MEE09:49

A Development Platform for Microcontroller STM32F103

This thesis is presented as part of Degree of
Master of Science in Electrical Engineering

Blekinge Institute of Technology
March 2009

By: Ehsan Tehrani

Blekinge Institute of Technology
School of Engineering
Department of Signal Processing
Supervisor: Prof. Jörgen Nordberg and Prof. Ingvar Gustavsson

TABLE OF CONTENTS

1 INTRODUCTION…………………………………………………………………..6
2 MICROCONTROLLER (MCU)………………………………………………………8
2.1 MCU Overview…………………………………………………………………….8
2.1.1 Family group of STM32F103………………………………………………..9
2.2 Components of STM32F103……………………………………………………10
2.2.1 System Architecture………………………………………………………..11
2.2.2 ARM Cortex-M3 core……………………………………………………....11
2.2.3 Memory system……………………………………………………………..12
2.2.4 Nested Vect It Ctrl (NVIC)………………………………………………....12
2.2.5 Ext. ITs (EXTI)……………………………………………………………..12
2.2.6 Clock system (SYSCLK)…………………………………………………...12
2.2.7 Startup clock………………………………………………………………..13
2.2.8 Boot modes…………………………………………………………………15
2.2.9 Power Supply………………………………………………………………15
2.2.10 DMA or Direct Memory Access……………………………………………15
2.2.11 Real Time Clock (RTC)……………………………………………………15
2.2.12 General Purpose Timers (TIMx)……………………………………………15
2.2.13 Advanced Control Timer (TIM1)…………………………………………..15
2.2.14 I²C……………………………………………………………………………15
2.2.15 Universal Synchronous / Asynchronous Receive Transmitter (USART)……….15
2.2.16 Serial Peripheral Interface (SPI)…………………………………………...15
2.2.17 Controller Area Network (CAN)…………………………………………..15
2.2.18 Universal Serial Bus (USB)………………………………………………..15
2.2.19 General Purpose Input / Outputs (GPIO)…………………………………..15
2.2.20 Analogue to Digital Convertor (ADC)……………………………………16
2.3 Microcontroller Configuration……………………………………………..16
2.4 STM32USB Preparation…………………………………………………...19
3 LabVIW…………………………………………………………………………..27
3.1 Virtual Instrument…………………………………………………………27
3.1.2 Front Panel…………………………………………………………………28
3.1.3 Block Diagram………………………………………………………………..28
3.1.4 Controls, Indicators and Palettes…………………………………………...29
3.1.5 LabVIEW Application……………………………………………………..30
3.2 LabVIEW Requirements for Thesis……………………………………..31
3.2.1 The USB Driver……………………………………………………………31
3.2.2 LabVIEW Functions………………………………………………………31
3.2.2.1 VISA Functions…………………………………………………………31
3.2.2.2 General Functions…..32
4 THESIS SOLUTION..………………………………………………………………34
4.1 Microcontroller Solution…………………………………………………...34
4.2 PC LabVIEW Solution…………………………….....................................37
4.3 Testing the Project………………………………………………………..40
5 CONCLUSION……………………………………………………………………..42
6 APPENDIX

LIST OF Abbreviations

A

ADC: *Analogue to Digital Convertor*

ADCCLK: *ADC Clock*

AHB: *Advanced High Performance Bus*

APB: *Advanced Peripheral Bus*

API: *Application Programming Interface*

C

CAN: *Controller Area Network*

CPU: *Central Processing Unit*

D

DAQ: *Data Acquisition*

DMA: *Direct Memory Access*

DIMPS: *Distributed Integrated message Processing System*

DSP: *Digital Signal Processor*

F

FPGAs: *Field Programmable Gate Arrays*

FSMC: *Flexible Static Memory Controller*

G

GPIB: *General Purpose Interface Bus*

GPIO: *General Purpose Input / Outputs*

GPS: *Global Positioning System*

H

HIS: *High Speed Internal*

HSE: *High Speed External*

I

I²Cs: *Inter-Integrated Circuit*

I/O: *Input/Output*

L

LabVIEW: *Laboratory Virtual Instrument Engineering Workbench*

LED: *Light-Emitting Diode*

LSI RC: *Low Speed Internal Resistor Capacitor oscillator*

LSE: *Low Speed External*

M

MCU: *MICROCONTROLLER*

O

One (1) Dimensional: *1D*

P

PC: Personal Computer

PCLK1: *Internal APB1 clock*

PCLK2: *Internal APB2 clock*

PDR: *Power Down Reset*

PLL: *Phase Locked Loop*

PLLCLK: *PLL Clock*

POR: *Power On Reset*

PVD: *Programmable Voltage Detector*

PWM: *Pulse Width Modulation*

R

RAM: *Random Access Memory*

RS-232: *A standard for computer serial ports*

RTC: *Real Time Clock*

S

SDIO: *Secure Digital Input Output*

SRAM: *Static Random Access Memory*

SPI: *Serial Peripheral Interface*

SYSCLK: *System Clock Source*

T

TIMx: *General Purpose and Advanced Control Timers*

U

USART: *Universal Synchronous / Asynchronous Receive Transmitter*

USB: *Universal Serial Bus*

V

VI: *Virtual Instrument*

VISA: *Virtual Instrument Software Architecture*

1 INTRODUCTION

Today computers, micro controllers and software can be used together in order to create many useful engineering applications. There are many advantages to use them together. They can be applied to calculate very complicated mathematical operations accurately. They are also cheap, can be prepared and used conveniently in many fields like research, education, industrial application, etc.

This thesis is regarding an interface project which consists of microcontroller, computer and software and it is tried to prepare appropriate software for the microcontroller and computer in order to communicate with each other via USB (Universal Serial Bus). The microcontroller software is written by C program and LabVIEW program is applied to make a LabVIEW application for PC.

The following part describes the work precisely in order give more information regarding project operation. The microcontroller STM32F103 receives an Analogue signal from signal generator. The ADC (Analogue to Digital Convertor) converts the input analogue signal to the Digital Samples (making digital samples of analogue signal). Each digital sample should be saved to make a pack or an array of digital samples. Then this array of digital samples should be sent to computer (LabVIEW: Laboratory Virtual Instrument Engineering Workbench) via USB port.

PC running LabVIEW (LabVIEW Application) is applied to resample the analogue signal from its samples and display it on screen. LabVIEW Application also modifies the sampling frequency (Fs) inside the microprocessor by sending a new sampling frequency to the STM32 development board through USB. Please consider the figure 1 which it is taken from whole project at the laboratory of Baltic Engineering Company.

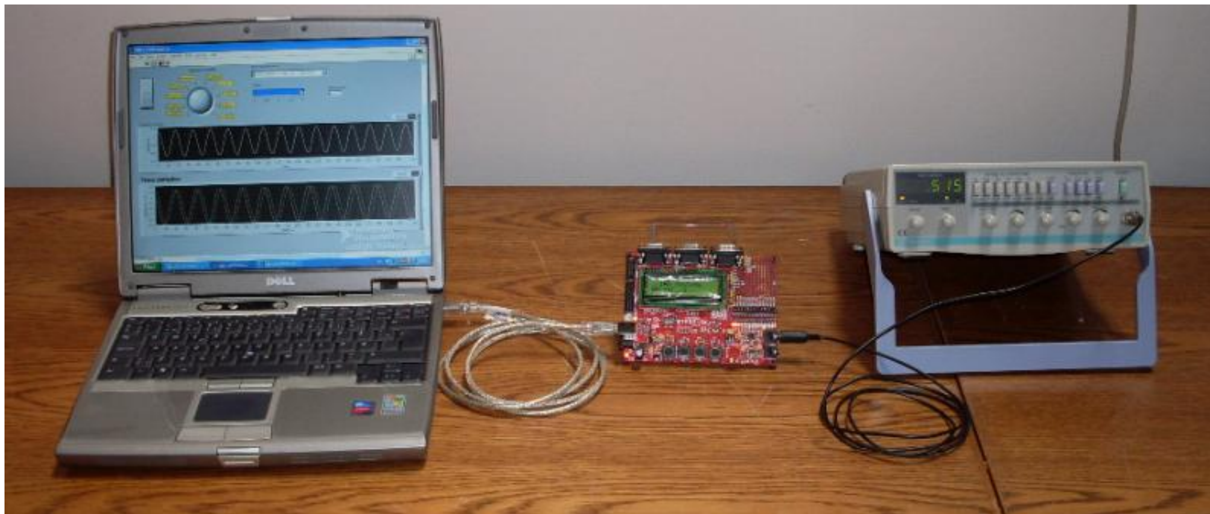

Figure1: Picture of the project

This figure 1 shows that an analogue signal is sent to the microcontroller STM32 development board. A program should be prepared for the microcontroller STM32F103xx in

order to convert analogue signal to the digital signal (as a pack of 32 bits of samples).

As it is obvious in figure1, this package of samples is sent to the PC via the USB port and PC reconstructs the analogue signal by using LabVIEW program and received digital samples.

Finally, this thesis demonstrates how to prepare proper software for the microcontroller STM32F103 in order to create a pack of 32bits samples, how to have the USB communication to sent date to PC and how to make PC running LabVIEW by using LabVIEW program.

1.1 Thesis Background

Baltic Engineering AB is a company designing electronic devices. They have designed and made different kind of electronic products by micro controllers. Some of these products are able to communicate with PC via RS-232 serial port (RS-232 is a standard for computer serial ports). Baltic company decided to design and make a board containing the micro controller STM32F103 and use USB port instead of RS-232 serial port in order to make communication with PC. This board should be able to use the Analogue to Digital Converter of the micro controller to sample the input signals and send a pack of samples to a PC running LabVIEW via USB port. Further more; it is possible to send data from PC running LabVIEW to the micro controller through USB port.

1.2 Thesis Organization

The second chapter of this report describes the microcontroller and its components. The third chapter consists of LabVIEW description and tries to introduce LabVIEW to readers. The fourth chapter mentions solution for this thesis work. The last chapter is conclusion.

2 **MICROCONTROLLER (MCU)**

The STM32F103xx Microcontroller is produced by *ST Company* and it is applied for this thesis work. Baltic Engineering Co. has decided to use the microcontroller STM32 f103 in the most projects because this MCU STM32 f103 has many high level features in compare of other microcontrollers. It is one of the best in class 32 bit MCU, best performance to control and connectivity in electronics projects, it is able to perform in DSP (Digital Signal Processor) solutions (High frequency performance), has low power application in order to save power for system, the speed of peripheral is increased for the better performance etc.

2.1 MCU Overview

The family of STM32F103xx Microcontrollers consists of ARM Cortex-M3 32-bit RISC core, high speed embedded memories (Flash memory is up to 128 Kbytes and Static Random Access Memory (SRAM) is up to 20 Kbytes), I/Os (Input/Output) and peripherals which they are cooperating together by connecting to two APB (Advanced Peripheral Bus) buses. The STM32F103xx microcontroller includes many peripherals as well as two 12-bits ADCs, an Advanced Control Timer, three General Purpose 16-bit timers and also a PWM (Pulse Width Modulation) timer. It is also provided by two I²Cs (Inter-Integrated Circuit) and SPIs (Serial Peripheral Interface), three USARTs (Universal Synchronous / Asynchronous Receive Transmitter), an USB and a CAN (Controller Area Network) as the communication interface system. Figure 2 presents the pinout for the STM32F103 family which is used in this project.

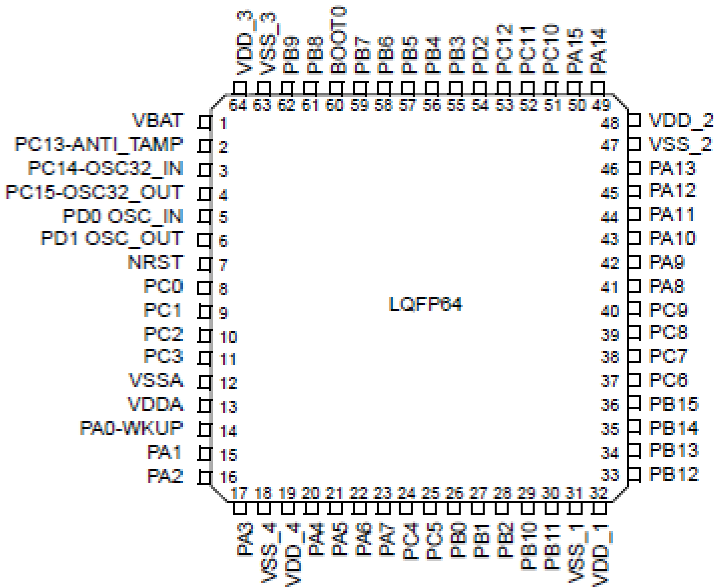

Figure 2: STM32F103 Performance line Pinout

The medium-density microcontroller is used and it has 64 pins. This microcontroller family consists of three ports which PA, PB and PC are MCU ports and each port has 16 pins as I/Os. V_{SS}, V_{DD} and V_{BAT} are used to bias microcontroller by using external power supply.

2.1.1 Family group of STM32F103

The STM32F103xx family microcontrollers are divided into three groups:

- **Low-density:** The STM32F103x4 and STM32F103x6 are Low-density devices.
- **Medium-density:** The STM32F103x8 and STM32F103xB are Medium-density devices.
- **High-density:** The STM32F103xC, STM32F103xD and STM32F103xE are High-density devices.

<table>
<tr><th rowspan="3">Pinout</th><th colspan="2">Low-density devices</th><th colspan="2">Medium-density devices</th><th colspan="3">High-density devices</th></tr>
<tr><th>16 KB Flash</th><th>32 KB Flash(1)</th><th>64 KB Flash</th><th>128 KB Flash</th><th>256 KB Flash</th><th>384 KB Flash</th><th>512 KB Flash</th></tr>
<tr><th>6 KB RAM</th><th>10 KB RAM</th><th>20 KB RAM</th><th>20 KB RAM</th><th>48 KB RAM</th><th>64 KB RAM</th><th>64 KB RAM</th></tr>
<tr><td>144</td><td></td><td></td><td></td><td></td><td colspan="3" rowspan="3">5 × USARTs
4 × 16-bit timers, 2 × basic timers
3 × SPIs, 2 × I2Ss, 2 × I2Cs
USB, CAN, 2 × PWM timers
3 × ADCs, 1 × DAC, 1 × SDIO
FSMC (100 and 144 pins)</td></tr>
<tr><td>100</td><td></td><td></td><td colspan="2" rowspan="3">3 × USARTs
3 × 16-bit timers
2 × SPIs, 2 × I2Cs, USB, CAN, 1 × PWM timer
2 × ADC</td></tr>
<tr><td>64</td><td colspan="2" rowspan="3">2 × USARTs
2 × 16-bit timers
1 × SPI, 1 × I2C, USB, CAN, 1 × PWM timer
2 × ADCs</td></tr>
<tr><td>48</td><td></td><td></td><td></td></tr>
<tr><td>36</td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Table 1: STM32F103 Family

These three groups are made according to the feature of the microcontroller STM32F103xx family members.

Low-density microcontrollers have lower Flash memory and RAM (Random Access Memory), less timer and peripherals in compare to the other two groups. Thus, Medium-density and High-density consist of higher Flash memory, RAM capacities and also have more additional peripherals.

Low-density families include 16 KB to 32 KB Flash memory and 6 KB to 10 KB RAM capacities. They consist of 1 × CAN, 1 × USB, 1 × PWM timer, 1 × I²C, 1 × SPI and 2 × ADCs, 2 × USARTs, and 2 × 16-bits timers. The differences between Low-density families are regarding the number of their pinout packages. There are three kinds of pinout packages which they made up 36, 48 and 64 pins. Their Flash memory is improved from 64 KB to 128 KB and RAM capacity is 20 KB.

Medium-density families have more properties in compare to Low-density families. The number of peripherals and pinouts of them are improved. They have 1 × CAN, 1 × USB, 1 × PWM timer, 2 × I²C, 2 × SPI, 2 × ADCs, 3 × USARTs, and 3 × 16-bits timers. They also have three kinds of pinout packages that consist of 48, 64 and 100 pins.

High-density families are completed more than others and have more peripherals. They made up these peripherals, such as 1 × CAN, 1 × USB, 1 × PWM timer, 2 × I²Ss (SPI), 2 × I²C,

3 × SPI, 2 × ADCs, 1 × DAC, 5 × USARTs, 2 × basic timers, 4 × 16-bits timers, 1 × SDIO (Secure Digital Input Output), and 1 ×FSMC (Flexible Static Memory Controller).

For this project the STM32F103xB microcontroller (Medium-density device) is applied and it should be mentioned that all STM32 families' performance line is fully compatible.

Table 1 introduces briefly all features for the STM32F103xx microcontrollers briefly. These three categories for the STM32F103xx microcontroller is useful for designers which they can select suitable family for their projects. This large verity of the STM32 causes to reduce the price of products and also products can have a better shape and size.

The STM32F103xx microcontrollers are so convenient in order to use for different kind of applications, as well as, Industrial, Control, Medical, PC peripherals gaming, GPS (Global Positioning System), Video intercom, Alarm system, etc.

2.2 Components of STM32F103 MCU

This section explains the microcontroller core, memories, I/Os and peripherals which are introduced at the block diagram 3. There are a brief explanations for the STM32F103 microcontroller parts in the below.

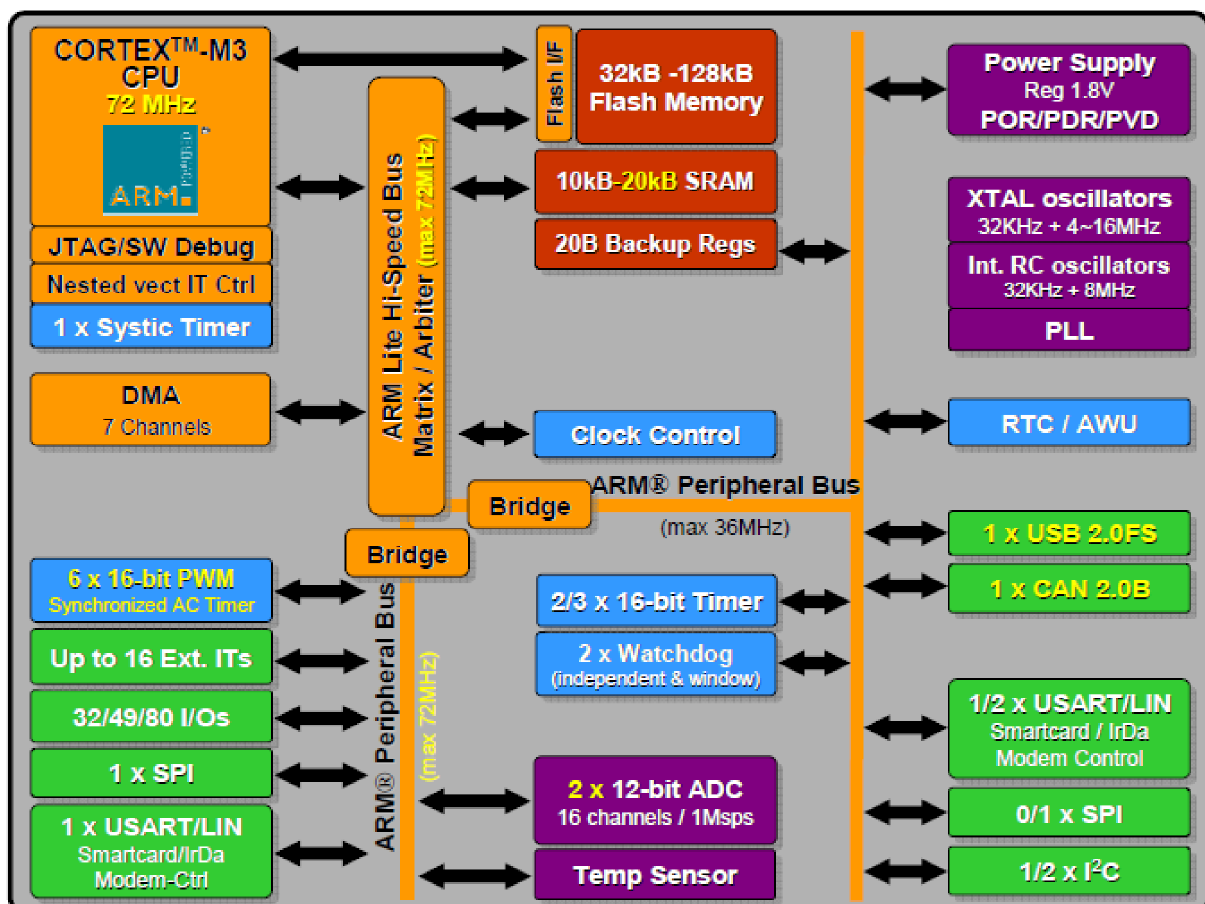

Figure 3: STM32F103 Performance line Block Diagram

2.2.1 System Architecture consists of Buses, and General purpose DMA (Direct Memory Access), Internal SRAM, Internal Flash Memory which some of them consider as masters and others consider as
slaves. Figure 4 is shown bus Architecture for the STM32 family microcontrollers. The access between Core system bus and DMA bus are controlled by BusMatrix.

