INTERNATIONAL UNIVERSITY COLLEGE
LEUVEN

DESIGN OF A WIRELESS SENSOR NETWORKING TEST-BED

Authors:
Bjorn Deraeve
Roel Storms

Supervisor:
Luc Vandeurzen

April 2013

DESIGN OF A WIRELESS SENSOR NETWORKING TEST-BED

Bjorn Deraeve • *Roel Storms*

Written on behalf of Mr. Luc Vandeurzen

April 2013

Abstract

Audio applications are on of the main subfields of digital signal processing. More specific, in audio and music one can use frequency modulation synthesis in order to make creative and totally different sounding tones. A theoretical background and practical approach with an Analog Devices DSP-board is reported here.

In the first two chapters it is explained how sound works and what aspects of frequency modulation are actually responsible for making a particular tone sound differently. Next a practical approach to add envelopes and effects to the tone is given.

Chapter 5 serves as introduction to 6. In this chapter some theoretical background and electronic designs of CPU architectures are explained. This chapter serves as a backbone for the practical implementations explored in chapter 6. Readers with a basic knowledge on this matter can skip this chapter and immediately start with chapter 6. The first part of this chapter handles the Analog Devices ADSP-21369 DSP board's initialisation routines and the second part focusses on the practical implementation of the signal creation and modulation algorithms.

Finally chapter 7 deals with the communication between the DSP-board and the Labview interface for the FM Synthesizer.

As a conclusion chapter 8 closes this report with some critical reflections on the implementation and team activities.

Contents

List of Figures

Abbreviations

RTC **R**eal **T**ime **C**lock

CPU **C**entral **P**rocessing **U**nit

Chapter 1

Introduction to Wireless Sensor Networks

1.1 Introduction

Sound is the human perception of little changes in air pressure. This oscillation of pressure, referred to as a sound wave, is a mechanical wave propagated through a solid, liquid or gas and is composed of frequencies within the range of hearing. For humans this is between about 20 Hz and 20 000 Hz, however the upper limit generally decreases with one's age. The changes in pressure must also be big enough in order to be audible.

$$L_P = 10 \cdot \log_{10}(\frac{p^2}{p_{ref^2}}) = 20 \cdot \log_{10}(\frac{p}{p_{ref}})\, dB \tag{1.1}$$

where p is the rms value of the measured sound pressure and pref is the reference sound pressure, pref = 20 microPa (rms), considered as the threshold of human hearing.

1.2 Sound characteristics

A sound is defined by several characteristics. The most important are pitch (dutch: 'toonhoogte'), timbre (dutch: 'klankkleur') and are briefly introduced in this section. Other characteristics are volume and the duration of the rising of the sound, of the persistence and of the damping.

1.2.1 Pitch and tone

For the human ear a pure tone is a sound with a constant frequency and timbre. This

1.2.2 Timbre

Timbre is the quality of a sound that distinguishes different musical instruments and

1.2.2.1 Spectral diagram

At this moment it is interesting to have a look at the spectral diagram of a sound. This for the way air pressure changes with time but not for the actual timbre of a sound.

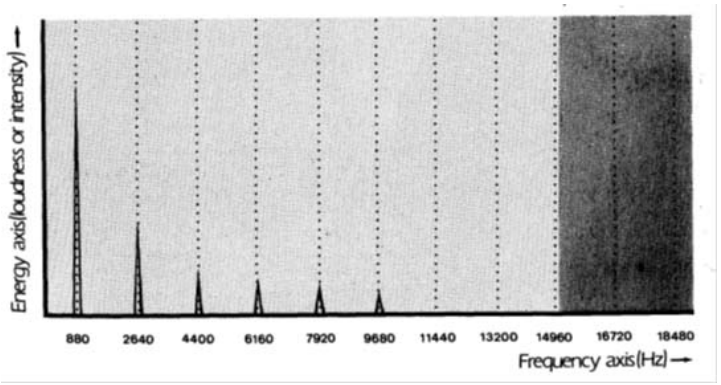

FIGURE 1.1: Spectral diagram of a timbre

How can we interpret figure 1.1? The line on the left is the fundamental fundamental frequency (pitch). In other words: each complex sound can be resolved into mixtures of sine functions which may differ in amplitude and period but of which all periods are related as an integer multiple. This resolving into different components is called a Fourier transformation or analysis of the signal.
The spectrum as in figure 1.1 shows clearly the present frequencies (= partials, overtones or harmonics) and their energy level.

FIGURE 1.2: Sketch of Fourier

Chapter 2

Why ZigBee

Written by Bjorn Deraeve

2.1 Introduction

Thanks to new digital technology there was room for a new synthesis technique, Frequency Modulation Synthesis. The advantage of FM is that a very large number of different sounds can be produced with few elemental units (oscillators). In other words the timbral space is very large. The one requirement is the use of optimized computers, which is discussed in chapter 5.

2.1.1 Basic concepts of modulation synthesis

It is important to know that in the synthesizer only one wave form, a sine wave, is stored. The trick of modulation synthesis is to remodel this wave until a desired sound is found.
An easy example is the square wave. Using additive synthesis a square wave can be built out of an infinite amount of sine functions. This example of additive synthesis also explains the Fourier transformation theory and is shown in figure 2.1.

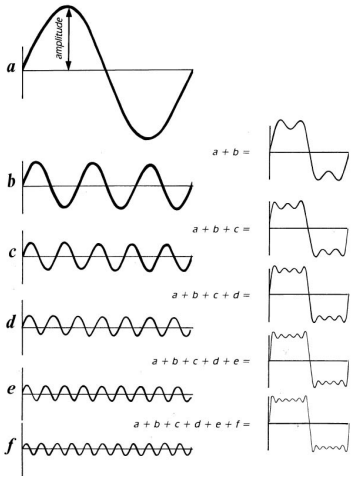

FIGURE 2.1: Additive synthesis of a square wave

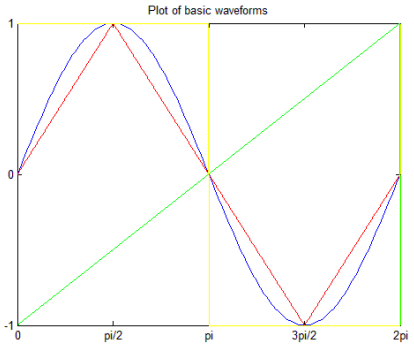

FIGURE 2.2: Basic waveforms

2.1.2 Oscillators and operators

The basic building blocks of FM synthesizers are operators. They are equivalent to oscillators in analogue synthesizers and produce the waveforms the user desires. Where an analogue oscillator produces a changing voltage the digital operator creates discrete samples according to a sine pattern. The operators invite the user to be creative in creating their own unique sound with the aid of different kinds of modulation techniques and waveforms. Knowing that most intersting sounds have many components, with the use of only four operators and modulation techniques one can generate sounds with much more than four partials.

2.1.3 Waveforms

To produce creative sounds it can be appropriate to combine sine waves with other waveforms. Synthesizers are typically able to produce sines, triangles, sawtooths and square waves. A square wave contains only odd-integer harmonic frequencies. To be an ideal wave, the signal should change from high to low instanaeously. However this is impossible to achieve in real-world systems because of the limited bandwith, though good approximations can be made with additive synthesis. The Fourier series of the square wave is given in equation 2.1

$$x(t) = \frac{4}{\pi} \sum_{n=1,3,5,\ldots}^{\infty} \frac{\sin(2\pi \cdot n \cdot f \cdot t)}{n} \tag{2.1}$$

$$x(t) = \frac{4}{\pi}\left(\sin(\omega t) + \frac{1}{3}\sin(3\omega t) + \frac{1}{5}\sin(10\omega t) + \ldots\right) \tag{2.2}$$

Triangle waves contain just like square waves only odd harmonics of the pitch. The difference is that the higher harmonics lose their energy faster. Equation 2.3 gives the Fourier series to approximate a triangle wave using additional synthesis.

$$x(t) = \frac{8}{\pi^2} \sum_{n=1,3,5,\ldots}^{\infty} (-1)^{(n-1)/2} \frac{\sin(2\pi \cdot n \cdot f \cdot t)}{n^2} \tag{2.3}$$

$$x(t) = \frac{8}{\pi^2}\left(\sin(\omega t) - \frac{1}{9}\sin(3\omega t) + \frac{1}{25}\sin(5\omega t) - \ldots\right) \tag{2.4}$$

Finally sawtooth waves contain both even and odd harmonics of the fundamental frequency. The sound of it is even more raw. The equation for additive synthesis is given by equation 2.5.

$$x(t) = \frac{2}{\pi} \sum_{n=1}^{\infty} (-1)^{n+1} \frac{\sin(2\pi \cdot n \cdot f \cdot t)}{n} \tag{2.5}$$

$$x(t) = \frac{2}{\pi}\left(\sin(\omega t) - \frac{1}{2}\sin(2\omega t) + \frac{1}{3}\sin(3\omega t) - \ldots\right) \tag{2.6}$$

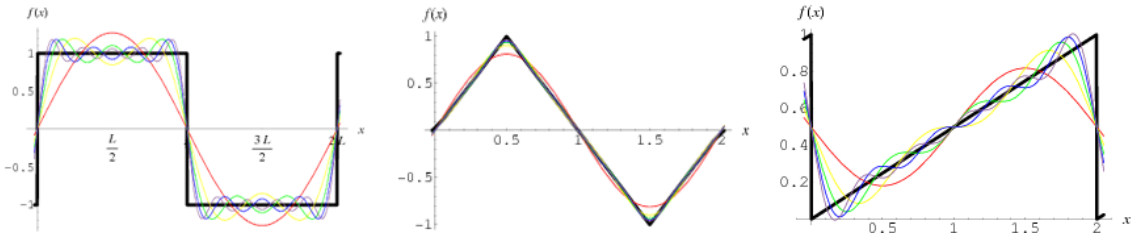

FIGURE 2.3: Square, triangular and sawtooth wave synthesis

Figures 2.4, 2.5 and 2.6 show the corresponding frequency responses of the square, triangle and sawtooth wave and confirm equations 2.1, 2.3 and 2.5.

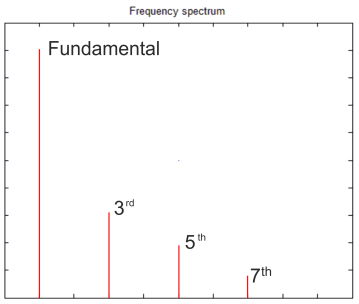

FIGURE 2.4: Frequency spectrum of square wave

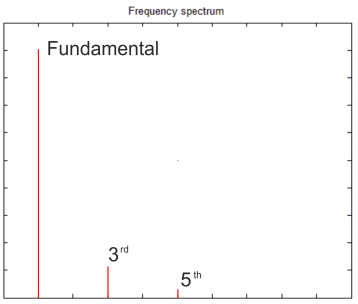

FIGURE 2.5: Frequency spectrum of triangular wave

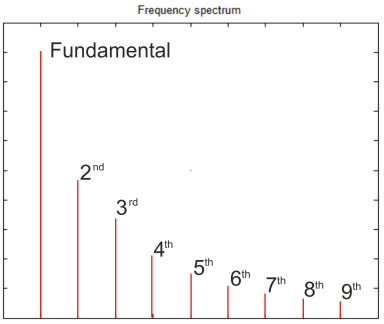

FIGURE 2.6: Frequency spectrum of sawtooth wave

2.2 Mathematical approach

The most basic form of frequency modulation is vibrato, in fact, FM is simply very fast vibrato. This possibility to create completely new sounds has been discovered by John Chowning at Stanford University in the mid-'60s. He found that when the frequency of the modulator increases beyond a certain point, the vibrato effect is replaced by a complex new tone.
We can create a vibrato sound by letting operator 2 cause a small change in frequency of operator 1. The rate and amount of frequency change is determined by operator 2. When the waveform of operator 2 increases the tone becomes brighter. This means that more harmonics are appearing in the frequency spectrum. Referring to the introduction section of this chapter, we remark again that FM is able to produce a wide variety of interesting spectra with only two basic oscillators.

Equations 2.7 and 2.8 are two basic sines. The instantaneous amplitude of the waves at any given point in time is called 'A' and are related to their gains 'a' (maximum amplitude of their cycle).

$$A_c(t) = a_c \sin(2\pi f_c t) \tag{2.7}$$

$$A_m(t) = a_m \sin(2\pi f_m t) \tag{2.8}$$

Equations 2.9 and 2.10 define the frequency modulation of the carrier wave $A_c(t)$ by the modulation wave $A_m(t)$.

$$\mathrm{A_c(t) = a_c \sin(\,(2\pi f_c + A_m)\,t)} \tag{2.9}$$

$$\mathrm{A_c(t) = a_c \sin(\,(2\pi f_c + a_m \sin(2\pi f_m t))\,t)} \tag{2.10}$$

Looking at equation 2.10 we see the modulated signal consists of a constant amplitude a_c and a time varrying argument to the sine function, or phase function $\phi(t)$. This phase function is the addition of a constant ramp with slope 2 πf_c and a sinusoidal variation, see picture 2.7. The

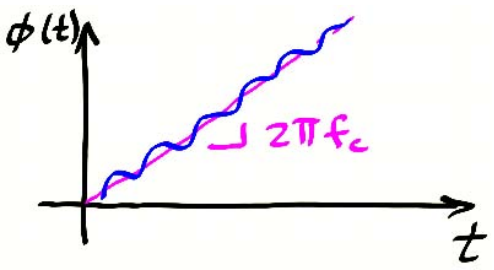

FIGURE 2.7: Phase function

phase function has two parameters that influence the nature of the modulated signal: a_m and f_m. The degree to which this variation is added to the constant ramp is given by the modulation index a_m or also referred to as I. So the modulation index describes the amount of frequency deviation. f_m is the modulation frequency and defines the rate at which the frequency deviation is to occur.
Now how does this influence the sound of a frequency modulated signal? First one must see that the side bands produced by FM lie around the carrier frequency, being plus or minus an integer multiple of the *modulator frequency* (the 1^{st} parameter of $\phi(t)$). This is expressed by equation 2.11 and visible in figure 2.8.

$$\mathrm{f_{sb} = 2\pi f_c \pm n \cdot 2\pi f_m} \tag{2.11}$$

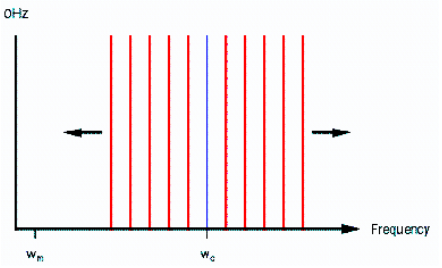

FIGURE 2.8: The position of the side bands

The other parameter of the phase function is the *amplitude of the modulator signal* and has a direct relation to the modulation index I. The modulation index is defined as the ratio of the change in carrier frequency to the modulator frequency:

$$\mathrm{I} = \frac{\Delta\omega_c}{\omega_m} \tag{2.12}$$

Obviously the change in carrier frequency is determined by the amplitude of the modulator signal a_m. For any given modulator frequency, it is the modulation index (and thus the amplitude of the modulator) that defines the energy of each of the components in the spectrum. Also the height of the carrier frequency can be influenced by the value of I.

In fact the amplitude of the sidebands is controlled by a Bessel function $J_k(\mathrm{a})$. Equation 2.10 can be rewritten as an infinite summation given by equation 2.13, in which θ corresponds to $2\pi f_c$, a corresponds to a_m (strictly spoken I) and β corresponds to $2\pi f_m$:

$$\sin(\theta + \mathrm{a}\sin\beta) = J_0(\mathrm{a})\sin\theta + \sum_{k=1}^{\infty} J_k(\mathrm{a})\left[\sin(\theta + k\beta) + (-1)^k \sin(\theta - k\beta)\right] \tag{2.13}$$

In this equation $\sin(\theta)$ corresponds to the component based on the carrier wave and the series of $\sin(\theta + \mathrm{k}\beta)$ are for the side band components. Note that the amplitude of all the components is controlled by the Bessel functions.
First we will show exactly how the frequency of the modulator signal defines the position of the side band components: in equation 2.13 we see "$\sin(\theta)$", this leads to the frequency component of the carrier f_c, see picture 2.9. If k equals one, the sine functions between brackets lead to frequency components of the modulator, $f_c + f_m$ and f_c - f_m. If k equals two components $f_c + 2f_m$ and $f_c + 2f_m$ are formed. And so on. The effect of the negative components will be ignored from now on[1]. Now we will have a look at how the Bessel functions control the amplitude of the

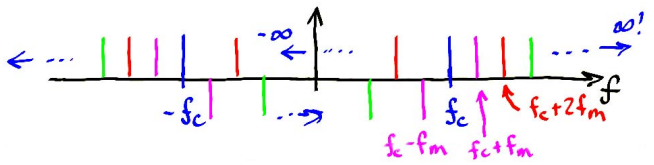

FIGURE 2.9: The position of the side bands

side bands. Equation 2.14 defines a Bessel function with order α.

$$x^2\frac{d^2y}{dx^2} + x\frac{dy}{dx} + (x^2 - \alpha^2)y = 0 \tag{2.14}$$

As defined in equation 2.13, the order of the Bessel function corresponds to the sideband number. So figure 2.10 shows 6 Bessel functions as a function of modulation index for a specific sideband number.
Note that, if the modulation index is zero (no modulation), the amplitude of the carrier frequency is equal to 1 (black curve). As we look at sidebands that are farther away from the carrier frequency we see that it takes longer for the amplitude of that sideband to become non-zero (red, blue,... curves). So one Bessel curve shows that the amplitude of the n^{th} sideband depends on the modulation index. Or, for a given modulation index, the amplitude of the n^{th} sideband is given by a Bessel function of order n.
When the modulation index of the signals in figure 2.8 is increased the spectrum will look like figure 2.11:
Appendix A contains a simple labview program to display Bessel functions of different orders and more examples that summarize this section.

2.3 Bandwidth considerations

The bandwidth of the signal can be defined as the range of frequencies occupied by the signal. Allthough the sum series of sidebands is theoretically infinite, the modulation index will make sure that sidebands of higher frequency are very small and negligible.
Equation 2.15 gives an approximation of the bandwidth of a FM signal.

$$B = 2\pi\,\mathrm{f_m}(1 + \mathrm{I}) \tag{2.15}$$

[1]Since we deal with an infinite summation, the negative components will kind of mix into the positive components. This can be thought of as a kind of aliasing phenomenon.

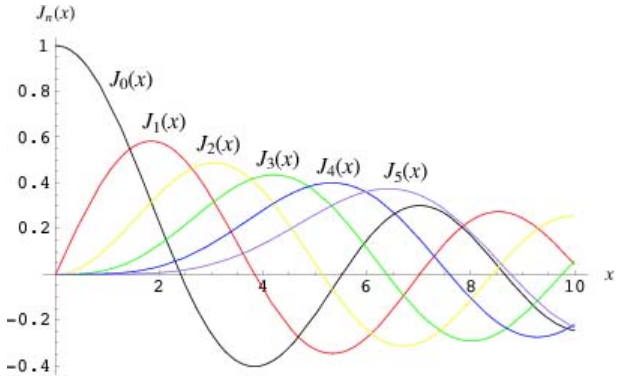

FIGURE 2.10: Bessel functions for orders n = 0,1,2, $\cdots$. The order defines the curve, x-axis defines the modulation index and the y-axis returns the amplitude.

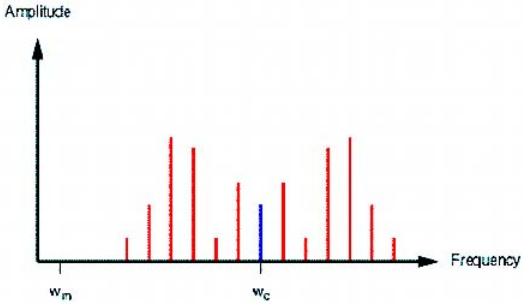

FIGURE 2.11: The amplitude of the sidebands.

