Department of Electrical Engineering

Examensarbete

Implementation of an FMCW Radar Platform With High-Speed Real-Time Interface

Examensarbete utfört inom

Elektroniksystem, Electronics Systems

av Jonny Svensson

LITH-ISY-EX--06/3779--SE

Linköping 2006

TEKNISKA HÖGSKOLAN

LINKÖPINGS UNIVERSITET

Department of Electrical Engineering
Linköping University
S-581 83 Linköping, Sweden

Linköpings tekniska högskola
Institutionen för systemteknik
581 83 Linköping

Implementation of an FMCW Radar Platform With High-Speed Real-Time Interface

.....

.....

Examensarbete utfört i Elektroniksystem

vid Linköpings tekniska högskola

av Jonny Svensson

.....

LITH-ISY-EX--06/3779--SE

Handledare: Niklas U. Andersson

Examinator: Per Löwenborg

Linköping 2006-03-29

Abstract

Acreo AB has developed a radar prototype used for illustrate how the SiGe technology could be used. The radar prototype needs further development with a fast interface and a more integrated design. The beginning of the report describes the radar technique theory and the composing equations. The theoretical background is used to explain each component of the system. The report continues by specifying the target of the next radar prototype. The chosen implementation is motivated and the mode of procedure is described in detail. Test benches were used to verify correct functionality and some limits were found. The report is concluded with test results and recommendations on further enhancements.

Acknowledgements

First I want to thank Acreo AB for the opportunity to perform my master thesis at their office in Norrköping. Behind my results conceals competent employees and high-technology equipment. My supervisor Niklas Andersson has helped me understand the current prototype with the related mathematical background of radar basics. In the area of USB and C programming I want to thank Joacim Haglund for his support and dialogue of experience. Under the continuous project meetings Patrick Blomqvist and Joakim Strömberg has contributed with knowledge and motivating discussions. My examiner Per Löwenborg has supported me in knowledge in ADC area and the administrative tasks. At last I thank Linköpings University for providing me with a fundamental ground which has been useful in completing this work.

Table of Contents

List of Figures

List of Tables

Chapter 1
Terminology

Abbreviation	Explanation
µC	microcontroller
ADC	Analog to Digital Converter
PCB	Printed Circuit Board
DDS	Direct Digital Synthesizer
DLL	Dynamic Link Libraries
FFT	Fast Fourier Transform
FIFO	First In First Out
FMCV	Frequency Modulated Continuous Wave
GaAs	Gallium Arsenide
IC	Integrated Circuit
IF	Intermediate Frequency
LO	Local Oscillator
MCM	Multi Chip Module
PLL	Phase-Locked Loop
Radar	Radio detection and ranging
RF	Radio Frequency
SIE	Serial Interface Engine
SiGe	Silicon / Germanium
SNR	Signal to Noise Ratio
VCO	Voltage Controlled Oscillator

Table 1: Terminology

Chapter 2
Introduction

2.1 Background

Acreo AB is doing research on and develops Silicon - Germanium, SiGe, ICs for 77 GHz FMCW-radar. The SiGe process is a cheaper alternative to the more conventional Gallium Arsenide, GaAs process and will by this prototype be demonstrated as an attractive option.

Through a combination of SiGe ICs and innovative antenna design the cost for this technique attracts new applications and markets as for example the automotive industry. The modern automotive industry involves many possible functions like Adaptive Cruise Control (ACC), Anti-Collision (AC) systems, blind spot surveillance, obstacle detection, parking assistance and other automotive applications. To realise these functions a car needs wide knowledge about its surroundings which increases the number of radar modules. This leads to more sophisticated systems and puts a higher requirement of accuracy. Other possible areas of usability for these modules are speed controls, night vision etc.

To analyze and demonstrate the performance in SiGe solutions, Acreo AB has developed a radar prototype. The radar module is functional but needs to be more compact and have a more straightforward interface to be able evaluate the radar performance.

2.2 FMCW radar basics

2.2.1 Detecting a stationary object

The principle used in FMCW radar is to modulate frequency periodically with time; a technique called FM-ranging. The technique used to detect a stationary object is by transmitting a linear sweep; shown in figure 1.

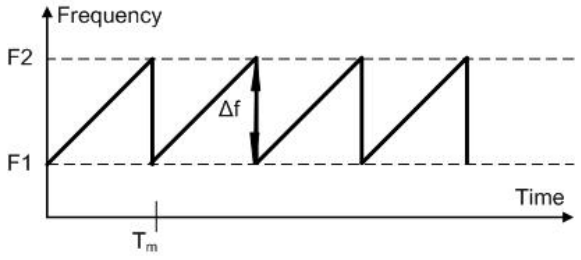


Figure 1: Transmitted signal, stationary object

The echo signal received from a stationary target is delayed by 2R/c, as illustrated in figure 2, where R is the distance to the target and c is the speed of light in vacuum.

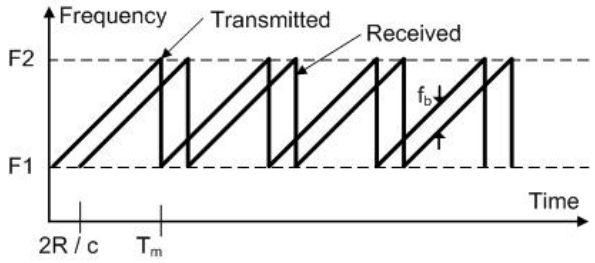


Figure 2: Transmitted and received signal, stationary object

The f_b is called beat frequency and can be calculated through:

$$f_b = \frac{2R}{c}\frac{\Delta f}{T_m} \quad \textbf{, Equation: 1}$$

As seen in figure 2 the beat frequency is the difference in frequency between transmitted and received signal and is proportional to the target R. i.e. the distance to the target is given by:

$$R = \frac{c}{2}T_m\frac{f_b}{\Delta f} \quad \textbf{, Equation: 2}$$

2.2.2 Detecting a moving object

A one-sweep transmission is not sufficient to detect both distance and speed. The transmitted signal is enhanced to send out an up- and down-sweep. The resulting signal has a triangular appearance in the time-frequency plot as shown in Figure 3.

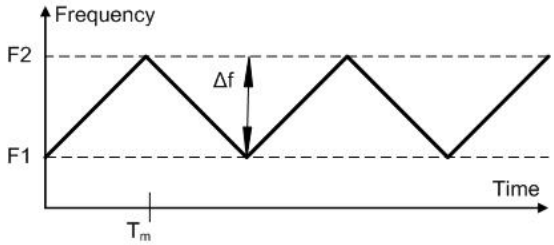


Figure 3: Transmitted signal, moving object

The relative speed causes the received waveform to be Doppler shifted.

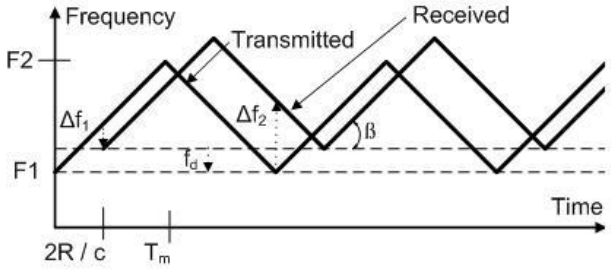


Figure 4: Transmitted and received signal at Doppler shift

The Doppler shift f_d in Figure 4 is calculated through:

$$f_d = \frac{4R}{c}\frac{\Delta f}{T_m} \quad \textbf{, Equation: 3}$$

Where the range R has following relation:

$$R = \frac{(-\Delta f_1 + \Delta f_2)c}{4\tan\beta} \quad \textbf{, Equation: 4}$$

The relative velocity of the signal is determined:

$$V_{rel} = -\frac{f_D * \lambda}{2} \quad \textbf{, Equation: 5}$$

The separation between the waveforms is defined as:

$$\Delta f_1 = f_d - f_b \quad \textbf{, Equation: 6}$$

$$\Delta f_2 = f_d + f_b \quad \textbf{, Equation: 7}$$

The linear relationship between f_b and f_d from equation 6 and 7 can be graphically illustrated:

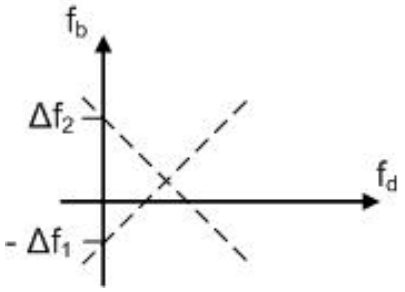


Figure 5: Graphical solution, moving object

Where the cross point between the lines, equals the solution for the f_b and f_d.

2.2.3 Multi-object situation

A third possible scenario is multiple targets. The graphical solution at double target situation would result in the appearance at Figure 6.

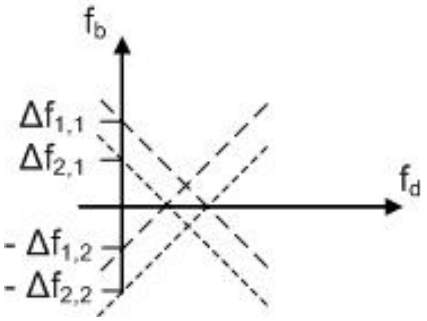


Figure 6: Problem at multi-object situation

The graph shows four possible solutions were only two is leading to correct objects. These mis-associations or ghost targets is a problem needed to be solved.

The last enhancement is to apply different sweep rates as illustrated in the figure below:

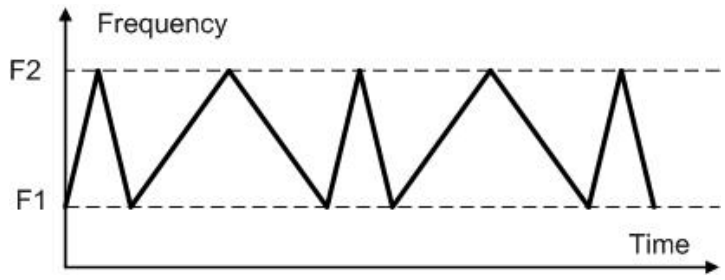


Figure 7: Transmitted signal, multi-object situation

The Figure 8 shows a graphical explanation of the technique:

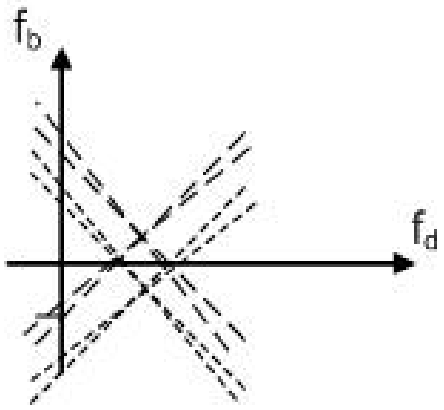


Figure 8: Graphical solution, multi-object situation

The ghost targets could thereby be found and removed from further analysis.

2.3 Radar equation

The radar can be further analyzed by focusing on the antenna signal as illustrated in figure below.

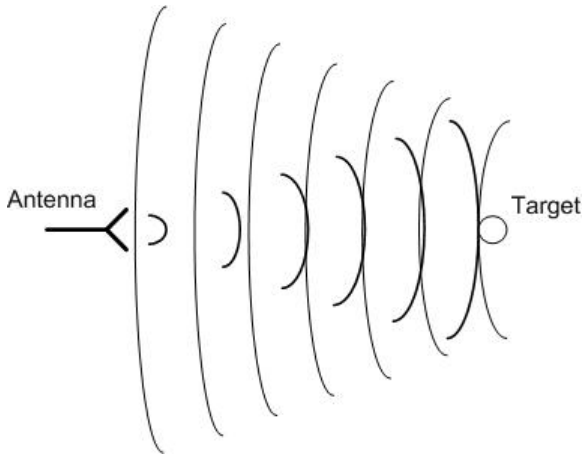


Figure 9: Antenna signal

When a wave is transmitted it will affect the radar target (at range R) with a power density which can be calculated through:

$$W_i = \frac{P_t G_t}{4\pi R^2} \qquad \textbf{, Equation: 8}$$

Where the transmitted power, P_t and gain, G_t is set by the system.

The wave front which touches the target can approximately describe a plane surface with an area σ . This assumption gives the power collected by the target to equal: $P_c = \sigma W_i$

With same principle used in the previous equation, the scattered power in the reflected wave is:

$$W_s = \frac{P_s}{4\pi R^2} \qquad \textbf{, Equation: 9}$$

The scattered power collected is described as: $P_r = W_s A_{er}$ where A_{er} is the effective area of received antenna. The power collected and scattered power is equal by definition ($P_s = P_c$).

The equations given above can be rewritten into:

$$P_r = \frac{P_t G_t \sigma A_{er}}{(4\pi R^2)^2} \quad \textbf{, Equation: 10}$$

This equation is the classic form of the radar range equation [1].

Through focusing on the differential power the following simplification can be made:

$$(P_{loss})_{\log} = -10 \log\left(\frac{P_t G_t \sigma A_{er}}{(4\pi R_2^{\,2})^2} \Big/ \frac{P_t G_t \sigma A_{er}}{(4\pi R_1^{\,2})^2}\right) = -40 * \log(\frac{R_1}{R_2}) \quad \textbf{, Equation: 11}$$

This equation is a useful instrument for relating power to range.

The $\mathbf{P}_{loss}$ = - 12 dB/octave and $\mathbf{P}_{loss}$ = - 40 dB/decade where octave corresponds to 2 times the distance and decade to 10 times distance. I.e. a doubling on the distance to the target corresponds to a power loss of 12dB.

2.4 Frequency bands for FMCW

According to SP Swedish National Testing and Research Institute, two frequency ranges are available for the automotive radar, 24 - 24.25 GHz and 76 - 77 GHz. The 24 GHz-range will not be allowed in new components after 2013, with the consequence of no research and development to proceed at this frequency range.

The next chapter describes the functionality of the current architecture. This requires a good knowledge of each consisting component.

Chapter 3

The hardware configuration

3.1 System blocks

3.1.1 The VCO

The output from a Voltage Controlled Oscillator, VCO is a frequency which is referenced or tuned by the voltage on its input V_{tune}. A fundamental setup consisting of the VCO and a triangle wave generator is illustrated in the figure below.

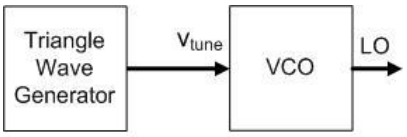


Figure 10: A VCO connection

The triangle wave generator produces a voltage with a triangular appearance which results in the desired signal; described in Figure 3. But the triangle wave generator is not a sufficient solution when there is need for a more dynamical appearance; as in Figure 1 and Figure 7.

3.1.2 The PLL

The Phase-Locked Loop, PLL is a component mainly consisting by a phase detector. The PLL is used to establish a steady phase in a system. The phase detector is comparing the phase between two input signals. One of the inputs is used as a reference while the other is tuned to an equal phase.

The functionality of a PLL is described using a crystal oscillator which is added to provide a fixed frequency, used as the reference. The output of from the phase detector or the error signal is composed of a group of pulses. Figure 11 presents three different signals which describe the functionality of the phase detectors.

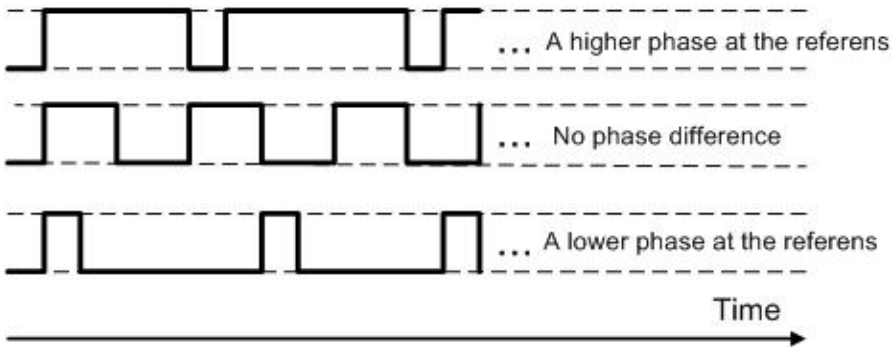


Figure 11: The error signal

Instead of studying the group of pulses the error signal is passed through a filter. The filter is a so called Loop filter which performs a mean value representation on the voltage. The final step is to translate the voltage level to a phase which is connected to the tuned input on the PLL. This is done through a VCO as described in the previous chapter. Figure 12 shows how PLL-loop is constructed.