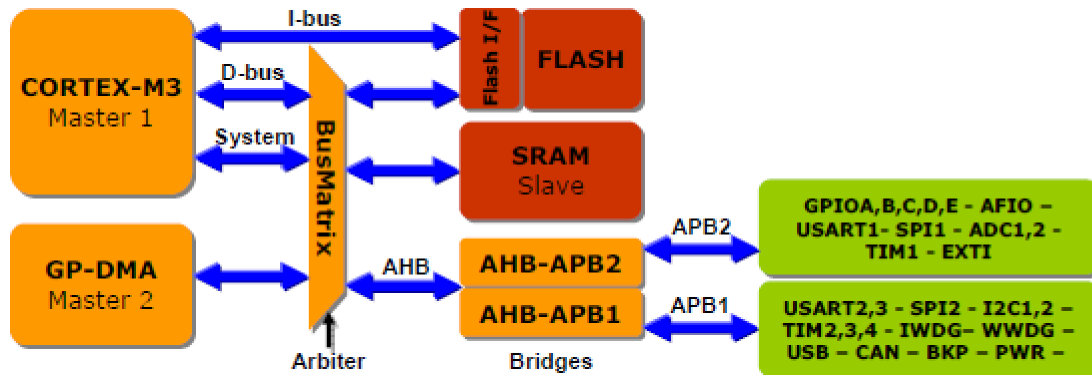

Figure 4: System Architecture

There are four master parts and three slave parts in the architecture which are mentioned below.

Masters:

I-bus (Cortex-M3 ICode bus): It connects the Cortex M3 core to the Flash memory instruction in order to do prefeching.
D-bus (DCode bus): It connects the Cortex-M3 core to the Flash memory Data interface.
S-bus (System bus): It connects the Cortex-M3 core peripheral bus to a BoxMatrix in order to control the arbitration between the DMA and Core.
GP-DMA bus (General Purpose DMA): It connects CPU(Central Processing Unit) and DMA to the Flash memory, SRAM and Peripherals through BoxMatrix in order to make communication between them.

Slaves:

Internal SRAM
Internal Flash Memory
AHB (Advanced High Performance Bus) to APB (Advanced Peripheral Bus) bridge: This bridge divides AHP bus into two buses, APB1 and APB2. APB1 is for peripheral which their frequency is 36 MHz and APB2 is for peripherals which they operate with 72 MHz frequency.

2.2.2 ARM Cortex-M3 core is the microcontroller ***CPU*** and is one of the most significant parts of the microcontroller. This core is the last version of ARM processors which is applied for embedded system. It has well specifications such as 72 MHz maximum frequency, 90DIMPS (Distributed Integrated message Processing System) with 1.25 DIMPS/MHz, performance at zero state memory access, Single-cycle multiplication and hardware division, Nested interrupt controller (maskable interrupt 43 channels, Interrupt processing), Low-power consumption, Low-price, Low-gate count, etc.

2.2.3 Memory system of this microcontroller consists of two parts which they are Flash memory and SRAM (Static Random Access Memory) memory. Flash memory is for storing data and program and its capacity is up to 128 Kbytes. SRAM memory is to read/write at CPU with zero wait state in order to store data for processing by USB and its capacity is up to 20 Kbytes.

2.2.4 Nested Vect It Ctrl (NVIC) stands for Nested Vector Interrupt Controller. It can be used to control up to 43 maskable interrupt channels (maskable interrupt is a special interrupt which could be enabled/disabled or manage by the programmer) and, it has 16 programmable priority levels. It is used to set IRQ (Interrupt Request) channel priorities.

2.2.5 Ext. ITs (EXTI) stands for External Interrupt/Event Controller; it has nineteen edge detector lines in order to create interrupt/event requests. In order to detect external triggers can be used, can be trigged by rising, falling or both trigger and it is also maskable. It is possible to be connected up to eighty GPIOs to sixteen external interrupt lines to detect external triggers (as interrupts).

2.2.6 Clock system (SYSCLK) consists of different clock sources in the microcontroller as well as: HSI (High Speed Internal) oscillator clock, HSE (High Speed External) oscillator clock, PLL (Phase Locked Loop) clock, and LSI RC (Low Speed Internal Resistor and Capacitor oscillator) and LSE (Low Speed External) Oscillator.

HSI oscillator clock is a High Speed Internal clock signal that is an internal (8 MHz) RC oscillator. It is always applied as a clock system for MCU by default.

HSE is a High Speed External clock signal and it can use 4 to 16 MHz external oscillator in order to produce a very accurate clock signal in order to prepare the clock system.

PLL can generate accurate and stable output signal from a fixed low frequency. It multiplies the HSI RC output or HSE crystal output clock frequency to create different frequencies for the microcontroller processor (CPU) and selected peripherals.

The LSI RC clock is a Low Speed Internal clock signal and its clock frequency is about 40 KHz. The LSI RC clock can be used for low power clock source and it is useful in Stop or Standby mode.

The LSE Oscillator is a Low Speed External crystal, its frequency is 32.768 KHz and prepares Real Time clock. These clock sources provide frequency clock for each part of the microcontroller such as CPU, peripherals, etc.

2.2.7 Startup clock is a clock system for the microcontroller at start up (when MCU begins to work). The startup clock is made by HSI RC by default and this clock is 8 MHz because of HSI.

2.2.8 Boot modes defines how to boot the CPU in order to work. There are three boot modes for this microcontroller. Memory map of the microcontroller is shown in figure 5.

It is possible to use different parts of the microcontroller memory in order to boot CPU.

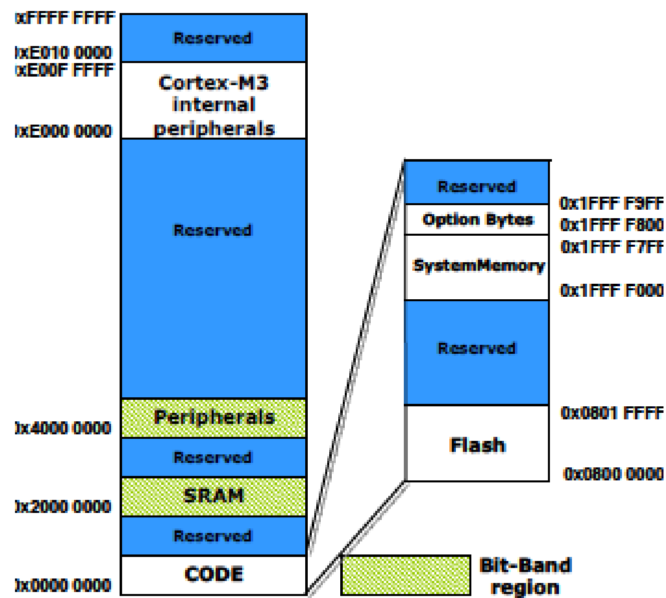

Figure 5: STM32F103 Memory Map

One of these boot modes which it is introduced in the below can be used at Startup in order to boot the CPU,

- User Flash: CPU will boot from User Flash.
- System Memory: CPU will boot from System memory.
- SRAM: CPU will boot from SRAM.

A boot loader is necessary in order to program or reprogram Flash memory by using the USART. This boot loader is placed at the system memory.

2.2.9 Power Supply is to bias the microcontroller to prepare sufficient electrical power for different parts of system. It is described which pins must connect to the power supply in the following.

- V_{DD}: This pin prepares power supply for I/Os and Voltage Regulator (internal). V_{DD} should be connected to 2.0 to 3.6 Volte and this voltage is provided externally via V_{DD} pin on the MCU.

- V_{SSA} , V_{DDA}: These pins prepare external analogue power supply for ADC, Reset blocks, RCs and PLL. These pins (V_{SSA} , V_{DDA}) should be connected to 2.0 to 3.6 Voltage.

- V_{BAT}: This pin prepares power supply for RTC (Real Time Clock), Internal Clock oscillator (32 KHz) and Backup registers.

This microcontroller consists of POR (Power On Reset)/PDR (Power Down Reset), it is an Integrated Circuitry and is always activated in order to fix operation starting from 2.0 Volt. When V_{DD} is less than specific threshold ($V_{POR/PDR}$), the microcontroller can not work and it stays on reset mood. PVD (Programmable Voltage Detector) compares V_{DD} with V_{PVD} threshold. The interrupt service routine informs the microcontroller or put it into safe state, if V_{DD} voltage goes up or down V_{PVD} (threshold voltage).

2.2.10 Direct Memory Access (DMA) transfers data from Memory to Memory, Memory to Peripheral and Peripheral to Memory. DMA consists of seven channels (each channel is used by the hardware which it requests DMA) and it can be applied with main peripherals such as ADC (Analogue to Digital Converter), TIMx (General Purpose and Advanced Control Timers), SPI, USART, I²C, etc.

2.2.11 Real Time Clock (RTC) *and Backup registers* prepares a 32 bits programmable counter for this MCU in order to provide a set of continuously running counters. They are applied for preparing a Clock Calendar function, a periodic interrupt and an alarm interrupt. Power is provided by V_{DD} or V_{BAT} pins for the RTC and Backup registers.

2.2.12General Purpose Timers (TIMx) makes up three standard timers which are able to be synchronized in order to be applied for MCU. Each of these timers consists of 16 bits counter (auto reloadable), Pulse mode output or PWM (Pulse Width Modulation), and they also have a 16 bit prescaler and four independent channels in order to capture input/output or compare.

2.2.13 Advanced Control Timer (TIM1) is as the same as TIMx if it is configured as a standard 16 bits timer, but it is more complete. It can use four independent channels for Input capture, Output compare, PWM generation, etc. It also can work as a three phase PWM multiplexed on six channels.

2.2.14 Inter-Integrated Circuit (I²C) is a bus interface and can perform in two modes which these modes are slave and master. This microcontroller consists of up to two **I²C** bus interfaces.

2.2.15 Universal Synchronous / Asynchronous Receive Transmitter (USART) is a serial port and there are two USARTs are available. One of them can communicate up to 4.5 Mbit/S and another one is up to 2.25 Mbit/S.

2.2.16 Serial Peripheral Interface (SPI) is a serial port and it is able to communicate at speeds up to 18 Mbit/S. There are two SPIs are available which can use DMA controller.

2.2.17Controller Area Network (CAN) is a standard bus which it is used to make communication between microcontrollers and devices without using any host (computer). It can transact with speed of 1 Mbit/S.

2.2.18 Universal Serial Bus (USB) is a serial bus which it is used in order to transact data between MCU and PC (host). There are four kinds of USB which they are grouped according their speeds.

- Low Speed: Its rate is 1.5 Mbit/S
- Full Speed: Its rate is 12 Mbit/S
- Hi Speed: Its rate is 480 Mbit/S
- Super Speed: Its rate is 5 Gbit/S

The USB peripheral for this MCU is Full Speed version (12 Mbit/S).

2.2.19 General Purpose Input / Outputs (GPIO) is an interface for the MCU in order to connect it to external devices which it can use as input to read data or use as output to write data, etc. There are available three GPIOs on this MCU (STM32F103 with 80 Pins). They

should configured by software and they are significant to consider during working with the MCU. All peripherals of the MCU can connect to the outside or other external devices by GPIOs.

2.2.20 Analogue to Digital Convertor (ADC) converts a continuous input analogue signals to digit values in order to use by digital devices such as MCU, PC, etc. This electronic device receives an analogue voltage or current as an input in order to create discrete values at its output. This microcontroller has two ADCs which their resolution is 12 bits and each one has 16 channels.

2.3 Microcontroller Configuration

The microcontroller configuration set the frequency clock for the Microcontroller's CPU and its peripherals. Configuration selects the type of the microcontroller's oscillator and set the exact frequency for each one of the microcontroller component. It also defines the microcontroller's pins for the peripherals as inputs/outputs, operation modes, etc. It just configures the necessary peripherals which they are needed and selected for the project. In this work, CPU, NVIC, GPIO, ADC, TIM1, and USB should be configured by the program in order to run the microcontroller.

In order to configure the microcontroller, it is necessary to define each pin, port or peripheral which it is plan to apply during the project. For example, if it is decided to use the USB peripheral at project, it must be considered all the USB definition. It should be define which pins of the microcontroller belongs to the USB, the USB clock must be defined and also the USB peripheral must be activated to apply for the project. All these points must be mentioned at configuration part of the microcontroller.

Configuration part of the microcontroller program is always executed by the program in advanced before any thing. Microcontroller configuration is operated after each microcontroller reset; it means configuration of the microcontroller is rebooted when ever the microcontroller is reset. In the following part, there is more explanation for the MCU configuration.

RCC Configuration

The first step in the configuration prepares the system clock for the microcontroller and it is significant to be first because the system clock turns on the microcontroller processor and also it is necessary to active the peripherals (each peripherals need clock to work). By configuring the RCC, it is possible to run the system clock. In this part is decided which oscillator could be applied for the microcontroller.

The source clock should be defined for the microcontroller (CPU and all peripherals) in the RCC configuration because this microcontroller consists of different kind of source clocks (it is described in part ***2.2.6Clock system (SYSCLK)***). It is important to pay attention to the microcontroller clock structure before setting clock sources for the microcontroller components. The microcontroller clock structure is named Clock Tree in the microcontroller STM32 documents. Figure 6 introduces the clock tree of this microcontroller. It should be mentioned that any clock for any peripherals must be set according of this map.

The High Speed External oscillator is defined as the source clock for the microcontroller (In this work), because it is possible to achieve 72 MHz (the maximum system clock frequency) if HSE is used as a PLL clock input. Another reason is that it can generate very accurate clock frequency for MCU. So, the HSE is preferred to generate as source clock frequency for the CPU and each peripheral in the microcontroller.

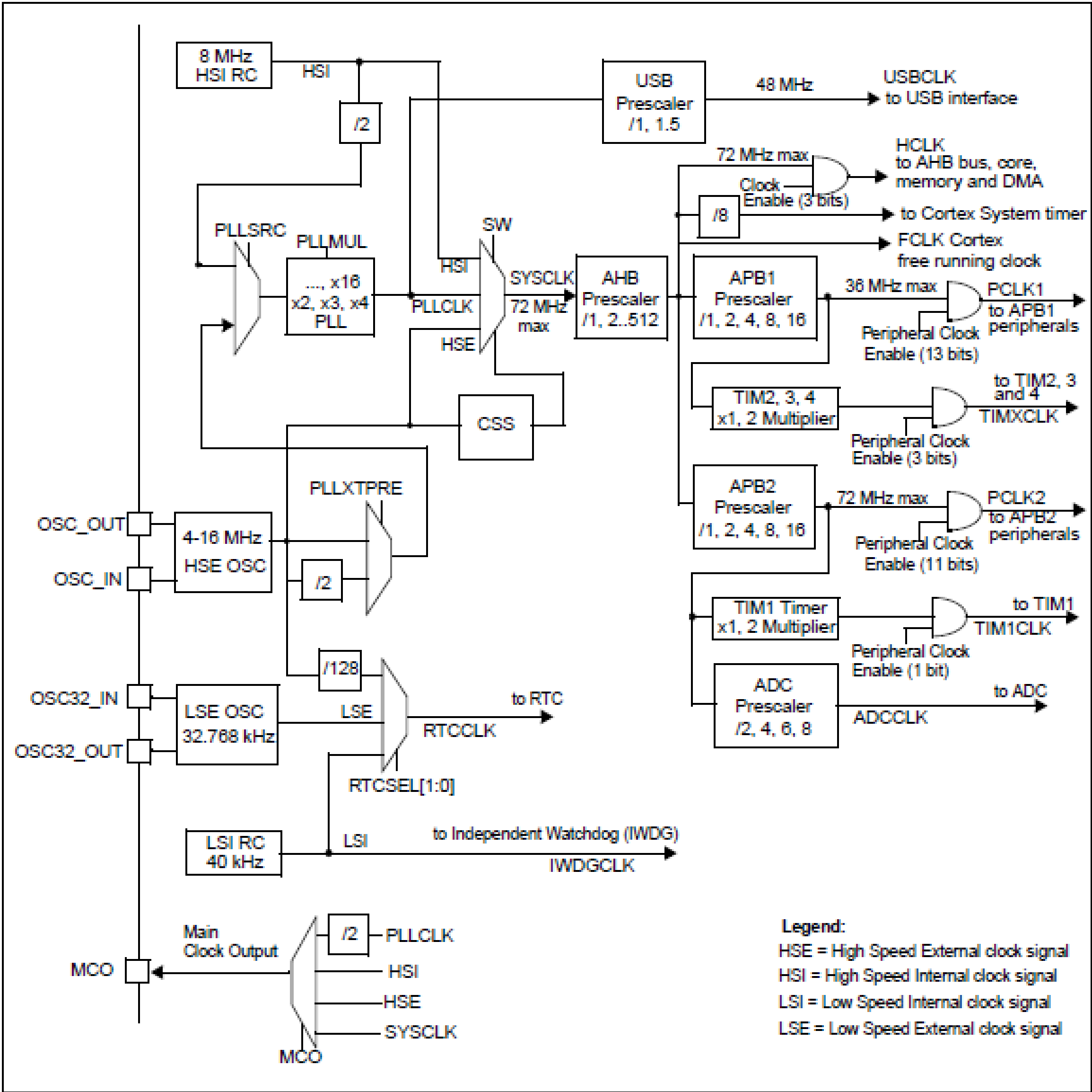

Figure6: Clock Tree for the microcontroller STM32F103xx

After this configuration, the proper clock frequency for the CPU and peripherals can be set. Then the CPU can start to work and the peripherals are ready to use.

Figure 6 shows that the HSE oscillator is used as a PLL clock input and the output of the PLL is SYSCLK (System Clock Source). This SYSCLK frequency should be adjust for different parts of the microcontroller, because each peripheral works with especial frequency clock.

In this configuration, frequency clock for all these parts PLLCLK (PLL Clock), SYSCLK, PCLK2 (Internal APB2 clock), PCLK1 (Internal APB1 clock), and ADCCLK (ADC Clock) should be set, because they prepare clock frequency for all the peripherals. Please remember that RCC Configuration must be consider as the most important microcontroller configurations (it has the first priority).

GPIO Configuration

The GPIO configuration selects and enables the microcontroller ports or any pins of the microcontroller in order to apply for the microcontroller during the task. This configuration defines inputs/outputs and the status of each port or each pin.