For example: the bandwidth of a signal with a 300Hz modulator with I = 5 will be 2 × 300Hz × (1+ 5) = 3600Hz. This show again that with FM it is easy to create complex signals with a much higher bandwidth.

2.4 Practical implementation

Figure 2.12 shows the practical implementation of the FM theory in our synthesizer program. In this figure the first modulation mode is shown where operator 2 modulates operator 1 according to formula 2.10. The user can choose from this list of modulation types:

- 2 –> 1
- 3 –> 2 –> 1
- 4 –> 3 –> 2 –> 1
- 4–> 3 + 2 –> 1

For more information on the implementation of the FM synthesizer program please read chapters 5, 7 and 7.

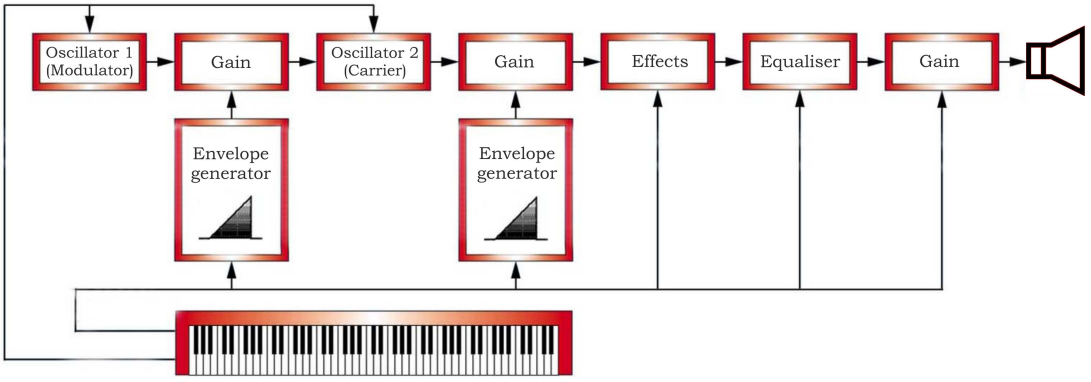

FIGURE 2.12: Functional design of the FM synthesizer program
(First modulation type from the list above
(Compare this figure with the program flow chart 6.7 discussed in chapter 6.)

2.5 Summary

We've showed that frequency modulation is a very powerful method for music synthesis. It is possible to generate sounds that are not obtainable by any other modulation technique.
The carrier frequency sets the center of the sideband cluster, the modulator frequency only sets the sideband positions.
It is the modulation index that determines the amplitude of the sidebands and the number of significant sidebands, doing so shaping the spectrum.

Chapter 3

ZigBee network: Waspmotes

3.1 Power considerations

3.1.1 Waspmote power modes

The libelium Waspmote has 4 operational modes: ON, Sleep, Deep Sleep and Hibernate. They differ from which type of interruptions they can be woken up and duration interval. For our application we want sleep intervals of 30 seconds and more, so only Deep Sleep and Hibernate mode are of interest. Table 3.1 summarizes the Waspmotes operational modes.

Mode	Consumption	CPU	Cycle	Accepted Interruptions
ON	9mA	ON	-	Synch and Asynch
Sleep	62μA	ON	31ms - 8s	Synch (WDT) and Asynch
Deep Sleep	62μA	ON	8s - min/hours/days	Synch (RTC) and Asynch
Hibernate	0.7μA	OFF	8s - min/hours/days	Synch (RTC)

TABLE 3.1: Operational modes of Libelium Waspmote V1.1

3.1.1.1 Deep Sleep

In deep sleep mode the main program is paused and the CPU passes to a latent state. Triggers are as well synchronous interruptions (RTC) as asynchronous interruptions. Examples of asynchronous interruptions are low battery level or a sensor that reaches a certain trigger value.

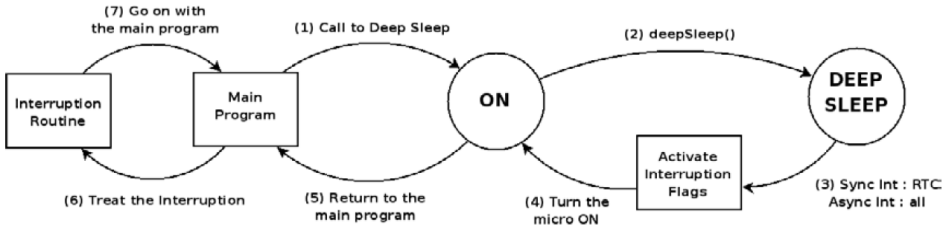

FIGURE 3.1: Waspmote going from ON to Deep Sleep

In figure 3.1 the process from going to operational mode ON to Deep Sleep is shown. The main

advantage of this mode is that the program is only paused, so the program stack and thus all variable values keep their values. When the Waspmote is turned back on it simply executes the next instruction.

3.1.1.2 Hibernate

Hibernate mode consumes roughly 100 times less energy than Deep Sleep. This is made possible by disconnecting all the Waspmote's modules, including the microcontroller. The RTC gets his power through the auxiliary battery. So if hibernate mode stops working it is probably necessary to replace the Waspmote's button battery. Figure 3.2 demonstrates the process from ON to hibernate.
This means the CPU is also switched of and does not remember any values from variables. When

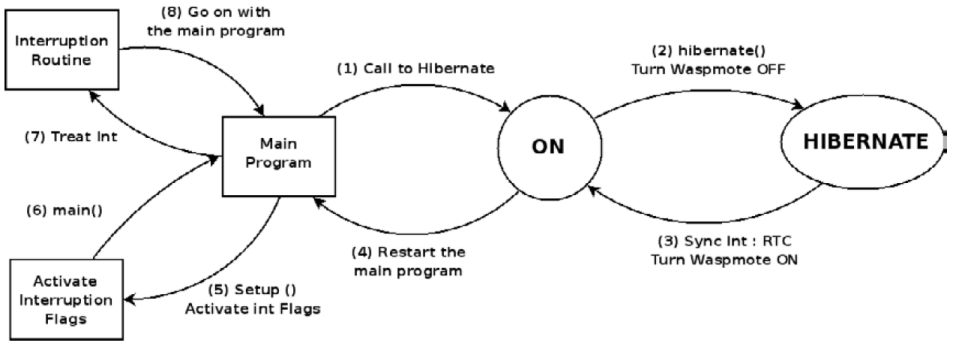

FIGURE 3.2: Waspmote going from ON to Hibernate

waking up the Waspmote reinitializes, the microprocessor is reset and the program restarts from the beginning. Both **setup** and **loop** routines are executed as if the main switch would be activated. By placing the `ifHibernate()` function in setup the program can determine if it came from a hardware reset or from a hibernate reset. To be able to wake up from hibernate mode the hibernate jumper must be removed correctly. See section .... for remarks on this issue. Because not all Libelium's API functions regarding hibernate in combination with the different alarm modes work, it is advised to use the functions provided in **WaspXBeeZBNode.h**.

3.1.2 Sampling sensors

To measure the sensors, originally we took 10 samples with 100 milliseconds recommended delay between the measurements and calculated the average. Since we want to make the energy consumption as low as possible we now do the iterations without delay. Appendix **??** contains 60 test samples per sensor and indicates that there is no significant difference on the average by removing this delay. Except for CO_2 measurements, removing the delay saves about 1000 milliseconds per measured sensor.
Because the first sample often shows a slight deviation, the program takes 11 samples but bases the average on the last 10 values.

3.1.3 Battery life estimation

In order to be able to give recommended sensor measuring intervals this section will analyse the estimated battery life of the Waspmotes. Table 3.5 enumerates the most common components typical consumption. The batteries included with our Waspmotes are rechargeable Lithium-ion batteries with a capacity of 6600mAH. The Waspmote that will be deployed on the roof of Group T, campus Vesalius will also have a 12V solar panel with a charging current up to 280 mA to

Action	Average Current
XBee, sending,	105mA
CO_2	50mA
XBee, ON	45mA
Waspmote, ON	9mA
Pressure	7mA
Humidity	380μA
Waspmote, sleep	62μA
Temperature	6μA
Waspmote, hibernate	0,007μA

TABLE 3.2: Operational modes of Libelium Waspmote V1.1

extend its battery life. The other batteries can by charged manually or by USB (5V, 100mA). Lithium-ion have a self-discharge rate of typcially 1 to 2 percent per month and since the used batteries are new we expect a high battery efficiency.

3.1.3.1 XBee and Waspmote start-up times

For the XBee node to join an existing network there are two power related possibilities. Either the Waspmote has been turned on already sufficiently long and the XBee had more than enough time to join the network, or either the XBee wasn't joined yet and the program needs to wait on this. From the experiments done at our apartment we came to following conclusions:

1. It takes about 2.5 seconds to join a network after powered on.
2. If the XBee is joined, the program still needs to confirm this. This takes 452 milliseconds.
3. The sending time is constant, about 158 ms, if the XBee had more than 2.5 seconds to join. However in case the XBee must send immediately after it is joined, the sending time is not constant and takes on average 611 milliseconds.
4. The sending time increases if there are more obstructions between the antennas.

With these characteristics we came to results discussed in the next sections. For the validation of these conclusions please see appendix A. Table 3.3 sums up the results of the distance-relation test.

Distance	Average sending time (ms)
Air	158
1 Floor	268
2 Floors	357
3 Floors	484
4 Floors	558
5 Floors	unreachable

TABLE 3.3: Distance consequence on send times

To save power the Waspmote can store the values for a user determined time. Taking samples and save them to EEPROM in case of hibernate mode takes only 6 - 7% of the time to measure and send. Table 3.4 confirms this.

Nr of samples per sensor	Average ON time (ms)
10	210
3	194

TABLE 3.4: Time needed to sample and store 4 sensors

3.1.3.2 Battery life with standard program optimizations

The application scenario for this battery test is as follow: the Waspmote will be turned on as short as possible and 4 sensors, namely temperature, humidity, pressure and battery level will be sampled. The node will take 10 samples for each sensor and calculate the average. Those values are put into one ZigBee packet and sent to the gateway. By adapting the sleep time between the event we came to the rather disappointing results shown in figure 3.3.
The graph in figure 3.4 breaks down the total energy consumption to five categories. It shows the monthly energy consumption as a function of the time between the events. For small intervals the active energy usage is huge. Only starting at 20 minutes sleep time the self-discharge becomes dominant and from 3 hours on the sleep mode current also becomes dominant.
Since the Waspmotes use this much energy when applied this way we will call this the High Performance mode from now on. The next section calculates an alternative approach, referred to as Power Saver mode. This nomenclature is continued in the program:

```
typedef enum {HIGHPERFORMANCE, POWERSAVER} PowerPlan;
```

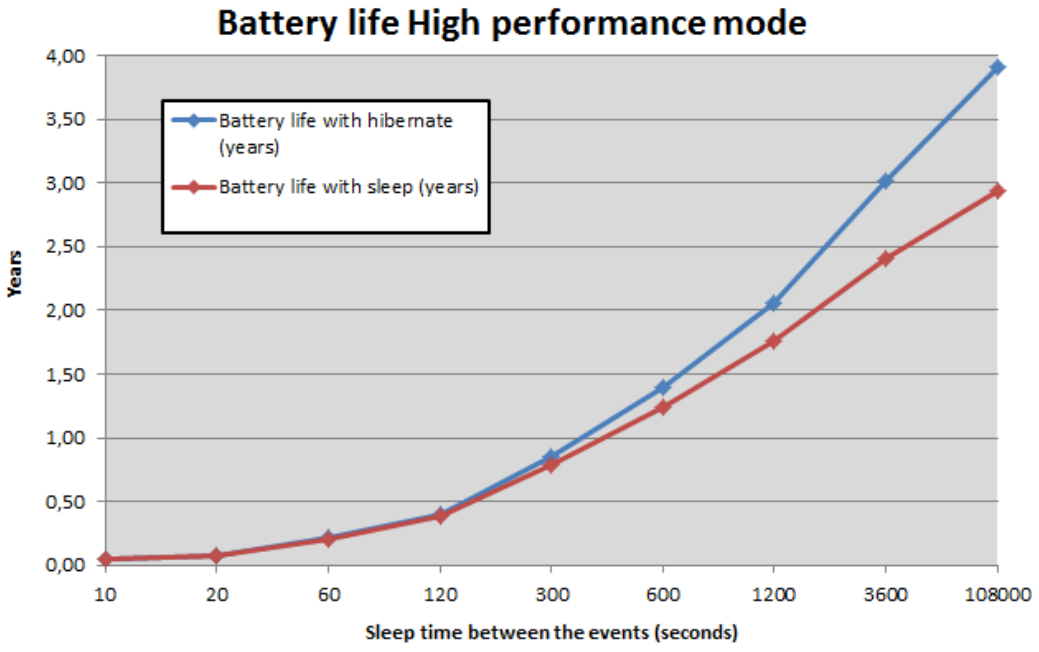

FIGURE 3.3: Battery life in High performance mode

Table 3.5 summarizes the battery duration in years of both performance and power saver mode.

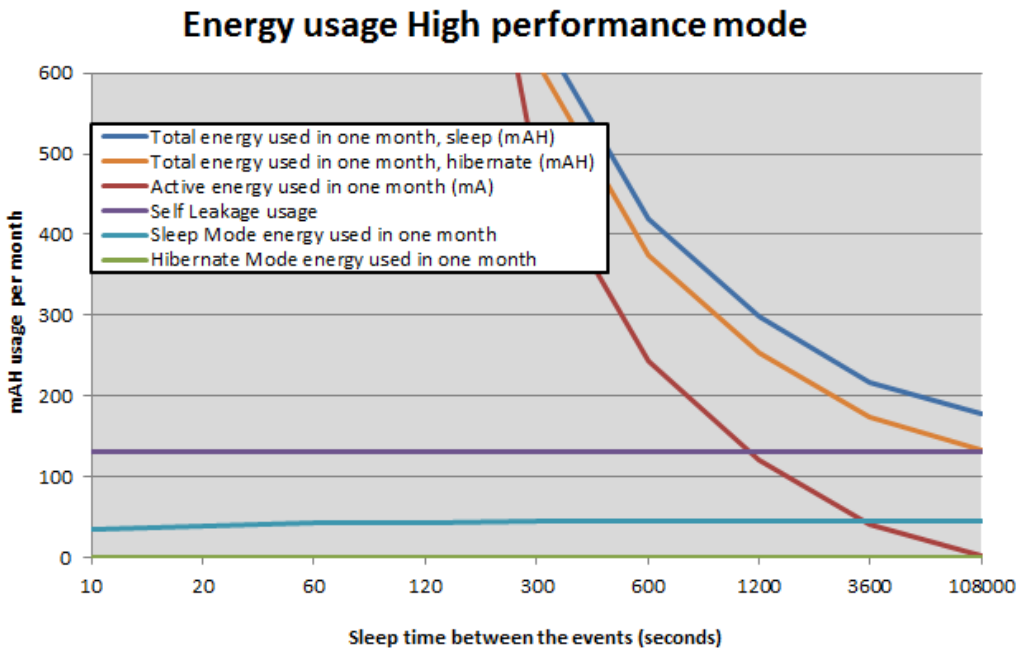

FIGURE 3.4: Energy usage in High performance mode

	Deep Sleep		Hibernate	
Sleep duration	High Performance	Power Saver	High Performance	Power Saver
10s	0,05	0,92	0,05	1,00
1min	0,21	2,15	0,21	2,63
3min	0,79	2,75	0,85	3,59
10min	1,25	2,85	1,39	3,76
20min	1,75	2,90	2,06	3,85
1h	2,41	2,94	3,02	3,91
3h	2,94	2,96	3,90	3,94

TABLE 3.5: Battery life in years for High Performance and Power Saver

3.1.3.3 Battery life with extra optimizations

As shown earlier in this section, sending values requires that the Waspmote is on for at least 3 seconds. In addition the XBee uses about five times the energy of the Waspmote. For end devices it is obviously recommended to turn on the XBee as little as possible, within a user defined limit.
Figure 3.5 shows the same results as the application scenario discussed in section 3.1.3.2 and adds the results for a mode further referred to as Power Saver.
The implementation of this mode will depend on the nodes sleep settings. For *Deep Sleep* the values can simply be stored on the heap, but for *Hibernate* the values must be written to EEPROM.
Because of the size limit of a ZigBee packet we can store maximum 30 values and send them in one packet. However, if the sensor measuring interval is small the user can opt to store more values and send two or more packets after each other. The values for Power Saver in table 3.5 are of an example scenario that takes 60 measurements and then sends them in two packets to the gateway. It are also those results which are put in function of time in figure 3.5.
As visible on figure 3.5 *Hibernate* has more influence in Power Saver mode, already extending battery life significantly at a 20 seconds interval comparing to a 10 minutes interval in High Performance mode. Also the energy breakdown graph in figure 3.6 shows that the interval times

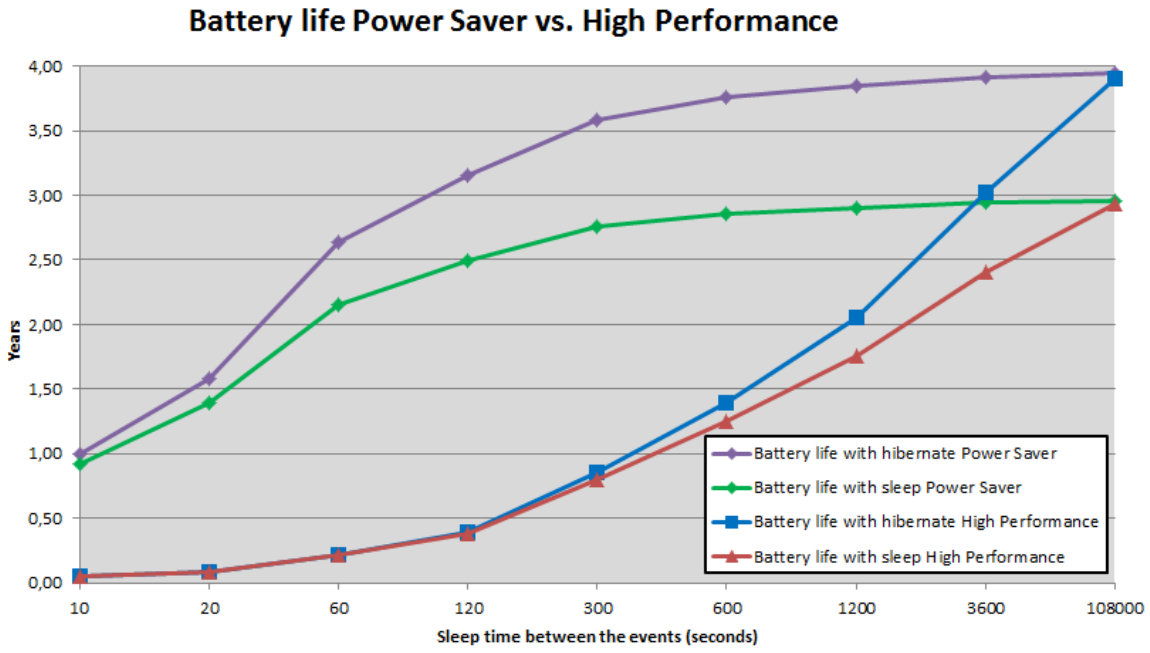

FIGURE 3.5: Battery life High Performance vs. Power Saver

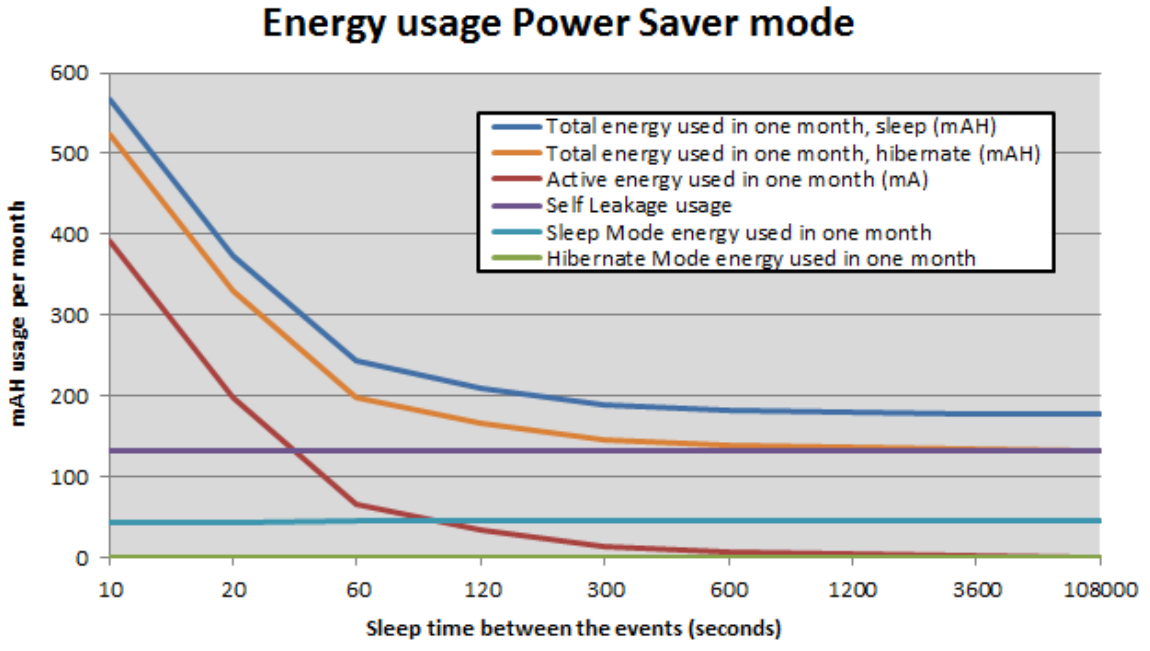

FIGURE 3.6: Energy usage in Power Saver mode

must be increased much less before the dominant factor is self-discharge and sleep mode energy consumption, compared to figure 3.4.
By reducing the sensor measurement accuracy battery life can be extended with modest 3 - 4%, best case scenario. Please see appendix **??** for details.
In case the measuring intervals are small it is recommended to use *Deep Sleep* instead of *Hibernate*, since in hibernate the values are written to EEPROM. Equation 3.1 shows this can be very destructive for the Waspmote. Depending on how much freedom the user is given, the program can make the decision to switch to *Deep Sleep* on itself, or the installation's administrator can control this.