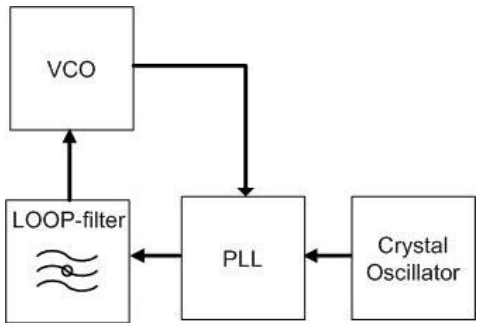


Figure 12: Block diagram of a PLL

A phase difference between the inputs of the PLL will develop to an error signal. The error signal passed through the Loop filter, representing the error as an up or down variation at voltage level. This voltage variation is passed as a variation on the phase performed by the VCO. The altered phase will once again be matched against the reference input on the PLL. The time needed for the PLL:s error signal to reach the centre state is directly describing how good the system can regulate.

The VCO is directly connected to the antenna and will thereby be working at a high frequency as mentioned in chapter 2.4. This will lead to same requirement on the crystal oscillator. But a closer look at the PLL shows a possibility to use dividers at the inputs. This feature makes it possible use a lower frequency on the input connected to the crystal oscillator.

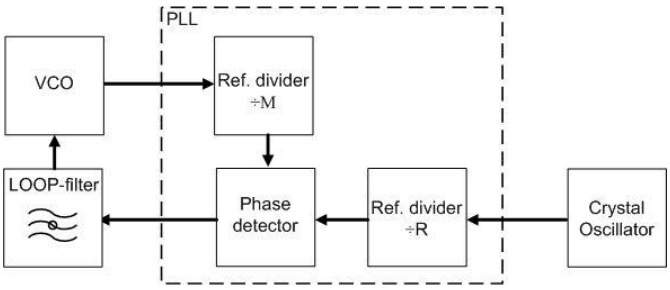


Figure 13: Inside the PLL

The range of the dividers depends on inner structure of PLL.

The crystal oscillator in Figure 12 and Figure 13 sends out a constant frequency. But the desired output is the triangular waveform shown in Figure 3. This is achieved by using a DDS which gives a more dynamical solution. A detailed description over the DDS follows in next chapter.

3.1.3 The DDS

The DDS has a ramped frequency mode which is used to achieve the look of a triangular oscillator. This is done through switching between positive and negative slopes, resulting in the appearance in Figure 3, but with an additional feature, it can change the angle, α and ß see illustration below.

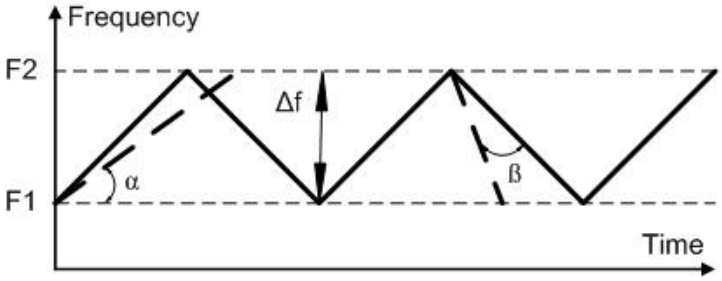


Figure 14: The features of DDS

Through this method a stationary target and optimal angle can be determined under simulation. How to tune in the DDS depends on how the manufacture has chosen to place the inner structure.

The transmitted signal is produced at the DDS and the phase is controlled by the PLL. The next chapter will describe how the received signal is handled.

3.1.4 The Mixer

The transmitted signal is produced by a triangle wave generator through the VCO as described in chapter 3.1.1. The received and transmitted signals are connected to a mixer as shown in Figure 15.

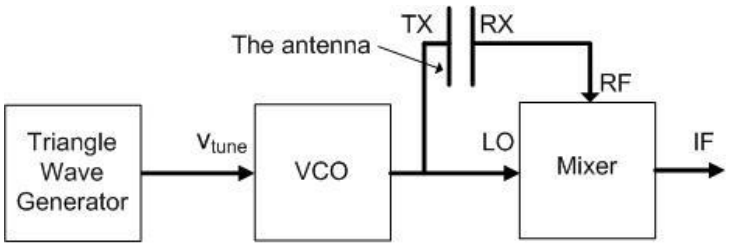


Figure 15: The mixer

The mixer obtains both transmitted, LO and received, RF signals as described in Figure 15. The functionality of the mixer can be described as a multiplier receiving the frequency w_s at LO and w_p at RF. By assuming that both LO and RF has the same amplitude the following simplification can be done.

$$\left.\begin{matrix} A(t)\cos(w_s t) \\ A(t)\cos(w_p t) \end{matrix}\right\}\otimes \Rightarrow \left\{ A(t)\cos(w_s t)\cos(w_p t) = \frac{A(t)}{2}(\cos[(w_s - w_p)t] + \cos[(w_s + w_p)t]) \right\} \Rightarrow$$

The signal is then passed through a filter to eliminate the high coefficient. This gives the following appearance of the IF signal: $\frac{A(t)}{2}\cos[(w_s - w_p)t]$ where its frequency is $w_{IF} = (w_s - w_p)$

The consisting parts of the current system are now introduced. This is essential to understand the total system presented in the next chapter.

3.2 Current system architecture

A high level schematic of the first prototype is shown in Figure 16. The VCO, mixer and antenna is placed on a RF-module and bounded to a PCB. The RF-module is designed on a thin film substrate to which different RF chips are bounded, a so called multi chip module, MCM. The PLL [4] is placed on the PCB while an external DDS [5] and ADC is used.

The DDS [5] and PLL [4] are initiated from a computer. The DDS continuously sends a square pulse at different frequencies to the PLL. A fixed phase will be assured with the loop consisting of the PLL and the VCO.

The mixer will detect the difference in frequency between received and transmitted signal, see section 3.1.4 for a mixer description. This signal is called the IF frequency and equals the beat frequency in Figure 2. The IF signal decreases with a longer distance as described in equation 11 which is counteracted by using an IF-filter. The IF-filter is constructed to amplify the signal in an opposite manner. This achieves constant amplitude of the IF signal independently of the range to target.

A side effect of the IF filter is the noise which also will be amplified. This will not be a problem provided that the SNR is good enough.

The IF signal is converted through an ADC and sent to a PC. In the PC, transformations and evaluation be performed which will specify the distance to target.

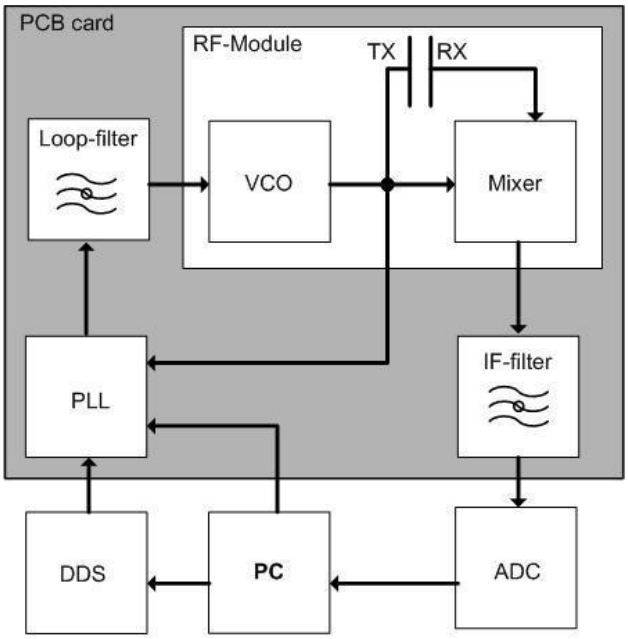


Figure 16: Basic flow schedule

3.3 Task

The work at Acreo AB involves further development of the radar prototype. This requires a good understanding of the platform used today as described in chapter 3.2.

From standards and discussions a test specification will be delivered which shows how to manage the requirements. The second task is to choose architecture and a communication interface. The new configuration is described in detailed to allow scheduling of the system. A microcontroller needs to be programmed for initiating and controlling the system. The control should be able to handle three types of situations. The ability for detecting: stationary objects, moving objects and multi-object situations as described in section 2.2.

The process and presentation of radar data will be performed on a PC. This requires a fast interface and application development.

This will lead to an enhanced prototype with the purpose of demonstrating advantages of SiGe technology in radar solutions.

Next part will describe the development of the next radar prototype.

3.4 Next version

The work of the thesis will involve adding DDS, µC, ADC and interface onto the PCB.

The µC will handle the initiations and manage the control of DDS and PLL. The ADC will convert the analogue data sent from the mixer to a digital representation. Through the interface the data is sent to a PC for further manipulations and finally, visualisation of the distance to the target.

To ensure a correct function the ADC and µC need to be selected carefully. Another issue is what type of interface to choose which will satisfy the requirements.

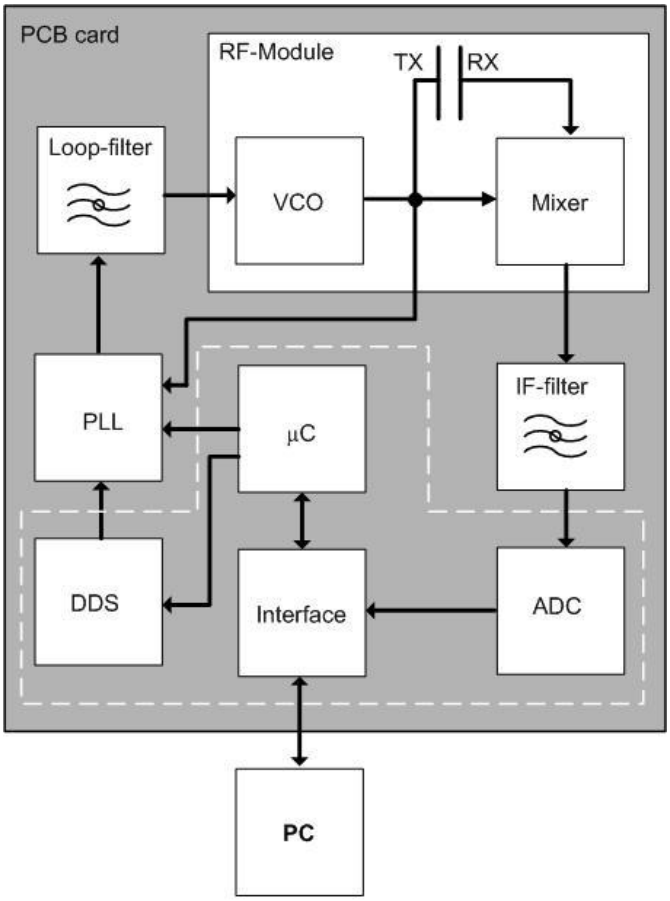


Figure 17: Next version

3.4.1 Architecture selection

The PLL and DDS can be chosen separately and from different manufacturers.

The limiting requirements in the μC are the memory size, interfaces for data transfer and number of ports. To estimate these restraining factors the surrounding components need to be analyzed in detail through their datasheets.

Both the PLL and DDS need to be initiated when the prototype is starting up. By reading through how each individual component is defined you can set up a protocol for the communication. The components may also have predefined serial and/or parallel communication protocol. This is preferable because you may avoid mistakes and get a straightforward code. The choice of communication line will affect the number of ports which the μC needs.

The DDS will be continuously re-programmed for establishing different spectrum. This may affect your choice of μC in terms of the internal memory depending on how you wish to implement the solution. An important observation when minimizing memory is the effect of a less flexible system.

By considering the system to be a prototype model it's not wise to minimize the μC. In this early stage it's better to oversize the requirements when designing the prototype. The constraining factors of the system can lead to a solution with less dynamic capacity.

An essential part of the PCB will be the interface which sends the data from the ADC to the PC. Through studies of different fast digital interfaces the prototype will have a more compact design.

The central issue for selecting an interface is the amount of data which is going to be transferred at a high rate.

3.4.2 Application selection

The communication with the prototype could be handled by different protocols and application languages. The possibilities can be limited by focusing on how to process the data.

The graphical environment and effectiveness of signal processing which Labview offers provides a fundamental ground to process and present the data. Another desirable feature available in Labview is the possibility to view the results in real-time.

Chapter 4

The implementation

At the first glance at the schematic below it appears to differ from the desired solution which was described in the previous chapter. A deeper look into the synthesizer and USB will motivate this configuration. In Appendix A a more detailed description of the environment setup is found.

Instead of a DDS and a PLL a synthesizer was chosen which includes both of these functionalities. The choice will naturally affect the flexibility of the system when losing the possibility to independently be able to switch between different components. The positive point of a synthesizer solution is a closed system which leads to less peripheral disturbances. In Appendix B a report of the evaluation between the synthesizer and the primary components are found.

The USB device could be divided into two separate blocks, a µC and dedicated hardware for data transmissions. The primary object of the µC is to initiate and control the synthesizer. Initiation data is transmitted from the Labview environment while the control configuration is placed directly on the µC. The dedicated hardware is configured from the µC to receive data from the ADC. The data is continuously streamed through the USB cable according to the USB 2.0 standardisation.

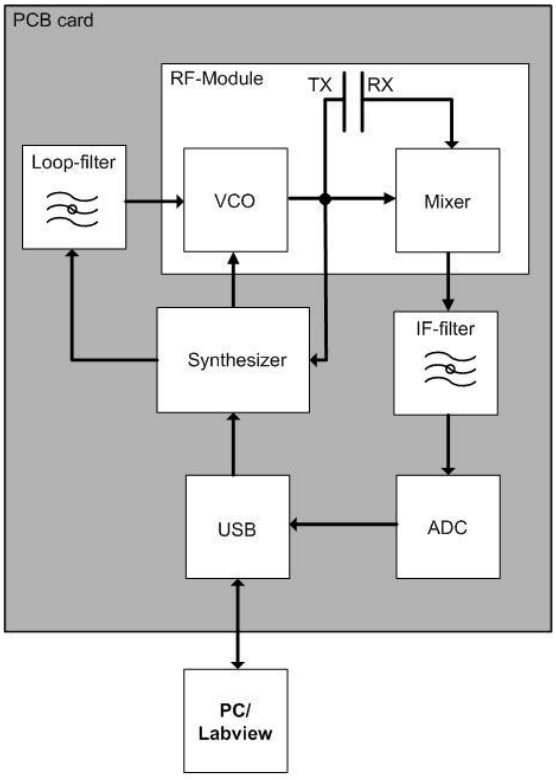


Figure 18: Achieved configuration

4.1 PCB interconnections

4.1.1 Streaming ADC data

The choice of USB chip was Cypress EZ-USB FX2LP which has a sufficient number of ports and suitable communication protocols.

To maximize the streaming process, a high-speed mode together with the bulk communication protocol is used. Through a slave FIFO interface mode it's possible to read in 16 bits of parallel data (FD [15:0]) and packaging it into 512 bytes. When a packet is filled it's forwarded to the PC through an auto committing feature. To assure that no data will get lost, a quad buffer is used to automatically continue to collect received data.

There are four separate channels for distinguished data sent between USB and host. These are called endpoints, EP or pipes and have the following predefined setup.

Selected EP	FIFOADR1 pin	FIFOADR0 pin	Direction
EP2	0	0	From host to USB device.
EP4	0	1	From host to USB device.
EP6	1	0	From USB device to host.
EP8	1	1	From USB device to host.

Table 2: Direction setup of endpoints

The setup described in Figure 19 is explained by following statements. Endpoint 6 is activated by hardwire FIFOADR0 and FIFOADR1. The EP 6 is used for uploading the streaming data (EP 8 could be used as well).

Connecting slave output enable (SLOE), slave read (SLRD), and slave write (SLWR) enables the system to continuously stream data from the ADC. The synchronisation is solved by applying the ADC oscillator. Using an ADC of 14 bits data allows two synchronisation bits (SYNC) to be included in every packet. These will be used to identify the mode of operation on the synthesizer. The functionality of the SYNC bits is described in chapter 4.1.3.

