Coding of Laser Pulse Transmission Time for SLR-Stations

Master Thesis

Wilhelm Alexander Steinegger

Institute for Broadband Communication
Graz University of Technology

in cooperation
with Observatory Lustbühel (Austrian Academy of Sciences)

Supervisor/Reviewer: Ao. Univ.-Prof. Dr. Erich Leitgeb
Co-Supervisor: Dr. Georg Kirchner (Observatory Lustbühel)

Graz, March 2011

Abstract

Satellite Laser Ranging (SLR) Stations determine distances to satellites by measuring the time-of-flight of short (10 ps to 100 ps) laser pulses, reflected from corner cube reflectors (CCR) on satellites. The laser repetition rates vary from 10 Hz to 2 kHz. This present Master Thesis describes and designs a system to detect coded laser pulses.

The main component of this system is the 2 kHz laser of the SLR Station Graz / Austria. This laser normally has a constant firing time of 500 μs. To construct a system based on the pulse position modulation, it is necessary to variate or to delay the firing times of the 2 kHz laser. The laser firing times are controlled within a FPGA circuit. This FPGA is on an ISA PC Card, and allows to program these additional delays.

The inaccuracy or jitter of Laser Time Firing is in the range of ± 7 ns and has to be considered in programming the final detector software. One more important part of this present work is the selection of a suitable detector, based on multiple SPADs.

This project can be divided into three main parts. First of all the ISA PC Card with the FPGA circuit, which is responsible for the firing commands of the laser. The second part of this Master Thesis is the detector software, which is programmed in Microsoft Visual Basic 6.0. The third part is the optical receiver with an Altera-Apex CPLD, a FT245RL interface and an optical detector called Multi-Pixel-Photon Counting module (MPPC). This MPPC module is manufactured by Hamamatsu and consists of 400 avalanche photodiodes (APDs) operating in Geiger mode.

With such a setup at any SLR station and a suitable detector plus simple time tagging electronics at Low Earth Orbiting (LEO; < 1000 km) satellites, it is possible for any kHz SLR station to transmit data to satellites with a rate of up to 2 kB/s - even during standard SLR tracking.

Kurzfassung

Satellite Laser Ranging (SLR) ist eine hochpräzise Methode der Satellitengeodäsie zur Bestimmung der Distanz zwischen einer Bodenstation und Satelliten mittels Laufzeitmessung kurzer Laserimpulse (10 ps bis 100 ps). Diese Satelliten sind mit *Corner Cube* Reflektoren ausgerüstet und reflektieren die von der SLR-Station ausgesendeten Laserimpulse. Die Laserpuls-Intervalle variieren hierbei von 10 Hz bis 2 kHz. In dieser Masterarbeit wird ein System entworfen, das es ermöglicht, codierte Laserimpulse zu übertragen und zu empfangen. Die Arbeit wurde dabei auf der 2 kHz Laserstation des Observatoriums Graz / Lustbühel durchgeführt.
Dieser Laser arbeitet mit einem konstanten Zeitintervall (*firing interval*) von 500 μs. Um nun ein System zu entwerfen, das auf der Puls-Positions Modulation basiert, ist es notwendig, das konstante Zeitintervall dieses 2 kHz Lasers zu variieren. Diese Zeitintervalle werden mittels eines FPGA gesteuert, das sich auf einer ISA PC-Steckkarte befindet. Diese ISA-Karte erlaubt es, diese Verzögerungszeiten zu programmieren.
Die zeitliche Abweichung oder auch Jitter der Laserimpulse bewegt sich im Bereich von $\pm$7 ns. Diese Ungenauigkeit muss bei der Programmierung der Detektorsoftware berücksichtigt werden.
Die vorliegende Masterarbeit kann in drei Hauptteile gegliedert werden. Der erste Teil ist die ISA-Steckkarte, die zur Steuerung der Laser-Feuerungszeiten dient. Der zweite Teil dieser Arbeit ist die Detektorsoftware, die in Microsoft Visual Basic 6.0 programmiert wurde. Der dritte Teil ist der optische Empfänger, der aus einem Altera-Apex FPGA, einer Schnittstelle und einem optischen Detektor besteht. Beim optischen Detektor handelt es sich um ein Multi-Pixel-Photon-Counting Modul kurz MPPC. Dieses Modul wird von der Firma Hamamatsu hergestellt und besteht aus 400 Lawinenphotodioden, die im Geiger-Modus betrieben werden. Alle diese Komponenten und die damit verbundenen Probleme bei der Realisierung werden in dieser vorliegenden Masterarbeit diskutiert, analysiert und realisiert. Diese Technik kann bei jeder kHz-SLR-Station eingesetzt werden und erlaubt, Daten mit einer Rate von bis zu 2 kB / s auf Satelliten zu übertragen.

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ……. ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

……. ……..
date (signature)

Acknowledgements

This Master Thesis was realized at the Institute for Broadband Communication at the Technical University Graz in cooperation with Observatory Lustbühel. I especially want to thank my advisor Erich Leitgeb, who encouraged me to write this present Master Thesis at the Observatorium Lustbühel. I mainly thank Georg Kirchner for his correspondance and for answering many questions. Without his help and without his technical and vocational skills I would not have been able to complete this project. Also special thanks to Franz Koidl for programming the hardware and for giving me tips for implementing the detection software. Last but not least I want to mention my parents, my friends and all people who supported and believed in me and in my studies. This was very crucial for this work and for the final degree.

Graz, Austria, March 2011 Wilhelm Alexander Steinegger

Contents

List of Figures

List of Tables

1 Introduction

The SLR Station Graz / Austria uses a 2 kHz Laser for measuring distances to satellites - equipped with corner cube reflectors (CCR) - very accurately ($\pm$2 to 3 mm). The reflected photons are detected and time tagged at the SLR station.
Normally, the laser pulses are fired in 500 μs intervals, but using pulse position modulation (PPM) scheme, the firing time varies from the nominal firing time. The PPM offers a efficient possibility to transmit data information with an adequate offset. The maximum possible data transmission rate is 1 byte / shot - for a 2 kHz SLR station like Graz, this allows 2 kbytes / s. Realizing a system, which consists of hard- and software needs a lot of technical considerations and improvements during the development phase e.g. the transfer speed between the hard- and software is always critical and has to be adapted. This Master Thesis demonstrates a system, which is able to transmit data based on the 2 kHz laser. The first part in this work explains the pulse position modulation applied on the constant laser firing times and extracting information from artificial epochs to demonstrate that it is possible to code and decode messages with the pulse position modulation.
The second part explains how it is possible to send and receive data by different interfaces. Two interfaces are demonstrated and compared, if they are suitable for this project or not. The third part describes the development of the final detector software. This software has first been tested with pseudo epochs generated by the 2 kHz and with no Multi-Pixel-Photon-Counter.
The fourth part discusses the used MPPC, the results of the final tests and measurements. At the end of this Master Thesis, some possible improvements e.g. to avoid transmissions errors are discussed.

1.1 Motivation

The main goal of this Master Thesis is to create a laser system, based on the 2 kHz laser used on the SLR station in Graz / Austria, which is able to transmit data information. Normally, the laser pulses are fired in 500 μs intervals. For this project, these laser pulses have to be varied with an adequate offset.
The pulse position modulation is a simple modulation technique to provide delays in constant laser firing times. Each delay describes a single ASCII character. The maximum possible data upload rate is 1 byte / shot; for a 2 kHz station like Graz, this allows 2 kbytes / s, which is already one order more than the microwave upload channel of satellites like CHAMP (119 bytes / s) [11]. At the SLR station in Graz, software and hardware were implemented to demonstrate such data transmissions via a fixed retro-reflector target at a distance of 4288 m;

using a separate detection unit at the station and suitable attenuation of the laser energy, the data uploading procedure was successfully simulated.

1.2 SLR-Technology

Satellite Laser Ranging (SLR) is a method to measure the distances between satellites with very short laser pulses. This short laser pulse is generated by a SLR Station, shot to the satellite and finally it is refected back to the station, where it is registered. A high-precision Event-Timer, which uses the time and the standard frequency, measures the flight time of the laser pulse with an accuracy of about 3 ps. The standard frequency is acquired with a dedicated GPS receiver. After measuring the flight time of the laser pulse and taking the known speed of light it is possible to determine the distance to the satellite with an accuracy of a few millimeters. These distance measurements are done by about 40 SLR Stations around the world. Precise orbits are the result of these measurements. The analysis of these orbits are crucial for determining the complex rotation parameters, the gravitation field and the continental drift of earth [9]. The computer generated image 1.1 shows the basic principle of SLR. On the top left corner of the image, a satellite, equipped with retro-reflectors can be seen.

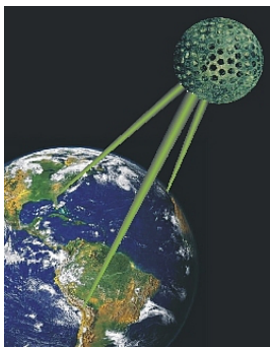

Figure 1.1: SLR [9]

The Institute for Satellite Geodesy practises Satellite Laser Ranging (SLR) at the station Graz / Lustbühel since 1982. The distances between ground station and satellites equipped with Corner-Cube reflectors can be measured with an accuracy of ± 2 - 3 mm. These measurements can be done day and night at seven days a week.
Since 2003 the station uses a Nd:Vanadate kHz laser system “Made in Austria”, which produces very short pulses (10 ps) - this equals to a lengh of just 3 mm, with a wavelength of 532 nm and a relatively low energy of 400 μJ / shot. On the way to the satellite and back, almost all transmitted photons get lost and just a few photons - in most cases just a single photon is catched by a Single-Photon Avalanche Detector (SPAD). The modular

realtime-observation-software enables a high automation for measurements. One advantage of a high automation is that untrained observers get a very high efficiency, high return quotas and highest data density during a short period of time [9]. The SLR Station on Graz / Lustbühel uses a Satellite Camera of type Zeiss BMK75, which was rebuilt to CCD-technology in cooperation with the Technical University Vienna. Since then, the Satellite Camera is also used for observations of asteroids and tested for space scrap detection. In the following, the equipment of the Satellite Camera is listed. Figure 1.2 is an original photo of the Zeiss BMK75 used on the Observatorium Graz / Lustbühel.

Figure 1.2: Satellite Camera Zeiss BMK75 used in Graz / Lustbühel [20]

The measurement unit for SLR, which is used is the third one, developed in Graz since the 1970s and it was the first measurement unit worldwide, which reached an accuracy of less than 1 cm. There is also a series of GPS- and GLONASS receivers for precise distance measurements (pseudo rangingand dGPS). Both measurement methods (LASER and GPS) are essential for the worldwide monitoring of the earth's rotation and pole movements. Further receivers and antennas for special cases e.g. micro-waves, doppler etc.
This Master Thesis enhances the SLR equipment of the Observatory Lustbühel to send and receive coded laser pulses. This procedure does not affect the normal SLR process. The first step is to modulate the epoch times of the laser pulses. This can be done by reprogramming the ISA Card, which controls the firing intervals. To generate time values from the received modulated laser pulses, a CPLD with an internal counter has to be programmed and a suitable photon detector has to be chosen. After some testing phases, some possible improvements and future work are discussed at the end of this work.

2 Modulation Schemes

In optical telecommunication systems several modulation schemes are used. Modulation means to mix the signal with the carrier frequency in relation to the modulating signal [12]. Demodulation is the physical reversal of modulation and extracts the original signal [12]. The transmission of a communication signal can either be done by electrical lines or by wave propagation [12].
In the following, some different modulation schemes are discussed. In telecommunications, analog and digital modulation schemes are used. In analog modulation, the modulation is applied continuously in response to the analog information signal. Digital modulation is the mapping of data bits to signal waveforms that can be transmitted over an (analog) channel. In this Master Thesis the pulse position modulation is used to transmit ASCII characters. The pulse position modulation can either be an analog or digital modulation scheme. It depends on the signal and / or the modulator used to obtain the modulation. In this Master Thesis, the digital pulse position modulation is used, because for demodulation a CPLD counter with a resolution of 5 ns is used. As a result, the output has a discrete timing interval. With the help of this discrete timing interval, the correct ASCII character can be assigned. In the following sections, different modulation schemes will be discussed shortly.

2.1 Analogue Modulation Schemes

In analog modulation schemes, the information travels in a signal that is permitted to take any value. Here, the information in the signal is used to continuously change some property of the carrier without the use of processes such as quantisation or digitization at any stage [16]. The following subsections will give a short overview of the most important analog modulation schemes.

2.1.1 Amplitude Modulation

In this form of modulation, the signal is used to modulate the amplitude of the carrier. The ampitude of the carrier becomes a function of the ampitude of the signal [16]. The process of generating an AM signal is simole. If a constant-amplitude carrier is multiplied by the signal, the amplitude of the carrier will have the shape of the signal. Allmost all modulators use variations of this principle. In frequency domain, this process affects a shift in the signal spectrum by frequency equal to that of the carrier [16].
AM is one of the oldest off all modulation schemes. It is popular in radio transmissions. The demodulator is very simple to design and makes AM receivers very cheap. But it has many

disadvantages, e.g. it is power-inefficient and it has a poor signal-to-noise ratio (SNR), which is a measure of the quality of the signal [16].

2.1.2 Angle Modulation

In this modulation scheme the angle of the carrier varies in a certain manner with the modulating signal. There are two modulation schemes, which are called angle modulation. These schemes will be discussed shortly.

Frequency Modulation: In FM, the instantaneous frequency of the carrier is varied in proportion to the modulating signal. In contrast to the AM , the carrier signal's amplitude is not changed. This improves the overall signal-to-noise ratio of the communications systems [16]. To transmit an FM signal, more analog bandwidth is necessary than for transmitting an AM modulated signal.

Phase Modulation: Phase modulation is similar to frequency modulation. Instead of the frequency of the carrier wave, the phase of the carrier changes [16].

2.1.3 Pulse Modulation Schemes

In pulse amplitude modulation, a pulse is generated with amplitude corresponding to that of the modulating waveform [16]. Similar to AM, it is very senitive to noise. PAM is an important first step in a modulation scheme known as pulse code modulation. The pulse amplitude may take any value - PAM is really an analog modulation scheme.

Pulse Position Modulation: PPM is used in this project and will be discussed in the next section. It is a scheme where pulses of equal amplitude are generated at a rate controlled by the modulating signal's amplitude.

Pulse Width Modulation: In PWM, pulses are generated at a regular rate. The duration of the pulse is controlled by the modulating signal's amplitude.

2.2 Digital Modulation Schemes

Nowadays, communication in digital form is even more common than the old analog form. This is the fact why digital modulation schemes became very important over the last years. The most important ones are explained shortly in the next few subsections.

2.2.1 Pulse Code Modulation

PCM is a digital scheme for transmitting analog data. The signals in PCM are binary - there are only two possible states, represented by logic 1 (high) and logic 0 (low). Using PCM, it is possible to digitize all forms of analog data.
Pulse code modulation is the most fundamental of all digital modulation schemes. It is not only digital but also discrete time. There is a certain restriction on the minimum rate at which samples have to be taken from the signal for this form of modulation. To prevent a condition known as “aliasing”, the sample rate must be at least twice that of the highest supported frequency [16].

2.2.2 Amplitude Shift Keying

ASK is one of the most simple digital modulation scheme. The two possible binary values, 1 and 0, are represented by two different amplitudes. This scheme only requires low bandwidth, but it is very susceptible to interference. Effects like multi-path propagation, noise or path loss heavily influence the amplitude. In a wireless environment, a constant amplitude cannot be guaranteed, so ASK is typically not used for wireless radio transmission [17].

2.2.3 Frequency Shift Keying

This modulation scheme is often used for wireless transmissions. The simplest form of FSK, also called binary FSK (BFSK), assigns one frequency f_1 to the binary 1 and another frequency f_2 to the binary 0 [17].

2.2.4 Phase Shift Keying

Phase shift keying (PSK) uses shifts in the phase of a signal to represent data. It involves representation of binary data states, 0 and 1, by the phase of a fixed frequency sinusoidal carrier wave, a difference of 180° being used to represent the respective values [16].

2.3 Selected Modulation Scheme

In this Master Thesis, the Pulse Position Modulation is used to generate coded laser firing pulses. In 2.1.3 the Pulse Position Modulation is already mentioned.

2.3.1 Pulse Position Modulation

Normally the laser shots are fired in constant 500 μs time intervals. The PPM varies this constant delay according to the transmitted ASCII character. The difference between the constant time interval of 500 μs and the alternative delay is called offset. The offset for every single ASCII character is unique and has to be determined after detecting a delayed respectively a coded laser pulse. The uncertainty of delayed firing times is about $\pm$7 ns. This means that the defined offset can vary, although the same character is transmitted.
The basic offset of the coded laser pulses is 40 ns. In the final version, the offset was chosen with 80 ns. This allows jitter or an uncertainty in the range of 75 ns.
Using the standard ASCII table, it is possible to code all ASCII characters - excluding the zero value. The decimal value of the desired ASCII value gets multiplied with the basic offset (40 ns) and the final result is the offset from the basic grid of 500 μs pulse intervals. The following diagram represents the pulse position modulation in its easiest way.

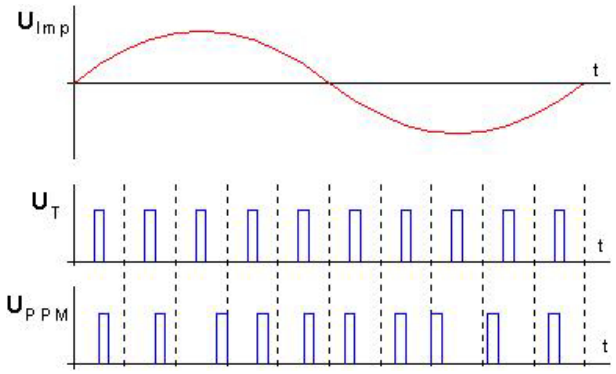

Figure 2.1: Pulse Position Modulation [5]

The sine curve U_{Imp} is an analogue function and describes one period t. U_T is an unmodulated rectangular signal with a fixed pulse width. The time between last and next pulse is always the same. That is why it is called an unmodulated rectangular signal. It is comparable with the reference signal in this project, which will be discussed in 2.3.2.
The most interesting rectangular pulse in this diagram is U_{PPM}, which is a modulated signal. This signal also has a constant pulse width, but the time period between last and next pulse in comparison to the unmodulated signal U_T is not equal. The time difference or the delay between the next pulse of the unmodulated signal U_T and the modulated signal U_{PPM} is the information. The diagram below is very general, because U_{PPM} is also leaded in phase to U_T. This does not happen, if ASCII characters are coded, because of the offset, the coded laser pulses are always delayed in relation to the unmodulated reference signal. Section 2.3.2 gives a short overview about the principle how ASCII characters can be coded by using this modulation scheme.

2.3.2 Examples for coded Laser Pulses

As above mentioned the transmitted laser pulses with about 500 μs intervals define a basic grid. Any ASCII character, which is transmitted is now coded as a variable offset from this grid. The offset is selected in steps of 40 ns, to avoid any troubles with the uncertainty in the range of $\pm$7 ns.
Every character with an ASCII decimal value of 1 to 255 can be transmitted. The offset from the basic grid of 500 μs pulse intervals is N$*$40 ns. The "N" stands for the decimal value of the ASCII character defined in the ASCII table. For example, if the ASCII character "A" is transmitted, the offset in relation to the basic grid amounts 2.6 μs, because the decimal ASCII value of "A" is 65 and "N" gets substituted for 65. Figure 2.2 shows that zero value laser pulses with the basic distance of 500 μs have to be transmitted before the coded laser pulses are send.

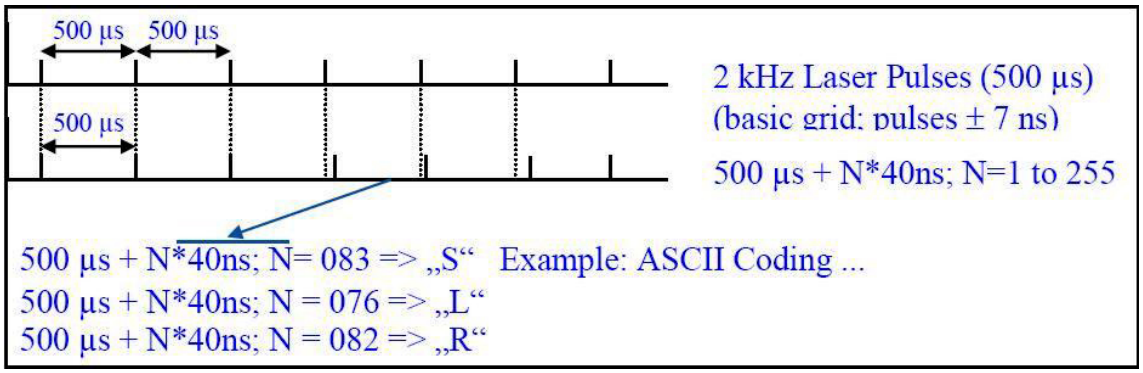

Figure 2.2: Pulse Position Modulation applied on constant laser pulses [11]

Furthermore figure 2.2 illustrates the coding of the word "SLR". The laser detection system needs to receive at least 100 equal values with a distance of 500 μs to determine a reference value. As already mentioned at the beginning, the epoch time values have an uncertainty in the range of $\pm$7 ns. In the final version of this project, the jitter is defined in the range of $\pm$75 ns, because problems occurred during the ASCII character recognition.
To determine an accurate reference value, the mean value of 100 equal values has to be calculated. Finally this mean value is the new reference value and important for the recognition of the ASCII symbols. In chapter 3.2 this method is explained in detail.

3 Software

To realize the detection software, a special software was programmed in Microsoft Visual Basic 6.0. This easy to learn programming language was chosen, because of the following reasons:

- Visual Basic 6.0 is an uncomplicated programming language for people, who have already made experiences in other programming languages e.g. Java, C++, Fortran etc.
- Visual Basic 6.0 can deal with the used interfaces like USB-PIO or UM245R . To be able to deal with the programming language, a special driver had to be installed and the provided functions of the DLL made it very easy to send and to receive data.
- In Visual Basic 6.0 it is possible to develop a Graphical User Interface in a very short periode of time. In Microsoft Visual C++ it would be also possible to realize a GUI, but it is more difficult to handle this programming language. There are a lot more instruction sets, which a programmer has to consider. Even though an extended instruction set opens more possibilities to solve problems, however it costs lots of precious time to get experiences in using these instructions e.g. pointers in C++ may be very powerful instruments and offer much more opportunities, if a programmer can deal with them correctly. Pointers allow to access to storage positions directly and make it possible to allocate storage space during runtime.
  The main disadvantage of pointers is that an incorrect usage of them could end in a disaster. The correct usage of pointers is not very easy and a programmer with rare skills is often overextended.
- Object oriented concepts are helpful for large software projects. They are not necessary in this case. This is yet another reason why a procedural programming language like Microsoft Visual Basic 6.0 was chosen.

Visual Basic 6.0 has also some disadvantages over C++ e.g. platform dependency. The programmer must use a Microsoft operating system, which is not for free like an Unix operating system. Programs written in Microsoft Visual Basic 6.0 are also not compatible with alternative operating systems. Another disadvantage to mention is that the installation of Microsoft Visual Basic 6.0 needs a lot more hard disk space compared to a simple GNU C compiler installation, because the programming environment in Visual Basic contains a lot more tools e.g. editor, different graphical interfaces etc.

3.1 First Steps

At first an ASCII file with generated artificial epoch times is generated. These epoch times can stand for different ASCII characters. The difference of two neighbored epoch times is always greater or equal to 500 μs - according to the fixed firing time.

3.1.1 Epoch Time Generation

In this section, the first generated ASCII file with epoch times will be introduced. Table 3.1 on next page contents the first generated epoch time values. The table has to be read from left to right. This means that the first 33 values are in the first column on the left. The second 33 values are in the center column and the last 14 values are in the right column of the table.
The units of these epoch time values are in seconds. To convert these seconds in microseconds, it is necessary to multiply these values with 10^{-6}. If comparing one epoch time with the following epoch time, we notice that the main difference between these two times is 500 μs. As already mentioned in 3.1, the distance can be a little bit larger or smaller because of jitter in the range of $\pm$5 ns. To extract the information from the epoch times, it is essential to check up the distances between them.
The first step is to find a reference value in this example. In this present case it is easy to find this value, because it is known that the first value in this table cannot hold any information. In other words, this value can become a reference value for determining the information.
A simple and successful method to find the right zero value on receiver side is to send ten epoch time values with a distance of 500 μs in series. After receiving these values, the deviation of the epoch times from the 500 μs grid is determined. This simple method will be discussed in the next section.

0.123456788	0.139956784	0.156456788	0.172956783
0.123956788	0.140456789	0.156956780	0.173460380
0.124456797	0.140956783	0.157456799	0.173961637
0.124956789	0.141456785	0.157956796	0.174462191
0.125456793	0.141956780	0.158456783	0.174962186
0.125956794	0.142456784	0.158956782	0.175462328
0.126456797	0.142956782	0.159456798	0.175958373
0.126956788	0.143456793	0.159956783	0.176461130
0.127456781	0.143956793	0.160456788	0.176962027
0.127956786	0.144456780	0.160956790	0.177462186
0.128456793	0.144956794	0.161456787	0.177962177
0.128956784	0.145456798	0.161956790	0.178462042
0.129456791	0.145956788	0.162456789	0.178958377
0.129956781	0.146456787	0.162956789	0.179458431
0.130456792	0.146956793	0.163456795	
0.130956784	0.147456798	0.163956781	
0.131456794	0.147956790	0.164456784	
0.131956794	0.148456785	0.164956791	
0.132456798	0.148956796	0.165456792	
0.132956794	0.149456795	0.165956795	
0.133456786	0.149956789	0.166456790	
0.133956788	0.150456781	0.166956785	
0.134456781	0.150956782	0.167456789	
0.134956786	0.151456785	0.167956797	
0.135456790	0.151956791	0.168456792	
0.135956791	0.152456789	0.168956783	
0.136456796	0.152956792	0.169456798	
0.136956779	0.153456780	0.169956794	
0.137456797	0.153956791	0.170456781	
0.137956781	0.154456789	0.170956794	
0.138456788	0.154956784	0.171456784	
0.138956797	0.155456785	0.171956792	
0.139456791	0.155956796	0.172456794	

Table 3.1: First generated epoch times

The following steps describe the procedure to decode the contained message from the given epoch time values in table 3.1.

1. An appropriate reference value has to be found. In this case it is the first value in table 3.1

2. After determining and storing the reference epoch time the next typical epoch time value has to be calculated. As aforementioned, the distance between the next neighbored value is at least 500 μs. If it is 500 μs + N$*$50 ns, the epoch time value represents an ASCII character. Later, an ASCII character is represented with an offset of N$*$40 ns and finally with N$*$80 ns to minimize negative effects of jitter.

