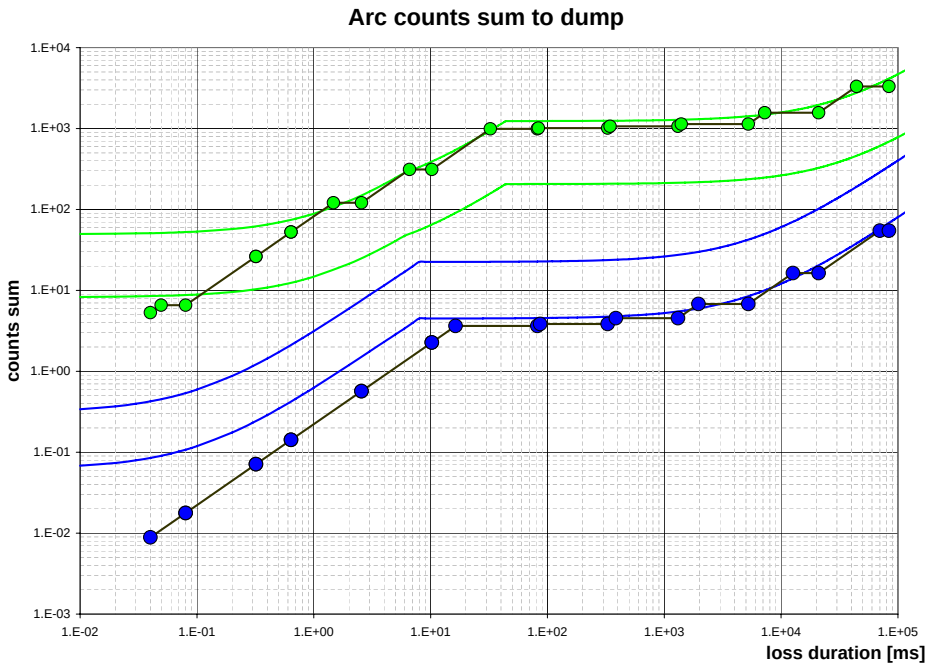


Figure 2.3.: Counts to dump : The runnings sums are used to fit the curve. [1]

Approximately all 0.8 seconds a new data-frame is send to the computer-interface. This timing is generate by a 25 bit counter which is counting with

40 MHz on the FPGA and is a chosen time constant. ($\frac{2^{25}}{40MHZ} = 0.8388sec$)

The calculated data-rate is:
$datarate_{reduced} = \frac{amount\ of\ data}{time-period} = \frac{256words}{0.8388sec} = 305.17578\frac{words}{sec}$

2.1.3. Conclusion:

The system on the FPGA should be able to handle both versions (full data-frame and reduced data). If the full data-frame is used the system has to transmit the data before the memory on the FPGA is full and data will be lost. In case of the reduced data-frame the system stores the data on the FPGA until a defined amount can be send to the computer.

2.2. The system on the FPGA

The FPGA receives the data from the mezzanine-card and transmits it to the computer. Because of the fact that every transfer protocol[3] has its own special header data which are needed for the transfer, the whole transmitted data will be bigger than the data from the BLM-system.

Whether the BLM-data is small or big, this header will be transferred on every time a transmission is started. When every short data-packet arriving is transmitted to the computer, the interface will be very busy because it has to start a transmission very often.

A better solution is to accumulate the data on the FPGA and send it when a bigger amount of data is on the FPGA. This will reduce the traffic and also increase the possible datarate of the system.

2.3. Connection to the PC

For the transfer to the computer a standard interface should be chosen, which is able to handle the amount of data provided by the Beam-Loss-Monitors.

RS232 : The serial interface is an older interface. It is easy to build up a communication using the RS232, but the data rate is to low for this project.

[3]USB,Firewire,Ethernet,...

Ethernet : Ethernet is a common interface to link computers together. It can handle big amounts of data. The possible datarate depends on the used protocol (TCP/IP, IPX, ...).

Firewire : Firewire is able to transfer bigger amounts of data in a short time (up to 800 Mbps). In this bus system clients can *talk* to each other without the host. It is used to transfer data from e.g. a digital video cameras to the computer.

USB : This interface can transfer data in full-speed mode with 480 Mbps. Every communication on the bus has to be transmitted to the host, who passes the data further to the target device. Because this is a Bus-interface the data-rate will be decreased if more devices are connected to the bus.

Conclusion:

Firewire and *USB2.0* are able to handle the required amount of data . Nowadays most of the computer have USB Port, but only a few of them have Firewire. To provide more flexibility the USB-interface is chosen.

Instead of building up an USB2.0 interface from scratch an external USB-module can be used to add this feature to the project. (chapter 4)

2.4. Processing of the data on the computer

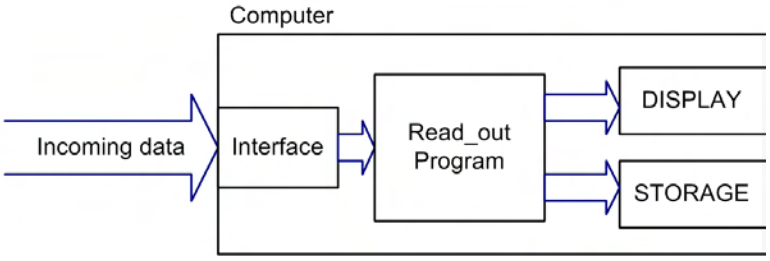


Figure 2.4.: Data-Processing on the computer: The data arrives via the Interface (USB, Firewire, Ethernet,..) at the computer. The read-out program has to display the data and store them on the computer.

After the data have been transferred to the computer it have to be processed. The program on the computer collects the data from the interface before new data are arriving and old data are lost. This *read_out* program also displays the data for online inspection and stores it on the hard-disk. For this reason the program *LabVIEW* is chosen.(chapter 5)

3. The FPGA-System

The *STRATIX development board* is made by Microtronix[1]. It consists of the STRATIX-FPGA[2] from Altera, 10 bit ADC and DAC, Ethernet, USB, PMC connectors and normal I/Os[3].

For this project only the PMC connectors and the Santa Cruz IOs of the FPGA are used. The mezzanine card is connected to the PMC, and the interface to the computer is connected to the Santa Cruz pins.

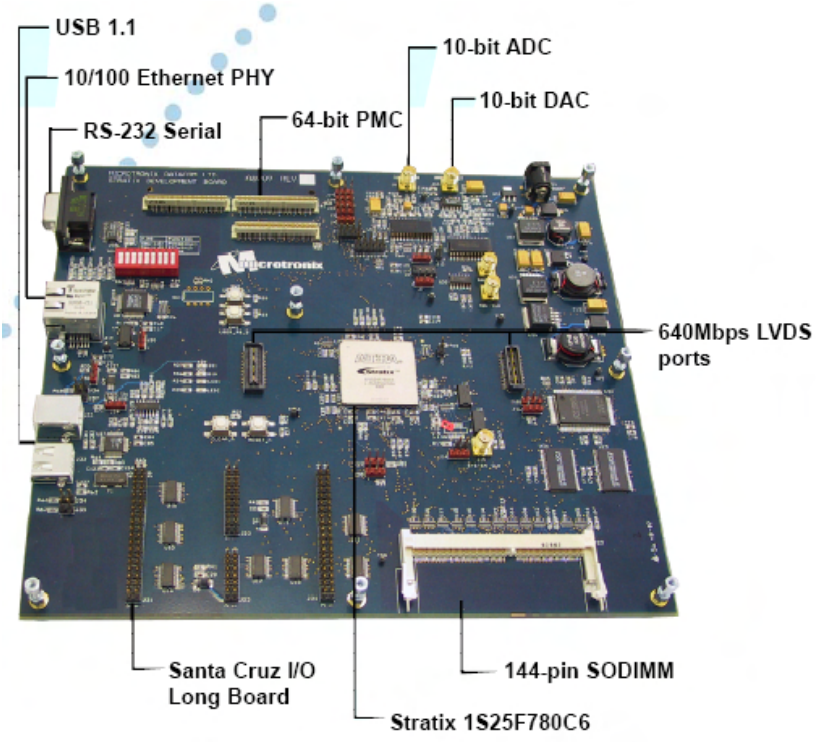


Figure 3.1.: FPGA Development board. [7]

The FPGA receives data from the mezzanine-card. These data are checked

[1] www.microtronix.com
[2] Stratix 1S25780
[3] Santa Cruz I/O Connectors

for error[4] during the transmission and prepared depending on the final usage of the project. On the FPGA a defined amount of data must be stored for a short period, until it is transferred to the computer.

The data from the mezzanine card arives synchronous with a 40 MHz-clock. On the other side the QuickUSB-device has a clock-frequency of 48 MHz. To connect these two systems the data must be collected from the first system and handed to the second system.

For the storage of the data on the chip a FIFO[5] can be used. This special memory does not need an address bus. The first data written to the FIFO will be the first data which will be available when reading from the memory.

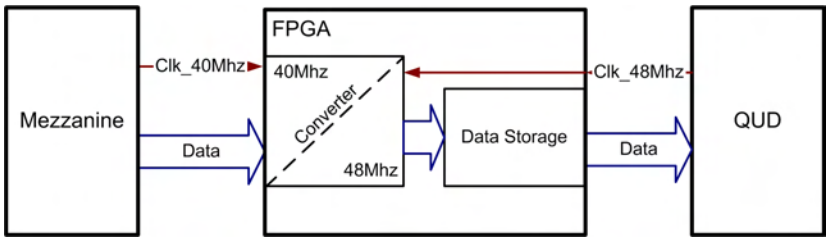


Figure 3.2.: Structure of the FPGA System. On the FPGA the data must be converted from one clock-system with 40 MHz to another clock-system with 48 MHz.

There were two versions of the FPGA design. In the first version a memory is used to convert the clock system and a FIFO is used to store the data on the FPGA. In the later version the system consists of a FIFO which is able to do both tasks.

3.1. Version 1 : A dual clock memory and a FIFO

3.1.1. Changing the clock system

To change the clock system a dual clock memory is used (fig. 3.3).

This memory has an own address- and data-bus for the *write* function and the same set of busses for the *read* function. When all data have been written to the memory with 40 MHz, it can be read from it with 48 MHz. Because this memory

[4]CRC-Check
[5]First-In, First-Out

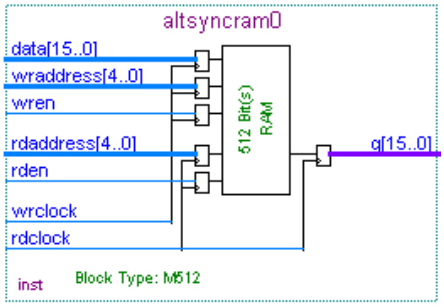


Figure 3.3.: Schematic fo the dual-clock memory

needs an address-bus for each procedure, two control modules must be developed (one for the write cycle, one for the read cycle, see Appendix for VHDL-Code).

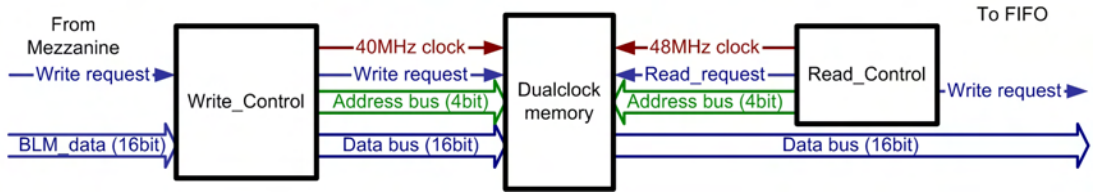


Figure 3.4.: Use a Memory to change the clocksystem: The this memory has an own address bus for write and for read procedures. The control modules *Write_module* and *Read_module* will generate the address bus and write or read the data.

The function '*write_data*' will take the data from the input and pass it over to the memory with the appropriate address-values. When all data has been written to the memory, the function '*read_data*' will read it from the memory and write it to the FIFO.

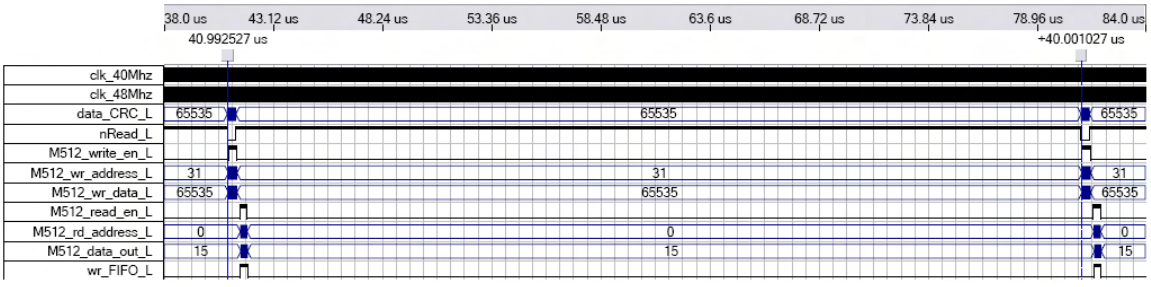


Figure 3.5.: Postsynthesis simulation of dual clock memory: Every 40μs data are available. This data are converted to the 48 MHz clock system and stored in the FIFO before new data arrives.

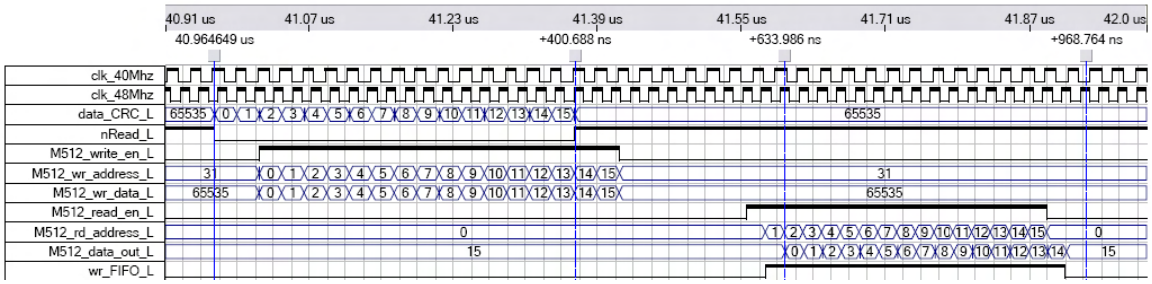


Figure 3.6.: Postsynthesis simulation of dual clock memory 2: After writing data to the memory, it is read using the other clock in the system, and passed to the FIFO.

In figure 3.5 the simulation of the write and read cycle is shown. The first two signals are the two different clocks of the system. The 16 bit wide data *data_CRC_L* [6] is generated by a testmodule on the board. It consists out of 16 values of 16 bit which is equal to the full data-rate of the BLM-system.

When the signal *nRead_L* is low, data are available. The module *write_data* passes the data to the memory generating the address bus at the same time. When this procedure (Figure 3.6 at time 41.42 μs) is finished, the VHDL-module '*read_data*' will read the data from the memory using the signal *M512*[7]*_read_en_L* and the address-bus *M512_rd_address_L*.

To write the data to the FIFO the signal *wr_FIFO_L* is generated.

Regarding the simulation this signal starts one cycle before the data are available. The reason is that the FIFO on the FPGA needs one cycle to recognise that data should be delivered. Compared to the input, the data are available at the same time as the signal *nRead_L* is low [8].

The figure 3.5 shows a detailed view of the write and read procedure. To write 16 words to the Memory 400 nsec are needed. ($\frac{16words}{40MHz} = 400nsec$). Because of the fact that the reading procedure is using the 48 MHz clock, it only takes 333 nsec to read the data. ($\frac{16words}{48MHz} = 333nsec$)

[6] The 'L' behind the signal name indicates that this is an internal signal of the chip. Internal signals can be accessed in a simulation by using an so called LCELL. For the final synthesis, these LCELLs will be ignored.

[7] M512 indicates that this memory is synthesised to an on-chip memory block of the size of 512 bit.

[8] For this signal negative logic is used.

3.1.2. Storage of the data

To store the data on the FPGA a FIFO[9] can be used (fig. 3.7).

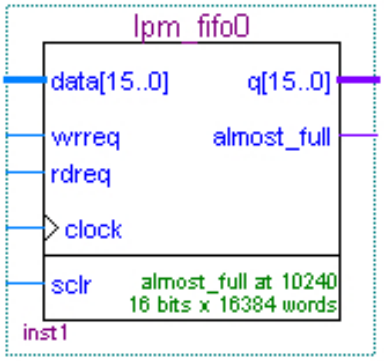


Figure 3.7.: Schematic of the FIFO

A FIFO is a special form of a shift register. The first value written to the FIFO is the first value which is available at the output. A FIFO has two pointer. One pointer indicates the first data in the memory, the other the last data in the memory. With subtracting the address values of these two pointers, the fill level of the FIFO can be calculated. If the fill level is equal to the size of the FIFO, the FIFO is full. In this case when new data are written to the FIFO old data will be overwritten. If the fill level is zero no data are stored in the FIFO.

Data can be written to the FIFO when the input *write_request* is high. The FIFO will store data on every clock cycle as long this signal is high. When reading from the FIFO, the data will be delivered synchronous to the clock as long as the signal *read_request* is high.

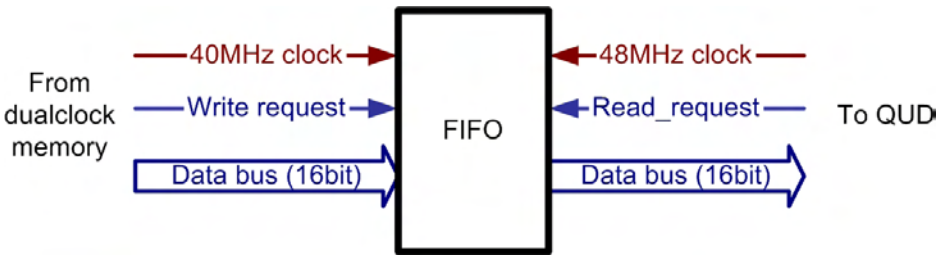


Figure 3.8.: The FIFO can be used as data storage until the data can be read from the computer.

[9]First in, First out

3.2. Version 2 : A FIFO can handle it all

Another alternative and even a better way is to choose a FIFO which has a separated read and write clock. The dual-clock FIFO(fig. 3.9) can be used instead of the dual-clock memory and another FIFO.

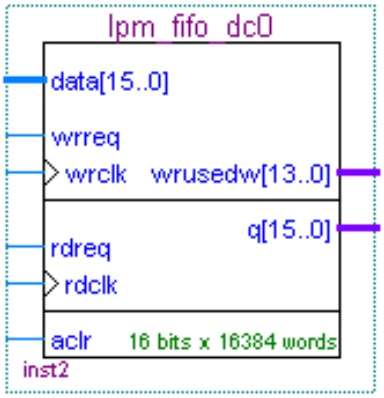


Figure 3.9.: Schematic of the dual-clock FIFO

Because it does not need an address-bus, the layout of the FPGA system is easier.

A FIFO has an additional output for the fill level. Depending on the size of the FIFO the fill level signal is an n-bit[10] wide bus, which value changes after every time data are written to the FIFO or read from it.

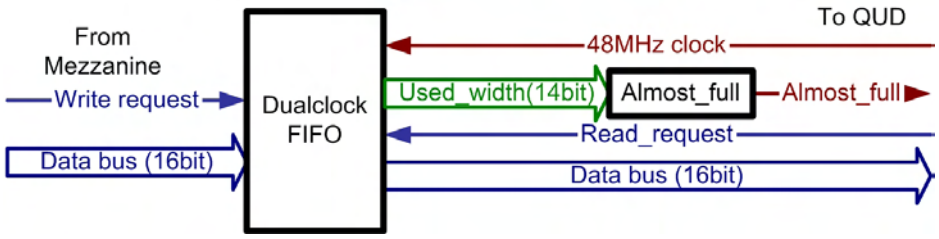


Figure 3.10.: The dual-clock FIFO can be used to convert the clock system and to data storage until the data can be read from the computer.