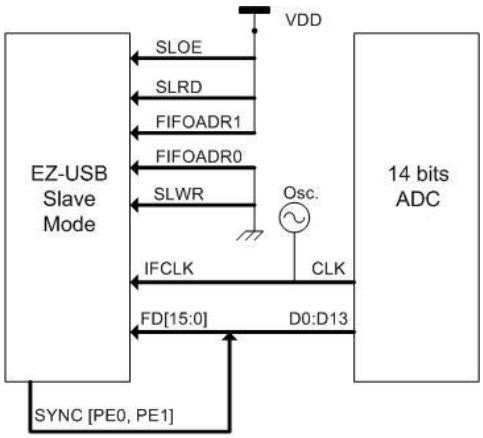


Figure 19: USB to ADC configuration

A firmware is the composing software inside a USB chip. By modifying the firmware you're able to control your device as desired.

The implementation of the firmware was developed on the frameworks source code for the EZ-USB FX2 chip supplied in Cypress developer kit. In Appendix C the firmware is configured for high speed and variables are defined.

The framework structure has two major predefined functions, an initiation part TD_Init () called once at start-up and a part TD_Poll () which is called repeatedly while the device is idle. For the streaming process only TD_Init () was used. The implementation code is located in Appendix D.

4.1.2 Initialization and control of the synthesizer

Besides using the endpoints for transmitting data between PC and USB this could be done using vendor commands. The vendor commands are not built for transmitting large packets but could be used for communicating with the μC.

In this application the vendor commands are used for sending data needed to initiate the synthesizer. The data received in the μC are immediately forwarded to the synthesizer leaving the implementation of the initiation phase at application level on the PC.

The firmware implementation is described in Appendix E where the sub function DR_VendorCmd () is handled by the frameworks and called every time a vendor command is received.

The control of the synthesizer will be implemented on the μC. The choice of placing it at application level would lead to unacceptable delays at runtime caused by the operating system.

There are two different functionalities which the control phase should be able to handle: stationary object and multi-object situation as described in section 2.2. The moving object is a special case of the multi-object situation. The implementation code over the control phase is located in Appendix F.

The first mode is illustrated in the figure below.

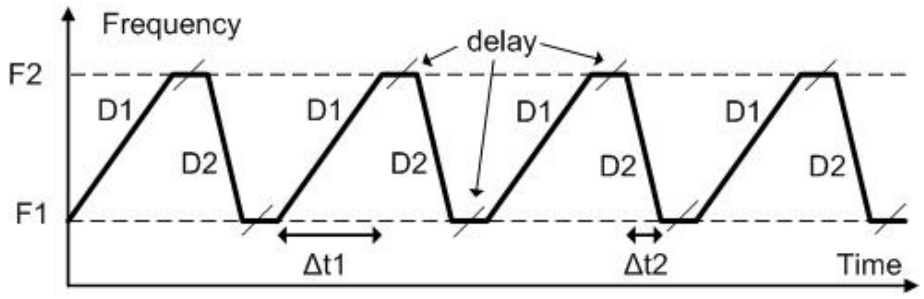


Figure 20: First mode of synthesizer

The frequency limits F1 and F2 are stored at different registers called profile control register 0 and 1. There are up to eight different profiles to be stored and these are reached through the three port select pins (PS0, PS1, and PS2). For our application there is only the need of switching between two profiles which simplifies the communication lines.

To implement this mode the register which holds the rising, RSRR and the falling, FSRR sweep ramp rate is initiated. After the initiation phase the only thing left is to toggle the PS0 pin at Δt1 and Δt2 for performing the proper appearance.

The second mode has a more complex appearance as showed in Figure 21.

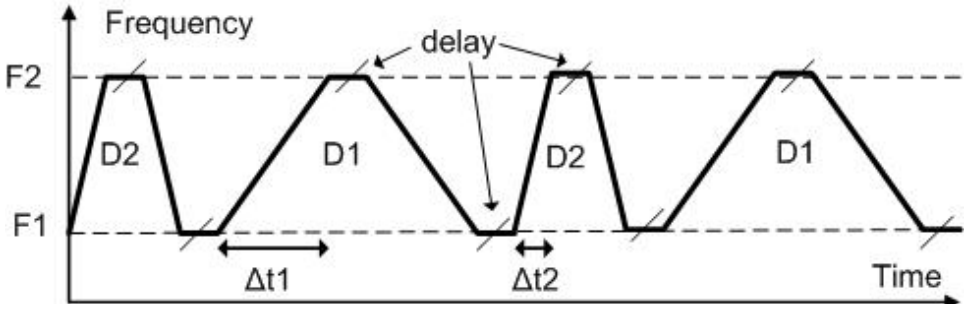


Figure 21: Second mode of synthesizer

To implement this mode efficiently the following flow graph was produced.

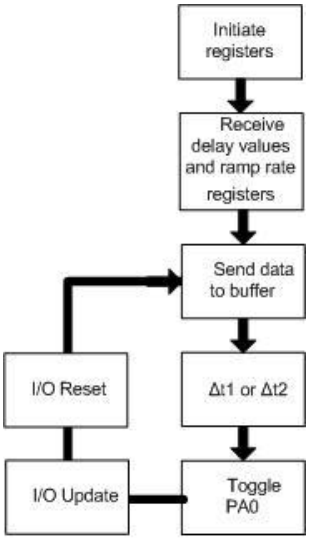


Figure 22: Flow graph over control phase

The counter values need to iterate according to Δt1, Δt1, Δt2, Δt2, Δt1… for producing an appearance as in Figure 21. The control sequence acquires delay values (Δt1 and Δt2) and ramp rate registers to be received from the application environment.

Between every rising (falling) ramp the buffer receives the data needed for the next increasing (decreasing) step. As soon as the delay has finished, the PA0 is toggled. This is followed by an "I/O update" which transfers the data from the buffer to the internal register. To assure a synchronized transfer an "I/O reset" clears the buffer before a new iteration.

The communication between synthesizer and USB is implemented for a 2 wire serial data input/ output (SDIO) and serial clock, (SCK). Through a separate serial data output (SDO) a possibility for using a 3 wire communication is prepared. This could be useful in future verification or debugging of the synthesizers register. The communication link is illustrated in Figure 23.

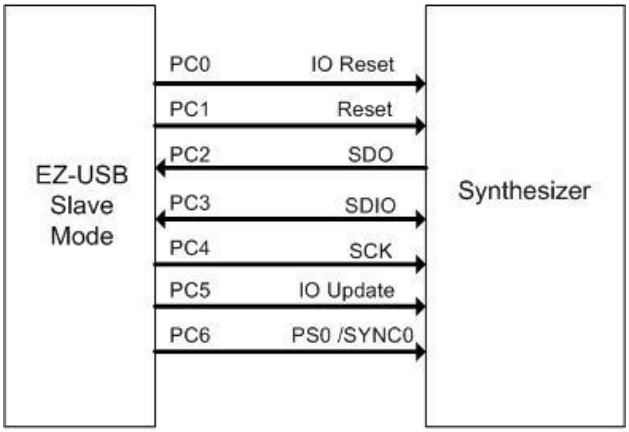


Figure 23: USB to synthesizer configuration

4.1.3 Mode of synthesizer

The PS0 is equivalent with the appearance used for one of the synchronization bits (SYNC0). The other SYNC1 bit identifies if the actual sweep are a down or up flank. These bits will be used for matching the transformation with correct mode at application level. The appearance of the synchronisation bits is showed in Figure 24.

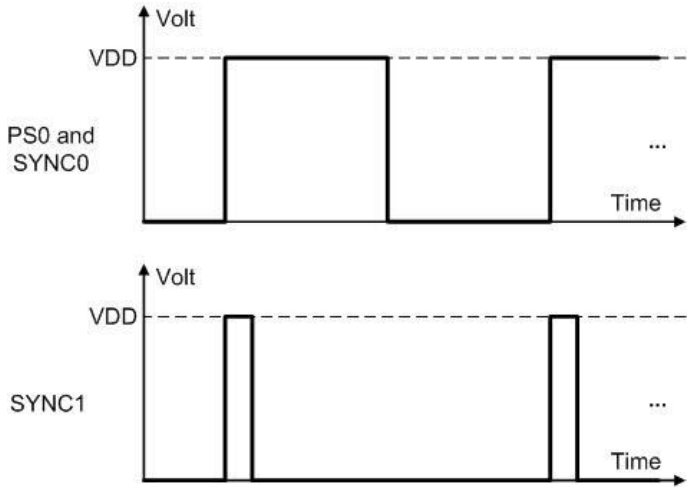


Figure 24: Synchronization bits

4.2 PC application

4.2.1 Labview configurations

As described in chapter 3.4.2 Application selection the calculation of real-time data is efficiently done in Labview. Implementing the communication to the USB device in a block diagram is impractical and can be hard to solve. Another possibility is to use a DLL which passes parameters to and from Labview. A DLL can be accessed through the "Call Library Function" in Labview. The DLL is automatically loaded by the operating system at execution of the application.

The "Call Library Function" block essentially requires information of search path to the DLL and data types of returning and incoming parameters. An example on the DLL syntax:

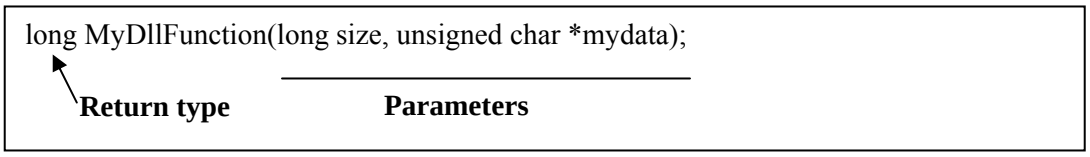


Figure 25: DLL syntax

Each function corresponds to a block diagram and each input parameter corresponds to an input terminal on the related block. An input parameter could be altered inside the DLL and is then available on the output terminal.

The syntax is set by a right-click on the "Call Library Function" block and choosing "configure" in the menu. The corresponding block diagram for the syntax is shown in Figure 26.

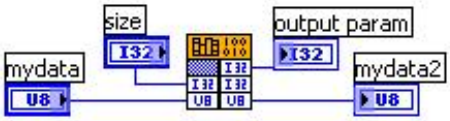


Figure 26: Basic block diagram

Where "output param" is equivalent to the returned parameter passed from the function. The "mydata2" corresponds to the returned value of variable "mydata".

4.2.2 The Labview setup

The appearance of the Labview console is added in Appendix G and the corresponding block diagram can be seen at Appendix H.

The Labview is handled through a case structure were four functionalities is implemented.

- Load firmware
- Load file
- Initiate
- Toggle Control mode ON/OFF

The firmware load is simply done by specifying the link to a bix-file containing the configurations described in chapter 4.1.1. The implementation of the download procedure is handled through the DLL when “Load firmware” button is pressed.

The initialization data for the synthesizer is read from an lvm-file and placed in a 2-dimensional array at “Load file”. The button “Initiate” forwards the array to the DLL. Each row is configured according to the following setup:

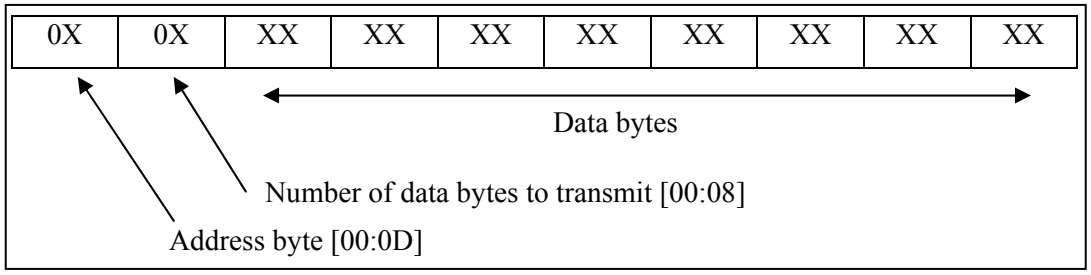


Figure 27: Row configuration of the synthesizer

Where the number of data bytes is fixed and depends on the definition of the related register. The amount of registers corresponds to the number of rows in the array.

The control case is forwarding the mode of the control, delay values ($\Delta t1$ and $\Delta t2$) and ramp rate registers to the μC. The control mode is either stationary object or multi-object situation.

The data stream is implemented as a parallel function and is enabled when the “Stream data” switch is activated; see Appendix G for the Labview console. Data is received at the same speed as the oscillator of the ADC (IFCLK). The space allocation of the streaming array is done in Labview and passed through to the DLL. The receiving data buffer consists of a one-dimensional row matrix at 32 k with one byte at each row. The array is modified to a 16 k with all 14 data bits represented on each row. The received ADC data is written to an output file and also further manipulated. Through an FFT spectrum the array is transformed and presented in a graph. This transformation needs to be synchronized with the synthesizer’s mode of operation (SYNC bits). These are available in the bit 7 and 8 which are masked out from the array.

4.2.3 The DLL setup

The dll project is composed of following four files:

Files	Explanation
Ezusbsys.h	A 'base code' for communicating with the USB GPD (General Purpose Driver) applied by Cypress and is available in there's developer kit.
Labview_to_Dll.cpp	Passes parameters between Labview and dll.
Dll_to_Cypress.cpp	The dll communicates with the ezusbsys.
Dll_to_Cypress.h	Defines variables and function declarations.

Table 3: Composing files of the DLL

Where the functions are documented in Appendix I and the implementation code is available in Appendix J. The interaction between the files can be described by following block diagram:

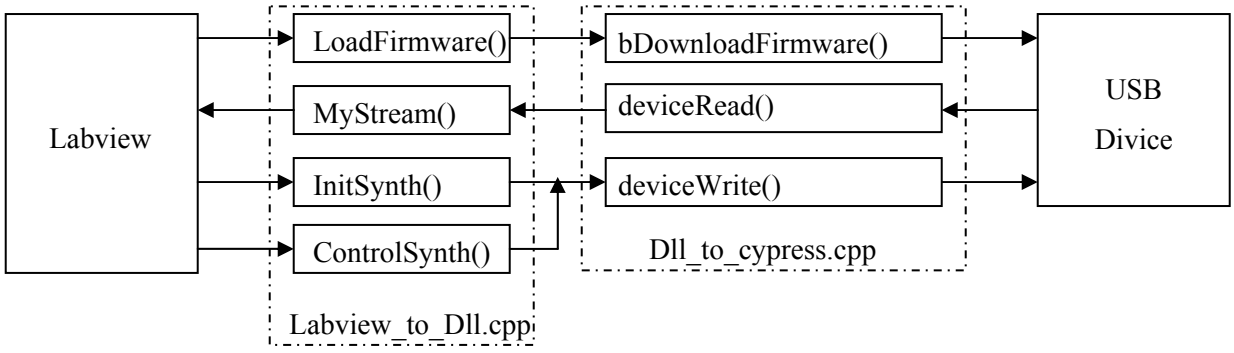


Figure 28: Dll block diagram

The following syntax was created for passing variables between Labview and the DLL.

1) long LoadFirmware(LStrHandle name)
2) long MyStream(unsigned char array[], unsigned long size, unsigned char stream)
3) long InitSynth(long nrofrows, long nrofcols, unsigned char myData[], long *ret_data)
4) long ControlSynth(unsigned char control_mode, unsigned char delta_t1,
   unsigned char delta_t2, unsigned char tinyArray[], long nrofrows, long nrofcols)

Table 4: Labview to DLL configuration

The first function loads and modifies the firmware file. The streaming function fills an array with the received data and returns to Labview. The final function sends initialization data to the synthesizer according to the following flow graph.