3. The next step is to compare the determined next epoch time value and the real epoch time value. This is just a simple subtraction of these two values. If the difference of the two value is less or equal than 5 ns, the present value holds no information and has to be ignored. After this step, the next but one theoretical value has to be determined and compared with the real next one. These operations have to be repeated until the last value of the table is reached.

3.1.2 Implementation of the Algorithm

In above subsection it is already mentioned that the easiest way to decode the message of the epoch time values is to develop an easy algorithm in Visual Basic. It is not necessary to type in every single value of the table. In Visual Basic 6.0 it is possible to get access to an ASCII file, if the path of that file is known by the programmer. There are lots of provided functions in Visual Basic, which allow to read every single value from the list automatically. Another function knows, when the last value of the list is reached and interrupts the reading process. On the next page, the most essential parts of the algorithm written in Visual Basic 6.0 will be discussed in detail. The declarations of the used variables will be ignored.

```
1  Private Sub Start_Click()
2
3  sFile = "D:\DA\WILLI.txt"
4  FNr = FreeFile
5
6  'open file
7  Open sFile For Input As #FNr
8
9  'read until the end
10 Counter = 0
11
12 Do While (Not (EOF(FNr)))
13
14     Line Input #FNr, sRow
15     Counter = Counter + 1
16     CurrentVariable = Val(sRow)
17     sArray(Counter) = CurrentVariable
18
19 Loop
20
21 Close #FNr
22 '------------------------------------------------------------------------
23 Counter = 0
24 FirstValue = sArray(1)
25
26 For Counter = 2 To 113 Step 1
27     CalculatedValue = FirstValue + (500 * 10 ^ -6) * (Counter - 1)
28     Difference = Abs(CalculatedValue - sArray(Counter))
29
30     If Difference = 0 Or Difference <= 10 * 10 ^ -9 Then
31         'do nothing
32
33     Else
34         AsciiN = Round(Difference / (50 * 10 ^ -9))
35         Message.Text = Message.Text & Chr(AsciiN)
36
37     End If
38 Next
39
40 End Sub
```

This source can be divided into two main parts. The first part is to get access to the ASCII file, which includes the list of the epoch time values of table 3.1. The second part decodes the message of the epoch time values. On line 1 a simple start button on the GUI starts the program. Line 7 opens the ASCII file with the Visual Basic function *Open*. This function needs for parameter the path of the file. This mentioned path is stored in a string variable called *sFile*. The while loop on line 12 is responsible for running through the list until the last value is reached. The actual value is always stored in the variable *CurrentVariable*. Before this procedure, the internal function *Line Input* has to read out the present value from the list. After that, this value has to be transformed in a "real value" by function *Val*. This means that after this transformation the value can be treated as a value of type double.
Every single epoch time is finally stored in an array of type double called *sArray* on line 17. The integer variable *Counter* is responsible for counting the positions of this array. The valid positions are from 1 to 113, because there are 113 values in table 3.1.
The main algorithm for extracting the message begins on line 23 and finally ends on line 38. As already discussed in subsection 3.1.1, the first value on the list is the reference value. With the help of this reference value, it is possible to calculate the theoretical next value. This value is needed for comparing it with the effective next value on the list.
On line 24 this first value of the list is stored into variable *FirstValue*. It is typical that this value can be found on the first position of the array *sArray*. After this storing, a simple for-loop is used on line 26 to go throw the *sArray* which holds the artificial epoch time values of the list. The integer variable *Counter* is used as counting variable and has to be initialized to zero on line 23.
The for-loop runs from position 2 until position 113, because on position 1 is the reference value and the total sum of values is 113. Inside this for-loop, the theoretical next value is determined and stored into the variable *CalculatedValue*. This value is 500 μs larger than the previous value. In this source code, the calculation always uses the reference value i.e. the first value of the list to determine the theoretical next value. That is the reason why these 500 μs are multiplied by *Counter-1*.
After this calculation, both values (the calculated next value and the real value) have to be compared. That's just a simple subtraction of these two values. This subtraction is executed on line 28. The possible difference is stored into the variable *Difference*. To be absolutely sure that the determined difference of the values is not negative, the internal function *Abs* is used and transforms a negative number in its positive number.
After this explained procedure, the program compares two cases with an if-condition to check if the distance of two neighbored times is greater or less than the usual jitter. The jitter is defined in the range of 10 ns. If the difference is inside this range, the relevant epoch time does not hold any information and will be ignored, but if it is outside this range, the program

calculates the hidden ASCII value on line 34 with the formula mentioned on the next page. Because of jitter, the result of equation in 3.1 could be a floating point number but ASCII characters are represented by integers. The ASCII value can be determined with following equation:

$$A = \frac{D}{50 * 10^{-9}} \tag{3.1}$$

$D =$ distance value

$A =$ ASCII value

In the present case these are integer values from 0 to 127. To be sure that the result is in a correct integer representation, it is finally important to round the result to the nearest integer value. On line 34 of the source code the function *Round* is used. After this function operation, it is guaranteed that the final result is an integer value. This final integer can be transfered to the function *Chr*, which substitutes the integer for the equivalent ASCII character. This can be seen on line 35. After this procedure, the message is displayed in a message box of the GUI. The final result of this program can be seen in the following.

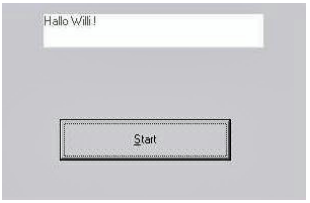

Figure 3.1: Program window after execution

The epoch time values represent the information “Hallo Willi !”. Next, this result of the Visual Basic program has to be analyzed in the following subsection.

3.1.3 Discussion of the Result

First of all, the algorithm needs a reference value for determining the theoretical next value. In this case, it is easy to find a reference value because it is the first value in table 3.1. After selecting this value, the theoretical next value is calculated.

$$ReferenceValue = 0.123456788 \tag{3.2}$$

The theoretical next value is determined by adding 500 μs to the reference value:

$$TheoreticalNextValue = ReferenceValue + 500\ \mu s \tag{3.3}$$
$$\Rightarrow TheoreticalNextValue = 0.123456788 + 500 * 10^{-6} \tag{3.4}$$
$$\Rightarrow TheoreticalNextValue = 0.123956788 \tag{3.5}$$

This result is equivalent with the second value of table 3.1. Finally the difference between the theoretical next value and the real next value is computed:

$$Difference = |TheoreticalNextValue - RealNextValue| \tag{3.6}$$
$$\Rightarrow Difference = |0.123956788 - 0.123956788| \tag{3.7}$$
$$\Rightarrow Difference = 0 \tag{3.8}$$

This means that this second value holds no information and has to be ignored. To be sure that the difference is always positive, the absolute value of the difference is used. If the difference was less or equal than 10 ns, the epoch time would not hold any information too. 10 ns is the maximum value of possible jitter. For a better understanding, the third value in table 3.1 is also determined in the following lines. The variable counter has to be increased, because the third value is normally $500 * 10^{-6} * 2$ greater than the reference value, which is also the first value.

$$ReferenceValue = 0.123456788 \tag{3.9}$$

$$TheoreticalNextValue = ReferenceValue + 500\ \mu s * Counter \tag{3.10}$$
$$\Rightarrow TheoreticalNextValue = 0.123456788 + 500 * 10^{-6} * 2 \tag{3.11}$$
$$\Rightarrow TheoreticalNextValue = 0.124456788 \tag{3.12}$$

After this calculations, the theoretical next value and the real next value are compared by computing the difference between them.

$$Difference = |TheoreticalNextValue - RealNextValue| \tag{3.13}$$
$$\Rightarrow Difference = |0.124456788 - 0.124456797| \tag{3.14}$$
$$\Rightarrow Difference = |-0.000000009| \tag{3.15}$$
$$\Rightarrow Difference = 0.000000009 \longrightarrow 9ns \tag{3.16}$$

This computed difference between these two values is in the range of the possible jitter. Therefore the third epoch time on table 3.1 is also useless and does not describe any ASCII character.The verification of the one hundredth and first value is a surprise. This epoch time

interprets an ASCII character and it is possible to extract it with the implemented functions of the source code on 3.1.2. The one hundredth and first value in the table is 0.173460380. To demonstrate, how the Visual Basic program works in this case, it is necessary to determine this ASCII character by hand.

$$TheoreticalNextValue = ReferenceValue + 500\ \mu s * Counter \tag{3.17}$$

$$\Rightarrow TheoreticalNextValue = 0.123456788 + 500 * 10^{-6} * 100 \tag{3.18}$$

$$\Rightarrow TheoreticalNextValue = 0.173456788 \tag{3.19}$$

As usual, the next step is to compare the computed theoretical next value with the real next value of the table 3.1

$$Difference = |TheoreticalNextValue - RealNextValue| \tag{3.20}$$

$$\Rightarrow Difference = |0.173456788 - 0.173460380| \tag{3.21}$$

$$\Rightarrow Difference = |-0.000000009| \Rightarrow Difference = 0.000003592 \longrightarrow 9\ ns \tag{3.22}$$

Equation 3.1 shows that this difference has to be divided by 50 ns.

$$ASCIIValue = \frac{Difference}{50 * 10^{-9}} \tag{3.23}$$

$$\Rightarrow ASCIIValue = \frac{0.000003592}{50 * 10^{-9}} \longrightarrow ASCIIValue = 71.84 \tag{3.24}$$

Unfortunately the result of the ASCII value is a floating point values and ASCII characters are just coded by integer numbers. It would not be possible to select the right ASCII character with a floating point value. The provided function *Chr* of Visual Basic, which transforms an ASCII value in its ASCII character would display an error on the screen, if this floating point value is used. Therefore the above result has to be rounded to the next nearest integer. This rounding is achieved by the function *Round* in the source code. After this rounding operation, the ASCII value is 72. That is the next nearest integer of 71.84. After that, this value can be looked up on an ASCII table e.g. in [6] and finally be substituted by the associated ASCII character. In the present case it is the ASCII character "**H**". The next twelve values in table 3.1 do also stand for ASCII characters. The introduced method for extracting them would of course also work but will not be demonstrated in this Master Thesis.

3.1.4 General Considerations

Above example is simple to analyse, because the reference value is very easy to select. It is just the first element in the list 3.1. The main goal of this example is to show and understand, how coding of ASCII characters works in this project and how these ASCII characters can be

decoded from the artificial epoch times.
Generally, it is not so easy to select a right reference value, because the SLR system and the CPLD can be switched on at anytime. In other words, the SLR system can send modulated laser signals before the CPLD can process them. This case appears e.g. if SLR system still sends laser signals before the CPLD is switched on. To select the first received epoch time value for reference value might become a big problem, because it is not known, what this value really means. It might be a redundant value, which does not present any ASCII character. In this case it would be correct to select it for the reference value, but in some circumstances the value stands for an ASCII character. To select such a value as reference value produces lots of problems and the correct decoding of a received information cannot be guaranteed.
To avoid these mentioned problems, it is necessary to choose an adequate agreement between sender and receiver. One possible agreement could be that the sender (SLR system) does not send the same character for more than nine times. If the receiver (CPLD and MPPC) detects the same character for ten times or more, the software determines the deviation of the epoch times from the 500 μs grid. Afterwards this deviation becomes the new reference value for extracting information. It is very important that the sender transmits the same values in a periodic interval for more then ten times to guarantee a correct interpretation of the transmitted information. In the final version of this project, a character cannot be send for more than 99 times, because to determine a reference value, the algorithm needs to find 100 equal values in the FIFO of the interface.
In the following example, it is agreed, that an ASCII character cannot be transmitted for more than nine times. If an ASCII character is transmitted for more than 99 times e.g. 100 times, the epoch time value, which represents this character would become the new reference value. Finally all transmitted characters are extracted with the help of this new reference value. This schema will be explained in the next section.

3.1.5 Determination of the Reference Value

As already mentioned in section 3.1.4, the turn-on instant of the CPLD and the Multi Pixel Photon Counting module could become a problem.
One solution of this problem might be that an equal epoch time value is send for at least ten times or more. The only difference between them is 500 μs (basic grid) plus the possible uncertainty of about $\pm$7 ns. The detailed explanation of this method is the main part of this section.
First of all, a new list of artificial epoch time values has to be generated. Some of these values hold again different ASCII characters, which have to be extracted by an algorithm programmed in Visual Basic. The old algorithm at 3.1.2 is not suitable for decoding the contained information, because it selects the first value as reference value. This procedure

would be totally wrong and a new algorithm has to be implemented, which counts equal epoch times. If the counter reaches the value 10, the present epoch time will become the new reference value. This process will be demonstrated in the next example. But before, some new epoch time values have to be generated.

0.123456788	0.139956784	0.156456788	0.172956783
0.123956788	0.140458539	0.156956780	0.173460380
0.124456797	0.140956783	0.157456799	0.173962037
0.124956789	0.141456785	0.157960296	0.174458391
0.125457043	0.141956780	0.158456783	0.174961136
0.125956794	0.142456784	0.158956782	0.175462028
0.126456797	0.142958782	0.159456798	0.175962173
0.126956788	0.143456793	0.159956783	0.176461980
0.127456781	0.143956793	0.160460538	0.176961827
0.127957286	0.144456780	0.160956790	0.177462186
0.128456793	0.144956794	0.161456787	0.177962227
0.128956784	0.145459048	0.161956790	0.178458392
0.129456791	0.145956788	0.162456789	0.178958427
0.129956781	0.146456787	0.162960789	0.179458431
0.130457542	0.146956793	0.163456795	
0.130956784	0.147456798	0.163956781	
0.131456794	0.147959290	0.164456784	
0.131956794	0.148456785	0.164956791	
0.132456798	0.148956796	0.165456792	
0.132957794	0.149456795	0.165956795	
0.133456786	0.149956789	0.166456790	
0.133956788	0.150459531	0.166956785	
0.134456781	0.150956782	0.167456789	
0.134956786	0.151456785	0.167956797	
0.135458040	0.151956791	0.168456792	
0.135956791	0.152456789	0.168956783	
0.136456796	0.152959792	0.169456798	
0.136956779	0.153456780	0.169956794	
0.137456797	0.153956791	0.170456781	
0.137958281	0.154456789	0.170956794	
0.138456788	0.154956784	0.171456784	
0.138956797	0.155460035	0.171956792	
0.139456791	0.155956796	0.172456794	

Table 3.2: Second generated epoch times

Table 3.2 has to be read in the same way like table 3.1. To extract the contained information of these published epoch times it is essential to count the epoch times, which do not describe any ASCII characters. These are epoch time values, with a distance of 500 μs plus an uncertainty of $\pm$7 ns to their neighbored values. In this example it is adequate to find ten such values. In practical realization, it is necessary to find 100 equal values.
This procedure will be explained in the next section of this thesis. There are again 113 artificial epoch time values (unit = 1 second) in table 3.2. As already mentioned, ten equal epoch times have to be detected. This is the main difference between the old and the new algorithm. The whole programmed algorithm, which was programmed in Microsoft Visual Basic 6.0 is shown and explained in detail. Non relevant statements for explanations e.g. declarations of variables are ignored.

```
1  Do While Counter < 9
2      ArrayIndex = ArrayIndex + 1
3      Difference = Abs(sArray(ArrayIndex + 1) - sArray(ArrayIndex))
4
5      If Difference = Distance Or Difference <= Distance + NFactor Then
6          Counter = Counter + 1
7      Else
8          Counter = 0
9      End If
10
11 Loop
12
13 If Counter >= 9 Then
14     ReferenceValue = sArray(ArrayIndex)
15     NValues = 113
16
17     For ArrayIndex = ArrayIndex + 1 To NValues Step 1
18         Factor = Factor + 1
19         CalculatedValue = ReferenceValue + (Distance) * Factor
20         Difference = Abs(CalculatedValue - sArray(ArrayIndex))
21         AsciiN = Round(Difference / (NFactor))
22         MessageArray(Factor) = Chr(AsciiN)
23     Next
24
25 End If
26
27 ArrayIndex = 0
28 Message.Text = " "
29
30 For ArrayIndex = 1 To 113 Step 1
```

```
31      Message.Text = Message.Text & MessageArray(ArrayIndex)
32  Next
```

This short algorithm is able to extract the contained information from the values in table 3.2. As already mentioned, the main difference of this algorithm is that equal epoch time values with 500 μs time intervals are counted and afterwards the counter is checked. On line 1, a simple while-loop checks if variable *Counter* has already reached value 9. If the value is less than 9, the integer variable *ArrayIndex* will be incremented. *ArrayIndex* is a control variable for array *sArray*, in which the 113 epoch time values are stored. In other words, *sArray* has 113 available valid storage locations. On line 3, the difference of two neighbored values is calculated and stored in variable *Difference*. To be absolutely sure that the difference is positive, the internal function *Abs* is used. On line 5, the calculated difference between these two value is checked. If the difference is equal to 500 μs or equal to 500 μs plus the *NFactor*, the variable *Counter* is incremented. *NFactor* is a constant and holds a value of 50 ns. This value describes the offset, when a single character is transmitted. If the difference between two neighbored epoch time values just amounts 500 μs, no ASCII character is transmitted. Otherwise if the difference amounts 500 μs plus 50 ns, the two relevant values describe the same character. Equation 3.1 defines this case.
If two neighbored values do not fulfill this condition, variable *Counter* is reseted, no matter how often this variable was incremented in the past. All other values, which were received before, are ignored. One of the most important facts is, that variable *Counter* has to be 10. In other respects, no reference value was found and the algorithm is not able to extract the transmitted information and the program would never leave the first nine lines and would terminate when *sArray* is fully processed. If *Counter* becomes 10, the present epoch time value is accepted as reference value.
NValues is initialized to 113, that is the number of the existing values in *sArray*. The for-loop on line 17 is responsible for the ASCII character calculation. This is similar to the computation in source 3.1.2 and need not to be explained furthermore. Finally the determined characters are stored in an character array called *MessageArray*. This array can store maximum 113 characters, but it will not become full in this present case, because most values of *sArray* are useless as it is discussed above.
The last six lines are responsible for displaying the extracted message on the computer screen. The simple for-loop runs through the *MessageArray* and prints out character for character. After this procedure, the algorithm terminates. In the following the final output of the introduced algorithm is shown.

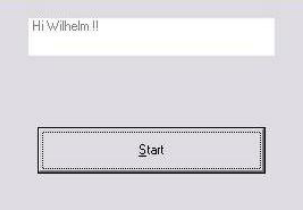

Figure 3.2: Final output

Figure 3.2 shows the final result of the calculation. The resultant message of the epoch times of table 3.2 is “Hi Wilhelm !!”. The used reference value is the 89th epoch time in the list. That is the tenth equal value. In other words, the previous nine values describe the same character or they are just empty epoch times, which do not mean anything. A closer look to these values makes it clear that they have a distance of 500 μs plus an uncertainty of maximum 5 ns. This is typical for epoch times, which do not stand for any ASCII characters. The next table contains these values for a better overview.

0.162456789	
0.162960789	
0.163456795	
0.163956781	
0.164456784	
0.164956791	*ten equal epoch times*
0.165456792	
0.165956795	
0.166456790	
0.166956785	

The last value in this table is the reference value. That is the tenth equal value of the epoch times. The algorithm compares each value with its neighbored value.

0.162456789	*Counter = 1*
0.162960789	
0.162960789	*Counter = 2*
0.163456795	
0.163456795	*Counter = 3*
0.163956781	
0.163956781	*Counter = 4*
0.164456784	

$$\left.\begin{array}{l}0.164456784\\0.164956791\end{array}\right\} \textit{Counter} = 5$$

$$\left.\begin{array}{l}0.164956791\\0.165456792\end{array}\right\} \textit{Counter} = 6$$

$$\left.\begin{array}{l}0.165456792\\0.165956795\end{array}\right\} \textit{Counter} = 7$$

$$\left.\begin{array}{l}0.165956795\\0.166456790\end{array}\right\} \textit{Counter} = 8$$

$$\left.\begin{array}{l}0.166456790\\\mathbf{0.166956785}\end{array}\right\} \textit{Counter} = 9$$

The tables above show, how the algorithm selects the right reference value for extracting the ASCII characters. Now it should be absolutely clear, that variable *Counter* has to be 9, when a matching reference value is found. This value is printed in bold. Afterwards the message is extracted with the same procedure like in the previous example.

3.1.6 Important Remarks

The main goal of these two very simple examples is to get a feeling for epoch time values and extracting messages, which they could transmit and to get experiences in developing small programs in Microsoft Visual Basic 6.0 without using any hardware. In a later section, it will become clear that also some hardware is needed. There are quite a few problems in using hardware and software together. One big problem is the connection or the communication to the software. Therefor a special driver is needed. Very often the manufacturers of the hardware offer drivers for several programming languages. These hardware drivers or DLLs provide lots of functions, which allow access to the hardware with the help of a programming language. To use these special provided functions is not always easy, because the programmer has to know quite a few technical facts e.g. data types of the parameter values or the structure of the hardware. In other words, the programmer has to study the hardware manual very strictly. But before discussing the hardware, it is important to think about developing a software, which is also able to generate epoch times and finally decode them in just one step. This software project will be introduced in the following section 3.2

3.2 Implemention of the Epoch Time Generator

In the last section, the epoch times were generated by an extern source and stored in an ordinary ASCII file. Afterwards this file was ported to a PC, on which algorithm 3.1.2 or algorithm 3.1.5 was executed. After these steps the ASCII characters were determined.
These procedures cost a lot of time to get a result. A good idea would be to think about generating and decoding epoch time values on the same system. For this challenge, it is needful to study some possibilities to read in a text file and to convert it in time values. Next, the program should read in the time values and extract the contained message. After decoding the message, the program should display the result on the screen. Extracting the message is not a real problem, because it is already done by the previous two algorithms. One of these two algorithms can be adopted without changing anything.
The main challenge is to read in any text from an ASCII file and to generate epoch times in Visual Basic 6. For the generation, an internal function called *timeGetTime* is needed. This function is able to access to the system time of the PC system and is included in "winmm.dll". The *timeGetTime* function retrieves the system time in milliseconds. The system time is the time elapsed since Microsoft Windows was started. On the next line the declaration of this function is shown.

```
Private Declare Function timeGetTime Lib "winmm.dll" () As Long
```

The return value of this function is from data type long. In other words, the system time is a 32-bit number which can range from -2.147.483.648 to 2.147.483.647. The return value or rather the system time of the PC system is used as starting point for the generated epoch time values. This means that the first epoch time is the return value of the function *timeGetTime*. After this procedure, 500 μs must be added for the second epoch time value, 500 μs $*$ 2 for the third epoch time value and 500 μs $*$ (n-1) for the n-th epoch time value.

3.2.1 Simulation of Jitter

To provide a veridical simulation, a special function, which includes a random generator was programmed. This function is responsible for simulating the uncertainty or rather the jitter. In this example, the jitter is defined in the range of $\pm$5 ns. In the following, this mentioned function is shown.

```
1 Public Function RandomValue() As Double 'for jitter
2
3 Dim RandomVariable As Double
4 RandomVariable = 0
5
6 RandomVariable = Rnd 'value between 0 and 1
```

```
7  RandomVariable = RandomVariable - 0.5
8  RandomVariable = RandomVariable * 10 ^ (-8) 'nanoseconds are needed
9  RandomValue = RandomVariable 'return value
10
11 End Function
```

As above seen, the return value is from data type double. The double data type is a 64-bit signed floating point number. On line 6, the internal random generator function *Rnd* is called. This function generates random numbers between 0 and 1. The random numbers are stored in the variable *RandomVariable*, a variable of data type double. The largest number, which can be provided by this function is 1 the smallest one is 0. To get numbers in the range of ±5 ns, some operations are needed. In the next line, the value 0.5 is subtracted from the random value. After that, it is obvious that the random value is in the range of ±0.5. To provide values in nanoseconds, it is necessary to multiply the final value with 10^{-8}. Finally, the function returns this value and added to the present epoch time value.
These are the main procedures to simulate the jitter of this software example. In the next subsection the main determination of epoch times will be discussed.

3.2.2 Determination of Epoch Times

The determination of the artificial epoch times is done in the self programmed function *GetEpochTime*. This function does not need any function parameters. The return value of this function is of data type double. That is the epoch time which describes an ASCII character. Before the coding starts, the function generates 100 zero epoch time values. These are values, which do not describe an ASCII character. They are necessary for finding the reference value, but this fact was already mentioned a few subsections before.
First of all, it is important to explain the function *GetCurrentTime*. This is a very short and simple function, which includes the *timeGettime* function. *GetCurrentTime* offers the current system time of the PC system, but it is not identical to *timeGettime*, because it transforms the system time in milliseconds to a system time in microseconds. That is just a simple multiplication with 10^{-9}.

```
1  Public Function GetCurrentTime() As Double
2
3  GetCurrentTime = timeGetTime * 10 ^ -9
4
5  End Function
```

As shown below, the function *GetCurrentTime* does not take any parameters. After calling, the function provides the current system time in microseconds. It is used by the function *GetEpochTime*, which will be introduced next.
GetEpochTime uses the mentioned functions *GetCurrentTime* and *RandomValue* for generating zero epoch time values.

```
1  Public Function GetEpochTime() As Double
2
3  Dim EpochTime As Double
4  Dim Counter As Integer
5  Dim Jitter As Double
6  Dim CurrentTime As Double
7  Const Distance = 500 * 10 ^ -6
8
9  List1.Clear
10 CurrentTime = GetCurrentTime
11 Counter = 0
12 Jitter = 0
13 EpochTime = CurrentTime
14
15 For Counter = 1 To 100 Step 1
16     Jitter = RandomValue
17     EpochTime = EpochTime + (Distance) + Jitter
18     List1.AddItem (EpochTime)
19 Next Counter
20
21 If Counter >= 100 Then
22     GetEpochTime = EpochTime 'return value
23 End If
24
25 End Function
```

The return value is from data type double. It does not need any functional parameters. After calling this function, some variables are defined on line 3 to line 7. These are explained on the next page

- *EpochTime:* Stores the final result and passes this value to the function. This is finally the return value of this function.
- *Counter:* Control variable of the for-loop on line 15.
- *Jitter:* Stores the return value of *RandomVariable*, which is added to the epoch time value.
- *CurrentTime:* Holds the system time in microseconds. This times is generated by the function *GetCurrentTime*.
- *Distance:* Is a constant and defines the distance between the epoch time values. The main distance between an epoch time value to its neighbored value is always 500 μs.