3.1.4 Device start-up

Whether we come out of a hardware reset or a hibernate reset, the first thing the node will do is try to establish a connection with the network. Depending which reset we come from different routines will be executed. In normal operation mode this process only takes about 2 seconds (Please see appendix 3.7 for more measurement results). However, when a node is not able to join a network it will go into panic mode. This means that if the operating sleeping time is small, this time will be ignored and the node will wake up less frequently until it is able to rejoin the network, that way saving battery power. Supposing the node is able to rejoin the network, it will send the number of panics it experienced to the coordinator so a network administrator can investigate of the severity of the problem.

3.1.4.1 Full initialization

When the program is executed for the first time or when a hardware reset is detected the XBee will execute a full initialization process and the RTC will be set to zero. This means the default PAN ID and possibly other user settings like node ID will be written to the XBee. After this write the XBee must be re-setted (turn the power off and back on) and only than the joining attempts can start. This means full initialization takes about 9 seconds on average.

3.1.4.2 Reduced initialization

By not resetting the PAN ID but fetching it from the XBee's memory the joining process only takes about two seconds. Unfortunately a disadvantage of this shortened setup is that the XBee is no longer able to detect if the coordinator or his parent is actually available. The program will only notices this for the first time when it is trying to send a message. If this function results in a send error the program will do a full setup routine and resend the message. If the node then fails again to send the message we can conclude that the coordinator is really off-line or that there are no joinable nodes within range.

"Now, let's try to make a simple code which hibernates well. We recommend you to put a delay of a few seconds or a led blink before PWR.hibernate() sentence to allow removing hibernate jumper correctly."

3.2 Power savings

3.2.1 In the algorithm

3.2.2 Sleep vs. Hibernate

Compare battery life

3.2.3 Variable sleep times

In default configuration the Waspmote will send only its battery level to the default gateway and go into hibernate mode for 15 minutes. After this the cycle repeats. However it is possible that the node has more sensors implemented and the user wishes to obtain these values at different frequencies. In that case the node will have various different sleep times.
The next subsections will discuss different techniques to determine those sleeping intervals. In hibernation mode the node is completely disconnected from the main battery and the program stops. This makes that all variables lose their values and must be stored in EEPROM memory if they must be known during the next cycle. Each of the next techniques present with benefits and drawbacks and since we are working with embedded systems with limited possibilities, one should also consider to limit the users options to facilitate the calculations.

3.2.3.1 Calculate only the next time to sleep

Each algorithm will have to store the individual sleep times per sensor. To support this algorithm also a copy of the original time will be saved and each time the node wakes up it will look for the smallest next time to sleep. This number will be subtracted from the other sleep times in the array. When a value becomes zero it will be restored with its original value and the cycle continues. The following example demonstrates the process:
This process is fast and simple. However, the main advantage is that the node has to write to

Sensor[4]	**Sensor[3]**	**Sensor[2]**	**Sensor[1]**	**Sensor[0]**
100	50	35	10	20

Table 3.6: Individual Sensor Sleep Times in seconds

Cycle	**Sensor[4]**	**Sensor[3]**	**Sensor[2]**	**Sensor[1]**	**Sensor[0]**	**Sleep time**
0	100	50	35	**10**	20	10
1	90	40	25	**10**	**10**	10
2	80	30	15	**10**	20	10
3	70	20	**5**	10	10	5
4	65	15	25	**5**	**5**	5
5	60	**10**	20	**10**	20	10
6	50	50	**10**	**10**	**10**	10

Table 3.7: Example of sleep algorithm 1

EEPROM each time it wakes up. According to Atmel the EEPROM of the ATmega1281 has an endurance of at least 100,000 write / erase cycles. The following equation indicates the problem for an interval of 10 seconds:

$$\frac{100000\text{writes} \cdot 10\text{s}}{60\text{s} \cdot 60\text{min} \cdot 24\text{h}} = 11,57 \text{ days} \tag{3.1}$$

But the processors has 4Kbytes EEPROM on board so we don't have to write to the same place every time. Since EEPROM is written on a 'per cell' basis this can extend the lifetime. Our sensor mask can contain up to 16 values of 2 bytes. This leads to the next result:

$$\frac{100000\text{writes} \cdot 10\text{s} \cdot 4\text{KB}}{60\text{s} \cdot 60\text{min} \cdot 24\text{h} \cdot 365\text{days} \cdot 32\text{B}} = 3,96 \text{ years} \tag{3.2}$$

We still must store where the data is stored but this won't cause big problems since we only have to rewrite this cell 125 times:

$$\frac{4\text{KB}}{32\text{B}} = 125 \text{ writes} \tag{3.3}$$

3.2.3.2 Calculate all next times to sleep

Another possibility is calculate as much as possible or if maybe even all sleep times.

3.2.3.3 Limit user control

When the synthesizer generates a sound, then the frequency of that signal determines how the signal sounds. The frequency, however, does not affect the loudness of the sound. Loudness is determined by the amplitude. At the start, when a sound gets generated or at the end, when the sound dies out, the amplitude changes. The way the amplitude changes during the life cycle of the sound is the envelope.
Figure 3.7 gives an example of the envelope of a signal. It is divided into 3 sections. The first section is called "attack". It is the rise of the amplitude at the start of the signal generation. The amplitude is at its maximum at the end of the attack. The time the total attack takes is predetermined but if the generation of the signal gets disrupted before the attack time is expired, then the envelope goes directly to the last(third) section, called "release".
The release is the section where the signal dies out. The amplitude will decrease until it finally becomes 0. The release time is just like the attack time predetermined.
The second section is called "decay". In this section of the envelope the signal's amplitude decreases from the final amplitude of the attack to a lower one. The decay time is also predetermined.

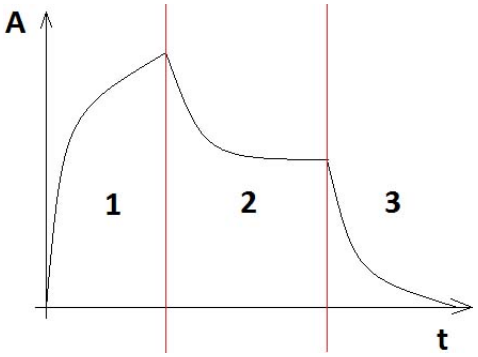

FIGURE 3.7: Example of envelope

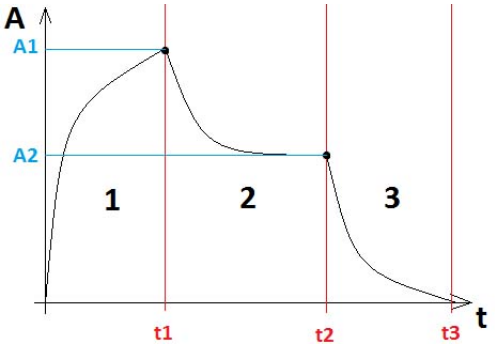

FIGURE 3.8: Envelope with the parameters for the equations

There is also a fourth section but it is not shown in figure 3.7. This section is called "sustain". It is actually the section between the decay and the release. The reason it is not shown is because sustain time is not predetermined. The sustain lasts until, in the case of a synthesizer , a button is released after the attack and decay. Then the envelope passes into the release. The amplitude of the sustain is constant and is equal to the final amplitude of the decay.

3.2.4 Mathematical equations

The attack, decay and release all have a different but similar equation, respectively equations 3.1, **??** and **??**:

$$A = \frac{A_1}{t_1^{slope_1}} \cdot t^{slope_1} \tag{3.4}$$

$$A = \frac{A_2 - A_1}{(t_2 - t_1)^{slope_2}} \cdot (t - t_1)^{slope_2} + A_1 \tag{3.5}$$

$$A = \frac{-A_2}{(t_3 - t_2)^{slope_3}} \cdot (t - t_2)^{slope_3} + A_2 \tag{3.6}$$

The variables A1, A2, t1, t2 and t3 are represented in figure 3.8. Slope 1, slope2 and slope 3 are the slopes of section 1, 2 and 3 respectively.
Figure 3.9 shows examples of the attack with different slopes. The straight line in the middle is an attack with a slope of 1. If slope becomes larger than 1, the attack will rise first at a slow rate and then faster in an exponential way. The larger the slope, the faster the attack will rise at the end. If the slope is larger than 0 but smaller than 1, the attack will first rise fast and then slower in an exponential way.

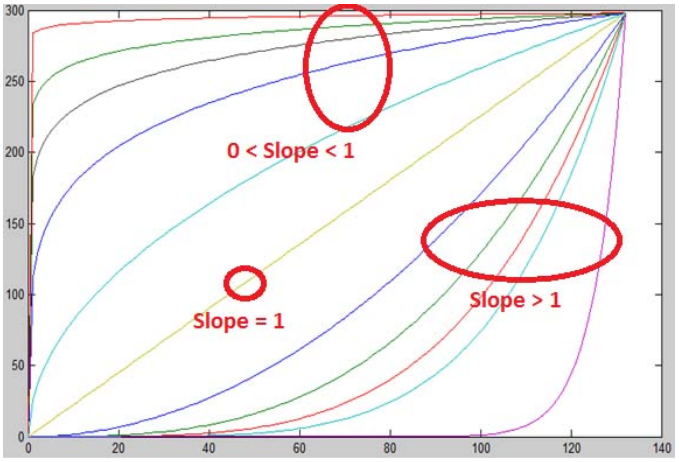

FIGURE 3.9: Different slopes for the attack

3.2.5 Envelopes and modulation

When modulating a signal, only the carrier is hearable (Bjorn does not agree with this!). With an envelope you can make a sound for example fade in or out. The modulators only change the original (carrier) signal but the modulators aren't directly hearable.
The modulators, however, do have an envelope as well. We know that when we modulate a signal, the modulation is also dependent on the modulation index "I". This value is an indication for how hard the original signal gets modulated. The envelope of the modulators change the value "I". In this way a modulation is affected by the envelope.

3.2.6 Envelopes in the program

Each operator has a variable" envEn"(Bool), "time"(int) and "count"(int). The parameter envEn is a boolean and it is used to switch the envelopes on or off. While modulating (Bjorn: also without modulation), the program checks if envEn is true. If this is the case time increases by 1. If this is not the case time and count each become 0. If envEn is true and time is equal to

the sampling frequency (96000Hz), count increases by one. This is to indicate that a period of 1 second has elapsed.
Below is the algorithm for the envelopes shown. The variable "totTime" is the total time of the envelope. The program checks if the envelopes are enabled. If this is true, it checks if the totTime <= endPointATKt, > endPointATKt and <= endPointDECt or > endPointRELt. This is a way for the program to know in which phase of the envelope totTime is located (attack, decay or release). If totTime is greater than endPointRELt, the variable envEn becomes false so the envelope will repeat itself as long as the user doesn't turn off the envelopes. If the envelopes are not enabled, the amplitude of the wave is constant and equal to the amplitude the user chose on the interface.

```
for(i=0; i<4; i++)

   (op[i].envEn == TRUE) ? (op[i].env.time)++ : (op[i].env.time = op[i].env.count = 0);
   if(op[i].envEn == TRUE && !(op[i].env.time%=sampleFreq)) op[i].env.count++;


if(op->envEn)

   totTime = op->env.time + op->env.count * sampleFreq;

   if(totTime <= endPointATKt)
      result=(op->amplitude/(powf(endPointATKt,slope1)))*powf(totTime,slope1);
   else if(totTime > endPointATKt && totTime <= endPointDECt)
      result=((((endPointDECa - op->amplitude)/powf((endPointDECt - endPointATKt),slope2))
             *powf((totTime - endPointATKt),slope2)) + op->amplitude);
   else if(totTime > endPointDECt && totTime <= endPointRELt)
      result=(((-endPointDECa/powf((endPointRELt-endPointDECt),slope3))*powf((totTime
             - endPointDECt),slope3)) + endPointDECa);
   else if(totTime > endPointRELt)
      op->envEn = FALSE;

else  result = op->amplitude;

return result;
```

3.2.7 Problems that occurred during the course of the project

There were a lot of problems with the envelopes. At first there was only one variable time for all the operators. This way the envelopes worked too but it was confusing to program with and the envelope would not start from the beginning. It was also not possible to make the envelopes repeat properly without individual timers for each operator (Bjorn: not true!).
Another problem was the variable count. At first count would not increase of each operator individually. The counts of all enabled envelopes would increase at the same time when time%=96000 would equal to 0. The problem that would occur was that count would only increase every 4 seconds instead of 1. Every time a block of 1024 samples is processed, time increases by 1024. This, however, would cause time%=96000 to be only 0 when time reached 384000 which is equal to 96000*4.
To solve this problem we would let the counts of the enabled envelopes increase in the function "createSignal" itself. The function `updateEnvelopes()` increases the counts of all enabled envelopes. Now count would increase properly every second. Another problem occurred when modulating. If we for example modulate with 2 operators. If time would become 96000, count of operator 1 and operator 2 would both increase. If the envelope of the first operator would then be calculated there would be no problem. The problem would occur with the second operator. Count of this operator would have increased at the same time as count of the first operator.

This would cause the envelope of the second operator to skip samples because totTime of that envelope would not have the right value. This problem could be solved by making the counts of the enabled envelopes increase individually.

```
void createSignal(Operator *op, int t)

   int i;
   float scalar;
   scalar=powf(2.0,23.0);
   switch(op->wave)

   case SINE:
      for(i=0; i<NUM_SAMPLES; i++)

            op->signal[i] = envelope(op, &t)*sin( 2.0 * 3.141592 *
                                          op->frequency / sampleFreq * t );
            t+= 1;
            if(!(t%=sampleFreq)) updateEnvelopes(); //nrT++
            t%=sampleFreq;

      break;
```

Remark 3.1. Please see the section "Key functions" 6.4.2 for the correct version of this function and without unused variables and double operations.

3.3 Keyboard control

The synthesizer must be able to play different music notes with the press of a key on the keyboard like a piano.
Figure 3.10 shows the labview algorithm for keyboard control of the synthesizer. This sample algorithm checks if "B" on the keyboard is pressed. In short these are the steps the program goes through to accomplish this:

1. The algorithm first checks if the Boolean to switch on the keyboard is true. If this is the case, the algorithm goes to step 2 (this step is not shown on figure 3.11).
2. The pressed key gets registered by the program. On figure 3.11 the keyboard is shown with all the buttons.
3. The name of the pressed key is stored in an array and the pressed keys on the keyboard interface get a darker color as long as the keys are pressed. Labview assumes you are using a QWERTY keyboard so in figure 2 the requirement is "Z" but on an AZERTY keyboard this is "W". The user doesn't have to know this because he/she only has to look at the interface of the keyboard.
4. The case structure checks if the name of the key is equal to "Z" in this case.
5. If the key is pressed: The associated key on the interface becomes darker and the frequency of the sound that gets created becomes 31Hz in this case. The frequencies of the other buttons are $2^{\frac{1}{12}}$ times the frequency of the previous one. For example, the frequency of "S" is $2^{\frac{1}{12}}$ times the frequency of "W" and the frequency of "X" is $2^{\frac{1}{12}}$ times the frequency of "S". If no key is pressed: frequency of the sound automatically becomes 0.
6. The while loop repeats.

We made 2 octaves of buttons for the interface which means that every button in the second octave has a frequency that is twice the frequency of the corresponding button on the first octave.

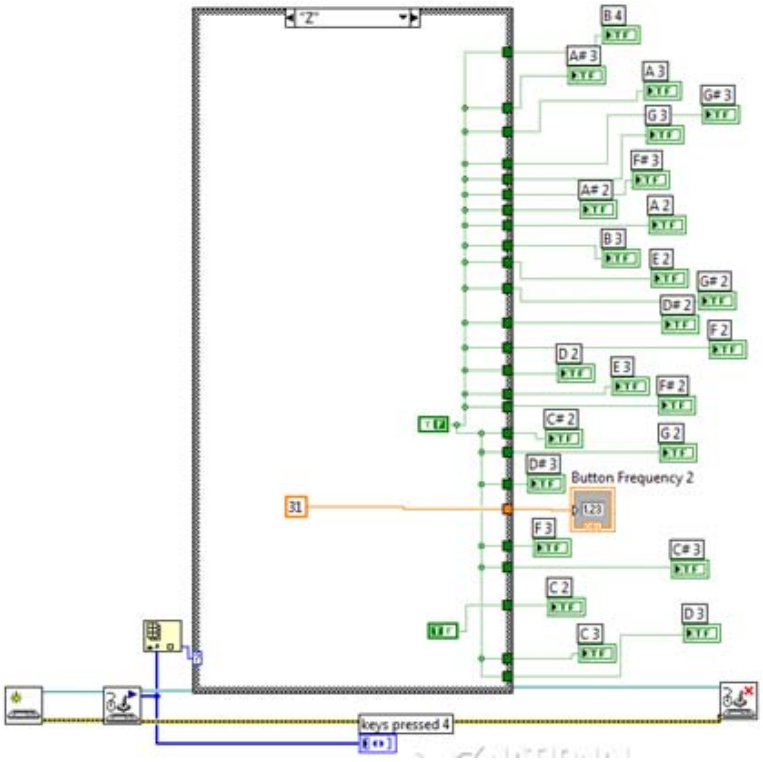

FIGURE 3.10: Labview algorithm for keyboard control

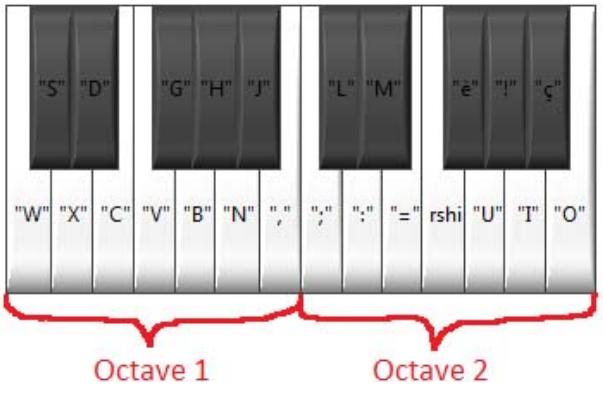

FIGURE 3.11: Keyboard buttons

Chapter 4

Effects and equaliser

Written by Frederik De Greef

4.1 Overdrive

4.1.1 Overdrive function

The overdrive effect amplifies the audio signal in a non-linear way. There are different possible schemes, we used the three layer non-linear soft saturation scheme below. The treshold value is set at 1/3, given that the input audio values are between -1 and 1.

$$f(x) = \begin{cases} 2x & \text{if } 0 \leq x < \frac{1}{3} \\ \frac{3-(2-3x)^2}{3} & \text{if } \frac{1}{3} \leq x < \frac{2}{3} \\ 1 & \text{if } \frac{2}{3} \leq x \leq 1 \end{cases}$$

- In the lower third the output is linear - multiplied by 2. for $\frac{2}{3} \leq x \leq 1$
- In the middle third there is a non-linear (quadratic) output response
- Above $\frac{2}{3}$ the output is set to 1.