Figure 7 shows the Pins Configuration of the microcontroller STM32F103. This pins configuration is important in order to configure the microcontroller ports or pins for programming the microcontroller and it should be considered. The number of pins and their specification is mentioned at the figure 7 and it is so useful for configuring the microcontroller pins.

Figure 7: Pins Configuration of the STM32F103

The microcontroller peripherals are available from pins of Port A, Port B and Port C. So, it is necessary to active these ports by the program according to the work request and it are

obtained at GPIO Configuration. In this configuration, GPIO clock is enabled for all the microcontroller ports in order to enable its ports. It is also defined the ADC pin that this pin is used as an input for the ADC and a pin for the USB circuit (the USB hardware is connected to the microcontroller and it is enabled by using this pin).

PA1 (Port A, pin 1) is set as an input of the ADC for this work and the mode of this pin should be set as Input, but the ADC has not been enabled yet. It is just considered as an input pin for the ADC peripheral.

PB1 is used to enable the USB device and mode of this pin should be set as an Output. Program set or reset (1 or 0 voltage) PB1 in order to active the USB device.

PC6 and PC7 are enabled as an output and their mode is set as Output. They are used to turn on two LEDs.

ADC Configuration

The ADC configuration actives Analogue to Digital Convertor1 (ADC1) for converting analogue input to digital values. This configuration defines the mode of ADC, ADC Data Align Right and also enables and calibrates the ADC in order to work properly. This configuration also mentions that the ADC is trigger by the program (not externally).

The mode of the ADC is Independent mode in this work (in this mode each ADC interfaces operates independently for dual ADC). ADC Data Align Right sets the ADC data alignment from left to right. It means the most valuable bit is written in left.

Timer Configuration

The Timer configuration set and enables the timer1 (TIM1) to operate as a counter for the microcontroller. This configuration enables the TIM1, prepare timer period in order to generate interrupt at the special time which it is set by the programmer and also enable timer interrupt to operate.

NVIC Configuration

At the NVIC configuration, the interrupt priorities of peripherals are set by the program. This configuration gives the first interrupt priority to the USB peripheral, the second interrupt priority to the TIM1 (timer 1) and the last interrupt priority to the ADC1.

USB Configuration

The USB configuration sets and enables the USB clock and all USB status. This configuration defines the USB clock frequency and enables it in order to be used at the project.

The code for the microcontroller and peripherals configuration is brought in appendix A.1.

2.4 STM32USB Preparation

In order to prepare USB communication, the STM32F10xxx USB development kit is applied for this thesis work. It is a USB firmware library and software package (it is used to control the STM32F10xxx USB 2.0 full speed device peripheral). It consists of very useful examples and demos for all USB transfer types. The aim of this firmware library is to provide resources for developing USB application easier and faster in the STM32F10xxx microcontroller family. For this project, the Virtual COM port demo is selected to create the

program for the STM32F10C microcontroller because this example is so close to the USB communication for this thesis.

It must be mentioned that it is also possible to create a new project file for programming the microcontroller by the IAR Embedded Workbench IDE, but the programmer must add each USB files to this new created project file, because it is an empty project file.

This Virtual COM port demo eases the USB programming, because all necessary files regarding to the USB were attached to this Virtual COM port. The USB library is existed in the Virtual COM port demo. This USB library consists of all necessary files for USB communication. These files are written by C language and they prepare all codes for the USB communication. The Virtual COM port demo program can be loaded to the STM32F10C microcontroller and this program (the Virtual COM port demo) can make communication between two computers via USB port and USART (COM Port).

The STM32F10C microcontroller uses as a USB to USART bridge in order to make communication between a USB port of one computer to a COM port (RS-232) of another computer. Figure 8 mentions that microcontroller board that works as a bridge in order to transfer data between desktop and laptop. The Virtual COM port program manages the COM port and USB port to connect the USB port of the laptop and the COM port of the PC in order to communicate directly.

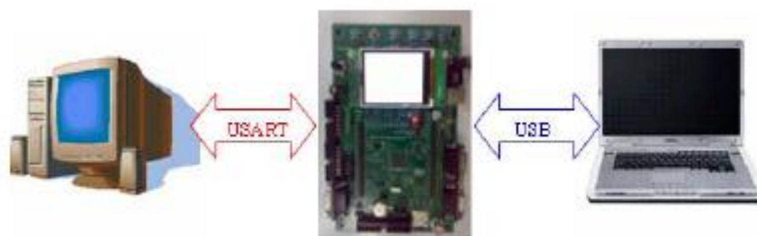

Figure 8: USB to USART communication by the STM32 Microcontroller

So it is possible to use of this program and idea to make communication between the microcontroller and computer via USB port. It just needs to remove the COM port commands or program and set the USB part of the program for this project.

At first, all the program which relates to the USART (COM port) and any program for other peripheral which it does not relate to the thesis goal should be deleted from the Virtual COM

port program, because the USB program is just needed. Then the name of the Virtual COM port demo program is changed to the USB_LabView (any name can be selected). It must be modified many of the USB files and remove unused parts of the program in order to create new program regarding the project desire.

The usb_desc.c, usb_desc.h, usb_endp.c and usb_prop.c should be modified for the USB program according to the project. ***All C code for these files is available in Appendix A.2.*** It is suggested to take a look at them; it may be helpful for you to get the description better.

The usb_desc.c file consists of the USB descriptors for the USB device and it describes the USB device information to the host. This information defines name of the device, version of the USB (USB 2.0), the USB device manufacture, the number of endpoints and endpoint types, etc.

The host system can recognize and learn the USB device (the microcontroller) by USB descriptors. The USB descriptors are the data structures or formatted block of information for the USB device. The device-specific information in device descriptor is used by the host system and it uses them to associate a device to a driver. This source file (usb_desc.c) defines USB Standard Device Descriptors, Configuration Descriptors, Interface Descriptors, Endpoint Descriptors and USB String Descriptors.

The device descriptor represents the device completely and there is only one device descriptor for the USB device and consists of the bLength, bDescriptorType, bcdUSB, bDeviceClass, bDeviceSubclass, bDeviceProtocol, bMaxPacketSize0, idVendor, idProduct, bcdDevice, iManufacturer, iProduct and bNumConfigurations.

bLength: It is the length of the descriptor in byte and is 18 (12 hex) bytes. It means 18 bytes are required to write the USB device descriptor.

bDescriptorType: it is DEVICE descriptor type and is to describe the type of each part in the USB descriptor (its value is1h).

bcdUSB: The binary coded decimal USB states the version of the USB specification. 0100h defines version 1.00; 0110h is for version 1.1 and version 2.00 is specified by 0200h in the program (0200h defines that the USB 2.00 is the USB version for the device). It also mentions that the USB format code is in binary.

bDeviceClass, bDeviceSubclass and bDeviceProtocol: The standard device and interface descriptor consists of field which they are related to the classification: class, subclass and protocol.

They applied for the operation systems in order to recognize a class driver for the USB device. The bDeviceClass can be only set at the device level and most class specifications identify their selves at the interface level. The host can use these fields to associate a device or interface to a driver. There is no class defined in the USB in this project, so there are also no subclass and protocol. The valid value for all should be 0x00. The bDeviceClass set 0x00 (Unspecified) and it allows that one device to support multiple classes. Then the bDeviceSubclass and bDeviceProtocol are set 0x00 because of the bDeviceClass.

bMaxPacketSize0: It set the maximum packet size for endpoint0 in the USB device. For high speed devices should be used 0x40 or 64 bytes (Maximum bytes can be send or receive by the USB buffer is 64 bytes). The host can read this maximum package size for each endpoint from device descriptors.

idVendor, idProduct and bcdDevice: These numbers are defined in device descriptor that they uniquely identify a device to a host. The host can use them to decide what driver should be loaded for the device. For this project, the Vendor ID is 0xffff, Product ID is 0x2008 and bcdDevice is 0x0000, because it is not used.

bNumConfigurations: It is to define the number of possible configurations which is supported by the device. The number of configuration is one in this project (0x01), it means device only support one configuration. The Configuration Descriptor is the second part of the USB descriptors (usb_desc.c). It describes the number of interfaces, the maximum power and how the device is powered. It is possible for the device to have several different configuration descriptors, but most of the devices only have one.

The Configuration Descriptor consists of the bLength, bDescriptorType, wTotalLength, bNumInterfaces, bConfigurationValue, iConfiguration, bmAttributes and MaxPower.

The bLength set the size of the configuration descriptor and it is 0x09 bytes (9 bytes). It means it is necessary nine bytes to be considered to describe the configuration descriptor for the device.
The bDescriptorType defines the type of the descriptor and its type is configuration descriptor.

The wTotalLength is the total length in bytes of data returned. After reading the configuration descriptor by the host, it returns the configuration hierarchy that makes of all related interface and endpoint descriptors. The wTotalLength field shows the number of bytes in the hierarchy. Figure 9 helps to get more information about the wTotalLength. There are no of returned bytes in this project, so the wTotalLength value is 0 (0x00).

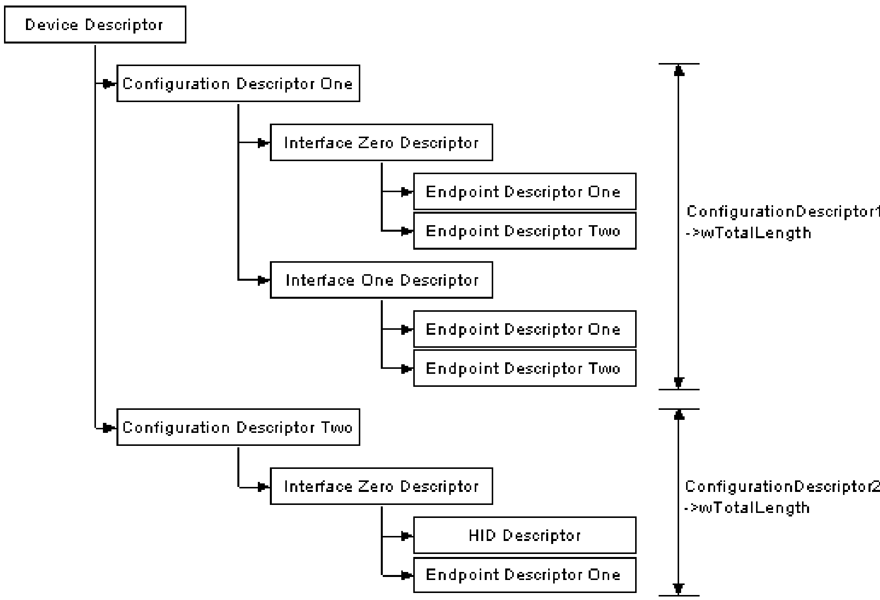

Figure 9: Total Length between configurations Descriptor

The bNumInterfaces stats the number of available interfaces for the configuration. There is only one interface available and the bNumInterfaces value is 1 (0x01) at this project.

The bConfigurationValue is applied by the Set_Configuration request in order to select this configuration. It should be 1 or higher and it is set for this device one (0x01).

The iConfiguration is an index to a string descriptor which describes the configuration in human readable form. The iConfiguration value is zero if the device does not have string descriptor. It is zero (0x00) for the device.

The bmAttributes describes power parameters for the configuration which is a bus powered or a self powered device. It is defined self powered for this device and its value should be 192 (0xC0).

The MaxPower defines how much the maximum power will be drained by the device from the bus. The device is considered self powered, so it does not drain from bus and its current is 0 mA. Then the MaxPower value is zero (0x00).

The interface Descriptors can be considered as a groping of endpoints into functional group performing a single feature of the device. It is to states zero or more endpoints for the endpoits descriptors. It contains the information regarding endpoint transfers data type. The interface Descriptors describes the bLength, bDescriptorType, bInterfaceNumber, bAlternateSetting, bNumEndpoints, bInterfaceClass, bInterfaceSubClass, bInterfaceProtocol and iInterface.

The bLength sets the size of the interface Descriptors for the device and its length is defined nine (0x09 bytes).

bDescriptorType defines the type of the descriptor and it is interface descript descriptor for this part.

The bInterfaceNumber describes the number of interfaces. Its based value is zero in order to introduce at least one interface descriptor (zero only presents one interface device). For more interface descriptor the bInterfaceNumber should increase according to the number of interface devices. This is zero for this device because there is only one USB device to use USB connection.

The bAlternateSetting set the alternative interfaces. It is to use when there are more than one interface to communicate via USB. Its value is set zero because there is only one interface needed for this thesis. If we have two interfaces in this project, then the bInterfaceNumber's value should be set one and the bAlternateSetting also should be set one.

The bNumEndpoints defines the number of endpoints which is used for the interface. Its value set two for this project, because two endpoints are considered for this device in this thesis. The endpoints 3 (OUT3) is used to receive data from PC and the endpoints 1 (IN1) is use to send data to PC.

The bInterfaceClass, bInterfaceSubClass and bInterfaceProtocol specify supported classes. Classes are such as, HID (The Human Interface Device), Communication, Mass Storage, etc. They allow many devices use the same drivers and it is not needed to write specific driver for the device. Their values are set zero because they are not used in this project.

The iInterface is an index of String Descriptor to describe this interface in order to allow having a string descriptor for the interface. This definition does not use in this project, so its value is zero.

The Endpoint Descriptors describes the endpoints and each endpoint must be defined instead of endpoints zero (endpoint0). Endpoint0 is always described by default and it is supported by every devices. There for it never has a descriptor. The host system learns about endpoints by studying the configuration descriptor which each endpoint belongs to it. At this program, two endpoints are defined, so both of them should be described at the endpoint descriptors file.

The Endpoint Descriptors 3 and 1 are described both and they include the bLength, bDescriptorType, bEndpointAddress, bmAttributes, wMaxPacketSize and bInterval.

The bLength sets the size of the Endpoint Descriptors in byte and its size is seven (0x07). This size is equal for any endpoint descriptors.

The bDescriptorType defines the type of the descriptor and the type of descriptor is Endpoint Descriptors.

The bEndpointAddress defines the endpoints number which it is used. The Endpoint 3 is defined by 0x03 and it means endpoint three is set and is 0x81 for the Endpoint 1 and it mentions endpoint 1 is set. (Bits 0 to 3 define endpoint number and bits 4 to 6 are zero as reserve and bit 7 should be zero or one. If it is one, it means endpoints is for IN and if it is zero, it means endpoint is for OUT).

The bmAttributes is to define the type of the transfer type. Four types of data transfer are available which they are selectable. They are Control (00b), Isochronous (01b), Bulk (10b) and Interrupt (11b). For this thesis, transfer type is set Bulk and it should be defined by 10 binary or 0x02 hex. (A Bulk transfer is used to send and receive for large busty data. High speed and full speed devices can use the Bulk transfer).

The wMaxPacketSize defines the Maximum Packet size for the endpoints which it is possible to transfer in a transaction. It is ignored for endpoint 3 and 1, because transfer type is selected bulk for both of them and it automatically should be used with the maximum number of data bytes.

The bInterval is set to specify the maximum polling interval of certain transfer. The control and bulk transfers do not need to have any value and they are always zero. The Isochronous transfer equals one.

The USB String Descriptors includes descriptive text and it is to define the USB descriptors. It prepares human readable information and it is optional. It explains the language, manufacturer, product and serial number for the device.
The language consists of the bLength, bDescriptorType and LangID.

The bLength defines the size of the descriptor in bytes. The bDescriptorType describes the type of the descriptor.

The LangID sets the language of the descriptor. It is set English and its value is 0x0409 hex for the English United State.

The manufacturer, product and serial number contain the bLength, bDescriptorType and bstring.

The bLength defines the size of the manufacturer, product and serial number and bDescriptorType defines the type of them. The bStrigs sets a Unicode Encoded String for each of them. The manufacturer code is considered BALTIC Engineering, the product is set USB to LabView If BE 2008 the serial number is set Ver 1.000 in this thesis.

After above setting it is possible to say that all setting for the usb_desc.c file is completed and it is time to set the usb_desc.h file.

The usb_desc.h file is a header file or include file for the usb_desc.c file. It is needed to define setting for the constants which they are used in the usb_desc.c. This file should be prepared according to the setting in the usb_desc.c.

There is an explanation for the constant definition in this usb_desc.h file.

USB_LABVIEW_PORT_DATA_SIZE is set to 64 decimal, it means the maximum byte in transaction is 64 and it is possible only send or receive 64 bytes data via USB port each time. USB_LABVIEW_PORT_INT_SIZE is set to 8, it means that buffer size for the USB is only 8 bits. USB_LABVIEW_PORT_SIZ_DEVICE_DESC is set to 18. It mentions the device descriptor has 18 descriptor parts. USB_LABVIEW_PORT_SIZ_CONFIG_DESC is set to 32. it defines that the configuration descriptor has 32 descriptor parts.

USB_LABVIEW_PORT_SIZ_STRING_LANGID is set 4 and consists of 4 descriptor parts. USB_LABVIEW_PORT_SIZ_STRING_VENDOR is set to 38, because it consists of 38 parts. USB_LABVIEW_PORT_SIZ_STRING_PRODUCT is set to 52 and it says that it made of 52 parts. USB_LABVIEW_PORT_SIZ_STRING_SERIAL is set to 22 in order to mention that it has 22 spaces for its descriptors.

It is also necessary to modify the usb_endp.c, usb_prop.c and usb_prop.h in order to prepare the USB communication. The usb_endp.c defines the endpoint routine for the USB peripheral. This file does not need too much modification; it is just to change the buffer variable name to u8 buffer_out[USB_LABVIEW_PORT_DATA_SIZE].

The usb_prop.c and usb_prop.h define the USB application related to properties. In order to modify them, it should recharge the each definition name from Virtual_Com_Port to USB_LabView (this is the name of the project file for this project).
All code for these files is available in the ***Appendix A.2*** for more information and using as the pattern for the similar project.

The usb_desc.h file is a header file for the usb_desc.c and this file also should be set according to changes in the usb_desc.c file. The usb_desc.c codes are placed at ***Appendix A.2.***

The USB specification version (bcdUSB) is set 2 and it means USB 2.0 is the version, idvendor number is set 0x0ffff

Now the microcontroller STM32F10 can use the USB peripheral to plug in to the host and the host system can recognize the USB in order to communicate with the microcontroller.
This description regarding the USB communication is general and it does not matter which kind of microcontroller is chosen. It is also possible to set the other microcontrollers which the USB peripheral is available for them. It is possible to apply above explanation to prepare USB communication for other version of microcontrollers.

3 LabVIEW

Laboratory Virtual Instrument Engineering Workbench (LabVIEW) is created by National Instrument Company. LabVIEW is a Graphical program language, does not need any text to create the program and it does not look like the text programming languages such as C/C++, Java, Basic, Fortarn, etc.

There for, LabVIEW is called G (Graphical programming language). Because, LabVIEW program would be created by using a graphical notation which is made by connecting functional nodes and wiring them together in order to flow data. It also has continuous Auto compiling, i.e. when ever an error happens, it immediately informs the user.

In order to use LabVIEW development environment, it should be installed on computer running system operation Windows, Linux, etc. Created LabVIEW program can run on Microsoft Windows, Microsoft Pocket PC, etc. LabVIEW also can work with embedded workbench as well as FPGAs (Field Programmable Gate Arrays), DSPs (Digital Signal Processors), and MCU (Microcontrollers).

LabVIEW is one of the most complete programming languages and is so easy to work in order to write complicated programs. It can be used to take measurements, analyze data, present results, etc. It also has graphical interface usage and can be applied for simulations, presentation of ideas or projects, etc.

LabVIEW consists of many specific library functions that they can be applied for Data Acquisition (DAQ), General Purpose Interface Bus (GPIB), Serial Interface, Communication over the Internet, etc. The analyze laboratory of LabVIEW includes of many functions such as signal generation, signal processing, filters, statistics, linear algebra, array arithmetic, and windows.

So, LabVIEW prepares a compete libraries of functions and subordinates to hint engineers and scientists. It is so useful and significant program for engineering applications, especially in electrical fields.