[10] $n = \frac{ln(size\ of\ FIFO)}{ln(2)}$

3.3. Size of the FPGA-FIFO

The size of the FIFO is an important parameter of the system. It determines how fast or slow the computer must read the data from the FPGA. Basically the FIFO should be as big as possible. The size is limited by the other functionality which is realised on the FPGA. When the whole design is on the FPGA, and there are some blocks of memory available, the size of the FIFO can be increased. The lower limit of the FIFO-size is determined by the interface to the FPGA and the computer. When the size is small, the computer has less time to receive and to store the data. The size of the FIFO on the FPGA is chosen to 16384 words with an bus-size of 16 bit.

3.4. Fill level indication

In chapter 5 a signal is needed to indicates when the fill-level is above a threshold level. This signal will be used for the handshake between the computer and the FIFO. To generate this signal a comparator is used. The value of the *used_width*-bus will be compared to a threshold value . This value should be a multiple of the frame size and about $\frac{2}{3}$ of the FIFO size. The remaining $\frac{1}{3}$ is used as a buffer, in case the computer cannot read the data in time.

The chosen threshold value is 10240 words.

When the value of the fill-level is bigger than this threshold the output signal *almost_full* will indicate that the FIFO is almost full by changing the value to *low*[11] When the fill level of the FIFO is below the threshold, the signal will be *high*.

Code-example of the threshold fill-level

When the *used_width* of the FIFO is equal to or bigger as 1024 words the indicator *almost_full* will be 0. When data are read from the FIFO the *used_width* will be reduced and the indicator will be 1.

```
Fifo_used_width : process(clock_40MHz) is
begin
     if rising_edge(clock_40MHz) then
       if used_width >= 10240 then
```

[11]Negative logic is used for this signal.

```
          almost_full <= '0';  -- negative logic
        else
          almost_full <= '1';
        end if;
      end if;
end process Fifo_used_width;
```

3.5. How long does it take to fill the FIFO?

To calculate how long it takes until the FIFO is full, the datarate of the system is necessary.

Full BLM-data:

data rate $= \frac{256bit}{40\mu sec} = 6.4\frac{Mbit}{sec} = 400,000\frac{words}{sec}$[12]

$$t_{FIFOfullA} = \frac{size\ of\ the\ FIFO}{data\ rate} = \frac{16384words}{400000\frac{words}{sec}} = 40.96msec \tag{3.1}$$

$$t_{FIFOalmostfullA} = \frac{size\ of\ the\ FIFO}{data\ rate} = \frac{10240words}{400000\frac{words}{sec}} = 25.6msec \tag{3.2}$$

The read_out program must be able to read the data before the FIFO is full.

$$t_{read\_out} = t_{FIFOfullA} - t_{FIFOalmostfullA} = 15.36msec \tag{3.3}$$

Reduced Data:

data rate $= \frac{256words}{0.8388sec} = 305.17578\frac{word}{sec}$

$$t_{FIFOfullB} = \frac{16384words}{305.17578\frac{words}{sec}} = 53.687sec \tag{3.4}$$

$$t_{FIFOalmostfullB} = \frac{10240words}{305.17578\frac{words}{sec}} = 33.554sec \tag{3.5}$$

$$t_{read\_out} = t_{FIFOfullA} - t_{FIFOalmostfullA} = 20.133sec \tag{3.6}$$

[12] 1 word = 16 bit

Conclusion:
In the first system it will take 41 msec (Equation 3.1) until the FIFO is full and data will be overwritten. The Labview program must have a loop time far below this time to make sure the data can be read out before the FIFO is full.

The second system will take about 54 seconds until the FIFO starts to overwrite data. Every 33 seconds a new set of data will be read from the computer. This update rate is not suitable for an online monitoring of the system.

By changing the threshold-value of the FIFO (*almost_full*) to 1024 words the update rate can be change to about 3 seconds.(refer to 3.8)

$$t_{FIFOalmostfullB} = \frac{1024 words}{305.17578 \frac{words}{sec}} = 3.355 sec \tag{3.7}$$

$$t_{read\_out} = t_{FIFOfullA} - t_{FIFOalmostfullA} = 50.332 sec \tag{3.8}$$

3.6. Phase locked loop

The design on the FPGA uses two different clocks. One clock (40 MHz) is coming from the mezzanine card. The other one(48 MHz) is taken from the QuickUSB module. It is necessary to use the clocks from these devices because when communicating with external devices it is important to be synchronous to their clock.

This board will be used as a mobile measurement system, which must be able to handle various and sometimes also very noisy environments. To prevent that the system has a malfunction because of the noise in its environment, the PLL[13] is used to reduce the influence of the noise to the function on the system.

The PLL can also be used to compensate the travel time of the clock signal to overcome the distance from one end to the other end of the cable. Because of the short cables and the low frequency in this project (about 20 cm) this is not necessary. (Pictures of the PLL can be found in the Appendix)

[13]Phase locked loop

3.7. Button debounce

A switch or a button is a oscillating mechanical system and its signal is not appropriate for the use in a digital system without preparation. When closing a switch, the two metal contact-plates will hit on each other and close the electrical circuit. Due to the mechanical system the contact-plates will open and close again several times until they stay closed. This procedure is called *bouncing* and lasts for about 200 to 400 μsec (depending of the quality of the switch). When an external switch or a pushbutton is used as an input of an digital Circuit, it must be debounced.

A FPGA, running with a clock of about 40 MHz, will scan this input every 25 nsec. This means that the FPGA will recognise multiple events on the switch until the bouncing is over. These multiple events can cause a malfunction on the FPGA.

To prevent this, these multiple events must be ignored by the system until the bouncing is over. This can be realised by comparing the last values in a shiftregister. Every clock cycle a new value is written to the shiftregister. If all values in the shiftregister have the same value, the bouncing is over and the value of the switch can be used inside the FPGA. Usually it is not necessary to scan an external switch with the onboard clock. Because for this a very big shiftregister would be necessary to accumulate the data for about 200μsec.

$$Size_{Shiftregister} \geq \frac{t_{bouncing}}{clock\ period} \tag{3.9}$$

Regarding to the equation 3.9 either the size of the shiftregister can be bigger or the period of the used clock can be increased. Using the onboard clock of 40 MHz a shiftregister of a size of 8000 bit would be necessary to debounce for 200μsec. But there is no need to scan the switch with the onboard clock. For the *debounce* process a much slower clock can be used. Using a slower clock it is not necessary to compare 8000 values in a shiftregister. To generate a clock out of a faster one, either a PLL or a simple counter can be used. The counter is incremented on every clock of the input clock (*clk_40 MHz*). When the value of the counter is equal to a specified value the output signal is inverted. The advantage of this signal is, that no specialized functions of a FPGA are used, and it can be reused in another FPGA family.

```
clk_divider : process(clk_40MHz) is
  begin
    if rising_edge(clk_40MHz) then
      if cnt /= 2000 then
        cnt <= cnt + 1;
      else
        cnt <= 0;
        clk_100Hz <= not clk_100Hz;
      end if;
    end if;
end process clk_divider;
```

To calculate the size of the shiftregister or the necessary clock frequency to debounce a signal the equation 3.10 can be used.

$$period_{clock} \geq \frac{t_{bouncing}}{Size_{Shiftregister}} \tag{3.10}$$

For a shiftregister of 10 values and an debounce time of about 1000μsec a minimum clock period of 100μsec is necessary. This is equal to a clock of 10 kHz.

```
debounce_pushbutton : process(clock_10000hz, debounced) is
  begin
     pb_debounced <= debounced;

     if rising_edge(clock_10000hz) then
      shift(9) <= not pb;                          --negative logic
      shift(8 downto 0) <= shift(9 downto 1);
        if shift(9 downto 0) = "1111111111" then
          pb_debounced <= '1'; --  if
        else
          pb_debounced <= '0'; --
        end if;
     end if;
    end process debounce_pushbutton;
```

Explanation of the VHDL-Code

Using a 10kHZ clock the value of the input signal *pb* is stored in a shiftregister. When all values in the shiftregister are equal to 1 then the bouncing is over.

This code covers the bouncing process when a switch or a button is closed. (When the button is not pressed, the value on the input is high, because this

signal line is connected to a pull-up resistance. When the button is pressed , a connection between the ground and the pull-up resistance is made. A current flows through the resistance, and generates a voltage on the resistance which is equal to the supply voltage. Because of that voltage drop down, the signal line of the button/switch is now low.

Simulation of the debounce process

In figure 3.11 the simulation of a debounce process is shown. Between 100 μsec and 450 μsec the input signal *pb* bounces. This signal is scanned with a clock of 10 kHz. On every rising edge of this clock the actual value of the input signal is written to the shiftregister *shift*. After 450 μsec the input signal is stable and after 1 msec the output signal *pb_debounced* is updated.

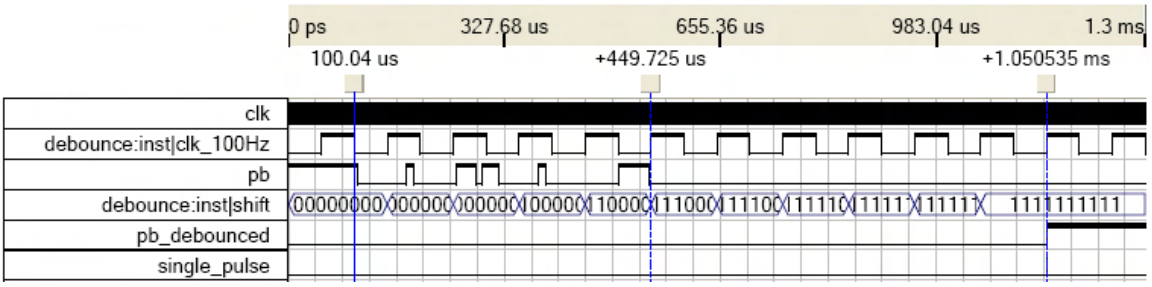


Figure 3.11.: Simulation of the debounce process.

Additional to this a single pulse is generated at the output *single_pulse*. This signal is used in the design to reset the FIFO when the pushbutton 1 is pressed. Because this pulse lasts only for one 40 MHz clock cycle, it is not visible in the figure and shown in a better resolution in figure 3.12.

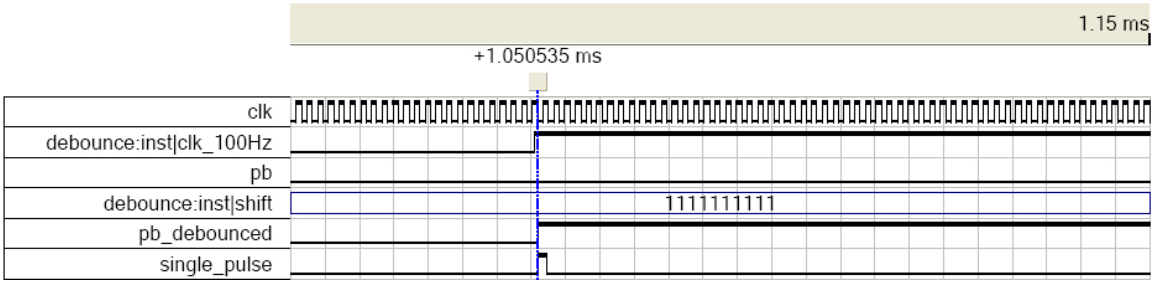


Figure 3.12.: Simulation of the debounce process, Singlepulse

3.8. Switches and buttons in the design

There are three switches used to change the behaviour of the module and to test the system. With these switches the system can be tested part by part.

Switch 1: Using switch 1 it can be chosen to read *real* data or only test values which are stored or generated on the FPGA to test the system.

(If 0 (closed), then real data from the mezzanine card are transmitted. If 1 (open), test data from the FPGA is forwarded to the system)

Switch 2: Switch 2 changes the behaviour of the system. It can be switched between transferring the raw data from the BLMs or only the maximum values of the running sums.

(If 0, the full data-frame , If 1, the reduces data frame.)

Switch 3: Switch 3 can be used to send either the maximum values of the running sums, or some test-data generated on the FPGA.This is useful to check If the whole system is on the chip is working.

(If 0, the test data from the CRC-block is taken (0 to 254). If 1, the data is generated on the FPGA (0 to 4))

Switch 4: Switch 4 changes the behaviour of LED[14]1.

(If 0, the clock from the QUD is used. If 1, the clock from the mezzanine card is indicated.)

The Pushbutton 1 can be used to reset the FPGA and to delete all data inside the FIFO.

3.9. Led indicators

On the development board there are 2 LEDs which are used to indicate the status of the system.

LED1 : is used to indicates that the clock from the mezzanine-card and the clock from the QuickUSB-device are available. The mode can be changed with

[14]Light-emitting-diode

the Switch 4. This is useful to check if the other devices are connected and if the FPGA is working. For example, when the USB-device is not recognised by the computer, it will not deliver a clock signal.

LED2 : This LED is internally connected to a counter. The counter value is incremented when the signal *nRead_en* is low, what means that data are send to the system. Depending on the datarate the LED will flash with a different speed. This indicator shows if data is arriving through the fibres.

3.10. Conclusion:

The final version of the FPGA system consists of ..

- .. two PLLs to receive the clocks from external source (QUD and mezzanine-card).
- .. a test system, to test the transmission without connection to the BLM-system. This test system is controlled by the switches.
- .. indicators to check if the system on the FPGA is working and to know if data arrive through the fibres.
- .. a dual-clock FIFO which is able to change the clock systems and accumulates the data.
- .. a fill-level indicator which modifies a signal, when there is enough data stored in the FIFO for the transmission.
- .. a *16 to 8 bit* Converter, to prepare the data for the transmission to the computer. [15]

[15]This part of the project was realized by Jonathan Emery.

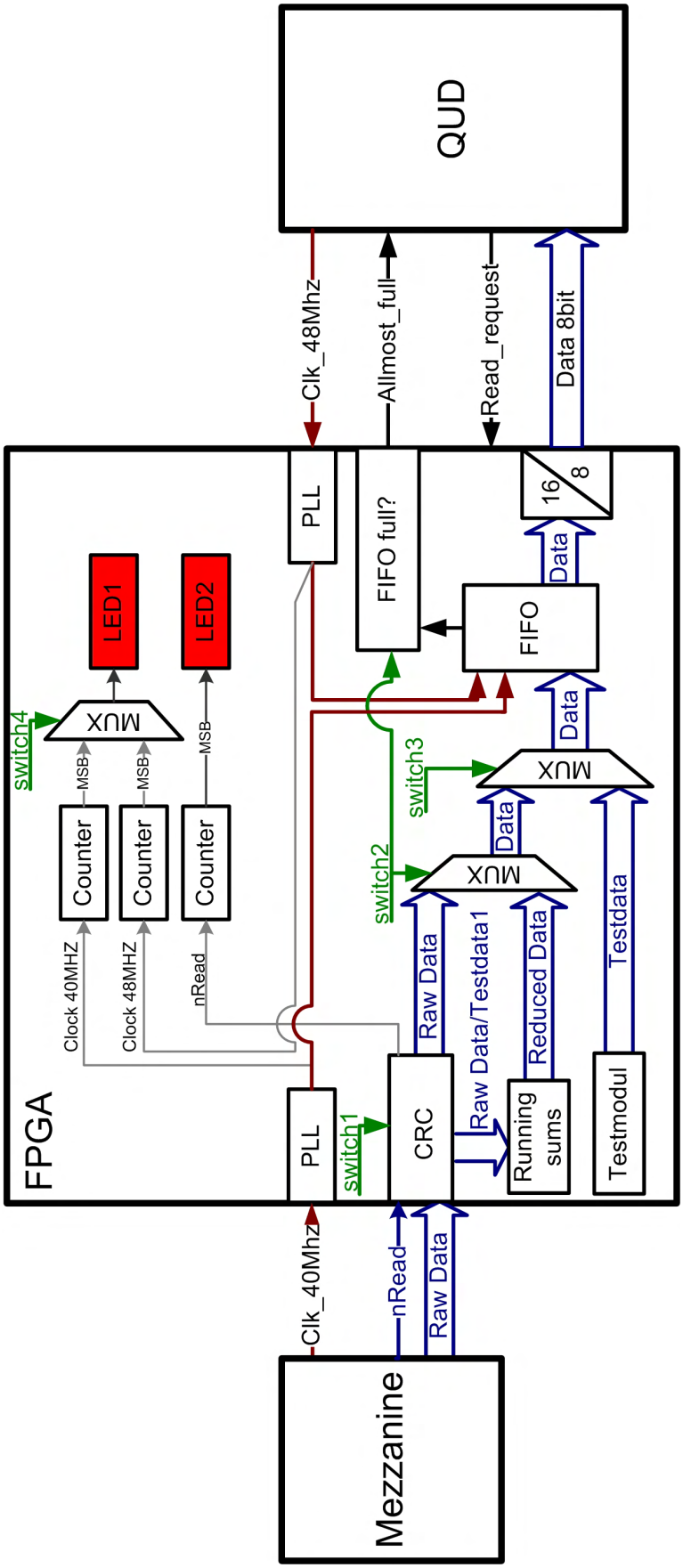


Figure 3.13.: Final overview of the implantation of the FPGA program.

4. The USB Interface

For the communication to the Computer the QuickUSB module from Bitwise-systems[1] is used. This device can be used to add a USB 2.0 interface to an existing project. It consist of a μ-controller from Cypress[2] and offers several ports like a 16 bit high speed parallel port(HSPP), three 8-bit general purpose ports, two RS-232 ports, one I^2C bus and an SPI port. The product is delivered with a precompiled library for Windows and Linux which can be used in the own project to communicate with the module.

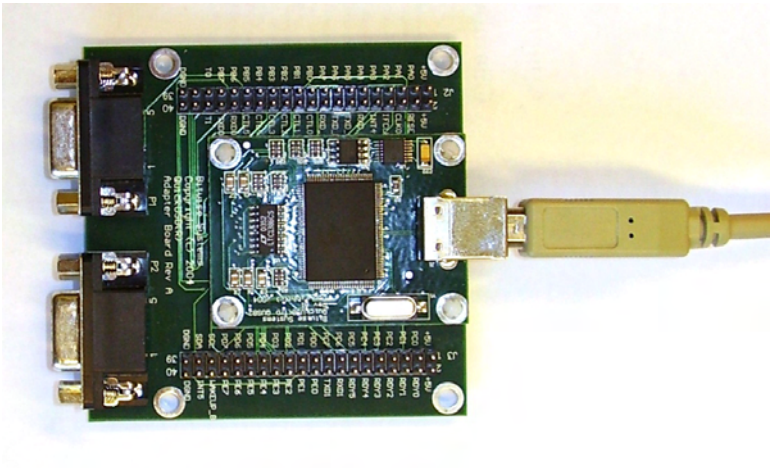


Figure 4.1.: QuickUSB module

For this project the HSPP[3] and one of the 8-bit general purpose ports will be used. The HSPP is a 16 bit wide synchronous bus which is able to transfer data up to a data rate of about 48 $Mbit/sec$. This data rate can be achieved because the HSPP is directly controlled by the USB-Engine and not by the 8051-Core of the μ - controller. All the other ports on the chip are controlled by the 8051-Core and then passed to the USB-Engine. (fig.4.2)

[1] www.bitwisesys.com
[2] CYC68013-128AC, also called EZ-USB FX2, `www.cypress.com`
[3] high speed parallel port

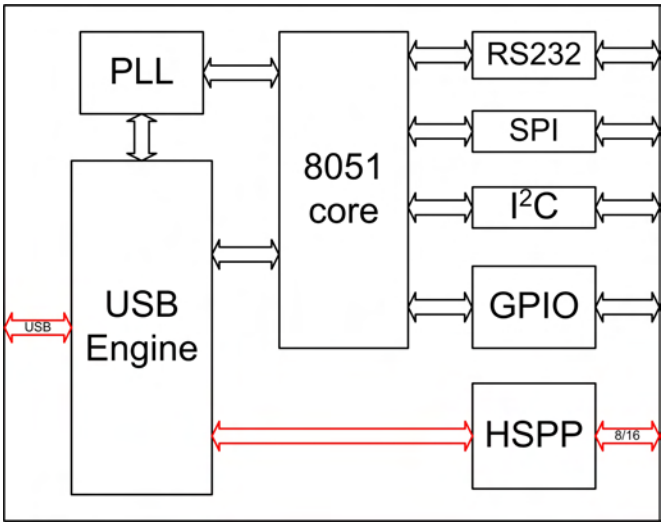


Figure 4.2.: Internal Structure of the μ - controller on the QuickUSB module. To transfer data at high speed the HSPP is directly connected to the USB-Engine.