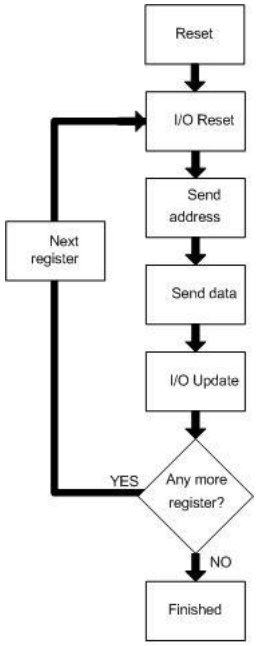


Figure 29: Flow graph over initiation phase

The initialization procedure starts with a reset which places the synthesizers register to default. The synchronization is assured with an "I/O Reset" before every new transmission. The buffer on the synthesizer is filled with address and data for related register. The "Send data" block consists of sending several data bytes depending on which register is addressed. This is followed by an "I/O update" which transfers data into the register in same way as the control phase.

Following syntax was made for the dll to USB communication:

```
1)    int bDownloadFirmware( const char* fwFileName= NULL)
2)    int deviceRead( uchar b[], unsigned long& bytesTransfered)
3)    int deviceWrite( uchar vx_cmd, uchar data)
```

Table 5: Labview to DLL configuration

The first function transfers the firmware according to the protocol available in Appendix K. The device read is set to read the bulk buffer at endpoint 6 and with 512 bytes per packet just as the USB was configured. The last function is passing incoming data from Labview as vendor commands to the USB device.

4.3 Test benches and verifications

4.3.1 Limiting components of streaming

The first verification was made based on a test bench described by Figure 19, but instead of using an ADC the signal was generated from a logic analyzer.

The data transfer needs to be analyzed for making sure no data loss will occur. The logic analyzer is configured to continuously send out a stream of data which is larger than the receiving quad buffer at the USB. Minimum amount of data needs to be larger than 16k (4*512 Bytes) of different bits. With 16 parallel data strobes it is possible to transmit about 65k (2^16) bits which sufficient for our application. The logic analyzer is configured to count from 0000_h up to $FFFF_h$. This is enough for filling the quad buffers with data. The dividing line between the buffer registers could later be inspected for possible data loss.

At this high rate of data transfer the bottleneck for reaching the theoretical limit is depending on hardware at the host.

The host hardware used in this test bench is a Motherboard based Host Controller, ICH5 which is limited to 11 packets per microframe. The microframe is by definition the scheduled and performed transactions during a 125 μs interval. The maximum throughput could thereby be calculated according to:

$$Maximal\ data\ rate = 11\ \frac{packets}{\mu Frame} * 8\ \frac{\mu Frame}{ms} * 512\ \frac{Bytes}{packets} * 8\ \frac{bits}{Byte} = 360\ Mbps \Rightarrow$$

$$Maximal\ \ frequency\ rate = \frac{360 Mbps}{16\ bits} = 22.5\ MHz$$

The threshold for maximum transmission frequency is 22.5 MHz with a controller as ICH5.

By placing the logic analyzer to operate at 20 MHz results in a throughput of 320 Mbps. The transfer is synchronized by applying a clock signal supported from the logic analyzer.

The final step is to analyze the incoming data received by Labview. Measurements was written to file and examined in the interval of 0000_h to $FFFF_h$ which includes two full quad buffer readings.

Without obtaining any bit errors or data losses of the examined data, the established system has obtained a fast digital transfer link according to the USB 2.0 protocol.

4.3.2 Real-time presentation

The logic analyzer is connected as earlier setup, using 16 parallel data strobes together with the clock signal for simulating an ADC. The logic analyzer was configured to continuously send out a fixed frequency at 3/8 of the 20 MHz which is the sampled speed. The frequency should result in 7.5 MHz, which is also the case according to the Figure 30.

The received data was transmitted into Labview followed by a real-time calculation. Through a FFT, the signal was successfully presented in frequency domain.

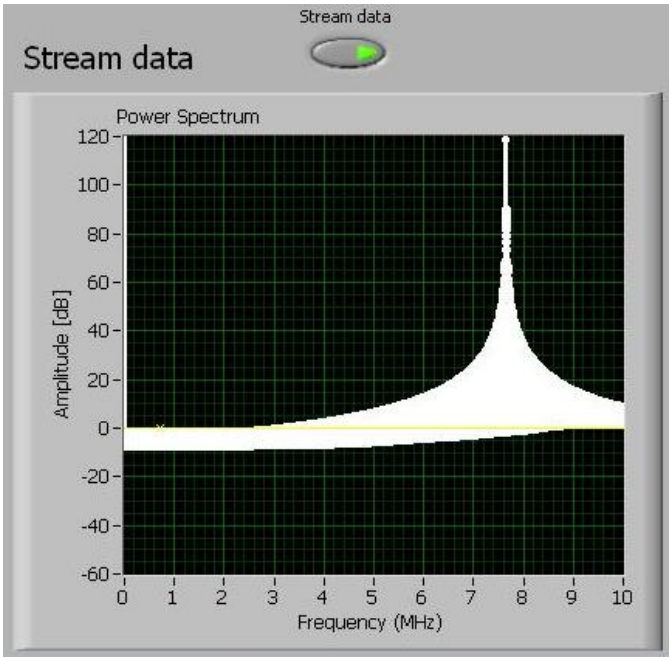


Figure 30: Streaming data

4.3.3 Initiation of the synthesizer

From the Labview console a predefined file (*.lvm) is loaded consisting of initiation data for the synthesizer. This file is configured as the array setup as seen in Figure 27. The application interface loads an array into the DLL which handles the initiation procedure. The DLL communicates through the μC according to the flow graph in Figure 29.

With a spectrum analyzer the functionality of the synthesizer was verified.

4.3.4 Control of the synthesizer

The synthesizer was first initiated according to the previous setup. The value of Δt1 and Δt2 together with the sweep rate register was loaded from Labview to the μC.

An oscillator was connected to the V_{tune} input of the VCO where functionality of the synthesizer was verified.

Chapter 5

Results

In the figure below the final implementation solution is described. In Appendix L the pin function description is given which explains the interconnections in detail.

The microwave module is explained with an example of how it could be possible to reach the 76.5 GHz. The three antennas have been added for the possibility to analyze a wide area.

The context of microwave module, Loop- and IF- filter is confidential information.

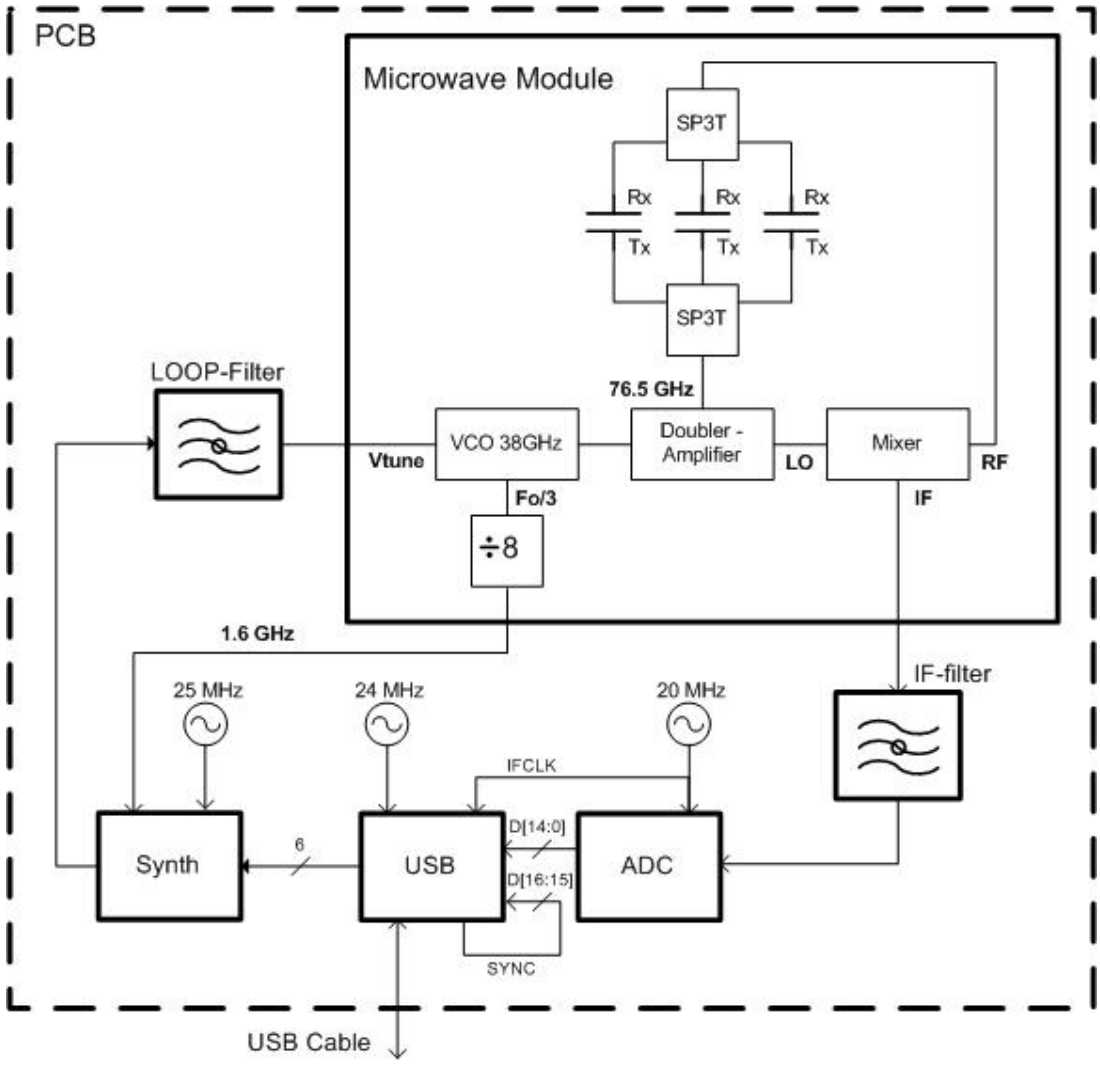


Figure 31: Final implementation

Through the Labview console presented in Appendix G the following functions are implemented:

- Loading firmware file to USB device.
- Load and transmission of initiation data to the synthesizer
- Switch between control modes of synthesizer, stationary object and multi-object situation. The moving object is a special case of the multi-object situation.
- Read streaming data to file.
- Presentation of streaming data.

Chapter 6
Recommendations

6.1 Unfinished implementations

The synchronisation bits were not implemented in either the firmware or the dll. This functionality is described in chapter 4.1.3 and is remains to be implemented.

6.2 Different communication strategy

In a first step in communicating with the synthesizer from Labview vendor commands is used which can transfer maximum of 2 bytes. This is an inefficient way to transmit the initiation data of totally 43 bytes. By implementing a transmission of bigger packets to the USB device all data could be downloaded faster. From the downloaded endpoint the data could then be reached by the μC and sent to the synthesizer. There are two aspects which recall this change of implementation. The initiation occurs ones at each sequence and is relative small compared with the initiation time of the surrounding components. The current implementation can be seen as a minimization of the firmware which doesn't require the full retail of Keil Software.

6.3 Possible port implementation on the USB

There is a possibility to implement a SPI (Serial Peripheral Interface) on the USB device which maximize the speed of initialization and control between the μC and synthesizer.

By connecting the slave output enable, SLOE to a port the possibility to stop streaming data is gained. This could become useful if the bandwidth is limited.

The empty I/O Port E could be used for enabling predefined modes of operations. With DIP switches 2^8 states could be reached by the firmware.

6.4 Host Controller issue

A regular PCI Host Controller can only receive a maximum data rate of 160 Mbps or 5 packets per microframe which is half the amount compared with the measured data of 320 Mbps in chapter 4.3.1. This could easily be solved by decreasing the oscillator frequency on the ADC by half to 10 MHz.

Appendix A -
Environment setup

The PCB will consist of these main components:

Component	Product specification	Manufacture
MCM	Microwave Module	Acreo AB
Synthesizer	AD9956	Analog Devices
ADC	AD9245 - 14-Bit 20 MSPS	Analog Devices
USB	CY7C68013A – Microcontroller	Cypress

Table 6: Main components

The oscillator and voltage levels which is needed:

Component	Typ. oscillator (MHz)	Voltage (V)
Synthesizer	25	3.3 and 1.8
ADC	20	3.3
USB	24 and 20 (from ADC)	3.3

Table 7: Oscillation and voltage levels

Following software was used:

Program	Developer	Explanation
µVision2 ver. 2.38j	Keil Software	Firmware for USB device.
Microsoft Visual C++ .NET	Microsoft	Created a Dll.
Labview 8.0	National Instruments	Data processing and presentation.

Table 8: Environment setup

Following hardware was used:

Hardware	Developer	Explanation
CY3684	Cypress	An evaluation board of the USB device.
AD9956	Analog Devices	An evaluation board of the synthesizer device.
Spectrum analyzer	Hewlett Packard	For measurement in frequency domain.
Pattern generator	Agilent	Producing different steaming data.

Table 9: Environment setup

Appendix B -
Evaluation of synthesizer

Test Report AD9956 2.7GHz synthesizer

Abstract

This is a short measurement report for the AD9956 2.7GHz synthesizer. This investigation was made to see if the PE3236 PLL [4] and the AD9852 DDS [5] could be replaced with the AD9956 synthesizer module [6]. From this investigation it can be concluded that:

The phase noise performance of the AD9956 is equal or better than our current solution, therefore the AD9956 can replace the current AD9852 and the PE3236 circuits.

Introduction

According to datasheets from Analog Devices the AD9956 synthesizer looks like a promising replacement circuit for the PE3236 PLL [4] and the AD9852 DDS [5] currently used in the A77x3M radar module. The AD9956 synthesizer has DDS, PLL and charge-pump functionality as well as prescalers that can be programmed to fit in our radar system. For a more detailed description of AD9956 see the datasheet [6].

In the Figure 32, the measurement schematic is shown.

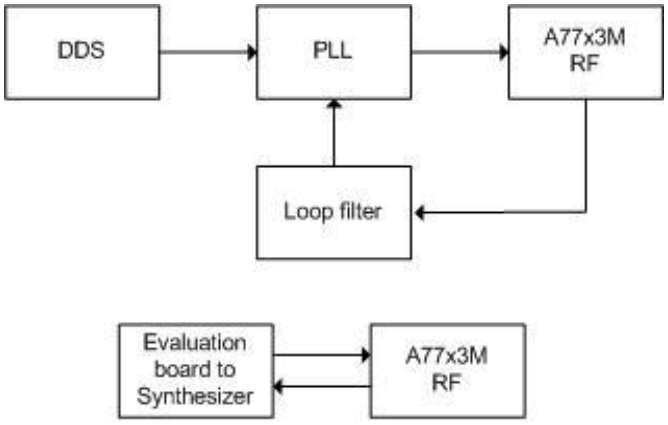


Figure 32: Measurement setup

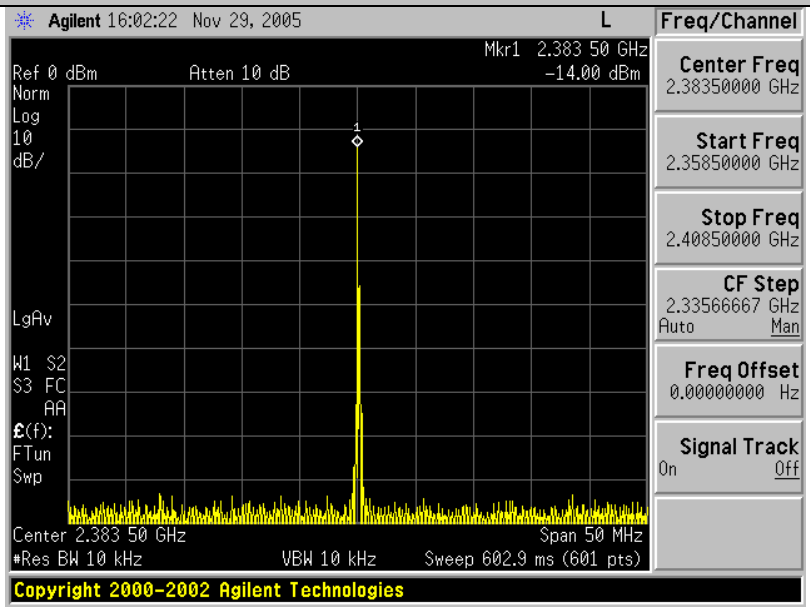
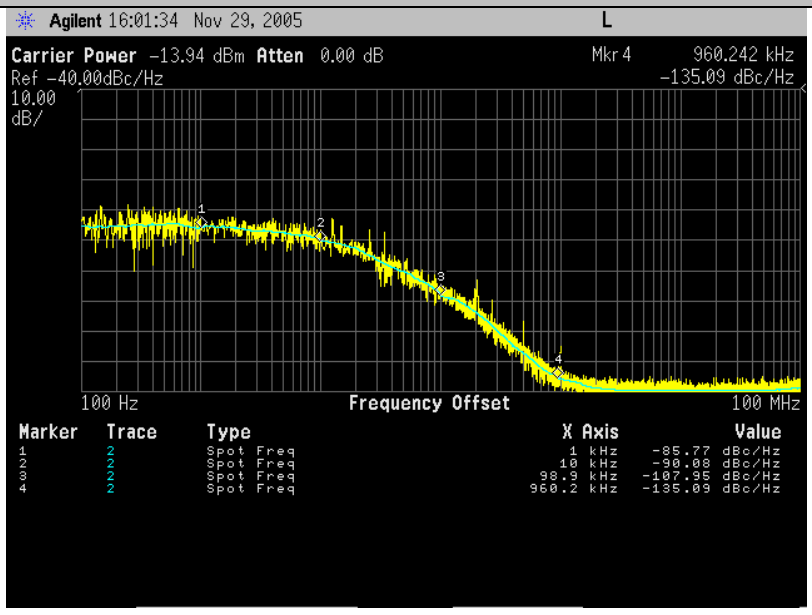
Redesigned Loop Filter, 2.4GHz VCO equipped AD9956 evaluation board				
Meas Nr:	DDS (DAC) Frequency			VCO Output Frequency
# 2	24.5M			~2.4GHz
Frequency Spectrum				
				