As it was discussed, LabVIEW is a high level and extended program and it is not possible to mention it by detail in this report. There is just a brief description regarding some parts of LabVIEW which it is applied at this project in the following. This chapter is just an introduction for LabVIEW.

3.1 Virtual Instrument

Virtual Instrument (VI) is a LabVIEW program which LabVIEW operates on data flow model in order to flow data sources to data sinks. VI is divided into two groups that are internal VIs and user created VIs. Internal VIs execute normal functions such as multiplication two number or opening/closing files and are packed or prepared functions by LabVIEW as available function boxes. User created VIs made of Front Panel (Graphical User Interface) and Block diagram (Code Pipeline) windows which they are applied by the programmer to create his/her program.

3.1.2 Front Panel

The *front panel* is a window and each VI has a front panel window. When ever user opens a VI or wants to create a new VI, the front panel window is appeared on the screen. See figure 10, it is to show a front panel with the Control palette.

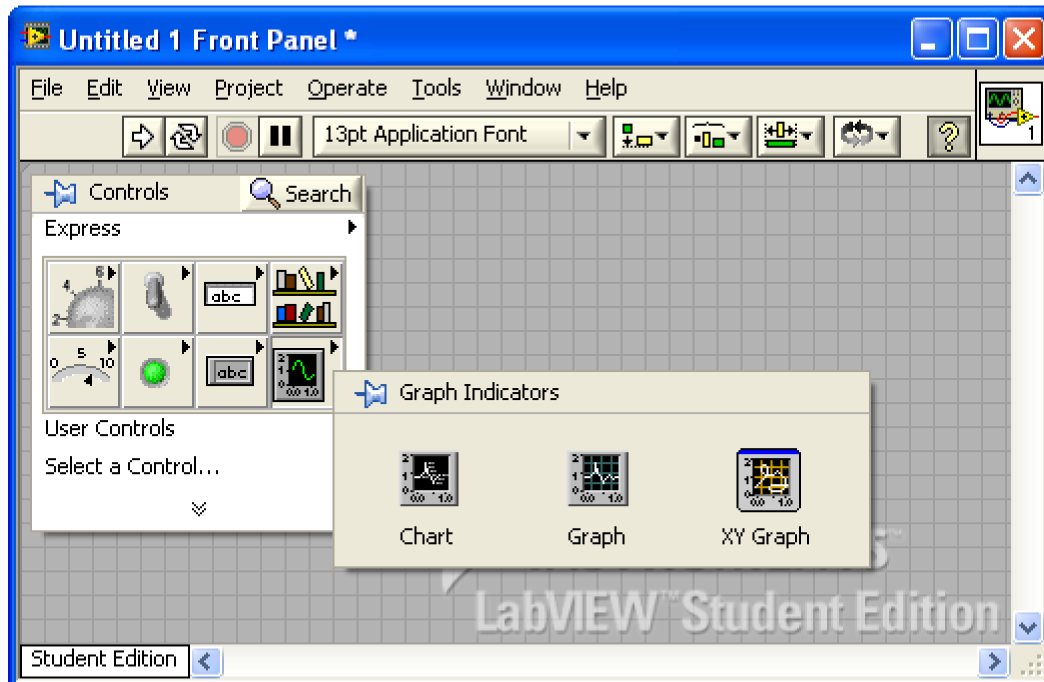

Figure 10: Front panel window

Figure 10 presents a front panel window that would be used to simulate a physical instrument. A physical instrument could be modelled by user and can be made by using knobs, push buttons, graphs, and many other controls (which they are considered as user inputs) and many indicators (which they are mentioned as program outputs). The control palette (that is in front panel) consists of knobs, push buttons, graphs, charts, switches, slides, gauges, LEDs (Light-Emitting Diodes), etc. Control palette is used to compose physical instrument at the front panel. In order to input data or set data as input, it is possible to use mouse, keyboard. It is also possible to view the result in the front panel window.

There are buttons in the front panel that four of them are more useful in order to run the program or stop it. These buttons are Run, Run Continuously, Abort Execution, and Pause. They are on the top of the front pane, respectively from left to right. Run button just executes program for one time and run continuously does it for ever. Aboard execution stop working the program and if user wants to run the program again, the run buttons should be clicked and the program would be run from the beginning. The pause button stop working program and it is possible to press it again to let program works, but not from the beginning. The program would be executed from when ever it was paused.

3.1.3 Block Diagram

The *Block diagram* is to present a pictorial (Graphical) explanation of LabVIEW program or algorithm. In order to create this pictorial description of program, nodes and wires are

needed. These nodes and wires are called executable icons. For creating program, nodes should connect to each other by wires in order to flow data between them. All nodes and required functions are in Functions palette. Figure 11 presents a Block diagram window with its functions palette.

The block diagram consists of a kind of program logic which data flow from sources to sinks and program control and modify the data flow. The block diagram belongs to a VI. The pipeline structures of sources, sinks, VIs are used to make the program logic.

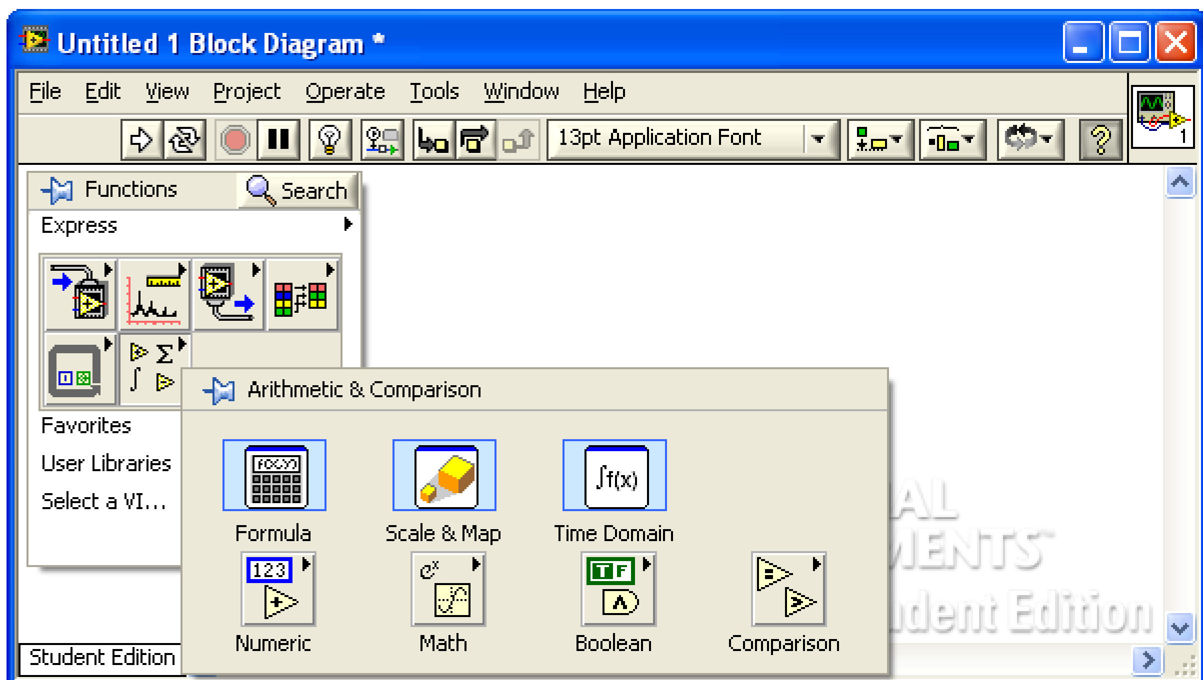

Figure 11: Block diagram window

For each source and sink in the front panel, there is a sources and sink in the block diagram and it is possible to execute program from each one of these windows.

The block diagram has buttons which some them more significant to drive the program. These buttons are Run, Run Continuously, Abort Execution, Pause, Highlight Execution, Retain Wire Value, Start Single Stepping, Start Single Stepping, and Step Out and are place in top of the window, from left to right respectively.

The **Highlight** buttons highlight the pipeline or highlight each part of the program during executing. The **Retain wire** value button presents the value of each pipeline in the program. The **Start Single Stepping** and **Start Single Stepping** buttons execute the program step by step and jump out from one step to another one. **Run**, **Run Continuously**, **Abort Execution**, and **Pause buttons** operate as the same as front panel.

3.1.4 Controls, Indicators and Palettes

A Control is a data source in LabVIEW which it presents a graphical element on the VI front panel. This element can receive its data input from a user or another VI. The value of a control can be read by using retain wire value button on the block diagram. It should be

mentioned that each controls has its own data type and it should be considered in order to connect them together for making a VI. Unless the program does not execute and error would be appeared.

An Indicator is a data sink in LabVIEW and also appears as pictorial elements on the VI front panel window. The output of a VI or a user can be display by using the indicators. The data flow of an indicator can be presented on the block diagram by using retain wire value button.

All controls and indicators on the front panel and also functions on the block diagram are accessible by the Palettes on the VIs the necessary indicators, controls and functions can be taken from palettes in order to create program.

3.1.5 LabVIEW Application

LabVIEW Application is a graphical program on VIs which is created by the controls, indicators and functions. Figure 12 is shown a very simple LabVIEW program, it is clear that is a pictorial program.

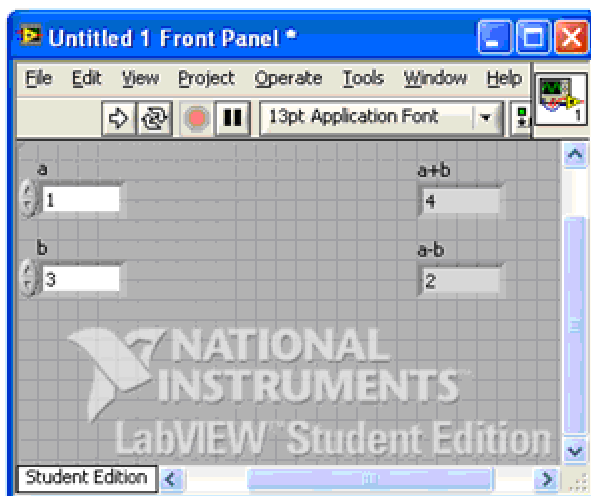

Figure12_A: Front panel of program

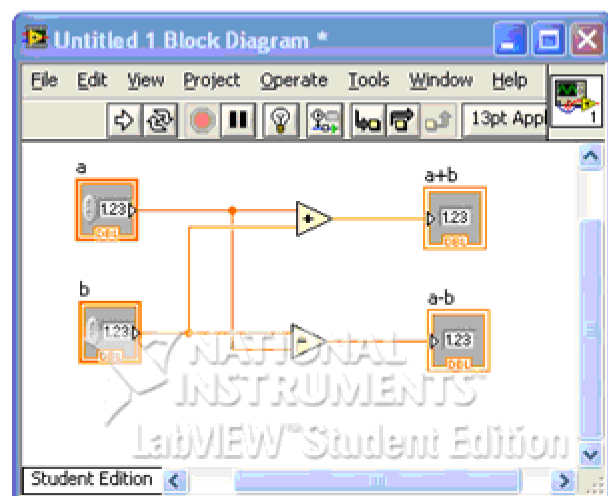

Figure12_B: Block diagram of program

Figure 12_A shows the controls and indicators on the front panel, a and b are inputs (sources) and they are called controls. a+b and a-b are outputs (sinks) and they are named indicators for the program.

Figure 12_B presents the controls and indicators graphically. Figure 12_B shows the functions that are applied on the block diagram. There are Add and Subtract functions on the block diagram which their name describe their operations.

This program just adds and subtracts two inputs and presents the result in the output. Input a is equal 1 and b is equal 3. The result for addition is 4 and for subtraction is 2. So, a graphical program was created by VIs element.

3.2 LabVIEW Requirements for Thesis

This part introduces and explains controls, indicators and functions which are used at this project. It is just a brief explanation regarding their performance in LabVIEW. It is also describe the USB driver in order to connect the USB device with VI.

3.2.1 The USB Driver

The last step for completing the communication is to prepare a driver for the USB device. A device driver is a computer program which allows a higher level computer program or operating system interacts with the device or hardware. Preparing an appropriate driver for the device is not easy and normally it needs to write a program as a driver for the device according to its futures.

It is a driver software structure and is to unify instrumentation software standard for each equipment which it uses GPIO, Serial port (RS-232/422/485), USB port, etc. By using VISA LabVIEW program, it is possible to solve this problem. VISA stands for Virtual Instrument Software Architecture and is an API (Application Programming Interface) which it is for instruments communication. VISA provides a group of functions in LabVIEW in order to send commands to the devices and reading responses back from them. It is also to use making a proper driver for the USB device to communicate with LabVIEW.

In order to create a driver, VISA is applied to create an INF file and it is a setup information file which is used by the Microsoft Windows to install software and drivers; after creating the INF, it should be copied and installed on the PC. Then it makes the PNF file (Precompiled Setup Information file and it is created by the windows to continent the processing). This PNF file is the driver for the USB device, so the driver is ready and the USB device can be plugged to the PC. After making the driver and plugging the device, the USB device can communicate with the LabVIEW.

In order to prepare driver for the USB device, the Vendor ID, Product ID, Manufacture Name and Model Name are needed to be used in the VISA. By them, it is possible to have a specific driver for own device. These should be defined at the microcontroller program for the USB peripheral and it was described how to build completely. Vendor ID, Product ID, Manufacture Name and Model Name are used to make an address for the device that is called Device Instance Id. It is needed when the I/O VISA functions are applied to write LabVIEW program.

A brief description is written regarding above explanation in Appendix B in order to create the driver for the USB device by using VISA. This description is prepared by using National Instrument website (http://zone.ni.com/devzone/cda/tut/p/id/4478#toc0).

3.2.2 LabVIEW Functions

In order to prepare the LabVIEW part of the project, it is necessary to apply LabVIEW functions. These functions are placed on front panel and block diagram palette which are

known as controls, indicators (on the front panel) and functions (on the block diagram). The necessary functions for this project are mentioned in the following.

3.2.2.1 VISA Functions

VISA is a standard I/O Application Programming Interface and it has a set of useful functions in LabVIEW. VISA functions are important to control serial ports and making communication between LabVIEW and instruments. They are applied to read data from instruments or write (send) commands into them which these functions are visible in the block diagram. These functions are shown in figure 13.

VISA Resource Name, VISA Open, Visa Read, VISA Write, and VISA Close functions are used in order to make communication between LabVIEW application and the USB port (serial ports of the PC, it does not matter the USB or the COM port). These four functions are applied to communicate with the USB device in this project.

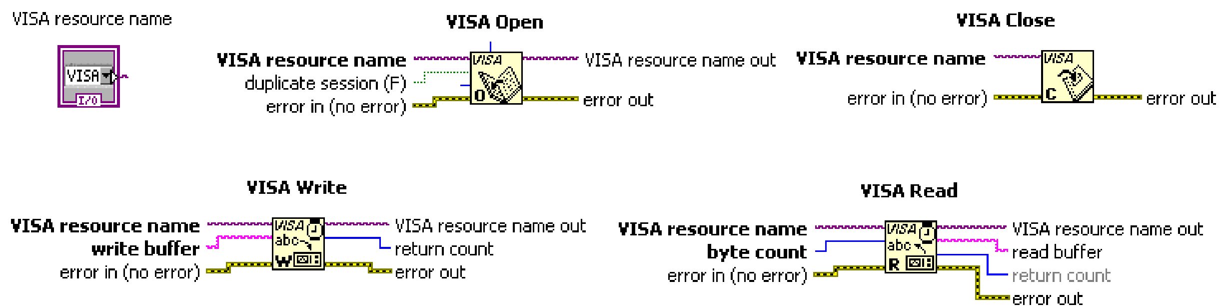

Figure 13: VISA Functions

VISA Resource Name function specifies the USB device to LabVIEW application by the USB RAW device format. It is used to introduce the USB device to LabVIEW application. The USB RAW device format is a descriptor for the device; it was made after creating the driver for the device. This device descriptor should be typed into VISA resource name and after that LabVIEW can recognize specific USB device.

VISA Open function opens a USB device which is specified by VISA Resource Name to LabVIEW. *VISA Read* function reads data from the USB device and it must be placed after VISA Open function in order to read data from USB device. *VISA Write* function writes data from LabVIEW to the USB device by using VISA Open and VISA Resource Name functions. *VISA Close* function closes a device specified by VISA Resource Name.

3.2.2.2 General Functions

In order to complete the LabVIEW application, it is necessary to apply other functions which they may be used for any application. There are just some functions to describe in this part that are applied in this project. It is described all functions which are applied for the thesis in the following.

Figure 14 presents Numeric functions which are available on the block diagram palette. The name of each function describes its operation in LabVIEW application.

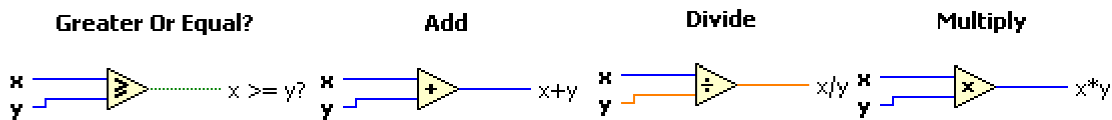

Figure 14: Arithmetic Numeric functions

There are functions that are applied for converting the type of data in order to mach different type of functions or pipeline together. These functions are shown in figure 15.

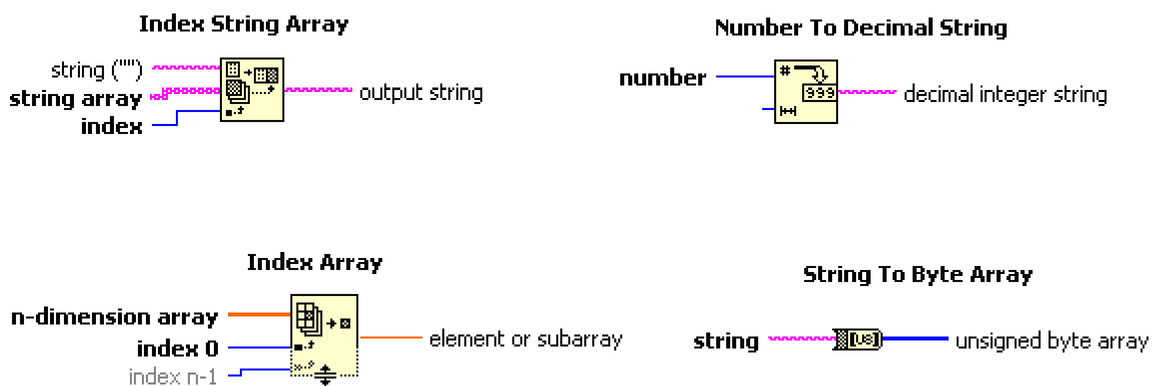

Figure 15: Converting functions

Index String Array function receives a string array as input and converts it to a string. *Number to Decimal String* function converts number to a string. *Index Array* Function returns the element or subarray as output from the n-dimensional array (is input). *String To Byte Array* converts input string to unsigned byte array.

Figure 16 is presents two kind of Loops functions and also a Time Delay functions. *Time Delay* function makes a time delay for the VI, it is manually loaded with a number (time delay) in order to generate delay into calling VI.

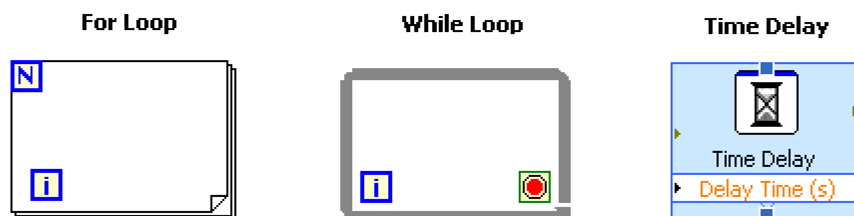

Figure 16: For Loops, While Loops and Time Delay functions

For Loop function makes a For Loop in the program and it executes the program n-times which n is a value defined for (N). n should be wired to the (N). Then Loop repeats the code n-times. (i) terminal counts the iteration and starts counting from 0 to n-1.