The small data sheet of the QUD can be found in the Appendix. The full data sheet including all library functions can be found `www.bitwisesys.com`.

4.1. Settings of the QUD

The device has several settings to affect its behaviour. For the HSPP it is possible to choose between an 8 bit or a 16 bit wide bus. If an external memory, which needs an address-bus, is connected the port_C can be configured as an address-bus with or without auto-increment function. The FIFO of the HSPP can be either modified as Master or Slave. The QUD can also be used to program a FIFO in passive serial mode.

All Settings of the QUD[4] are saved in a volatile memory. They are not stored, if the module is not powered ! It is necessary to rewrite the settings to the QUD when it is attached to a computer. ([8], page 8)

The HSPP has an own FIFO which has a direct connection to the USB-Core. Because of that fact it must not pass all the data over the μ-controller and can

[4]QuickUSB device

transmit data with an data-rate up to 48 Mbyte/sec in Burst Mode.

The QUD-FIFO can be controlled using the signals *REN*, *WEN* and *CMD_DATA*. To read data from the HSPP to the computer the signal *REN* has to be high. It is possible to transfer the data in the other direction using the signal *WEN*. The signal *CMD_DATA* determines if the data on the address bus (Port C) is an address or a data.([8], page 11, 12)

4.1.1. Width of the HSPP

The bus-width can be switched between 8 - and 16 bit, depending on the project. When using the 16 bit version, the data is read in two steps. The lower 8 bit of the 16 bit arrives first, the higher 8 bit arrives second.

The reason is that the function of the QUD-Library can only read a 8 bit value at a time. To display the data correctly on the computer the two bytes must be shifted and the result must be merged to one 16 bit variable. The computer will take more time to rearrange the data and to store it to the computer. When the computer is not fast enough to handle all this in time, the FPGA-FIFO will overflow and data will be lost.

It is faster to merge only the two bytes, than to switch and to merge them. To prevent a time consuming data shifting on the computer the FPGA will only transmit on a 8 bit wide bus. (16 to 8 bit converter)

4.1.2. HSPP FIFO settings

The FIFO of the QUD can be either configured in *master-* or *slave-* mode.

Master mode

The FIFO of the HSPP is controlled by the QUD-library. Every transaction must be started from the computer using the QUD-library. Using this mode it is possible to write data to a memory using the additional address bus.

To change the FIFO to master mode the command *WriteSettings* must be used. For master mode the setting value is 0xFA. It is possible to write to and read data from the FIFO using the commands *QuickUSBWriteData* and *QuickUSBReadData*.

Slave mode

The difference to the Master mode is that in this mode the FIFO can be loaded by an external source. The FIFO is totally controlled by the external device. To load and read the FIFO the signals '*REN*','*WEN*' and '*CMD_DATA*' are used. In this mode the QUD must not wait for the computer to read or write data to an external device.

Conclusion:

On the first view the *salve*-mode would be the best solution for this project. The FPGA writes the data direct to the QUD-FIFO. Then the QUD will take care that the data is send to the computer.

But when using the master mode another advantage is available. The computer can control the read_out process and read the data when the computer is not busy.

In master mode the computer can ask continuously the system if enough data is accumulated on the FPGA.

Because windows is not a real time operating system, it will not process the data immediately when it is available. This method ensures that the computer will receive data when it is ready. (Refer to 5.3 for the structure of the Readout program.)

4.2. Connection between the QuickUSB and the FPGA

The cable between the QUD and the FPGA contains the HSPP (port B and port D) plus its control signals (*REN, WEN and CMD_DATA*). Also one signal is necessary to carry the fill-level for the FPGA-FIFO. For this the general purpose port A is used. Because the FPGA must work synchronous with the QUD the clock of the QUD must be transmitted. For later enhancements (usage of the DAC on the FPGA, ..) all general purpose ports are added to the cable. On the FPGA-side the *Santa Cruz* connectors are used. These are 72 IOs of the FPGA which can be used for prototyping.

The cable contains following signals:

- HSPP (Port B and D) plus its control signals

- all general purpose IOs (Port A,C,E)
- 48 MHz clock

The detailed pin assignment of the cable can be found in the appendix.

5. The readout program

Part of this project is to visualise the data on the screen and store it on the hard-disc for further analysis.

The QUD includes a c-library to communicate and to control the device with a computer. This offers the possibility to create an own c-project to store the data on the hard-disc.

But for this project the better solution is to use the National Instruments Software "LabVIEW".

5.1. Why use LabVIEW?

Depending on the measurements it is sometimes better to display the data rather in a graph than in a table. For example when the data consists of the average-values of the running sums a graphical display is more convenient than a table with the values. In LabVIEW both versions can be realised easily and even at the same time.

LabVIEW is also a proper tool to acquire, visualise, analyse and store data. On the National Instruments Homepage[1] LabVIEW is described like this:

> *LabVIEW is the development environment of choice for many engineers because of its unique ability to acquire, analyze, and present data. LabVIEW contains a graphical programming model that is easy to use and is easy to visualise because LabVIEW programs consist of nodes connected by wires. Using LabVIEW reduces development time because of LabVIEW's built-in measurement and automation functionality. In addition, LabVIEW automatically handles many low-level programming concepts such as memory management and multi-threading. LabVIEW also includes features designed to create*

[1] www.ni.com

> *readable, safe, and reliable code. LabVIEW code easily interfaces with other programming languages, allowing developers to take advantage of the benefits of LabVIEW while always using the appropriate tools.* [10]

The QUD comes with c-library which cannot be used in LabVIEW directly. First the functions in the library must be linked to LabVIEW.

Regarding to a benchmark test[11] made by NI[2] an ...

> ... application developed using LabVIEW executes in the same amount of time as the same application would if it were built in a general-purpose programming language such as C. In some cases, applications built in LabVIEW run faster, in some cases, applications built in a general-purpose programming language run faster. On average, the application executes in the same time.

The *key* task for this project is to check if LabVIEW is able to store the data fast enough. (fig. 5.1

For the *File Write* test the task of the two programs was to write a string of 10,000,000 characters into a text file.[3]

Conclusion:
In average Labview takes only 19% more time to write the data to a file, which is negligible for this project. If there would be a major difference in the time it should be considered to use C instead of LabVIEW.

5.2. Use the C-Library in LabVIEW

To use external Code in LabVIEW it is necessary to include the C-Library in LabVIEW. The tool for this operation is the '*call library function node*'. This function is a link to the target library. It passes the inputs to the external library and receives the results to make it available inside LabVIEW. Every function inside the Library can be accessed with the '*call library function node*'. To do this it is necessary to enter the path of the library and to select the function. The

[2]National Instruments
[3]For all other tasks of the benchmark, please proceed to the cross-reference.

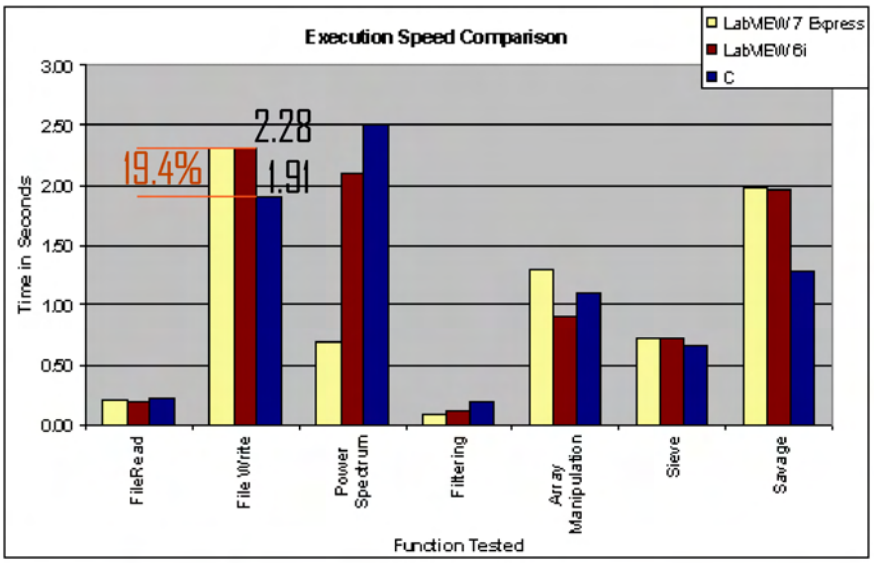


Figure 5.1.: LabVIEW benchmark : In this benchmark the same task has to be achieved by LabVIEW 7 Express, LabVIEW 6i and C. In the *File Write* task LabVIEW 7 Express was about 19% slower than C.[11]

'*call library function node*' will change its appearance depending on the amount of inputs and outputs of the c-library function.

Figure 5.2.: Symbol of the call_library_function_node : Interface to use an external C-Code in LabVIEW. It passes over the input values to the external library and receives the outputs.

The most important part is to choose the right calling convention for the library. In this case the QUD-library needs to be called in STANDARD_CALL-MODE (WINAPI), not in the C-MODE. This information depends on the library and the developer and should be supported by the developer. If the calling convention is wrong, the Labview will not be able to set up the link to the library and the program will crash.

Also the datatype of the variable must be correct. It is necessary to know if the variable is a value or a pointer to a value. If the data type is not correct or a value is assigned to a pointer the program will not work properly.

It is important to know that a data type in C is not equal a data type in LabVIEW. (table 5.1)

data type		**Explanation**
C	**LabVIEW**	
char	XXX	8-bit, signed or unsigned
signed char	I8	8-bit, signed
unsigned char	U8	8-bit, unsigned
int	XXX	either 16- or 32-bit type, signed
unsigned int	XXX	either 16- or 32-bit type, unsigned
short	I16	16-bit, signed
unsigned short	U16	16-bit, unsigned
long	I32	32-bit, signed
unsigned long	U32	32-bit, unsigned

Table 5.1.: Comparison of the data types in C and in LabVIEW : It is important to pass the variables in the correct data type to the c-library, otherwise the program will not work. In this table there are only the data types which are used in the program. [12]

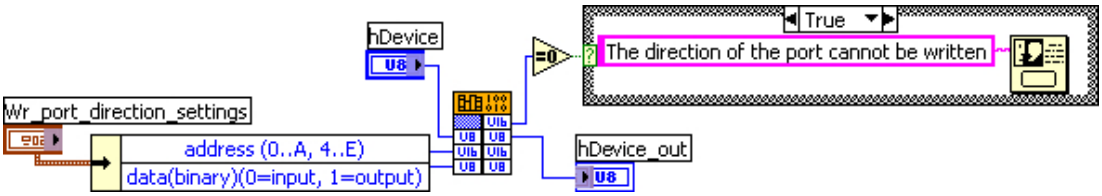


Figure 5.3.: Labview Code for using the *call library function node* to include a function of a c-library to LabVIEW. Depending on the amounts of inputs and outputs the *call library function node* will change its appearance.

Example using the QuickUsbWritePortDir-function:

Here is an example to link a function in the c-library to Labview. Full calling-name of the function:

int QuickUsbWritePortDir(HANDLE hDevice, unsigned short address, unsigned char data)

Regarding the manual of the QuickUSB module [8] the return value of type *integer* must be assigned in LabVIEW to a variable of type *unsigned 16-bit integer*.

The variable **hDevice** is a of type '*handle*' so in this case LabVIEW has to use an *unsigned 8-bit char* variable.

For the variable **address**(*unsigned short*) LabVIEW needs to assign an '*unsigned 16-bit integer*' and for the **data** (*pointer to an unsigned char*) an '*unsigned 8-bit char*' must be used.

Explanation of the LabVIEW code in Figure 5.3:
The variable **hDevice** and the cluster **Wr_port_direction_settings**, containing the two variables **address** and **data**, feed the *call library function node*-function. If an error occurs the return value will be zero and in this case a pop up-window will appear telling the user : *"The direction of the port cannot be written."*

A usual mistake in this case is that the device is not opened before or the hDevice(Device ID) is invalid. If the return value is not equal to zero the values of the input-cluster are successfully written to the QUD.

Like in this example all necessary functions of the c-library were linked to LabVIEW. For a list of all linked functions refer to the Appendix : *Functions in Labview.*

5.3. Structure of the LabVIEW program

After including all necessary functions of the QuickUSB-Library to LabVIEW, the code must be developed. First of all a basic structure of "How the program should work" must be determined.

The basic functions are:

1. All QUD-modules attached on the computer must be found.
2. Open the QUD which is attached to the computer to make it accessible for the other functions.
3. The QUD must be configured.
    - The FIFO of the HSPP must be configured in the desired way. (write_settings.vi) Refer to chapter 4.1.

- If a general purpose I/O is used, it has to be configured. (write_port_Dir.vi)

4. Waiting for the data.
5. Read data when it is available. Please refer to chapter 5.3.2.
   - Store the data on the hard-disc.
   - Display the data in an appropriate way.
6. Close the QUD when the program is stopped.

After linking the QuickUSB-Library to LabVIEW most of these functions are available in Labview. For this project not all features of the QUD-module are needed. Refer to chapter 5.3.1 for further information. A detailed description of the *Read_data* - block can be found in chapter 5.3.2.

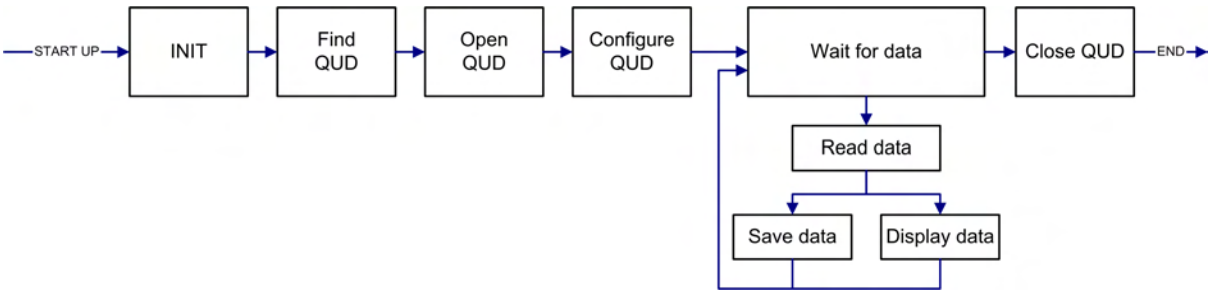


Figure 5.4.: Flowchart of the LabVIEW readout program.

In figure 5.5 the user interface of the LabVIEW read-out program. When the program is running the indicator *waiting for module* flashes to show that the program is searching for a connected QUD. When a QUD is found the indicator will stop flashing and the name of the QUD will appear next to the indicator. Now it is possible to change the settings of the QUD using the *Settings* box.

If it is not necessary to store the data on the computer, the check box *store data* can be turned off. The file for the data can be chosen in the text box *file path*. It is possible to choose an existing file or to append new data. If the file does not exist, it will be generated by the program.

When all settings are correct, the acquisition can be started using the button *read_data*. The acquisition process can be interrupted and continued using this button. The data will be shown in the display *data*. When the acquisition is done, the program can be stopped using the button *Stop*.

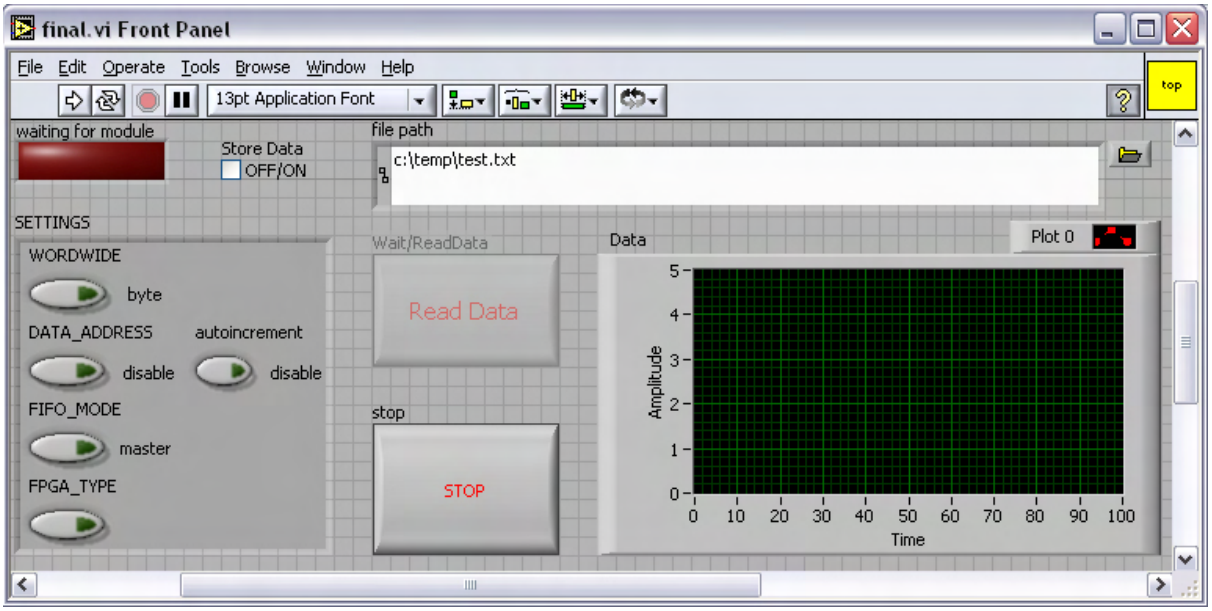


Figure 5.5.: User interface of LabVIEW program

5.3.1. Configure the QUD in LabVIEW

Because the settings of the QuickUSB-device are not stored in a non-volatile memory, it will lose all settings when it is not supplied with power. So it is necessary to configure the QUD every time it is connected to a computer.

To do this in LabVIEW the subVI '*write_settings.vi* can be used. With the input variable *settings* all functions of the QUD can be modified.

The actual values of the variables are shown next to the buttons. In this project the width of the data bus is 8 bit. The FIFO of the QUD must be configured as *master*. On the FPGA the data is stored in a FIFO.
Since the FIFO does not need an address bus, this settings should be disabled. Furthermore it is not necessary to program a FPGA in *passive serial* mode.

5.3.2. Read the data from the QUD

As long as the FPGA-FIFO is not full it collects the incoming data. When it is full and data is arriving, old data will be overwritten by the new data. To prevent a loss of data it is necessary to read out the data before. There are two possible versions of 'how to read the data'.

1. If the datarate of the incoming data and the size of the FPGA-FIFO is

known, it is possible to use a **timed loop** in LabVIEW to read the data on a constant timebase.

pro: The LabVIEW program is simpler. There is only a timed loop to read the data from the QUD. The timed loop will take care that the data is read from the FPGA-FIFO on a regular timebase.

con: This version needs more user interaction, and user need to know, how many values are in the FIFO after a certain amount of time.

If the time is too short, or the amount of value which should be read too big, the program will try to read more data than there is in the FIFO.

On the other hand side, if the time between two reading cycles is too long, or the amount of values to read too small, the data in the FIFO will increase, until the FIFO is full. When the design continues to write data into the FIFO, the new data will overwrite the old data, and the old data will be lost.

Another possibility for the read-out program is ...

2. ... to wait until the FIFO has reached a certain fill-level which is then indicated by a signal called *almost_full*). When this event occurs, the program reads a certain value of data from the FIFO . After the readout process there will still be some data in the FIFO, but this will remain inside the FIFO until the next readout. (fig. 5.6)

pro: If data is available Labview will read a number of data, otherwise the program will wait. The system is independent from the data rate of the data from the BeamLossMonitors, as long as the data rate is not too big to be transmitted via USB 2.0. When the data rate changes, the time slice to read the data will also change. If the data rate is low, the LabVIEW program will wait until there is enough data stored on the FIFO. If the data rate is high, the program will read the data more often to make sure that the FIFO will not overflow.

con: This *handshake* must be implemented to the Labview program. The program must wait for the *trigger* signal (*almost_full*) to read the

data from the FIFO on the FPGA. The LabVIEW program must periodically scan this signal to know, when data is available. When use a timed loop, LabVIEW will take care that this signal is scanned with a fixed timebase.