Phase Noise				
				

Table 10: Pure AD9956

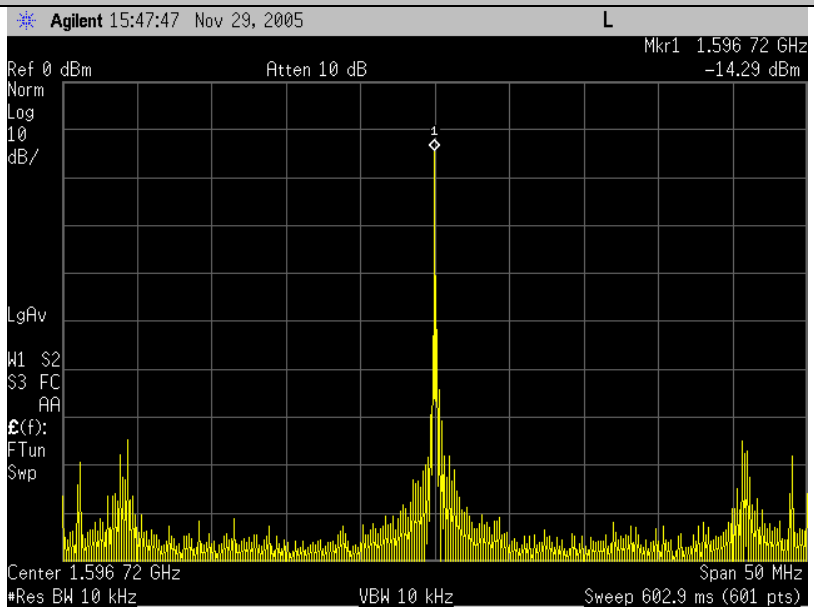
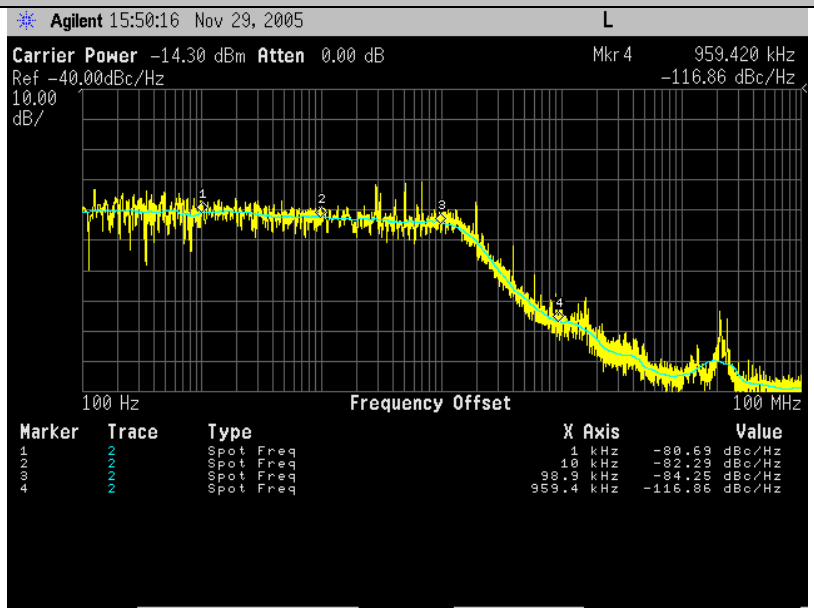
Redesigned Loop filter, 77/48GHz VCO (A77x3M) + AD9956 evaluation board PLL				
Meas Nr:	DDS (DAC) Frequency			VCO Output Frequency
# 3	25.05MHz			
Frequency Spectrum				
				
Phase Noise				
				

Table 11: 77GHz VCO, AD9956 DDS

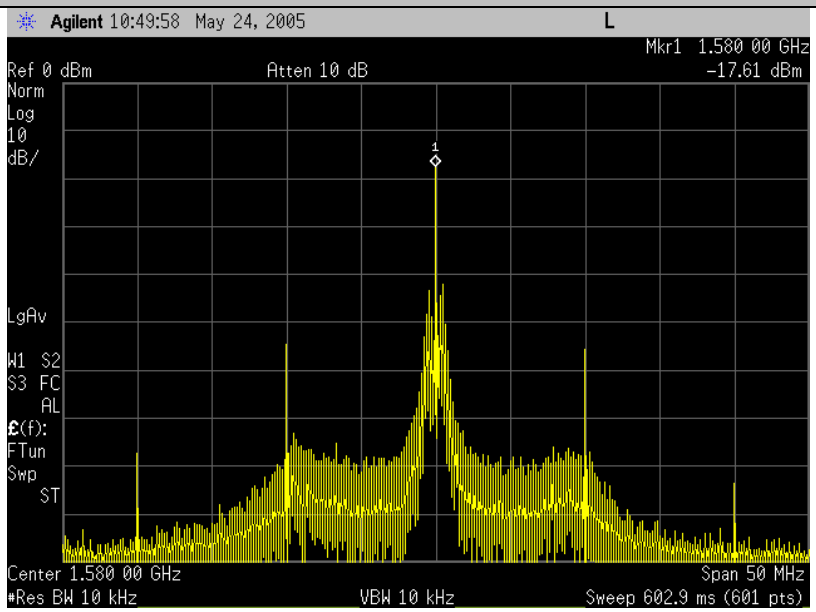
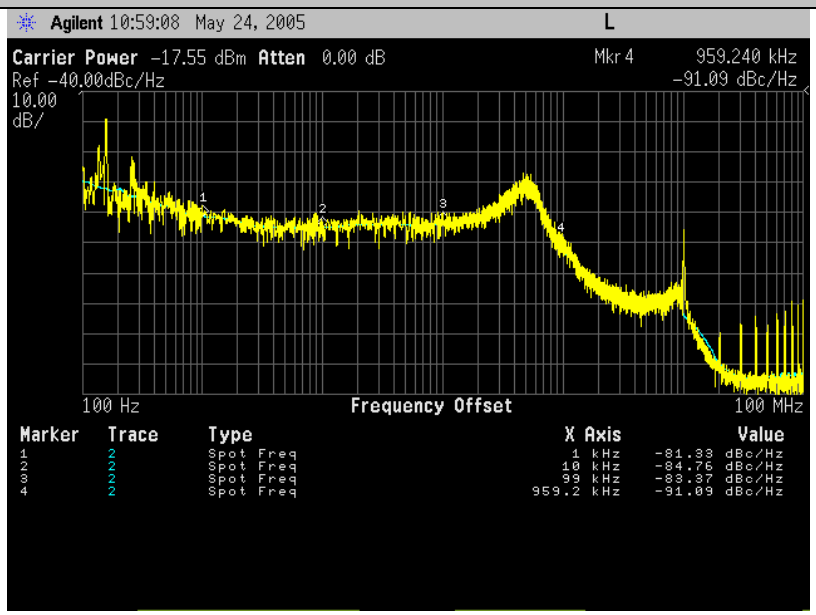
A77x3M with DDS reference added for reference				
	PLL control Registers			**Carrier Frequency, *Fin***
Meas Nr:	R [0:15]	M [0:127]	A [0:15]	
# 4	0	14	8	1580 MHz
Frequency Spectrum				
			
Phase Noise				
			

Table 12: Measurement of the A77x3M, AD9852 DDS reference Peregrine PLL

Conclusions

The phase noise of the AD9956 evaluation board can be seen in Table 10.

To see if the AD9956 can replace the current DDS/PPL configuration, the AD9956 was used together with the 77GHz VCO used in the A77x3M radar module. The loop filter is not optimized for this VCO but it will hopefully give us a hint on possible performance.

Table 11: 77GHz VCO, AD9956 DDS shows measurement results for the AD9956 used together with the 77GHz VCO. Comparing this with our current solution, shown in Table 12, it's seen that the new solution shows a much less distorted PSD spectrum. The phase noise performance also looks better. The noisy behavior of the old filter can be explained by the fact that the old filter is a 2^{nd} order active filter and the new filter is a 2^{nd} order passive filter. The active filter might introduce noise, e.g., via the supply voltage degrading the over all noise performance of the system.

Conclusions by this investigation:

- The phase noise performance of the AD9956 is equal or better than our current solution. Therefore the AD9956 can replace the current AD9852 and the PE3236 circuits.
- An active loop filter might introduce more noise than a passive filter unless special care is taken to the supply strategy.
- Great care must be taken to the reference frequency to the PLL input. A crystal oscillator is suggested as reference in the new demonstrator.

Appendix C -

Configure firmware

```
// usrp:
#define bRequest        SETUPDAT[1] // vendor request
#define wValueL         SETUPDAT[2] // vendor value

// Vendor requests
// Vendors for "stationary" and "multi-object" control mode
#define VR_INIT_DELTA_T1         0xB0               // Initiate delta t1
#define VR_INIT_DELTA_T2         0xB1               // Initiate delta t2
#define VR_TOGGLE_CONTROL_MODE   0xB2               // Start control mode
// Vendors for " multi-object" control mode
#define VR_REG_ADDR_1                   0xB3   // Receive address1
#define VR_REG_DATA1_1                  0xB4   // Receive data1
#define VR_REG_DATA2_1                  0xB5   // Receive data2
#define VR_REG_ADDR_2                   0xB6   // Receive address2
#define VR_REG_DATA1_2                  0xB7   // Receive data13
#define VR_REG_DATA2_2                  0xB8   // Receive data14
// Vendors for initiate the synthesizer
#define VR_IO_RESET        0xC0   // PC0 I/O reset, clears buffer
#define VR_MASTER_RESET    0xC1   // PC1 Master reset on synthesizer
// PC2 Serial data output (3 wires). Not implemented
#define VR_SDIO            0xC3   // PC3 Serial data in-/ out- put (2 or 3 wire)
// PC4 Serial clock (2 or 3 wire) SCLK. Handled by USB chip
#define VR_IO_UPDATE       0xC5   // PC5 I/O update, moves buffer to register

// Declarations
sbit IO_RESET       = IOC ^ 0;    // PC0 I/O reset, clears buffer
sbit M_RESET        = IOC ^ 1;    // PC1 Master reset on synthesizer
// PC2 Serial data output (3 wires). Not implemented
sbit SDIO           = IOC ^ 3;    // PC3 Serial data in-/ out- put (2 or 3 wire)
sbit SCK            = IOC ^ 4;    // PC4 Serial clock (2 or 3 wire)
sbit IO_UPDATE      = IOC ^ 5;    // PC5 I/O update, moves buffer to register
sbit PS_OR_SYNC0    = IOC ^ 6;    // PC6 Profile select on synthesizer and Sync
bit0

// Variables
BOOL run_control_mode = FALSE;          // flag to start operate in control mode
BOOL run_control_mode_stat = FALSE;     // flag to which control mode
BYTE delta_t[2];                        // Delay value
BYTE update_reg[2][3];                  // Address and data for two registers

//-------------------------------------------------------------------------
// My private functions
//-------------------------------------------------------------------------
// Reset synthesizer
void master_reset(void)    // Resets the synthesizer
{
      M_RESET=1;           // Put signal high
      _nop_( );
      M_RESET=0;           // Put signal low
      _nop_( );
}
```

```
// IO reset synthesizer, clears buffer
void IO_reset(void)
{
	IO_RESET=1;		// Put signal high
	_nop_( );
	IO_RESET=0;		// Put signal low
	_nop_( );
}

// IO update synthesizer, transport buffer to register
void IO_update(void)
{
	IO_UPDATE=1;		// Put signal high
	_nop_( );
	IO_UPDATE=0;		// Put signal low
	_nop_( );
}

// Transmits one byte to synthesizer
void CLK_byte(BYTE TransmittedByte)
{
	int c;
	for (c=7; c>-1; c--)
	{
		SDIO=(TransmittedByte >> c) & 0x01;
		_nop_( );
		SCK=1;		//Put clock high
		_nop_( );
		SCK=0;		//Put clock low
	}
	SDIO=0;			// Put Data low

}
```

Appendix D -

Initiation of USB

```
void TD_Init(void)             // Called once at startup
{
       // set the CPU clock to 48MHz
       CPUCS = 0x12;
       SYNCDELAY;

       // use IFCLK pin driven by external logic (5MHz to 48MHz)
       // use slave FIFO interface pins driven sync by external master
       IFCONFIG = 0x03;
       SYNCDELAY;
       REVCTL = 0x03; // use enhanced packet handling
       SYNCDELAY;

       EP2CFG = 0x00;     // EP2 not valid
       SYNCDELAY;
       EP4CFG = 0x00;     // EP4 not valid
       SYNCDELAY;
       EP6CFG = 0xE0;     // EP6IN, bulk, size 512, 4x buffered
       SYNCDELAY;
       EP8CFG = 0x00;     // EP8 not valid
       SYNCDELAY;

       FIFORESET = 0x80;  // set NAKALL bit to NAK all transfers from host
       SYNCDELAY;
       FIFORESET = 0x02;  // reset EP2 FIFO
       SYNCDELAY;
       FIFORESET = 0x04;  // reset EP4 FIFO
       SYNCDELAY;
       FIFORESET = 0x06;  // reset EP6 FIFO
       SYNCDELAY;
       FIFORESET = 0x08;  // reset EP8 FIFO
       SYNCDELAY;
       FIFORESET = 0x00;  // clear NAKALL bit to resume normal operation
       SYNCDELAY;

       EP6FIFOCFG = 0x0D;
       // this lets the EZ-USB auto commit IN packets, gives the
       // ability to send zero length packets,
       // and sets the slave FIFO data interface to 8-bits
       SYNCDELAY;
       OUTPKTEND = 0x82; // arm first buffer
       SYNCDELAY;
       OUTPKTEND = 0x82; // arm second buffer
       SYNCDELAY;
       OUTPKTEND = 0x82; // arm third buffer
       SYNCDELAY;
       OUTPKTEND = 0x82; // arm fourth buffer

       FIFOPINPOLAR = 0x00; // set all slave FIFO interface pins as active low
       SYNCDELAY;
       OEC |= 0xFF; // Set portc to outputs
}
```