Image 4.1 below shows the effect of the overdrive on an audio sample shown in the time domain. The blue signal is the original, the red signal is overdriven signal. (Time on the x-axis, amplitude on the y-axis.) Bringing this non-linearity in the signal is called clipping. Clipping is a process that produces frequencies not originally present in the audio signal. These frequencies can either be "harmonic", meaning they are whole number multiples of the signal's original frequencies, or "inharmonic", meaning dissonant odd-order overtones.
Setting the treshold value also determines wich type of clipping is applied. Increasing the treshold value will result in a softer clipping. Whereas decreasing the treshold value wil result in a harder clipping. When a treshold value of 1 is defined, there is no overdrive applied. The output remains the original signal.

The hardness or softness of the clipping matters. Hard clipping results when the output wave equals the input up/down to a certain level, then stays at the clipping level until the input drops below the clipping level again, giving perfectly flat tops and bottoms to the clipped output. Soft clipping has no abrupt clipping level, but gently rounds the top/bottom of the output wave so the waveform is "softly" rounded on top/bottom, not flat-topped.
Soft clipping emphasizes the lower- order harmonics, the third and fifth, etc. Hard clipping has a mix slewed to the higher order seventh and up harmonics, which are harsher sounding.

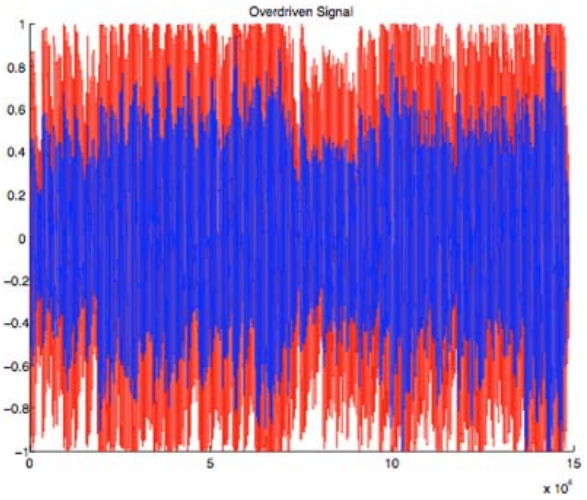

FIGURE 4.1: Effect of overdrive in time domain

In the figure 4.2, a normal sinewave with amplitude 1 is shown after beeing processed by the overdrive effect with a threshold equal to 1. With a treshold of 0.5 there is a small soft clipping visible, see figure 4.3. Both images have time on the x-axis and amplitude on the y-axis.

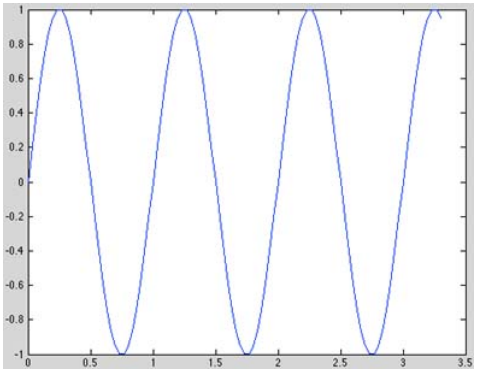

FIGURE 4.2: Overdrive threshold 1

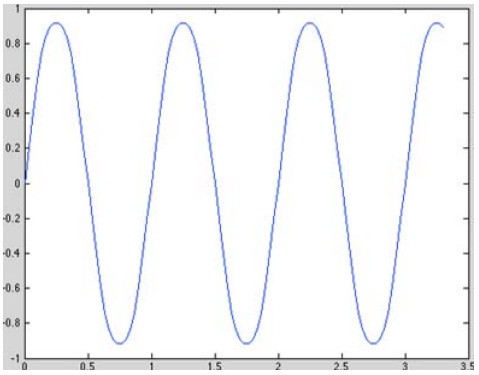

FIGURE 4.3: Overdrive threshold 0.5

4.1.1.1 Influence of the overdrive effect on the frequency spectrum

To test the effects of the overdrive effect on signals, a few examples are fed into a Fast Fourier Transform (FFT) application. We test the FFT with the following signal:

$$y(t) = 0.7\sin(2\pi \cdot 50t) + \sin(2\pi \cdot 120t) \tag{4.1}$$

This signal combines a sinewave of frequency 50 Hz and amplitude 0.7 with a sinewave of frequency 120 Hz and amplitude 1. When the FFT is finished and plotted,figure 4.4 below is the result. The graph shown in figure 4.5 the FFT of an overdrive with treshold equal to 0.1 on a sinewave with frequency 50hz. We can clearly see that there are only odd harmonics in the frequency spectrum, this is because the overdrive effect was set to symmetrical clipping.

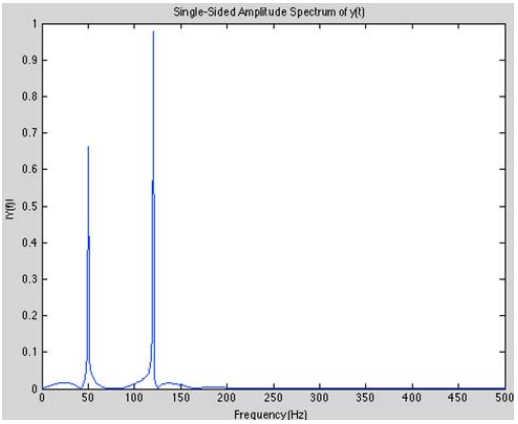

FIGURE 4.4: FFT of a basic signal

4.1.1.2 Symmetrical clipping

For a given input waveform, say a sine wave, the tops and bottoms of the waveform are clipped equally, symmetrically. For a simple sine wave, symmetrical clipping generates only odd-order harmonics, giving a reedy, or raspy sound to the resultant waveform.

4.1.1.3 Asymmetrical clipping

The top(or bottom) of the waveform is clipped more than the bottom (top) half. This causes the generation of both even and odd harmonics, in contrast to symmetrical clipping's odd-order only. The even harmonics are smoother and more musical sounding, not as harsh as the odd ones. The hardness of the clipping and the degree of asymmetry affect the sound. The more asymmetrical, the more pronounced the even-order harmonics, the harsher the clipping, the more the harmonics are slewed toward higher orders.
The graph in figure 4.6 shows the Fourier transform of a the same signal, this time overdriven using asymmetrical clipping. Here the part of the sinewave above the x-axis was clipped using the same function as in the symmetrical clipping. The part of the sinewave below the x-axis is still the original. As we can see in the graph, using the asymmetrical clipping generates not only the odd but also the even harmonics. These even harmonics give an other type of sound.

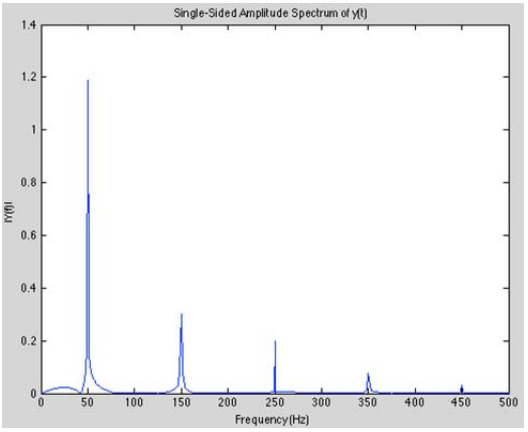

FIGURE 4.5: Symmetric clipping

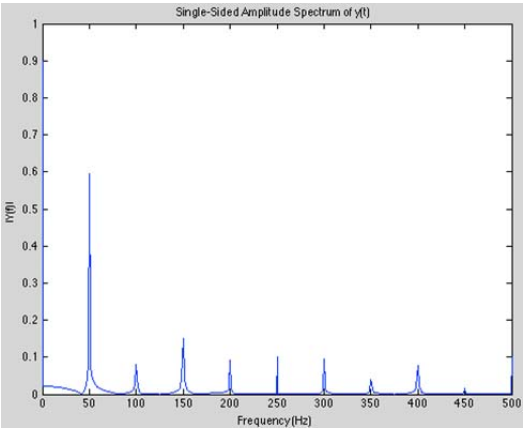

FIGURE 4.6: Asymmetric clipping

4.2 Equaliser

4.2.1 Design of the equaliser

The equaliser consists of three filters:

- Lowpass filter with cutoff frequency equal to 800Hz.
- Bandpass filter with cutoff frequencies equal to 800Hz and 6000Hz.
- Highpass filter with cutoff frequency equal to 6000Hz.

All the filters are constructed as FIR (Finite impulse response) with order 100. The filter components are created in MATLAB using the fir1 command.
The architecture of these Finite impulse response filters is displayed in figure 4.7.

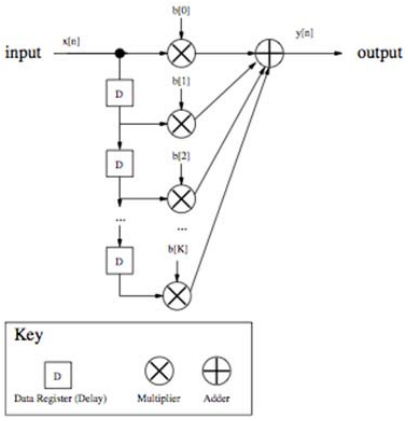

FIGURE 4.7: Architecture of the equaliser

4.2.2 Amplitude response of the FIR filters

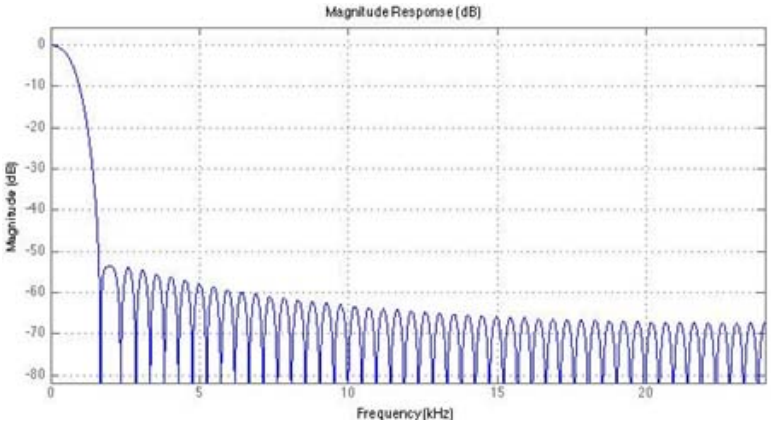

FIGURE 4.8: Magnitude response lowpass filter

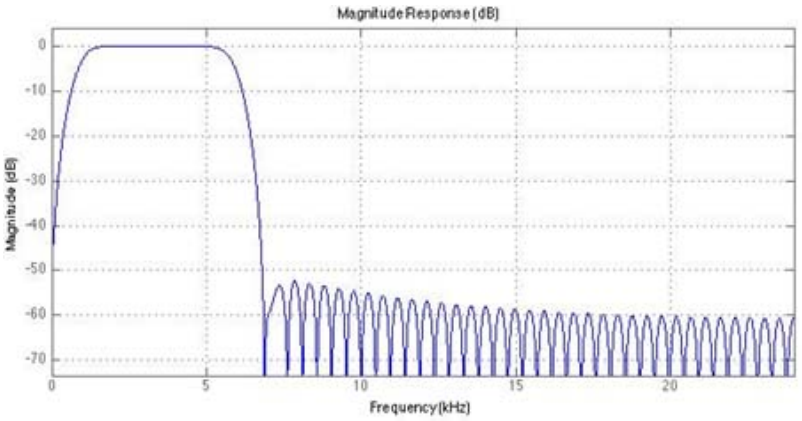

FIGURE 4.9: Magnitude response bandpass filter

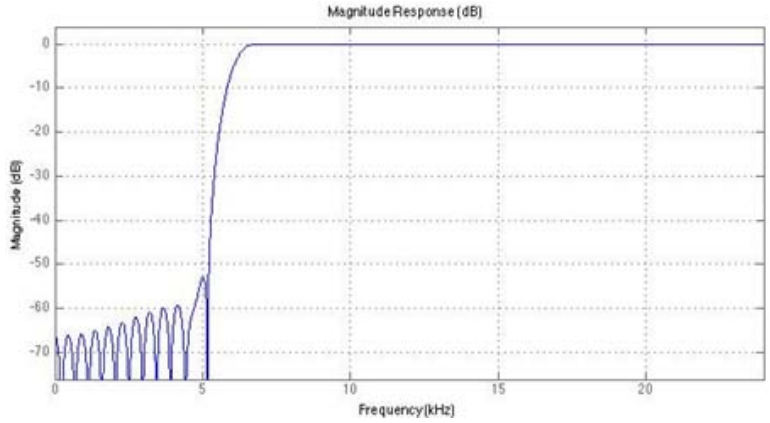

FIGURE 4.10: Magnitude response highpass filter

4.2.3 Testing of the equalizer in MATLAB

The equalizer has been tested by processing blocks of 1024 samples filled with white gaussian noise, the result has been fed into an FFT. Blocks are processed with the equalizer settings displayed in the caption of figures 4.11, 4.12 and 4.13.

The equaliser settings can be changed in the interface, using the sliders for all the filters where a value between 0 and 10 can be set. The interface will respond by sending separate gain integers to the dsp board. Here every filter is independently multiplied by it's gain and devided by 10. Afterwards all the filter coefficients are added together to deliver the 101 final coefficients for the selected equaliser setting. These values will than be used to process the sample blocks entering the equaliser.
This is not the traditional way of FIR filtering, normally different coefficients are created for every specific equalizer setting. This way of working saves lots of processing power.

Because the FIR filters work with samples 'from the past', every sample block will first be prepadded with the 100 last samples of the previous block. These are stored in the so called prepad array. After the incomming block is prepadded, it 100 last samples will be stored in the prepad array. In this way always 1024 samples are loaded into the filer, this filter uses 1124 samples to create a 1024 sample outputblock.

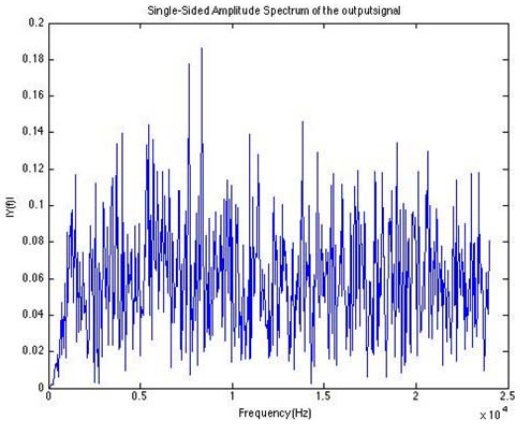

FIGURE 4.11: $\text{Gain}_{LP} = 0$, $\text{Gain}_{BP} = 1$, $\text{Gain}_{HP} = 1$

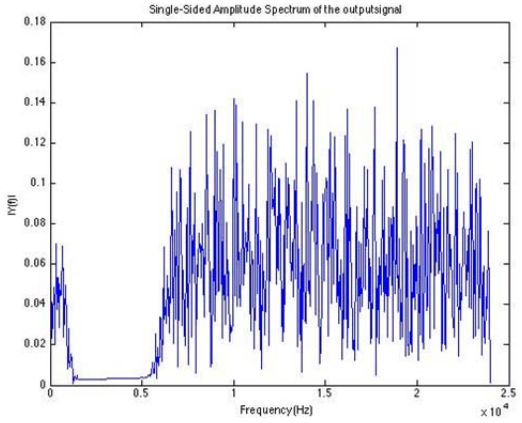

FIGURE 4.12: $\text{Gain}_{LP} = 1$, $\text{Gain}_{BP} = 0$, $\text{Gain}_{HP} = 1$

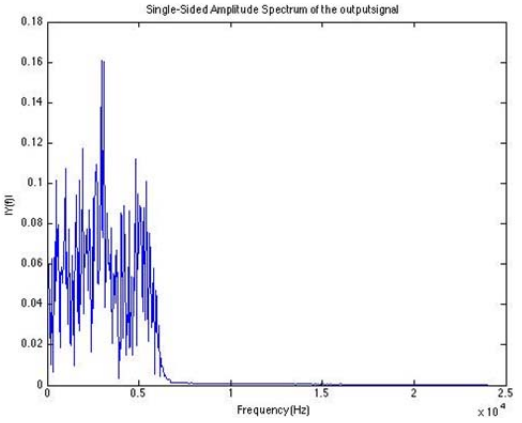

FIGURE 4.13: $\text{Gain}_{LP} = 1$, $\text{Gain}_{BP} = 1$, $\text{Gain}_{HP} = 0$

4.2.4 Problems and solutions

- **Problem:** The overdrive effect can only generate odd harmonics.
  - **Solution:** An extra option is added, which lets the user choose between 'symclip' (odd harmonics) or 'asymclip' (even harmonics).
- **Problem:** The interface was not able to generate specific filter coefficients for all the different equalizer settings.
  - **Solution:** Solved by designing a filtering method which can use 'static' coefficients. Here the interface only has to send the different gain integers to the dsp board.
- **Problem:** The FIR filter goes back 100 samples in the past to proces the current sample.
  - **Solution:** An extra 'prepad' array is added, initiated by zeros when the equalizer starts, and always saves the last 100 samples of the 1024 sample block. Allowing the filter to use these samples.
- **Problem:** The filter coefficients for the highpass filter are to large for the 32-bit double values, the dsp board uses.
  - **Solution:** We first changed the compiler settings to use 64-bit memory for the values, but this caused the rest of the program to stop working. It was finally solved by creating new filter coefficients for the high pass filter who where small enough to fit in 32 bits.