The next lines initialize these discussed variables with function calls or zero values. Line 13 provides the first epoch time. That is the system time in microseconds. The for-loop on line 15 generates 100 other epoch time values with an artificial jitter, provided by function *RandomValue*. Line 17 generates a zero epoch time value. The result is stored in variable *EpochTime*. After the first cycle, the next epoch time is determined with the help of the last epoch time. Only an addition of 500 μs and random jitter is needed to get the next time value. Unfortunately, this line contains an error propagation, because the jitter values of all epoch times are summed up.
The next step on line 18 is that the present zero epoch time is written into a list. Later, the program reads this list line by line to determine the ASCII characters. This will be discussed later.
After 100 cycles, the last generated epoch time value is the return value of this function. Line 22 shows this procedure. The goal of this function is to produce empty epoch time values. It does not code any information. The meaning of zero epoch times is to find an adequate reference value for decoding the contained information.

3.2.3 Coding of ASCII Characters

One of the main parts of this algorithm is to code the ASCII characters of the ASCII file. This is done in function *ReadLetter*. The ASCII file on the hard disk is read character by character. After reading a letter, the letter is coded into an ASCII value. This job is done by the already discussed internal function *Chr*. After a line break, the function reacts with "carriage return" and "line feed" values. In other words, after a line break, the function generates two additional epoch time values, which describe the carriage return and the line feed. The coding of these two instructions is equal to the coding of usual ASCII characters. The ASCII code for the carriage return is 13 and for the line feed 10.
On the next page, some details of this function are shown. Important to mention is that

the declarations of the used variables are not contained in the following source in order to provide a better overview.

```
Public Function ReadLetter(ByVal sFilename As String) _
    As String

Text1.Text = "" 'textbox must be empty for restart
Counter = 0
LastEpochTime = GetEpochTime
FNr = FreeFile
Open sFilename For Input As #FNr
Text1.Text = " "

Do While (Not (EOF(FNr)))
    Line Input #FNr, strInput

    For Counter = 1 To Len(strInput) Step 1
        Jitter = RandomValue 'random generator
        Letter = Mid(strInput, Counter, 1)
        AsciiValue = Asc(Letter)
        Text1.Text = Text1.Text & Chr$(13) & Chr$(10)
        Text1.Text = Text1.Text & "   Letter:" & Letter
        Text1.Text = Text1.Text & "   AsciiValue:" & AsciiValue
        EpochTime = LastEpochTime + (Distance) * Counter
        CodedValue = EpochTime + Jitter + (N * AsciiValue)
        List1.AddItem (CodedValue)
    Next Counter

    If Counter = Len(strInput) + 1 Then
        'we know that there is a new line in the ASCII file
        'if this condition is fulfilled
        EpochTime = LastEpochTime + (Distance) * (Counter)
        CodedValue = EpochTime + Jitter + (N * 13)
        List1.AddItem (CodedValue)
        EpochTime = LastEpochTime + (Distance) * (Counter + 1)
        CodedValue = EpochTime + Jitter + (N * 10)
        List1.AddItem (CodedValue)
        LastEpochTime = EpochTime
    End If
Loop
ReadLetter = strInput
Close #FNr
End Function
```

This function needs a function parameter. This parameter is the filename respectively the path of the information file. Line 6 retrieves the last empty epoch time, generated by the last discussed function *GetEpochTime*. This epoch time value is the return value of *GetEpochTime*. Line 7 and line 8 is responsible for access to the ASCII file. This file contains the information, which the function has to code in epoch time values. The while loop on line 11 reads line by line from the file and stops after the end of the file is reached.
The for-loop on line 14 reads a line and stops after the end of the line is reached. *Len(strInput)* stores the length of the present line. The variable *strInput* is from data type string and stores the whole present line of the ASCII file. *Len* is also an internal function and provides the length of any string.
The internal function *Mid* analyzes the present string, respectively the present line character by character. After that, the present character is transformed into its ASCII value on line 17. Line 18 to line 20 are just debug statements and not important to mention. They are just for displaying the results in a text box. The program would also compile correctly and work without these few lines.
Line 21 determines the present epoch time. This time consists of the last empty epoch time, generated by the discussed function *GetEpochTime* and the distance. The distance is calculated by the help of the control variable *Counter*. For example, if the *Counter* is 3, the distance between the last epoch time and the present epoch time is 1500 μs (500 μs $*$ 3).
Line 23 codes the ASCII character with the result of line 21. N is the offset and has to be added to all epoch times, which hold any characters. In this program, the variable N is defined as a constant. This constant holds a value of 50 ns. This is the offset between the coded values. On the next line, the ASCII character is coded and stored in variable *CodedValue*. As it is below mentioned, the variable *AsciiValue* stores the value of the present character. This value is determined on line 17.
On line 23, the final value, respectively the epoch time value is added into a list.
Line 26 manages the end of a line. When the end of a line is reached, the program runs to this if-condition and adds two coded epoch time values. In the ASCII world, a line break is defined by two special values, a carriage return and a line feet. These are the values 13 and 10. After coding a line break, the two epoch time values, are also added in the list of the coded values.
This is the main procedure, how a line of any ASCII file is coded into its epoch time values. This sequence is repeated until the end of the file is reached. Responsible for reaching the end of the file is the discussed while-loop on line 11.
After reaching the end of the file, the program retrieves and reads in the epoch time values and decodes them into its original ASCII characters. In the next subsection, this procedure will be explained.

3.2.4 Decoding of ASCII Characters

In principle, the decoding of the generated epoch time is still the same as already explained in subsection 3.1.5. It is not necessary to publish the decoding algorithm in this thesis, because the difference between the implemented algorithm in this explained program and the decoding algorithm in 3.1.5 is quite smallish. The only difference is, that the algorithm needs to find 100 empty epoch time values to determine a reference value. These 100 epoch times are generated by the discussed function *GetEpochTime* and finally added to the list. This is the only difference.

3.2.5 Execution of the Program

Before the program can be executed, a simple ASCII file has to be written. Next, the main program is explained.

```
1 Private Sub Start_Click()
2
3 Const Path = "C:\Dokumente und Einstellungen\Wilhelm Steinegger\
    Desktop\string_readin\read3.txt"
4 Call ReadLetter(Path)
5 Call Decode
6
7 End Sub
```

When the Command1 button is clicked, the algorithm gets executed. Important is, that the program has to know, where the ASCII file is stored on the hard disk.
On line 3, the path of the file is stored in a constant. This path has to be adapted, if the program has to run on an other PC system. Of course, it would be reasonable, if the path could be changed on the Graphical User Interface, but it is not the goal of the program to develop a user friendly interface. It is much more important to show, how coding and decoding of epoch times can happen.
Line 4 calls the function *ReadLetter*. As already discussed, this function reads letter by letter and codes them to their epoch time values. The function gets as parameter the defined path, which is a constant.
After this event, the function *Decode* is called. This function gets access to the list of all epoch time values and determines the ASCII characters. Finally it writes character by character to a text box, which can be seen on the GUI. On the next page, the Graphical User Interface will be discussed.

3.2.6 Graphical User Interface

The GUI is very simple and without any extensive functions, as already mentioned in 3.2.5. It only has one window to interact with the user. A start button executes the program and the algorithm begins to translate the stored ASCII file. After the execution, the original information is printed in a textbox.

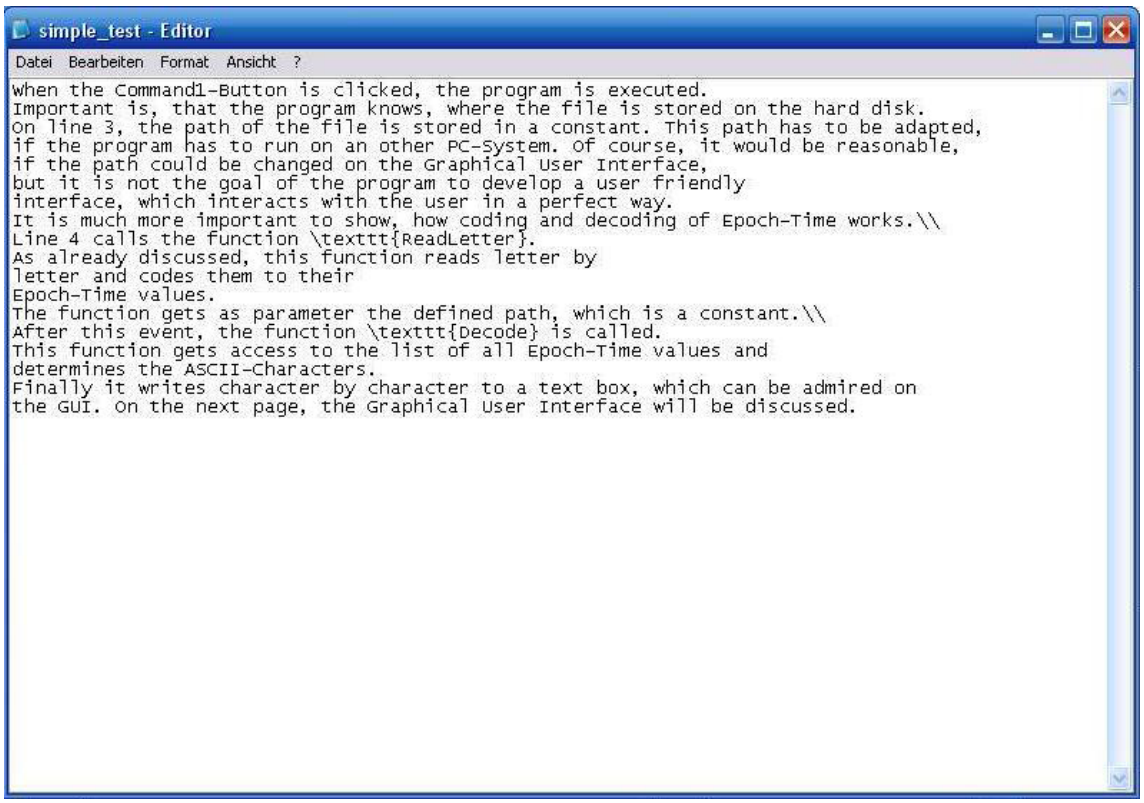

Figure 3.3: ASCII editor

The figure 3.3 shows the ASCII editor with the contained information. Above mentioned, the information is the LaTeXsource in subsection 3.2.5. The file is called “simple_test”. This is important for the constant path in the main program.

```
Const Path = "C:\Dokumente und Einstellungen\Wilhelm Steinegger\
    Desktop\string_readin\simple_test.txt"
```

After adapting the path, the user can start the program by clicking the start button. Next, the GUI is shown after executing the program.

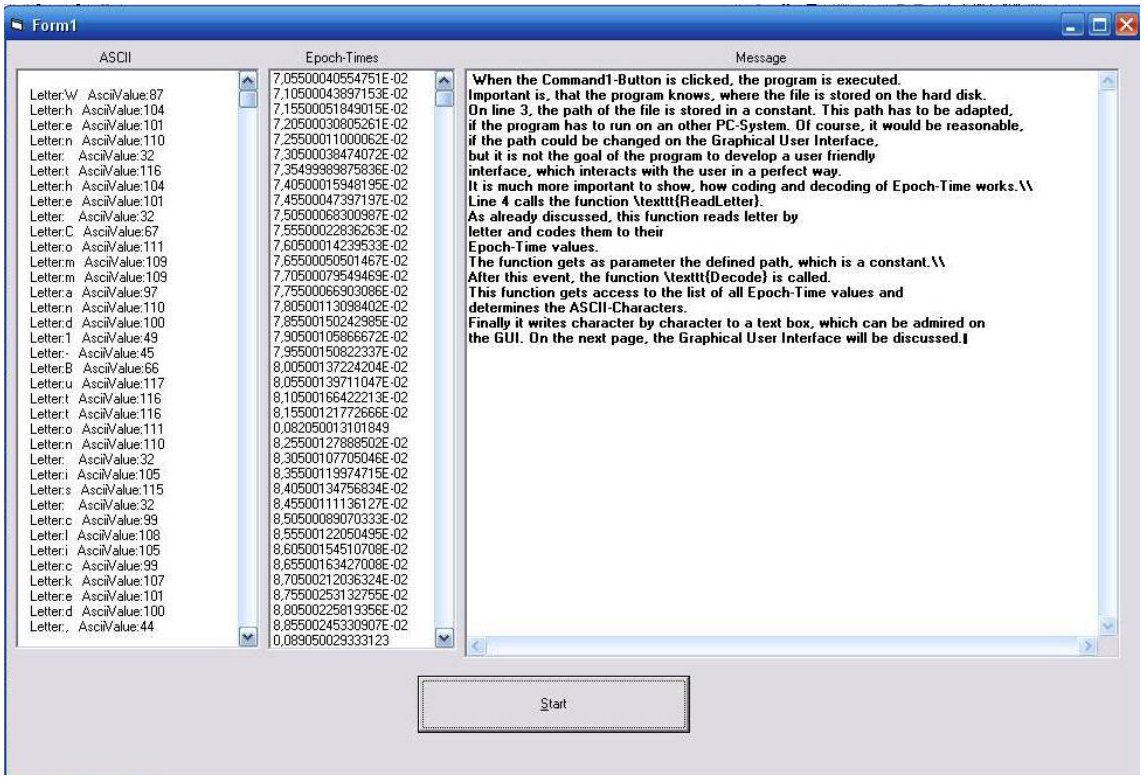

Figure 3.4: **G**raphical **U**ser **I**nterface (GUI)

The list box with the scrollbar on the left contents the ASCII characters, which were coded and finally decoded in its original form. The list box in the middle of the GUI shows the epoch time values, which are generated by the two discussed functions. The algorithm reads in these values and transforms them into their original characters. The message box on the right side finally contains the decoded message. This message has to be equivalent to the message shown on figure 3.3. After taking a deep look to the GUI respectively to the message box, the original message can be recognized. Important is, that the line breaks are handled in a correct way.

3.3 Final Remarks to chapter 3

This chapter explained some basic steps to develop an adapted software for coding and decoding information with the help of artificial epoch time values.
These discussed programs are not suitable for the final project, because they do not interact with any hardware.
Coding information as it is shown in the last example is not necessary, because that is done by an ISA PC card. This PC card controls the laser firing times and delays them to generate coded ASCII characters.

The laser signals are detected by a MPPC (Multi-Pixel-Photon-Counter) module. This module is connected to a CPLD, which determines the delay times of the laser pulses. This is managed by a counter in the CPLD. Finally the values of this counter are stored in a register, which may hold a specific number of values. The values are four bytes long. The next part of this project is to get these mentioned values into the PC. This is done with a fast USB interface. The last part of this project is to analyze the counter values with a programmed software.
One of the main challenges in this project was to find a suitable interface for reading in the epoch time values. As mentioned previously, the laser of Graz / Lustbühel has a frequency of 2 kHz. From this it follows that a used interface has to be able to read in at least 2000 epoch time values per second. One epoch time consists of four bytes. Therefore the interface has to read 8000 bytes per second.
Unfortunately, the first selected interface had a too slow throughput and had to be replaced. Another requirement is an easy connection to the PC or rather to a programming language like Visual Basic. In the next chapter, two different interfaces will be introduced.

4 Hardware

This chapter discusses the used hardware of this laser project. As already mentioned in section 3.3, this work needs to deal with four hardware parts.
The first hardware part is the ISA card, which is responsible for controlling the laser firing intervals.
The second part is the USB interface. Many of them have too slow transfer rates and the FIFO (First-In-First-Out) of the CPLD, which stores the four bytes epoch time values, gets a memory overflow, because the software would not be able to fetch the values from the FIFO in a suitable period of time. The effect of a too slow interface is data loss. Most of the values are corrupted, because they cannot get any storage positions in the FIFO of the CPLD. The software would read in the values in a wrong sequence and the effect would be wrong interpreted informations with no meaning.
The MPPC is manufactured by Hamamatsu. It is a new type of photon counting device made up of multiple APD (avalanche photodiode) pixels operated in Geiger mode. The MPPC is essentially an opto-semiconductor device with excellent photon-counting capability and which also possesses great advantages such as low voltage and insensitivity to magnetic fields. The MPPC is available with 25 μm pitch (1600 pixels), 50 μm pitch (400 pixels) and 100 μm pitch (100 pixels). For this project, a MPPC with 50 μm pitch (400 pixels) was chosen.

4.1 ISA Card

The above mentioned ISA card is responsible for controlling the laser firing times. On the ISA card there is a programmable FPGA from Altera. Originally, the laser firing times were in constant 500 μs time intervals. To code information with the pulse position modulation, as explained in 2, the laser firing times have to vary depending on the present ASCII character. To provide this, the FPGA chip on the ISA card has to be reprogrammed. This reprogramming of the ISA card was not part of this Master Thesis and will not be explained.

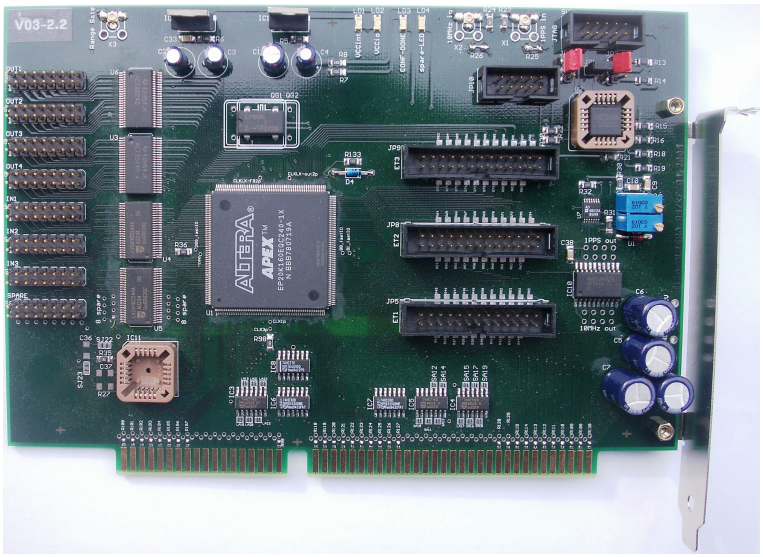

Figure 4.1: ISA Card for controlling the laser firing times

As mentioned in the introduction, the transfer rate of the interface is one of the most important factors in developing software, which has to deal with hardware. In this Master Thesis, two different USB interfaces are introduced. The USB-Programmed Input / Output is an USB interface. The biggest drawback of it is the transfer rate. It is much too slow for this laser communication system. At the beginning of this project, the decision came to the USB-PIO, because it is one of the easiest to program interfaces. After some experiments with empty epoch times in the CPLD respectively in the FIFO, the very slow transfer rate were revealed. Next, an USB interface, USBMOD245R was installed to read out the data from the CPLD. This interface has a very high throughput respectively a very fast transfer rate up to 1 MByte/s Before this interface is introduced, it is important to illustrate some facts about the USB-PIO.

4.2 USB-Programmed Input / Output

The following facts about the digital I/O interface are cited from the data sheet [3]. The USB-PIO has three 8 bit ports. The direction of these ports can be switched. The connection of the USB-PIO can be managed via USB. The device is included in a Sub-D case. The current supply is also arranged via USB. No extern power supply is needed. Figure 4.2 shows the functional principle of the interface. The 24 Pins are divided into three 8 bit ports (0-7). Pin 25 is the ground and has no function and it is not possible to switch this pin. The device is offered with Next View 4 Live, a software, which can test the whole functionality of this product. Unfortunately, the provided version on the CD-Rom is just a demo version. So it is not possible to store any data or graphs. The full version, which offers a lot more functions,

has to be bought. By the way, that is just an additional information. Next View 4 is not be used in this project. Before this device can be used, the driver package supported by the manufacturer has to be installed.

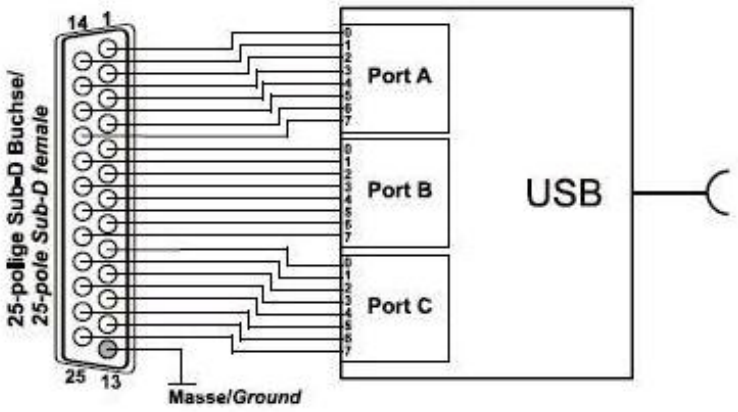

Figure 4.2: Functional diagram of the USB-PIO [3]

Digital I/0

The USB-PIO contains a μ-Controller, which offers three 8-Bit digital ports. The lines are bidirectional i.e. the input and output directions can be programmed via software. Important is, that a single line cannot be switched to another direction, because all lines are related to one of the three ports. It is just possible to switch the whole port with 8 lines. On port C, the direction can be switched in two groups of four lines. This is useful for implementing an easy handshake method.

Programming the USB-PIO

To program the USB-PIO, the manufacturer provides a special library for common programming languages (C++, Microsoft Visual C++, Microsoft Visual Basic (Version 4 to Version 6), Delphi and LabView). LibadX is implemented as *ActiveX Control (OCX)* that is registered by the installation program on the PC. After installing the OCX, the programmer can access to the USB-PIO very easy. Many digital functions are provided. The most important ones will be explained in the following.

OCX Functions

The OCX functions can be used for programming the interface, e.g.: for opening the device or for setting the line directions. All these functions are introduced in the programmers guide. Every USB-PIO has an unique serial number - this number must be used for opening the connection. The following source code is just a short sets the direction of the existing ports.

```
1  Private Sub SetPortDirection()
2  'This function sets the direction of the USB-PIO at the
3  'beginning of the program.
4  'It has no return value, because it just sets the direction
5  'of the port. If a port is defined as "DirOut", it is set as an
6  'output port. If a port is defined as "DirIn", it is set as
7  'an input port. The function uses the DLL "meMPIO1"
8
9  With meMPIO1
10         .DirPort1 = DirOut 'Port1 1...8
11         .DirPort2 = DirIn  'Port2 1...8
12         .DirPort3L = DirOut 'Port3L 1...4
13         .DirPort3H = DirIn  'Port3H 4...8
14 End With
15
16 End Sub
```

This example shows how the directions of the existing ports may be switched. The first port is set for output like the port 3L. The rest of the ports are set for input. The output ports can manage the communication with the CPLD, for example to check if the CPLD is ready for data transmission or if there are any data in the FIFO of the CPLD. The CPLD can communicate with the interface respectively with the software by the help of the lines of port 3H. The data transmission can be done by port 2. It is remarkable, that one port can receive 1 byte, because a port contains of 8 lines. 1 Line can describe 1 bit (voltage = 1, no voltage = 0). In this case, the port needs to be set as input respectively as *DirIn*. Also transmitting one byte is possible. Before that is possible, the port has to be set as output *DirOut*. For setting one line to high, the function *SetLine(Port, Line)* is used. To set a line to status low, the function *ResetLine(Port, Line)* is used. Normally, if one line is not set to high, the line is low. But to be sure that it is really low, it is recommendable to reset the line with the below mentioned function *ResetLine(Port, Line)*. For this project, a program for reading out epoch time values was designed and programmed with the help of these functions and this interface, but finally it was too slow for transferring data to the PC and has not been used in the final version. In the following, the main concept of this software is explained, but before, it is necessary to discuss the functions of the single pins.

Port	Line	Direction	Function
1	1	OUT	Binary Position: 2^0 or 2^8
1	2	OUT	Binary Position: 2^1 or 2^9
1	3	OUT	Binary Position: 2^2 or 2^{10}
1	4	OUT	Binary Position: 2^3 or 2^{11}
1	5	OUT	Binary Position: 2^4 or 2^{12}
1	6	OUT	Binary Position: 2^5 or 2^{13}
1	7	OUT	Binary Position: 2^6 or 2^{14}
1	8	OUT	Binary Position: 2^7 or 2^{15}
2	1	IN	Binary Position: 2^0 or 2^8
2	2	IN	Binary Position: 2^1 or 2^9
2	3	IN	Binary Position: 2^2 or 2^{10}
2	4	IN	Binary Position: 2^3 or 2^{11}
2	5	IN	Binary Position: 2^4 or 2^{12}
2	6	IN	Binary Position: 2^5 or 2^{13}
2	7	IN	Binary Position: 2^6 or 2^{14}
2	8	IN	Binary Position: 2^7 or 2^{15}
3L	1	OUT	STOP
3L	2	OUT	RESET
3L	3	OUT	READ
3L	4	OUT	SIMULATE
3H	5	IN	NOT USED
3H	6	IN	NOT USED
3H	7	IN	NOT USED
3H	8	IN	DATA AVAILABLE

Table 4.1: Description of the Single Pins

Table 4.1 gives a detailed overview about the different functions of the available pins. Port 1 and port 2 are for representing epoch time values. One single pin represents one single bit. It is necessary to get a 16-bit epoch time value, which is almost impossible, because the port only have 8 lines respectively pins. So in the normal case, the port can hold an 8 bit value. To provide more than 8 bits, the epoch time can be factorized into two 8 bit values. In other words, the FIFO of the CPLD holds 8 bit values and these 8 bit values can be read out by the port 2 which contains of 8 lines. The first 8 bits are the least significant ones, the second 8 bits are the ”most-significant“ ones. This is the most important fact, which the programmer has to know.
Port 1 is similar. It is also for representing an 8 bit. The only difference is that the software can set these lines and write it to the FIFO of the CPLD. That is possible, because port 1 is set as output. The main idea was to upgrade the software, presented in section 3.2, with this option. The program in section 3.2 reads letter by letter from the provided ASCII-File and

codes letter by letter in an epoch time. After that procedure, the epoch time value has to be transformed into a binary representation and transmitted to the CPLD by the lines of the port 1.
Unfortunately, this option could not be realized, because while programming and simulating the read out process, an interesting fact occurred. The USB-PIO is much too slow for transferring data to the PC and to the CPLD. The effect of this fact is that the FIFO of the CPLD becomes full and can not store any more values. The new values get lost respectively cannot be stored in the FIFO anymore. This is, because the read out process has to be faster than the read in process to provide free storage positions in the FIFO.
With a very simple handshake procedure it is possible to control the access to the CPLD respectively to the data of the FIFO. Therefor port 3L and port 3H can be helpful to realize this access control. In this project, this simple handshake protocol was implemented on the Altera CPLD, but as already mentioned below, it was too slow for the data communication. For measuring the time for reading out values, a simple stop watch function *GetTickCount Lib "kernel32" () As Long* was used in Visual Basic. This function returns values in the microsecond range. In the worst case, it delivers a value of about 25 ms for reading out one epoch time value from the CPLD. After realizing this fact, it is absolutely clear that this interface is useless for this job, because epoch time values arrive in 500 μs intervals to the CPLD and it becomes conceivable that the FIFO produces a data overflow in a short period of time. The exact time was not measured, because the main concern was to find an optimal interface for this project. Next, the mentioned handshake process will be explained.