While Loop function executes the code and it is possible to set a condition for it. This code operate these command:

Continue if True/ Stop if True or **Continue if Error/ Stop if Error** (i.e. if condition is True continue/stop the operation or if condition is Error continue/stop the operation).

The above functions are available on the block diagram palette. There are Waveform Charts, Bundle, and Waveform Graphs. They are accessible on the front panel palette to plot diagrams on front panel. They are presented by figure 17, they are useful to plot the diagrams.

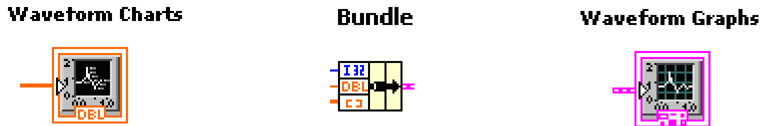

Figure 17: Waveform Charts, Bundle, and Waveform Graph

Waveform Chart and Graph are applied for plotting the output data; it is possible to set the start point of signal by using waveform graph.

Bundle is to prepare cluster from individual numbers.

The necessary LabVIEW parts for this thesis were discussed shortly in this chapter. It was tried to cover each LabVIEW session which is used for the LabVIEW solution.

4 Thesis Solution

This chapter discuses regarding the solution for the thesis and consists of three parts. First part is regarding the microcontroller solution. Second part explains LabVIEW solution. Last step is regarding final solution and presents the project result.

4.1 Microcontroller Solution

It is always supposed, digital samples of analogue signal are ready to apply for digital signal processing and is just important how to use these prepared digital samples in different fields of signal processing (for example, speech processing, image processing, sound and vibration, etc.). So there is not too much attention how to prepare the digital samples at schools practically.

The aim of this section is to explain how it is possible to have digital samples by a microcontroller practically. The sampling theory idea is applied to prepare a program for the microcontroller in order to run it for creating digital samples of the analogue signal. According to the signal theory, the input signal should be convolved to sampling frequency in time domain (Spatial domain) or their multiplication in frequency domain to sample the analogue signal. In order to have accurate samples for reconstructing the signal again, it is necessary to apply a proper sampling frequency (Fs) according to the sampling theory rule. The microcontroller must be programmed in the way which it can satisfy a situation similar to the sampling theory (Nyquist-Shannon).

Let explain a little bit more regarding the microcontroller task by detail. The microcontroller receives an analogue signal from signal generator and the input signal to the ADC of MUC should be sampled to the digit values. Then these digit values must be packed as a 32 bytes array in order to send to the PC. The size of this array can be defined by the programmer (optional). It is also desired to change the sampling frequency in order to sample input signal with different range of frequency. The frequency of input signal can change between 100 Hz to 10 KHz.

To achieve this goal, it is needed to apply the microcontroller's ADC and its timer (they are the microcontroller peripherals). The input signal is converted or translated to the digital value by the ADC and timer should be used to take care of the time interval between each produced sample (timer makes sampling frequency). But it is not possible to run the microcontroller for this aim easily, there are problems regarding this job.

One of these problems is that input of the ADC is an alternative signal, so the amplitude of signal is changing continuously. There for, the ADC's output value (digit value or sample) is also changing frequently. These taken samples or digit values must be saved in memory and also it is important that time interval between each taken sample be equal in order to create a pack of samples and reconstructing the signal on the PC.

The last problem is how to send this pack signal via the USB port to the PC, because of the buffer size in the USB peripheral. The result of ADC output (Taken sample) is saved in 12

bites, but the buffer size of the USB is 8 bites. For transferring these samples, it must be used the USB buffer and this problem should be considered in order to send data correctly.

It is discussed how these problems can be solved for this part at the following statements. The figure 18 shows the flow chart for the program. Code for this flow chart is considered at **Appendix A.0**.

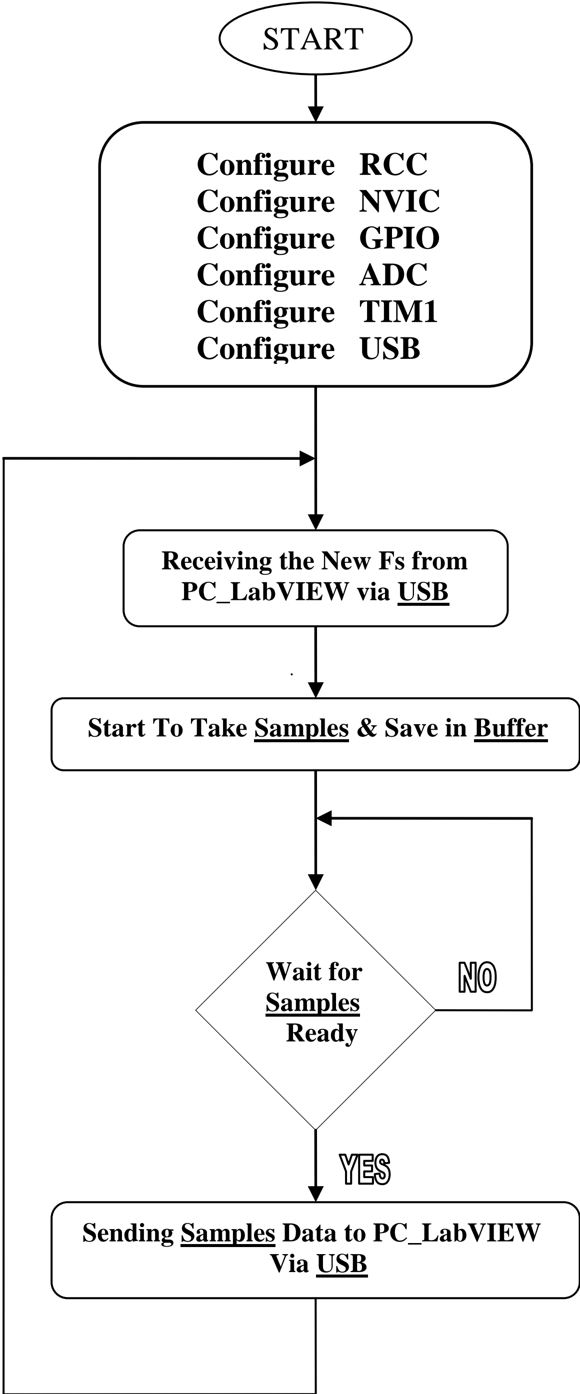

Figure 18: Microcontroller program flow chart

The alternative signal is received by the ADC, it is not a constant signal (dc) and the amplitude of the signal is always changing from zero to the pick voltage. So the sample value in the output of the ADC is always changing according to the time period of the signal.

In order to create a pack of samples, it is necessary to save each sample with an equal time interval. The timer of the microcontroller should be set as the same as required time interval between each digital sample. The timer is activated by the program and it begins to work. After each over flow (when the set value is counted by the timer) the timer generates an interrupt to inform the ADC that one period timing is finished. The program is always waiting for this interrupt and after getting this interrupt the ADC is activated to work. When the conversation is finished, the prepared sample value is saved at the memory. It means, timer always switch on the ADC to take samples and program after taking sample turns off the ADC and save the sample. This operation is always repeating. These procedures will be continued up to taking 32 samples of the input analogue signal and completing the pack. When a pack of the digital samples is ready and it consists of the 32 bytes, then this pack is sent to the PC. Code for the ADC and TIM1 execution is ready in **Appendix A.2**.

It is also possible to check the sample values easily by the MATLAB or Microsoft Excel program which they are taken correctly or not. The sampled values can be read from the program in the microcontroller Embedded Workbench. In order to check the samples value, an analogue signal with 500 Hz frequency and 1 volte amplitude is given to the microcontroller's ADC. As illustrate, one pack of samples for the analogue signal consists of these values; 3227, 3023, 2783, 2508, 2237, 1970, 1740, 1549, 1419, 1354, 1373, 1454, 1605, 1802, 2044, 2306, 2590, 2845, 3082, 3266, 3405, 3474, 3482, 3412, 3266, 3082, 2860, 2534, 2286, 2013, 1779, 1575 and by applying these samples, it is possible to reconstruct the analogue signal again. Figure 14 presents reconstructed signal by these taken samples.

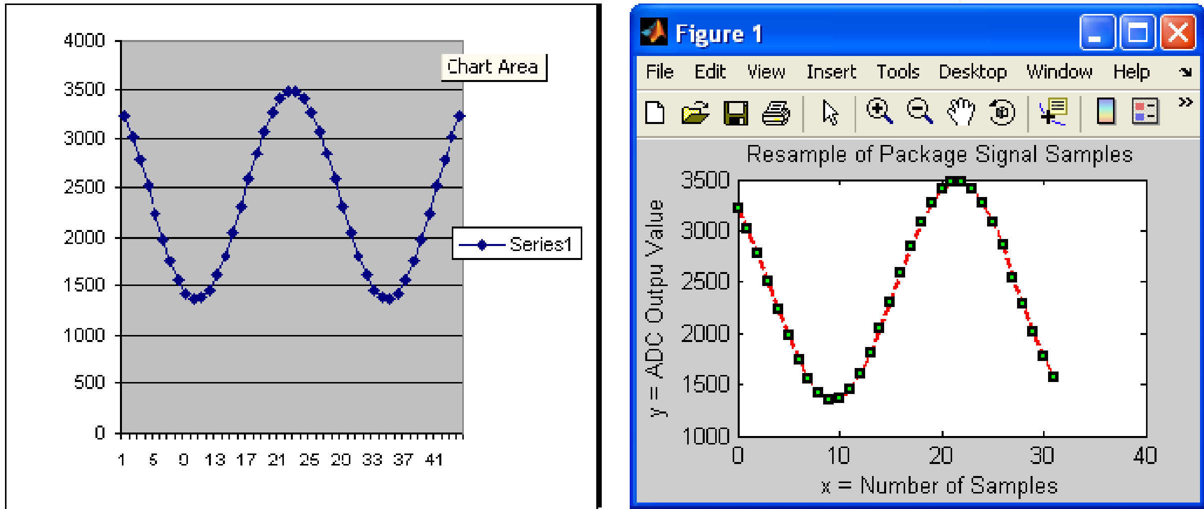

Figure 19: Resample signal

So the figure 19 proves that samples of the pack are correct and it is possible to send them to the PC via USB port.

The size of each sample is 12 bits, because of the resolution of the ADC and the buffer size of the USB is 8 bits. The program must split each sample to two bytes (each sample should be divided to the least and most valuable part) in order to use the USB buffer. There for each sample occupies two buffers of the USB and it can send 32 samples of signal per each transaction, because the USB port can just transact 64 bytes together in the highest speed.

This is so important to pay attention to send the data to the PC via USB and it is also should be mentioned during the reconstructing the signal at the PC. These divided samples should join together In order to reconstruct the signal again at the PC. Code and function for sending samples to PC through USB is available in **Appendix A.3**.

The sampling frequency (or time interval between each digital sample) can be set by using different ways. In this work, it is demanded to set this time interval manually from the PC LabVIEW running program. Ten numbers from zero to 9 can be selected by user in LabVIEW program and this selected number is sent to the microcontroller in order to change sampling frequency. This selected number is detected by the microcontroller program and program will change the timer setting according to this number in order to change the time period.

The microcontroller program can receive a number from 0 to 9 and resets the timer again according of these numbers. Each one of these numbers presents a new value for the timer. The sampling frequencies are 100 Hz, 125 Hz, 200 Hz, 250 Hz, 400 Hz, 1 KHz, 4 KHz, 5 KHz and 10 KHz which they can be made by the timer. For example, when the microcontroller program receives the number 0 from LabVIEW application via USB, the program should set the timer with 100Hz to work (the time interval between each sample is 0.01 Second). By applying this method it is possible to define many different sampling frequencies for the microcontroller to set the timer for making sampling frequency (to make a proper time interval for sampling the analogue signal). It is mentioned the functions and code for receiving data from PC in **Appendix A.3**.

It is discussed in this part how a proper program was prepared for the microcontroller and its peripherals in order to obtain required goals in the microcontroller side.

Now the microcontroller is ready to do its job completely and the last task is to accomplish PC side in order to cooperate with microcontroller.

4.2 PC LabVIEW Solution

This section explains how the LabVIEW program is applied to create the LabVIEW application (PC running LabVIEW) in order to reconstruct the signal form the received samples pack. After preparing the microcontroller programming, it is time to create a LabVIEW application by LabVIEW program in order to cooperate with the microcontroller via the USB. During this cooperation the LabVIEW application has two significant responsibilities. The first one is read and write data by using USB, and second is to reconstruct the original signal again to present it on the PC screen. In order to achieve the goals, there are some points which must be considered to write the program for creating an appropriate LabVIEW application. This program is introduced in figure 20.

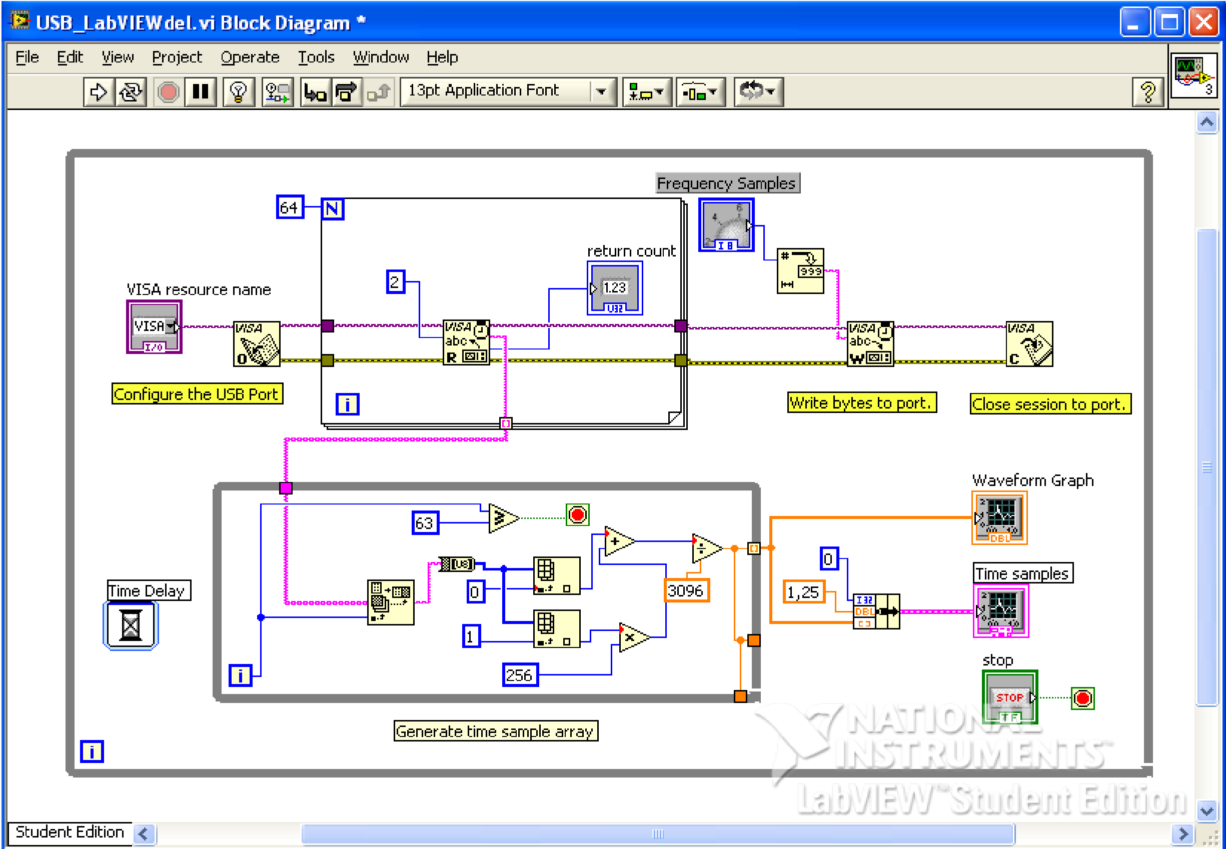

Figure 20: Final LabVIEW program for the project

According to the task, at first the LabVIEW application must be able to prepare a well environment for the user to receive the data and send it easily to the microcontroller via the USB by using the PC. Before any operation on data, it is essential to have transaction between the USB and LabVIEW application. It will be explained how to connect the LabVIEW application with data bus.

After this the LabVIEW application should be capable to process the data pack of the samples in order to detect the taken samples from received data. It was mentioned that the taken samples from signal must be spitted into two parts in order to use the USB buffer for sending them through the USB port. It means before sending samples to the PC, the samples

must be divided into two 8 bites (two bytes). The LabVIEW application should join them together to rebuild the right sample values which it was translated by the ADC in the microcontroller. The LabVIEW application also must plot detected samples on the PC.

The first part of the LabVIEW application should be able read and write data by the USB port (serial ports of the PC); VISA functions should be used to satisfy this request. In order to solve this problem, it is necessary to applied VISA resource name, VISA Open, Visa Read, VISA Write and VISA Close. Figure 21 shows the program for making transaction.

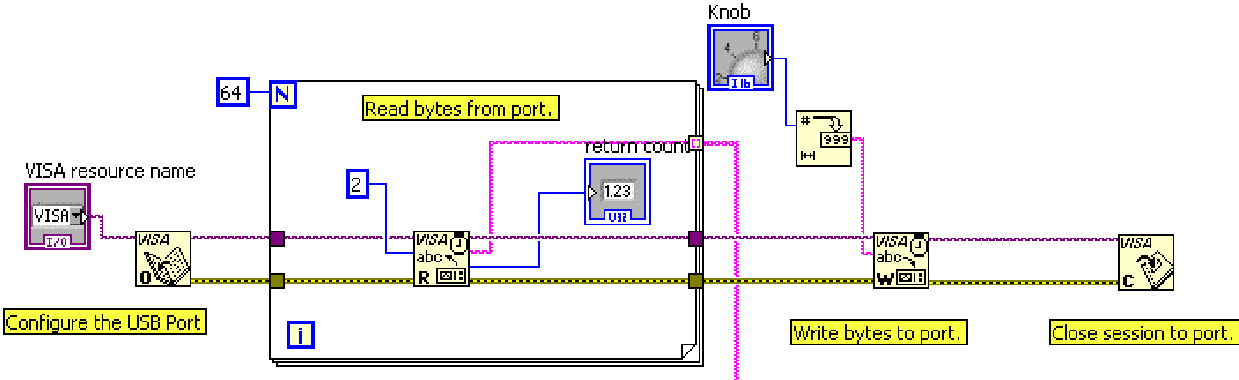

Figure 21: Program for creating communication between the USB port and LabVIEW

By Appling this code, LabVIEW application can recognize the USB device (MCU), it will be recognize by using VISA Resource Name function. VISA Resource Name function specifies the device by the USB RAW device format.

USB[board]:: manufacturer ID::model code:: serial number[::USB interface number]::RAW is the format for the USB RAW device and for the USB device in thesis is: USB\VID_FFFF&PID_2008\VER__1.000 and this USB device descriptor should be written on VISA resource name function for making communication between device and VISA.

VISA Open function opens the USB device that is specified by VISA Resource Name. VISA Read function reads data from the USB device and its job is to get samples data which are sent from the microcontroller to the PC via the USB port. *For Loop* executes 64 times to gather all received samples and prepares an array which this array would be used to detect the samples of signal.VISA Write function writes data to the USB device and it receives values from a knob in order to write these values in the microcontroller for changing the sampling frequency (this knob is wired to VISA Write function and can send decimal number to VISA Write function. VISA Close function closes the device that is specified by VISA Resource Name.