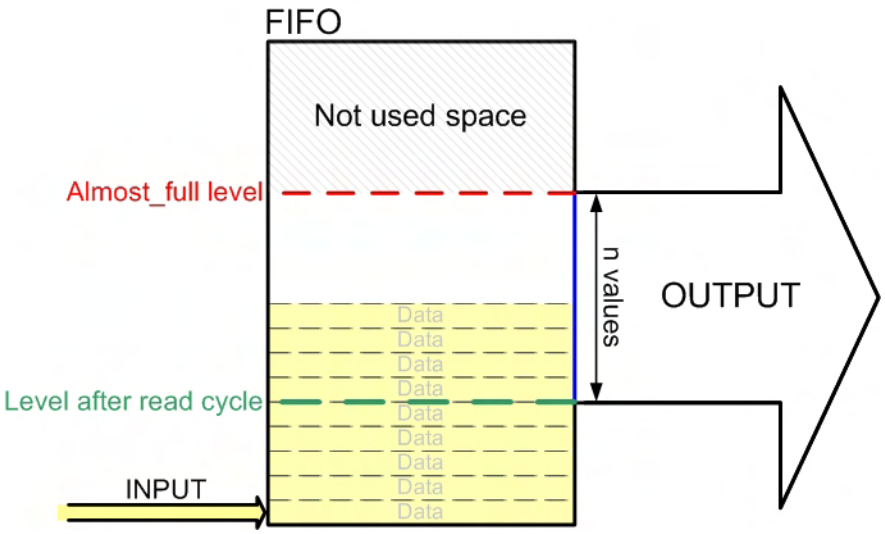


Figure 5.6.: Schematic view of the usage of the FPGA-FIFO. When data is arriving, it will be stored in the FIFO. When the FIFO reached a certain fill level this will be indicated by a Signal (*almost_full*). The LabVIEW program will scan this signal periodically. When a certain amount of data is stored in the FIFO LabVIEW will read n values from the FIFO.

Conclusion:

The first mode to ask for data after a certain amount of time needs more user interaction and the knowledge of the data-rate. Because of this it is vulnerable for errors due to the wrong setup. If one of the parameters is not adequate for the current system, the data will be lost, or the program will read wrong data, which is not good either.

In the *handshake* - mode the program will wait until enough data is stored in the FIFO. Depending on the datarate this can be between 25 milliseconds or even up to about 33 seconds (or 3 seconds when using a smaller threshold). 5.3.5

5.3.3. Realisation of the handshake

To realise this handshake one of the GPIO[4] ports of the QUD can be used. For this reason the functions *Write_prot_Dir* were also implemented to LabVIEW. This one pin is used for the *almost_full* - signal and must be configured as an input pin. The other signal pin which is used to notify the FIFO that data is requested (*Read_Data_Request*) belongs to the HSPP.

5.3.4. The readout procedure

When the LabVIEW program is running, it will ask the QuickUSB-device to periodically scan the signal *almost_full*. When this signal indicates that data is available, the program will force the QUD to read **n** values of data. The QUD will use the REN[5] - signal to indicate the FIFO on the FPGA that it should deliver data. As long as the the signal *Read_Data_Request*[6] is *1*, the FIFO will deliver data. This time depends on the amount of values that should be read.

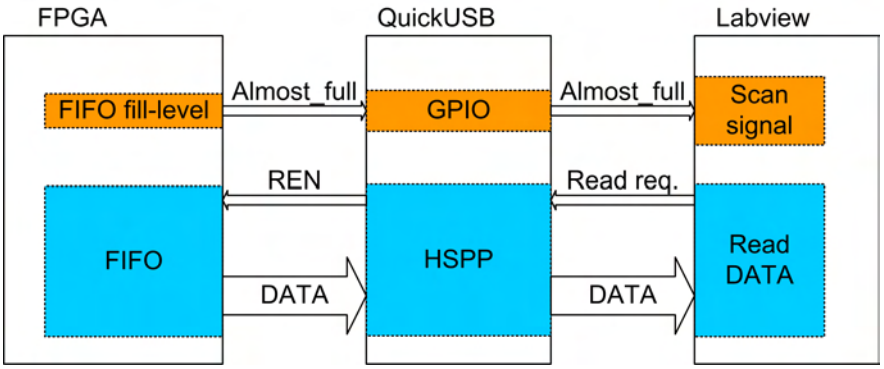


Figure 5.7.: Schematic of the transmission of the signals between the development FPGA-board and the PC LabVIEW program. Labview will wait until data on the FIFO is available. When LabVIEW is sending a read-request the QUD will forward this signal to the FIFO on the FPGA. As long as this read request is active, the FIFO will deliver data.

[4]General Purpose I/O
[5]Read Enable
[6]This signal is one of the control lines of the HSPP (=REN).

5.3.5. The data-rate of the system

The LabVIEW program needs to work properly for two versions of a system. In one system (**A**) the raw data of the BLM system must be transmitted. This means the data from up to 8 ionisation chambers plus some ID-data. This version will be used by Markus Stockner at the DESY laboratories in Hamburg.

The other system (**B**) delivers the maximum values of the running sums every second. (chapter 2.1)

5.4. Explanation of the Labview Code

In the next columns the most important parts of the Labview program *final.vi* will be explained. The full source code and the explanation of the subVIs can be found in the appendix.

The first element is a stacked sequence (fig.5.8). This will run once when the program is started. In this sequence all buttons and values were initialled to their default value.

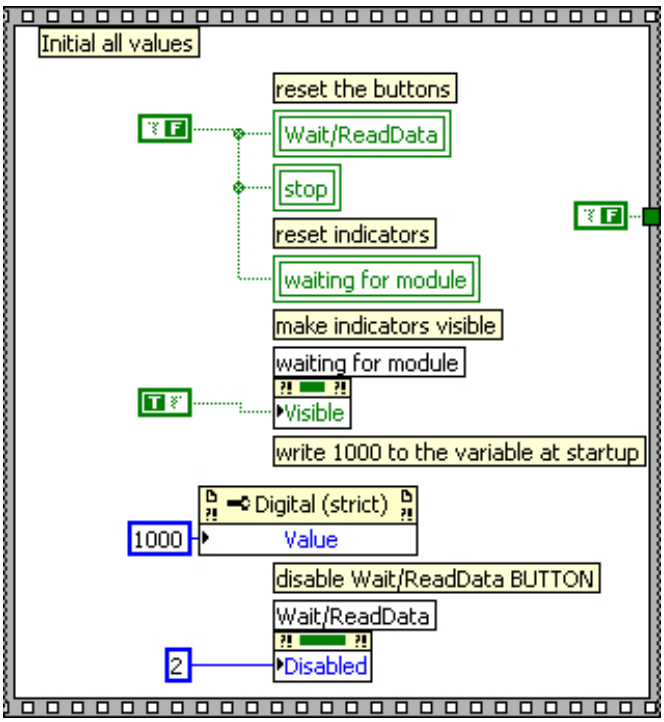


Figure 5.8.: Picture of the initializing part of the LabVIEW program.

The next element (fig.5.9) is a timed loop with the function *find_module.vi*. As the code will only receive data when a module is found on the system, the program will stay in this loop until a module is found. When more than one modules are found the first module will be chosen. This can be modified by changing the value of the *index array* - function. [7]

To indicate the loop until a module is found, the indicator will flash during the search. When one or more module is found this indicator will disappear and the loop will be terminated. The recently found module will now be opened (*open_device.vi*), so that it can be used by the next modules.

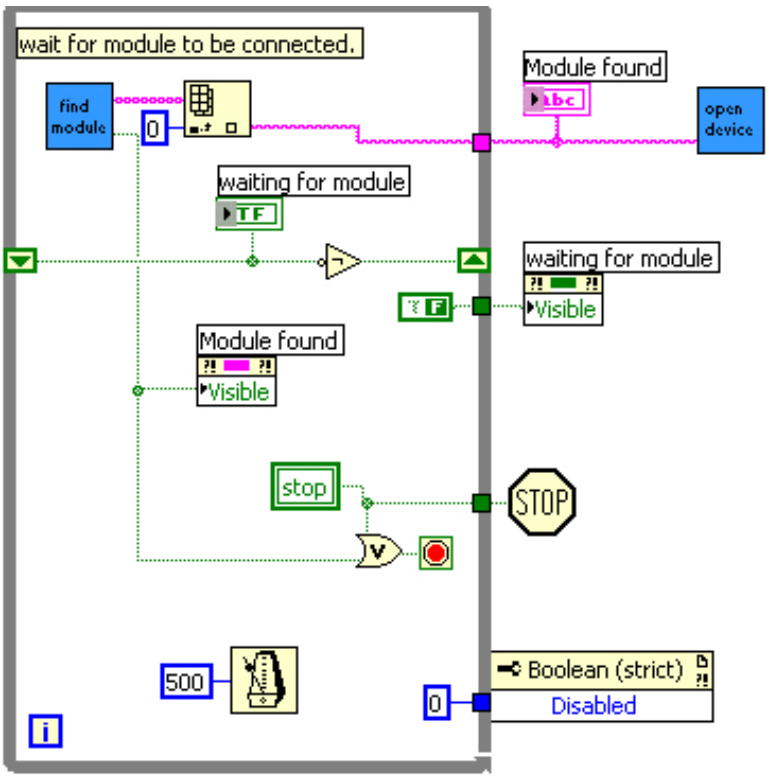


Figure 5.9.: Picture of the wiat-part of the LabVIEW program. The program will wait until a QUD is found on the computer. A flashing LED will indicate that the program is searching for a module.

The next operation(fig.5.10, outer while-loop) is to modify the settings for this project. As mentioned above all settings must be written after the Power ON, because they are stored in a volatile memory. For the *almost_full* signal, the pin 0 of port A is configured as input by the function *Write_Port_Dir.vi*. All this steps

[7] value = 0 → first module, 1 → second module will be chosen

will be run without user interaction when the program is started and a QUD is connected.

Now when the module is fully configured the program waits for the user to start the acquisition by pressing *Wait/ReadData* (fig.5.10, inner while-loop). Before the user does this it is possible to enter a new filename for the data. If the file already exists, the program will append the new data after the old data in the file, so no data will be overwritten by accident.

The core of the code is the *while* - loop where the signal *almost_full* is checked (fig. 5.10). In this part of the code the function *Read_port.vi* scans the Pin 0 from Port A continuously. Which pin should be scanned can be changed by the inputs of the subVI *U8array_2_bit*. If the value of this signal is **false**[8] data is available in the FPGA-FIFO. For development the boolean signal *almost_full* is converted to an integer number (0 or 1) and connected to a chart display.

When the user presses the button *ReadData*, the program will periodically scan the Signal *almost_full*. In case that there is enough data in the FIFO the program will read a certain amount of values. The subVI *modify_dataarray.vi* will concatenate the two 8 bit arrays to one 16 bit array. The the data can be stored and displayed.

The user can interrupt and continue the acquisition by pressing *Wait/ReadData*. When the program is ended by pressing *Stop* the QUD will be closed by the subVI *close_device*.

Indicators in the program

There are plenty of indicators in the program to visualise what is going on. First of all there is the indicator *'waiting for module'* which is flashing while no module is found on the system. This indicator will disappear when a module is found. The most important one is the chart-indicator for the incoming data. The maximum values of the x- and y-axis can be changed by selecting them with the mouse and type in the new value. The numeric indicator *'check for data'* is connected to the iteration counter of the loop. The indicator *'read data and store'* is a counter which only increments when data is read from the FIFO and stored on the hard-disc.

[8] The signal *almost_full* uses negative logic. When the value is **true** the FIFO is accumulating data. When it is **false** the FIFO is "almost_full"

Developer version of the program

For debugging and for the tests in chapter 6 a special version of the program was used. The difference is that the signal *almost_full* and the loop-time are also stored in a file during the acquisition. These values are necessary to verify that the read-out program is working properly. The code of the developer version can be found in the appendix.

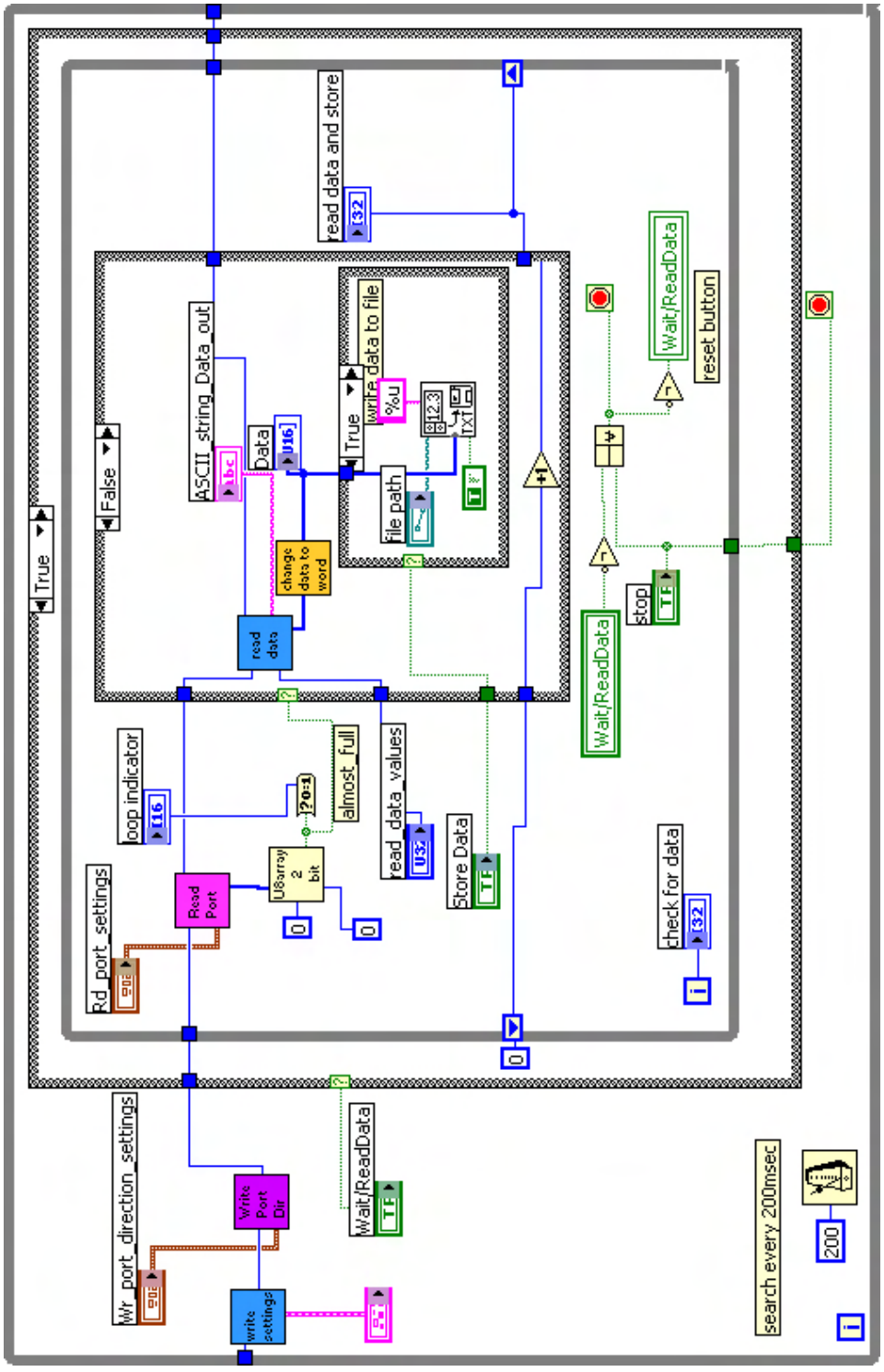


Figure 5.10.: Picture of the *wait_for_data*-part in LabVIEW. The loop will scan the signal. When data is available, it will be read and stored to a file.

6. Test measurements

On the next pages the setup of the test measurements and the results are shown. Two tests were made to prove the functionality of the system. The first will work with the reduced data-rate and will prove that the data is stored correctly on the computer. The second test will work with the full data-rate to prove that the system is fast enough.

As mentioned before the FPGA-system has two test modes which can be activated by using the switches. Because the running sums calculation was not fully implemented when this measurements were made, the test of the low-datarate version will use one of the test modes of the system. The other test will use an external device which sends data over the fibres to the system.

Specifications of the computer:

- P4 - 3.0GHz
- 512 MB RAM
- Win XP

6.1. Test of the version : LOW DATA-RATE

To prove that the data is stored correctly on the computer this test uses the onboard test functions. The test-data is taken from the CRC-block. The data-rate of this test is equal to the date-rate when transmitting only the reduced data-frame.

6.1.1. Measurement setup

The test values are stored in a memory and can be accessed by using the switches of the FPGA board. These values reach from 0 to 255 and have the shape of a

sawtooth. The threshold of the system is set to 1024 values, the update rate is $\frac{256 words}{0.8388 sec}$. This means that every 3.355 seconds a bunch of 1024 values are read from the FPGA and stored on the computer.

Settings of the switches for this test:

Switch 1: 1 : The FPGA is in test mode.

Switch 2: 1 : The data amount is equal to the reduced data-frame.

Switch 3: 0 : The data is produced on the FPGA for tests.

Switch 4: X[1] (both possible)

6.1.2. Results

In figure 6.1 the results of two read cycles *(2 times 1024 values)* are shown. The values reach from 0 to 255 as expected. No data is lost during the transmission. The interface is working as expected.

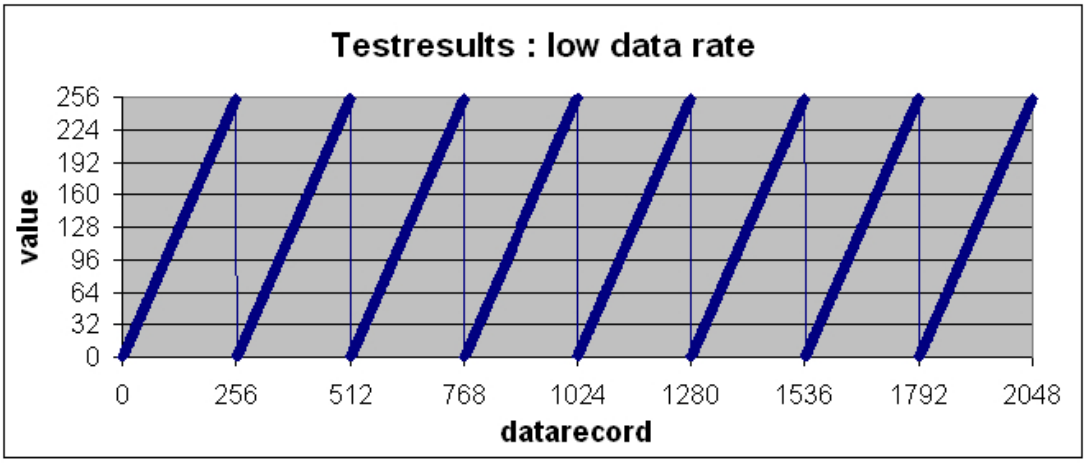


Figure 6.1.: Test data saw tooth values as function of the data index. (low datarate)

6.1.3. Conclusion

Due to the low data-rate of this test, Labview has no timing problem to read and store the data. The stored data is equal to the data which is send. This means, that this version of the system is working correctly.

[1]Switch4 has no influence to the measurement, because it changes only the behavior of LED1.

6.2. Test of the version : HIGH DATA-RATE

Because the tunnel-card is not ready for tests, a fibre test module(fig. 6.2) is used for this test. Using this module it is possible to send any values over the fibres. This is the final test of the whole system starting from the mezzanine card to the computer with the full data-frame.