Appendix E -

Initiation of synthesizer

The four first functions are used in initiation of the synthesizer while the last nine cases are used at controlling the synthesizer.

```
BOOL DR_VendorCmnd(void)
{
	// Use Port configuration
	IFCONFIG = 0x80;
	SYNCDELAY;

	switch(bRequest)
	{ //TPM handle new commands

		//int c;

		case VR_MASTER_RESET:
			// Master Reset synthesizer
			master_reset();
		break;
		case VR_IO_RESET:
			// IO Reset synthesizer, clears the buffer
			IO_reset();
		break;
		case VR_IO_UPDATE:
			// IO update synthesizer, transport buffer to register
			IO_update();
		break;
		case VR_SDIO:
			// Send address
			CLK_byte(wValueL);
		break;
		case VR_INIT_DELTA_T1:
			// Receive delta time t1
			delta_t[1]=wValueL;
		break;
		case VR_INIT_DELTA_T2:
			// Receive delta time t2
			delta_t[0]=wValueL;
		break;
		case VR_TOGGLE_CONTROL_MODE:
			// Toggle between control mode ON or OFF
			if(run_control_mode)
				run_control_mode = FALSE;
			else
			{
				run_control_mode = TRUE;
				if(wValueL)
					run_control_mode_stat = FALSE;
				else
					run_control_mode_stat = TRUE;
			}
		break;
```

```
            case VR_REG_ADDR_1:
                  // Receive address to register for "multi-object" control mode
                  update_reg[0][0]=wValueL;
            break;
            case VR_REG_DATA1_1:
                  // Receive data1 to register for "multi-object" control mode
                  update_reg[0][1]=wValueL;
            break;
            case VR_REG_DATA2_1:
                  // Receive data1 to register for "multi-object" control mode
                  update_reg[0][2]=wValueL;
            break;
            case VR_REG_ADDR_2:
                  // Receive address to register for "multi-object" control mode
                  update_reg[1][0]=wValueL;
            break;
            case VR_REG_DATA1_2:
                  // Receive data1 to register for "multi-object " control mode
                  update_reg[1][1]=wValueL;
            break;
            case VR_REG_DATA2_2:
                  // Receive data1 to register for "multi-object" control mode
                  update_reg[1][2]=wValueL;
            break;
            default:
              ;
      }

      // Return to FX2 in slave FIFO mode
      IFCONFIG = 0x03;
      SYNCDELAY;

      return(FALSE);
}
```

Appendix F-

Control of synthesizer

```
void TD_Poll(void) // Called repeatedly while the device is idle
{

       if(run_control_mode)
       {

         if(run_control_mode_stat)
         {
                    PS_OR_SYNC0=1;
                           EZUSB_Delay(delta_t[1]*1000);    // Wait delta t2
                    PS_OR_SYNC0=0;
                           EZUSB_Delay(delta_t[0]*1000);    // Wait delta t1

         }else{     // if multi-object mode

                int i, k, j;             // Loop variabels

                // Loop between four different states
                if (j==1 & k==1)
                     k=0;
                else if(j==1 & k==0)
                {
                     j=0;
                     k=1;
                }else if(j==0 & k==1)
                     k=0;
                else //(j==0 & k==0)
                {
                     j=1;
                     k=1;
                }

                // Send address
                CLK_byte(update_reg[k][0]);

                // Send data
                for (i=0; i<2; i++)
                {
                     CLK_byte(update_reg[j][i+1]);
                }

                EZUSB_Delay(delta_t[j]*1000);  // Wait

                PS_OR_SYNC0=k;           // change profile

                // IO update synthesizer IO_UPDATE, transport buffer to register
                IO_update();

                 // IO Reset synthesizer IO_RESET
                IO_reset();
         }
       }
}
```

Appendix G-

Labview console

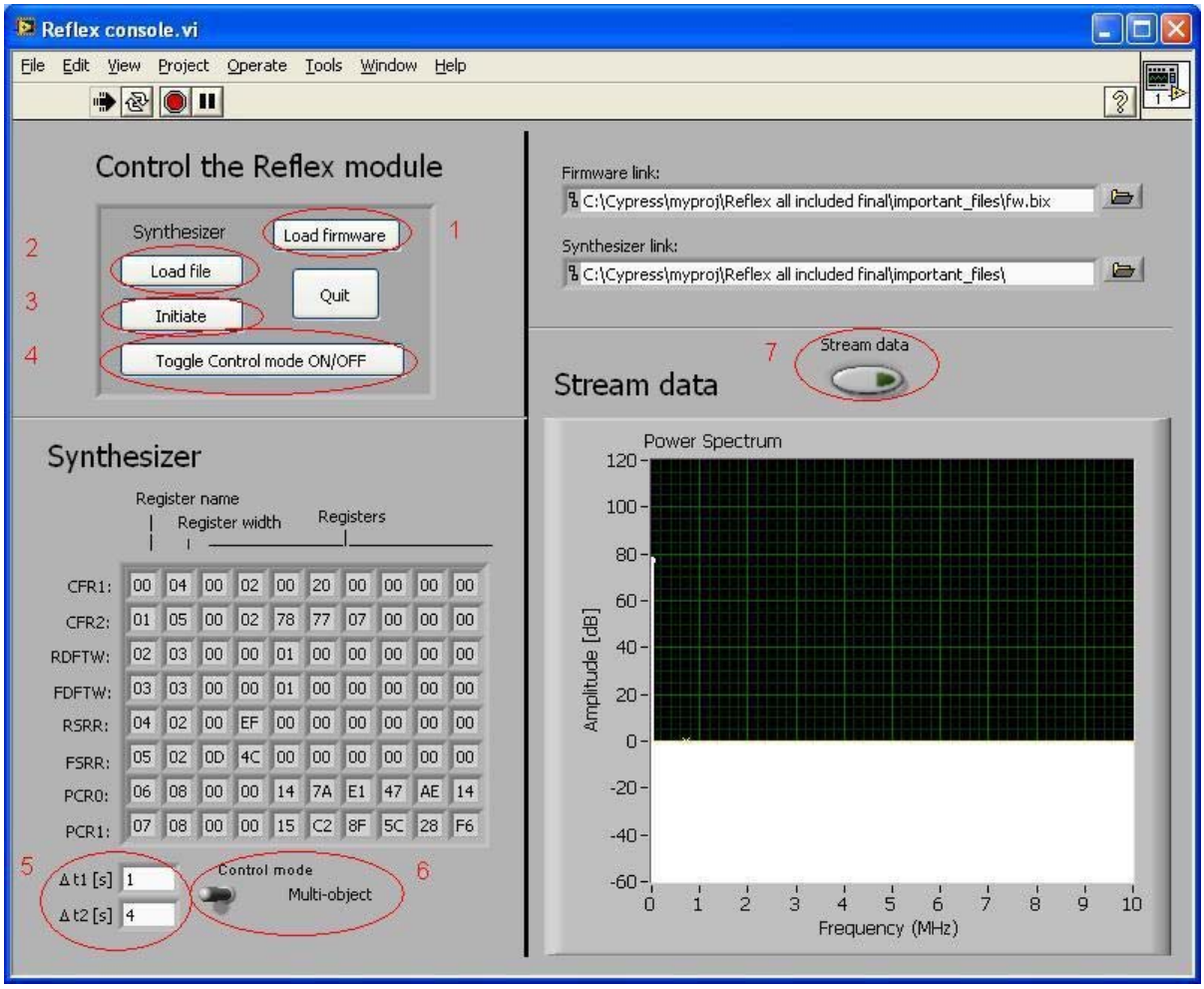


Figure 33: Labview console

1. Load firmware.
2. Load synthesizer file into Labview.
3. Initiate synthesizer.
4. Toggle control mode, ON/ **OFF.**
5. Place value for Δt1 and Δt2.
6. Change control mode: stationary object or **multi-object**.
7. Streaming data, **ON**/ **OFF.**