Handshake Model

Normally all lines, which are not set to high are in status low, but to be absolutely sure, these lines are reseted with the provided function *ResetLine(Port,Line)* before the USB-PIO begins to operate with the CPLD. Most important to know is, that just lines with output direction can be reseted. All lines, which are set as input, can just be sampled. For reseting all lines, special functions were programmed to provide a better overview. In the following, a simple function is shown, which resets the simulation line.

```
1 Private Sub ResetSimulation()
2 'resets the Simulation Pin on the CPLD. This Subroutine uses the
3 'dll meMPIO1 the Simulation Pin is on Port 3, Pin 4
4
5 meMPIO1.ResetLine 3, 4
6
7 End Sub
```

The other reset functions are quite similar. To begin a simulation with empty epoch time values, the simulation pin has to be set to status high. This can be done with the function *SetLine(Port,Line)*. After that procedure, the CPLD starts simulating and providing epoch time values for the USB interface.

```
1  Private Sub SetSimulation()
2  'sets the Simulation Pin on the CPLD. This Subroutine
3  'uses the dll meMPIO1
4  'the Simulation Pin is on Port 3, Pin 4
5  'at the beginning, the Simulation Pin is reset, to make sure that it
6  'is low. After that it is set to high with SetLine.
7
8  With meMPIO1
9
10     .ResetLine 3, 4
11     .SetLine 3, 4
12
13 End With
14
15 End Sub
```

After turning the simulation line to high, the program has to check, if the data available line is high. With this signal, the CPLD shows if there is data in the CPLD respectively in the FIFO available or not available.

The next step is to scan the data available pin periodically, if it is high or low. This must be done in an endless loop, because this cycle must not be interrupted. In Microsoft Visual Basic 6.0 an endless loop is not very easy to realize, because it is a very intensive process and other processes on the PC get blocked. After a short period of time, the PC crashes down. To avoid this effect, Microsoft Visual Basic 6.0 provides a function called *DoEvents*. This function allows other processes running on the PC too. In other words, the other processes, which run on the system also get the CPU of the PC for a short time. This sounds quite reasonable, but after testing it, it becomes clear that the read out process becomes much too slow. For about 1 ms, the read out process stands still. The next idea was to call up *DoEvents* just every 1000-th loop cycle. With this interface, this procedure was not successful, because as already mentioned, the USB-PIO had a slow transfer rate. The self programmed function *isDataAvailable* checks the data available pin after the program was started. If the data available pin is high, the program realizes that the CPLD has any data in the FIFO and that they can be read out. After checking the status of the data available pin, two functions write on the GUI, if data is available or not available. When data is available, the function *GetDelayTime* returns the present delay time respectively the epoch time value.

```
1  Private Function GetDelayTime() As Double
2  'This function reads out data from the CPLD from Port 2.
3  'It returns the value as a double-value before the function
4  'is able to read out the data, the read pin (port 3, pin 3)
5  'is set to status high and then it is reset to low.
6  'to get the correct data, it is necessary to read
7  'out the least significant value and the most significant value
8  'first of all, the function reads out the least significant value
9  '(LSV) on Port 2. To get the most significant value,
10 'it is necessary to set the read-pin high. The function uses
11 'the dll meMPIO1 and the function GetPort(Port). This function returns
12 'a decimal value of the Port. The port is 8 bits large.
13 'First of all, we get the least significant value of port 2,
14 'which is 8 Bits long. After that, the function has to set
15 'the read pin to status high and reads out the most significant value.
16 'At least, the LSV and the MSV must be in the correct order.
17 'After that the function has to compute the Delay time.
18 'The Delay Time is the Epoch Time - multiplied with the factor
19 '5 *10^-9, because the CPLD counts with 5 nanosecond steps.
20 'That's the resolution of the counter. The Return-Value (DelayTime)is
       a
21 'Double Value!
22
23 Dim EpochTime_LSV_5ns As Integer 'variable that holds the least
       significant value of the Epoch Time
24 Dim EpochTime_MSV_5ns As Integer 'variable that holds the most
       significant value of the Epoch Time
25 Dim EpochTime As Integer 'variable that stores the calculated epoch
       time value consisting of the LSV and the MSV
26 Dim DelayTime As Double 'variable of the
27
28 GetStatusRead 'before it is possible to read the correct data of the
       port, the read pin (Port 3, Line 3) must be set
29              'high for a short periode of time
30
31 EpochTime_LSV_5ns = meMPIO1.Port(2) 'get the LSV of the actual Epoch
       Time and store it in the correct variable
32
33 GetStatusRead
34
35 EpochTime_MSV_5ns = meMPIO1.Port(2) 'get the MSV of the actual Epoch
       Time and store it in the correct variable
```

```
36
37  EpochTime = (EpochTime_LSV_5ns + EpochTime_MSV_5ns * 2 ^ 8)
38  DelayTime = EpochTime * (5 * 10 ^ -9)
39
40  GetDelayTime = DelayTime 'function returns the Delay Time
41
42  End Function
```

The previous page shows the source code of the main function, which is responsible for reading out the values from the CPLD. The only thing which is quite necessary to mention is that the function *Port(Port)* reads out the whole eight bit port and transforms the eight bit value into an eight bit decimal value.

The introduced USB-PIO is a simple interface, which can be programmed easily. The included functions in the *meMPIO1* dll provide easy handling. The programmer just needs to know how to deal with these functions and how to connect the USB-PIO to the CPLD and the PC. The main task is to study the programmers guide and to include the dll to the Microsoft Visual Basic 6.0 software. The provided digital functions are easy to handle and allow easy access for transferring and receiving data. The biggest drawback is the slow transfer rate. As already mentioned, the interface is useless for critical time operations. After programming the interface, the averaged transfer time for reading out an epoch time - consisting of four bytes, was about 25 ms. That is much too slow in relation of the time intervals of the epoch times. These values are stored in microsecond intervals in the FIFO. After a very short period of time, the FIFO gets overflowed and new epoch time values cannot be stored anymore. In other words, in most instances the time values get lost and decoding them into a correct and meaningful ASCII message is impossible. After discussing these facts respectively these drawbacks it becomes clear that the USB-PIO is not the right interface for this project and that it has to be replaced by another much faster interface. This primitive example presents one of the most important procedures before realizing any kind of hardware and software. A wrong decision can effect complications, which are not easy to handle without redesigning the project. Very often, the hardware components have to be changed and the software has to be reprogrammed. The wrong selection of the interface, cost a lot of time, but it was a very good practice for the future work. Also experimenting with more than one interface brings a not underestimated gain in experience. On the following page, a new interface with a very high transfer rate will be introduced. It is able to deal with the existing hardware and with the used programming language.

4.3 UM245R Interface

To avoid transfer speed problems, the UM245R module is used for this project. It is parallel FIFO interface (TTL) development module. The following characteristics and informations about this interface are retrieved from the datasheet [1].

- Single chip USB to parallel FIFO bidirectional data transfer interface.
- Entire USB protocol handled on the chip. No USB specific firmware programming required.
- Fully integrated 1024 bit EEPROM storing device descriptors and FIFO I/O configuration.
- Fully integrated USB termination resistors.
- Fully integrated clock generation with no external crystal required.
- Data transfer rates up to 1 Mbyte / second.
- 128 byte receive buffer and 256 byte transmit buffer utilizing buffer smoothing technology to allow for high data throughput.
- FTDI's royalty-free Virtual Com Port (VCP) and Direct (D2XX) drivers eliminate the requirement for USB driver development in most cases.
- Unique USB FTDIChip-ID feature.
- Configuable FIFO interface I/O pins.
- Synchronous and asynchronous bit bang interface options with RD# and WR# strobes.
- Device supplied pre-programmed with unique USB serial number.
- Supports bus powered, self powered and high-power bus powered USB configurations.
- Integrated +3.3 V level converter for USB I/O.
- Integrated level converter on FIFO interface for interfacing to external logic running at between +1.8 V and +5 V.
- True 5 V / 3.3 V / 2.8 V / 1.8 V CMOS drive output and TTL input.
- Configurable I/O pin output drive strength.
- Integrated power-on-reset circuit.
- Fully integrated AVCC supply filtering - no external filtering required.
- +3.3 V (using external oscillator) to +5.25 V (using internal oscillator) Single Supply Operation.
- Low operating and USB suspend current.
- Low USB bandwidth consumption.
- UHCI/OHCI/EHCI host controller compatible.
- USB 2.0 Full Speed compatible.

- -40° C to 85° C extended operating temperature range.
- Available in compact Pb-free 28 Pin SSOP and QFN-32 packages (both RoHS compliant).

The range of applications of this interface is miscellaneous. One basic reason for that is the high transfer rate. As already above mentioned, the transfer rate is specified with up to 1 MByte / second. Some typical applications of the UM245R, which contains the FT245RL chip. The list is not completed. For further possible applications, the data sheet has to be read.

- PDA to USB interface
- USB Smart Card Readers
- USB MP3 Player Interface
- USB Digital Camera Interface
- USB Wireless Modems
- USB Hardware Modems
- Interfacing MCU / PLD / CPLD based designs to USB

Driver Support

To use the UM245RL interface, several drivers for all major operating systems are available. These drivers are contained on the provided CD-ROM.

Virtual Com Port (VCP) Drivers for...

- Windows 7 32, 64-bit
- Windows XP and XP 64-bit
- Windows XP Embedded
- Windows 98, 98SE, ME, 2000, Server 2003, XP and Server 2008
- Windows CE 4.2, 5.0 and 6.0
- Mac OS 8/9, OS-X, Linux 2.4 and greater

Royalty free D2XX Direct Drivers (USB Drivers + DLL S/W Interface)

- Windows 7 32 and 64-bit
- Windows XP and XP 64-bit
- Windows Vista and Vista 64-bit
- Windows XP Embedded
- Windows 98, 98SE, ME, 2000, Server 2003, XP and Server 2008
- Windows CE 4.2, 5.0 and 6.0, Linux 2.4 and greater

For programming this interface, a driver is needed. First of all, there are two possibilities to get access to this interface with a programming language like Visual Basic. These drivers can be downloaded for free from the FTDI website. The FT245R is fully compliant with the USB 2.0 specification. In this project, a D2XX Direct Driver is used, because it is much easier to handle than the VCP Driver and the transfer rate becomes much higher. This fact will be discussed a little bit later in this Master Thesis.

4.4 FT245R Block Diagram

Figure 4.3 shows the internal hardware architecture of the FT245R chip, which is implemented on the used UM245RL interface.

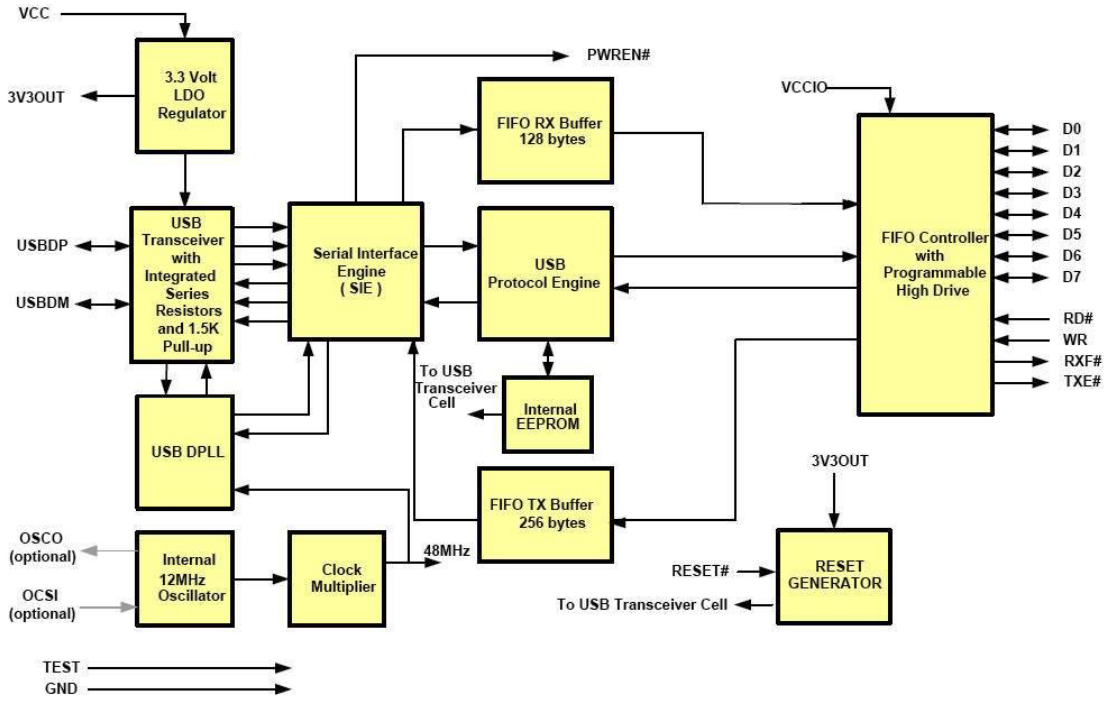

Figure 4.3: Block diagram of FT245R [1]

Functional Block Descriptions

The following itemization explains every single block of figure 4.3 [1].

- **Internal EEPROM:** The internal EEPROM in the FT245R is responsible for storing the USB Vendor ID (VID), the Product ID (PID), the device serial number, the product description string and various other USB configuration descriptors. In this internal

EEPROM, a special user area is defined for system designers to allow storing additional data. This can be done over USB and without any additional voltage requirement.

- **+3.3 V LDO Regulator:** The +3.3 V LDO regulator generates the 3.3 V reference voltage for driving the USB transceiver cell output buffers. The main function of the LDO is to power the USB Transceiver and the Reset Generator Cells rather than to power external logic.
- **USB Transceiver:** The USB Transceiver provides the USB 1.1 / USB 2.0 full-speed physical interface to the USB cable. The output drivers provide +3.3 V level slew rate control signaling, whilst a differential input receiver and two single ended input receiver provide USB data in, Single-Ended-0 (SE0) and USB reset detection conditions respectfully.
- **USB DPLL:** The USB DPLL cell locks on to the incoming NRZI USB data and generates recovered clock and data signals for the Serial Interface Engine (SIE) block.
- **Internal 12 MHz Oscillator:** The internal 12 MHz Oscillator generates a 12 MHz reference clock. This provides an input to the x4 Clock Multiplier function. It is also used as the reference clock for the Serial Interface Engine, USB Protocol Engine and FIFO controller blocks.
- **Clock Multiplier / Divider:** The Clock Multiplier / Divider takes the 12 MHz input from the Internal Oscillator function and generates the 48 MHz. The 48 MHz clock reference is used by the USB DPLL and the Baud Rate Generator blocks.
- **Serial Interface Engine (SIE):** The Serial Interface (SIE) block performs the parallel to serial and serial to parallel conversion of the USB data. It also checks the CRC on the USB data stream.
- **USB Protocol Engine:** The USB Protocol Engine manages the data stream from the device USB control endpoint. It handles the low level USB protocol requests generated by the USB host controller and the commands for controlling the functional parameters of the FIFO.
- **FIFO RX Buffer (128 bytes):** Data sent from the USB host controller to the FIFO via the USB data OUT endpoint is stored in the FIFO RX (receive) buffer and is removed from the buffer by reading the contents of the FIFO using the RD# pin. (Rx relative to the USB interface).
- **FIFO TX Buffer (256 bytes):** Data written into the FIFO using the WR pin is stored in the FIFO TX (transmit) Buffer.

- **FIFO Controller with Programmable High Drive:** The FIFO Controller handles the transfer of data between the FIFO RX, the FIFO TX buffers and the external FIFO interface pins (DO - D7).
- **RESET Generator:** The integrated Reset Generator Cell provides a reliable power-on reset to the device internal circuity at power up. The RESET# input pin allows an external device to reset the FT245R.

Inputs and Outputs

In the following tables, the input and output pins are explained. The pins can be separated into five main groups.

Pin Number	Name	Type	Description
1	D0	I/0	FIFO Data Bus Bit 0
2	D4	I/O	FIFO Data Bus Bit 4
3	D2	I/0	FIFO Data Bus Bit 2
5	D1	I/0	FIFO Data Bus Bit 1
6	D7	I/0	FIFO Data Bus Bit 7
9	D5	I/0	FIFO Data Bus Bit 5
10	D6	I/0	FIFO Data Bus Bit 6
11	D3	I/0	FIFO Data Bus Bit 3
12	PWREN#	Output	Goes low after the device is configured by USB, then high during USB suspend. It can be used to control power to external logic P-Channel logic level MOSFET switch. Enable the interface pull-down option when using the PWREN# pin in this way. Should be pulled to VCCIO with 10 kΩ resistor.
13	RD#	Input	Enables the current FIFO data byte on D0…D7 when low. Fetched the next FIFO data byte (if available) from the receive FIFO buffer when RD# goes from high to low.
14	WR	Input	Writes the data byte on the D0…D7 pins into the transmit FIFO buffer when WR goes from high to low.
22	TXE#	Output	When it is in status high, it is not possible to write any data into the FIFO. When it is low, data can be written into the FIFO by strobing WR high, then low. During reset this signal pin is tri-state.
23	RXF	Output	When it is high, it is not possible to read data from the FIFO.

Table 4.2: FIFO Interface Group [1]

Pin Number	Name	Type	Description
15	USBDF	I/0	USB Data Signal Plus, incorporating internal series resistor and 1.5 kΩ pull up resistor to 3.3 V.
16	USBBDM	I/O	USB Data Signal Minus, incorporating internal series resistor.

Table 4.3: USB Interface Group [1]

Pin Number	Name	Type	Description
4	VCCIO	PWR	+1.8 V to +5.25 V supply to the FIFO Interface group pins (1…3, 5, 6, 9…14, 22, 23). This pin can be supplied with an external +1.8 V to +2.8 V supply in order to drive outputs at lower levels.
7, 18, 21	GND	PWR	Device ground supply pins
17	3 V 3 OUT	Output	+3.3 V output from integrated LDO regulator. The main use of this pin is to provide the internal +3.3 V supply to the USB transeiver cell and the internal 1.5 kΩ pull up resistor on USBDP. This pin can also be used to supply the VCCIO pin.
20	VCC	PWR	+3.3 V to +5.25 V supply to the device core.
25	AGND	PWR	Device analogue ground supply for internal clock multiplier

Table 4.4: Power and Ground Group [1]

Pin Number	Name	Type	Description
8, 24	NC	NC	No internal connection
19	RESET#	Input	Active low reset pin. This can be used by an external device to reset the FT245R. It can be also left unconnected, or pulled up to VCC.
26	TEST	Input	Puts the device into IC test mode. For normal operation, it has to be tied to GND. Otherwise the device will appear to fail.
27	OSCI	Input	Input 12 MHz Oscillator Cell. It can be left unconnected for normal operation.
28	OSCO	Output	Output from 12 MHz Oscillator Cell. It can be left unconnected for normal operation, if internal Oscillator is used.

Table 4.5: Miscellaneous Signal Group [1]

General Remarks

In this project the FT245RL is used. That is the newer version of the FT245BL, which was also used for some experiments e.g. for a simple loopback test. Incidentally, a loopback test could be useful to test the read and write process. For this purpose, two interfaces were used. One for the write process and the other one for read process, although it would also possible to use just one interface. A loopback test writes a simple string into the FIFO of the used CPLD by using the first interface and the second interface finally reads out the transmitted string from the FIFO of the CPLD. These two strings have to be equal. Otherwise the read or write process was not successful. The FT245BL was used for sending the string and the FT245RL for reading it. According to the fact that the final version has just to read data and the transfer rate is up to 1 Mbyte per second, only one interface is probably used.
The main difference between these two devices is that the FT245BL needs an external EEPROM and quartz. As already said, the FT245RL is the newer version and has an internal EEPROM and an internal quartz.

First Preparations

To program an UM245R interface for the read out process, one of the two introduced drivers in 4.3 is necessary to use. The VCP (Virtual Com Port) is much easier to handle but provides a lower transfer rate than the D2XX driver [1]. According to the programming guide, the FT245RL can be programmed like a RS232 interface. Microsoft Visual Basic 6.0 provides a possibility to interact with this type of interface, but it has not been used in this present Master Thesis.
In consideration of the fact that the transfer rate can be much higher, the D2XX driver were downloaded from the manufacturer website and afterwards installed on the PC. The driver can be integrated into many programming languages like C++. A very important fact is that this driver can also communicate with Microsoft Visual Basic 6.0.
After installing the D2XX driver (a short manual can be found on the manufacturer website), Visual Basic 6.0 can interact with it and with the FT245RL interface - assuming that the FT245RL is already connected with the USB interface from the PC. The access to the FT245RL is provided by a DLL, which can be used, after installing the D2XX Direct Driver. First of all, the functions of the DLL have to be declared at the beginning of the program. The use of this functions was not very easy, because the programmers guide just explains the use of these functions in C++.
In Visual Basic 6.0, the declaration of the provided functions is quite similar but not equal. The biggest difference is the read out process and the use of the read-out function, because the 4 Bytes are not stored into a character array like in C++, but they are stored into a string

and have to be extracted afterwards. This will be discussed a little bit later. The declarations and the use of the D2XX Direct Driver in Microsoft Visual Basic 6.0 were looked up in sample codes. Important to mention is that in the final Visual Basic program all possible functions of the D2XX dll are declared. Also functions, which are not used from the program. This method avoids any later problems.

4.5 Programming the FT245RL

After finishing the preparations in 4.4, the interface is ready to get programmed with Visual Basic 6.0. Before the beginning of the implementation, it is important to think about the functions, which the software has to perform. The following points describe the functionality in detail:

1. Declaration of the functions, which the dll provides for programming the interface. As already discussed, all known functions are declared to avoid any possible problems after using them.

2. Defining the constants for the return codes. Almost every single function returns an integer after the execution. A return code can be understood as "status answer" and shows if the operation of the respective function was successful.

3. Before any data can be read out, the interface has to be opened. The D2XX Direct Driver offers a special function for that. This principle is similar to the opening process of the USB-PIO, which was explained in section 4.2.

4. When the start button on the GUI is clicked, the program checks, how many devices are connected on the USB bus of the PC. After checking the number of connected interfaces, the program opens all connected interfaces. In this present case, only one interface can be found by the responsible function.

5. After opening the interface, the program is able to read / write from / to the interface. To avoid any possible inconsistency, all connected interfaces are described by an unique id, which is stored into a special variable after executing the responsible function. Next the program writes two characters respectively two bytes into the FIFO of the interface (in this case "HA") and waits 30 ms. After this time period, the interface gets reseted by the provided reset function.

6. When the program reaches this unit, the interface is still open and the algorithm can deal with it. An endless loop checks if the queue status is higher than zero. The function, which checks this status, stores the numbers of bytes of the FIFO into a parameter value. It is important to mention that the return value of the function only describes

the status of the execution of the function and has nothing to do with the number of bytes in the FIFO. If the number of bytes is higher or equal to 1024, the read-function reads out the whole bytes and stores it into a string, which has to be separated into its single bytes. If the program is closed, the algorithm jumps to the label *CloseHandle*.

This is just a very simplified description of the implemented algorithm, but it can be helpful to get a better overview for the main components of the source. These components respectively the main parts of the source code need to have a more detailed view in the next section.

Provided Functions

Before the different parts of the source code will be discussed in detail, it is necessary to explain the main characteristics of the used functions. As already announced, the functions of the D2XX dll are not similar to any other "normal" functions in Visual Basic 6.0. The main difference can be found on the parameters of the functions and on the return values. The return value of a D2XX function is a kind of status message and describes, if the function was successful with its operation or not. There are quiet a few different status messages, which can be seen in B.3. If the return value of a function is equal to zero (*FT_OK*), the function did not have any problems and can express its results by its parameter values. How this works will be explained in the following example.

```
1 'Open device FT245RL with Open_By_Description
2 If FT_OpenEx(strDescription, FT_OPEN_BY_DESCRIPTION, LngHandle) <>
     FT_OK Then
3     ErrorCode = FT_OpenEx(0, FT_OPEN_BY_DESCRIPTION, LngHandle)
4     LoggerList.AddItem ("Open_EX failed")
5     LoggerList.AddItem ("ErrorCode: " & ErrorCode)
6     Exit Sub
7 Else
8     LoggerList.AddItem ("FT_OpenEx Successful, FT245RL is open now :-)
          ")
9 End If
```

This section of the source code written in Visual Basic 6.0 opens a FT245RL interface with the "Open By Description" method.The function on line 2 has the following structure:

```
Private Declare Function FT_OpenEx Lib "FTD2XX.DLL" (ByVal arg1 As
    String, ByVal arg2 As Long, ByRef LngHandle As Long) As Long
```

In the following, the most important function parameters are explained in detail.

- **arg1:** Stands for the description of the FT245RL. Every single FT245RL has its own unique description string, which is created by the function *FT_ListDevices*. Without

executing *FT_ListDevices* it is not possible to open a FT245RL with a description. However it would also be possible to open the FT245RL by serial number. In this case, the serial number has to be entered, which would be also a string and not a long integer.

- **arg2:** Is a flag and describes how the interface has to be opened. There are two different opening modes. The first one is *FT_OPEN_BY_SERIAL_NUMBER* and the second one is *FT_OPEN_BY_DESCRIPTION*. These constant are defined at the beginning of the source code.
- **LngHandle:** Is also a unique number and describes one single FT245RL. The function stores this number into this parameter. Before the function is executed, this parameter value is still empty respectively equal to zero. After a successful execution (return value is equal to zero), the parameter value *LngHandle* can be read out. This unique number is very important for writing and reading data from the FT245RL and has to be passed as function parameter to every function. Without this number, the function would not know, which FT245RL the programmer means. In other words, *LngHandle* is like a serial identification number.

Back to the source in 4.5. The most essential details can be seen on line 2 and line 3. In these lines, the error code respectively the return value is sampled by the if-condition. As already mentioned, the operation of a function was successful, if the return value is equal to *FT_OK* respectively equal to zero. All other possible return values describe an error e.g. if the FT245RL is still not connected with the USB bus of the PC. Finally the error code is displayed on the loggerlist (see line 4 and 5).
If everything was okay, the program jumps to line 8 and displays an acknowledgment that the opening operation was successful.