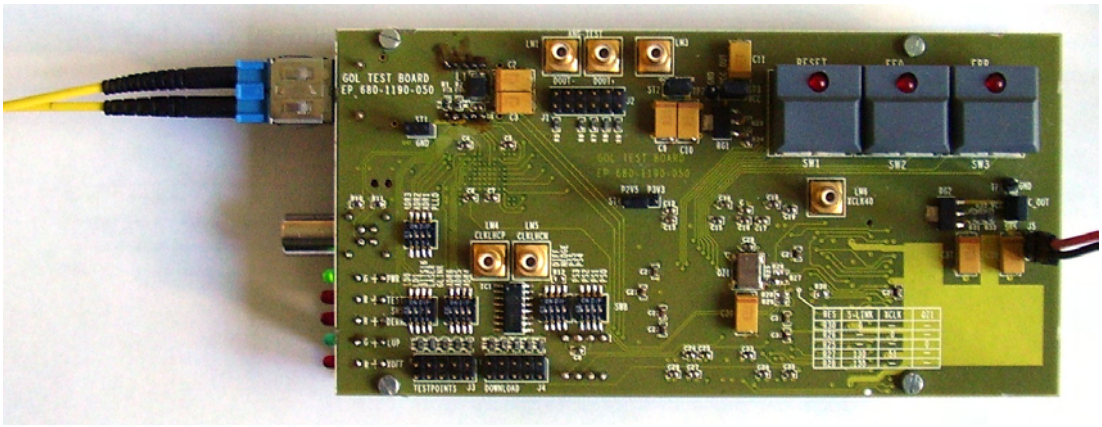


Figure 6.2.: Fibretest module

6.2.1. Measurement setup

To test the whole chain of the system the data must arrive from the fibres. In this measurement an external device is used to generate a test-pattern and send it with a laser-diode via the fibres.

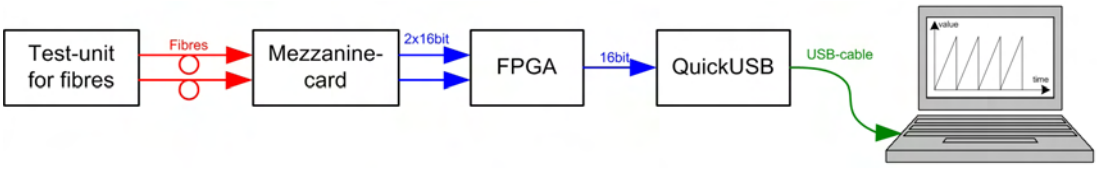


Figure 6.3.: A block diagram view of the test system.

In figure 6.3 a block diagram of the test system is shown. The picture below (fig. 6.4) shows the arrangement of the devices. The additional device on top of the mezzanine card is to rearrange the connectors on the mezzanine so that it fits on the tunnel-card. In this setup it is necessary to connect the signal via a cable to the FPGA-board.

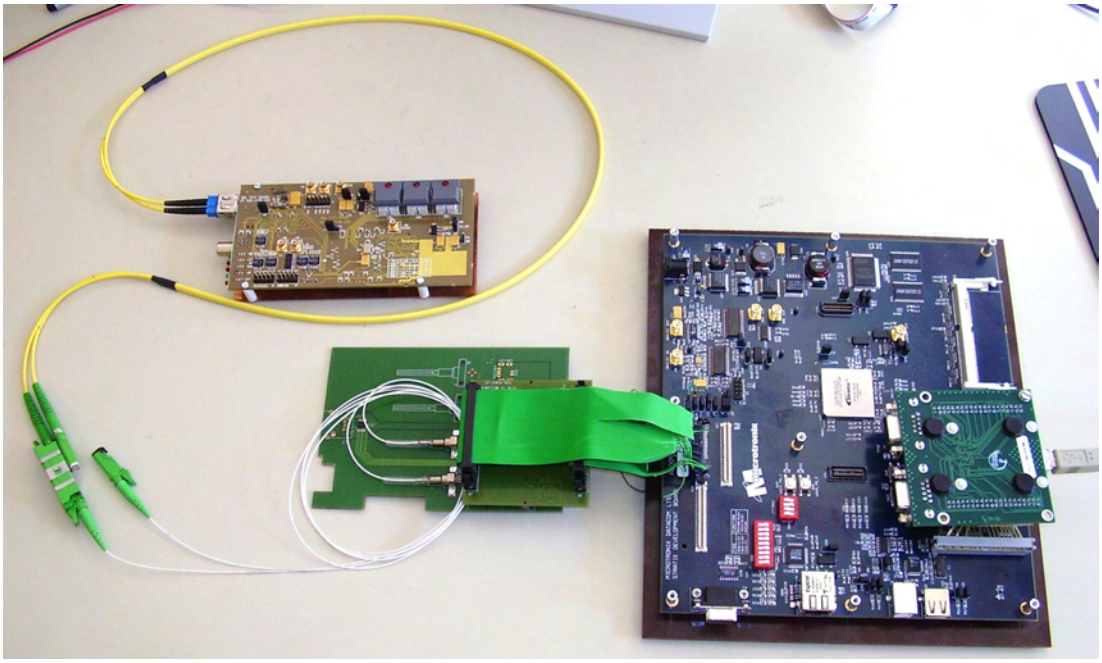


Figure 6.4.: A picture from the test system.

Settings of the switches for this test:

Switch 1: 0 : The data is taken from the input of the fibres.

Switch 2: 0 : The full data-frame is transmitted.

Switch 3: X[2] : (both possible)

Switch 4: X[3] : (both possible)

This test is realised in two versions:

In one version the data is displayed in LabVIEW. In the other version all displays in LabVIEW are disabled. This is to show the influence of the displays to the looptime of the program.

To verify that all data is stored and the FPGA-FIFO did not overrun, the signal *almost_full* and the LabVIEW variable *loop-time* were also stored. With this values it can be verified that LabVIEW was able to read the data from the QuickUSB and store it on the hard-disc in time. When data-rate is too high so that LabVIEW cannot empty the FPGA-FIFO fast enough, the signal *almost_full*

[2]Switch3 not relevant because it changes only which test data is chosen.
[3]Switch4 has no influence to the measurement, because it changes only the behavior of LED1.

will stay *low*. This means that even when data is read from the FPGA-FIFO the data is exceeding the threshold. When the FPGA-FIFO is read continuously without overrun, this signal will be 1 for about 25 milliseconds (refer to chapter 3.5) when the threshold is reached for one loop-cycle 0. The time of the loop will be higher because the program needs more time to read the data from the USB interface and store it on the computer.

Loop-time of the program

The loop-time can be calculated after the acquisition using the stored values of the LabVIEW-variable *looptime*. The smallest value of this signal is 1 msec which is given by LabVIEW itself. When this variable stays 0 for five cycles and then the value is 1 for one cycle, that means that this value of this variable was 5 loops below 1 millisecond. This means further that the program made 6 loops in about 1 millisecond.

The time between the read_out cycles can be estimated by summing up the loop-times.

e.g.: 16 x 1 msec + 1 x 9 msec (for the storage process) = 25 msec. (fig. 6.7)

This time should be equal to the time it takes until the data in the FPGA-FIFO reaches the threshold which is about at 25 milliseconds.

6.2.2. High data rate transmission results : with the LabVIEW display

Figure 6.5 shows the signals *almost_full* and *looptime* at each cycle of the program.

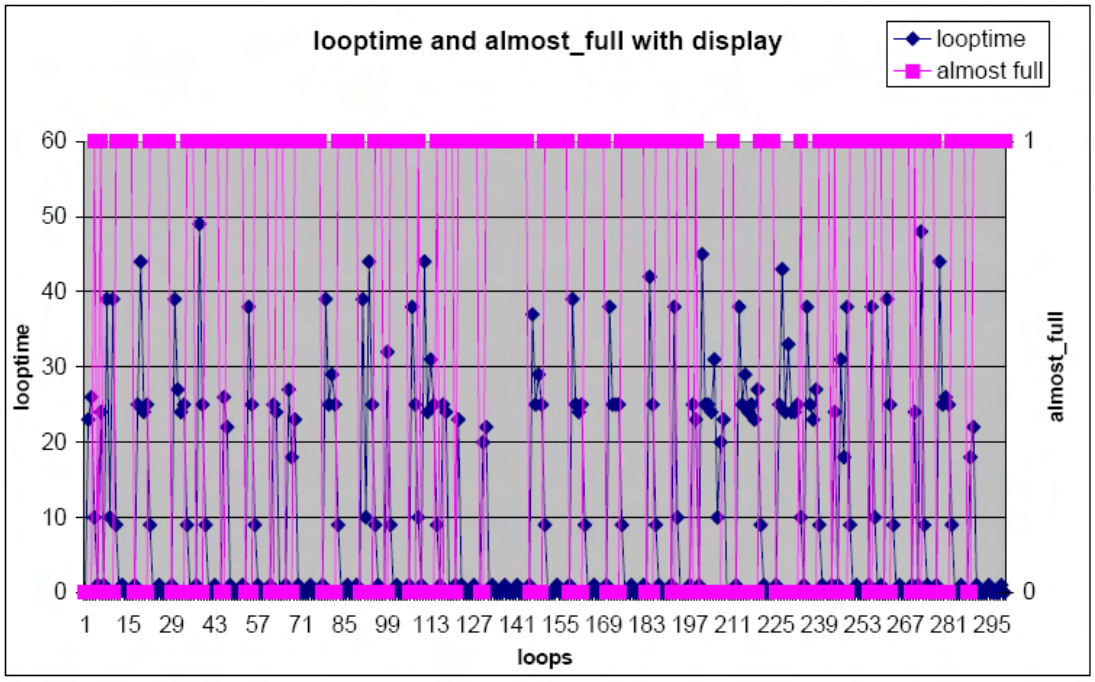


Figure 6.5.: Read out cycle loop time as function of the loop index. In this test the displays were activated.

6.2.3. High data rate transmission results : without the LabVIEW display

In figure 6.6 a small part of the results from the stored file are shown. Figure 6.7 shows the signals *almost_full* and *looptime*. The values of the looptime are summed up after each read-out cycle and displayed as signal *looptime absolut*.

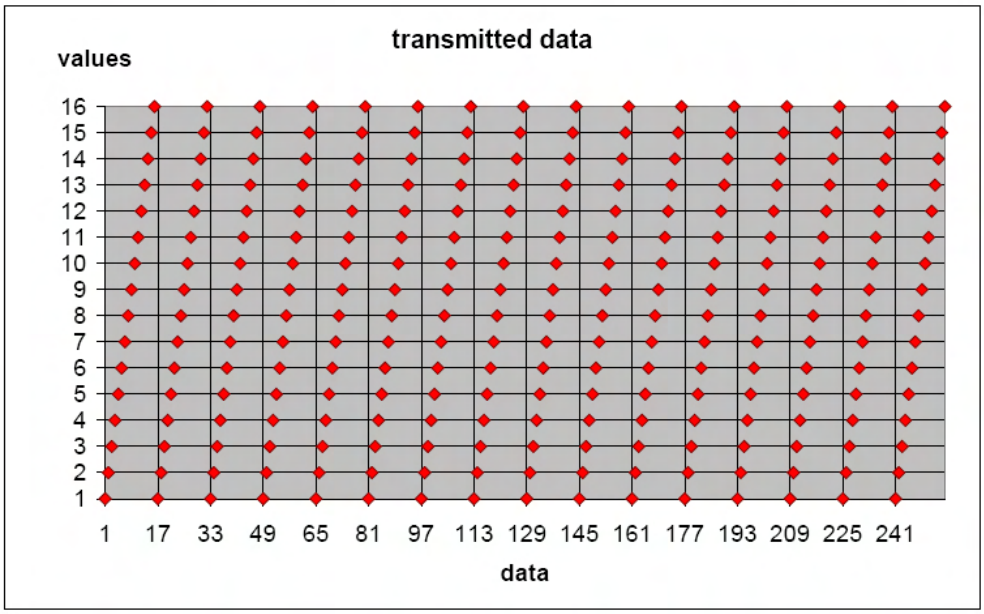


Figure 6.6.: Test saw tooth values generated by the optical transmitter as function of the data index. This is a short sector or the data stored on the computer. The displays are disabled during the acquisition.

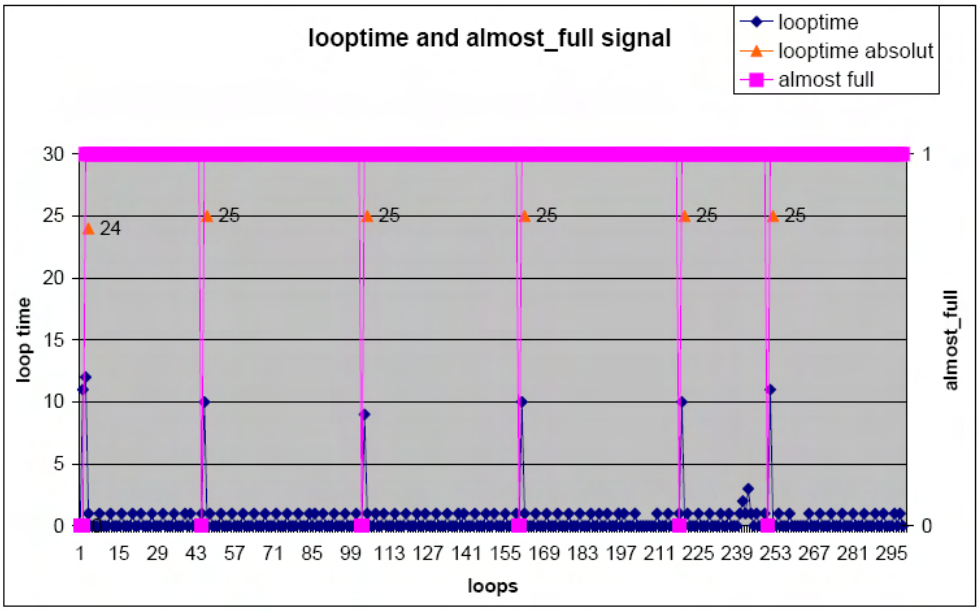


Figure 6.7.: Read out cycle loop time as function of the loop index. In this test the displays were deactivated to increase read out speed. The timing of the program can be estimated using the time of the loops.

6.2.4. Interpretation of the results

With display

When the display is enabled (figure 6.5) the loop-time is up to 50 milliseconds. This causes the FIFO to overrun continuously and turns the value of the signal *almost_full* to 0 for more than one cycle. The whole procedure takes longer to update the displays, because the data cannot be read in time they are lost.

It is recommended for the acquisition with the full datarate to disable the displays. The actual stored values are not displayed here, because it is obvious that this data is not correctly stored to the hard-disc.

Without display

In figure 6.6 the correctly stored data without the display are shown. All data are stored correctly on the computer.

In the figure 6.7 it is shown, that the computer scans the signal *almost_full* continuously, until the data exceeds the threshold and is transmitted to the computer. As expected, the signal *almost_full* in figure 6.7 remains at 1 most of the time. Only when data is in the FPGA-FIFO it changes the value for **one** cycle. At loop number 239 the value of the looptime is up to 3. In this moment the LabVIEW program needs more time to finish the loop, because the computer was busy with another operation.

7. Conclusions

7.1. FPGA

The implementation of the handshake allows that the system can run without user interaction. When data are in the FPGA-FIFO they will be transmitted to the computer. If not, the FPGA-FIFO will wait and accumulate the data. Because of the indicators on the board it is possible to check if the system is working and if data are arriving. The possibility to change behaviour of the system is important to check if there is a problem on the FPGA or on the transmission line.

Improvements

When the full functionality is implemented in the FPGA, it might be possible to increase the size of the FPGA-FIFO.

7.2. QuickUSB

Using the QuickUSB module from bitwisesys was a good idea to add a common computer interface to the project. A big advantage is that the library is already included in the package, so it is easy to develop a program in C, VisualBasic or in this case LabVIEW. One disadvantage is that the module cannot transmit 16 bit at once. The library can only send 2 times 8 bit.

7.3. LabVIEW

The possibility to link an external library to LabVIEW and make the functions available increases the operational area of LabVIEW.

Regarding the positive results of the tests LabVIEW is able to handle the high data-rate without the displays.

Improvements

There is room for several improvements. To proove that the system is working as defined the transmitted data was stored and displayed. When sending real data not every bit of this signal is necessary for the display. A display which shows the values of the 8 chambers in a graph will ease the data check.

Another improvement will be a warning indicator, using the signal *almost_full*, to show that the FPGA-FIFO overruns when it is 0 for more than one loop.

Furthermore it is possible to use another signal of the QUD to tell the LabVIEW program the actual data-rate of the system. It will then change the amount of data that should be read automatically and without user intervention.

8. Abbreviations, list of figures and list of tables

Abbreviations

CERN : Conseil Europèen pour la Recherche Nuclèaire

LHC : Large Hadron Collider

BLM : Beam Loss Monitors

ASIC : Application Specific Integrated Circuit

ADC : Analog digital converter

VHDL : VHSIC Hardware Description Language

VHSIC : Very High Speed Integrated Circuit

VI : Virtual Instrument (Program in Labview)

subVI : subprogram in Labview

FPGA : Field-Programmable Gate Array

QUD : QuickUSB-Device

HSPP : High-Speed Parallel Port

GPIO : General Purpose I/O

FIFO : First In, First Out

List of Figures

List of Tables

Bibliography

[1] internal Document Server of the BLM-section, April 2005

[2] internal CERN Document Server, April 2005

[3] Arauzo A., Dehning D., Ferioli G., Gschwendtner E., *"LHC Beam Loss Monitors"*, CERN-SL-2001-027-BI, 5th European Workshop on Diagnostics and Beam Instrumentation, Grenoble, France, 13-15 May 2001.

[4] Quench levels and transient beam losses in LHC magnets, J.B. Jeanneret, D. Leroy, L. Oberli and T. Trenkler, LHC Project Report 44, July 1996.