Appendix H-

Labview block diagram

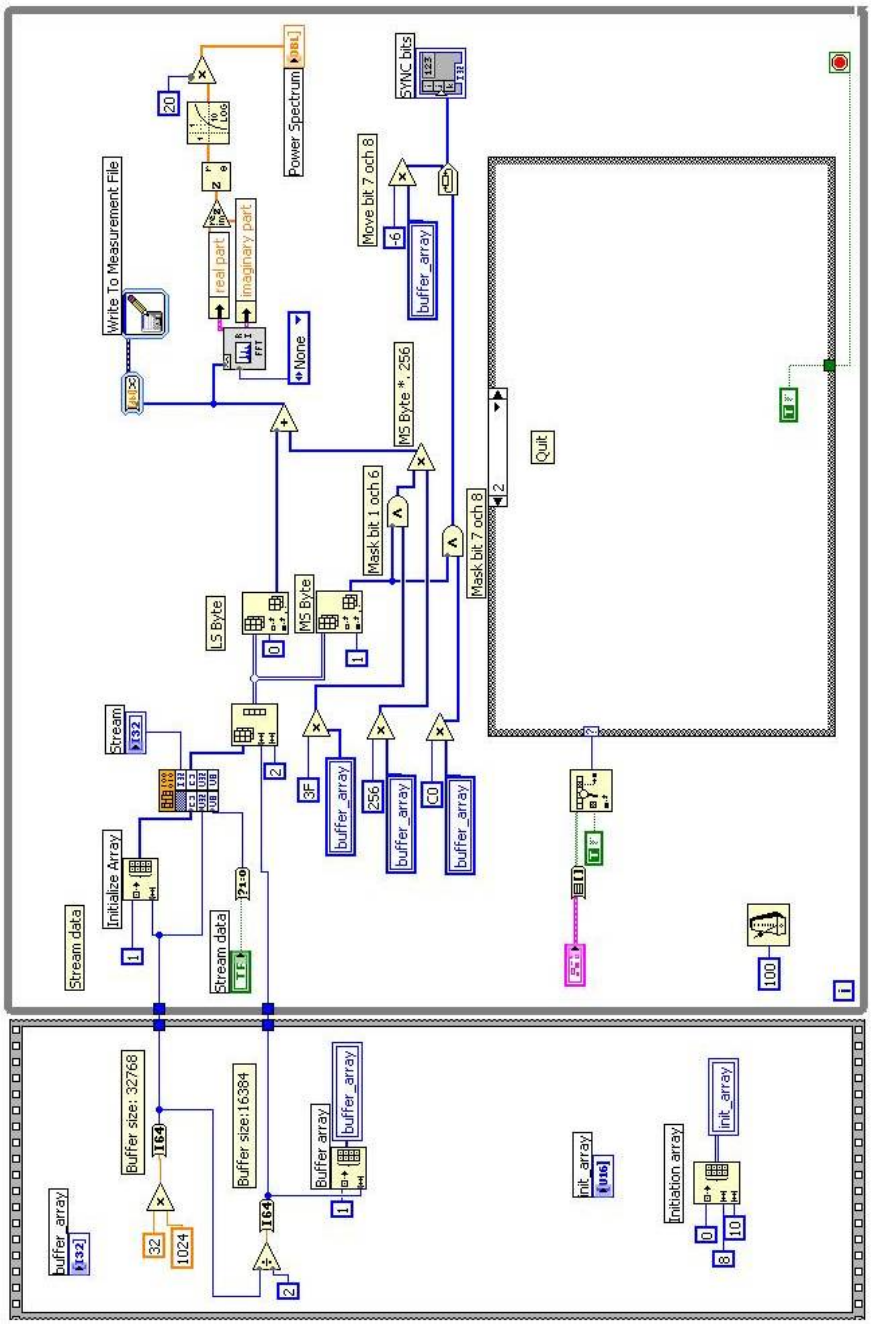


Figure 34: Labview block diagram

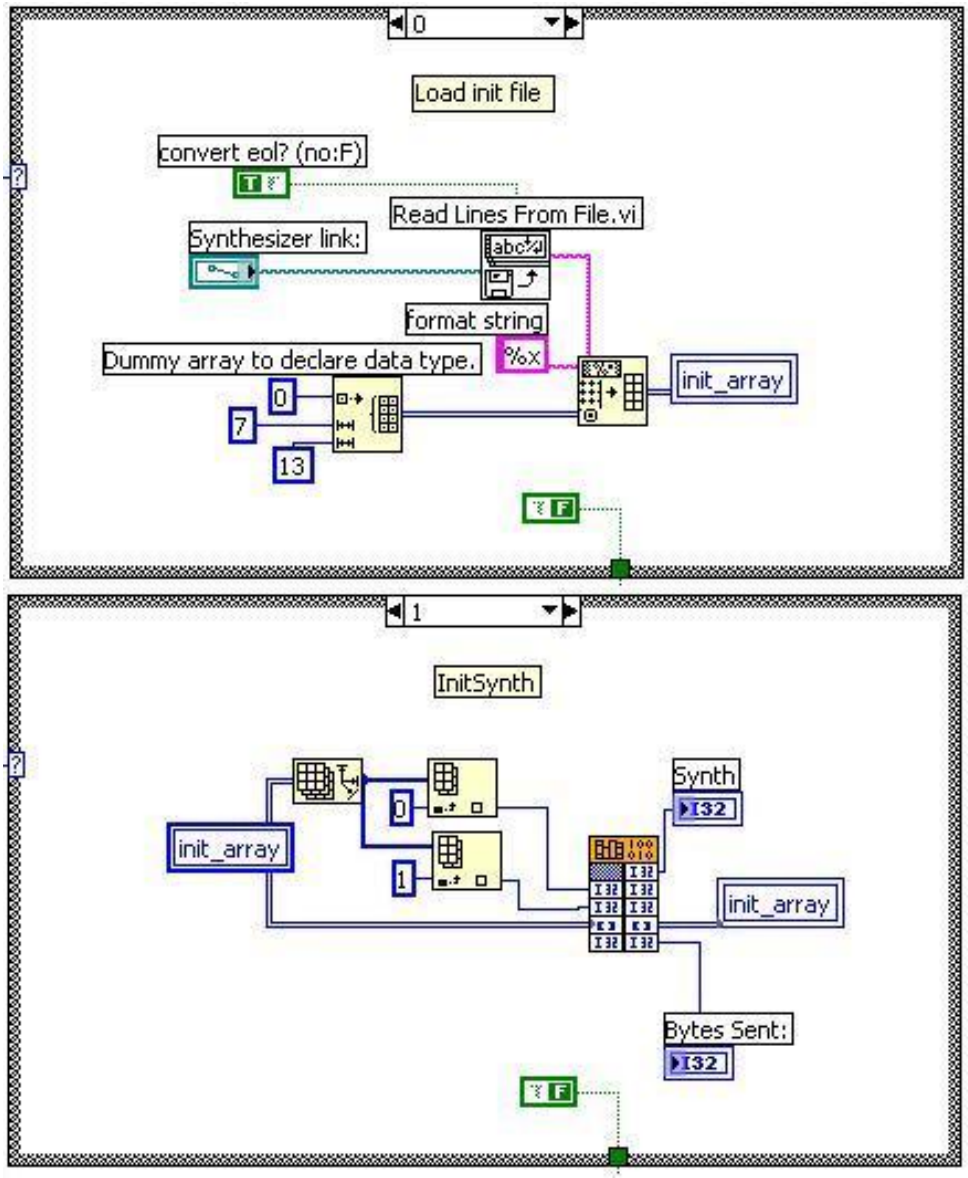


Figure 35: Labview block diagram, case 1 and 2

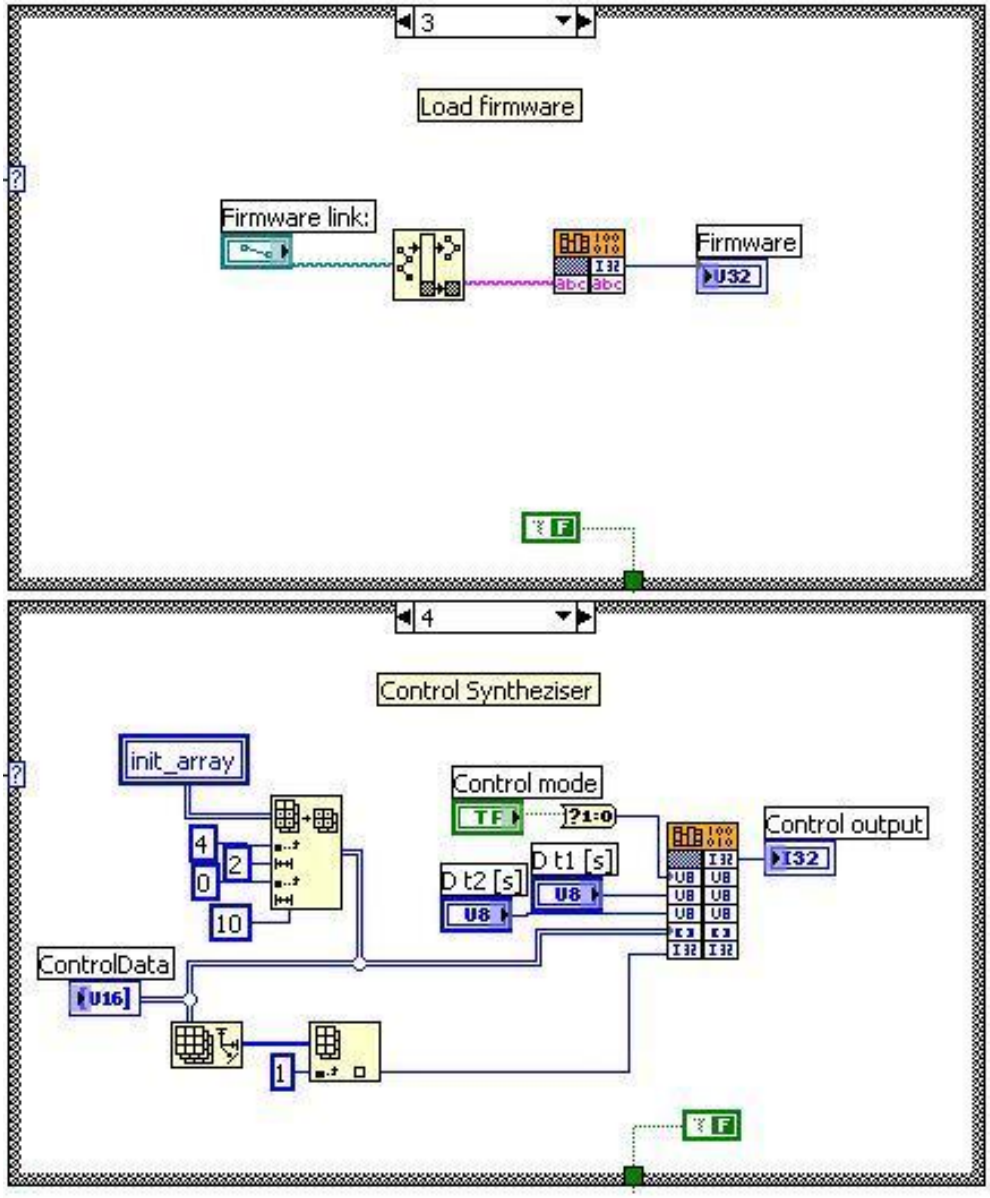


Figure 36: Labview block diagram, case 3 and 4

Appendix I -
File Reference

Functions Documentation of Labview_to_Dll.cpp:

```
int MyStream(    uchar *in_array,
    unsigned long size,
    uchar stream)
```

Streams data from ADC

Parameters:

in_array	Array containing the received data.
size	Constant buffer size at 32k for windows XP.
stream	Disable buffer read

Returns:

0	No error
-1	No stream
x	Error code

```
int LoadFirmware(LStrHandle name)
```

Sending firmware to USB device

Parameters:

name	String consisting of the firmware file.

Returns:

0	No error
x	Error code

```
int InitSynth(     int nrofrows,
    int nrofcols,
    uchar *myData,
    int *ret_data)
```

Initiate the synthesizer with attached array

Parameters:

nrofrows	Number of rows in array
nrofcols	Number of columns in array
myData	Array containing data to send
ret_data	Variable calculating number of data sent

Returns:

0	No error
x	Error code

```
init ControlSynth(      uchar control_mode,
    uchar delta_t1,
    uchar delta_t2,
    uchar *tinyArray,
    int nrofcols)
```

Controls the synthesizer with attached array

Parameters:

delta_t1	Delay parameter
delta_t2	Delay parameter
tinyArray	Array containing data to send
nrofcols	Number of columns in array

Returns:

0	No error
x	Error code

Functions Documentation of Labview_to_Dll.cpp:

Functions Documentation

```
int deviceRead(         uchar b[],
                        unsigned long& bytesTransfered)
```

Fill buffer with data from bulk read.

Parameters:

b	buffer to fill (BUFSIZE bytes).
bytesTransfered	In: bytes to use in the buffer. (Will be BUFSIZE if 0 or > BUFSIZE.) Out: bytes actually read.

Returns:

0	If read were ok.
1	ERROR: Failed to open driver.
2	ERROR: Incomplete read: bytesTransfered < bytesDesired
3	ERROR: Device call (BULK_TRANSFER) failed.

int deviceWrite(	uchar vx_cmd,
	uchar data)

Write vendor command to device.

Parameters:

vx_cmd	Command code
data	Command data

Returns:

0	Vendor were sent ok.
1	ERROR: Failed to open driver.
2	ERROR: Write failed or unknown command.

int bDownloadFirmware(const char* fwFileName= NULL)

Read firmware file and transfer to device.

Parameters:

fwFileName	Firmware filename.

Returns:

0	If all went well.
1	ERROR: Failed to open (firmware) file.
2	ERROR: Failed to put device in reset.
3	ERROR: Failed to open driver.

4 ERROR: Firmware download request failed.

Variables Documentation

```
const uint BUFSIZE= 32*1024
```

Max number of bytes to read from device.

```
const char COMPLETE_DEVICE_NAME[64] = "\\\\.\\EZUSB-0"
```

Device name of USB device.

```
const uint MAX_FILE_SIZE=1024*8
```

Size of firmware transfer buffer (max size).

```
#define VR_INIT_DELTA_T1          0xB0          // Initiate delta t1
#define VR_INIT_DELTA_T2          0xB1          // Initiate delta t2
#define VR_TOGGLE_CONTROL_MODE  0xB2      // Start/ Stop control mode
#define VR_REG_ADDR_1   0xB3    // Receive address register for "multi-object" control mode
#define VR_REG_DATA1_1 0xB4    // Receive data1 register for "multi-object" control mode
#define VR_REG_DATA2_1 0xB5    // Receive data1 register for "multi-object" control mode
#define VR_REG_ADDR_2   0xB6    // Receive address register for "multi-object" control mode
#define VR_REG_DATA1_2 0xB7    // Receive data1 register for "multi-object" control mode
#define VR_REG_DATA2_2 0xB8    // Receive data1 register for "multi-object" control mode

#define VR_IO_RESET               0xC0    // PC0   I/O reset, clears buffer
#define VR_MASTER_RESET         0xC1    // PC1   Master reset on synthesizer
#define VR_SDIO                   0xC3    // PC3 Serial data in-/ out- put (2 or 3 wire)
#define VR_IO_UPDATE            0xC5    // PC5 I/O update, moves buffer to register
```

Definition of vendor commands.

Appendix J -

DLL Code

```
// Labview_to_Dll.cpp : Communicates with Labview

#include "stdafx.h"
#include "Dll_to_Cypress.h"
#include <stdlib.h>
#include <extcode.h>

//Function declarations
extern "C" __declspec(dllexport) int __cdecl MyStream(uchar *in_array, unsigned
long size, uchar stream);
extern "C" __declspec(dllexport) int __cdecl LoadFirmware(LStrHandle name);
extern "C" __declspec(dllexport) int __cdecl InitSynth(int nrofrows, int nrofcols,
uchar *myData, int *ret_data);
extern "C" __declspec(dllexport) int __cdecl ControlSynth(uchar control_mode, uchar
delta_t1, uchar delta_t2, uchar *tinyArray, int nrofcols);

BOOL APIENTRY DllMain( HANDLE hModule,
                                    DWORD  ul_reason_for_call,
                                    LPVOID lpReserved
                                    )
{
       return TRUE;
}

/** Streams data from ADC
       Parameters:
              in_array     Array containing the received data.
              size         Constant buffer size at 32k for windows XP.
              stream              Disable buffer read
       Returns:
              0      no error
              -1     no stream
              x      error code
*/
__declspec(dllexport) int __cdecl MyStream( uchar *in_array, unsigned long size,
uchar stream)
{

       if (stream)
       {
              // Receive data...
              uchar b[BUFSIZE];
              unsigned long bytesTransfered = size;

              int transfer_error = deviceRead( b, bytesTransfered );

              if(!transfer_error)
                     memcpy(in_array,b,BUFSIZE); // Copy buffer to Labview array

              return transfer_error;
```

```
	}
	return -1;	//No data to transfer...
}

/** Sending firmware to USB device
	Parameters:
		name	String consisting of the firmware file.
	Returns:
		0	No error
		x	Error code
*/
__declspec(dllexport) int __cdecl LoadFirmware(LStrHandle name)
{
	int fw_error_code;  // Holds returned error

	// Convert from 'uChar *' to 'char *'
	char* filename = (char*) malloc((*name)->cnt*sizeof(uChar)+1);
	for (int i=0; i<(*name)->cnt; ++i)
		filename[i] = (*name)->str[i];
	filename[(*name)->cnt]=0;

	fw_error_code = bDownloadFirmware( filename );
	free(filename);

	return fw_error_code;	// Return error code
}

/** Initiate the synthesizer with attached array
	Parameters:
		nrofrows	Number of rows in array
		nrofcols	Number of columns in array
		myData		Array containing data to send
		ret_data	Variable calculating number of data sent
	Returns:
		0	No error
		x	Error code
*/
__declspec(dllexport) int __cdecl  InitSynth(int nrofrows, int nrofcols, uchar
*myData, int *ret_data)
{
	int vc_error_code;  // Error code
	uchar send_data;    // Data to be sent
	uchar nr_of_data;   // Number of data to send
	int bytes_sent=0;   // Counts how many bytes were sent


	// Reset synthesizer
	vc_error_code= deviceWrite( VR_MASTER_RESET, 0x00);

	// Reset buffer on synthesizer
	vc_error_code= deviceWrite( VR_IO_RESET, 0x00);

	for (int i=0; i<nrofrows; i++)   // Loops each row
	{
		send_data= myData[i*nrofcols+0]; // Get address from this row
		vc_error_code= deviceWrite( VR_SDIO, send_data); // Write to device

		// How many bytes data to send
		nr_of_data= myData[i*nrofcols+1];
```

```
		for (int j=0; j<nr_of_data; j++) // Loops each element
		{
			// Get data from this element
			send_data= myData[i*nrofcols+(j+2)];

			// Write to device
			vc_error_code= deviceWrite( VR_SDIO, send_data);
		}

		// Transfer synthesizer buffer into register
		vc_error_code= deviceWrite( VR_IO_UPDATE, 0x00); // Write to device

		// Assure buffer is empty
		vc_error_code= deviceWrite( VR_IO_RESET, 0x00);

		// Increase number of bytes were sent
		bytes_sent += 1 + nr_of_data;	// Address plus data
	}


	// Return how many bytes were sent
	(*ret_data) = bytes_sent;

	return vc_error_code;

}

/** Controls the synthesizers
	Parameters:
		control_mode	Diffrent control modes
		delta_t1	Delta t1, delay parameter
		delta_t2	Delta t2, delay parameter
		tinyArray	Array containing data to send
		nrofcols	Number of columns in array
Returns:
		0	no error
		x	error code
*/
__declspec(dllexport) int __cdecl ControlSynth(uchar control_mode, uchar delta_t1,
uchar delta_t2, uchar *tinyArray, int nrofcols)
{

	int vc_error_code;  // Error code

	if (control_mode)   // Stationary mode
	{

		// Send Delta time 1
		vc_error_code= deviceWrite( VR_INIT_DELTA_T1, delta_t1);
		// Send Delta time 1
		vc_error_code= deviceWrite( VR_INIT_DELTA_T2, delta_t2);

		// Toggle on or off the control mode
		vc_error_code= deviceWrite( VR_TOGGLE_CONTROL_MODE, 0x00);

		return vc_error_code;

	} else {	// Multi-object mode
```

```
            uchar send_data;    // Data to be sent

            // Send Delta time 1
            vc_error_code= deviceWrite( VR_INIT_DELTA_T1, delta_t1);
            // Send Delta time 1
            vc_error_code= deviceWrite( VR_INIT_DELTA_T2, delta_t2);

            // First row
            send_data= tinyArray[0*nrofcols+0];     // Get address
            vc_error_code= deviceWrite( VR_REG_ADDR_1, send_data);
            // Write to device

            send_data= tinyArray[0*nrofcols+2];// Get data from first element
            vc_error_code= deviceWrite( VR_REG_DATA1_1, send_data);

            send_data= tinyArray[0*nrofcols+3];// Get data from second element
            vc_error_code= deviceWrite( VR_REG_DATA2_1, send_data);

            // Second row
            send_data= tinyArray[1*nrofcols+0];     // Get address
            vc_error_code= deviceWrite( VR_REG_ADDR_2, send_data);

            send_data= tinyArray[1*nrofcols+2];// Get data from first element
            vc_error_code= deviceWrite( VR_REG_DATA1_2, send_data);

            send_data= tinyArray[1*nrofcols+3];// Get data from second element
            vc_error_code= deviceWrite( VR_REG_DATA2_2, send_data);

            // Toggle on or off the control mode
            vc_error_code= deviceWrite( VR_TOGGLE_CONTROL_MODE, 0x01);

            return vc_error_code;
        }
}
```

```
// Dll_to_Cypress.h : Defines variables

#ifndef DLL_TO_CYPRESS_H
#define DLL_TO_CYPRESS_H

typedef unsigned int uint;
typedef unsigned char uchar;
#include <stddef.h>

/** Max number of bytes to read from device. */
const uint BUFSIZE= 32*1024;

/** Device name of USB device. */
const char COMPLETE_DEVICE_NAME[64] = "\\\\.\\EZUSB-0";

/** Size of firmware transfer buffer (max size). */
const uint MAX_FILE_SIZE=1024*8;

/** Vendor request */
#define VR_IO_RESET        0xC0   // PC0 I/O reset, clears buffer
```

```
#define VR_MASTER_RESET    0xC1   // PC1 Master reset on synthesizer
//PC2 Serial data output (3 wires). Not implemented
#define VR_SDIO            0xC3   // PC3 Serial data in-/ out- put (2 or 3 wire)
// PC4 Serial clock (2 or 3 wire) SCLK. Handled by USB chip
#define VR_IO_UPDATE       0xC5   // PC5 I/O update, moves buffer to register

//Function declarations
int deviceRead( uchar b[], unsigned long& bytesTransfered);
int deviceWrite( uchar vx_cmd, uchar data);
int bDownloadFirmware( const char* fwFileName= NULL);

#endif // DLL_TO_CYPRESS_H
```

```
// Dll_to_Cypress.cpp : Communicates with USB device

#include "stdafx.h"
#include "Dll_to_Cypress.h"

// We only need to compile this on XP
#ifdef _MSC_VER            // Microsoft VC compiler
#ifndef DUMMY_USB          // Not using dummy driver

#include <iostream>
#include <extcode.h>
#include <windows.h>
#include <winioctl.h>
#include "ezusbsys.h"

/** Open the driver COMPLETE_DEVICE_NAME and assign to handle pointer.
      Parameters:
            handle       Pointer to the handle created for the driver.
            exclusive    Open the driver without FILE_SHARE_WRITE (should block
other open requests).
      Returns:
            TRUE   if successful, else FALSE
*/
BOOL bOpenDriver( HANDLE* handle, BOOL exclusive= FALSE )
{
      uint mode= exclusive ? 0 : FILE_SHARE_WRITE;
      // Open driver handle
      *handle= CreateFile( COMPLETE_DEVICE_NAME,
            GENERIC_WRITE,
            mode,
            NULL,
            OPEN_EXISTING,
            0,
            NULL );
      if ( *handle == INVALID_HANDLE_VALUE )
      {
            // ERROR: Invalid handle.\n
            return FALSE;
      }
      else
            return TRUE;
}

/**    A part of firmware load procedure
```

```
	Parameters:
		reset  Put 8051 in reset if TRUE
	Returns:
		TRUE   if successful, else FALSE
*/
BOOL bSetFX2Reset( BOOL reset )
{
	HANDLE  hDevice = NULL;
	DWORD nBytes;
	BOOL bResult;
	VENDOR_REQUEST_IN resetRequest;
	if ( ! bOpenDriver( &hDevice, TRUE ) ) // Open exclusive
		return FALSE;// ERROR: Failed to open driver.

	resetRequest.bRequest = 0xA0;
	resetRequest.wValue       = 0xE600; // using CPUCS.0 in FX2
	resetRequest.wIndex       = 0x00;
	resetRequest.wLength      = 0x01;
	resetRequest.bData  = reset ? 1 : 0;
	resetRequest.direction = 0x00;
	// Perform the Get-Descriptor IOCTL
	bResult = DeviceIoControl ( hDevice,
		IOCTL_Ezusb_VENDOR_REQUEST,
		&resetRequest,
		sizeof(VENDOR_REQUEST_IN),
		NULL,
		0,
		(unsigned long *)&nBytes,
		NULL );
	// Close the handle
	CloseHandle( hDevice );
	if ( bResult != TRUE )
	{
		if ( !reset )
			// ERROR: 'Leave reset' request failed.
			return FALSE;
		else
			// ERROR: 'Enter reset' request failed.
			return FALSE;
	}
	return TRUE;
}

/** Fill buffer with data from bulk read.
	Parameters:
		b	buffer to fill (BUFSIZE bytes).
		bytesTransfered	In: bytes to use in the buffer. (Will be BUFSIZE
if 0 or > BUFSIZE.) Out: bytes actually read.
	Returns:
		0	If read were ok.
		1	ERROR: Failed to open driver.
		2	ERROR: Incomplete read: bytesTransfered < bytesDesired
		3	ERROR: Device call (BULK_TRANSFER) failed.
*/
int deviceRead( uchar b[], unsigned long& bytesTransfered)
{
	HANDLE hInDevice = 0;
	unsigned long bytesDesired= bytesTransfered;
	if ( ( bytesDesired == 0 ) || ( bytesDesired > BUFSIZE ) )
```

```
            bytesDesired= BUFSIZE;     // Make sure we have a sane value
        bytesTransfered= 0;
        if ( ! bOpenDriver( &hInDevice, FALSE ) )
            return 1;     // ERROR: Failed to open driver.

        // Read
        BULK_TRANSFER_CONTROL inBulkControl; // struct w. ULONG pipeNum
        inBulkControl.pipeNum= 2;         // Use endpoint 6
        BOOL status= FALSE;
        status= DeviceIoControl( hInDevice,
            IOCTL_EZUSB_BULK_READ,
            (PVOID)&inBulkControl,
            sizeof(BULK_TRANSFER_CONTROL),
            b,
            bytesDesired,
            &bytesTransfered,
            NULL );
        CloseHandle( hInDevice );
        if ( bytesTransfered != bytesDesired )
            return 2; // ERROR: Incomplete read: bytesTransfered < bytesDesired

        if ( status != TRUE )
            return 3;     // ERROR: Device call (BULK_TRANSFER) failed.

        return 0;
}

/** Write vendor command to device.
        Parameters:
            vx_cmd       Command code
            data   Command data
        Returns:
            0      Vendor were sent ok.
            1      ERROR: Failed to open driver.
            2      ERROR: Write failed or unknown command.
*/

int deviceWrite( uchar vx_cmd, uchar data)
{
        HANDLE hInDevice = 0;
        if ( ! bOpenDriver( &hInDevice ) )
            return 1;     // ERROR: Failed to open driver.
        // Write
        VENDOR_OR_CLASS_REQUEST_CONTROL vxControl;
        vxControl.requestType= 2; // 1=class 2=vendor
        vxControl.request= vx_cmd;
        vxControl.recepient= 0; // 0=device, 1=interface, 2=endpoint,3=other
        vxControl.direction= 0; // 0=host to dev, 1=dev to host
        vxControl.value= data;
        vxControl.index= 0;
        ULONG bytesTransfered;
        BOOL status= DeviceIoControl( hInDevice,
            IOCTL_EZUSB_VENDOR_OR_CLASS_REQUEST,
            &vxControl,
            sizeof(VENDOR_OR_CLASS_REQUEST_CONTROL),
            NULL,
            0,
            &bytesTransfered,
            NULL );
```

```
	CloseHandle( hInDevice );
	if ( status != TRUE )
		return 2;		// ERROR: Vendor command write failed.

	return 0;
}

/** Read firmware file and transfer to device.
	Parameters:
		fwFileName	Firmware filename.
	Returns:
		0	If all went well.
		1	ERROR: Failed to open (firmware) file.
		2	ERROR: Failed to put device in reset.
		3	ERROR: Failed to open driver.
		4	ERROR: Firmware download request failed.
*/
int bDownloadFirmware( const char* fwFileName )
{
	int numread = 0;
	BOOL bResult= FALSE;
	DWORD nBytes;
	HANDLE  hDevice = NULL;
	char temp[128]  = "";
	unsigned char buffer[MAX_FILE_SIZE];

	// Get the firmware
	FILE *fp;
	if ( (fp = fopen( fwFileName, "rb" ) ) == NULL ) {
		// ERROR: Failed to open (firmware) file.
		return 1;
	}
	numread = fread(buffer,sizeof(unsigned char),MAX_FILE_SIZE,fp);
	fclose(fp);


	// Transfer to device
	if ( ! bSetFX2Reset( TRUE ) ) // Set the CPU in the reset state
	{
		// ERROR: Failed to put device in reset.
		return 2;
	}
	if ( ! bOpenDriver( &hDevice ) )
	{
		// ERROR: Failed to open driver.
		return 3;
	}

	bResult= DeviceIoControl( hDevice,
		IOCTL_Ezusb_ANCHOR_DOWNLOAD,
		buffer,
		numread,
		NULL,
		0,
		(unsigned long *)&nBytes,
		NULL); // Transmit data
	CloseHandle( hDevice ); // Close the handle

	// Make the CPU leave the reset state
```

```
        BOOL leftReset= bSetFX2Reset( FALSE );
        if ( bResult != TRUE ) {   // Check result
               // ERROR: Firmware download request failed.
               return 4;
        }
        Sleep( 5000 );
        return 0;                  // If all went well.
}

#endif // Not DUMMY_USB
#endif        // Microsoft VC compiler
```