Read / Write Process

This simple source code in 4.5 shows the main structure of all contained functions in the D2XX dll. After the opening process, the interface can be used for receiving / writing data from / to the CPLD. As already explained at the beginning, the received data is packed into a string and can be extracted after the reading process was successful. The string, which includes the received data has a length of 63488 bytes. That is the maximum number of bytes, which the FT245RL is able to store in its integrated FIFO. Normally, the string is smaller than the maximum number of bytes after the read out process, because most of the time the FIFO is not full before it is read out by the readout function.
Reading out one byte from the FIFO with the integrated function of the D2XX Direct Driver is fast enough for this task. It can be done in the microseconds range. The programming

style is crucial for the transfer speed e.g. unnecessary loops or arrays waste lots of computing time. A previous version of the program contained lots of of these time consuming constructs and the effect was a very slow execution of the source code. After a very short period of time the queue status of the FIFO was 63488 bytes - that is the maximum number of storage positions of the FIFO. Further bytes got lost and could not get processed by the software.
Received epoch time value consists of four bytes. Therefore 63488 bytes describe 15872 values and that is also the maximum number of epoch time values, which the internal FIFO of the interface can store. The software only reads in a number of bytes, which can be divided by four without any division remainder to ensure that only integral epoch time values are computed. The following source code statement is responsible for determining the actual number of bytes, which can be read out from the FIFO.

```
BytesReadable = lngRxBytes - lngRxBytes Mod 4
```

lngRxBytes describes the number of all available bytes in the FIFO. As it is explained, one epoch time value contains of four bytes. Therefore there must not arise any division remainder after dividing the number of bytes by four. This source code below accomplishes this operation by calculating the remainder of the division with four. The next step is subtracting the division remainder from the total number of bytes and the final result is stored into the variable *BytesReadable*.
In most cases, the variable *BytesReadable* contains the same number of bytes than *lngRxBytes*, but to be absolutely sure that everything gets all right, this part of the source code is indispensable. The next part of the algorithm is to read in this number of bytes from the FIFO. As it is explained in the previous section, the D2XX Direct Driver provides a function to read in a defined number of bytes.

```
FT_Read(LngHandle, strReadBuffer, BytesReadable, lngBytesRead)
```

The return value of this function is an error code. If it is zero, the execution of the function was successful. All other integer values describe an error during execution. Next, the parameter values of the readin function are explained.

- **LngHandle:** Unique number for triggering the requested FT245RL.
- **strReadBuffer:** Is a string with a length of 63488 and contains the read bytes. The bytes are stored in ASCII characters and they have to be converted into their ASCII values. This procedure will be explained later. Before the function is executed, *strReadBuffer* is empty.
- **BytesReadable:** Presents the number of the requested bytes, which have to be read by the function.

- **lngBytesRead:** Describes the number of bytes, which were definitely read by the function. In the normal case, this number is equivalent with the number of *LngHandle*. If the value of *lngBytesRead* is lower than the value of *BytesReadable*, the function did not read all requested bytes.

4.6 Computing the Epoch Times

The next step of this algorithm is to extract the received bytes out of the *strReadBuffer* string. First of all, it is important to know that one epoch time consists of four bytes. In the string variable, the maximum number of characters is 63488. This is also the maximum number of values in the FIFO as already mentioned before.
The for-loop is needed to analyze every single character of the string. Four characters in the string stand for one epoch time value. The highest value, which one byte is able to describe is 255.
The for-loop is not like a common for-loop with step 1, but a for-loop with step 4. This means that only the first byte of an epoch time is reached by the for-loop. The counter of the for-loop finally gets increased for three times after a loop cycle, to get the remaining three bytes of a single epoch time value. After that procedure, the epoch times are computed with their four bytes. Important is that the first, the fifth, the ninth, the thirteenth and all the rest of it describe the first byte of an epoch time in each case. The byte string can hold maximum 63488 bytes. But most of the time there are less bytes stored in the string. In other words, the program has to realize, that it reached the end of the string. Normally, the end of the string is reached, when the software reads zero on a string-position. That would be possible to check at each step. But there is a better way to find out the "real" end of a string. The source code fragment in 4.5 computes the number of bytes, which the read-function has to read out of the FIFO. The length of the string is described by the value of *BytesReadable*. The maximum value of *BytesReadable* is again 63488. Due to this conclusion, the program does not need to check every single byte, if it is a value of an epoch time or if it describes the end of the string. In other words, the for-loop runs until the first byte of the last epoch time value is reached.

```
1 For Counter = 1 To BytesReadable - 3 Step 4
2     Byte_0 = Asc(Mid$(strReadBuffer, Counter, 1))
3     Byte_1 = Asc(Mid$(strReadBuffer, Counter + 1, 1))
4     Byte_2 = Asc(Mid$(strReadBuffer, Counter + 2, 1))
5     Byte_3 = Asc(Mid$(strReadBuffer, Counter + 3, 1))
6
```

```
7     EpochTime = CDec(Byte_0) + CDec(Byte_1 * 2 ^ 8) + CDec(Byte_2 * 2 ^
          16) + CDec(Byte_3 * 2 ^ 24)
8     Nanoseconds = CDec(EpochTime) * 5
9
10    'RelevantValue = RelevantValue Mod 100000
11    RelevantValue = Nanoseconds - Int(Nanoseconds / 100000) * 100000
12
13    List5.AddItem Nanoseconds 'Display the NanoSeconds on List5
14    List5.ListIndex = List5.ListCount - 1
15    List5.Selected(List5.ListIndex) = False
16
17    Index = Index + 1 'Increase the Index Value
18    InformationArray(Index) = RelevantValue
19 Next Counter
```

The for-loop can be seen on line 1. It analyses the string with step 4. The second line until the fifth line separates the sting in its bytes. Important is that at first the least significant byte of the present epoch time value is reached. This mentioned least significant byte is computed in the second row. The internal function *Mid$* is able to separate this character respectively the byte from the string. If the counter variable of the for-loop is next increased, the function *Mid$* is able to analyze the second significant byte of the present epoch time. This can be seen in the third line of the code fragment. The most significant byte is finally determined in the fifth line. As it is known, a string in Visual Basic always consists of characters. One character presents a byte and a byte can describe at maximum a value of 255. To get a integer value from a character, the character has to be transformed in its equivalent number. This manages the internal function *Asc*. On line 7, the epoch time value is calculated with the four bytes of the string. The used function *CDec* converts every single byte in its decimal value - a base 10 floating point number.

To get the correct result, the epoch time is multiplied by five because the resolution of counter is 5 ns. This means, that the counter counts in 5 ns steps. The basic grid of the epoch time values is 500 μs. To simplify the next operations e.g. determining the zero value, the epoch time values are truncated after the 500 μs position on line 11. For example, if there is an epoch time value of 156456788 ns, the value is converted into 56788. The advantage of this method is that the basic grid does not have to be considered anymore. This “cut-off method” is just a simple modulo operation as it is explained by the comment on line 10. For determining the ASCII values only the last fourth positions of the epoch time values are essential. These important positions represent the offset, the ASCII value and at least the jitter. In other words, the value *RelevantValue* does not represent the 500 μs any more. That is a very big advantage for calculating the final character, because the counter of the CPLD

gets overflowed at a defined value.
After this operation, this value is stored into a array called *InformationArray*. This array can store at least $\frac{63488}{4} \Rightarrow 15872$ values. All other not explained lines in the source code fragment are unimportant e.g. the lines 13 to 15, which write the calculated values into a list of the GUI. After a loop cycle, the counter variable *Counter* gets increased. This is shown on line 17. This procedure is repeated until the end of the string is reached.

Determining the Zero Value

In section 3.1 it is mentioned, that for determining the characters a reference value or a zero value is needed. This determination is not new, because it was already used in the source code 3.1.5. It is important to remember that the reference value has to be subtracted from the coded epoch time value. The determination of this important value is not easy to find, because there must be 100 equal values for defining the zero value. After 100 values with an interval of 500 μs (in due consideration of possible jitter) were detected, the algorithm starts with determining the deviation of the epoch times from the 500 μs grid. This is necessary to provide tight tolerances. It is considered that this method does not allow sending 100 same characters. That would cause some difficulties by determining of the reference value. As it is already mentioned, the epoch time values are afflicted with jitter in a range of $\pm$7 ns. This also applies for the zero values of course and that is the reason why 100 zero values are stored into an array and finally sum up to determine the deviation of the zero values from the 500 μs grid. As previously mentioned only the last fourth positions of the epoch times are important and after eliminating the irrelevant positions, the basic grid does not have to be considered by determining the deviation. Another method for computing the reference value would be to use regression lines. The advantage of this method is that uncertainties like jitter could be nearly eliminated. For this project, the implementation of this regression algorithm was not necessary, because the effects of uncertainties became insignificant low. Assuming that e.g. only 30 equal values were found, all elements of this array would get erased and the algorithm starts again searching for 100 equal values and storing them into the zero value array. The determination of the reference value has to be done after every reading process and the old reference value of the last reading cycle becomes invalid. In other words, after storing the new epoch time values in the *InformationArray*, the determination of the reference value begins afresh to provide a better accuracy. The laser system has to provide zero values when no characters are transmitted. The worst case would be that there are not 100 equal values stored in the array. In this case, the algorithm uses the old reference value of the previous cycle. The old reference value of the previous cycle is always stored in a variable. After a new reference value is found, the algorithm overwrites this value with the new reference value at the end of the present program cycle. If the algorithm does not find

any equal values, the reference value remains zero and extracting messages are not possible. After the reference value is found, the algorithm starts to determine the ASCII characters by retrieving the epoch times, which are stored in the *InformationArray*. This determination happens in the same way like in the previous source code e.g. in 3.1.5. The zero values in the *InformationArray* are also considered on the determination of the ASCII characters, although they do not represent any characters. They are not erased from the array. This method does not critically increase the computation time and does not effect the final results, because the difference between the reference value and a zero value is accordingly low. The complete source code of this project can be read in the appendix.

4.7 Graphical User Interface of the final Detection Software

The GUI is very simple and offers just the most important options. As already mentioned, it contains four text fields. The information field, the readed bytes field, the determined null value (zero value) field and the main field, which contains the received message after the execution. A detailed description will follow after figure 4.4.

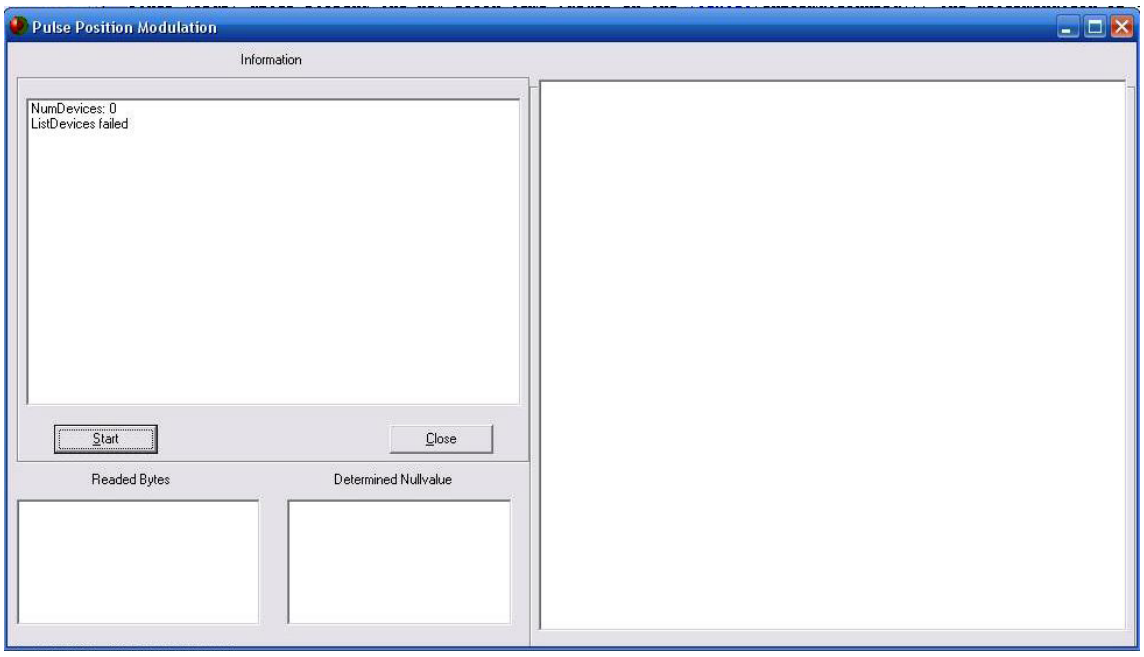

Figure 4.4: GUI of the final software

There are just two buttons. The button “Start” opens the connection for the UM245R interface and requests if the FIFO contains any data respectively bytes. If there is no data available - e.g. if the CPLD is not switched on, the program stays in this state as long as the

number of bytes in the FIFO is zero.
The number of bytes in the internal FIFO of the interface has to be greater or equal to 4096. Only then, the algorith begins to read out bytes from the USB interface. This minimum number of bytes can be changed very easily with no time exposure. This is important, if files with less than 1024 chars are sent.
One other possibility could be that shorter files with less than 1024 chars are sent for more than one time. In this case, the limit of 4096 bytes would exceed and the algorithm would start determining the correct ASCII values. Here, no changing of the minimum number of bytes would be necessary. One other special case would be if a file with a length of 5000 bytes respectively 1250 chars is transmitted. In due consideration that a 2 kHz laser (2 bytes / s) is used, the total number of transmitted bytes is always more than 5000 bytes. The owerflow of bytes are zero values, which are used for determining the reference value.
Important to mention is, that the minimum number of bytes cannot be arbitrary low e.g. 4 bytes, because the algorithm determines the reference value after every readout cycle. For determining this reference value, there must be at least 100 epoch time values stored in the array - the algorithm has to find at least 100 zero values.
Selecting 400 bytes as lowest limit might cause problems, because there would be just 100 epoch time values in the array and it is not guaranteed that these time values are zero values. Assuming that these values describe no ASCII characters, but zero values, the algorithm could determine the correct reference value. Important is that the algorithm finds a reference value after the first reading cycle. If not, data loss would prevent a correct interpretation of the transmitted file. If no reference value is found in the next cycles, the algorithm takes the old reference value from the previous reading cycles. So it does not cause any problems if there do not exist 100 zero values in the next cycles.
The conclusion of this fact is, that using a limit of 2048 bytes would be a good choice. Remembering that 2048 bytes can describe 512 characters - the probability that the array contains 100 zero values would be high.
After retrieving the bytes from the FIFO of the interface, the algorithm continues by calculating the epoch time values based on four bytes for each epoch time. As already explained, on line 7 of source 4.6, the epoch time gets calculated and stored in a variable called *EpochTime*. After reading out all values, the algorithm continues with determining the zero value. These steps have already been explained in section 4.6.
The button “Close” just closes the present connection between the interface and the USB bus of the PC.
The text field on the left side is a information field, which can contain some important informations about the connection e.g. how many bytes are ready for reading out from the FIFO of the interface or how many interfaces are connected on the USB bus. These status

values are updated from different functions provided from the D2XX dll. As it is seen, figure 4.4 was currently created, when there was no interface connected to the USB bus of the PC. Otherwise the text box would show the explained informations. This text box is just for fixing bugs in the program source.
On the right side there is a textbox *Determined NullValue*. This field displays the present zero value, which is used for determining the ASCII characters. This text field has been most important for debugging, because determining the zero value is the most critical part of this software.
The message field is for displaying messages, determined from the epoch time values. This is almost the same text field as it can be seen on figure 3.4 on the right side.

4.8 Final Discussions of chapter 4

This detection software was successfully tested at the 2 kHz Satellite Laser Ranging station in Graz / Austria. Important to mention is, that the offset of 40 ns, discussed in 2 has to be changed up to 80 ns. On the hardware side, changing the offset was more difficult to realize e.g. reprogramming the FPGA on the ISA card, shown in 4.1. By using an offset of 40 ns the transmission tests were not successful. Most of the ASCII characters were not recognized correctly. One basic cause for this problem is the uncertainty respectively the jitter, caused by hardware delays.
The maximum tolerable value when using 40 ns offset is in the range of 35 ns. If the jitter is outside this range, a correct interpretation of transmitted messages cannot be guaranteed. The most common errors of not using a suitable offset are rounding failures. As already mentioned, the program uses a rounding procedure, because for an assignment to an ASCII character, the determined ASCII value has to be a positive integer.
If the offset is increased up to 80 ns, the highest tolerable offset is in the range of 75 ns. With this modification, the transmitted messages were extracted without any failures. In the next chapter, some results of these measurements will be shown.

5 Detection Hardware

In this chapter, the most important component of this project will be introduced. It is about a Multi-Pixel Photon Counter, which is manufactured by Hamamatsu. The standard detector for SLR is a SPAD - a Single-Photon-Avalanche-Diode. This is an excellent device for SLR, but it is not suitable as a PPM detector. It is single-photon sensitive, thus reacting on any arriving background photon. At 2 kHz it produces a dark noise of about 400 kHz - all that ends up in a lot of noise points, prohibiting its use as a PPM detector.
Instead, a Hamamatsu Multi-Pixel Photon Counter Module (MPPC; C10507-11-050U) is used. The MPPC is essentially an opto-semiconductor device with photon-counting capability and which also possesses great advantages such as low voltage operation and insensitivity to magnetic fields [2]. Each APD pixel of the MPPC outputs a pulse signal when it detects one photon. Finally the signal output from the MPPC is the total sum of the outputs from all APD pixels operating in Geiger mode. The MPPC offers the high performance needed in photon counting and is used in diverse applications for detecting extremely weak light at the photon counting level [2]. It has many features like high gain (10^5 to 10^6), low dark count rate ($<$ 1 MHz with 0.5 p.e. threshold level), low bias voltage operation ($<$ 100 V), insensitive to magnetic fields, room temperature operation, high Photon Detection Efficiency (PDE), high time resolution, low power consumption and mechanical robustness [21].

5.1 Single Photon Avalanche Diodes

Single photon avalanche diodes (SPADs) are p-n junctions reverse-biased above breakdown voltage (V_{bd}) for single photon detection. When a diode is based above V_{bd}, it reamins in a zero current state for a certain period of time - in this present case in the microseconds range. During this time, a very high electric field exists within the p-n junction generating the multiplication region [14]. Under these conditions, if a primary carrier enters the multiplication region and triggers an avalanche, several hundreds of thousands of secondary electron-hole pairs are generated by impact ionization, thus causing the diode's depletion capacitance to be rapidly discharged [14]. Thus, a sharp current pulse is generated and can be easily measured [14]. This mode is commonly known as Geiger mode.
In cases where the area of the APD or the generation volume of the depleted space charge region of the p-n junction is large, the thermal (dark) generation current results in individual breakdown events and creates dark counts [19].

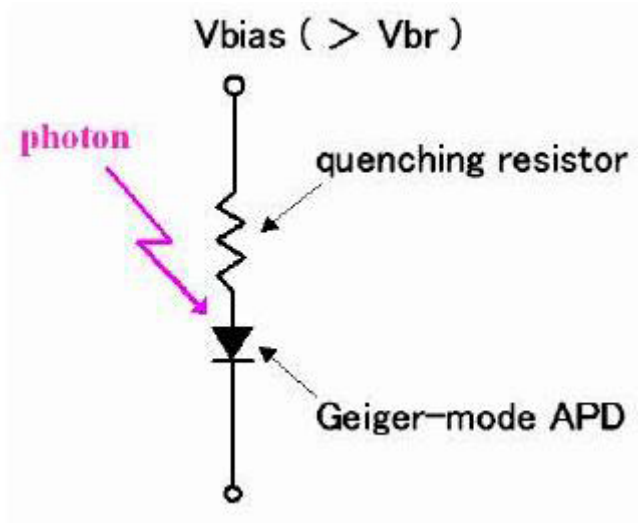

Figure 5.1: SPAD in Geiger mode [22]

Figure 5.1 on the left shows a single photon avalanche potodiode in Geiger mode. This mode can be achieved by biasing the reverse voltage larger to an APD than the breakdown voltage (V_{br}). The operation voltage V_{op} can either be called bias voltage. The gain becomes extremely high in Geiger mode in comparison to linear mode. This high gain is responsible for a large measurable output pulse. On the other hand, the thermal excitation noise grows as a result of high gain to 1 photon equivalent (p.e.) pulse size [22]. The output pulse charge can be determined by equation 5.1.

$$Q = C(V_{op} - V_{br}) \tag{5.1}$$

Q = output pulse charge

C = capacitance of APD pixel

V_{op} = operation voltage

V_{br} = breakdown voltage

The three modes of the basic operation of a SPAD are charge, discharge and quenching [22]. The Geiger mode APD can be assumed as capacitance for an easier explanation of figure 5.2.

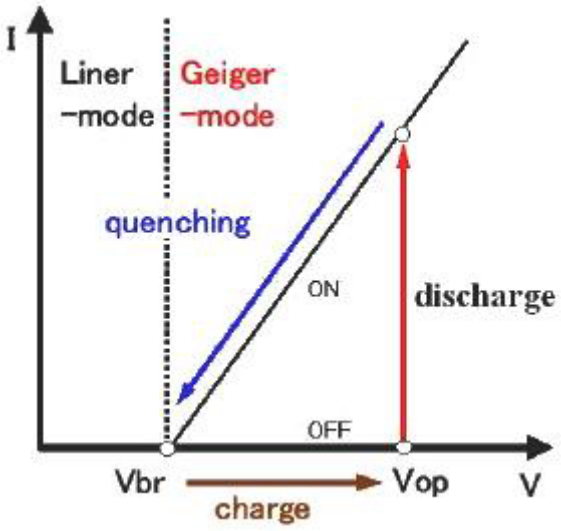

Figure 5.2: Basic Operation of a Geiger mode APD [22]

The APD respectively the capacitance is charged by the operation voltage V_{op} and stays in this state until an incidence of photon occurs. This photon incidence generates the avalanche process in the Geiger mode APD and the current will begin to flow or to discharge [22]. The recharge of an APD can be limited by a quenching resistor. In other words, decreasing Q also decreases $V_{op} - V_{br}$ and the avalanche process will stop, but recharge through the quenching resistor will continue after the quenching process [22].

The second possible mode for SPADs is the linear mode. In this mode, the reverse bias applied to the APD is held constant, and the primary photocurrent generated in the APD's absorber is amplified by a proportional multiplication factor that is independent of signal strength (below saturation) [8]. The output of a linear APD is proportional to the level of illumination it receives. The typical field of application for linear APDs is in optical receiver to boost weak signals above the noise floor of the receiver's amplifier [8]. The linear mode is insignificant for this project.

Also the precise arrival time of individual photons can be detected. These devices are used in diverse applications e.g. for pulse position modulation or time correlated single photon counting [7]. This feature is used in this present project. The main problem in this case is the uncertainty or rather the low time resolution known as jitter. This applications require high timing resolutions. The lowest reported SPAD jitter at room temperature is 28-ps full-width at half-maximum (FWHM) using a custom double-epitaxial-layer structure [4].

Jitter optimized SPADs are a field of research in many research papers. Many substantial progresses have been reached during the studies. Using an area-efficient SiO2 shallow-trench-isolation (STI) guard ring can improve the timing resolution [7].

In a well designed system, the electronic jitter can be reduced below 10 ps FWHM, therefore, the ultimate timing performance is limited by the detector itself [18]. Time resolution problems were eliminated by programming roundings and ranges in this project. Above mentioned research results should show that time resolution problems by using SPADs for using the pulse position modulation are familiar. After connecting a quenching resistor to a

Geiger mode APD, the circuit outputs a pulse at a constant level when a photon is detected. When a diode is biased above breakdown voltage, it remains in a zero current state for a relatively long period of time, usually in the microsecond range. During this time, a high electric field exists within the p-n junction generating the avalanche multiplication region [14]. The SPAD can remain in Geiger mode until a free carrier is generated in or enters into the depletion region, is accelerated by the electric field and acquires sufficient energy and enough subsequent impact ionizations to result in a self-sustaining breakdown of the diode [10].
A single photon is able trigger an easy measurable avalanche charge. The electronic of the detector is responsible for avoiding damages on the diode and to reset the current after triggering. Thermal effects can also cause an avalanche charge. That is one of the main problems by using SPADs. To be used as SPAD, a diode needs to have a structure that fulfills some basic requirements, see below [15]:

1. The breakdown must be uniform over the whole active area to produce a standard macroscopic pulse.
2. The dark counting rate must be sufficiently low.
3. The probability to generate afterpulses should be low.

To characterize a SPAD device, it is essential to estimate some basic facts e.g. dark counting rate (thermal and afterpulsing components), photon detection efficiency, time resolution, maximum excess bias voltage, optimal working temperature, etc.
Modern technology allows the production of SPAD detectors with an integrate quenching mechanism based on a Metal-Resistor-Semiconductor structure [15].
In the next section, some unwanted effects by using SPADs for photon detection will be presented.

5.1.1 Dark Count

Carriers generated by detected photons produce an output pulse. So called dark current carriers, generated by high temperature or emitted by trapping levels in the semiconductor, can also trigger output pulses [18].If the detector is held in the dark, there is an avalanche triggering rate that is called the dark-counting rate [18]. This rate reduces the sensitivity of the detector, because dark counts compete with photons in triggering the detector.Dark Count is usually expressed in electrons per unit of time at a given temperature.
The used MPPC, which is explained in 5.2 is a solid-state device and generates noise due to termal exitation. The noise component is amplified in Geiger mode operation and the original photon detection signal cannot be discriminated from the noise. This noise occures randomly and due this fact its frequency (dark count) is a crucial parameter in determining

MPPC device characteristics.
A selected threshold level of 0.5 p.e. is defined as the dark count (number of times that one or more photons are detected). The dark count in the MPPC is output as a pulse of the 1 p.e. level. It is difficult to discern a dark count from the output obtained when one photon is detected. According to the datasheet it is unlikely that dark counts at 2 p.e., 3 p.e. or 4 p.e. level are detected. To reduce dark counts, different technological recipes have been developed to lower defect densities, and available silicon devices have dark count rates from 2 x 10^{-3} to 10^3 s^{-1} per μm^3 of active volume at room temperature [18].

5.1.2 Afterpulse and Crosstalk

Afterpulses are spurious pulses, which follow the true signal - caused by cystal defects, which trap the generated carriers and then release them at a certain time delay. Afterpulses cause detection errors. The lower the temperature, the higher the probability that carriers may be trapped by crystral defects and afterpulses will increase [2].
In an avalanche multiplication process, photons might be generated which are different from photons initially incident on an APD pixel. In case that these generated photons are detected by other APD pixels, the MPPC output shows a value higher than the number of photons that were actually input and detected by the MPPC. This phenomenon is thought to be one of the causes of crosstalk in the MPPC [2].