[5] Friesenbichler W., *"Developement of the of the Readout electronics for the Beam Loss Monitors of the LHC"*, June, 2002

[6] Hodgson M., *(*"Beam Loss Monitor Design Investigations for Particle Accelerators"), April, 2005

[7] Microtronix: *"Stratix Development Kit Product Prochure"*,
`http://www.microtronix.com`, April 2005

[8] Bitwise Systems: *"QuickUSB User's-Guide"*,
`http://www.bitwisesys.com`, Version 1.30, March , 2003

[9] Universal Serial Bus: *"USB 2.0 Specification"*,
`http://www.usb.org/developers/docs/`, April 2002

[10] National Instruments: *"Why Learn LabVIEW?"*,
`http://zone.ni.com/devzone/conceptd.nsf/webmain/B149ABA93A241CFE86256EE600648668`, April 2005

[11] National Instruments: *"Benchmark Execution Speed of LabVIEW Applications"*,
`http://zone.ni.com/devzone/conceptd.nsf/webmain/DC9B6DD177D91D6286256C9400733D7F`, April 2005

[12] Christian Gindorf *"Datentypen in LabVIEW"*,
`http://www.christianhamp.privat.t-online.de/LabVIEW/Datentypenubersicht/datentypenubersicht.html`, Januar 2004

9. Appendix

A. VHDL-Code and Quartus-Projectfiles

A.1. Project Package

```
library ieee;
use ieee.std_logic_1164.all;
--package-declaration     types, subtypes, constants,
package proj_pak is
--Type declaration----------------
subtype sl  is std_logic;        --  alias for std_logic;
subtype slv is std_logic_vector; --  alias for std_logic_vector;
end proj_pak;

package body proj_pak is
end package body;
```

A.2. Testmodule

```
-- librarys ----------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.proj_pak.all;
---------------------------------
entity testmodul_0_to_5 is
    Port (
          -- clk
          clk_40Mhz         :  in  sl;
          reset_int         :  in  sl;
          -- output signals
          data_CRC          :  out  slv(15 downto 0);
          nRead             :  out  sl    -- read CRC
          );
end testmodul_0_to_5;

architecture struct of testmodul_0_to_5 is
   -- signals
signal cnt       : integer ;
signal cnt_data  : slv(15 downto 0) ;
signal nread_ind : sl ;
```

```
    signal pkg_cnt   : integer;
    signal nread_ind_shift : slv(1 downto 0);
    begin  -- struct

    process_nRead : process(clk_40Mhz, nread_ind) is
    begin
      if rising_edge(clk_40Mhz) then
        nRead <= nRead_ind;
          if cnt = 0 then      -- count up
            nRead_ind <= '0';
            cnt <= cnt + 1;
          elsif cnt = 5 then  -- count from 0 to 5
            nRead_ind <= '1';
            cnt <= cnt + 1;
          elsif cnt >=  40000 then   -- 40000 every every 1msec a package
            cnt <= 0;          -- reset counter
          else                 -- count up,
            cnt <= cnt + 1;
          end if;
      end if;
    end process process_nRead;

    data_incr : process(clk_40Mhz, nread_ind, cnt_data) is
    begin
      if rising_edge(clk_40Mhz) then
        if nRead_ind = '0' then
          cnt_data <= cnt_data + '1';
        elsif nread_ind ='1' then
          cnt_data <= X"ffff";
        end if;
      end if;
    data_CRC <= cnt_data;
    end process data_incr;
    end struct;
```

A.3. write_data

```
entity wr_data is
    Port (
          -- clk
          clk_40Mhz          :  in  sl;
          -- input signals
          nRead_en           :  in  sl;
          data_IN            :  in  slv(15 downto 0);

          -- output signals
          wr_data            : out slv(15 downto 0);
          wr_address         : out slv(4 downto 0) := "11111";
          write_en           : out sl;
```

```
          write_done         : out sl

          );
end wr_data;

architecture struct of wr_data is
-- signals
signal write_done_int : sl;
signal q1 : sl ;
signal q2 : sl ;
signal q3 : sl ;
signal buf : slv(15 downto 0) := (others => '1');
signal wr_address_cnt : slv(4 downto 0) := "11111";
signal write_enable_int   : sl;
begin  -- struct

process_write_M512 : process (clk_40Mhz ) is
begin
  if rising_edge(clk_40Mhz) then
    wr_address <= wr_address_cnt;
      if nRead_en = '0' then                    -- data available
        wr_address_cnt   <= wr_address_cnt + 1; -- increment address counter
        write_enable_int <= '1';                -- write data into M512

        if wr_address_cnt = 31 then
          write_done_int <= '1';                -- write done
          wr_address <= (others => '1') ;       -- reset counter
        else
          write_done_int <= '0';                -- write in progress
        end if;

      else
        wr_address_cnt <= (others => '1') ;     -- reset counter
        write_enable_int <= '0';                -- dont write data
        write_done_int <= '1';             -- write done
      end if;
  end if;
end process process_write_M512;

delay_data : process(clk_40Mhz) is
begin  -- pull data through modul to delay it for 1 cycle,
  if rising_edge(clk_40Mhz) then
    buf <= data_in;
    wr_data <= buf;
  end if;
end process delay_data;

delay_write_en : process(clk_40Mhz) is
begin  -- delay write_en for 1 cycle
  if rising_edge(clk_40Mhz) then
```

```
    write_en <= write_enable_int;
  end if;
end process delay_write_en;

delay_write_done : process(clk_40Mhz) is
begin -- delay signal write_done for 3 cycles
  if rising_edge(clk_40Mhz) then
      q1 <= write_done_int;
      q2 <= q1;
      q3 <= q2;
      write_done <= q3;
  end if;
end process delay_write_done;

end struct;
```

A.4. read_data

```
   entity rd_data is
    Port (
          -- clk
          clk_40Mhz          :  in  sl;
          clk_48Mhz          :  in  sl;
          n_reset            :  in  sl;
          -- input signals
          write_en           :  in  sl ;
          -- output signals
          rd_address         : out slv(4 downto 0) := "11111";
          read_en            : out sl;
          write_FIFO         : out sl
          --tb_START, tb_STOP  : out sl;
          --tb_shiftreg        : out slv( 1  downto 0);
          --tb_cnt             : out slv(6 downto 0)
          );
end rd_data;

architecture struct of rd_data is

signal shiftreg : slv(1 downto 0);
signal START : sl ;
signal cnt : slv(6 downto 0) ;
signal STOP : sl;

-- shiftreg for read_en and rd_address
signal read_en_int :sl ;
signal q1,q2,q3 : sl;
signal rd_address_int : slv(4 downto 0);
signal z1,z2,z3 : slv(4 downto 0);
```

```
-- shiftreg for write_FIFO
signal w1,w2,w3 : sl;

begin --struct
-------------------------------------------------------------
--0
--tb_START <= START;
--tb_STOP  <= STOP;
--tb_shiftreg <= shiftreg;
--tb_cnt   <= cnt;

shiftreg_write_en : process(clk_40Mhz) is

begin
  if rising_edge(clk_40Mhz) then

    shiftreg <= shiftreg(0) & write_en ;
  end if;
end process shiftreg_write_en;
-------------------------------------------------------------

-------------------------------------------------------------
--1

Startsignal : process(clk_40Mhz) is
begin
  if rising_edge(clk_40Mhz) then
    case shiftreg is
         when "10" =>
           START <= '1';
         when "01" =>
           START <= '0';
         when others =>
           START <= START;
    end case;
  end if;
end process startsignal;
-------------------------------------------------------------

-------------------------------------------------------------
--2
counter  : process(clk_48Mhz) is
begin
  if rising_edge(clk_48Mhz) then
    if START = '1' then
      if cnt < 38 then
        cnt <= cnt + 1;
      else
        cnt <= cnt;
      end if;
    else
```

```
      cnt <= (others => '0');
    end if;
  end if;
end process counter;
-------------------------------------------------------------------

-------------------------------------------------------------------
--3
STOPsignal : process(cnt) is
begin
  if cnt < 32 then    -- cnt from 0 to 31,  == 32 cycles
    STOP <= '0';
  else
    STOP <= '1';
  end if;
end process STOPsignal;
-------------------------------------------------------------------

-------------------------------------------------------------------
--4
--ausgaenge synchron mit 48Mhz
outpt_decoder : process(clk_48Mhz, START, STOP, cnt) is
begin

  if rising_edge(clk_48Mhz) then
    if (START and (not STOP)) = '1' then
      read_en_int <= '1';
      rd_address_int <= cnt(4 downto 0);
    else
      read_en_int <= '0';
      rd_address_int <= (others => '0');
    end if;
  end if;
end process outpt_decoder;
-------------------------------------------------------------------

-------------------------------------------------------------------
--5
shift_read_en : process (clk_48Mhz) is
begin
  if rising_edge(clk_48Mhz) then
    if n_reset = '1' then
      q1      <= read_en_int;
      q2      <= q1;
      q3      <= q2;
      read_en <= q3;
    else
      q1      <= '0';
      q2      <= '0';
      q3      <= '0';
      read_en <= '0';
```

```
    end if;
  end if;
end process shift_read_en;

shift_address : process (clk_48Mhz) is
begin
  if rising_edge(clk_48Mhz) then
    if n_reset = '1' then
      z1         <= rd_address_int;
      z2         <= z1;
      z3         <= z2;
      rd_address <= z3;
    else
      z1         <= (others => '0');
      z2         <= (others => '0');
      z3         <= (others => '0');
      rd_address <= (others => '0');
    end if;
  end if;
end process shift_address;
-------------------------------------------------------------
-- For FIFO write_FIFO must be synchronous with the data
-- => data 2cycles delay -->
shift_write_FIFO : process (clk_48Mhz) is
begin
  if rising_edge(clk_48Mhz) then
    if n_reset = '1' then
      w1         <= q3;
      w2         <= w1;
      write_FIFO <= w2;
    else
      w1         <= '0';
      w2         <= '0';
      w3         <= '0';
      write_FIFO <= '0';
    end if;
  end if;
end process shift_write_FIFO;

end struct;
```

A.5. Quartus Project version 1 : dual-clock Memory and FIFO

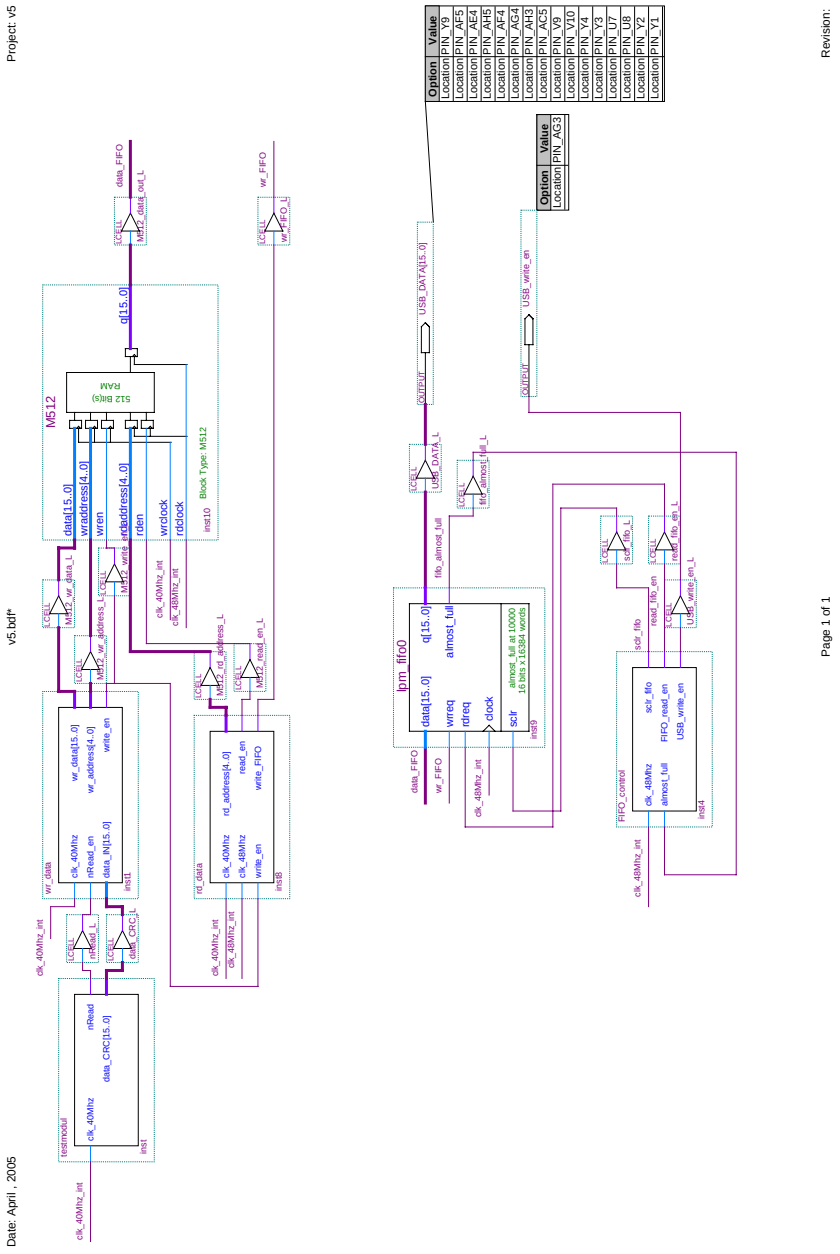


Figure A.1.: FPGA Version 1 : dual-clock Memory and FIFO

A.6. Quartus Project version 2 : dual-clock FIFO

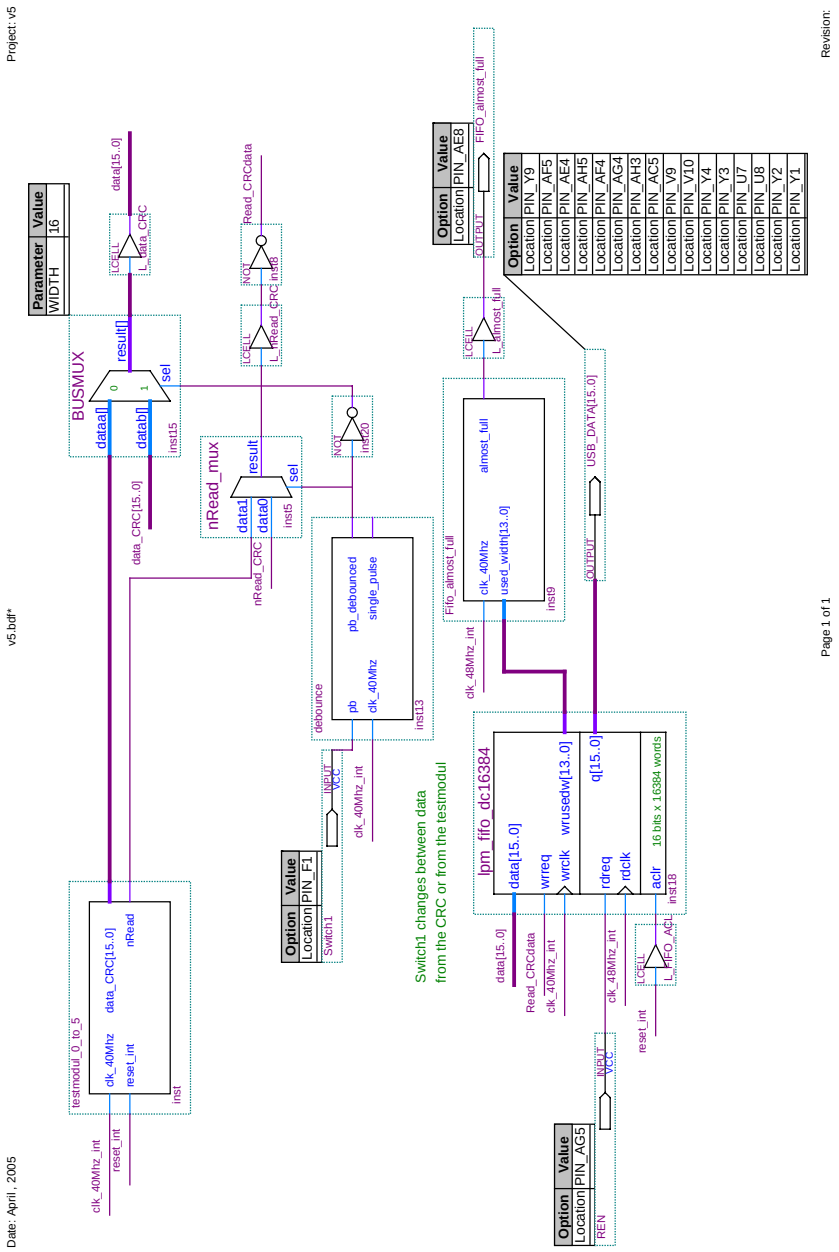


Figure A.2.: FPGA Version 2 : dual-clock FIFO

A.7. Schematic : PLL

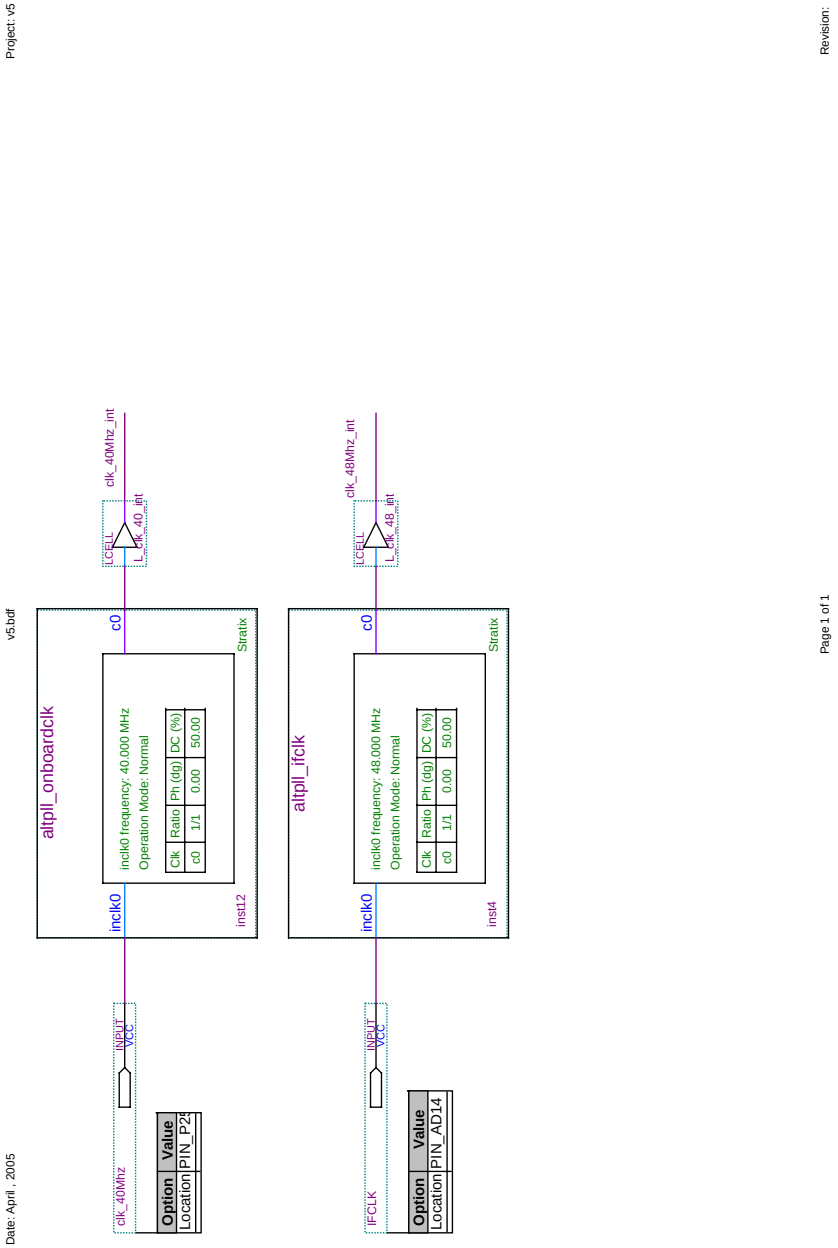
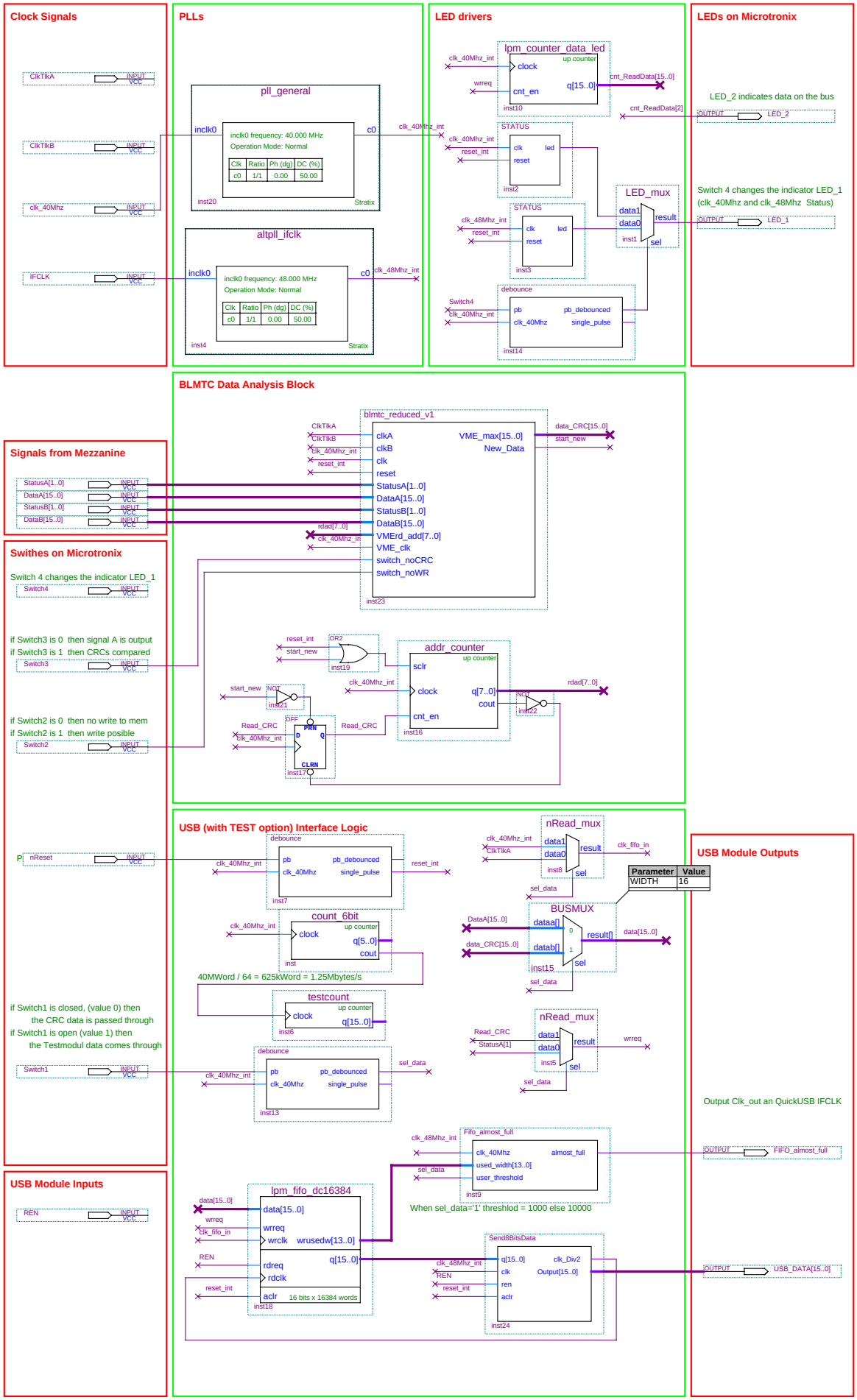