- **Problem:** The equalizer added an extra noise to the signal
  - **Solution:** The problem here was that the prepad array was not filled correctly, so it prepadded every sample block with 100 zeros as it was initiated. The last 100 samples where always saved in a wrong place in the memory. The use of pointers for this storing and fetching caused the program not to crash during runtime. Correcting the save location solved this.
- **Problem:** The bandpass filter acts as a highpass from its first cutoff frequency
  - **Solution:** at implement

Chapter 5

Digital Signal Processor

Written by Bjorn Deraeve

5.1 Introduction

5.1.1 What is digital signal processing?

Signal processing falls within the scope of electrical engineering and applied mathematics that deals with the analysis of signals or operations on signals. In order to do this signals are presented in descrete time, discrete frequency or other discrete signal domains. The set of algorithmic solutions that are used to process digital signals are limited by the processing capabilities of the available hardware. Digital Signal Processing (DSP) algorithms have long been run on standard computers. Today additional technologies such as specialized processors called Digital Signal Processors (DSP) are used. DSP processors can perform typical DSP operations more efficiently thanks to single-cycle multiply-accumulate (MAC) units, shift registers and extra large accumulators. To increase speed and possible DSP realizations DSP processors are used in purpose-built hardware such as application-specific integrated circuits (ASICs) or field-programmable gate arrays (FPGAs).

5.1.2 How are DSP processors used?

DSPs receive real-world signals like voice, audio, video, ... that have been dididtized and then mathematically manipulate them. DSP processors are a special type of microprocessors which are designed for performing mathematical functions like "add", "substract", "multiply" and "divide" very quickly. Real-world analog signals are converted by Analog-to-Digital converters and are turned into the digital format of 0s and 1s. Now the DSP can start working on this digitized information. When all processing is done the new samples are mostly feed back in the real world via Digital-to-Analog converters. This whole process happens at very high speeds.

For our FM synthesizer application the first step is different. The DSP processing algorithms consist of algorithms that create the signal and signals that manipulate those signals so the AD converters on our FPGA are not used.

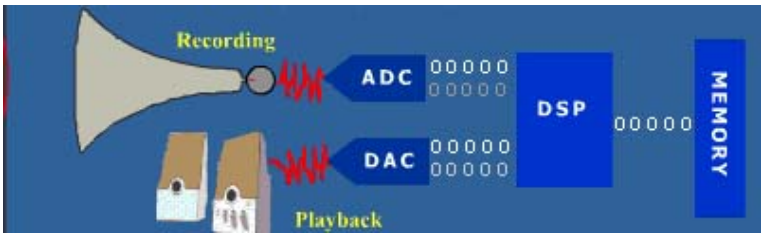

FIGURE 5.1: DSP Principle

5.2 Digitization

Real-world continuous signals have to be reduced to discrete signals (a numeric sequence) before a DSP can manipulate them. This sampling is done by an analog-to-digital converter (ADC) and the resulting samples are transported serially to the DSP.

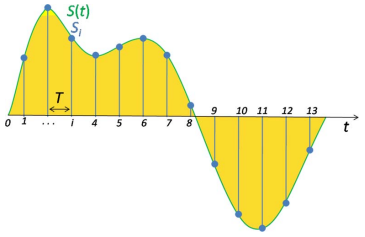

FIGURE 5.2: Signal sampling representation

The sampling frequency f_s is the number of samples taken in one second. According to the Nyquist criterion, the sampling frequency must at least be twice the frequency of the highest frequency component of interest. If the sampling frequency is greater than or equal to twice the bandwidth of a bandlimited signal then the signal can be (re)constructed without aliasing. This frequency is called the Nyquist rate.

5.3 Digital Signal Processors

As briefly mentioned in this chapter's introduction DSPs are processors with hardware, software, instruction sets, parallelism and data addressing that are optimized for high-speed numeric operations. A DSP contains the following key components:

- **Program Memory:** here the processing algorithms are stored
- **Data Memory:** stores the signals to be processed
- **Compute Engine:** performs the mathematical operations, coordinates access to program and data memory
- **Input/Output:** connections to the outside world

The need for high speed processors is because most DSP calculations are done on real-time signals. The inputsignal comes to the DSP as a train of individual samples from the AD converter. To do for example filtering in real-time, all calculations and operations on a sample must be completed before the next sample arrives. Usually also previous samples are needed so these must be stored somewhere and have to be quickly accesible.

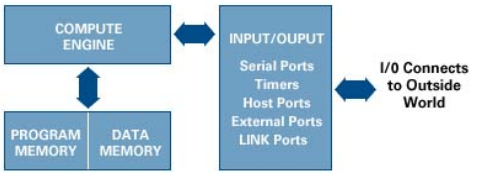

FIGURE 5.3: DSP structure

5.3.1 Numeric architecture

We know that DSPs must complete multiply-accumulate, additions and bit-shift operations in a single instruction cycle. Hardware optimized for such numeric operations is typical to DSP processors and distinguishes them from other general-purpose microprocessors. The numeric operations are done by the DSP's MAC units, the arithmetic-logic unit (ALU) and barrel shifters:

- **MAC:** performs sum-of-products operations (used by FIR and IIR filters and fast Fourier transforms). The multiply-accumulate operation computes the product of two numbers and adds it to an accumulater:

$$a \leftarrow a + (b \times c) \tag{5.1}$$

  The multiplier is typically implemented in combinatonial logic followed by an adder and finally the accumulator register to store the result. If the output of the register is fed back to the adder each clock cycle the output of the multiplier is added to the register. Calculating multiplications with combinatorial logic requires a large amount of logic but is much faster than the method that uses shifts and additions.

- **ALU:** performs standard addition, substraction and basic logical operations

- **Barrel shifter:** performs operations on bits and words. This digital circuitry can shift a word by a specified number of bits in one clock cycle and is implemented as a sequence of multiplexers where the output of one mux is the input of the next mux. The input depends on the shift distance. An example of an 8-bit barrel shifter is shown in Figure 5.4.

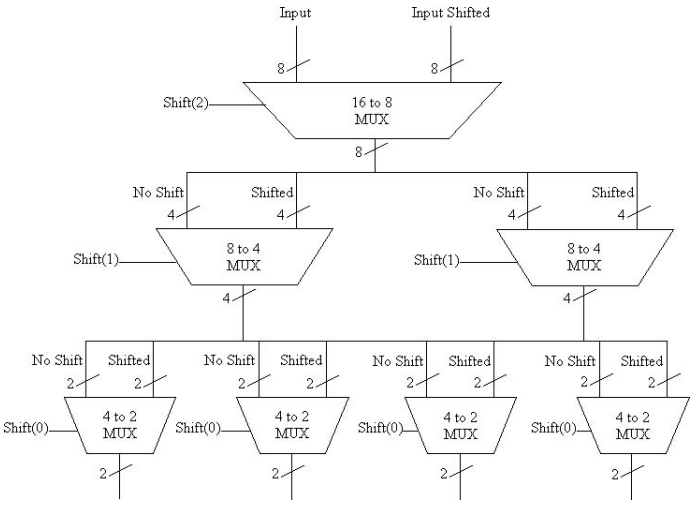

FIGURE 5.4: 8-bit barrel shifter

A barrel shifter is for example used for the implementation of floating-point arithmetic. To add two floats their significands must be aligned, which requires shifting the smaller number until it matches the exponent of the larger number.

As mentioned also parellelism plays a role in the efficiency of DSP processors. Figure 5.5 shows the parellelism of the ALU, MAC and shifter.

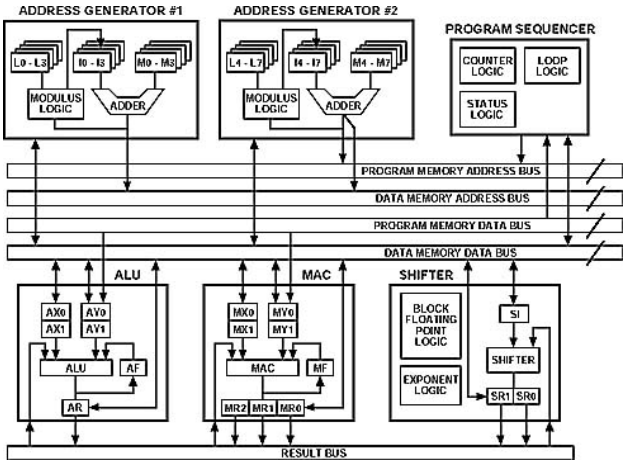

FIGURE 5.5: Useful DSP architecture

5.3.2 Memory architecture

Of course the memory of a DSP processor must be able to follow the high speed of the operations. Bus architecture must be optimized, there cannot be delays or bottlenecks.

- Most general-purpose microprocessors use the von Neumann architecture and throughput is limited because of having to choose between either fetching data or fetching an instruction in each cycle. In DSP processors program memory and data memory have there own space and busses and can both be fetched in one cycle, doubling throughput. This structure is known as the Harvard architecture. Naturally additional optimizations present on general-purpose processors, like instruction cache for example, are also present on DSPs.

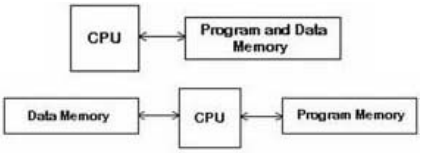

FIGURE 5.6: Memory architecture: Von Neumann (top) vs. Harvard (below)

- A lot of DSP algorithms need to get data from memory in repeating patterns. For example the fetching and manipulation of samples stored in an array. By reducing instructions needed for memory access instruction cycles can be saved. To reduce this overhead DSPs use specialized data address-generators (DAG) to automatically manage these types of repeated memory accesses. Because most DSP algorithms use two operands the DSP has two DAGs. One address generator creates the address for the data memory bus, the other one for the program memory bus. By this way the DSP is able to sustain execution of instructions in one single cycle.
- Often DSP algorithms require data in a range of addresses going from the end of the buffer back to the start of the buffer (wrap around). Therefore the DSP is provided with hardware circular buffers. The DAGs are specially designed for this task. General-purpose processors perform these functions in software and limit the ability to handle real-time signals.

5.3.3 Sequencer architecture

Since most algorithms runned on a DSP processor are by nature repetitive, the compute engine's program counter must be able to loop through the code without overhead while getting from the end of the loop back to the start. In general-purpose CPUs the loop's end condition is fully maintained in software. This condition instruction requires the fetching of addresses from memory and evaluation and takes time (cycles). DSP processors perform these tests in hardware, storing the needed addresses. In one cycle the condition is evaluated and either the first instruction of the loop is executed or the first instruction outside the loop is executed.

5.3.4 Input/Output architecture

Because of the high data throuhput requirement of the DSP the whole design is focused on making data accessible for the numeric, memory and sequencer sections. To transfer data into and out of the DSP quickly serial communications and DMA is used. The use of serial communications above parallel is naturally since this way data is sended one bit at a time, following the DSPs operating frequency and distributing work load. So the DSP communicates with ADCs, DACs or other devices through synchronous serial ports (SPORT). Other functions such as timers and boot logic (in other words: hardware interrupts) also ease DSP system design.

5.4 Fixed-point vs. floating-point DSP

Digital signal processing can either be done with fixed point or with floating point data storage. DSPs designed to represent and manipulate integers are fixed point realizations. Floating-point DSPs represent and manipulate rational numbers. The number is represented in a similar manner to scientific notation, with a mantissa and an exponent (e.g. $A \times 2^B$, where 'A' is the mantissa and 'B' is the exponent).

5.4.1 Advantages of floating-point digital signal processors

In a fixed point DSP numbers are represented with a fixed number of digits after the decimal point, for example: 123.45, 1234.56, 12345,67, etc. In a floating point implementation the decimal point can 'float'.In addition a floating point representation can represent 1.234567, 123456.7, 0.001234567, etc. Thanks to the exponential representation floating-point DSPs support a much larger dynamic range: very small numbers and very large numbers can be stored. Another advantage of this that floating-point processing yields much higher precision than fixed point processing.

5.4.2 Considerations

In digital signal processing rounding and/or truncating numbers yields quantization error or noise, a difference between the real-world value and the quantized digital values. This also happens each times a DSP does a mathematical calculation. As such, floating-point processors are the ideal DSP when accuracy is critical, like in audio applications. This is noticable in the program of our FM synthesizer, where constantly calculations are done with float data. When choosing between fixed-point or floating-point processor there are many other important factors to consider, e.g. processor cost, ease of development, performance. In summary, because of the greater number of general purpose applications that can use fixed point processors those are typically less expensive than floating-point versions due to the scale of manufacturing. Floating-point DSPs are optimized for computationally intensive applications.

Chapter 6

Implementation: C

Written by Bjorn Deraeve

6.1 Introduction

In chapter 5 we already looked at the DSP architecture and it's advantages compared to traditional CPU's. Now we will have a closer look at programming concerns unique to DSP algorithms and some related hardware provisions.
More specifically real-time aspects will be discussed. Obviously in analog systems every task is performed in real-time. All signals and processing are continuous. However in digital systems signals are discrete and represented by a set of samples. The interpretation of real-time is depending on the sampling rate. In order for a digital system to be operating in real-time, all processing of a given amount of data must be completed before the next amount of data arrives. Thus real-time is limited by the amount of data to process and by the type of processing that must be done on that data. These two constraints correspond to the sample frequency and the complexity of the running algorithms.

6.2 Time management

Since there is only a limited amount of time to perform any given algorithm, time management is an essential part of DSP software design. The resulting strategies determine how the processor is notified about events, how data is handled and how communication is done.

6.2.1 Interrupts

Event handling by interrupts logically is the most appropriate way to meet to the timing constraints. Naturally the interrupt rate is based on the data sampling rate. Typically DSP processors have extra built-in hardware mechanisms to handle these interrupts because they are more efficacious than software. Interrupt service routines (ISR) on a DSP processor must comply to all of the following demands:

Fast context switching: can be accomplished by using a second set of data registers. Only one set is active at a time, containing all the relevant data for the active context. When an ISR is called the switch can take place immediately without first having to temporalily save the active registers.

Nested interrupt handling: to handle nested interrupts, DSPs record their state corresponding to a certain context on a stack located in the DSPs program sequencer when an event occurs. This data structure allows that interrupts with higher priority can interrupt one with lower priority.

Continue to accept data while the ISR is running: for this two hardware features are provided:

- Interrupt latch: makes sure that no important events can be missed while servicing an interrupt.
- Automated I/O: external devices can put data into the DSP's memory without requiring intervention from the processor, so no data is missed while the DSP is busy (serial ports, DMA, autobuffering).

6.2.2 Data handling

Interrupts are typically generated due to data flow and may either occur with each sample, or after a block of samples has been collected. Each DSP's serial port has two I/O registers: a receive (Rx) and a transmit (Tx) register. When a (serial) word is ready to transmit or be received an interrupt is generated. In the ISR either the value of register Rx is read into a data register or a value is copied from a data register to the SPORT's Tx register. Obviously if I/O have the same sampling frequency one interrupt suffices.
Real-time systems typically introduce processing delay. A delay is introduced due to the A/D and D\A converter, the execution of the algorithm on the samples and by block processing.
The idea of block processing is to process the signal not sample-by-samply but in frames of n samples. Block processing allows that multiple operations are performed in parallel: samples are collected, samples are processed and samples are output. With this data handling technique performance can also be increased if the samples are transformed into the frequency domain and for example a Fast Fourier Transform (FFT) is executed.

6.2.3 Real-time interface issues

To create a real-time environment not only fast interrupt response, efficient data handling and fast program execution is important, but also the flow of data in and out the processor is significant.
However parallel transport is for the less obvious, it can also be used. In that case they are typically at least as wide as the processor's word width. Parallel transfer is always faster than serial transfer while this high speed is often not necessary. Each cycle data can be sent or received. This requires very fast peripherals to keep up with it (for example SRAM chips). Because this very high speed is mostly unnecessary and because of the higher cost mostly serial transfer is used.
Unlike the RS-232 protocol the serial ports (SPORT) of DSPs use a synchronous interface. The clock, receive data, transmit data and important frame (cf. block of samples) synchronization signals are passed through the interface and guarantee optimal communication. For example for left- and right-channel audio signals the serial data stream can also be time-division multiplexed (TDM).

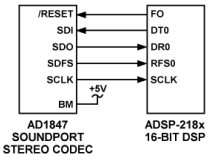

FIGURE 6.1: Serial communication interface

6.3 Analog Devices Sharc DSP Board

6.3.1 Introduction

Analog Devices has a wide portfolio of embededd processors, DSPs and microcontrollers for lots of general-purpose and application-specific goals. The SHARC processor we used dominates the floating-point DSP market. It's key features are summarized below. In appendix **??** a full functional block diagram for the ADSP-21369 is added for completeness.

Performance: 32-bit, 400MHz SIMD Core capable of 2.4 GFLOPS peak performance

32-bit external memory interface: supporting SDRAM, SRAM and FLASH

Digital Audio Interface (DAI): supporting access to user-definable peripherals like SPORTs, S/PDIF, precision clock generators, ... *In this project the DAI is used to output the generated signal to the audio jack.*

Digital Peripheral Interface (DPI): supporting access to user-definable peripherals like SPI ports, UARTs, I^2C and timers. *In this project the DPI is used to communicate with the labview interface and will be further discussed in chapter 7.*

DMA: contains 34 zero-overhead DMA channels

6.3.2 Initializing communication between the DSP and the audio jack

6.3.2.1 Digital Applications Interface (DAI)

As mentionned above, the DAI is used to output our sound signal through the headphone jack. A simplified system architecture block diagram is showed in figure 6.2 and 6.3.
The DSP's DAI pins (DAI-P20-1) are connected to signal routing unit 1 (SRU) of the processor. The SRU is a flexible routing system which allows to route the DAI pin to different internal peripherals (SPORTs, S/PDIF, PCGs) in numerous combinations. Because the inputs and outputs of the peripherals are not directly connected to the processor but via the SRUs, an arbitrary amount of peripherals is possible which allows a wider variety of compactible applications without having to increase the pin count. So the SRU is simply a group of multiplexers under software control (memory-mapped registers). For our synthesizer application the SPORT peripheral is used.
The DAI is also connected to the AD1835 audio codec which makes in turn the connection to the audio jack, see figure 6.6.

6.3.2.2 Signal Routing Unit (SRC)

Like is said above, the SRUs are used to connect inputs and outputs. The signals routed by the SRUs are diveded into functional groups. Each group routes a set of signals with a specific purpose. For example, SRU1 Group B routes serial data signals. It are both the peripherals control registers and the SRU configuration that determine the direction of the signal flow in a pin buffer.
To configure the SRU we must thus write values that correspond to signal sources into the bit fields that further correspond to the signal inputs. However the registers are arranged into functional groups, this is stays a very difficult task. In order to ease this coding process one of the following two procedures can be used:

Expert DAI plug-in: This software plug-in is included in the VisualDSP++ IDE and greatly simplifies the task of connecting the signals in the SRUs.