5.2 Multi-Pixel Photon Counting Module

For normal Satellite Laser Ranging (SLR), the Graz 2 kHz SLR system uses a Single-Photon-Avalanche-Diode (SPAD) to detect single and multiple photons, as they are returning from the satellite. For the pulse position modulation, a SPAD is not suitable for detecting the modulated laser pulses, because the main drawback of its high sensitivity is the inherent noise of the detector. It produces more than 400 kHz of dark noise, when it is gated with 2 kHz [11].
For routine SLR, this mentioned noise is not a major problem. It handles several MHz of daylight background noise anyway, but for proposed data transmission scheme it is more or less prohibitive, because there is no simple way to discriminate arbitrary noise points from pulse position modulated data [11].
A single APD, which is operating in Geiger mode cannot distinguish between a single photon and multiple photons that arrive simultaneously. To avoid the problem with noise, a different detector is used for this pulse position modulation project. It is a so called Multi-Pixel-Photon-Counting Module (MPPC: Hamamatsu C10507-11-050U). This MPPC consists of a 20 x 20 array of single SPAD units on the chip. It is a so called Si-PM (Silicon Photomultiplier)

device. Althouh it produces even more dark noise (up to 800 kHz) than our SLR SPAD, the resulting dark noise pulse amplitudes are according to single photons only; however, if a 10 ps laser pulse from the SLR laser arrives, it triggers many or all of the 400 SPAD elements simultaneously; the combined outputs are superimposed in a much higher analogue output pulse. Thus, although such a MPPC remains basically single-photon sensitive, it can easily discriminate laser pulses against very high dark noise and against significant background noise.

The leading edge of the MPPC analogue output is discriminated by a fast comparator with adjustable trigger level; the TTL output of this comparator is connected to the CPLD. A simple 5 ns counter in this CPLD time tags the events. The 5 ns resolution of this counter is sufficient to resolve the number of 80 ns multiples. The PC reads these event times, and decodes the corresponding ASCII values.

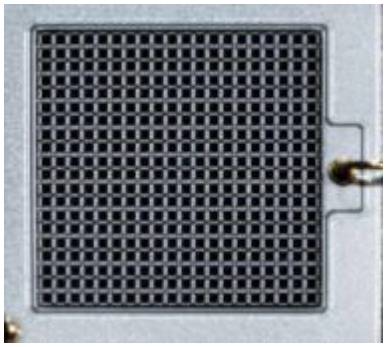

Figure 5.3: 20 x 20 array of single SPADs

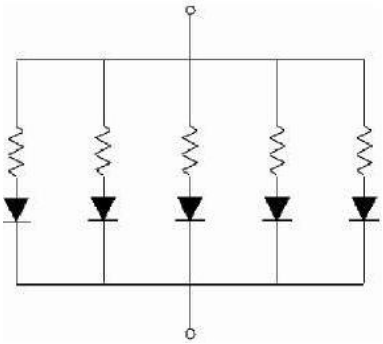

Figure 5.4: Equivalent circuit diagram of a MPPC

Each SPAD in figure 5.3 measures 50 x 50 μm. All of them deliver their current into a common output line, where the total current of all contributing SPADs is summarized - assumed that they are triggered by photons. The total current of all contributing SPADs is summarized in figure 5.6. While the dark noise or the daylight background noise of each single SPAD on the MPPC chip mainly appears as uncorrelated 1 p.e. (photon electron) pulses, the relatively strong laser pulses produce much higher signals, which are easily detected and discriminated with a simple analogue comparator, giving a TTL compatible pulse for any detected laser pulse, but suppressing all noise pulses [11].

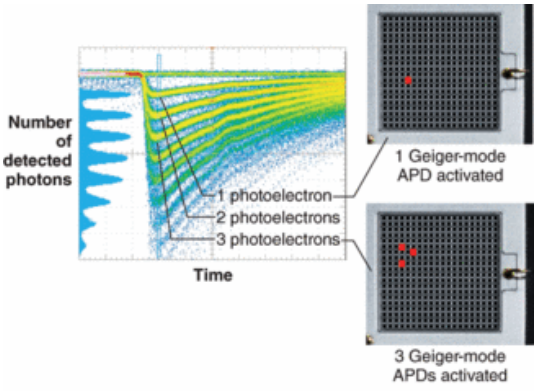

Figure 5.5: Output current of triggered SPADs

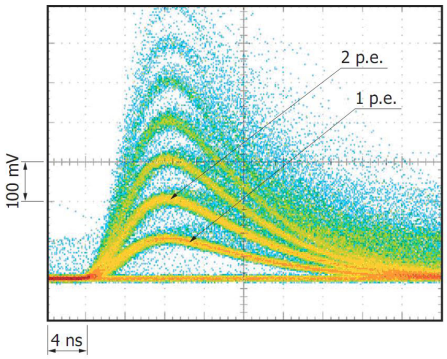

Figure 5.6: Accumulated output of Hamamatsu MPPC

Each pixel, which detects a photon provides an output current. The total sum of all outputs forms the MPPC output (see figure 5.5). The marked pixels represent the discharge of the pixel when an incident photon is detected. In the top array, only one photon is detected and the result is one photoelectron output on the left. In the lower array, three incident photons are simultaneously detected and the resulting output pulse has a three times higher amplitude. This technique allows the counting of single photons or the detection of pulses of multiple photons. When photon flux is low and photons arrive at a time interval that is longer than the recovery time of a pixel, the MPPC will output pulses that equate to a single photoelectron. These pulses can be converted to digital pulses and counted [13].
When the photon flux is high or the photons arrive in short pulses (pulse width less than the recovery time), the pixel outputs will add up as it is shown in figure 5.5. To determine the incident number of photons per pulse is not possible with SPADs, described in section 5.1.
One more advantage of using a MPPC is the high gain and a low noise, caused by the multiplication process. The high gain produces a measurable output signal. Unfortunately the gain is temperature dependent. If the temperature rises, the lattice vibrations in the crystal become stronger [2]. This increases the probability that carriers may strike the crystal before the accelerated carrier energy has become large enough, and make it difficult for ionization to occur. If the temperature becomes higher, the gain becomes lower at a fixed reverse voltage. To avoid any troubles, it is essential to vary the reverse voltage according to the temperature or to keep the temperature of the device constant. For this project, the temperature of the device is not considered.

5.2.1 Setting the Photon Detection Threshold

The phenomenon of dark count, described in 5.1.1 is also a problem of using a MPPC for photon detection. The generation of noise is supported by thermal excitations. The noise component is amplified in Geiger mode operation and the original photon detection signal cannot be discriminated from the noise. To eliminate the dark count problem it is necessary to set a suitable threshold level. According to the datasheet, it is very unlikely that dark counts at 2 p.e., 3 p.e. or 4 p.e level are detected. In other words, if a large number of photons is detected, the effects of dark counts can be virtually eliminated by setting a proper threshold level. The SLR station Graz delivers 400 μJ / shot at 532 nm, which corresponds to about 10^{14} photons, with a divergence of the laser beam of 10 arc seconds. If the MPPC is mounted on a LEO satellite in about 1000 km distance, every SPAD would receive about 100 photons. For the satellite CHAMP, in a distance of about 500 km, each diode would receive about 400 photons, which is more than enough to trigger a break of each SPAD [11].
For this project, a threshold level of 2.5 p.e. (highest possible threshold level) would be inappropriate, because of a too low trigger level. Instead of, the analogue out is used. The quantum efficiency of each SPAD is about 27% at 532 nm. That means that the break of a SPAD is basically of statistical nature. There is a small percentage of SPADs which will not trigger due to the arriving photons. Important is that also the trigger threshold of the analogue comparator can be set accordingly, so that a defined number of SPADs (e.g. 50 or 100 out of the 400 SPADs on the chip) will always result in a TTL pulse output.

5.2.2 Epoch Timing of Returns

The TTL compatible output of the analogue comparator output is connected to a so called event timer. This event timer is implemented as a 200 MHz counter on the CPLD. The CPLD itself is connected to the 1 pps and the 10 MHz reference frequency of the GPS Time and Frequency Receiver of the Graz SLR station. This reference frequency is very accurate and makes it possible to determine the epoch times of detected laser pulses with 5 ns resolution. The following figure 5.7 in section 5.3 on the next page shows the final circuit configuration. The second interface on the circuit board does not have any function or importance. Only one interface (with the FT245RL chip) is important for reading out the epoch times from the 200 MHz counter of the CPLD. The other interface (with the FT245BL chip) was used for realizing the so called loopback test, which just tested the correct operating principle of the CPLD and the interfaces. For this test two interfaces were used. One for reading in any string and the other one was for retrieving or receiving this string. These two strings were equal and the test was therefore successful.

5.3 Test Setup and Results

The Laser Control PC reads one of several ASCII text files. The telescope is pointed to a retroreflector in a distance of 4288 m. The attenuated laser is fired to this target and on request of the observer it applies PPM to the laser firing epochs to encode the characters of the selected ASCII text file. The photons reflected from the retro-reflector are detected by the MPPC, which is mounted in the main SLR receiving telescope. The incoming photon stream has a proper attenuation - the MPPC produces the required output pulses and their epoch times are decoded and the complete ASCII text file is recovered and compared with the original file. This is explained in 5.3.1. For the main project, only one interface is used. Only receiving data from the CPLD is necessary.

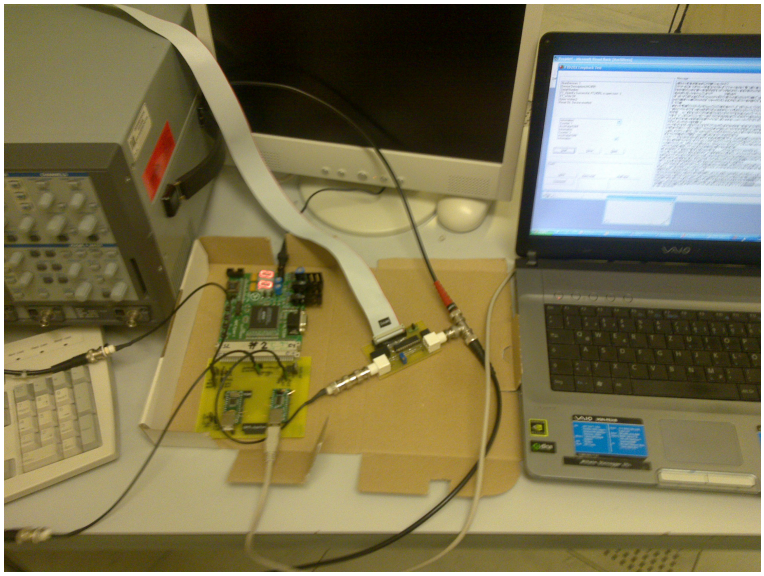

Figure 5.7: The final circuit configuration

Image 5.7 above shows that the CPLD is connected via USB to the laptop and delivers the epoch times of all detected laser pulses. On the laptop, the discussed detection software is executed and determines the ASCII values. How this happens can be read in the last sections of this Master Thesis.

5.3.1 Test Results

For test transmissions, different texts were used. The first one was the description of the diploma work of this project with a length of 1510 characters.
At the beginning this file was read by the routine calibration program, when pointing to the remote target in 4288 m. The calibration program added the offsets for each character to the basic firing commands. The epoch times of the MPPC-received returns were finally

decoded by the laptop. At a laser firing rate of 2 kHz, the transmission took about 0.76 s. The transmitted laser energy was attenuated as well as the received energy, to get photon numbers per SPAD similar to those calculated for LEO satellites. To make it possible to transmit the coded laser pulses during full daylight conditions, a standard wavelength filter with a band width of 0.3 nm was placed in front of the detector. To use this system on a LEO satellite, a similar filter with a larger band width - concerning to large incident angle variations - has to be used. To check for transmission errors, the laptop program stored the decoded messages into an ASCII file on hard disk. This ASCII file with the received messages was finally opened by an ASCII editor with a spell checker. In case of too low received energy (e.g. due to intentional offset pointing), transmission errors occurred. After checking the errors, the basic offset of 40 ns was changed to 80 ns. The following text contains the original text, which was first transmitted.

```
Coding of Laser Pulse Transmission Time

Basic Task: Laser Firing Times are shifted by N*50 ns,
encoding thus information

Position Coding of Laser Pulses

-  Laser Pulses of the SLR Station Graz-Lustbühel are fired at 500 as Time
   intervals (2 kHz)

-  Laser Firing Times are controlled within an FPGA circuit;

-  The FPGA is on an ISA PC Card, and allows easy programming of such
   additional delays;

-  The Accuracy of Laser Time Firing is  +- 7 ns

-  Apply additional Delays for each firing time: N*50 ns

-  This allows Coding of Information:

   N can be substituted with e.g. ASCII character set definition;
   e.g. N= 65 means "A"  (i.e. 65*50 ns = 3250 ns additional delay)
   up to 255x50 ns = 12750 ns (512.75/500 = 1997 Hz instead of 2000 Hz)
   Consequence: Coding does NOT reduce significantly the Ranging Frequency

   Purpose / Advantage:
   Allows transmission of information (e.g. to a satellite)
   Data Rate:  ? 2000 Characters / second
   Detection of this is limited to vicinity around laser beam
   Difficult for "unauthorized listening";

Tasks to do:

-  Design / program the additional circuitry in the FPGA to allow for these
   additional delays;

-  Design a simple optical receiver, to detect these laser pulses;

-  The detected pulses should be re-translated into the original code,
   using a simple program in a Laptop, or PDA, and display / store it

Graz, 14.02.2008                                          Georg Kirchner
```

As already mentioned, the first test run was not successful because the decoded text contained a lot of errors. Most errors were rounding errors e.g. “c” instead of “b” or “a”. In some cases the letters were skipped and it was just possible to read half words. The following sentence is a short excerpt from the received text, which was stored in ASCII format on hard disk after decoding.

```
– Eeign a uimle opticamseceiver– to detect these laserpuse;
```

The original sentence, which was transmitted would be:

```
– Design a simple optical receiver, to detect these laser pulses;
```

As we can see the received sentence has some typical errors, which were already explained. Some characters were ignored, because the laser pulses did not reach the MPPC and in the following the CPLD could not generate any epoch time for the relevant character. The next strategy was to increase the offset of the coded laser pulses up to 80 ns and to vary the operating voltage of the MPPC. In case of an offset of 80 ns, the maximal acceptable jitter or uncertainty can be 75 ns. Until this jitter value a correct decoding of ASCII values is possible but if the uncertainty becomes higher than the offset of 80 ns, the determination of ASCII characters will not work any more - a wrong interpretation of characters like it is shown above would happen. The offset of the coded laser pulses cannot be arbitrary increased. The distance between two basic epoch time values amounts 500 μs plus some jitter. This distance must not become higher than these 500 μs. A closer look to the ASCII table in [6] gives information about the highest possible ASCII value. If the standard character set is used, the maximum offset can be 10160 ns (80 ns $*$ 127) - alternatively 10.160 μs. In case of using the country-specific character set, the maximum possible offset can be 20400 ns (80 ns $*$ 255) - alternatively 20.400 μs. After looking at the transmission text, “ü” can be seen.
From this it follows that also the country specific character set can be recognized from the hard- and software without any interpretation problems. After these mentioned modifications, the transmitted text was interpreted correctly by the detection hard- and software.

5.3.2 Offset from the Basic Grid

The transmitted text has a length of 1510 characters. In times of transmitting an ASCII character by using a 2 kHz laser firing rate, the epoch time deviates from a basic grid of 500 μs. In all other times, the epoch time values have a constant distance of 500 μs plus a possible uncertainty. The following diagram shows the deviation from the basic grid in a diagram.

Figure 5.8 shows the deviations from the nominal 500 μs grid on the left Y-axis of the diagram. The corresponding ASCII value is illustrated on the right Y-axis.

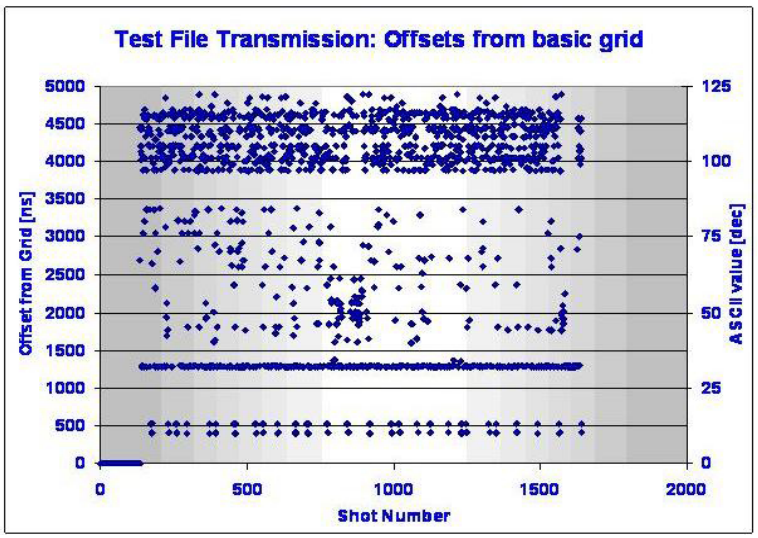

Figure 5.8: Deviation from nominal 500 μs grid

As mentioned in 5.3.1 the transmitted text has a total length of 1510 characters. This can be seen on the shot numbers on the X-axis of the diagram.
After taking a closer look to the plotted laser shots, it can be recognized that blanks alternatively spaces (ASCII 32) are the most frequent ASCII values in this text, followed by carriage returns (ASCII 13) and line feeds (ASCII 10). On the left, epoch time values without any offset or without any deviation from the basic grid can be seen. These values do not describe any ASCII characters and can be used for determining the zero value or reference value. These values touch the X-axis of the diagram.
ASCII values between 65 and 90 are upper case characters. These are in the normal case initial letters of a sentence. The values between ASCII 97 and ASCII 122 are lower case characters, which are of course in superior numbers. The rest of the data points are special characters like “!”, “?” etc.
Unfortunately the diagram only shows characters from ASCII range 1 to ASCII range 125. If the diagram describes all possible values from ASCII range 1 to ASCII range 255, it can be seen that the text also contains an ASCII character with ASCII value 252. That would be the “ü” in the word “Lustbühel”, which was also detected and decoded correctly by the detection hardware and software.

5.4 The final Test Transmission

This section just introduces the second text, which was used for testing the detection equipment - especially the detection software for the last time. The main goal of this was to finish this project.

```
A man walks into a bar and orders a drink.
The bar has a robot bartender.
The robot serves him a perfectly prepared cocktail,
and then asks him, "What's your IQ?"

The man replies "150" and the robot proceeds
to make conversation about global warming factors,
quantum physics and spirituality, biomimicry,
environmental interconnectedness, string theory,
and nano-technology.

The customer is very impressed and thinks:
"This is really cool." He decides to test the robot.
He walks out of the bar, turns around,
and comes back in for another drink.

Again, the robot serves him the perfectly prepared
drink and asks him, "What's your IQ?"

The man responds, "About a 100."
Immediately the robot starts talking, but this time
about football, baseball, supermodels,
favorite fast foods, guns, and women's breasts.

Really impressed, the man leaves the bar and decides
to give the robot one more test. He heads out and
returns, the robot serves him and asks, "What's your IQ?"

The man replies, "Er, 50, I think."
And the robot says... real slowly...
"So.............. ya gonna vote for Bush again?"
```

Important to mention is that this text was copied to hard disk in ASCII format by the detection software. This happened after decoding the ASCII characters - described by epoch times. The text above was checked by a spell checker provided by Microsoft Word. The result was that this text does not contain any misspellings or interpretation errors. From this it follows that the detection software and hardware had worked correctly.

6 Conclusions

In this work detection hardware and software were analyzed to transmit and to receive coded laser pulses. The main task in this Master Thesis was to design and to program a detection software to extract ASCII characters described by coded laser pulses. The coded laser pulses were transmitted by the 2 kHz SLR station Graz / Lustbühel to a fixed retro-reflector target at a distance of 4288 m.

The first task of this thesis was to study the principle of decoding or extracting ASCII characters from varied epoch time values. These studies did not use any hardware but rather software - programmed in Visual Basic 6.0. The next task in this thesis was to select a suitable hardware, which included the MPPC from Hamamatsu. To get correct results an optimal threshold value was chosen - to reduce effects of dark count. Hardware programming e.g. the programming of the Altera CPLD and the ISA card, which controlled the laser firing commands were not part of this work. Therefore just a few important facts were discussed. During the preparations of selecting the hardware parts the idea came up to use a Jupiter GPS board to generate the 10 MHz reference frequency. This hardware part was not mentioned in this work because it did not came in general use. For the final tests, the 10 MHz reference frequency of the GPS Time and Frequency Receiver of the Graz SLR station were used.

The next part was to find an optimal interface to get the generated time values from the CPLD into the PC and detection software. Very important was the transfer rate, which varied from interface to interface - a RS 232 interface would be easy to program but the transfer rate would be much too slow. This problem was also discussed in the hardware chapter of this present thesis.

After selecting the interface, the detection software was programmed. This contained the sequential control of receiving time values from the selected interface. This was described in detail in the same chapter. The decoding of the coded laser pulses to get the ASCII characters was discussed in the first chapters with the help of a few examples. Basically these perceptions for decoding the contained ASCII characters were assumed for implementing the final detection software.

In the theoretical part the functionality of a SPAD was explained and in addition to it the MPPC, which consists of a fixed number of SPADs (SPAD array). In the last chapter the test results were presented and adequate methods for avoiding the upcoming detection errors (e.g. raising the offset up to 80 ns) were introduced.

Applying the Pulse Position Modulation to the firing times of laser pulses at the existing and operational kHz SLR stations, offers an opportunity for a new data upload channel to satellites in orbits up to about 1000 km. The only requirement at the satellite side is a photon

detector, without the need of optics, a 5 ns resolution time tagging unit and possibly a CCR. On the kHz SLR station side, the upgrade depends on the specific hardware, but in many cases that would be simply a few more lines of software to control the laser firing times of the SLR station with an accuracy of some nanoseconds.

6.1 Main Challenges

During this project some problems occurred, which were already mentioned in the last chapters. One of the main problems was to chose a convenient interface. At first, an interface called USB-PIO was chosen. This was very easy to program in Visual Basic 6.0 but it provided a too slow transfer rate (ca. 30 measured points per second).
After some researches and studying some data sheets of different interfaces, the UM245R was chosen for the final version of the detection system. The biggest benefit of using this interface was the high transfer rate. The data sheet of it indicates a transfer rate of 1 megabyte per second. But this were not reached in this project, although the functions of the D2XX Direct Driver were used and not the Virtual Com Port driver. The provided transfer rate of the interface was high enough to provide enough free FIFO storage positions in the CPLD for new data.
Not only problems concerning the interface occurred but also problems with regard to the selected programming language. A detection software is not allowed to terminate and has to check if data is available or not. This is an uninterruptible process. To guarantee that all available data is read, an endless loop had to be implemented. Such an “endless program” highly stresses the processor of the PC and in this case the program often hanged up. One possibility to solve this problem was to use the “DoEvents” instruction of Visual Basic. This instruction allows multitasking and the PC is able execute other programs too. In other words, the detection software loses the processor for 1 ms after a “DoEvents” instruction. Within 1 ms the processor can be used by other open programs or the PC can react on user instructions. One problem of using the DoEvents statement was that the program stopped reading in time values for 1 ms. The effect was that the FIFO of the CPLD and the interface became full and no more new time values could be stored (data loss). Unfortunately therefore it was not possible to decode ASCII characters in a technically correct manner.
The solution was to allow a DoEvents instruction just every 10000 loop cycles. To put it another way the detection software was interrupted after 10000 cycles for 1 ms. This worked well enough and the mentioned FIFOs were read out in an acceptable period of time. Data loss was from now on non-issue.
After programming the detection software, some errors occurred relating on the final tests. Coded laser pulses were not decoded in a correct way or were missed. Afterwards the operating

voltage of the MPPC were varied and the offset of the coded pulses were doubled up to 80 ns. This aim was reached by reprogramming the ISA card, which was responsible for the laser commands. Due to optical wavelengths (532 nm in almost all SLR stations), optical visibility in the line of sight to the satellite is required. Any cloud, fog etc. will prohibit SLR as well as successful data transmission. In section 6.2 some conventional techniques for error detection and / or correction are mentioned and should be applied.

6.2 Future Work

The main goal of this project was to show that the 2 kHz SLR station Graz / Lustbühel is able to transmit coded laser pulses after varying the fixed laser pulses. This challenge was successful but in some cases it would not be possible to avoid transmission errors e.g. inaccurate pointing, due to clouds - blocking the laser path. To avoid these problems in future, any simple check method can be implemented e.g. checksums, even- or odd parity checks, error-correcting codes like Verhoeff algorithm, cyclic redundancy checks etc. The use of linear regressions could be useful to determine zero values more exactly, especially if the influence of jitter is much higher.
Another simple check, shortly discussed in [11] is to monitor the SLR return amplitude at the station. If the station received a valid return with some minimum energy level, the transmission should have been successful.
Before the MPPC can be used as detector for the PPM SLR uplink channel, its space qualification has to be proven. This has not been done yet. In case of no space qualification, any other linear detection device based on a photon detector in linear gain mode, which is space qualified may be used. The first test could be with ACES; flying on the ISS in 2013. For time transfers, there will be already a CCR on board; adding the MPPC and the electronics would complete the test setup there.

Bibliography

[1] *Future Technology Devices International Ltd. FT245R USB FIFO IC.*

[2] *MPPC Multi-Pixel Photon Counter, Technical Information - Hamamatsu.*

[3] *USB-PIO Digitale I/0 Schnittstelle (USB), BMC Messsysteme Gmbh.*

[4] S. Cova, A. Lacaita, M. Ghioni, and G. Ripamonti, *20-ps Timing Resolution with Single-Photon Avalanche Diodes*, Rev. Sci. Instrum. no.6 **60** (1989), 1104–1110.

[5] Elektronik-Kompendium, *Pulse Position Modulation (PPM).*, `http://www.elektronik-kompendium.de/sites/kom/0401121.htm`, May 2010.

[6] T. Horn, *ASCII-Tabelle*, `http://www.torsten-horn.de/techdocs/ascii.htm`, June 2010.

[7] M. J. Hsu, H. Finkelstein, and S. C. Esener, *A CMOS STI-Bound Single-Photon Avalanche Diode With 27-ps Timing Resolution and a Reduced Diffusion Tail*, IEEE electron device letters **30** (2009), 641–643.

[8] A. S. Huntington, M. A. Compton, and G. M. Williams, *Improved Breakdown Model for estimating Dark Count Rate in Avalanche Photodiodes with InP and InAlAs Multiplication Layers*, SPIE Defense and Security Symposium - Pre-release Manuscript 6214-29.

[9] IWF, *SLR-Station Graz - Lustbühel*, http://www.iwf.oeaw.ac.at/de/forschung/erdkoerper/slr-technologie/slr-station-graz-lustbuehel.html, December 2007.