Appendix K -

Firmware initiation phase

1) Use IOCTL_Ezusb_VENDOR_REQUEST to write 0x01 to CPUCS_REG_EZUSB to reset the 8051.

2) If your firmware fits completely into internal ram, go to step 4.

3) If any part of your firmware will be in external ram, you must do these steps:

a) Download the file "Vend_AX.hex". Open that file. With each line of data from the file, convert the string fields to numeric or binary data according to the format of INTEL_HEX_RECORDS (length, address, type, data). Download each data segment using IOCTL_EZUSB_ANCHOR_DOWNLOAD.

b) After all Vend_AX data has been downloaded, use IOCTL_Ezusb_VENDOR_REQUEST to write 0x00 to CPUCS_REG_EZUSB to release the 8051.

c) Open your firmware .hex file. Read each line of your firmware file and convert the INTEL_HEX_RECORDS data to numeric values as described in step 3.a. If the target address of any segment is in external ram, then download that data segment using IOCTL_EZUSB_VENDOR_OR_CLASS_REQUEST. In this step, ignore internal ram segments. Repeat this step for all external ram segements.

d) Use IOCTL_Ezusb_VENDOR_REQUEST to write 0x01 to CPUCS_REG_EZUSB to reset the 8051.

4) Open your firmware .hex file. Read each line of your firmware file and convert the INTEL_HEX_RECORDS data to numeric values as described in step 3.a. If you skipped step 3, then download each data segment using IOCTL_EZUSB_ANCHOR_DOWNLOAD. Otherwise, just download the internal ram segments. Repeat this step for all internal ram segments.

5) After all data has been downloaded; use IOCTL_Ezusb_VENDOR_REQUEST to write 0x00 to CPUCS_REG_EZUSB to release the 8051.

[Reference Cypress.com]

Appendix L -

Pin function description

The USB Device, CY7C68013A, 100 pins, VCC = 3 to 3.6 V:

Pin nr. on USB	Direction	Function	Connected to (Pin nr.)
1, 9, 16, 20, 33, 38, 49, 53, 66, 78, 85,	I	--	VCC
2, 12, 19, 21, 39, 48, 50, 65, 75, 94, 99	O	--	GND
3	I	Read from slave	VCC
4	I	Write to slave	GND
5, 6, 7, 8, 13, 14, 15, 22, 23, 24, 25, 29, 30, 31, 32, 40, 41, 42, 43, 51, 52, 67, 68, 70, 76, 79, 84, 100	--	Not used	Not connected
10, 11	I, O	XTALOUT, XTALIN	24 MHz oscillator
17	I/O	USB activity, DPLUS	USB connector, DP
18	I/O	USB activity, DMINUS	USB connector, DM
26	I	External clock, IFCLK	Oscillator at synthesizer
27	I	RESERVED	10K Resistor to GND
28	O	Break signal	Diode or measurable pin
34, 35, 36, 37, 44, 45, 46, 47	I, PBx	Port B FD[0:7]	D[0:7] on ADC
54, 55, 56, 74	O	Configurable flags for debug	Diode or measurable pin
57	O, PC0	I/O reset, clears buffer	Sync_IO (9) on synthesizer
58	O, PC1	Master reset on synth.	Reset (10) on synthesizer
59	I, PC2	Serial data output (3 wire)	SDO (13) on synthesizer
60	O, PC3	Serial data in-/ out- put (2 or 3 wire)	SDIO (14) on synthesizer
61	O, PC4	Serial clock (2 or 3 wire)	SCLK (15) on synthesizer
62	O, PC5	I/O update, moves buffer to register	IO_Update (20) on synthesizer
63	O, PC6	SYNC0, distinguee packets and profile select on synth.	Port D FD[14] on USB (97) and PS0 (21) on synthesizer

64	I, PC7	SYNC1, distinguee packets	Port D FD[15] on USB (98)
69	I	Slave output enable	GND
71	I	FIFOADR0	GND, Use endpoint 6
72	I	FIFOADR1	VCC, Use endpoint 6
73	I	Commits none full IN packets	VCC
77	I	Reset USB chip	Switch and diode
80, 81, 82, 83, 95, 96	I	Port D FD[8:13]	D[8:13] on ADC
97, 98	I	Port D FD[14:15]	SYNCx (63, 64) on USB
86, 87, 88, 89, 90, 91, 92, 93	I/O	Port E unused	unused

Table 13: Pin function description for USB device

The synthesizer, AD9956, 48 pins, VDD = 1,8 V ±5% VDD_IO = CP_VDD = 3.3 V ±5%

Pin nr. on synth.	Direction	Function	Connected to (Pin nr.)
2, 4, 7, 11, 25, 27, 38, 44, 48	I	--	VDD
17, 31, 35	I	--	VDD_IO
1, 3, 8, 12, 16, 22, 23, 24, 26, 30, 34, 37, 43	O	--	GND
18, 19, 32, 33	--	Not used	Not connected
5 (6)	O	DAC Analog output (complementary)	Through Loop filter to PLLOSC (42 and complementary 41)
9	I	Sync_IO, reset buffer	PC2 on USB (57)
10	I	Reset, reset accumulators and registers	PC3 on USB (58)
13	O	Serial data output (3 wire)	PC4 on USB (59)
14	I	Serial data in-/ out- put (2 or 3 wire)	PC5 on USB (60)
15	I	Serial clock (2 or 3 wire)	PC6 on USB (61)
20	I	I/O update, moves buffer to register	PC7 on USB (62)
21	I	SYNC0, distinguee packets and profile select on synth.	PC7 on USB (63)

29 (28)	I	R Divider and DDS input (complementary)	VCO output from MMIC
36	O	Charge pump output	To Loop filter
39 (40)	I	PLLREF reference (complementary)	25 MHz oscillator
42 (41)	I	PLLOSC oscillator (complementary)	From DAC(5 complementary 6) through Loop filter
45	O	--	Resistor 3.09K to GND
46	O	--	Resistor 4.02K to GND
47	O	--	Resistor 3.92K to GND

Table 14: Pin function description for synthesizer

The ADC, AD9245, 32 pins, VDD = 2.7 to 3.6 V

Pin nr. on ADC	Direction	Function	Connected to (Pin nr.)
16	I	--	VDD
27, 32	I	AVDD	TBD, depends on which mode
15, 28, 31	O	--	GND
1,3	--	--	Do not connect
2	I	Clock	20 MHz oscillator
4, 21, 24, 25, 26	--	Not used	Not connected
5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 17, 18, 19, 20	O	D[0:13]	Port B FD[0:13] on USB
22	I	Enabling 2-compl. mode	VDD
23	I	Internal 1 V reference	GND
29 (30)	I	Analog input (complementary)	IF signal from MMIC

Table 15: Pin function description for ADC

Bibliography

[1] Curry, R: (2004). Radar System Performance Modeling (Second Edition). Ebrary, http://site.ebrary.com/lib/linkoping/Doc?id=10081989&ppg=1. (2006-02-08)

[2] Komarov, I: (2003). Fundamentals of Short-Range FM Radar. Ebrary, http://site.ebrary.com/lib/linkoping/Doc?id=10082002&ppg=1. (2006-02-09)

[3] Flikkema, P.: (2001). The RF and Microwave Handbook, EngNetBase http://www.engnetbase.com/ejournals/search/advsearch1.asp. (2006-02-09)

[4] Peregrine. PE3236 PLL datasheet http://www.peregrine-semi.com/content/products/wireless/wireless_pe3236.html. (2006-02-09)

[5] Analog Devices. AD9852 DDS datasheet http://www.analog.com/en/prod/0%2C2877%2CAD9852%2C00.html. (2006-02-09)

[6] Analog Devices. AD9956 2.7Ghz synthesizer http://www.analog.com/en/prod/0%2C2877%2CAD9956%2C00.html. (2006-02-09)

[7] Cypress. CY7C68013A EZ-USB FX2LP USB Microcontroller http://www.cypress.com/portal/server.pt?space=CommunityPage&control=SetCommunity&CommunityID=209&PageID=259&fid=14&rpn=CY7C68013A. (2006-02-09)

[8] Analog Devices. AD9245 ADC datasheet http://www.analog.com/en/prod/0%2C2877%2CAD9245%2C00.html. (2006-02-09)

Copyright

Defence date

2006-03-24

Publishing date (Electronic version)

2006-03-29

Department and Division

Institutionen för systemteknik

Department of Electrical Engineering

Linköpings universitet

Language

X English

Other (specify below)

_____

Report category

Licentiate thesis

Degree thesis

Thesis, C-level

X Thesis, D-level

Other (specify below)

_____

ISBN:

ISRN: LITH-ISY-EX--06/3779--SE

Title of series

Series number/ISSN

URL, Electronic version

http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-6163

Title

Implementation of an FMCW Radar Platform With High-Speed Real-Time Interface

Author

Jonny Svensson

Abstract

In English

Acreo AB has developed a radar prototype used for illustrate how the SiGe technology could be used. The radar prototype needs further development with a fast interface and a more integrated design. The beginning of the report describes the radar technique theory and the composing equations. The theoretical background is used to explain each component of the system. The report continues by specifying the target of the next radar prototype. The chosen implementation is motivated and the mode of procedure is described in detail. Test benches were used to verify correct functionality and some limits were found. The report is concluded with test results and recommendations on further enhancements.

På svenska

Acreo AB har utvecklat en radarprototyp för att illustrera hur SiGe teknologi kan användas. Prototypen behöver vidareutvecklas med ett snabbt digitalt interface och en kompaktare design. Rapporten inleds med att beskriva radarteknikens funktionalitet och de utgörande ekvationerna. Den teoretiska bakgrunden används för att förklara varje komponent som systemet utgörs av. Rapporten fortsätter med att specificera målet med nästa radarprototyp. Den valda implementationen motiveras och tillvägagångssättet beskrivs detaljerat. Testuppkopplingar verifierade korrekt funktionalitet och begränsningar insågs. Rapporten avslutas med en sammanfattning av uppnådda testresultat och rekommendationer på framtida förbättringar.

Keywords

FMWC, radar, MMIC, USB, Labview, dll, firmware