It is explained, by using these functions can be managed the USB communication for the LabVIEW application. But it is necessary to have received samples as a pack (array of samples) in LabVIEW application. It was mentioned, a pack of samples with 64 bytes was prepared in the microcontroller and sent to the PC in order to use in LabVIEW application. By using *For loop*, it is possible to recreate a 1D (one dimensional) array in LabVIEW application. Figure 7 shows a *For loop* which it is used for VISA Read function and it surrounded this function to make a 1D array and its size is 64. There is a box in the top- left corner of the *For loop* that it connected to N, there is a number inside this box. This number defines how many times the function should read the samples value and after reaching 64

samples, 1D array (its size is 64) is ready (a pack of samples with 64 elements) to send to the next step in order to detect the original samples from these samples (as the same as prepared samples by ADC).

A pack of samples is available in order to reconstruct continues signal. As it is remembered, 32 samples were taken from analogue signal by the ADC and microcontroller program saved them for sending to the PC. The USB buffer just has capacity to transfer 8 bites, so samples were split into two parts in order to preserve the data completely. So, the number of sent buffers to the PC is 64 and after getting them by LabVIEW application, they must join together in order to recover the original value of the samples again. Figure 22 presents the part of program which must detect samples of the original signals again.

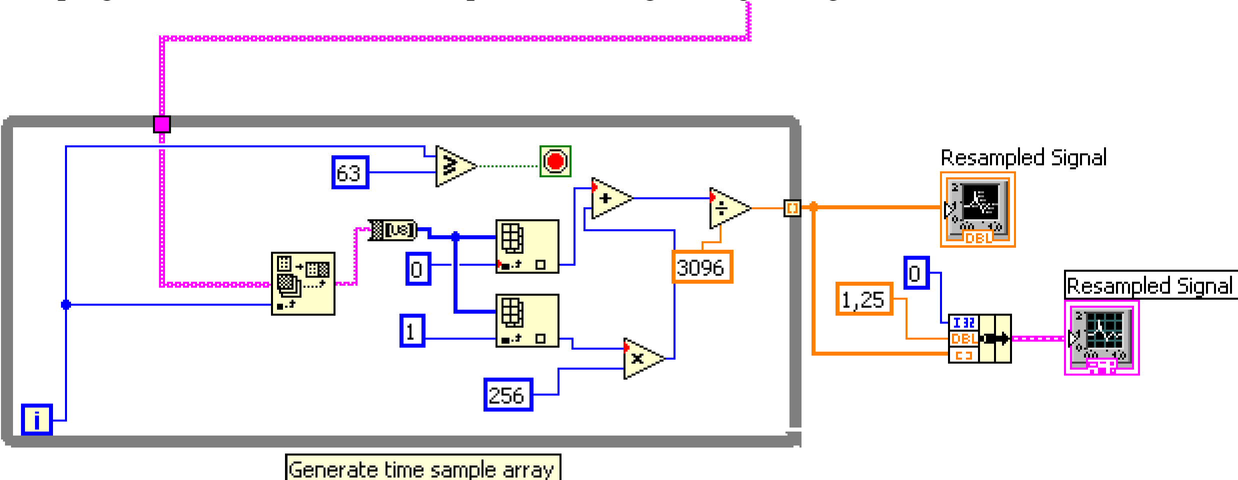

Figure 22: Detecting the real samples and Reconstructing the signal

This part of program consists of *Index String Array, String to* Byte *Array, Grater to Equal?, Index Array, Add, Multiply and Divide* function which they also surrounded by *While Loop*.

A pack of string array would be prepared in previous part and it sent to detect real value of samples (before splitting). The Program in figure 22 receives the samples and detects the most valuable and least valuable part of samples, then recreate the original sample which it was taken by the ADC.

There are functions in LabVIEW program which they can be applied to plot the signal by using the samples value as well as *Waveform chart*, *Waveform graph*, etc. The Waveform chart is used to rebuild the signal in this these.

The input of the Waveform chart is a 1D array which it produced by the While Loop. 1D samples array send to Waveform chart directly and it rebuild the signal. For another one, samples array send to the *Bundle* function and its output is sent to the Waveform chart in order to resample the signal. The result of the both methods is the same, but the second one can set the start point of the signal on the X-dimension and the space or distance interval between samples on the chart. One of the inputs of the Bundle is zero and another is 1.25. They mention, plotted signal on the chart is started from zero point and the space between each samples is 1.25 units.

4.3 Testing the Project

It was discussed regarding the microcontroller and LabVIEW solution, both part should be joined together in order to observe the result.

For testing them, an analogue signal with 500 Hz frequency is generated as an input signal for the microcontroller' ADC and the USB ports of the MCU should be plugged in to PC by cable. The reconstructed signal would be presented on the front panel.

The LabVIEW application should be run and the Sampling frequency is set to 4 KHz. Figure 23 presents the reconstructed signal with both waveform on the front panel.

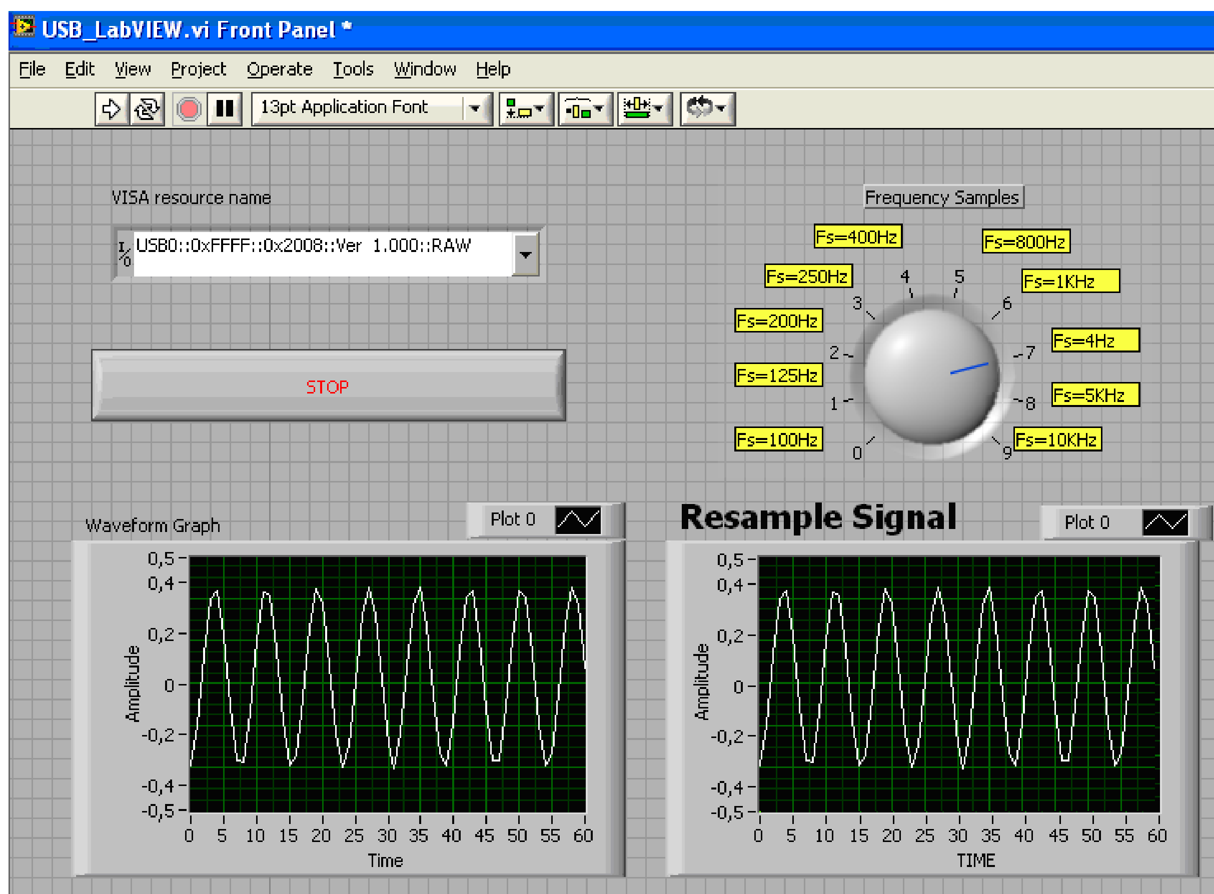

Figure 23: Reconstructed analogue signal with Fs = 4 KHz

It is clear that reconstructed signal is not smooth enough, because the number of samples is not enough. In order to enhance the signal, the sampling frequency should be increased.

This time the sampling frequency is increased to 10 KHz, so the reconstructed signal must become much better than reconstructed signal with Fs = 4 KHz.

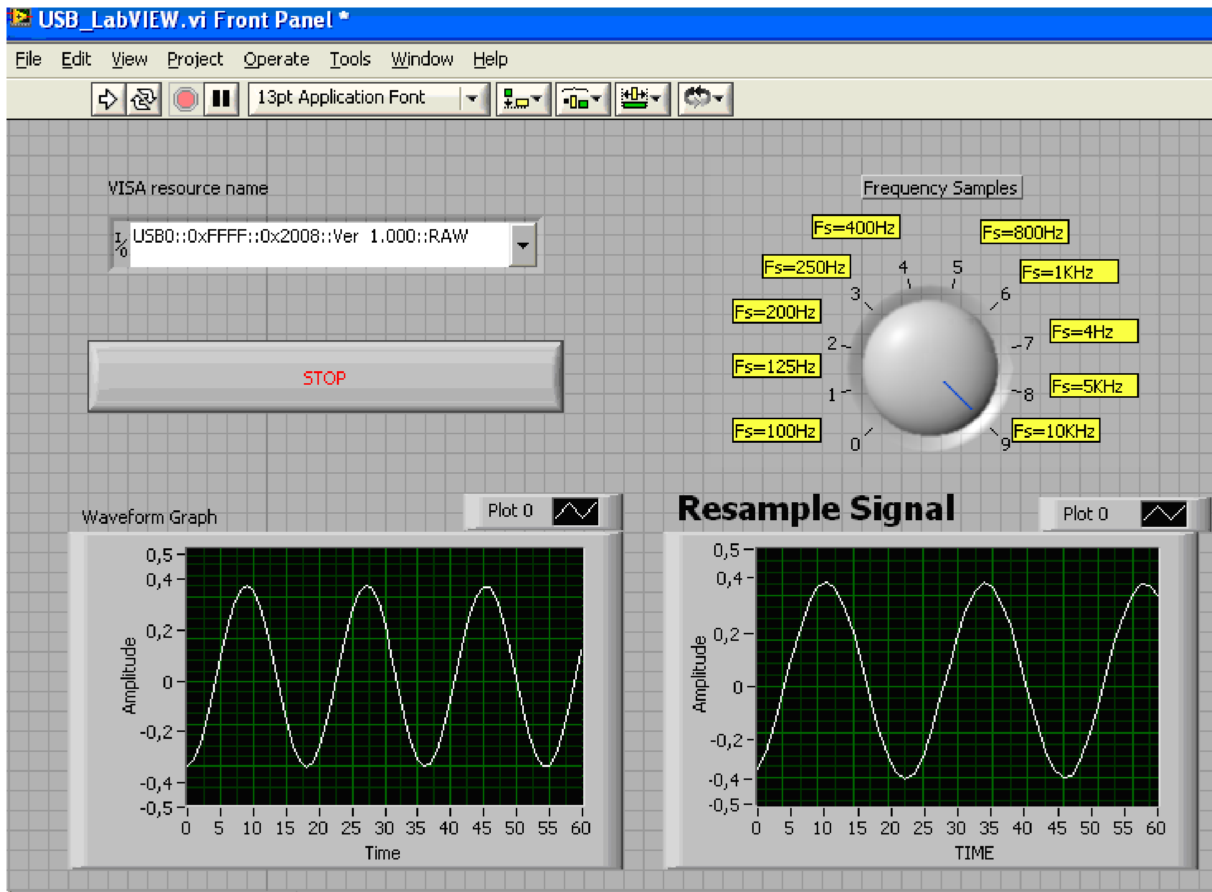

Figure 24: Reconstructed analogue signal with Fs = 10 KHz

The reconstructed signal with Fs = 10 KHz is shown by figure 24. It was seen that the microcontroller program and LabVIEW application could cooperate together completely in order to sample analogue signal and resample it again on the PC screen.

5 Conclusion

In this work, it has been achieve to apply the STM32F10xxx microcontroller, compute and LabVIEW program in order to prepare an interface work. It is discussed what have been exactly demonstrate at this work and it is mentioned in the below:

1) To receive analogue signal as input and creating the pack of digital samples from signal, and also it is proved the prepared samples are correct and accurate before sending the to computer via USB.

2) To prepare the USB port in order to create transaction between the microcontroller and computer.

3) To make a proper driver for the USB device (microcontroller) in order to specify the USB device for LabVIEW application by using LabVIEW VISA.

4) To create a LabVIEW application by using LabVIEW program in order to detect analogue samples from pack of samples for plotting signal on the screen again and change sampling frequency in side the microcontroller manually.

The result of this work would be observed on the PC screen and proves which all procedures have accomplished successfully. There is just one point regarding number of samples which should be mentioned. In order to reconstruct a very well signal especially during increasing the sampling frequency, it was much better to have a package of samples with more samples of signal. It is remembered, a package of samples consists of 32 samples of input signal. By increasing this number to 128 or 256 samples per package, this work was really perfect and creates a better result especially during increasing the frequency sampling.

It is possible to be continued this project, because applying the microcontroller, C program and LabVIEW increases the flexibility of this work in order to improve it for applying in other applications. The microcontroller and PC (LabVIEW program) have communication together, so they could have cooperation to cover a large area.

There are a lot of operations or mathematical calculation that could be achieved by microcontroller. It can be taken FFT, Power Spectrum, mean, filtering, enhancing the samples value by preparing a proper C program for the microcontroller. It is also possible to use keyboard, switches, etc in order to write information inside the microcontroller. Then, result of these calculations or procedures would be easily presented on PC LabVIEW application or is also possible to control the PC LabVIEW application from microcontroller.

In other hand, LabVIEW program is also a very powerful tool to apply for creating many useful applications. LabVIEW program prepares graphical functions which they can be use in Analysing and Processing Signals, User Interfaces, Communication with External Applications (Matlab, etc), Hardware Input and Output (GPIO, Serial, VISA, etc), Toolkits and Modules Not Installed (FPGA, Internet, Signal Processing, etc), etc.
By using these LabVIEW abilities, it is possible to solve complicated problems and applying

its result in order to control microcontroller for displaying result by LCD or Seven Segments. It is possible to drive microcontroller for controlling relay or other connected peripheral from PC (LabVIEW application).

APPENDIX A

This part is to present necessary code for the microcontroller.

A.0 Main program

Microcontroller is begun to operate with the Main program of the microcontroller program.

```
/******************** (C) COPYRIGHT 2008 Baltic Engineering AB ********************
* File Name          : main.c
* Author             : Ehsan Tehrani
* Date First Issued  : 2008/11/04
* Description        : USB to LabView communication main file
********************************************************************************/
int main(void)
{

  /* System clocks configuration ---------------------------------------------*/
  RCC_Configuration();

 /* NVIC configuration ------------------------------------------------------*/
  NVIC_Configuration();

 /* GPIO configuration ------------------------------------------------------*/
  GPIO_Configuration();

/* ADC1 configuration ------------------------------------------------------*/
  ADC1_Configuration();

 /* TIM1 configuration ------------------------------------------------------*/
  TIM1_Configuration();

/* USB configuration -------------------------------------------------------*/
  Set_USBClock();
  USB_Interrupts_Config();
  USB_Init();

/* LCD configuration -------------------------------------------------------*/
  HD44780_PowerUpInit();
  // Show message on the LCD
  LCD_LIGHT_ON();
  HD44780_StrShow(1, 1,  "  IAR Systems   ");
  HD44780_StrShow(1, 2,  "    STM32-SK    ");
  Dly100us((void*)(10000));

  StartToSample();

while (1)  //Main Loop

  {

 //It is to read received data from PC via USB
  if (count_out != 0)
    {
     USB_Get_Data(&buffer_out[0], count_out);
     count_out = 0;
    }
```

```
 LCD = PCtoUSB - 48;
  void Fs_presented (void) ;

 TIM1_Cmd(DISABLE);
 TIM1_Configuration();

//Wait until DIGITAL Samples completion(DIGITAL Sampling is Finished)
  while(ArrayDigitalSamples == FALSE);

//Send Digital Samples to PC Via USB Port
  Coun = 0;

 //Send Digital Samples to PC Via USB Port
  for(i = 0; i <32; i++)
    {
       buffer_in[Coun++] = Sample[i];
       buffer_in[Coun++] = Sample[i]>>8;
    }

     count_in =64;//count_in =64;
     USB_Send_Data();

     //Start To Sample again
      StartToSample();

   // Show the current value of ADC1 on LCD
  sprintf(Str,"AN_TR:%3d       ",LCD);// Sample[Sample_Count] or PCtoUSB
  HD44780_StrShow(1, 2,(pInt8S)Str);

  }
}
/****************** (C) COPYRIGHT 2008 Baltic Engineering AB *****END OF FILE****/
```