[10] J. C. Jacksona, D. Phelanb, A. P. Morrisonc, R. M, Redfern, and A. Mathewson, *Characterization of Geiger Mode Avalanche Photodiodes for Fluorescence Decay Measurements*, **4650-07** (2002), 1–5.

[11] G. Kirchner, F. Koidl, W. Steinegger, F. Iqbal, and E. Leitgeb, *Data Transmission to Satellites using SLR Systems*, not published **1** (2010), 1:6.

[12] O. Koudelka, *Nachrichtentechnik 441.103, Skriptum zur Vorlesung*, (2003), 23.

[13] E. Leitgeb and P. Fasser, *Optische Nachrichtentechnik 441.023, Skriptum zur Vorlesung*, (2005).

[14] C. Niclass, M. Sergio, and E. Charbon, *A Single Photon Avalanche Diode Array Fabricated in Deep-Submicron CMOS Technology*, **1** (2006), 2.

[15] S. Privitera, S. Tudisco, L. Lanzanò, F. Musumeci, A. Pluchino, A. Scordino, A. Campisi, L. Cosentino, P. Finocchiaro, G. Condorelli, M. Mazzillo, S. Lombardo, and E. Sciacca, *Single Photon Avalanche Diodes: Towards the large Bidimensional Arrays*, (2008).

[16] A. K. Ray and T. Acharya, *Information Technology: Principles and Applications*, Prentice-Hall of India, 2004.

[17] J. Schiller, *Mobile Communications*, Pearson Education Limited, 2003.

[18] A. Spinelli and A. L., *Physics and Numerical Simulation of Single Photon Avalanche Diodes*, IEEE TRANSACTIONS oN ELECTRON DEVICES, NO. 1 **44** (1997), 1931–1935.

[19] S. Vasile, P. Gothoskar1, R. Farrell, and D. Sdrulla, *Photon Detection with High Gain Avalanche Photodiode Arrays*, **45** (1998), 1,2.

[20] J. Weingrill, *Zeiss Ballistische Messkammer mit 75 cm Brennweite, 18x18cm Bildfeld und Blende 1:2,5*, http://www.flickr.com/photos/joerg73/3040699984/, May 2005.

[21] K. Yamamoto, *Newly developed Semiconductor Detectors by Hamamatsu*, International workshop on new photon-detectors PD07 **1** (2007), 1–12.

[22] K. Yamamoto, K. Yamamura, K. Sato, T. Ota, H. Suzuki, and S. Ohsuka, *Development of Multi-Pixel Photon Counter (MPPC)*, IEEE Nuclear Science Symposium Conference Record (2007), 1094–1097.

A Abbreviations and Glossary

AM	Amplitude Modulation
ASCII	American Standard Code for Information Interchange
ASK	Amplitude Shift Keying
CCR	Corner Cube Reflector
CPLD	Complex Programmable Logic Device
FM	Frequency Modulation
FPGA	Field Programmable Gate Array
FSK	Frequency Shift Keying
GNU	GNU's Not Unix
GUI	Graphical User Interface
ISA	Industry Standard Architecture
LASER	Light Amplification by Stimulated Emission of Radiation
MByte	Megabyte
MPPC	Multi Pixel Photon Counter
PAM	Pulse Amplitude Modulation
PC	Personal Computer
PCM	Pulse Code Modulation
P.E.	Photon Equivalent
PIO	Programmed Input/Output
PM	Phase Modulation
PSK	Phase Shift Keying
PWM	Pulse Width Modulation
SLR	Satellite Laser Ranging
SPAD	Single Photon Avalanche Diode

TTL Transistor Transistor Logic

USB Universal Serial Bus

VB Visual Basic

B Source Codes

B.1 Source Code for decoding ASCII Characters from Epoch Times

This source code is explained in 3.1.2 and 3.1.5. The "old algorithm", which is commented out takes the first epoch time value as reference value. The newer algorithm searches ten equal epoch time values (with 500 μs distance to the neighbored value) to determine a suitable reference value for decoding the contained message.

```
'Version 22.01.2009
'Wilhelm Steinegger

Option Explicit
Private Ending As Long
Private Declare Function GetTickCount Lib "kernel32" () As Long

Private Sub Command1_Click()
End
End Sub

Private Sub End_Click()
Beep
End

End Sub

Private Sub Form_Load()
'to read in the lines of the testfile

Start = GetTickCount

Dim sFile As String
Dim sRow As String
Dim FNr As Double
Dim Factor As Integer
Dim CurrentVariable
Dim DestinationVariable_1
Dim DestinationVariable_2
Dim sArray(1 To 113) As Double
Dim Counter As Integer
```

```
32 Dim ReferenceValue As Double
33 Dim CurrentValue As Double
34 Dim Difference As Double
35 Dim CalculatedValue As Double
36 Dim AsciiN As Double
37 Dim ValueCounter As Integer
38 Dim ArrayIndex As Integer
39 Dim BasicValue As Integer
40 Dim NextValue As Double
41 Dim NextElementIndex As Integer
42 Dim Difference_2 As Double
43 Dim NValues As Integer
44 Const Distance = 500 * 10 ^ -6
45 Const NFactor = 50 * 10 ^ -9
46 Dim MessageSymbol As String
47 Dim MessageArray(1 To 113) As Variant
48
49 'Dim test As Integer
50 'First we read in the txt-File and save the values in an array called
      sArray
51 '--------------------------------read in--------------------------------
52 sFile = "D:\DA\willi2.txt"
53 'sFile = "D:\DA\WILLI.txt"
54 FNr = FreeFile
55 'open File
56 Open sFile For Input As #FNr
57 'read until the End
58
59 Counter = 0
60
61 Do While (Not (EOF(FNr)))
62     'read line
63     Line Input #FNr, sRow
64     'Text1.Text = " "
65         Counter = Counter + 1
66     CurrentVariable = Val(sRow)
67     'MsgBox "CurrentVariable" & CurrentVariable
68     sArray(Counter) = CurrentVariable
69
70 Loop
71 Close #FNr
72 '------------------------------------------------------------------------
```

```
'the old algorithm

'Counter = 0
'FirstValue = sArray(1)

'For Counter = 2 To 113 Step 1
'    CalculatedValue = FirstValue + (500 * 10 ^ -6) * (Counter - 1)
'    Difference = Abs(CalculatedValue - sArray(Counter))

'  If Difference = 0 Or Difference <= 10 * 10 ^ -9 Then
'MsgBox "No Ascii"

'  Else
'AsciiN = Round(Difference / (50 * 10 ^ -9))
'MsgBox Chr(AsciiN)

'End If
'Next
'-------------------------------------------------------------------------
Counter = 0
ArrayIndex = 0

Do While Counter < 9
    ArrayIndex = ArrayIndex + 1
    Difference = Abs(sArray(ArrayIndex + 1) - sArray(ArrayIndex))

    If Difference = Distance Or Difference <= Distance + NFactor Then
        Counter = Counter + 1
    Else
        Counter = 0
    End If

Loop

If Counter >= 9 Then
    ReferenceValue = sArray(ArrayIndex)
    NValues = 113
    MsgBox "ArrayIndex" & ArrayIndex
    MsgBox "ReferenceValue" & ReferenceValue

    For ArrayIndex = ArrayIndex + 1 To NValues Step 1
        Factor = Factor + 1
```

```
115             CalculatedValue = ReferenceValue + (Distance) * Factor
116             Difference = Abs(CalculatedValue - sArray(ArrayIndex))
117             AsciiN = Round(Difference / (NFactor))
118             MessageArray(Factor) = Chr(AsciiN)
119         Next
120     End If
121
122     ArrayIndex = 0
123     Message.Text = " "
124
125     For ArrayIndex = 1 To 113 Step 1
126         Message.Text = Message.Text & MessageArray(ArrayIndex)
127     Next
128
129     End Sub
```

B.2 Generating artificial Epoch Times and decoding ASCII Characters

This algorithm is explained in 3.2. This source code is able to generate artificial epoch times from an ASCII file. After the values are generated, the algorithm starts with determining the ASCII characters. Finally the ASCII file and the determined text are equal. The ASCII character determination works in the same way like in B.1.

```
1  'Last Update 29.04.2009
2  Private Declare Function timeGetTime Lib "winmm.dll" () As Long
3  Public LastEpochTime As Double 'global variable witch stores the last
       EpochTime of the function
4  Option Explicit
5  Public Function RandomValue() As Double 'for jitter
6
7  Dim RandomVariable As Double
8  RandomVariable = 0
9
10 RandomVariable = Rnd 'Value between 0 and 1
11 RandomVariable = RandomVariable - 0.5
12 RandomVariable = RandomVariable * 10 ^ (-8) 'We need nanoseconds
13 RandomValue = RandomVariable 'Return Value
14
15 End Function
16 'Just an experiment
17 Public Function txt_ReadAll(ByVal sFilename As String) _
18   As String
19
20   Dim F As Integer
21   Dim sContent As String
22
23   If Dir$(sFilename, vbNormal) <> "" Then
24     F = FreeFile
25     Open sFilename For Binary As #F
26     sContent = Space$(LOF(F))
27     Get #F, , sContent
28     Close #F
29     Else
30     MsgBox "File not available"
31   End If
32
33   txt_ReadAll = sContent
```

```
34
35 End Function
36 Private Sub Start_Click()
37
38 Const Path = "C:\Dokumente und Einstellungen\Wilhelm Steinegger\
       Desktop\string_readin\simple_test.txt"
39 Call ReadLetter(Path)
40 Call Decode
41
42 End Sub
43 Public Function ReadLetter(ByVal sFilename As String) _
44     As String
45
46 Dim Length As Integer
47 Dim Counter As Integer
48 Dim AsciiValue As Integer
49 Dim FNr As String
50 Dim strInput As String
51 Dim Letter As String
52 Dim LastEpochTime As Double
53 Dim CodedValue As Double
54 Dim Jitter As Double
55 Dim CurrentTime As Double
56 Dim EpochTime As Double
57 Dim CarriageReturn As Double
58
59 Const N = 50 * 10 ^ -9
60 Const Distance = 500 * 10 ^ -6
61
62 Text1.Text = "" 'Textbox should be empty for restart
63 Counter = 0
64 LastEpochTime = GetEpochTime
65 FNr = FreeFile
66 Open sFilename For Input As #FNr
67 Text1.Text = " "
68
69 Do While (Not (EOF(FNr)))
70     Line Input #FNr, strInput
71
72     For Counter = 1 To Len(strInput) Step 1
73         Jitter = RandomValue 'Random generator
74         Letter = Mid$(strInput, Counter, 1)
```

```
75          AsciiValue = Asc(Letter)
76          Text1.Text = Text1.Text & Chr$(13) & Chr$(10)
77          Text1.Text = Text1.Text & "    Letter:" & Letter
78          Text1.Text = Text1.Text & "    AsciiValue:" & AsciiValue
79          EpochTime = LastEpochTime + (Distance) * Counter
80          CodedValue = EpochTime + Jitter + (N * AsciiValue)
81          List1.AddItem (CodedValue)
82      Next Counter
83
84      If Counter = Len(strInput) + 1 Then
85          'we know that there is a new line in the Ascii-File if this
                condition fulfills
86          EpochTime = LastEpochTime + (Distance) * (Counter)
87          CodedValue = EpochTime + Jitter + (N * 13)
88          List1.AddItem (CodedValue)
89          EpochTime = LastEpochTime + (Distance) * (Counter + 1)
90          CodedValue = EpochTime + Jitter + (N * 10)
91          List1.AddItem (CodedValue)
92          LastEpochTime = EpochTime
93      End If
94  Loop
95
96  ReadLetter = strInput
97  Close #FNr
98
99  End Function
100 Public Function GetEpochTime() As Double
101
102 Dim EpochTime As Double
103 Dim Counter As Integer
104 Dim Jitter As Double
105 Dim CurrentTime As Double
106 Const Distance = 500 * 10 ^ -6
107
108 List1.Clear
109 CurrentTime = GetCurrentTime
110 Counter = 0
111 Jitter = 0
112 EpochTime = CurrentTime
113
114 For Counter = 1 To 100 Step 1
115     Jitter = RandomValue
```

```
116     EpochTime = EpochTime + (Distance) + Jitter
117     List1.AddItem (EpochTime)
118 Next Counter
119
120 If Counter >= 100 Then
121     GetEpochTime = EpochTime 'Return Value
122 End If
123
124 End Function
125 Public Function GetCurrentTime() As Double
126
127 GetCurrentTime = timeGetTime * 10 ^ -9
128
129 End Function
130
131 Public Function Decode()
132
133 Dim Counter As Integer
134 Dim CurrentValue As Double
135 Dim ReferenceValue As Double
136 Dim NextValue As Double
137 Dim Difference As Double
138 Dim ValueCounter As Integer
139 Dim ValueCounterIncr As Integer
140 Dim CalculatedValue As Double
141 Dim AsciiN As Double
142 Dim ListCounter As Integer
143 Dim Factor As Integer
144 Dim ValuesInList As Integer
145 Dim Letter As Variant
146 Dim ArrayCounter As Integer
147 Dim MeanValue As Double
148 Const Distance = 500 * 10 ^ -6
149 Const NFactor = 50 * 10 ^ -9
150 Const Jitter = 5 * 10 ^ -9
151 Text2.Text = " "
152 'List2.Clear
153
154 ValuesInList = List1.ListCount
155
156 MsgBox "ValuesInList" & ValuesInList
157 ArrayCounter = 0
```

```
158
159 Do While Counter < 100
160     If CurrentValue = 0 Or NextValue = 0 Then
161         ListCounter = 0
162         CurrentValue = List1.List(ListCounter)
163         NextValue = List1.List(ListCounter + 1)
164         ListCounter = ListCounter + 1
165     Else
166         CurrentValue = NextValue
167         NextValue = List1.List(ListCounter + 1)
168         Difference = Abs(CurrentValue - NextValue)
169         ListCounter = ListCounter + 1
170     End If
171
172     If Difference = Distance Or Difference >= Distance - Jitter Or
        Distance <= Distance + Jitter Then
173             Counter = Counter + 1
174         Else
175             Counter = 0
176     End If
177 Loop
178
179 If Counter >= 100 Then
180         ReferenceValue = List1.List(ListCounter - 1)
181         MsgBox "Reference Value:" & ReferenceValue
182
183         Do While ListCounter < List1.ListCount - 1
184                                                 Letter = 0
185             Factor = Factor + 1
186             CalculatedValue = ReferenceValue + (Distance) * Factor
187             CurrentValue = List1.List(ListCounter)
188             Difference = Abs(CurrentValue - CalculatedValue)
189             AsciiN = Round(Difference / (NFactor))
190             Letter = Chr(AsciiN)
191             'Text2.Text = Text2.Text & Letter
192             Text2.SelText = Letter
193             'List2.AddItem (AsciiN)
194             'List3.AddItem (Letter)
195             ListCounter = ListCounter + 1
196         Loop
197 End If
198 End Function
```