Figure A.3.: FPGA : PLL

A.8. Final structur of Quartus program(assembled by Christos Zamantzas)


Clock Signals
ClkTlkA
ClkTlkB
clk_40Mhz
IFCLK
INPUT
VCC
PLLs
pll_general
inclk0
c0
inclk0 frequency: 40.000 MHz
Operation Mode: Normal
Clk | Ratio | Ph (dg) | DC (%)
c0 | 1/1 | 0.00 | 50.00
inst20
Stratix
clk_40Mhz_int
altpll_ifclk
inclk0 frequency: 48.000 MHz
clk_48Mhz_int
inst4
LED drivers
lpm_counter_data_led
up counter
clock
cnt_en
q[15..0]
wrreq
cnt_ReadData[15..0]
cnt_ReadData[2]
inst10
STATUS
clk
led
reset
inst2
LED_mux
data1
data0
result
sel
inst1
inst3
debounce
pb
pb_debounced
clk_40Mhz
single_pulse
Switch4
inst14
LEDs on Microtronix
LED_2 indicates data on the bus
OUTPUT
LED_2
Switch 4 changes the indicator LED_1
(clk_40Mhz and clk_48Mhz Status)
LED_1
BLMTC Data Analysis Block
blmtc_reduced_v1
ClkTlkA
ClkTlkB
clk_40Mhz_int
reset_int
clkA
clkB
clk
reset
StatusA[1..0]
DataA[15..0]
StatusB[1..0]
DataB[15..0]
VME_max[15..0]
New_Data
data_CRC[15..0]
start_new
rdad[7..0]
VMErd_add[7..0]
VME_clk
switch_noCRC
switch_noWR
inst23
Signals from Mezzanine
Swithes on Microtronix
Switch 4 changes the indicator LED_1
Switch4
if Switch3 is 0 then signal A is output
if Switch3 is 1 then CRCs compared
Switch3
if Switch2 is 0 then no write to mem
if Switch2 is 1 then write posible
Switch2
OR2
inst19
addr_counter
sclr
q[7..0]
cout
inst16
INV
inst21
inst22
DFF
PRN
D
Q
CLRN
Read_CRC
inst17
USB (with TEST option) Interface Logic
nReset
if Switch1 is closed, (value 0) then
the CRC data is passed through
if Switch1 is open (value 1) then
the Testmodul data comes through
Switch1
inst7
inst13
count_6bit
q[5..0]
inst
40MWord / 64 = 625kWord = 1.25Mbytes/s
testcount
q[15..0]
inst6
nRead_mux
clk_fifo_in
ClkTlkA
inst8
sel_data
BUSMUX
DataA[15..0]
data_CRC[15..0]
dataa[]
datab[]
result[]
data[15..0]
inst15
Parameter | Value
WIDTH | 16
StatusA[1]
inst5
wrreq
USB Module Outputs
Output Clk_out an QuickUSB IFCLK
Fifo_almost_full
clk_40Mhz
almost_full
used_width[13..0]
user_threshold
inst9
When sel_data='1' threshlod = 1000 else 10000
lpm_fifo_dc16384
data[15..0]
wrclk
wrusedw[13..0]
rdreq
rdclk
aclr
16 bits x 16384 words
inst18
REN
Send8BitsData
clk_Div2
Output[15..0]
ren
inst24
USB Module Inputs
REN
USB_DATA[15..0]

B. QuickUSB

Product Overview

The Universal Serial Bus is a popular computer interface that has become the de-facto standard interface for PC peripherals. Now, Hi-Speed USB 2.0 is the new standard in PC peripheral connectivity. With a speed of 480Mbps, USB 2.0 is up to 40 times faster than the 12Mbps USB 1.1 most computers use today. It uses the same type cabling and is backward compatible with USB 1.1.

Implementing a USB peripheral typically requires in-depth knowledge of the USB protocol, a considerable firmware and software development effort and rigorous compliance testing. But now there's an alternative.

The QuickUSB™ QUSB2 Plug-In Module makes adding Hi-Speed USB 2.0 to new or existing products quick by integrating all the hardware, firmware and software needed to implement a general-purpose USB endpoint as an easy-to-use plug-in module. The QuickUSB™ Plug-In Module also includes the QuickUSB™ Library.

The QuickUSB plug-in module contains hardware parallel and serial ports that are connected to circuitry in the peripheral. The QuickUSB library provides user-callable software functions that transfer data to and from the hardware ports over the USB. So the designer gets multiple ports of flexible, high-speed USB connectivity and no knowledge of USB is required.

Schematic Symbol

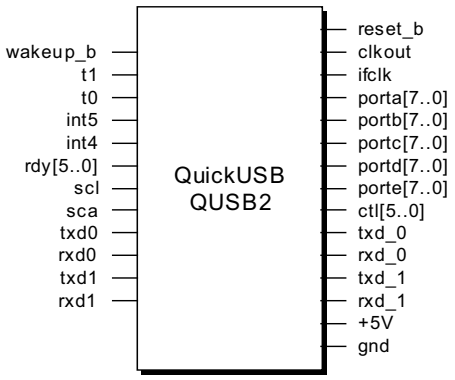


Figure 1 – QUSB2 Symbol

Functional Description

The QuickUSB Plug-In Module

The QuickUSB QUSB2 Plug-In Module is a 2" x 1 ½" circuit board that implements a bus-powered Hi-speed USB 2.0 endpoint terminating in a single 80-pin target interface connector.

The target interface consists of:

- One 8 or 16-bit high-speed parallel port
- Up to three general-purpose 8-bit parallel I/O ports
- Two RS-232 ports
- One I^2C port
- One soft SPI port or FPGA configuration port

High-Speed Parallel

The high-speed parallel port is configurable as an 8 or 16 bit synchronous parallel port. It delivers a sustained data rate of up to 12 MB/s and a burst rate of up to 48MB/s for packets up to 512 bytes long. The high-speed interface consists of the data port FD[15:0], control lines CMD_DATA, REN, WEN and GPIFADR [8:0]. The port can be used as a multiplexed command/data bus by decoding CMD_DATA (CMD = 0, DATA = 1 in the target logic. Reads are indicated by REN = 1 and writes are indicated by WEN = 1. If the address bus is configured to be active, concurrent with reads or writes the GPIFADR bus contains the address of each data element read from or written to FD [15:0].

General Purpose Parallel I/O

General purpose I/O pins must be configured to indicate whether they are being used as input or output pins. This is accomplished using library calls documented in the QuickUSB User's Guide.

The parallel ports have multiple functions and may not be available if alternate functions are enabled. The general-purpose I/O ports are ports A, C & E. Ports B & D are reserved for the High-Speed Parallel port. The Port E alternate function is FPGA configuration and the soft SPI port. If these alternate functions are used, port E is reserved. Otherwise, port E can be used as general-purpose I/O.

RS-232

The module has two RS-232 ports with a configurable baud rate. Both ports use the

same baud rate. These interrupt-driven ports internally buffer data as it arrives and when queried return the contents of the internal buffer. The interrupt buffer depth is 32 characters per port.

I²C

An I2C compatible port is included on the QuickUSB module. The port is a bus master only. Address 1 is reserved for on-board functions. The QuickUSB library provides functions to write and read blocks of data to and from I2C peripherals.

SPI

The module supports SPI peripherals through a 'soft' SPI port, which uses pins on port E. The pins MOSI, SCK, MISO and nSS are shared with the FPGA configuration function and will not interfere with each other *if* the SPI peripherals only drive the MISO when nSS is asserted (nSS=0).

FPGA Configuration

The QuickUSB Plug-In Module can program SRAM-based Altera programmable logic devices using five pins of port E. When designing your peripheral to use this feature, consult the 'PS Configuration with a Microprocessor' section of Altera Application Note 116, 'Configuring SRAM-Based LUT Devices'. This document specifies the circuitry needed to configure an Altera device with a microcontroller. The QuickUSB module provides the DCLK, DATA0, nCE, nCONFIG, nSTATUS and CONF_DONE signals required to configure Altera devices in passive-serial mode. If more than one Altera device must be configured over the interface, the devices should be 'daisy-chained' and the programming files combined into a single 'RBF' file. Consult AN116 for details on this configuration or contact Bitwise Systems.

The QuickUSB Library

The QuickUSB™ Library is included with the QUSB2 and provides DLL and C library interfaces to the QuickUSB Plug-in Module. The QuickUSB Library hides the complexity of USB 2.0 behind a port-based programmer's interface. A complete description of each library function is provided in the QuickUSB™ User's Guide

Contact

For pricing and other information contact:

Bitwise Systems
697 Via Miguel
Santa Barbara, CA 93111
Phone (805) 683-6469
Fax (805) 683-6469
www.bitwisesys.com
sales@bitwisesys.com

Pin Descriptions

Pin	Name	Dir	Description	Pin	Name	Dir	Description
1	GND	N/A	Ground	2	+5V	N/A	Unregulated +5V from the USB bus (300mA total)
3	PA0	I/O	Port A, Bit 0	4	RESET_B	OD	FX2 reset, Active low.
5	PA1	I/O	Port A, Bit 1	6	CLKOUT	Output	48MHz CPU clock
7	PA2	I/O	Port A, Bit 2	8	IFCLK	Output	48MHz GPIO clock
9	PA3	I/O	Port A, Bit 3	10	INT4	Input	8051 INT4 IRQ. Active high, edge sensitive
11	PA4	I/O	Port A, Bit 4	12	RXD_0	Input	Serial Port 0 RS-232 RxD
13	PA5	I/O	Port A, Bit 5	14	TXD_0	Output	Serial Port 0 RS-232 TxD
15	PA6	I/O	Port A, Bit 6	16	TXD_1	Output	Serial Port 1 RS-232 TxD
17	PA7	I/O	Port A, Bit 7	18	RXD_1	Input	Serial Port 1 RS-232 RxD
19	GND	N/A	Ground	20	+5V	N/A	Unregulated +5V from the USB bus (300mA total)
21	PB0	I/O	Port B, Bit 0 / FD0	22	CTL0	Output	GPIF ctl out 0 / CMD_DATA
23	PB1	I/O	Port B, Bit 1 / FD1	24	CTL1	Output	GPIF ctl out 1 / REN
25	PB2	I/O	Port B, Bit 2 / FD2	26	CTL2	Output	GPIF ctl out 2 / WEN
27	PB3	I/O	Port B, Bit 3 / FD3	28	CTL3	Output	GPIF ctl out 3
29	PB4	I/O	Port B, Bit 4 / FD4	30	CTL4	Output	GPIF ctl out 4
31	PB5	I/O	Port B, Bit 5 / FD5	32	CTL5	Output	GPIF ctl out 5
33	PB6	I/O	Port B, Bit 6 / FD6	34	RXD0	Input	Serial Port 0 TTL RxD (Do not use if U1 is populated)
35	PB7	I/O	Port B, Bit 7 / FD7	36	TXD0	Output	Serial Port 0 TTL RxD (Do not use if U1 is populated)
37	GND	N/A	Ground	38	+5V	N/A	Unregulated +5V from the USB bus (300mA total)
39	PC0	I/O	Port C, Bit 0 / GPIFADR0	40	RDY0	Input	GPIF input signal 0
41	PC1	I/O	Port C, Bit 1 / GPIFADR1	42	RDY1	Input	GPIF input signal 1
43	PC2	I/O	Port C, Bit 2 / GPIFADR2	44	RDY2	Input	GPIF input signal 2
45	PC3	I/O	Port C, Bit 3 / GPIFADR3	46	RDY3	Input	GPIF input signal 3
47	PC4	I/O	Port C, Bit 4 / GPIFADR4	48	RDY4	Input	GPIF input signal 4
49	PC5	I/O	Port C, Bit 5 / GPIFADR5	50	RDY5	Input	GPIF input signal 5
51	PC6	I/O	Port C, Bit 6 / GPIFADR6	52	RXD1	Input	Serial Port 1 TTL RxD (Do not use if U1 is populated)
53	PC7	I/O	Port C, Bit 7 / GPIFADR7	54	TXD1	Output	Serial Port 1 TTL RxD (Do not use if U1 is populated)
55	GND	N/A	Ground	56	+5V	N/A	Unregulated +5V from the USB bus (300mA total)
57	PD0	I/O	Port D, Bit 0 / FD8	58	PE0	I/O	Port E, Bit 0 / DATA0 / MOSI
59	PD1	I/O	Port D, Bit 1 / FD9	60	PE1	I/O	Port E, Bit 1 / DCLK / SCK
61	PD2	I/O	Port D, Bit 2 / FD10	62	PE2	I/O	Port E, Bit 2 / nCE
63	PD3	I/O	Port D, Bit 3 / FD11	64	PE3	I/O	Port E, Bit 3 / nCONFIG
65	PD4	I/O	Port D, Bit 4 / FD12	66	PE4	I/O	Port E, Bit 4 / nSTATUS
67	PD5	I/O	Port D, Bit 5 / FD13	68	PE5	I/O	Port E, Bit 5 / CONF_DONE / MISO (see note)
69	PD6	I/O	Port D, Bit 6 / FD14	70	PE6	I/O	Port E, Bit 6 / nSS
71	PD7	I/O	Port D, Bit 7 / FD15	72	PE7	I/O	Port E, Bit 7 / GPIFADR8
73	SCL	OD	Clock for I2C interface	74	WAKEUP_B	Input	USB Wakeup. Active low.
75	SDA	OD	Data for I2C interface	76	INT5_B	Input	INT5 Interrupt Request. Active low, edge sensitive
77	T0	Input	Input for Timer0	78	T1	Input	Input for Timer1
79	GND	N/A	Ground	80	+5V	N/A	+5V

Table 1 – QUSB2 Pin Description

QuickUSB Socket Layout

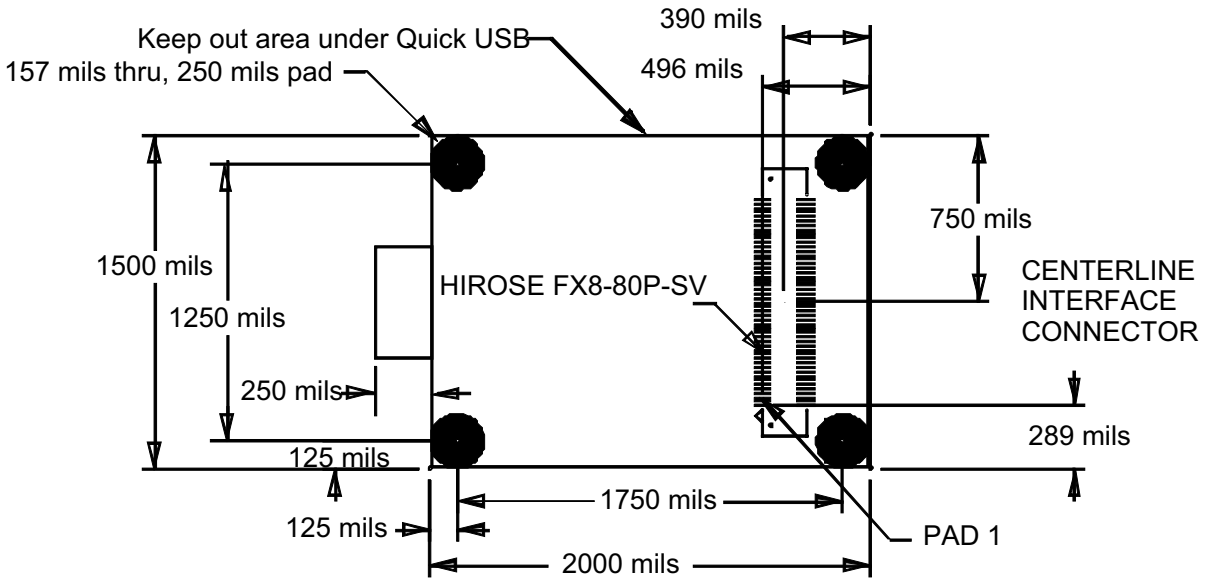


Figure 2- QuickUSB Socket Layout

Cable between Stratix-SantaCruz and QuickUSB

	QUD	FPGA
signal	J2 (20x2)	J21 (20x2)
a0	3	3
a1	5	5
a2	7	7
a3	9	9
a4	11	11
a5	13	13
a6	15	15
a7	17	17
b0	19	21
cmd_data	20	28
b1	21	23
ren	22	32
b2	23	25
wen	24	36
b3	25	27
b4	27	29
b5	29	31
b6	31	33
b7	33	35
gnd	40	40

	QUD	FPGA
signal	J2 (20x2)	J22 (10x2)
CLKIN	8	13

	QUD	FPGA
signal	J3 (20x2)	J24(20x2)
c0	3	3
c1	5	5
c2	7	7
c3	9	9
c4	11	11
c5	13	15
c6	15	17
c7	17	19
d0	19	21
e0	20	22
d1	21	23
e1	22	24
d2	23	25
e2	24	26
d3	25	29
e3	26	30
d4	27	31
e4	28	32
d5	29	33
e5	30	34
d6	31	35
e6	32	36
d7	33	37
e7	34	38
gnd	40	40