A.1 Configuration Code

The configuration code for the microcontroller CPU and peripherals is available in the below.

```
/*******************************************************************************
* Function Name  : RCC_Configuration (Set_System)
* Description    : Configures the different system clocks
* Input          : None.
* Return         : None.
*******************************************************************************/
void RCC_Configuration(void)
{

GPIO_InitTypeDef GPIO_InitStructure;

/* TIM2 clock enable */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);

 /* SYSCLK, HCLK, PCLK2 and PCLK1 configuration -----------------------------*/
 /* RCC system reset(for debug purpose) */

 RCC_DeInit();
```

```
  /* Enable HSE */
  RCC_HSEConfig(RCC_HSE_ON);

  /* Wait till HSE is ready */
  HSEStartUpStatus = RCC_WaitForHSEStartUp();

  if(HSEStartUpStatus == SUCCESS)
  {
    /* HCLK = SYSCLK */
    RCC_HCLKConfig(RCC_SYSCLK_Div1);

    /* PCLK2 = HCLK */
    RCC_PCLK2Config(RCC_HCLK_Div1);

    /* PCLK1 = HCLK/2 */
    RCC_PCLK1Config(RCC_HCLK_Div2);

    /* ADCCLK = PCLK2/4 */
    RCC_ADCCLKConfig(RCC_PCLK2_Div4);

    /* Flash 2 wait state */
    FLASH_SetLatency(FLASH_Latency_2);

    /* Enable Prefetch Buffer */
    FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);

    /* PLLCLK = 8MHz * 9 = 72 MHz */
    RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9);

    /* Enable PLL */
    RCC_PLLCmd(ENABLE);

    /* Wait till PLL is ready */
    while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET)
    {
    }

    /* Select PLL as system clock source */
    RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);

    /* Wait till PLL is used as system clock source */
    while(RCC_GetSYSCLKSource() != 0x08)
    {
    }
  }

/* Enable peripheral clocks --------------------------------------------------*/
/* Enable ADC1, GPIOA, GPIOB and GPIOC clock */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB |
                         RCC_APB2Periph_GPIOC, ENABLE);

}
```

```
/*******************************************************************************
* Function Name  : GPIO_Configuration
* Description    : Configures the different GPIO ports.
* Input          : None
* Output         : None
* Return         : None
*******************************************************************************/
void GPIO_Configuration(void)
{
  GPIO_InitTypeDef GPIO_InitStructure;

  /* Configure PA.01 (ADC Channel1) as analog input(To Use the Mic) -------------------------*/
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
   GPIO_Init(GPIOA, &GPIO_InitStructure);

  // Enable GPIO clock and release reset
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB |
                         RCC_APB2Periph_GPIOC, ENABLE);
 RCC_APB1PeriphResetCmd(RCC_APB2Periph_GPIOA |
                  RCC_APB2Periph_GPIOB | RCC_APB2Periph_GPIOC, ENABLE);

    /* Configure all the GPIOC in Input Floating mode */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(GPIOC, &GPIO_InitStructure);
    GPIO_WriteBit(GPIOC,GPIO_Pin_7 | GPIO_Pin_6,Bit_SET);

  /* PB.01 used as USB pull-up */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_Init(GPIOB, &GPIO_InitStructure);

}
/*******************************************************************************
* Function Name  : ADC1_Configuration
* Description    : Configures the ADC.
* Input          : None
* Output         : None
* Return         : None
*******************************************************************************/
void ADC1_Configuration(void)
{
ADC_InitTypeDef ADC_InitStructure;

  ADC_DeInit(ADC1);

  // ADC Structure Initialization
  ADC_StructInit(&ADC_InitStructure);

  ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
  ADC_InitStructure.ADC_ScanConvMode = DISABLE;
  ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;
  ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
  ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
  ADC_InitStructure.ADC_NbrOfChannel = 1;
  ADC_Init(ADC1, &ADC_InitStructure);
```

```
/* ADC1 regular channel1 configuration */
  ADC_RegularChannelConfig(ADC1,ADC_Channel_1,1,ADC_SampleTime_55Cycles5);

// Enable the ADC1
  ADC_Cmd(ADC1, ENABLE);

  // ADC1 Reset calibration
  ADC_ResetCalibration(ADC1);
  while(ADC_GetResetCalibrationStatus(ADC1) == SET);

  // ADC1 Calibration
  ADC_StartCalibration(ADC1);
  while(ADC_GetCalibrationStatus(ADC1) == SET);
}


/*******************************************************************************
* Function Name  : TIM1_Configuration
* Description    : Configures the Timer.
* Input          : None
* Output         : None
* Return         : None
*******************************************************************************/
u32 TS = 10;
void TIM1_Configuration(void)
{
 //NVIC_InitTypeDef NVIC_InitStructure;
 TIM1_TimeBaseInitTypeDef TIM1_TimeBaseInitStruct;

 // Timer1 Init
  // Enable Timer1 clock and release reset
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1,ENABLE);
  RCC_APB1PeriphClockCmd(RCC_APB2Periph_TIM1,ENABLE);

 // Set timer period 1 sec              //10us
  TIM1_TimeBaseInitStruct.TIM1_Prescaler = 719;//0x1C1F = 7199 for 10kHz or 0.1ms resolution
  TIM1_TimeBaseInitStruct.TIM1_CounterMode = TIM1_CounterMode_Up;
  TIM1_TimeBaseInitStruct.TIM1_Period = TS;//0x2710=10000;  for 1sec Delay
  TIM1_TimeBaseInitStruct.TIM1_ClockDivision = TIM1_CKD_DIV1;
  TIM1_TimeBaseInitStruct.TIM1_RepetitionCounter = 0;
  TIM1_TimeBaseInit(&TIM1_TimeBaseInitStruct);

  // Enable timer counting
  TIM1_Cmd(ENABLE);

  // Clear update interrupt bit
  TIM1_ClearITPendingBit(TIM1_FLAG_Update);

  // Enable update interrupt
TIM1_ITConfig(TIM1_FLAG_Update,ENABLE);
```

```
/*******************************************************************************
* Function Name  : NVIC_Configuration
* Description    : Configures Vector Table base location.
* Input          : None
* Output         : None
* Return         : None
*******************************************************************************/
void NVIC_Configuration(void)
{
  NVIC_InitTypeDef NVIC_InitStructure;

#ifdef  VECT_TAB_RAM
  /* Set the Vector Table base location at 0x20000000 */
  NVIC_SetVectorTable(NVIC_VectTab_RAM, 0x0);
#else  /* VECT_TAB_FLASH  */
  /* Set the Vector Table base location at 0x08000000 */
  NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0);
#endif

  /* Configure one bit for preemption priority */
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);

  /* Enable the USB Interrupt*/
  NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN_RX0_IRQChannel;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);

  /* Enable the ADC1 Interrupt */
  NVIC_InitStructure.NVIC_IRQChannel = ADC_IRQChannel;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);

  /* Enable the TIM1 Interrupt */
  NVIC_InitStructure.NVIC_IRQChannel = TIM1_UP_IRQChannel;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}
/*******************************************************************************
* Function Name  : Set_USBClock
* Description    : Configures USB Clock input (48MHz)
* Input          : None.
* Return         : None.
*******************************************************************************/
void Set_USBClock(void)
{
  /* USBCLK = PLLCLK / 1.5 */
  RCC_USBCLKConfig(RCC_USBCLKSource_PLLCLK_1Div5);

  /* Enable USB clock */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_USB, ENABLE);
}
```

A.2 Code for Sampling

After executing the configuration code, this function StartToSample(); is executed in order to prepare the timer interrupt flag for preparing a pack of samples. This function is consists of following code.

```
void StartToSample(void)
{
   //ArrayDigitalSamples = FALSE;
  Sample_Count = 0;

  // Clear update interrupt bit
  TIM1_ClearITPendingBit(TIM1_FLAG_Update);

  // Enable update interrupt
  TIM1_ITConfig(TIM1_FLAG_Update,ENABLE);

 // Enable the EOC interrupt Flag
  ADC_ITConfig(ADC1, ADC_IT_EOC, ENABLE);

  //Start the ADC1 Conversion
  //ADC_SoftwareStartConvCmd(ADC1, ENABLE);

  //Enable TIM1counting to Generate Time Sampling
  //TIM1_Cmd(ENABLE);

  //DIGITAL Samples has  not been ready yet
 ArrayDigitalSamples = FALSE;//In order to undrestand when ADC1 samplse are ready
}
```

Now timer is starting to work completely in order to count period time and generate interrupt after each over flow. When the timer interrupt is occurred following codes would be executed.

```
TIM1_UP_IRQHandler( )
{
  /*Timer0IntrHandler();*/

    // Clear update interrupt bit
  TIM1_ClearITPendingBit(TIM1_FLAG_Update);

    // Start the conversion
  //ADC_SoftwareStartConvCmd(ADC1, ENABLE);

  // Start the conversion
  ADC_SoftwareStartConvCmd(ADC1, ENABLE);

  /* Toggle PC.06 pin */  /*This is just to turn on LEDs.*/
  GPIO_WriteBit(GPIOC, GPIO_Pin_6, (BitAction)(1 - GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_6)));
  GPIO_WriteBit(GPIOC, GPIO_Pin_7, (BitAction)(1 - GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_7)));

}
```

Above code prepares timer to generate interrupt again and active the ADC1 to operate conversion for taking samples of input signal. After operating above code, the ADC1 begins working and it generates an interrupt when the conversion is finished or the output of ADC1 is ready to save and the below code is executed after the ADC1 interrupt.

```
ADC_IRQHandler( )
{
 /* Clear ADC1 EOC pending interrupt bit */
 ADC_ClearITPendingBit(ADC1, ADC_IT_EOC);

  /* Get regular channel15 converted value */
 Pos = ADC_GetConversionValue(ADC1);

 /*Save to Array Samples*/
 Sample[Sample_Count] = Pos;

 // Stop the conversion
 ADC_SoftwareStartConvCmd(ADC1, DISABLE);

  Sample_Count++;

if(Sample_Count > 31)

 {
              //To inform Main Loop which sampling process is finished
             ArrayDigitalSamples = TRUE;
             Sample_Count = 0;
 }
}
```

Above code takes 32 samples of analogue signal and build the pack of samples. It can be explained which timer generates interrupts and these interrupts has been occurred with equal time intervals. After each timer interrupts, the ADC1 is turned on and one sample is taken. In order to prepare a pack of samples, these procedures should be done continuously.

A.3 Code for USB Transaction

There is a main Loop inside the main program for the microcontroller which it has three major responsibilities. This loop is an unlimited loop and its jobs is to check whenever the pack of samples is ready, to receive new frequency sampling from PC via USB to reload the timer and sending the pack of samples to PC through USB. This main loop starts with following code.

```
//It is to read received data from PC via USB
  if (count_out != 0)
    {
      USB_Get_Data(&buffer_out[0], count_out);
      count_out = 0;
    }
 // Detect New Fs to reload the timer
 LCD = PCtoUSB - 48;
 void Fs_presented (void) ;
 TIM1_Cmd(DISABLE);
 TIM1_Configuration();
```

Received data from PC is obtained by the above code and new Fs is detected to reload the timer again. Then, the main loop is waiting for pack of samples to be ready to send to PC by using following code.

```
//Wait until DIGITAL Samples completion(DIGITAL Sampling is Finished)
  while(ArrayDigitalSamples == FALSE);

//Send Digital Samples to PC Via USB Port
  Coun = 0;
  for(i = 0; i <32; i++)
    {
      buffer_in[Coun++] = Sample[i];
      buffer_in[Coun++] = Sample[i]>>8;
    }
     count_in =64;
     USB_Send_Data();
```

By using this main loop, the program always checks USB transaction.