B.3 Source Code for reading out Epoch Time Values from the Detection Hardware

The main part of this Master Thesis was to develop an algorithm, which is able to communicate with the detection hardware. The documentation of this algorithm starts at 4.4.
The following source code contains this algorithm in Visual Basic 6.0.

```
1  'Last Update 1.06.2010
2  'This program has been successfully tested on 09.06.2010
3  'Wilhelm Steinegger
4  'Declarations of FTD2XX.DLL Functions
5  Private Declare Function FT_Open Lib "FTD2XX.DLL" (ByVal
       intDeviceNumber As Integer, ByRef LngHandle As Long) As Long
6  Private Declare Function FT_OpenEx Lib "FTD2XX.DLL" (ByVal arg1 As
       String, ByVal arg2 As Long, ByRef LngHandle As Long) As Long
7  Private Declare Function FT_Close Lib "FTD2XX.DLL" (ByVal LngHandle As
        Long) As Long
8  Private Declare Function FT_Read Lib "FTD2XX.DLL" (ByVal LngHandle As
       Long, ByVal lpszBuffer As String, ByVal lngBufferSize As Long,
       ByRef lngBytesReturned As Long) As Long
9  Private Declare Function FT_Write Lib "FTD2XX.DLL" (ByVal LngHandle As
        Long, ByVal lpszBuffer As String, ByVal lngBufferSize As Long,
       ByRef lngBytesWritten As Long) As Long
10 Private Declare Function FT_SetBaudRate Lib "FTD2XX.DLL" (ByVal
       LngHandle As Long, ByVal lngBaudRate As Long) As Long
11 Private Declare Function FT_SetDataCharacteristics Lib "FTD2XX.DLL" (
       ByVal LngHandle As Long, ByVal byWordLength As Byte, ByVal
       byStopBits As Byte, ByVal byParity As Byte) As Long
12 Private Declare Function FT_SetFlowControl Lib "FTD2XX.DLL" (ByVal
       LngHandle As Long, ByVal intFlowControl As Integer, ByVal
       byXonChar As Byte, ByVal byXoffChar As Byte) As Long
13 Private Declare Function FT_ResetDevice Lib "FTD2XX.DLL" (ByVal
       LngHandle As Long) As Long
14 Private Declare Function FT_SetDtr Lib "FTD2XX.DLL" (ByVal LngHandle
       As Long) As Long
15 Private Declare Function FT_ClrDtr Lib "FTD2XX.DLL" (ByVal LngHandle
       As Long) As Long
16 Private Declare Function FT_SetRts Lib "FTD2XX.DLL" (ByVal LngHandle
       As Long) As Long
17 Private Declare Function FT_ClrRts Lib "FTD2XX.DLL" (ByVal LngHandle
       As Long) As Long
18 Private Declare Function FT_GetModemStatus Lib "FTD2XX.DLL" (ByVal
       LngHandle As Long, ByRef lngModemStatus As Long) As Long
```

```
19 Private Declare Function FT_Purge Lib "FTD2XX.DLL" (ByVal LngHandle As
       Long, ByVal lngMask As Long) As Long
20 Private Declare Function FT_GetStatus Lib "FTD2XX.DLL" (ByVal
       LngHandle As Long, ByRef lngRxBytes As Long, ByRef lngTxBytes As
       Long, ByRef lngEventsDWord As Long) As Long
21 Private Declare Function FT_GetQueueStatus Lib "FTD2XX.DLL" (ByVal
       LngHandle As Long, ByRef lngRxBytes As Long) As Long
22 Private Declare Function FT_GetEventStatus Lib "FTD2XX.DLL" (ByVal
       LngHandle As Long, ByRef lngEventsDWord As Long) As Long
23 Private Declare Function FT_SetChars Lib "FTD2XX.DLL" (ByVal LngHandle
       As Long, ByVal byEventChar As Byte, ByVal byEventCharEnabled As
       Byte, ByVal byErrorChar As Byte, ByVal byErrorCharEnabled As Byte)
       As Long
24 Private Declare Function FT_SetTimeouts Lib "FTD2XX.DLL" (ByVal
       LngHandle As Long, ByVal lngReadTimeout As Long, ByVal
       lngWriteTimeout As Long) As Long
25 Private Declare Function FT_SetBreakOn Lib "FTD2XX.DLL" (ByVal
       LngHandle As Long) As Long
26 Private Declare Function FT_SetBreakOff Lib "FTD2XX.DLL" (ByVal
       LngHandle As Long) As Long
27 Private Declare Function FT_ListDevices Lib "FTD2XX.DLL" (ByVal arg1
       As Long, ByVal arg2 As String, ByVal dwFlags As Long) As Long
28 Private Declare Function FT_GetNumDevices Lib "FTD2XX.DLL" Alias "
       FT_ListDevices" (ByRef arg1 As Long, ByVal arg2 As String, ByVal
       dwFlags As Long) As Long
29 Private Declare Function FT_SetBitMode Lib "FTD2XX.DLL" (ByVal
       LngHandle As Long, ByVal ucmask As Byte, ByVal ucenable As Byte)
       As Long
30 Private Declare Function FT_GetBitMode Lib "FTD2XX.DLL" (ByVal
       LngHandle As Long, ByRef mode As Byte) As Long
31 Private Declare Function FT_ResetPort Lib "FTD2XX.DLL" (ByVal
       LngHandle As Long) As Long
32 Private Declare Function FT_EE_UASize Lib "FTD2XX.DLL" (ByVal
       LngHandle As Long, ByRef lpdwSize As Long) As Long
33 Private Declare Function FT_SetEventNotification Lib "FTD2XX.DLL" (
       ByVal LngHandle As Long, ByVal dwEventMask As Long, ByVal Arg As
       Long) As Long
34
35 'System Windows Functions
36 Private Declare Function GetTime Lib "winmm.dll" Alias "timeGetTime"
       () As Long 'timegetTime for stopping the time
```

```
37 Private Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As Long
       ) 'Sleep function to hibernate the program
38 'Declare Function GetInputState Lib "user32" () As Long
39 Private Declare Function GetInputState Lib "user32" () As Long
40
41 'Return codes
42 Const FT_OK = 0
43 Const FT_INVALID_HANDLE = 1
44 Const FT_DEVICE_NOT_FOUND = 2
45 Const FT_DEVICE_NOT_OPENED = 3
46 Const FT_IO_ERROR = 4
47 Const FT_INSUFFICIENT_RESOURCES = 5
48 Const FT_INVALID_PARAMETER = 6
49 Const FT_INVALID_BAUD_RATE = 7
50 Const FT_DEVICE_NOT_OPENED_FOR_ERASE = 8
51 Const FT_DEVICE_NOT_OPENED_FOR_WRITE = 9
52 Const FT_FAILED_TO_WRITE_DEVICE = 10
53 Const FT_EEPROM_READ_FAILED = 11
54 Const FT_EEPROM_WRITE_FAILED = 12
55 Const FT_EEPROM_ERASE_FAILED = 13
56 Const FT_EEPROM_NOT_PRESENT = 14
57 Const FT_EEPROM_NOT_PROGRAMMED = 15
58 Const FT_INVALID_ARGS = 16
59 Const FT_NOT_SUPPORTED = 17
60 Const FT_OTHER_ERROR = 18
61
62 'Word Lengths
63 Const FT_BITS_8 = 8
64 Const FT_BITS_7 = 7
65 Const FT_BITS_9 = 9
66
67 'Stop Bits
68 Const FT_STOP_BITS_1 = 0
69 Const FT_STOP_BITS_1_5 = 1
70 Const FT_STOP_BITS_2 = 2
71
72 'Parity
73 Const FT_PARITY_NONE = 0
74 Const FT_PARITY_ODD = 1
75 Const FT_PARITY_EVEN = 2
76 Const FT_PARITY_MARK = 3
77 Const FT_PARITY_SPACE = 4
```

```
78
79  'Flow Control
80  Const FT_FLOW_NONE = &H0
81  Const FT_FLOW_RTS_CTS = &H100
82  Const FT_FLOW_DTR_DSR = &H200
83  Const FT_FLOW_XON_XOFF = &H400
84
85  'Purge rx and tx buffers
86  Const FT_PURGE_RX = 1
87  Const FT_PURGE_TX = 2
88
89  'Flags for FT_OpenEx
90  Const FT_OPEN_BY_SERIAL_NUMBER = 1
91  Const FT_OPEN_BY_DESCRIPTION = 2
92
93  'Flags for FT_ListDevices
94  Const FT_LIST_BY_NUMBER_ONLY = &H80000000
95  Const FT_LIST_BY_INDEX = &H40000000
96  Const FT_LIST_ALL = &H20000000
97
98  'Modem Status
99  Const CTS = &H10
100 Const DSR = &H20
101 Const RI = &H40
102 Const DCD = &H80
103
104 'Modem Status, Line Status
105 Const OE = &H2
106 Const PE = &H4
107 Const FE = &H8
108 Const BI = &H10
109
110 'Bit Modes Set_Bit_Bang
111 Const RESET = &H0
112 Const ASYNCHRONOUS_BIT_BANG = &H1
113 Const MPSSE = &H2
114 Const SYNCHRONOUS_BIT_BANG = &H4
115 Const MCU_HOST_BUS_EMULATION = &H8
116 Const FAST_OPTO_ISOLATED_SERIAL_MODE = &H10
117 Const CBUS_BIT_BANG = &H20
118 Const SINGLE_CHANNEL_SYNCHRONOUS_245_FIFO_MODE = &H40
119
```

```
120 'Baud Rates
121 Const BAUD_300 = 300
122 Const BAUD_600 = 600
123 Const BAUD_1200 = 1200
124 Const BAUD_2400 = 2400
125 Const BAUD_4800 = 4800
126 Const BAUD_9600 = 9600
127 Const BAUD_14400 = 14400
128 Const BAUD_19200 = 19200
129 Const BAUD_38400 = 38400
130 Const BAUD_57600 = 57600
131 Const BAUD_115200 = 115200
132 Const BAUD_230400 = 230400
133 Const BAUD_460800 = 460800
134 Const BAUD_921600 = 921600
135
136 ' Type declaration for EEPROM programming
137 Private Type PROGRAM_DATA
138
139 VendorId As Integer                 '0x0403
140 ProductId As Integer                '0x6001
141 Manufacturer As Long                '32 "FTDI"
142 ManufacturerId As Long              '16 "FT"
143 Description As Long                 '64 "USB HS Serial Converter"
144 SerialNumber As Long                '16 "FT000001" if fixed, or NULL
145 MaxPower As Integer                 ' // 0 < MaxPower <= 500
146 PNP As Integer                      ' // 0 = disabled, 1 = enabled
147 SelfPowered As Integer              ' // 0 = bus powered, 1 = self
        powered
148 RemoteWakeup As Integer             ' // 0 = not capable, 1 = capable
149
150 ' Rev4 extensions:
151 Rev4 As Byte                        ' // true if Rev4 chip, false
        otherwise
152 IsoIn As Byte                       ' // true if in endpoint is
        isochronous
153 IsoOut As Byte                      ' // true if out endpoint is
        isochronous
154 PullDownEnable As Byte              ' // true if pull down enabled
155 SerNumEnable As Byte                ' // true if serial number to be
        used
156 USBVersionEnable As Byte            ' // true if chip uses USBVersion
```

```
157 USBVersion As Integer                ' // BCD (0x0200 => USB2)
158
159 End Type
160
161 'Handler of the interfaces
162 Public LngHandle As Long
163 Public LngHandle2 As Long
164 Public Function OpenUSBRead() As Long
165
166 Dim vbNullString As String
167 Dim strSerialNumber As String * 256
168 Dim strDescription As String * 256
169 Dim strSerialNumber2 As String * 256
170 Dim strDescription2 As String * 256
171 Dim LngHandle2 As Long
172
173 'Open the Interfaces
174 If FT_GetNumDevices(lngNumDevices, vbNullString,
      FT_LIST_BY_NUMBER_ONLY) <> FT_OK Then
175     MsgBox FT_GetNumDevices(lngNumDevices, vbNullString,
          FT_LIST_BY_NUMBER_ONLY)
176     LoggerList.AddItem ("FT_GetNumDevices failed")
177     GoTo CloseHandle
178     Exit Function
179 Else
180     LoggerList.AddItem ("NumDevices: " & lngNumDevices)
181 End If
182
183 If FT_ListDevices(0, strDescription, FT_LIST_BY_INDEX Or
      FT_OPEN_BY_DESCRIPTION) <> FT_OK Then
184     LoggerList.AddItem ("ListDevices failed")
185     Exit Function
186 Else
187     LoggerList.AddItem ("Device Description " & strDescription)
188 End If
189
190 If FT_ListDevices(0, strSerialNumber, FT_LIST_BY_INDEX Or
      FT_OPEN_BY_SERIAL_NUMBER) <> FT_OK Then
191     LoggerList.AddItem ("ListDevices failed")
192     Exit Function
193 Else
194     LoggerList.AddItem ("Serial Number: " & strSerialNumber)
```

```
195 End If
196
197 'Get some information of Device 2 (when available)
198 If FT_ListDevices(1, strDescription2, FT_LIST_BY_INDEX Or
    FT_OPEN_BY_DESCRIPTION) <> FT_OK Then
199     LoggerList.AddItem ("ListDevices failed")
200     'Exit Sub
201 Else
202     LoggerList.AddItem ("Device Description2 " & strDescription2)
203 End If
204
205 If FT_ListDevices(1, strSerialNumber2, FT_LIST_BY_INDEX Or
    FT_OPEN_BY_SERIAL_NUMBER) <> FT_OK Then
206     LoggerList.AddItem ("ListDevices failed")
207     'Exit Sub
208 Else
209     LoggerList.AddItem ("Serial Number2: " & strSerialNumber2)
210 End If
211
212 If FT_OpenEx(strDescription2, FT_OPEN_BY_DESCRIPTION, LngHandle2) <>
    FT_OK Then
213     ErrorCode = FT_OpenEx(strDescription2, FT_OPEN_BY_DESCRIPTION,
        LngHandle2)
214     LoggerList.AddItem "ErrorCode" & ErrorCode
215     LoggerList.AddItem "Open Device by Description 2 Failed"
216     GoTo CloseHandle
217     Exit Function
218 Else
219     LoggerList.AddItem "Device 2 opened"
220     LoggerList.AddItem "LngHandle2 of Device 2:" & LngHandle2
221 End If
222
223 OpenUSBRead = LngHandle2
224
225 CloseHandle: 'label
226
227 ' close the devices
228 If FT_Close(LngHandle) <> FT_OK Then
229     LoggerList.AddItem "Closing the Device 1 failed"
230 End If
231
232 If FT_Close(LngHandle2) <> FT_OK Then
```

```
233     LoggerList.AddItem "Closing of Device 2 failed"
234 End If
235
236 If flFailed = True Then
237     LoggerList.AddItem "Test Failed"
238 End If
239
240 End Function
241 Public Function OpenUSBWrite() As Long
242
243 Dim vbNullString As String
244 Dim strSerialNumber As String * 256
245 Dim strDescription As String * 256
246 Dim strSerialNumber2 As String * 256
247 Dim strDescription2 As String * 256
248 Dim LngHandle2 As Long
249 Dim LngHandle As Long
250
251 'Open the Interfaces
252 If FT_GetNumDevices(lngNumDevices, vbNullString,
    FT_LIST_BY_NUMBER_ONLY) <> FT_OK Then
253     MsgBox FT_GetNumDevices(lngNumDevices, vbNullString,
        FT_LIST_BY_NUMBER_ONLY)
254     LoggerList.AddItem ("FT_GetNumDevices failed")
255     GoTo CloseHandle
256     Exit Function
257 Else
258     LoggerList.AddItem ("NumDevices: " & lngNumDevices)
259 End If
260
261 If FT_ListDevices(0, strDescription, FT_LIST_BY_INDEX Or
    FT_OPEN_BY_DESCRIPTION) <> FT_OK Then
262     LoggerList.AddItem ("ListDevices failed")
263     Exit Function
264 Else
265     LoggerList.AddItem ("Device Description " & strDescription)
266 End If
267
268 If FT_ListDevices(0, strSerialNumber, FT_LIST_BY_INDEX Or
    FT_OPEN_BY_SERIAL_NUMBER) <> FT_OK Then
269     LoggerList.AddItem ("ListDevices failed")
270     Exit Function
```

```
271 Else
272     LoggerList.AddItem ("Serial Number: " & strSerialNumber)
273 End If
274
275 'open device 1 FT245BM by Description
276
277 If FT_OpenEx(strDescription, FT_OPEN_BY_DESCRIPTION, LngHandle) <>
    FT_OK Then
278     LoggerList.AddItem "Open Device by Description 1 Failed"
279     GoTo CloseHandle
280     'Exit Sub
281 Else
282     LoggerList.AddItem "Device 1 opened"
283     LoggerList.AddItem "LngHandle of Device 1:" & LngHandle
284 End If
285
286 OpenUSBWrite = LngHandle
287
288 '#####################################################################
289 '                              CLOSE HANDLE
290 '#####################################################################
291
292 CloseHandle: 'label
293
294 ' close the devices
295
296 If FT_Close(LngHandle) <> FT_OK Then
297     LoggerList.AddItem "Closing the Device 1 failed"
298     'Write #1, "Closing the Device 1 failed"
299 End If
300
301 If FT_Close(LngHandle2) <> FT_OK Then
302     LoggerList.AddItem "Closing of Device 2 failed"
303 End If
304
305 If flFailed = True Then
306     LoggerList.AddItem "Test Failed"
307 End If
308
309 End Function
310
311 Private Sub Close_Click()
```

```
312
313 If FT_Close(LngHandle) <> FT_OK Then
314     LoggerList.AddItem "Closing the Device 1 failed"
315     'Write #1, "Close D1 not OK!"
316 Else
317     LoggerList.AddItem "Close Device1 OK"
318     'Write #1, "Close D1 OK!"
319 End If
320
321 If FT_Close(LngHandle2) <> FT_OK Then
322     LoggerList.AddItem "Closing of Device 2 failed"
323     'Write #1, "Close D2 not OK!"
324 Else
325     LoggerList.AddItem "Close Device2 OK!"
326     'Write #1, "Close D2 OK!"
327 End If
328
329 If flFailed = True Then
330     LoggerList.AddItem "Test Failed"
331 End If
332
333 End
334
335 Close #1
336 Beep
337
338 End Sub
339
340 Private Sub Command1_Click()
341
342 Text2.Text = ""
343
344 End Sub
345
346 'Private Function GetEpochTimeFromString(ByVal BufferString As String)
        As Long
347 Public Sub Open_Click()
348
349 Dim vbNullString As String
350 Dim strSerialNumber As String * 256
351 Dim strDescription As String * 256
352 Dim strSerialNumber2 As String * 256
```

```
353 Dim strDescription2 As String * 256
354
355 'Open the Interfaces
356 If FT_GetNumDevices(lngNumDevices, vbNullString,
        FT_LIST_BY_NUMBER_ONLY) <> FT_OK Then
357     MsgBox FT_GetNumDevices(lngNumDevices, vbNullString,
            FT_LIST_BY_NUMBER_ONLY)
358     LoggerList.AddItem ("FT_GetNumDevices failed")
359     GoTo CloseHandle
360     Exit Sub
361 Else
362     LoggerList.AddItem ("NumDevices: " & lngNumDevices)
363 End If
364
365 'FT_ListDevices with Description
366 If FT_ListDevices(0, strDescription, FT_LIST_BY_INDEX Or
        FT_OPEN_BY_DESCRIPTION) <> FT_OK Then
367     LoggerList.AddItem ("ListDevices failed")
368     Exit Sub
369 Else
370     LoggerList.AddItem ("Device Description " & strDescription)
371 End If
372
373 'FT_ListDevices with unique Serial Number
374 If FT_ListDevices(0, strSerialNumber, FT_LIST_BY_INDEX Or
        FT_OPEN_BY_SERIAL_NUMBER) <> FT_OK Then
375     LoggerList.AddItem ("ListDevices failed")
376     Exit Sub
377 Else
378     LoggerList.AddItem ("Serial Number: " & strSerialNumber)
379 End If
380
381 'Get some information of Device 2 (when available)
382
383 'If FT_ListDevices(1, strDescription2, FT_LIST_BY_INDEX Or
        FT_OPEN_BY_DESCRIPTION) <> FT_OK Then
384 '    LoggerList.AddItem ("ListDevices failed")
385     'Exit Sub
386 'Else
387 '    LoggerList.AddItem ("Device Description2 " & strDescription2)
388 'End If
389
```

```
390 'If FT_ListDevices(1, strSerialNumber2, FT_LIST_BY_INDEX Or
      FT_OPEN_BY_SERIAL_NUMBER) <> FT_OK Then
391  '    LoggerList.AddItem ("ListDevices failed")
392      'Exit Sub
393 'Else
394 '    LoggerList.AddItem ("Serial Number2: " & strSerialNumber2)
395 'End If
396
397 'open device 1 FT245BM by Description
398
399 'If FT_OpenEx(strDescription, FT_OPEN_BY_DESCRIPTION, LngHandle) <>
      FT_OK Then
400  '    LoggerList.AddItem "Open Device by Description 1 Failed"
401   '   GoTo CloseHandle
402   '   Exit Sub
403 'Else
404  '    LoggerList.AddItem "Device 1 opened"
405   '   LoggerList.AddItem "LngHandle of Device 1:" & LngHandle
406
407 'End If
408
409 'open device 2 FT245RL by Description
410 If FT_OpenEx(strDescription2, FT_OPEN_BY_DESCRIPTION, LngHandle2) <>
      FT_OK Then
411     ErrorCode = FT_OpenEx(strDescription2, FT_OPEN_BY_DESCRIPTION,
          LngHandle2)
412     LoggerList.AddItem "ErrorCode" & ErrorCode
413     LoggerList.AddItem "Open Device by Description 2 Failed"
414     GoTo CloseHandle
415     Exit Sub
416 Else
417     LoggerList.AddItem "Device 2 opened"
418     LoggerList.AddItem "LngHandle2 of Device 2:" & LngHandle2
419 End If
420
421 CloseHandle: 'label
422
423 ' close the devices
424 If FT_Close(LngHandle) <> FT_OK Then
425     LoggerList.AddItem "Closing the Device 1 failed"
426 End If
427
```

```
428 If FT_Close(LngHandle2) <> FT_OK Then
429     LoggerList.AddItem "Closing of Device 2 failed"
430 End If
431
432 If flFailed = True Then
433     LoggerList.AddItem "Test Failed"
434 End If
435
436 End Sub
437 Public Sub Start_Click()
438
439 Open "D:\vb-versuch\Werte.txt" For Output As #1
440 'When the Start Button is clicked, the Interface is opened by the
    functions of the D2xxx.dll.
441 'After that, the program begins to read out the FIFO of the interface,
     if it holds any bytes.
442 'When reading is done, the program calculates the Epoch Times from the
     bytes and decides if the epoch time holds any
443 'information or not. If the epoch time represents information, the
    information is displayed in a textfield. If the epoch
444 'time holds no information, no information (e.g. letter or number)
    will be displayed ;-)
445 'To open the FT245RL correctly, the program writes two bytes into the
    FIFO (First In First Out) of the FT245RL and after
446 'that it resets the Interface to clear the two transmitted bytes. That
    's necessary to be sure that no problems occure with the
447 'Combined Driver Model
448
449 Dim lngBytesWritten As Long
450 Dim strReadBuffer As String * 63488 'buffer which holds the
    information of the FIFO
451 Dim EpochTime As Variant
452 Dim LenghOfString As Long
453 Dim lngBytesRead As Long
454 Dim lngTotalBytesRead As Long
455 Dim strLoggerBuffer As String
456 Dim flFailed As Boolean
457 Dim flTimedout As Boolean
458 Dim flFatalError As Boolean
459 Dim ftStatus As Long
460 Dim lngNumDevices As Long
461 Dim strSerialNumber As String * 256
```

```
462 Dim strDescription As String * 256
463 Dim Counter As Long
464 Dim lngRxBytes As Long
465 Dim ErrorCode As Long
466 Dim mStartTime As Long
467 Dim ByteArray(1 To 63488) As Variant
468 Dim ArrayZero(1 To 100) As Long
469
470 Dim InformationArray(1 To 15872) As Long 'Array that holds the Epoch
    Times
471 Dim BytesToRead As Long
472 Dim Byte_0 As Byte
473 Dim Byte_1 As Byte
474 Dim Byte_2 As Byte
475 Dim Byte_3 As Byte
476 Dim Index As Long 'Index of InformationArray
477 Dim Nanoseconds As Variant
478 Dim StartTime As Variant
479 Dim EpochCounter As Long
480 Dim ReferenceSeconds As Variant
481 Dim Difference As Variant
482 Dim AsciiN As Double
483 Dim RelevantValue As Variant
484 Dim NullValue As Long
485 Dim ReferenceValue As Variant
486 Dim Letter As Variant
487 Dim AsciiValue As Variant
488 Dim StringToSend As String
489 Dim BytesReadable As Long
490 Dim TestString As Variant
491 Dim NullCounter As Integer
492 Dim EpochTimesIndexCounter As Long
493 Dim EqualCounter As Integer
494 Dim SumOfMeanValue As Long
495 Dim DoEventsCounter As Long
496 Dim OldNullValue As Long
497 Dim TxT As Variant
498 Dim Testi As Variant
499
500 Const JITTER = 75 'The Jitter of the Laser is +-15ns
501 Const NFACTOR = 80 'The N-Factor of the coded information is 40ns.
502 'Const START_TEXT = 2
```

```
503 'Const END_OF_TEXT = 4
504
505 LenghOfString = 2
506 StringToSend = "HA"
507
508 Text1.Text = vbNullString 'Erase the message textfield
509
510 Const BYTES_TO_READ = 4096 'we have to read in 64 Byte Blocks
511 'Open "C:\Test.txt" For Output As #1 'create a testfile called Test.
        txt for
512 'writing the epoch times
513 EpochCounter = 0
514
515 '-----------------------------------------------------------------
516 'Open the Interfaces 'device 1 and device 2 with functions of d2xxx.
        dll
517 '-----------------------------------------------------------------
518 'FT_STATUS FT_Read (FT_HANDLE ftHandle, LPVOID lpBuffer, DWORD
        dwBytesToRead,
519 'LPDWORD lpdwBytesReturned)
520 'If FT_GetNumDevices is not equal to FT_OK then program will terminate
521 'Otherwise the parameter lngNumDevices holds the number of devices
522
523 If FT_GetNumDevices(lngNumDevices, vbNullString,
        FT_LIST_BY_NUMBER_ONLY) <> FT_OK Then
524     MsgBox FT_GetNumDevices(lngNumDevices, vbNullString,
            FT_LIST_BY_NUMBER_ONLY)
525     LoggerList.AddItem ("FT_GetNumDevices failed")
526     'Write #1, "FT_GetNumDevices failed"
527     GoTo CloseHandle
528     Exit Sub
529 Else
530     LoggerList.AddItem ("NumDevices: " & lngNumDevices)
531     'Write #1, lngNumDevices
532 End If
533
534 '#################################################################
535 'FT_STATUS FT_ListDevices (PVOID pvArg1, PVOID pvArg2, DWORD dwFlags)
536 'Returnes a list of connected devices. If FT_ListDevices is not equal
        to FT_OK,
537 'the program will terminate. Otherwise the parameter strDescription
        holds the discription of
```

```
538 'the device
539
540 If FT_ListDevices(0, strDescription, FT_LIST_BY_INDEX Or
        FT_OPEN_BY_DESCRIPTION) <> FT_OK Then
541     LoggerList.AddItem ("ListDevices failed")
542     'Write #1, "ListDevices failed"
543     Exit Sub
544 Else
545     LoggerList.AddItem ("Device Description" & strDescription)
546     'Write #1, strDescription
547 End If
548
549 '#####################################################################
550 'The same, but the parameter strSerialNumber holds the serial number
        of the device
551 If FT_ListDevices(0, strSerialNumber2, FT_LIST_BY_INDEX Or
        FT_OPEN_BY_SERIAL_NUMBER) <> FT_OK Then
552     LoggerList.AddItem ("ListDevices failed")
553     'Write #1, "ListDevices failed"
554     Exit Sub
555 Else
556     LoggerList.AddItem ("Serial Number:" & strSerialNumber2)
557     'Write #1, strSerialNumber
558 End If
559
560 '#####################################################################
561 'Open device FT245RL with Open_By_Description
562 If FT_OpenEx(strDescription, FT_OPEN_BY_DESCRIPTION, LngHandle) <>
        FT_OK Then
563     ErrorCode = FT_OpenEx(0, FT_OPEN_BY_DESCRIPTION, LngHandle)
564     LoggerList.AddItem ("Open_EX failed")
565     LoggerList.AddItem ("ErrorCode: " & ErrorCode)
566     Exit Sub
567 Else
568     LoggerList.AddItem ("FT_OpenEx Successful, FT245RL is open now :-)
            ")
569 '   Write #1, "FT_OpenEx Successful, FT245RL is open now :-)"
570 End If
571
572 Sleep 30 'Wait 30ms
573
574 '#####################################################################
```

```
575 'Write 2 Bytes on the FT245RL
576 If FT_Write(LngHandle, StringToSend, LenghOfString, lngBytesWritten)
      <> FT_OK Then
577     LoggerList.AddItem ("FT_Write not OK")
578     LoggerList.AddItem ("Bytes Written" & lngBytesWritten)
579     GoTo CloseHandle
580     Exit Sub
581 Else
582     LoggerList.AddItem ("FT_Write OK")
583     LoggerList.AddItem ("Bytes Written" & lngBytesWritten)
584
585 End If
586
587 Sleep 30 'Wait 30ms
588
589 '#################################################################
590 'Reset FT245RL
591 If FT_ResetDevice(LngHandle) <> FT_OK Then
592     LoggerList.AddItem ("Reset not OK")
593     GoTo CloseHandle
594     Exit Sub
595 Else
596     LoggerList.AddItem ("Reset Ok, Device resetted")
597
598 End If
599
600 '#################################################################
601 '-----------------------ENDLESS LOOP-----------------------------
602 '#################################################################
603 'Please do not change ;-) - here begins the main algorithm
604 Do
605     If (DoEventsCounter Mod 10000) = 0 Then
606         DoEvents
607     End If
608
609     If DoEventsCounter = 65500 Then
610         DoEventsCounter = 0
611     End If
612 '-----------------------------------------------------------------
613     DoEventsCounter = DoEventsCounter + 1 'increases the
          DoEventscounter
614     Index = 0
```

```
615
616     If FT_GetQueueStatus(LngHandle, lngRxBytes) <> FT_OK Then
617         LoggerList.AddItem "GetQueueStatus failed!"
618         LoggerList.ListIndex = LoggerList.ListCount - 1
619         LoggerList.Selected(LoggerList.ListIndex) = False
620         GoTo CloseHandle
621         Exit Sub
622     Else
623         On Error Resume Next
624     End If
625
626     If lngRxBytes >= BYTES_TO_READ Then
627         BytesReadable = lngRxBytes - lngRxBytes Mod 4 'To be sure that
                we only read full epoch values
628
629         If FT_Read(LngHandle, strReadBuffer, BytesReadable,
                lngBytesRead) <> FT_OK Then 'an error was detected
630             ErrorCode = FT_Read(LngHandle, MyBytes, BYTES_TO_READ,
                    lngBytesRead)
631             GoTo CloseHandle
632         Else
633             LoggerList.AddItem "Reading Ok :-)" 'display "Reading Ok"
                    if reading successful
634             LoggerList.AddItem "Bytes read:" & lngBytesRead 'how many
                    bytes were read
635             List9.AddItem lngBytesRead
636
637             For Counter = 1 To BytesReadable - 3 Step 4
638                 Byte_0 = Asc(Mid$(strReadBuffer, Counter, 1)) '
                        function "MID" which analyzes strReadBuffer
                        element by element
639                 Byte_1 = Asc(Mid$(strReadBuffer, Counter + 1, 1))
640                 Byte_2 = Asc(Mid$(strReadBuffer, Counter + 2, 1))
641                 Byte_3 = Asc(Mid$(strReadBuffer, Counter + 3, 1))
642                 EpochTime = CDec(Byte_0) + CDec(Byte_1 * 2 ^ 8) + CDec
                        (Byte_2 * 2 ^ 16) + CDec(Byte_3 * 2 ^ 24) '
                        Computes the Epoch Times. CDec is for datatype
                        DECIMAL
643                 Nanoseconds = CDec(EpochTime) * 5
644                 RelevantValue = Nanoseconds - Int(Nanoseconds /
                        100000) * 100000 'RelevantValue = RelevantValue
                        Mod 100000
```

```
645                 Write #1, RelevantValue
646                 Write #1, "----------"
647                 Index = Index + 1 'Increase the Index Value
648                 InformationArray(Index) = RelevantValue
649             Next Counter
650
651             'Calculation of the NullValue
652             EpochTimesIndexCounter = 0
653             ArrayZeroIndex = 0
654             EqualCounter = 0
655             MeanValue = 0
656             SumOfMeanValue = 0
657             OldNullValue = NullValue
658
659             Do While EqualCounter < 100 'Do as long as EqualCounter is
                    less than 100, because in this case, there's no
                    NullValue
660                 If EpochTimesIndexCounter = BytesReadable / 4 - 1 And
                        EqualCounter = 99 Then 'if program reaches the
                        second last position of the array
661                     For Counter = 1 To 99 Step 1 'calculation of the
                            mean value of the array
662                         SumOfMeanValue = SumOfMeanValue + ArrayZero(
                                Counter) 'sum up the numeric elements of
                                the array.
663                     Next Counter
664
665                 NullValue = SumOfMeanValue / 99
666                     If NullValue = InformationArray(1) Or Abs(
                            NullValue - InformationArray(1)) <= JITTER
                            Then
667                         EqualCounter = EqualCounter + 1
668                         ArrayZero(100) = InformationArray(1)
669                         'Exit Do
670                     Else
671                         NullValue = OldNullValue 'if there aren't 100
                                equal values in the array, program should
                                use the old NullValue
672                         List10.AddItem NullValue
673                         GoTo NULLVALUEFOUND 'Nullvalue found. In this
                                case the program continues on next label
674                     End If
```

```
675                 End If
676
677                 If EpochTimesIndexCounter = BytesReadable / 4 - 1 And
                        EqualCounter <> 99 Then
678                 'in this case, the InformationArray has been analyzed
                        until the first position, but
679                 'there aren't 99 equal values until yet.The program
                        takes in this case the old
680                 'NullValue (OldNullValue)
681                     NullValue = OldNullValue
682                     List10.AddItem NullValue
683                     GoTo NULLVALUEFOUND 'jump to label NULLVALUEFOUND,
                            if NullValue = OldNullValue
684                 End If
685
686                 Difference = Abs(InformationArray(BytesReadable / 4 -
                        EpochTimesIndexCounter) - Abs(InformationArray(
                        BytesReadable / 4 - EpochTimesIndexCounter - 1)))
687                 'Write #1, Difference
688                 EpochTimesIndexCounter = EpochTimesIndexCounter + 1
689
690                 If Difference = 0 Or Difference <= JITTER Then 'Jitter
691                     ArrayZeroIndex = ArrayZeroIndex + 1
692                     ArrayZero(ArrayZeroIndex) = InformationArray(
                            BytesReadable / 4 - EpochTimesIndexCounter)
693                     EqualCounter = EqualCounter + 1
694                     Write #1, EqualCounter
695                     'Write #1, "----------"
696                 Else
697                     EqualCounter = 0
698                     ArrayZeroIndex = 0
699                     'Write #1, EqualCounter
700                 End If
701             Loop
702
703         SumOfMeanValue = 0
704
705         If EqualCounter >= 100 Then
706         'There are 100 equal EpochValues together and now we
707         'can compute the NullValue
708             For Counter = 1 To 100 Step 1
709                 SumOfMeanValue = SumOfMeanValue + ArrayZero(Counter)
```

```
710             Next Counter
711
712             NullValue = SumOfMeanValue / 100
713             List10.AddItem NullValue
714         End If
715
716 NULLVALUEFOUND:
717
718             For Counter = 1 To (BytesReadable / 4) Step 1
719                 Difference = (InformationArray(Counter) - NullValue)
720                 '_____________________________________________________
721                 If Difference < 0 Then 'if the difference is negative,
                        tbe carry has to be added!!!
722                     Difference = ((InformationArray(Counter) + 100000)
                            - NullValue)
723                 End If
724                 '_____________________________________________________
725                 If Difference <> 0 And Difference > JITTER Then
726                     AsciiValue = Round(Difference / NFACTOR) 'We have
                            to round the value
727                     Text1.SelText = Chr$(AsciiValue)
728                     On Error Resume Next
729                 End If
730             Next Counter
731         End If
732     End If
733 Loop 'End Do
734
735 'CloseHandle Label
736 'If an error occures (e.g. by opening the interface), the program will
        jump to label "CloseHandle"
737 'Here, the interface will be closed by special functions, which
       depends to the d2xxx.dll
738
739 CloseHandle:
740
741 If FT_Close(LngHandle) <> FT_OK Then
742     LoggerList.AddItem "Closing the Device 1 failed"
743 End If
744
745 If flFailed = True Then
746     LoggerList.AddItem "Test Failed"
```

```
747 End If
748
749 End Sub
750 Private Sub Write_Click()
751
752 'Sends messages to the interface (FT245BL). The messages are written
      into a textbox called "Message".
753 'The content of "Message" is stored into the variable "StringToSend".
      The lengh of the String is stored
754 'into the variable "LenghOfString".
755 'FT_Write is a function of the FTD2xxx.dll, which allows to send
      strings to FT245RL.
756 'Syntax of FT_Write:
757 'FT_Write(ftHandle, TxBuffer, sizeof(TxBuffer), &BytesWritten)
758 'FT_Write returns an Error Code after calling the function. If the
      returned Error-Code is not equal to FT_OK (FT_OK =1)
759 'an error occured. If no error occured, the return value is FT_OK. The
       variable "lngBytesWritten" holds the number
760 'of the written bytes. If lngBytesWritten is equal to StringToSend -
      everything is OK!
761
762 Dim StringToSend As String
763 Dim LenghOfString As Long
764 Dim lngBytesWritten As Long
765
766 StringToSend = Message.Text 'stores the text of the message field into
       "StringToSend"
767 LenghOfString = Len(StringToSend)
768
769 If FT_Write(LngHandle, StringToSend, LenghOfString, lngBytesWritten)
      <> FT_OK Then
770     LoggerList.AddItem "Writing Not OK!"
771
772 Else
773     LoggerList.AddItem "Bytes Written" & lngBytesWritten
774 End If
775
776 End Sub
```