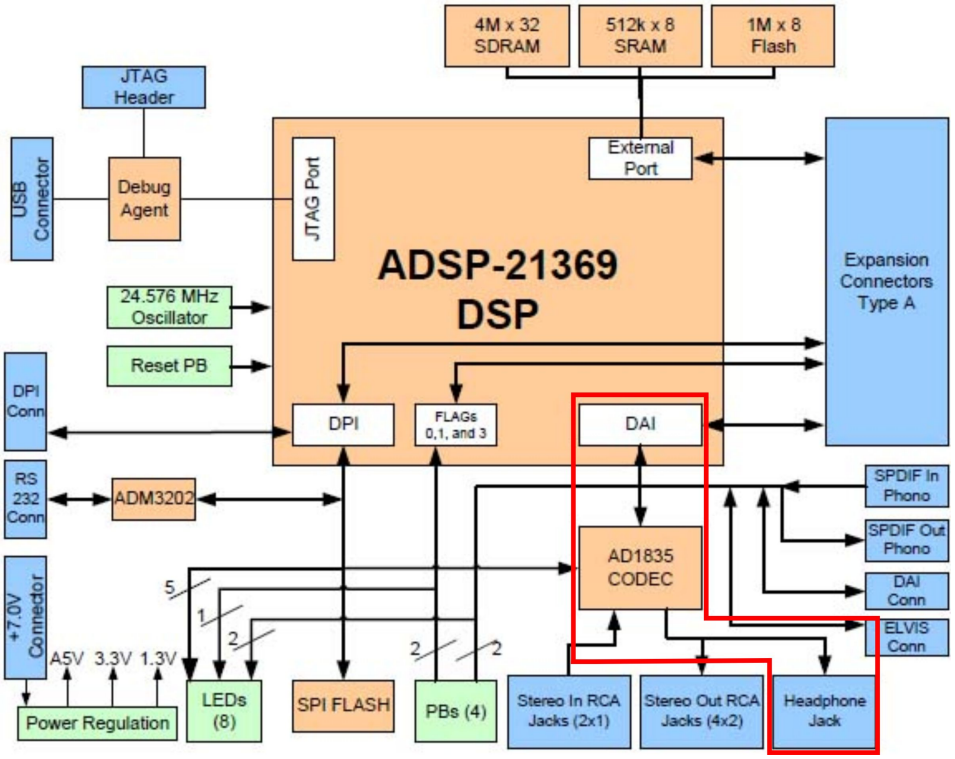

FIGURE 6.2: System Architecture Block Diagram

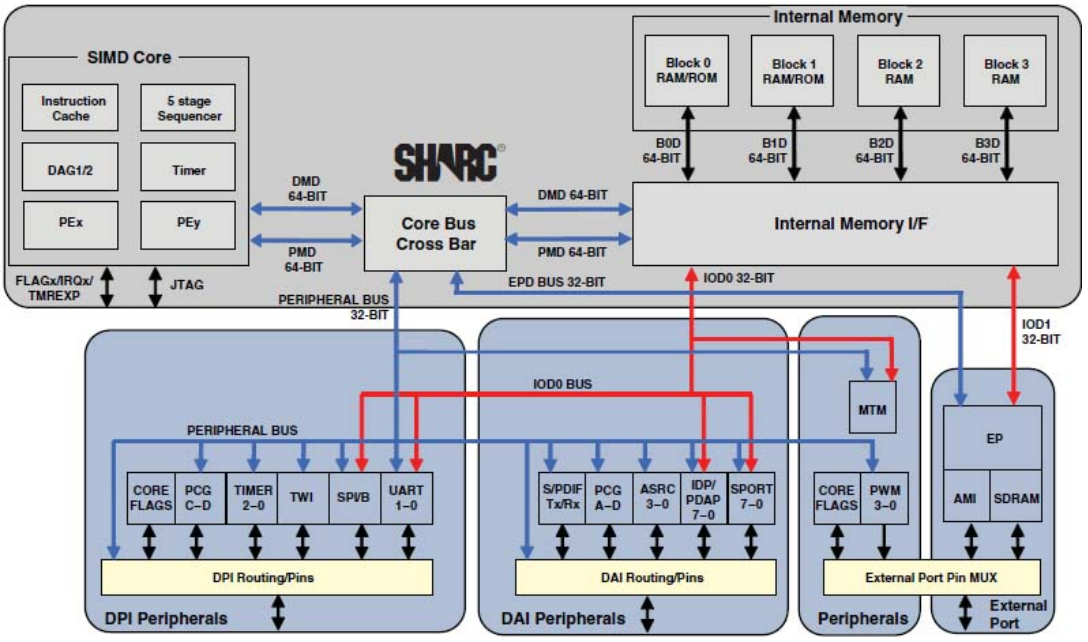

FIGURE 6.3: Peripherals Architecture Block Diagram

sru.h: This is a macro implementation that automates most of the work of signal assignments and functions. The macro can be used like is shown below to connect the ADC to SPORT0 using data input A. (See file *initSRU.c*)

```
#include <sru.h>;
/* The following lines illustrate how the macro is used: */
/* Route SPORT 1 clock output to pin buffer 5 input */
      SRU(DAI_PB05_O,SPORT0_DA_I);
```

To explain the input/output mnemonic, which always ends with `_I` if the signal is an input and with `_O` if the signal is output, we must explain one more thing in this section. Because of the context of the SRU, a pin is not just a pin anymore. Pins are now replaced by logical interfaces (pin buffers), shown in figure 6.4.

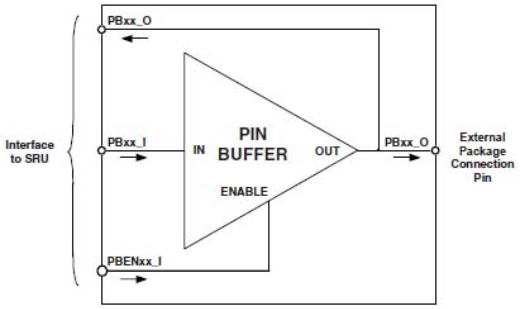

FIGURE 6.4: Pin Buffer

The interface's input must be thought of as the input to a buffer amplifier which can drive a load on the external pin. The pin buffer enable pin is an input signal that enables the buffer if its value is logic high and disables the buffer when its logic low. When the pin enable is set the output pin is logically equal to pin input. If not, the output of the buffer becomes high impedance and an external device can sent a logic value to the input of the ADSP-21369. The SRU receives this pin as an pin interface output and routes it as an input to the SHARC processor. If pin enable is asserted pin output is equal to pin input (the amplifier then acts as a current source, low impedance). See appendix **??** for more explanation.

6.3.2.3 AD1835A

The pins of the DAI are connected to an AD1835A audio codec. This is a high performance, single-chip codec with four stereo DACs and one stereo ADC, so 8 DACs and two ADCs.
The engine has a programmable interpolator which allows the user to select different interpolation rates. By this way the desired sample rate can be selected. For the DACs the user can choose between 48kHz, 96kHz and one DAC even supports 192kHz, see the table below extracted from the AD1835A's datasheet. The location of this chip on the ADSP-21369 is shown in figure 6.5. Table 6.1 shows the connections between the processors DAI and the AD1835A audio chip. We used the 96kHz sample frequency.

Sample Rate	Interpolator Rate	DAC Control 1 Register
48 kHz	8x	000000xxxxxxxx00
96 kHz	4x	000000xxxxxxxx01
192 kHz	2x	000000xxxxxxxx10

TABLE 6.1: AD1835A DAC Sample rate settings

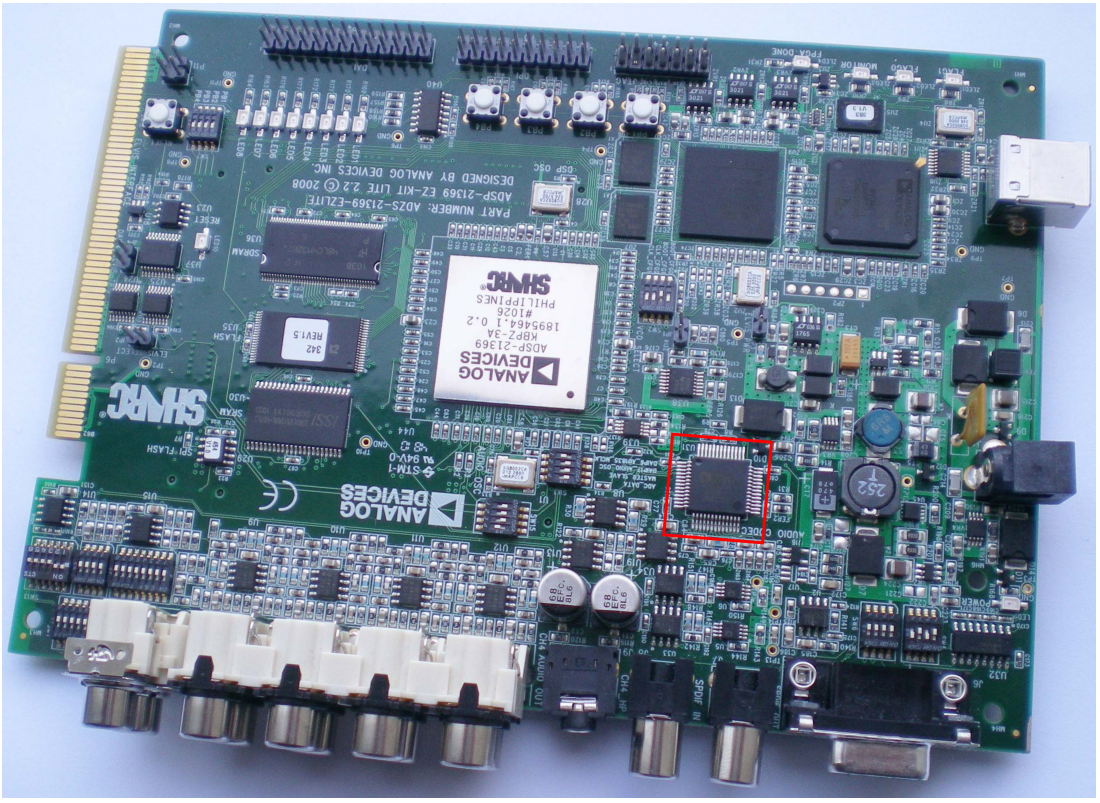

FIGURE 6.5: AD1835A on the ADSP-21369 DSP Board

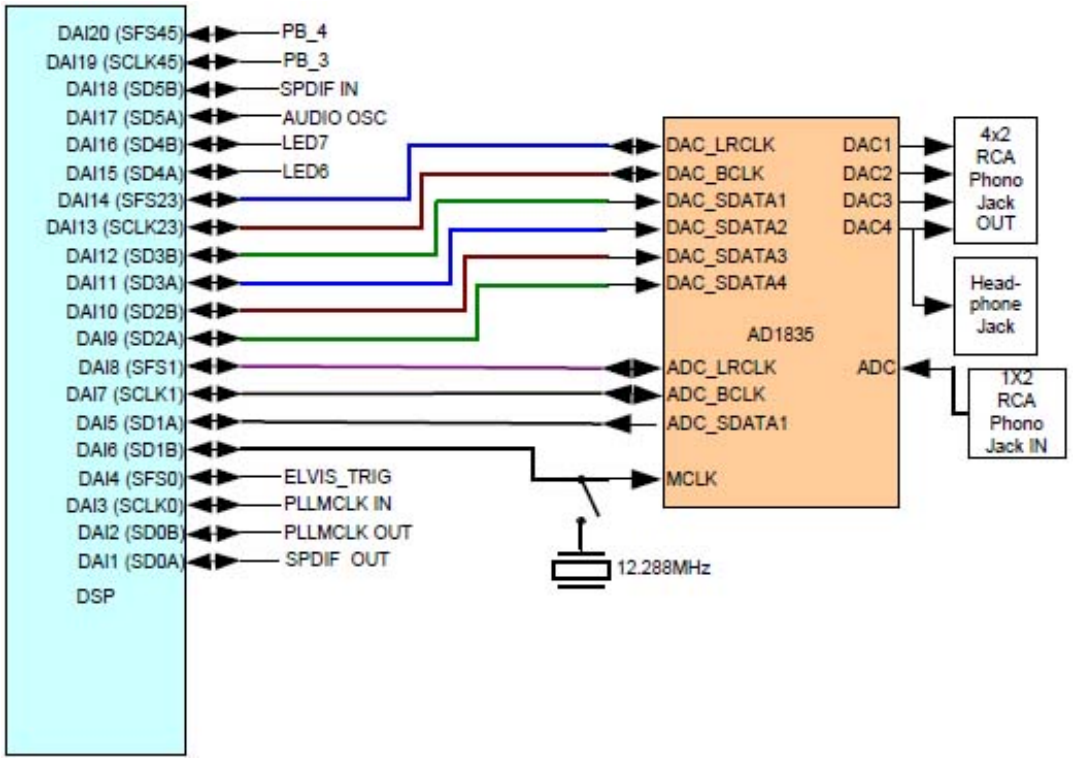

FIGURE 6.6: Peripherals Architecture Block Diagram

To make the AD1835A's four serial data output connections shown in figure 6.6 we use the macro defined in <*sru.h*>, which is shown below. For the complete initialization (clock and frame sync) we refer to the file *initSRU.c*.

```
SRU(SPORT2_DB_O,DAI_PB09_I);  <-- notice that DAC4 is connected to SPORT2B
SRU(SPORT2_DA_O,DAI_PB10_I);
SRU(SPORT1_DB_O,DAI_PB11_I);
SRU(SPORT1_DA_O,DAI_PB12_I);
```

6.3.2.4 SPORT

The previous sections explained how to set up the connections between the DAI and the DAC but we still have to initialize the communication protocol to be used. The ADSP-21369 has eight independent, synchronous serial ports (SPORT) that provide us with an I/O interface. Each SPORT has a set of control registers and data buffers which allows a variety of serial communication protocols.
In the synthesizer program we wil use SPORT2B to communicate from processor to DAC4 in I^2C mode. The serial data will be transferred automatically from on-chip memory using DMA block transfers in a chained DMA process.

Chained DMA: In such a process the block to transfer comes from the (next) chain pointer register. When the current DMA transfer is complete the processor automatically begins the next DMA transfer indicated by the pointer register. To set up a chain of DMA operations the following steps are used:

1. Set up all TCBs in internal memory:

```
unsigned int Block_A[NUM_SAMPLES] ;
unsigned int Block_B[NUM_SAMPLES] ;
unsigned int Block_C[NUM_SAMPLES] ;

TCB_Block_A[0] = (int) TCB_Block_C + 3 - OFFSET + PCI ;
TCB_Block_A[3] = (unsigned int) Block_A - OFFSET ;

TCB_Block_B[0] = (int) TCB_Block_A + 3 - OFFSET + PCI ;
TCB_Block_B[3] = (unsigned int) Block_B - OFFSET ;

TCB_Block_C[0] = (int) TCB_Block_B + 3 - OFFSET + PCI ;
TCB_Block_C[3] = (unsigned int) Block_C - OFFSET ;
```

2. Set up the appropriate DMA control registers: set DMA enable bit `SDEN_B`, set chaining enable bit `SCHEN_B`. This information is written in the I^2C mode control bits of the SPORT, `SPCTLx`.

```
/* Channel enable: SPEN_B
 * Word length: SLEN24
 * Operation mode enabled: OPMODE
 * Activate TXSPxy data buffers and transmit shift registers: SPTRAN
 */
    *pSPCTL2 |= (SPTRAN | OPMODE | SLEN24 | SPEN_B | SCHEN_B | SDEN_B) ;
```

3. Write the adres of the first (next) TCB to the chain pointer register. For our program we use the SPORT Chain Pointer Registers `CPSPx`, so for SPORT2B this gives:

```
    *pCPSP2B = (unsigned int) TCB_Block_C - OFFSET + 3 ;
```

For the complete initialization of the SPORT please have a look at file *initSPORT.c*. Remark that the data for the transmit buffers `TXSPxA/B` is loaded automatically by the DMA controller.

SPORT interrupt: The SPORT DMA mode provides a mechanism to transmit an entire block of serial data and creates an interrupt afterwards rather than after each transmitted word, significantly reducing overhead.
As is explained in section 6.1 all processing of the algorithms must be done before the next block of data is provided. To keep track of this real-time constraint the semaphores *isProcessing* and *blockReady* are introduced and update the real-time flag each time a SPORT DMA interrupt occurs. If the program becomes too slow, real-time is set false and a warning sound will be produced. If the processing is done before the interrupt occurs the SPORT chain pointer register is updated:

```
void TalkThroughISR()
{
    if(isProcessing) //still busy calculating
    realtime=0;

    (int_cntr++); //Increment the block pointer
    int_cntr %= 3;

    blockReady = 1;
}
```

6.4 The DSP Synthesizer program

All instructions for the Sharc processor are written in C or assembly. Once finished all source code is translated and linked by the compiler integrated in VisualDSP++. Then this executable machine code is uploaded to the device.

6.4.1 Program execution

To follow the next descriptions please take a look at figure 6.7. The *main.c* file contains the central starting point, namely the `main()` function. At run time, the execution will start at the first line of this function. In our case this is a series of `Init()` functions, which were briefly discussed in the previous section.
Once all initializations are done the real work can be started. At this point we enter the `while(!stop)` loop. As long as the user does not press the stop button the microprocessor continues running and must do something, so an infinite loop keeps it busy. This is typically for embedded controllers.
Next is another while loop: `while(blockReady)`. After the previous block has been processed and the SPORT DMA interrupt has occured the next block can be processed. This next block will either produce the pursuit of the previous frame or will produce a warning tone, indicating that the calculations on the previous block were not finished before the block was needed by the DAC, and thus the signal could not be produced in realtime. We will first explain the not-realtime case since this will also be helpful in the realtime process.

6.4.1.1 Not-realtime

The not-realtime-function takes two parameters as argument, both pointers and creates a 8000Hz warning tone. The first pointer is the address of the array containing the samples to be processed. This array contains 1024 samples and is contained in a TCB block, used by the DMA chain to feed the DAC. The second is a simple int pointer which is used to indicate which particular samples to create, fulfilling the role as the time of the sine function in this case.

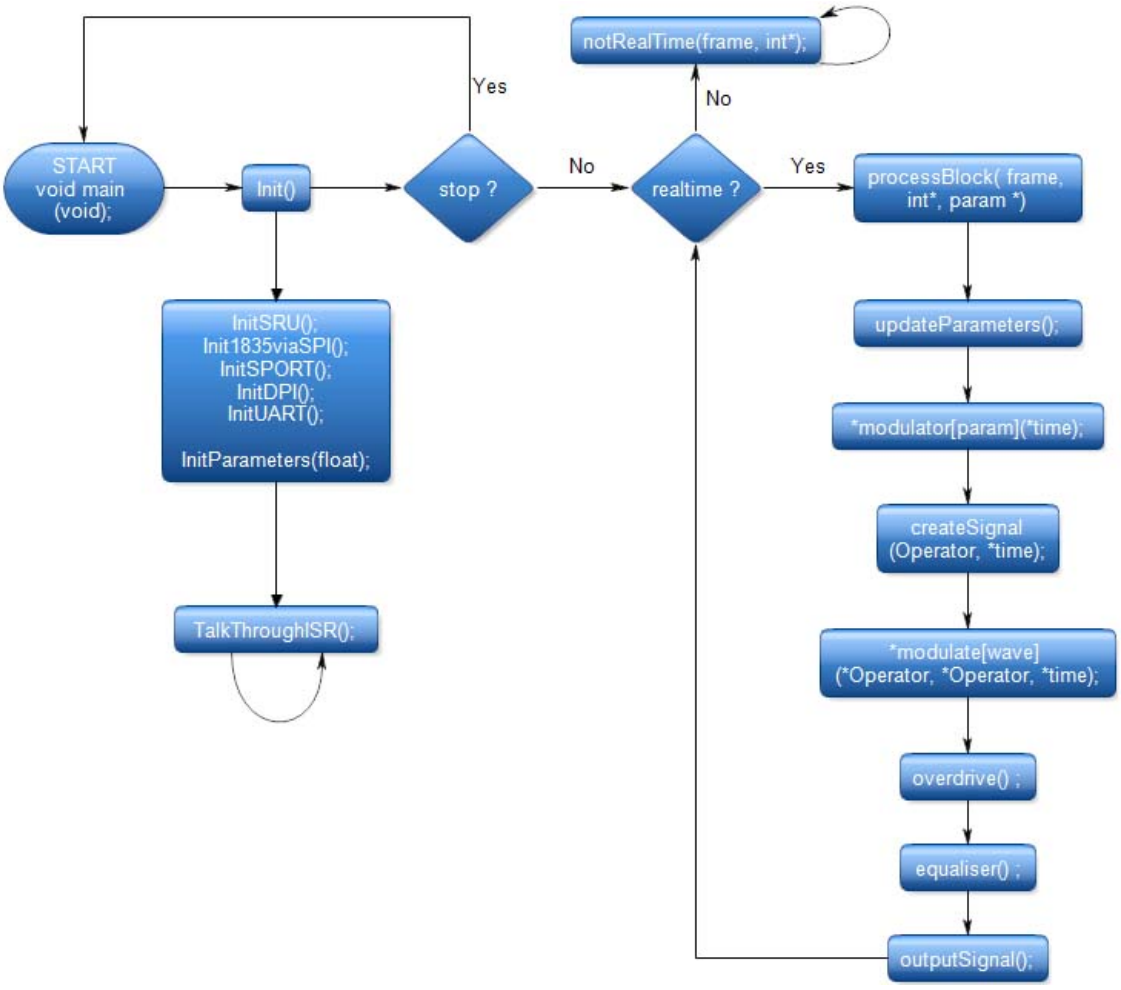

FIGURE 6.7: Flow chart of the DSP Synthesizer program
(Compare this figure with figure 2.12)

```
void notrealtime(unsigned int *block_ptr, int *time)
{
   int i;
   float scalar = powf(2.0,23.0);
   long j,N,readptr,ref;

   //Clear the Block Ready Semaphore, Set the isProcessing Semaphore before starting
   blockReady = 0;
   isProcessing = 1;

   for(i=0;i<NUM_SAMPLES;i++)
   {
       *(block_ptr+i) = 0.1*scalar*sin(2.0*3.141592*8000.0/sampleFreq * ( (int)(*time)++ ));
       (*time)%=sampleFreq;
   }

    //Clear the Processing Active Semaphore after processing is complete
    isProcessing = 0;
}
```

6.4.1.2 Realtime: let's play

In order to easily manipulate our sound processing algorithms we need the same arguments as indicated in section 6.4.1.1 plus an addiditonal pointer used to communicate with the Labview interface.