A.4 Code of USB files

The code of usb_desc.c, usb_desc.h, usb_endp.c and usb_prop.c files is mentioned in this part.

```
/******************* (C) COPYRIGHT 2008 Baltic Engineering AB ********************
* File Name          : usb_desc.c
* Author             : Ehsan Tehrani
* Date First Issued  : 2008/11/04
* Description        : Descriptors for USB to LabView communication
********************************************************************************
* History:
* 2008/11/04: V0.0
********************************************************************************/
/* Includes ------------------------------------------------------------------*/
#include "usb_lib.h"
#include "usb_desc.h"
/* USB Standard Device Descriptor */
const u8 USB_LabView_Port_DeviceDescriptor[] =
  {
    0x12,   /* bLength */
    USB_DEVICE_DESCRIPTOR_TYPE,     /* bDescriptorType */
    0x00,
    0x02,   /* bcdUSB = 2.00 */
    0x00,   /* bDeviceClass  */
    0x00,   /* bDeviceSubClass */
    0x00,   /* bDeviceProtocol */
    0x40,   /* bMaxPacketSize0 */
    0xff,
    0xff,   /* idVendor = 0x0ffff */
    0x08,
    0x20,   /* idProduct = 0x2008 */
    0x00,
    0x00,   /* bcdDevice = 0.00 */
    1,            /* Index of string descriptor describing manufacturer */
    2,            /* Index of string descriptor describing product */
    3,            /* Index of string descriptor describing the device's serial number */
    0x01    /* bNumConfigurations */
  };

const u8 USB_LabView_Port_ConfigDescriptor[] =
  {
    /*Configuration Descriptor*/
    0x09,   /* bLength: Configuation Descriptor size */
    USB_CONFIGURATION_DESCRIPTOR_TYPE,      /* bDescriptorType: Configuration */
    USB_LABVIEW_PORT_SIZ_CONFIG_DESC,       /* wTotalLength:no of returned bytes */
    0x00,
    0x01,   /* bNumInterfaces: 1 interface */
    0x01,   /* bConfigurationValue: Configuration value */
    0x00,   /* iConfiguration: Index of string descriptor describing the configuration */
    0xC0,   /* bmAttributes: self powered */
    0x00,   /* MaxPower 0 mA */
    /*Interface Descriptor*/
    0x09,   /* bLength: Interface Descriptor size */
    USB_INTERFACE_DESCRIPTOR_TYPE,  /* bDescriptorType: Interface */
    /* Interface descriptor type */
    0x00,   /* bInterfaceNumber: Number of Interface */
    0x00,   /* bAlternateSetting: Alternate setting */
    0x02,   /* bNumEndpoints: Two endpoints used */
```

```
    0x00,   /* bInterfaceClass */
    0x00,   /* bInterfaceSubClass */
    0x00,   /* bInterfaceProtocol */
    0x00 ,   /* iInterface: */

    /*Endpoint 3 Descriptor*/
    0x07,   /* bLength: Endpoint Descriptor size */
    USB_ENDPOINT_DESCRIPTOR_TYPE,   /* bDescriptorType: Endpoint */
    0x03,   /* bEndpointAddress: (OUT3) */
    0x02,   /* bmAttributes: Bulk */
    USB_LABVIEW_PORT_DATA_SIZE,             /* wMaxPacketSize: */
    0x00,
    0x00,   /* bInterval: ignore for Bulk transfer */
    /*Endpoint 1 Descriptor*/
    0x07,   /* bLength: Endpoint Descriptor size */
    USB_ENDPOINT_DESCRIPTOR_TYPE,   /* bDescriptorType: Endpoint */
    0x81,   /* bEndpointAddress: (IN1) */
    0x02,   /* bmAttributes: Bulk */
    USB_LABVIEW_PORT_DATA_SIZE,             /* wMaxPacketSize: */
    0x00,
    0x00    /* bInterval: ignore for Bulk transfer */
  };
/* USB String Descriptors */
const u8 USB_LabView_Port_StringLangID[USB_LABVIEW_PORT_SIZ_STRING_LANGID] =
  {
    USB_LABVIEW_PORT_SIZ_STRING_LANGID,           /* bLength */
    USB_STRING_DESCRIPTOR_TYPE,                   /* bDescriptorType*/
    0x09,
    0x04 /* LangID = 0x0409: U.S. English */
  };
const u8 USB_LabView_Port_StringVendor[USB_LABVIEW_PORT_SIZ_STRING_VENDOR] =
  {
    USB_LABVIEW_PORT_SIZ_STRING_VENDOR,     /* bDescriptorType*//* Size of Vendor string */
    USB_STRING_DESCRIPTOR_TYPE,             /* bDescriptorType*/
    /* Manufacturer: "BALTIC Engineering" */
    'B', 0, 'A', 0, 'L', 0, 'T', 0, 'I', 0, 'C', 0, ' ', 0, 'E', 0,
    'n', 0, 'g', 0, 'i', 0, 'n', 0, 'e', 0, 'e', 0, 'r', 0, 'i', 0,
    'n', 0, 'g', 0
  };
const u8 USB_LabView_Port_StringProduct[USB_LABVIEW_PORT_SIZ_STRING_PRODUCT] =
  {
    USB_LABVIEW_PORT_SIZ_STRING_PRODUCT,          /* bLength */
    USB_STRING_DESCRIPTOR_TYPE,        /* bDescriptorType */
    /* Product name: "USB to LabView If BE 2008" */
    'U', 0, 'S', 0, 'B', 0, ' ', 0, 't', 0, 'o', 0, ' ', 0, 'L', 0,
    'a', 0, 'b', 0, 'V', 0, 'i', 0, 'e', 0, 'w', 0, ' ', 0, 'I', 0,
    'f', 0, ' ', 0, 'B', 0, 'E', 0, ' ', 0, '2', 0, '0', 0, '0', 0,
    '8', 0
  };
const u8 USB_LabView_Port_StringSerial[USB_LABVIEW_PORT_SIZ_STRING_SERIAL] =
  {
    USB_LABVIEW_PORT_SIZ_STRING_SERIAL,           /* bLength */
    USB_STRING_DESCRIPTOR_TYPE,        /* bDescriptorType */
    /* Serial no: "Ver  1.000"*/
    'V', 0, 'e', 0, 'r', 0, ' ', 0, ' ', 0, '1', 0, '.', 0, '0', 0,
    '0', 0, '0', 0
  }
  ;/****************************END OF FILE************************************/
```

```
/******************** (C) COPYRIGHT 2008 Baltic Engineering AB ********************
* File Name          : usb_desc.h
* Author             : Ehsan Tehrani
* Date First Issued  : 2008/11/04
* Description        : Descriptor Header for USB to LabView communication
********************************************************************************
* History:
* 2008/11/04: V0.0
********************************************************************************/
/* Define to prevent recursive inclusion -------------------------------------*/
#ifndef __USB_DESC_H
#define __USB_DESC_H

/* Includes ------------------------------------------------------------------*/
/* Exported types ------------------------------------------------------------*/
/* Exported constants --------------------------------------------------------*/
/* Exported macro ------------------------------------------------------------*/
/* Exported define -----------------------------------------------------------*/
#define USB_DEVICE_DESCRIPTOR_TYPE              0x01
#define USB_CONFIGURATION_DESCRIPTOR_TYPE       0x02
#define USB_STRING_DESCRIPTOR_TYPE              0x03
#define USB_INTERFACE_DESCRIPTOR_TYPE           0x04
#define USB_ENDPOINT_DESCRIPTOR_TYPE            0x05

#define USB_LABVIEW_PORT_DATA_SIZE              64
#define USB_LABVIEW_PORT_INT_SIZE               8

#define USB_LABVIEW_PORT_SIZ_DEVICE_DESC        18
#define USB_LABVIEW_PORT_SIZ_CONFIG_DESC        32
#define USB_LABVIEW_PORT_SIZ_STRING_LANGID      4
#define USB_LABVIEW_PORT_SIZ_STRING_VENDOR      38
#define USB_LABVIEW_PORT_SIZ_STRING_PRODUCT     52
#define USB_LABVIEW_PORT_SIZ_STRING_SERIAL      22

#define STANDARD_ENDPOINT_DESC_SIZE             0x09

/* Exported functions ------------------------------------------------------- */
extern const u8 USB_LabView_Port_DeviceDescriptor[USB_LABVIEW_PORT_SIZ_DEVICE_DESC];
extern const u8 USB_LabView_Port_ConfigDescriptor[USB_LABVIEW_PORT_SIZ_CONFIG_DESC];

extern const u8 USB_LabView_Port_StringLangID[USB_LABVIEW_PORT_SIZ_STRING_LANGID];
extern const u8 USB_LabView_Port_StringVendor[USB_LABVIEW_PORT_SIZ_STRING_VENDOR];
extern const u8 USB_LabView_Port_StringProduct[USB_LABVIEW_PORT_SIZ_STRING_PRODUCT];
extern const u8 USB_LabView_Port_StringSerial[USB_LABVIEW_PORT_SIZ_STRING_SERIAL];

#endif /* __USB_DESC_H */
/***********************************END OF FILE*******************************/
```

```
/******************** (C) COPYRIGHT 2008 Baltic Engineering AB ********************
* File Name          : usb_endp.c
* Author             : Ehsan Tehrani
* Date First Issued  : 2008/10/22
* Description        : Endpoint routines
********************************************************************************/

/* Includes ------------------------------------------------------------------*/
#include "usb_lib.h"
#include "usb_desc.h"
#include "usb_mem.h"
#include "hw_config.h"
#include "usb_istr.h"

/* Private typedef -----------------------------------------------------------*/
/* Private define ------------------------------------------------------------*/
/* Private macro -------------------------------------------------------------*/
/* Private variables ---------------------------------------------------------*/
u8 buffer_out[USB_LABVIEW_PORT_DATA_SIZE];
u32 count_out = 0;
u32 count_in = 0;

/* Private function prototypes -----------------------------------------------*/
/* Private functions ---------------------------------------------------------*/
/*******************************************************************************
* Function Name  : EP3_IN_Callback
* Description    :
* Input          : None.
* Output         : None.
* Return         : None.
*******************************************************************************/
void EP3_OUT_Callback(void)
{
  count_out = GetEPRxCount(ENDP3);
  PMAToUserBufferCopy(buffer_out, ENDP3_RXADDR, count_out);
  SetEPRxValid(ENDP3);
}

/*******************************************************************************
* Function Name  : EP1_IN_Callback
* Description    :
* Input          : None.
* Output         : None.
* Return         : None.
*******************************************************************************/
void EP1_IN_Callback(void)
{
  count_in = 0;
}

/******************************************END OF FILE****************************/
```

```
/******************** (C) COPYRIGHT 2008 Baltic Engineering AB ********************
* File Name          : usb_prop.c
* Author             : Ehsan Tehrani
* Date First Issued  : 2008/10/22
* Description        : All processing related to USB to LabView communication
********************************************************************************
* History:
* 2008/10/22: V0.0
********************************************************************************/
/* Includes ------------------------------------------------------------------*/
#include "usb_lib.h"
#include "usb_conf.h"
#include "usb_prop.h"
#include "usb_desc.h"
#include "usb_pwr.h"
#include "hw_config.h"

/* Private typedef -----------------------------------------------------------*/
/* Private define ------------------------------------------------------------*/
/* Private macro -------------------------------------------------------------*/
/* Private variables ---------------------------------------------------------*/
u8 Request = 0;

LINE_CODING linecoding =
  {
    115200, /* baud rate*/
    0x00,   /* stop bits-1*/
    0x00,   /* parity - none*/
    0x08    /* no. of bits 8*/
  };

/* -------------------------------------------------------------------------- */
/*  Structures initializations */
/* -------------------------------------------------------------------------- */

DEVICE Device_Table =
  {
    EP_NUM,
    1
  };

DEVICE_PROP Device_Property =
  {
    USB_LabView_Port_init,
    USB_LabView_Port_Reset,
    USB_LabView_Port_Status_In,
    USB_LabView_Port_Status_Out,
    USB_LabView_Port_Data_Setup,
    USB_LabView_Port_NoData_Setup,
    USB_LabView_Port_Get_Interface_Setting,
    USB_LabView_Port_GetDeviceDescriptor,
    USB_LabView_Port_GetConfigDescriptor,
    USB_LabView_Port_GetStringDescriptor,
    0,
    0x40 /*MAX PACKET SIZE*/
  };
USER_STANDARD_REQUESTS User_Standard_Requests =
  {
    USB_LabView_Port_GetConfiguration,
    USB_LabView_Port_SetConfiguration,
```

```
    USB_LabView_Port_GetInterface,
    USB_LabView_Port_SetInterface,
    USB_LabView_Port_GetStatus,
    USB_LabView_Port_ClearFeature,
    USB_LabView_Port_SetEndPointFeature,
    USB_LabView_Port_SetDeviceFeature,
    USB_LabView_Port_SetDeviceAddress
  };
ONE_DESCRIPTOR Device_Descriptor =
  {
    (u8*)USB_LabView_Port_DeviceDescriptor,
    USB_LABVIEW_PORT_SIZ_DEVICE_DESC
  };

ONE_DESCRIPTOR Config_Descriptor =
  {
    (u8*)USB_LabView_Port_ConfigDescriptor,
    USB_LABVIEW_PORT_SIZ_CONFIG_DESC
  };

ONE_DESCRIPTOR String_Descriptor[4] =
  {
    {(u8*)USB_LabView_Port_StringLangID, USB_LABVIEW_PORT_SIZ_STRING_LANGID},
    {(u8*)USB_LabView_Port_StringVendor, USB_LABVIEW_PORT_SIZ_STRING_VENDOR},
    {(u8*)USB_LabView_Port_StringProduct, USB_LABVIEW_PORT_SIZ_STRING_PRODUCT},
    {(u8*)USB_LabView_Port_StringSerial, USB_LABVIEW_PORT_SIZ_STRING_SERIAL}
  };
/* Extern variables ---------------------------------------------------------*/
/* Private function prototypes -----------------------------------------------*/
/* Extern function prototypes ------------------------------------------------*/
/* Private functions ---------------------------------------------------------*/
/*******************************************************************************
* Function Name  : USB_LabView_Port_init.
* Description    : USB to LabView init routine.
* Input          : None.
* Output         : None.
* Return         : None.
*******************************************************************************/
void USB_LabView_Port_init(void)
{
  pInformation->Current_Configuration = 0;
  /* Connect the device */
  PowerOn();
  /* USB interrupts initialization */
  /* clear pending interrupts */
  _SetISTR(0);
  wInterrupt_Mask = IMR_MSK;
  /* set interrupts mask */
  _SetCNTR(wInterrupt_Mask);
  pInformation->Current_Feature = USB_LabView_Port_ConfigDescriptor[7];
  /* configure the USART 1 to the default settings */
//  USART_Config_Default();
  /* Wait until device is configured */
  while (pInformation->Current_Configuration == 0)
  {
    NOP_Process();
  }
  bDeviceState = CONFIGURED;
}
```

```
/*******************************************************************************
* Function Name  : USB_LabView_Port_Reset
* Description    : USB to LabView reset routine
* Input          : None.
* Output         : None.
* Return         : None.
*******************************************************************************/
void USB_LabView_Port_Reset(void)
{
  /* Set LabView_Port DEVICE as not configured */
  pInformation->Current_Configuration = 0;

  /* Set LabView_Port DEVICE with the default Interface*/
  pInformation->Current_Interface = 0;
  SetBTABLE(BTABLE_ADDRESS);

  /* Initialize Endpoint 0 */
  SetEPType(ENDP0, EP_CONTROL);
  SetEPTxStatus(ENDP0, EP_TX_STALL);
  SetEPRxAddr(ENDP0, ENDP0_RXADDR);
  SetEPTxAddr(ENDP0, ENDP0_TXADDR);
  Clear_Status_Out(ENDP0);
  SetEPRxCount(ENDP0, Device_Property.MaxPacketSize);
  SetEPRxValid(ENDP0);

  /* Initialize Endpoint 1 */
  SetEPType(ENDP1, EP_BULK);
  SetEPTxAddr(ENDP1, ENDP1_TXADDR);
  SetEPTxStatus(ENDP1, EP_TX_NAK);
  SetEPRxStatus(ENDP1, EP_RX_DIS);

  /* Initialize Endpoint 2 */
  SetEPType(ENDP2, EP_INTERRUPT);
  SetEPTxAddr(ENDP2, ENDP2_TXADDR);
  SetEPRxStatus(ENDP2, EP_RX_DIS);
  SetEPTxStatus(ENDP2, EP_TX_NAK);

  /* Initialize Endpoint 3 */
  SetEPType(ENDP3, EP_BULK);
  SetEPRxAddr(ENDP3, ENDP3_RXADDR);
  SetEPRxCount(ENDP3, USB_LABVIEW_PORT_DATA_SIZE);
  SetEPRxStatus(ENDP3, EP_RX_VALID);
  SetEPTxStatus(ENDP3, EP_TX_DIS);

  /* Set this device to response on default address */
  SetDeviceAddress(0);
}
/*******************************************************************************
* Function Name  : USB_LabView_Port_Status_In.
* Description    : USB to LabView Status In Routine.
* Input          : None.
* Output         : None.
* Return         : None.
*******************************************************************************/
void USB_LabView_Port_Status_In(void)
{
  if (Request == SET_LINE_CODING)
  {
//    USART_Config();
    Request = 0;
```

```
  }
}

/*******************************************************************************
* Function Name  : USB_LabView_Port_Status_Out
* Description    : USB to LabView Status OUT Routine.
* Input          : None.
* Output         : None.
* Return         : None.
*******************************************************************************/
void USB_LabView_Port_Status_Out (void)
{
}

/*******************************************************************************
* Function Name  : USB_LabView_Port_Data_Setup
* Description    : handle the data class specific requests
* Input          : Request Nb.
* Output         : None.
* Return         : USB_UNSUPPORT or USB_SUCCESS.
*******************************************************************************/
RESULT USB_LabView_Port_Data_Setup(u8 RequestNo)
{
  u8    *(*CopyRoutine)(u16);

  CopyRoutine = NULL;

  if (CopyRoutine == NULL)
  {
    return USB_UNSUPPORT;
  }

  pInformation->Ctrl_Info.CopyData = CopyRoutine;
  pInformation->Ctrl_Info.Usb_wOffset = 0;
  (*CopyRoutine)(0);
  return USB_SUCCESS;
}
/*******************************************************************************
* Function Name  : USB_LabView_Port_NoData_Setup.
* Description    : handle the no data class specific requests.
* Input          : Request Nb.
* Output         : None.
* Return         : USB_UNSUPPORT or USB_SUCCESS.
*******************************************************************************/
RESULT USB_LabView_Port_NoData_Setup(u8 RequestNo)
{

  if (Type_Recipient == (CLASS_REQUEST | INTERFACE_RECIPIENT))
  {
    if (RequestNo == SET_COMM_FEATURE)
    {
      return USB_SUCCESS;
    }
    else if (RequestNo == SET_CONTROL_LINE_STATE)
    {
      return USB_SUCCESS;
    }
  }

  return USB_UNSUPPORT;
```

```
}
/*******************************************************************************
* Function Name  : USB_LabView_Port_GetDeviceDescriptor.
* Description    : Gets the device descriptor.
* Input          : Length.
* Output         : None.
* Return         : The address of the device descriptor.
*******************************************************************************/
u8 *USB_LabView_Port_GetDeviceDescriptor(u16 Length)
{
  return Standard_GetDescriptorData(Length, &Device_Descriptor );
}
/*******************************************************************************
* Function Name  : USB_LabView_Port_GetConfigDescriptor.
* Description    : get the configuration descriptor.
* Input          : Length.
* Output         : None.
* Return         : The address of the configuration descriptor.
*******************************************************************************/
u8 *USB_LabView_Port_GetConfigDescriptor(u16 Length)
{
  return Standard_GetDescriptorData(Length, &Config_Descriptor );
}

/*******************************************************************************
* Function Name  : USB_LabView_Port_GetStringDescriptor
* Description    : Gets the string descriptors according to the needed index
* Input          : Length.
* Output         : None.
* Return         : The address of the string descriptors.
*******************************************************************************/
u8 *USB_LabView_Port_GetStringDescriptor(u16 Length)
{
  u8 wValue0 = pInformation->USBwValue0;
  return Standard_GetDescriptorData(Length, &String_Descriptor[wValue0] );
}

/*******************************************************************************
* Function Name  : USB_LabView_Port_Get_Interface_Setting.
* Description    : test the interface and the alternate setting according to the
*                  supported one.
* Input1         : u8: Interface : interface number.
* Input2         : u8: AlternateSetting : Alternate Setting number.
* Output         : None.
* Return         : The address of the string descriptors.
*******************************************************************************/
RESULT USB_LabView_Port_Get_Interface_Setting(u8 Interface, u8 AlternateSetting)
{
  if (AlternateSetting > 0)
  {
    return USB_UNSUPPORT;
  }
  else if (Interface > 1)
  {
    return USB_UNSUPPORT;
  }
  return USB_SUCCESS;
}
```

```
/*******************************************************************************
* Function Name  : USB_LabView_Port_GetLineCoding.
* Description    : send the linecoding structure to the PC host.
* Input          : Length.
* Output         : None.
* Return         : lnecoding structure base address.
*******************************************************************************/
u8 *USB_LabView_Port_GetLineCoding(u16 Length)
{
  if (Length == 0)
  {
    pInformation->Ctrl_Info.Usb_wLength = sizeof(linecoding);
    return NULL;
  }
  return(u8 *)&linecoding;
}

/*******************************************************************************
* Function Name  : USB_LabView_Port_SetLineCoding.
* Description    : Set the linecoding structure fields.
* Input          : Length.
* Output         : None.
* Return         : Linecoding structure base address.
*******************************************************************************/
u8 *USB_LabView_Port_SetLineCoding(u16 Length)
{
  if (Length == 0)
  {
    pInformation->Ctrl_Info.Usb_wLength = sizeof(linecoding);
    return NULL;
  }
  return(u8 *)&linecoding;
}

/******************************************END OF FILE*************************/
```

APPENDIX B

Create an INF file

LabVIEW VISA 3.0 or higher version must be installed on the PC in order to make an INF file for creating the driver. Windows must be informed by the VISA Driver Developer Wizard (DDW) to use VISA as default. Controller Board (USB device) should not plug in to the PC.

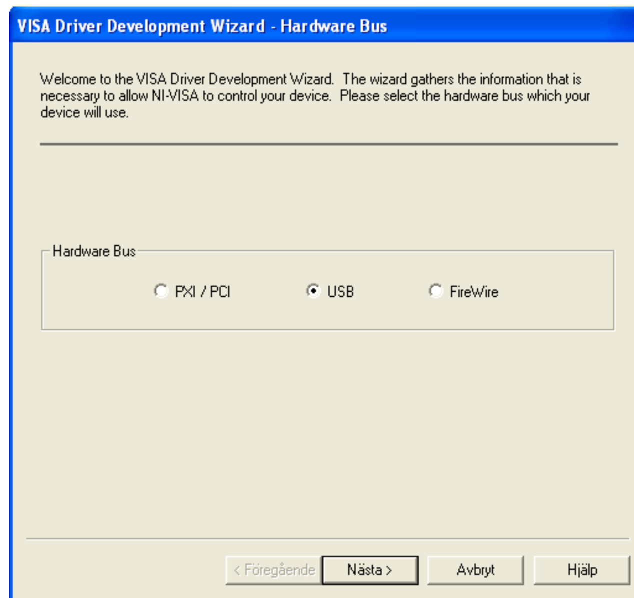

Figure 1: VISA Hardware Bus Selection

First step: DDW is applied by selecting:
Start > Programs > National Instrument > VISA Driver Developer Wizard
in order to create INF file. Figure 1 shows its open screen. Choose USB and press next.

VISA Driver Development Wizard-USB-Basic Device Information window appears on the open screen. See figure 2.

Figure 2: VISA DDW – USB – Basic Device Information

Second step: The empty places on figure 2 should be filled to identify the USB device to PC. If these are not known, USB device should be plug to the PC. The Device Manager can find the new device; it is usually under 'Other Devices'. Click USB device to open USB device Properties. Figure 12 shows the Vendor ID and Product ID to write on right places. The characters to the right of VID_ and PID (USB\VID_FFFF&PID_2008\VER__1.000) introduce the Vendor and Product ID numbers in order to use in figure 3 window.

For this project: Vendor ID: 0xFFFF, Product ID: 0x2008,
Manufacture Name: BALTIC Engineering and Model Name: STM32 are defined.
Unplug the USB device from PC, fill the empty places on figure 2, and click NEXT.

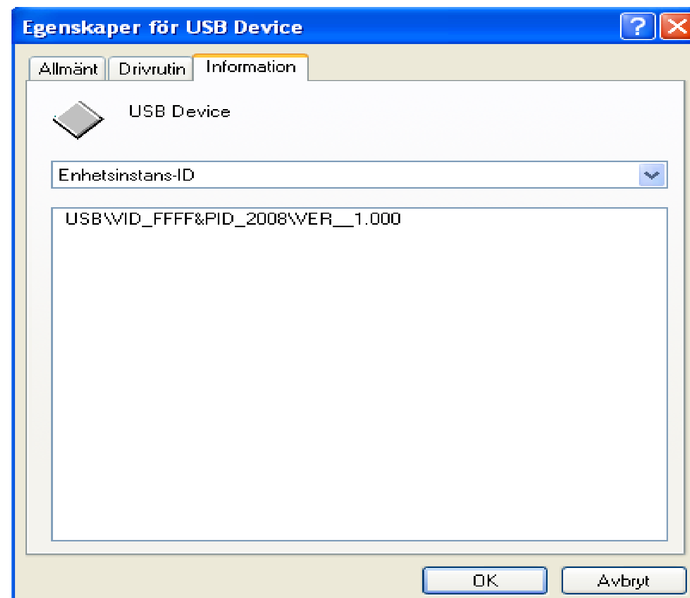

Figure 3: To find Vendor ID and Product ID

The VISA DDW – USB – Output Files Properties is displayed on screen (Figure 4).

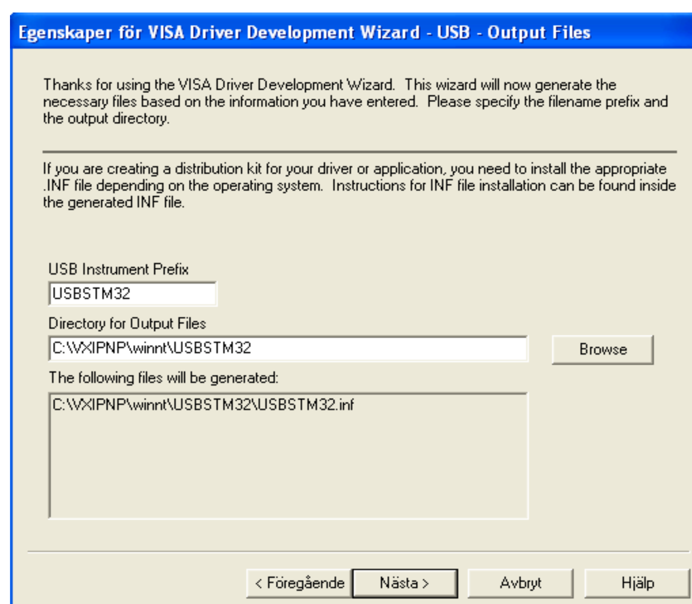

Figure 4: VISA DDW – USB – Output Files Properties

Third step: The USB Instrument Prefix is used to identify the files, which they are for the USB device. A desired name is written for the USB Instrument Prefix (it is optional); USBSTM32 is chosen for this thesis. Figure 4 shows the USB Instrument Prefix name and press NEXT.

Installation option windows appear to install the setup information for operation system. The best option to select is default selection.
Click Finish button to create the INF file in the prepared directory. The INF file is finally created and it should be installed on the operation system. In the next section, it is described how to install the INF file.

Installing the INF file and USB device

The INF file includes an installation instruction that is placed in a header at the top of the INF file. The installation instructions should be study because installing the INF file is different for each version of Windows. Windows XP is installed on PC for this project.

First step: The INF file must be copied to the INF folder which the INF folder can be usually found at C:\WINDOWS\INF.

Second step: To install the INF file, right click on the INF file at C:\WINDOWS\INF and press Install. After installing it a PNF file is made for the USB device. Now, the USB device can be installed.

Third step: Connect the USB device to PC and it is detected by PC because USB is hot pluggable. After connecting the USB device, the Add New Wizard appears automatically. Browse to the INF folder to select the driver which the INF file is created there by using the DDW.
Driver is made and installed on PC, by completing the last step.

CHAPTER 7 REFERENCES

[1] Programming 32-bit Microcontrollers in C (Lucio Di Jacio)

[2] PROGRAMMING LANGUAGE (BRIAN W.KERNIGHAN & DENNIS M.RITCHIE)

[3] USB Complete (JAN AXELSON)

[4] Getting Started with LabVIEW (National Instrument_web site or LabVIEW help)
digital.ni.com/manuals.nsf/websearch/07F9CD824F66237A86257046006F7C5D

[5] LabVIEW For Everyone (JEFFREY TRAVIS & JIM KRING)

[6] Introduction to LabVIEWTM Six-Hour Course (National Instrument_web site)
zone.ni.com/devzone/cda/tut/p/id/5241

[7] USB Instrument Control Tutorial (National Instrument_web site)
zone.ni.com/devzone/cda/tut/p/id/4478

[8] LabVIEW Graphical Programming Course (Connexions)
cnx.org/content/col10241/latest/

[9] Firmware_lib_users_manual

[10] REF_Manual_STM32F103

[11] Cortex_Technical_Reference_Manual

[12] IAR Embedded Workbench IDE User Guide
users.ece.gatech.edu/~mmckeown3/ECE3884/pdf/IAR%20Embedded%20Workbench%20IDE%20User%20Guide%20for%20ARM.pdf

[13] Datablad_STM32F103

[14] STM32F10xxx USB development kit
www.st.com/stonline/products/literature/um/13465.htm

[15] Analogue-to-Digital Converters
ww1.microchip.com/downloads/en/devicedoc/adc.pdf

[16] USB Documents on website
www.beyondlogic.org/usbnutshell/usb1.htm (USB in a Nutshell)
www.usb.org/developers/usb20/ (Universal Serial Bus)