C. LabVIEW

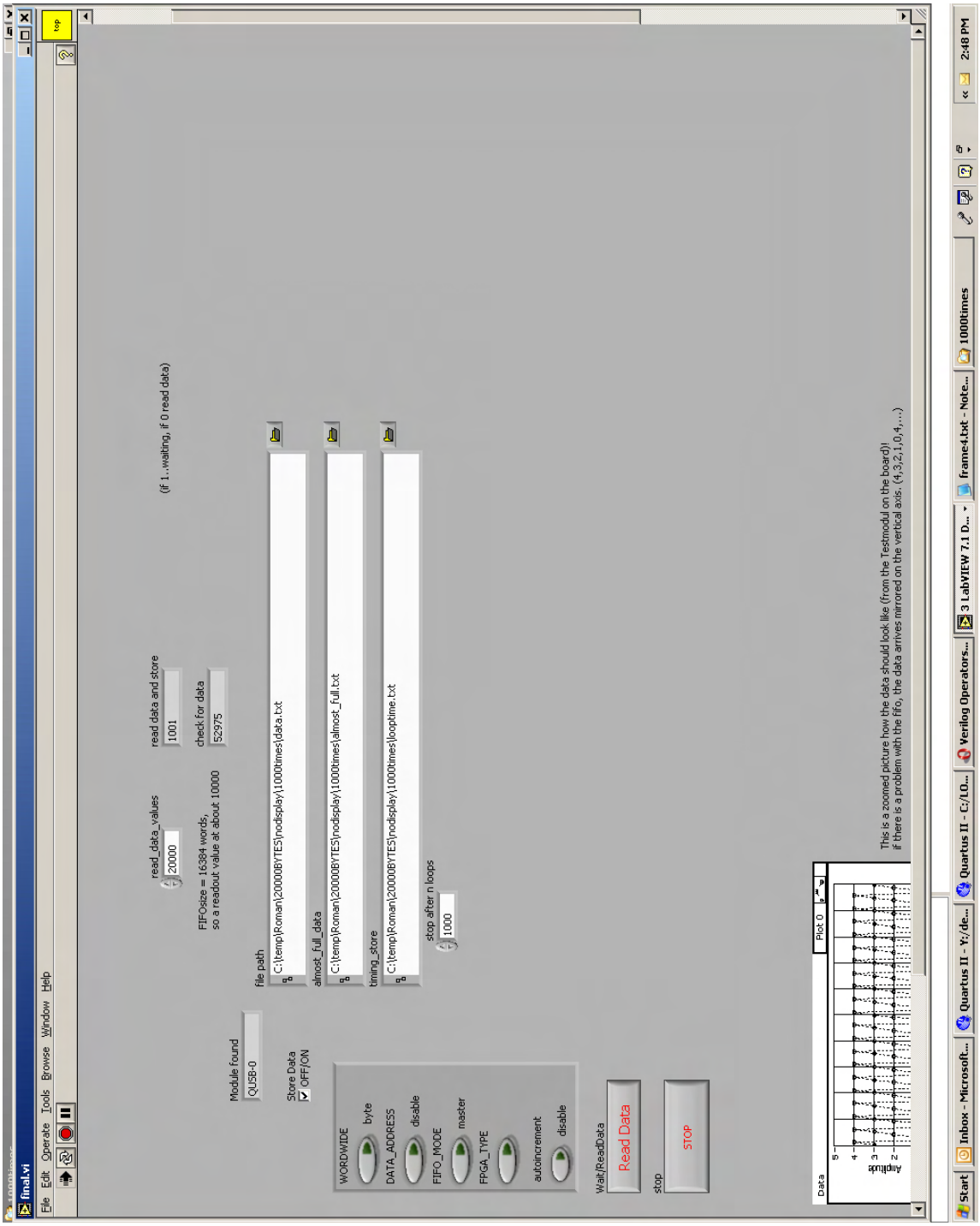


Figure C.1.: Labview read-out program without display (developer version)

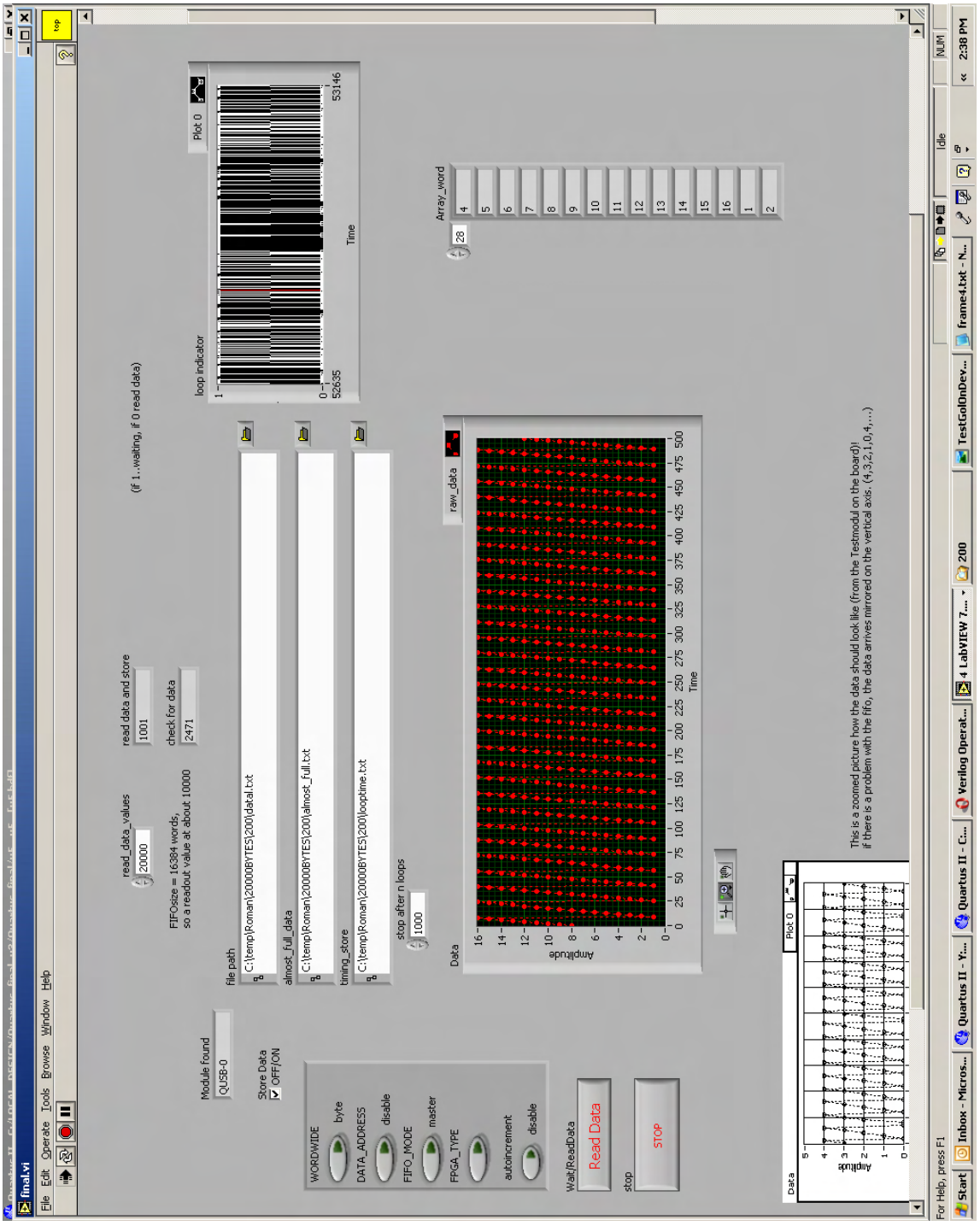


Figure C.2.: Labview read-out program with display during an acquisition.(developer version)

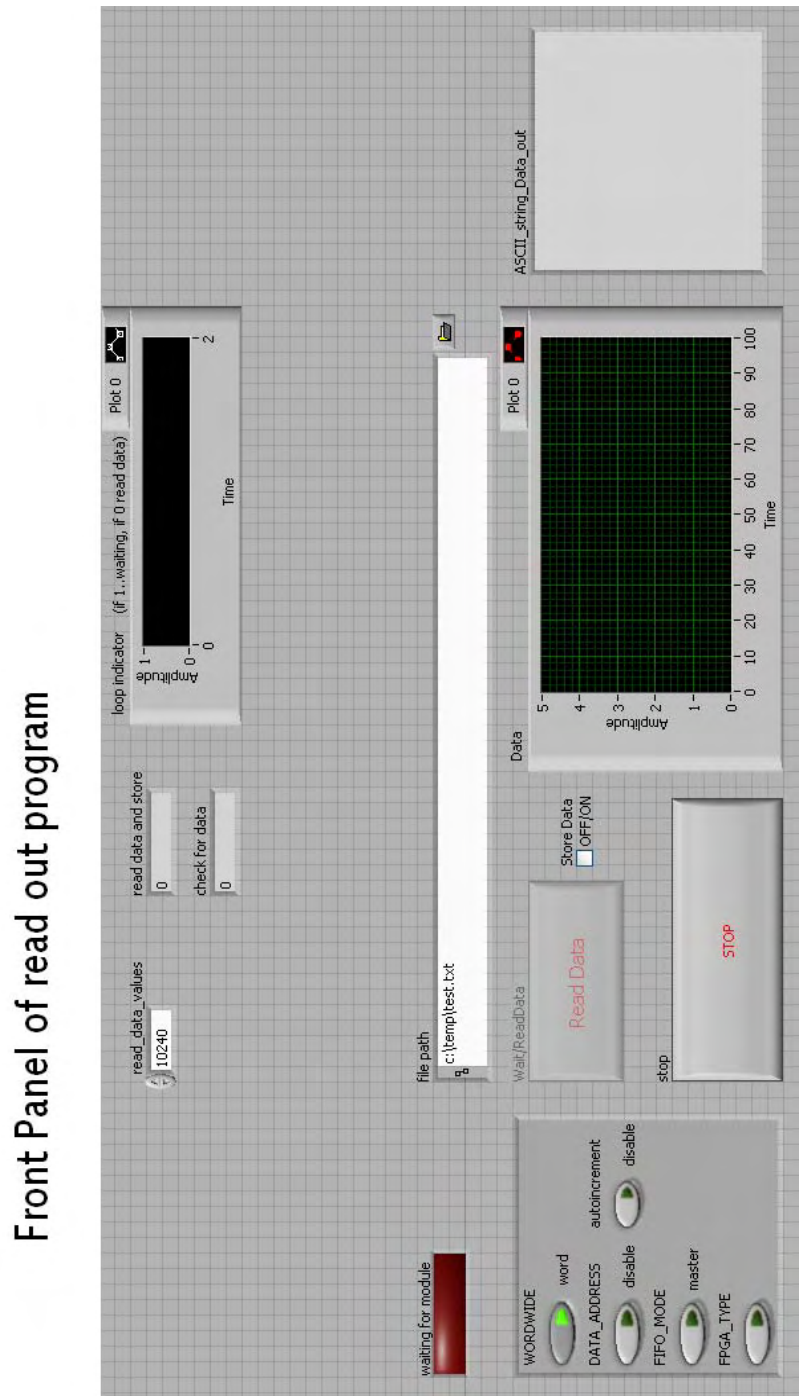


Figure C.3.: Labview read-out program (final version) : front panel

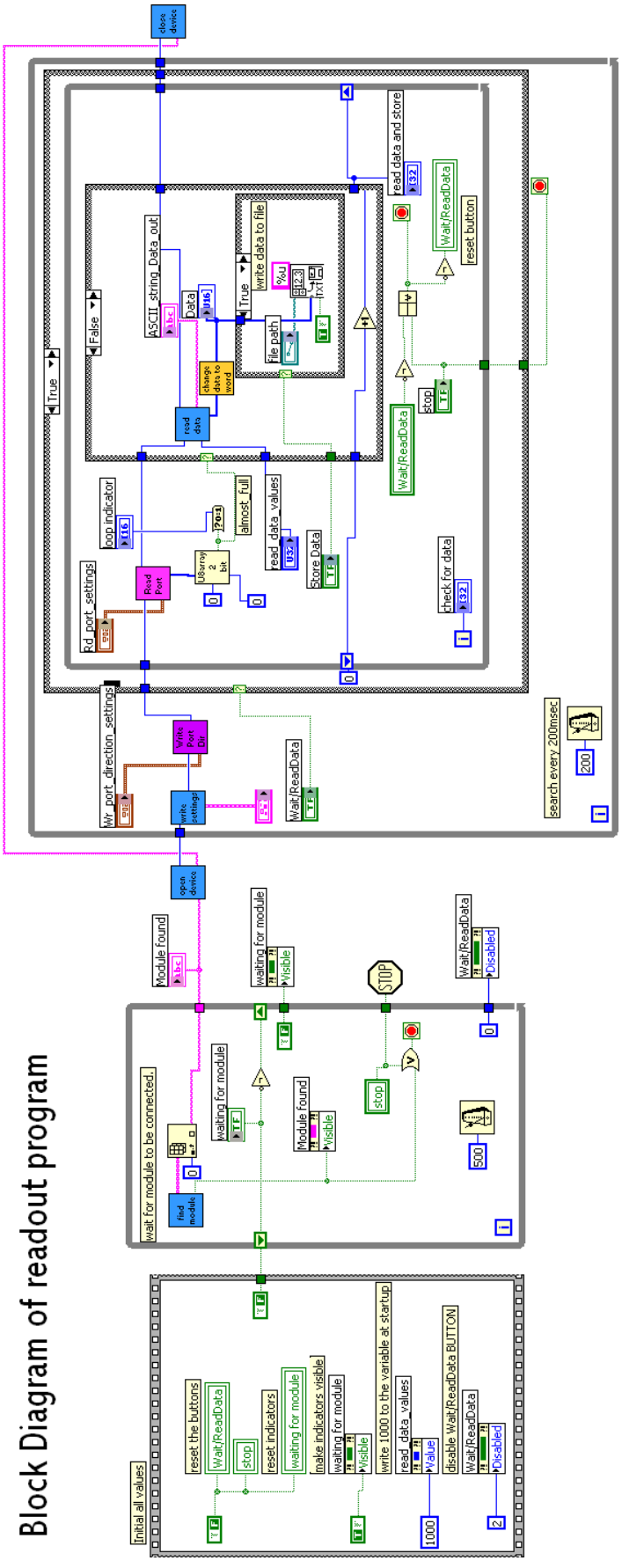
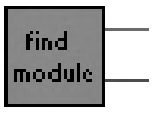


Figure C.4.: Labview read-out program (final version) : block diagram

C.1. Functions in Labview



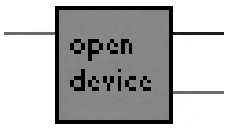
find_module.vi

This function finds all QuickUSB-modules which are attached to the computer. When this function is running, for example in a lo

op and a QUD[1] is connected, it may take some seconds until it is recognised by the function. The reason for this is that the computer needs some time to identify every new USB device. The function returns an array of strings which contains the names of all QUD found on the system and a boolean variable. This variable indicates if a device is found. It can be used to terminate a loop calling the *find_module.vi*.

input : no input

output :
- modules : Stringarray of all modules attached to the computer. The size of the array is equal to the amount of modules.
- module_found? : Boolean variable. TRUE when a module is found. Can be used to terminate a loop.

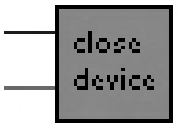


open_device.vi

Before a USB-Module can be used, it must be opened. This subVI requires the name of the QuickUSB-Module and returns the ID[2] of the new device. This ID is needed by all other functions to identify the device.

input :
- devName : Contains the name of the module which should be opened.

output :
- hDevice : Is a ID number which is used by other functions to identify the module.
- devName : The name of the opened module is passed through the subVI and can be used in 'close_device.vi'.



close_device.vi

This SubVI is used to close the USB-Module when it is not used anymore. If the chosen module cannot be closed, a Message occurs : *"The device '**devName**' cannot be closed."*.

input :
- hDevice : device ID-number.

[1] QuickUSB-Device
[2] hDevice

– devName : the name of the opened module.

output : no output, but a warning occurs if the device cannot be closed.

write_settings.vi

Due to the fact that only one setting per loop can be written to the QUD the VI runs a loop of 4 iterations to write each setting to the device. Inside the VI the subVI *cluster_read_settings.vi* is used to prepare the settings for the writing-process.

All Settings of the QUD are saved in a volatile memory. They are not stored, if the module is not powered! It is necessary to rewrite the settings to the QUD when it is attached.

input :
– hDevice : device ID-number.
– Settings : clustervariable containing the settings : Word/Byte, DataAddress, Auto-Increment, FIFO-Mode and FPGA-Type.

output :
– hDevice : device ID-number.

read_settings.vi

Using this vi the actual settings can be read from the QUD. Inside the subVI *settings_read_cluster.vi* is used to change the appearance of the settings. *settings_cluster_read.vi* is used to make the settings more readable.

input :
– hDevice : device ID-number.

output :
– hDevice : device ID-number.
– Settings : clustervariable containing the settings : Word/Byte, DataAddress, Auto-Increment, FIFO-Mode and FPGA-Type. Refer to chapter 4.1

write_settings_read.vi

This subVI is used in *read_settings.vi* and converts the return value to an user-readable indicator.

input :
– settings : Arrayform of settings-variable.

output :
– Settings_cluster : user-friendly format of the Settings-variable.

write_settings_write.vi

To simplify the User Interface this subVI converts a cluster variable to a format which is used to write the settings to the QUD.

input : – Settings_cluster : User-friendly format of the Settings-variable.

output : – address : Address-array of the Settings.

– setting : Array of Settings.

Read_port_Dir.vi

Each Bit of the General Purpose IO-ports (port_A, port_C and port_E) can be configured either as input or as output. port_B and port_D are reserved for the High-Speed Parallel port. This subVI reads the current direction of the bits of the indicated port. (0 = input, 1 = output)

input : – hDevice : device ID-number.

– port : number of port. $0 \rightarrow A, 2 \rightarrow C, 4 \rightarrow E$

output : – hDevice : device ID-number.

– direction : direction of port. (0 = input, 1 = output)

Write_port_Dir.vi

With this subVI all bits of the General Purpose IO-ports can be configured as either as input or as output. The direction of a bit or a port is controlled by the variable *direction* in the input-parameter *port_settings*. The binary value 1 indicates an *output* and the value 0 an *input*.

e.g.: To configure the whole port as Output the Cluster-variable *port_settings* must be set to 0xFF (0b11111111). The value 0xF0 (0b11110000) configures the upper 4 bit of the port as *output* and the lower 4 bit as *input*.

input : – hDevice : device ID-number.

– port_settings : Cluster-variable containing the port address and the direction.

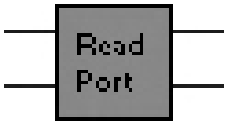
output : – hDevice : device ID-number.

U8Array_2_bit.vi

This subVI returns a single boolean variable (*true* or *false*) from an U8Array. It can be used in combination with the VI *read_port.vi* to get the boolean value of one pin on the QUD.

input :
- U8_Array : binary data from the General Purpose ports.
- Index_of_array : usually 0 (for the first array), but it is also possible to read more than one value from the ports. (maximum length is 64 bytes)
- Index_of_byte : This values determines which bit of the byte is requested. (0 for LSB[3], 7 for MSB[4])

output :
- bit : boolean-variable



Read_port.vi

To read one of the General Purpose ports they must be configured as *input* first. This VI returns the value of the selected port from the QUD. The port and the amount of bytes to read from it are selected by the inputsignal *RD_port_settings*.

input :
- hDevice : device ID-number.
- Rd_port_settings : cluster-variable including the port_address ($0 \rightarrow A, 2 \rightarrow C, 4 \rightarrow E$) and the lenght of bytes to read. The maximum amount of bytes which can be transmitted at once is 64. [8].

output :
- hDevice : device ID-number.
- Read_Port_Data : Array of unsigned 8bit integer containing the value of the port.

Write_port.vi

This VI is used to write values to the General Purpose IO ports of the QUD. Before it can be used, the used IO pins must be configured as Output using the VI *Write_port_Dir.vi*.

input :
- hDevice : device ID-number.
- address_data : cluster-variable containing the port_address ($0 \rightarrow A, 2 \rightarrow C, 4 \rightarrow E$) and an array of unsigned 8bit integer containing the data. The maximum size of the array is 64. [8].

output :
- hDevice : device ID-number.

Read_data.vi

To receive data from the HSPP it must be configured either as slave or as master. (please refer to chapter 4.1 for more details to the HSPP settings.) Then the data can be read using this VI. When using the whole 16 bits from the HSPP, the data arrives in two stages. Because the datatype of the **data**-variable is an 8bit integer, the first byte arriving is the lower byte of the 16bit wide bus. The high byte arrives as second value.

[3]Least Significant Bit
[4]Most Significant Bit

input : – hDevice : device ID-number.
–

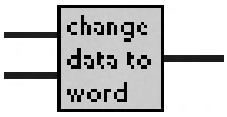
output : – hDevice : device ID-number.
– data : array of data from the HSPP.

U8Array_2_ASCII.vi

If the data consists of ASCII code (readable letters and numbers) this subVI can be used to change the data to from binary to ASCII character. This subVI is already used inside the VI *read_data.vi*.

input : – Array : binary data.

output : – ASCII-String : string-variable



modify_dataarray.vi

Because the data arrives in two bytes instead of one full word (16 bit) it is necessary to concatenate these two bytes to one word. This subVI generates out of two 8 bit arrays (LSByte_array, MSByte_array) one 16 bit array (array_word).

input : – LSByte_array : 8 bit variable containing the lower 8 bit of the 16 bit data word.
– MSByte_array : 8 bit variable containing the higher 8 bit of the 16 bit data word.

output : – Array_word : 16 bit variable excisting of the LSByte_array and MSByte_array.)