Refresh parameters: Before we start to create our signal we first fetch the parameters as the user defines them in the Labview interface. Those values are put into global variables so they have a reserved spot in the program memory.

Select modulation type: Depending on which modulation algorithm the user selects a different function will be executed. This is almost literally the definition of function pointers, isn't it? Since the different modulation modes have a fixed value in the labview interface we can define the function pointers very efficiently. The definition of the function pointer we used is a function that takes one int pointer as argument end returns void. Below the creation and call of the (fancy) function pointers is shown:

```
Modulation *modulator[4] = {&TwoOperators, &ThreeOperators, &FourOperators,
&FourOperatorsPlus};

(*modulator[(int) parameters[0]])(time);
```

Create and modulate signal: For efficiency reasons we mostly pass arguments by reference. However in the case below we use pass by value because we don't want to adapt the value of the pointer immediately (and copying the value to a local variable is also a bit sloppy). So to create and modulate the signal first dereference the time pointer received by the function pointer:

```
void TwoOperators(int *t)
{
   int i;
   Operator *carrier = &op[0];
   Operator *modulator = &op[1];

   createSignal(modulator, *t);
   (*modulate[carrier->wave])(carrier, modulator, *t);
}
```

As you can see, the modulation algorithm also depends on which type of wave is selected and this is also fixed with function pointers. For the complete implementation of the modulate algorithms we refer to the file *operator.c*.

Overdrive: If the effect is enabled the function overdrive is called. This function takes as arguments operator 0 and a pointer to the effect's parameters. The creation and modulation algorithms are so that operator 0 always contains the whished signal, this is why it is used as argument here and in the next functions.

```
if( effect.enabled ) overdrive(&op[0], &effect);
```

Equaliser: If the equaliser is enabled the following function is called:

```
if( equaliser.enabled ) enableEqualiser(&op[0], &equaliser);
```

Output signal: Now we have fully modulated our signal we still have to put it into the TCBs to send it to DAC4. One last thing that is added here is a control of the overal gain of the signal.
We may not forget that because we wanted to use the timing values several times we used a pass by value. That's why after the whole frame is processed we still have to update the time reference.

```
for(i=0;i<NUM_SAMPLES;i++)
{
       *(block_ptr+i) = (unsigned int)(signed int)(op[0].signal[i] *
              Gain(&equaliser)*0.1*scalar );
}
(*time)+= 1024;
(*time)%=sampleFreq;
```

6.4.2 Key functions

void createSignal(Operator *, int) {

```
   int i;
   switch(op->wave)
   {
   case SINE:
      for(i=0; i<NUM_SAMPLES; i++)
      {
            op->signal[i] = envelope(op) * sin( 2.0 * 3.141592 *
                     op->frequency / sampleFreq * t++ );
      }
      break;
   case TRIANGULAR:
      for(i=0; i<NUM_SAMPLES; i++)
      {
            op->signal[i] = 1-(fabs((t+++op->frequency%sampleFreq)-sampleFreq/2)
                        / (sampleFreq/4)  - 1 );
            op->signal[i] *= envelope(op);
      }
      break;
   case SAWTOOTH:
      for(i=0; i<NUM_SAMPLES; i++)
      {
            op->signal[i] = 2*( t+++(op->frequency)%sampleFreq )/sampleFreq - 1;
            op->signal[i] *= envelope(op);
      }
      break;
   default:
      break;
   }
}
```

Remark: Why is there no t%=sampleFreq in this function?

void ModulateSine(Operator *carrier, Operator *modulator, int t) Please remember formula 2.10 discussed in chapter 2 on frequency modulation.

```
   int i;
   for(i=0; i<NUM_SAMPLES; i++)
   {
      carrier->signal[i] = sin( 2.0 * 3.141592 * carrier->frequency /
                  sampleFreq * t++ + (modulator->signal[i]) );
      carrier->signal[i] *= envelope(carrier);
   }
```

Chapter 7

Communication with LabVIEW User Interface using UART Port Controller

Written by Thuy Pham

7.1 Introduction

Communicating with peripheral devices plays a vital role in system design in general and in DSP systems specifically. For this purpose, SHARC Processor uses Digital Peripheral Interface (DPI). The interface provides both connections to two serial peripheral interface ports (SPI) and two universal asynchronous receiver-transmitters (UARTs). In this report, Universal asynchronous receiver transmitters (UARTs) will be examined in relationship with LabVIEW User Interface. It is necessary to notice that the processor provide a full duplex universal asynchronous receiver/-transmitter (UART) port, which is fully compatible with PC-standard UARTs. The UART port provides a simplified UART interface to other peripherals or hosts, supporting full duplex, DMA supported asynchronous transfers of serial data. The UART also has multiprocessor communication capability using 9-bit address detection. This allows it to be used in multi-drop networks through the RS-485 data interface standard. The UART port also includes support for five data bits to eight data bits, one stop bit or two stop bits, and none, even, or odd parity. More specifically, the UART port supports two modes of operation as mentioned below:

- **PIO (programmed I/O):** The processor sends or receives data by writing or reading I/O mapped UART registers. The data is double-buffered on both transmit and receive.
- **DMA (direct memory access):** The DMA controller transfers both transmit and receive data. This reduces the number and frequency of interrupts required to transfer data to and from memory. The UART has two dedicated DMA channels, one for transmit and one for receive. This is the mode that will be used for transferring data between LabVIEW program and DSP board.

7.2 UART data frame

When two UARTs communicate, both transmitter and receiver need to know the signaling speed. The receiver does not know when a packet will be sent (no receiver clock); thus, the protocol

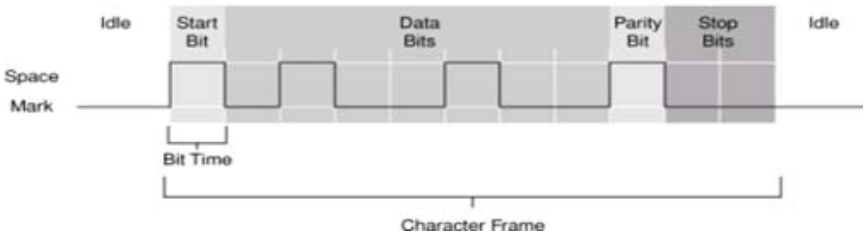

FIGURE 7.1: A typical UART data frame

is termed "asynchronous." This is very important because if we set up parameters (baud rate, start bit, stop bits, and parity bit) in both transmitter and receiver is incompatible, we will get incorrect data. In addition, the receiver circuitry is correspondingly more complex than that of the transmitter. The transmitter simply has to output a frame of data at a defined bit rate. Contrastingly, the receiver has to recognize the start of the frame to synchronize itself, and therefore determine the best data sampling point for the bit stream.
The processor supports a set of parameters' values for UART communication as below:

SST	Parameters	Value
1	Baud rate	2400, 4800, 9600, 19200, 38400, 57600, 115200, 921600, 6250000
2	Data bits	5 to 8
3	Stop bits	1 or 2
4	Parity	None, even, odd

7.3 UART external interface

The DSP processor communicates with peripheral devices in DSP board is described as the block diagram in figure 7.2:

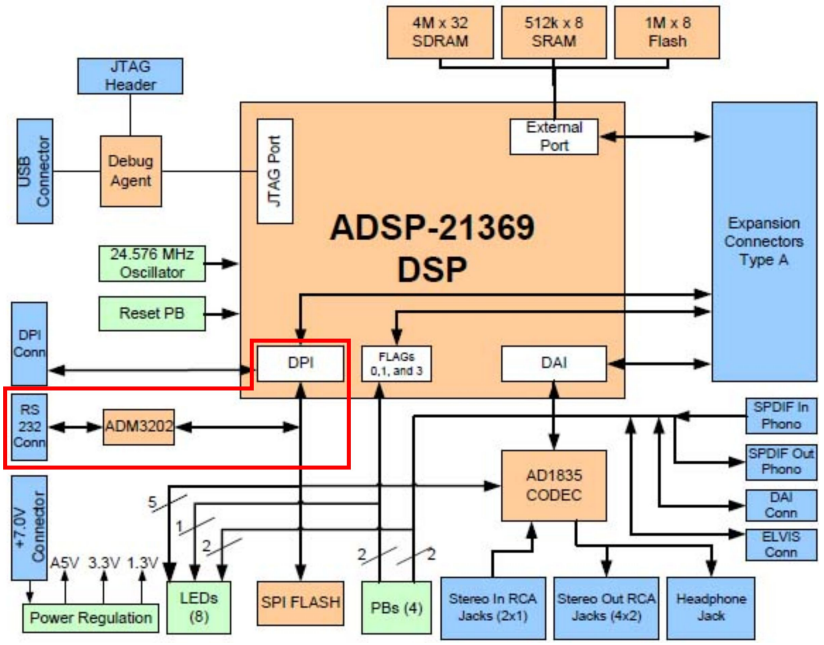

FIGURE 7.2: System Architecture Block Diagram

In this board, the EIA-232E interface is used to communicate with other external serial com-

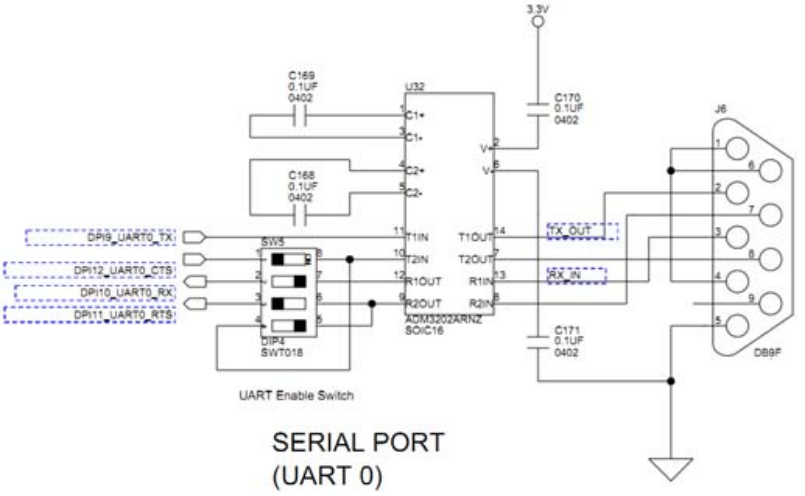

FIGURE 7.3: Schematic for connection of UART port

munication devices. Instead of using traditional circuit for compatible purpose of power levels, in the DSP board, manufacture uses ADM3202 chip. The significant features of the chip are low power consumption and can operate at data rates up to 460 kbps make them ideal for battery powered portable instruments and high speed requirement.

7.4 UART configuration in DSP Processor for communication

The DSP Processor provides a set of PC style, industry standard control and status registers for the UART.
Register Overview for UART Module:

- Line Control Register (UARTxLCR). Controls format of the data character frames. It selects word length, number of stop bits and parity.
- Divisor Latch High/Low Register (UARTxDLL, UARTxDLH). Characterize the UART bit rate. The divisor is split into the divisor latch low byte (UARTxDLL) and the divisor latch high byte (UARTxDLH).
- Mode Control Register (UARTxMODE). Controls packing and address modes.
- Transmit Buffer Control Register (UARTxTXCTL). Controls core or DMA operation.
- Receive Buffer Control Register (UARTxRXCTL). Controls core or DMA operation.
- Interrupt Enable Control Register. Enables interrupt requests from system handling.
- Line Status Register (UARTxLSR). Returns status of controls format of the data character frames as overrun or framing errors and break interrupts.
- Transmit Status Register (UARTxTXSTAT). Returns status of core or DMA operations.
- Receive Status Register (UARTxRXSTAT). Returns status of core or DMA operations.
- Interrupt Identification Status Register (UARTxIIR). The register is used to get the status of all interrupts into one channel.

To configure parameters for transmitting data between PC and the DSP board, the number of simple steps can be followed:

1. **Route the UART to the DPI of DSP Processor**
   The routing is implemented by software, in particular in the VisualDSP++, it is in the SRU macro. It is necessary to notice that, data is transmitted and received by the least significant bit (LSB) first (bit 0) followed by the most significant bits (MSBs).

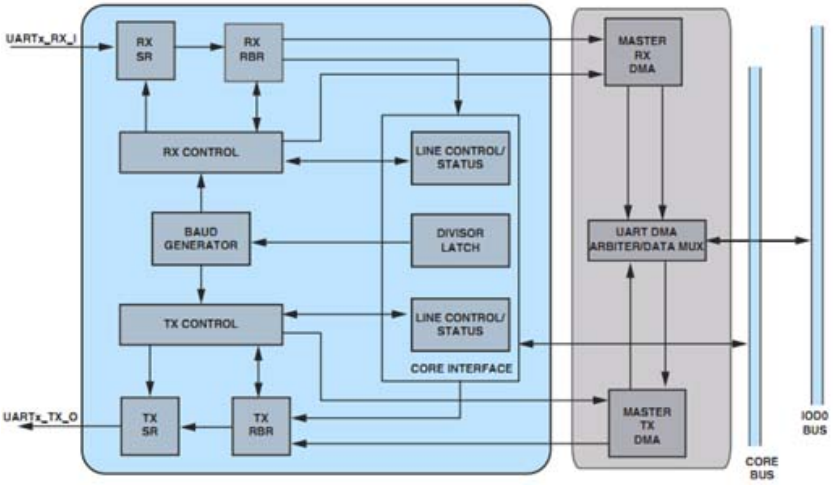

FIGURE 7.4: UART Functional Block Diagram

2. **Set up baud rate, data bits, stop bits, parity bit**
   The bit rate is characterized by the peripheral clock (PCLK) and the 16-bit divisor. The divisor is split into the UART divisor latch low byte register (UARTxDLL) and the UART divisor latch high byte register (UARTxDLH). UART Baud rate = PCLK/(16 divisor), where PCLK is the system clock frequency.
   All data words require a start bit and at least one stop bit. With the optional parity bit, this creates a 7 to 12-bit range for each word. The format of received and transmitted character frames is controlled by the line control register (UARTxLCR).

3. **Set up core mode of operation**
   This requires software management of the data flow using either interrupts or polling.
   Core transfers move data to and from the UART by the processor core. To transmit a character, load it into the UARTxTHR register. Received data can be read from the UARTxRBR register. The processor must write and read one character at time. To prevent any loss of data and misalignments of the serial data stream, the UART line status register (UARTxLSR) provides two status flags for handshaking - UARTTHRE and UARTDR.
   Core transfers through the UART is started by setting up and writing to transmit and receive control registers, enabling the module using the UARTEN bits in the UARTxTXCTL and UARTxRXCTL registers.

4. **Set up the UART interrupt**
   The UART receive and transmit interrupts are programmed through the peripheral interrupt control registers (PICRx) as separate interrupts. (By default, these interrupts are not configured in the IRPTL register - the PICRx register has to be programmed to configure them.)
   The UART interrupt enable register (UARTxIER) is used to enable requests for core system handling of empty or full states of UART data registers. When polling is used as a means of action, the UARTRBFIE and/or UARTTBEIE bits in this register are normally set.

7.5 Programming for communication between DSP Processor and LabVIEW User Interface

7.5.1 LabVIEW program for writing data

LabVIEW program provides a set of powerful tools for programming User Interface. Based on these things, user can make a program that clearly and visually shows parameters, graphs and results. LabVIEW also has many instrument drivers, which are a set of modular software functions that use the instrument commands or protocol to perform common operations with the instrument, for a variety of programmable instruments that use the GPIB, VXI, PXI, or serial interfaces.
With serial communication, in particular in RS-232, which is a standard developed by the Electronic Industries Association (EIA), LabVIEW provides some VISA functions for the target. They are located in **All Functions** >> **Instrument** >> **I/O** >> **Serial** The VISA Configure

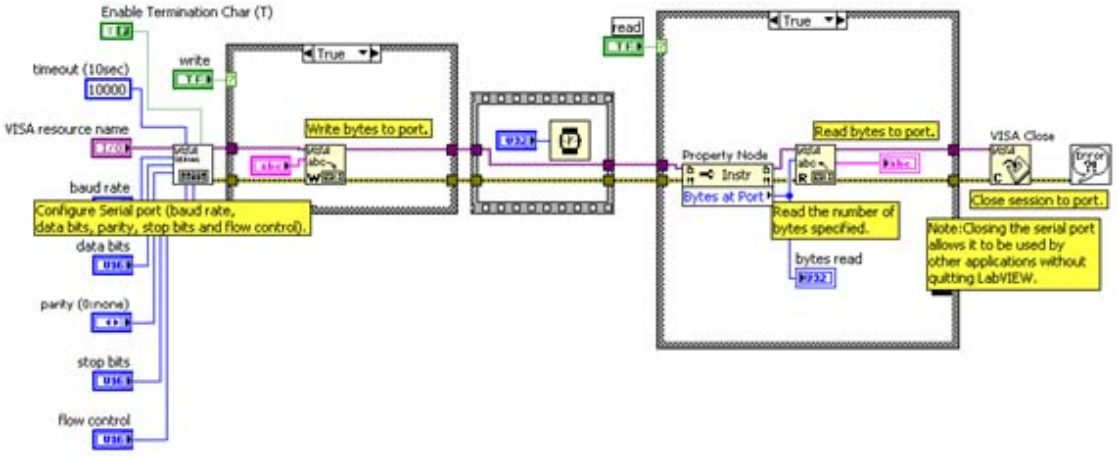

FIGURE 7.5: Write/Read a string to and from a COM port

Serial Port VI initializes the port identified by VISA resource name to the specified settings: timeout sets the timeout value for the serial communication; baud rate, data bits, parity, and flow control specify those specific serial port parameters.

1. The VISA Configure Serial Port VI initializes the port identified by VISA resource name to the specified settings: timeout sets the timeout value for the serial communication; baud rate, data bits, parity, and flow control specify those specific serial port parameters.
2. The VISA Write function sends the string.
3. The VISA Read function reads back up to bytes into the buffer, and the Simple Error Handler VI checks the error condition.
4. The VISA Close function terminates the communication channel to the instrument and deal locates the resources for the DSP.

Depending on the purpose, the string can be written to the device in different forms. As the subVI described below, the string is combined by three parts, which are identified parameter order (two characters), the value needed to pass, and the character that uses for identifying read or write (character "w"). It is important to notice that maximum character transmission rate is depended on the baud rate and the bits per frame. More specifically, this rate is just the baud rate divided by the bits per frame. So, when there is a consecutive transfer required, the number of characters per string needs to be cared. Otherwise, there will be data lost. Based on the prime

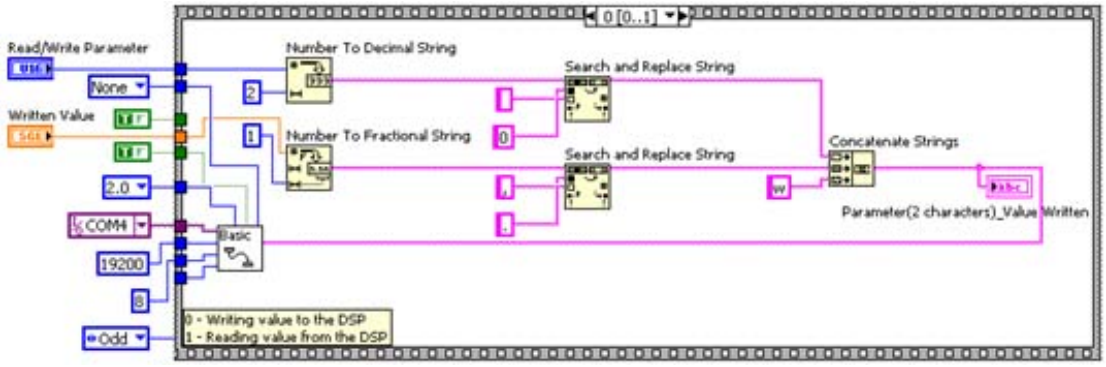

FIGURE 7.6: Formatting a string before writing to DSP

subVI as showed in figure 7.6, there are a number of ways to program an application that writes and reads multiple parameters to and from a DSP Processor. It needs to be reminded that, the principle of transferring data between DSP and PC is based on the interrupt. In other words, every time data is written to the DSP from PC, the serial DSP interrupt occurs and then its ISR (interrupt service routine) will run and process the received data. As a result, it is impossible to write more than one parameter at the same time that the DSP can process independently. Of course, each parameter has a different purpose. Because of this, the parameters should be organized so that only one parameter can write its data to the DSP at the time. It is very important for copying data between parameters that will be studied later.
Figure 7.7 shows an example program that wants to write multiple parameters to the DSP.

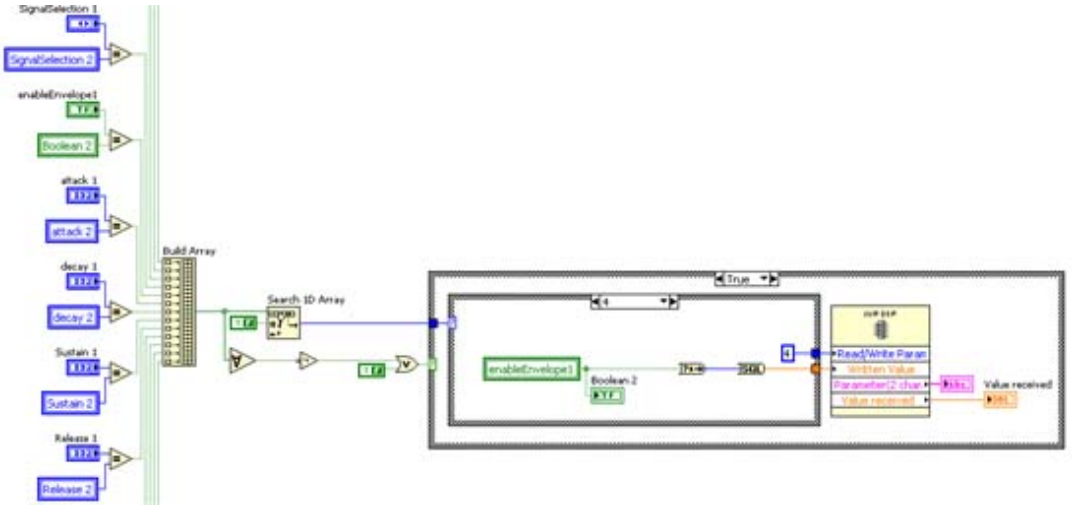

FIGURE 7.7: A part of writing parameters to Operator 1 program

The idea of the program is that every time a parameter changes its value, which is detected due to search 1D array, the read/write part starts working. Otherwise, the program does nothing. This reduces a lot of works for the DSP Processor. Then, the latest value of the parameter and its order will be combined into string before writing to the processor. Notice that, LabVIEW also provides some data convert functions so that different data types of parameters can work properly.

7.5.2 UART Interrupt

There are three things needed to be cared while using UART interrupts. First, you must map the UART interrupts to one of the interrupts in the interrupt vector table. In particular, the UART has to be mapped to exactly the peripheral interrupt sources. This can be achieved by changing the default source of the peripheral interrupt priority control register with the UART source, using the interrupt select values of the UART receive interrupt. The receive interrupt select values for UART0 (using in our application program) and UART1 are 0x13 and 0x14, respectively.

IIUART0RX	0x3E00	Internal Memory Address for UART0 Receiver

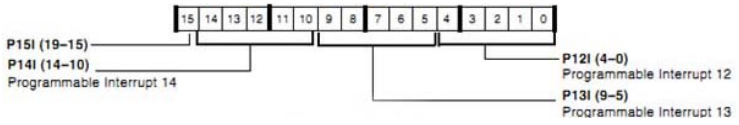

FIGURE 7.8: PICR2 Register

```
*pPICR2 &= ~(0x3E0);  // Sets internal memory address for UART0 receiver
*pPICR2 |= (0x13<<5); // Sets default value (0x13) for the URAT0 receive interrupt
```

The second thing is that enabling the UART interrupts internally by setting the corresponding bits in the UART interrupt enable register (UARTxIER). This needs to be done after all the UART settings (such as word length, parity, and so on) have been programmed, because in transmit mode as soon as the transmit buffer empty bit is enabled in the UARTxIER register, it vectors to the interrupt. If this bit is enabled before any of the UART settings are programmed, the data transmitted in the transmit interrupt service routine does not comply with the UART settings that are programmed later, leading to a communication error.

```
*pUART0IER = UARTRBFIE;   // Enables UART0 receive interrupt
```

The third, which is also the most important, is that using C Interrupt Handler with own Interrupt Service Routine. C provides its own set of interrupt handlers via the interrupt() and signal() functions. VisualDSP++ has extended the interrupt() function with **interruptf()** and **interrupts()** versions. The interrupt handler consists of three parts - the initialization, the interrupt vector, and the interrupt service routine.

```
interrupt(SIG_P13, UARTisr);
```

Initialization performs the tasks previously associated with the C interrupt() function - assigning the interrupt service routine to an interrupt vector (dynamic ISR vectors only), unmasking the interrupt, and enabling global interrupts. The interrupt vector routes execution to the appropriate interrupt service routine. This routine must be compatible with the procedure of writing data of LabVIEW program. Otherwise, the data that DSP Processor expected to receive is different with data transmitted.

```
// Read the data from the buffer
// Receive Buffer Registers (UARTxRBR)
   value = *pUART0RBR;

/* Echoes back the value on to the hyperterminal*/
```

```
// Wait until it is sent
   while ((*pUART0LSR & UARTTHRE) == 0){ ; }

// Writting a string to DSP
// The character in buffer is "w" (ASC-II of "w" = 0x77 equivalent to 119 decimal)
   if(value == 119)
   {
      for(i = 0; i < 2; i++)  address[i] = inputstream[i];

      // The two first characters writting from LabView is the index
      // int atoi(const char *str); - Convert string to integer
         index = atoi(address);

      // The rest of the writting string is value (with a character - w)
      // Double atof ( const char * str ); - Convert string to double
      // 2 - is the place of starting value string
         writtenValue = atof(inputstream + 2);

      // Writting value to the global variable
         algoparam[index] = writtenValue;

      // Adds data to the buffer
      // Transmit Holding Registers (UARTxTHR)
         *pUART0THR = value;

         teller = 0;
   }
```

7.5.3 The FM synthesizer application

The application is programmed to provide two services. First, the program can write the values of parameters to the DSP so that DSP board can produce different sounds. Secondly, the LabVIEW User Interface also can show a number of features of the signal, such as the form of the envelope, frequency domain and the spectrogram.
To provide the features above, the application is programmed following a hierarchy as figure 7.9.
There are four main parts programmed in the application, which are programs for different

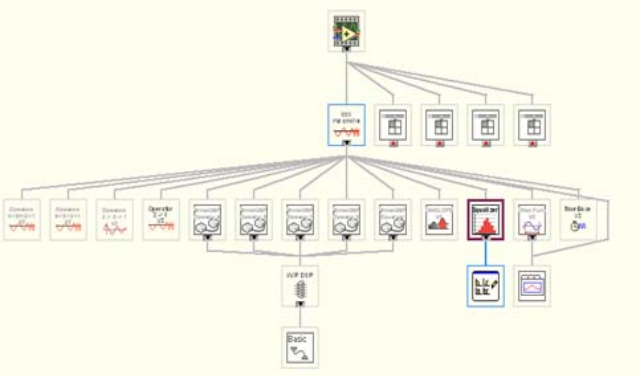

FIGURE 7.9: VI Hierarchy of LabVIEW User Interface

algorithms of combining operators, programs for the envelope, programs for equalizer, sound effect and display, and writing data to the DSP board. The first part includes four algorithms: two operators, three operators, four operators modulated straightforward and four operators

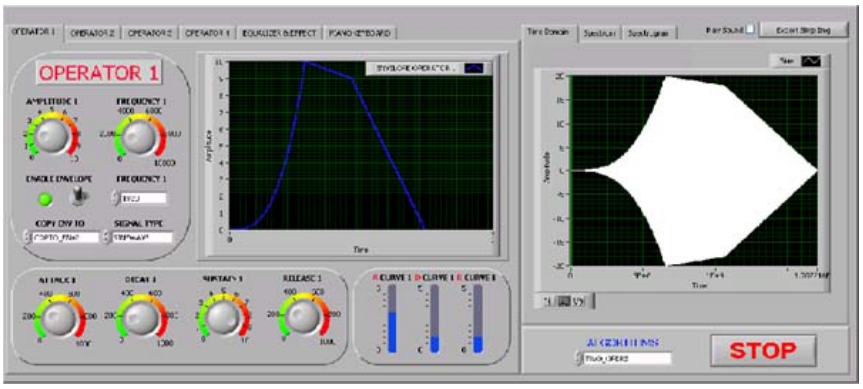

FIGURE 7.10: Front Panel of the application

that actually consist of a pair of two operators. This part can be expanded in future to have more possibilities for producing sounds. The second part of the application is the envelope.

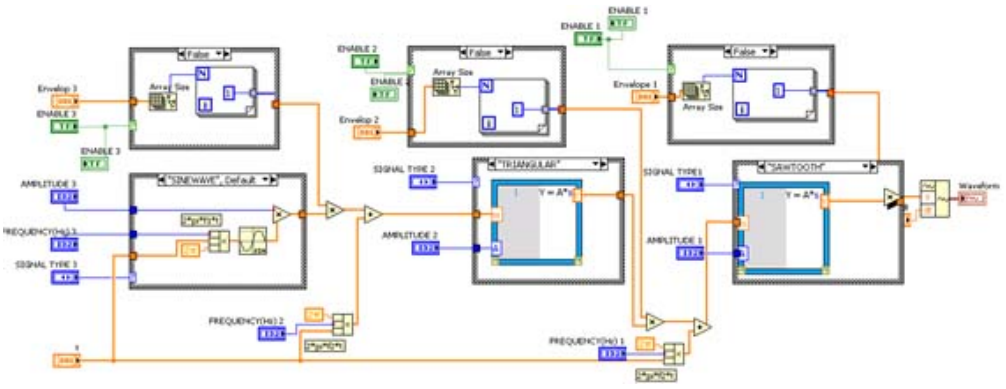

FIGURE 7.11: The algorithm for three operators

It is used to modify the shape of the signal so that the output can have different frequency components. The third is programmed to produce more flexibility for the application. With this, user can choose specific range of frequency of the output signal (using the equalizer) and also can make distortion (using the effect part). The last one is writing data part, which is discussed in "LabVIEW program for writing data", see section 7.5.1 above. More detail about the application, subVIs is available in appendix part in this report.

7.6 Conclusion

This report discusses a number of things, which are needed to take into account when working with UART controller of the DSP processor and LabVIEW program. From my point of view, despite the fact that each part plays a different role to make the application work properly, the most important thing is that the data (a string of characters) need to be define completely the same in LabVIEW User Interface and UART interrupt service routine. Otherwise, the data that DSP Processor wants to receive for processing is not expected.
To improve quality of communication between LabVIEW User Interface and DSP board, context switch of UART interrupt needed to be considered. However, because of time and the limited ability, it could be studied in the future.

Chapter 8

Critical reflections

Written by Bjorn Deraeve

8.1 Implementation problems

In chapter 6 we already explained how the program on the SHARC processor is designed. Here we will have a closer look at some problems encountered during the development.

8.1.1 Communication with the Labview interface

As explained in chapter 7, the parameters read from the interface are interrupt driven. As a consequence the program on the SHARC processor does not receive values if the user doesn't change any particular input on the interface.
For normal use this is not a problem since the values are initialized correctly when the program is started. However during the debugging process this often led to confusion and searching on problems in the code that actually weren't there.

8.1.2 Combining different program parts and general programming issues

Including smaller topics like the envelopes, effects and equaliser into the main program sometimes led to major problems. Next to the inevitable minor adjustments in the program parts also the interaction with the interface caused problems. However, most of these problems were small mistakes that could easily be fixed. Nevertheless not all team members succeeded into fixing those little errors.
Sometimes it was necessary to change some details of the main program in order for the subprograms to work fluently. However, in order to do so efficiently one must at least understand the template we received at the beginning of the project and think well before changing stuff. There had to be intervened several times because a team member unknowingly stopped the whole program instead of just disabling his own algorithm. For example, as explained in chapter 6, processors on embedded systems need a `while(TRUE)` loop. Another useful thing to realize is that the periodic functions generating the waveforms need a variable that keeps track of the time. Updating this time in a local variable does not have the same effect like updating the global variable and again stops the program!
Also the modulo %-operator is common in DSP algorithms and all team members should its meaning. Looking this up with google would have been a great idea!

Finally before asking to include a subprogram into the main program the smaller part must first have been debugged for syntax errors (since notepad doesn't highlight such errors).
Next, some other concrete programming issues are described:

Sampling frequency: The template we started from mentionned a sampling frequency of 48 kHz. However that project used a stereo audio channel which meant the samples were sent to the AD1835A at a rate of 96 kHz. Because we filled the algorithm's frames with mono channel samples (instead of two samples per calculation) our sampling frequency in software matched the sampling frequency of the hardware.

Modulation: Enabling modulation caused some minor timing problems, however this could simply be fixed by passing the function's argument by value instead of by reference.

Envelopes: There were several problems with the envelope. In a first stage nothing worked and a trick must be implemented to create envelopes with a duration longer than one second. To do so the function received a counter in its arguments.
The updating of this counter was at first done in the main synthesize sequence, when the program's time was an integer multiple of the sampling frequency. However because of the fixed frame length of 1024 samples this happend only every 4 seconds. To avoid this the updating of counter was originally moved to the create signal function. This has no influence and even made things worse actually. The problem with placing the flag here was that if modulation is enabled the flag updated several times. However this whole time the counter argument worked and the envelope refused working even without modulation.
The underlying cause was that the envelope function made use of the wrong envelopes in memory so the parameters had no influence. Unfortunatelly not all team members were able to find such mistakes
Finally to fix all little problems the envelope received its own space in memory for keeping track of time. So now this isn't done anymore in the synthesize function but in the envelope itself. Because now the time is obviously updateted instaneously all problems are fixed.

8.1.3 Connecting the DSP board and use of the IDEs

There were numerous of problems with Analog Devices' IDE. Personally I could not connect with the board on my normal Windows. Running the program from within a virtual machine on that same computer seemed to magically solve the problem (after a few years of internetting). If we have to invent an explanation I'd say that Sony tampered with the OS' USB drivers to make them more energy efficient and thus not supporting the DSP board.
Getting VisualDSP++ and Labview to run fluently also took a lot of time. For VisualDSP++ an activation code had to been registered. This became a problem since by the end of the project we got warnings that the activation would expend in less than a few days. Another issue with this activation code was that in week 10 a team member asked for it while it had already been e-mailed to all team members in week 4.
VisualDSP++ also seemed to be a very hard program to work with since a team member needed step-by-step click instructions for how to activate the editor's line numbers.
For Labview there were no valid licenses at all so the program had to be reinstalled a few times. Also to use all aspects of the interface several extra Labview plugins needed to be installed and again the only way to do is by completely reinstalling Labview, this became more or less a routine.

8.1.4 Extreme programming

Due to the tricky situation with the interface several problems asked a lot of time to get fixed or didn't get fixed at all. For those harder to find errors a second pair of eyes would have been welcome.
Though in an attempt to improve the overal impression of the project it was best to do have look at those extra problems and happily been able to fix some of them. *However it cannot be that there is only one person responsible to make all things work properly, the lack of a good programming partner was big. With such a partner a lot of improvements could have been made, some of them are discussed in section 8.4.*

8.1.5 Latex

Latex' standard color package does not support the color pink. My disappointment was huge, after heavy consideration I will declare this as the biggest lack encountered during this EE5-project.

8.2 Team problems

Next to common problems teams sometimes encounter like poor communication, group thinking and difficulty making decisions we also suffered from other problems. They are described shortly below:

- **Major issues**
  - Lack of basic programming knowledge
  - Update files to the shared folder
  - Lack of inspiration for simple program features
- **Minor issues**
  - Update timesheets in time
  - Labview vs. C distinction
  - Give feedback on the programs: this only happened for the interface
  - Chaotic organisation of files in the shared folder

8.3 Achieved activities

Student	Hours
Bjorn Deraeve	210
Frederik De Greef	140
Kenneth Piot	140
Thuy Pham	180

Table 8.1: Project logbook

Student	Topic
Bjorn Deraeve	Admin: Meeting reports Labview: Connect the basic interface C: General program structure C: Signals and modulation C: Import and adapt (fix) the other programs C: Debug and fix envelope program Matlab: plots for report Labview: Bessel functions LaTeX: cls, bib and tex template, chapters 1,2,5,6,8
Frederik De Greef	C & Matlab: effects C & Matlab: equaliser Report: chapter 4
Kenneth Piot	C: Envelopes Labview: Piano interface Report: chapter 2
Thuy Pham	Labview: full interface C: Configuration of UART communication Labview: waves and specta Labview: envelopes and graphs Labview: equaliser Labview: piano interface Report: chapter 7

Table 8.2: Activities

8.4 Future work

This section summarizes some features that could add additional value to DSP Synthesizer.

Envelopes: It is no coincidence the piano interface is located in the chapter on the envelopes. Combining the piano keystroke with the envelopes should have been an easy to implement extension.

Equaliser: An extended equaliser would provoke the musicians' creativity more.

SDRAM: Being able to save created sounds and reuse them to modulate signals. Also prestored effects could be offered this way. Search how to initialize the chip, ...

SHARC-ADSP 21369 IRQs: Using the board's buttons to increase or decrease the volume. This is an interrupt driven event. Combining this interrupt with the other, time pressured interrupts, would have presented some interesting programming problems.

Hardware controller: Instead of using the labview interface use a real controller with a more advanced communication protocol. The board supports SPI, I^2C,...

Bibliography

[1] Analog Devices, Inc., *ADSP-21367/ADSP-21368/ADSP-21369 SHARC Processor Data Sheet*, a ed.

[2] Analog Devices, Inc., *ADSP-21369 EZ-KIT Lite Evaluation System Manual*, 2.2 ed., September 2009.

[3] Analog Devices, Inc., *ADM3202/ADM3222/ADM1385 Data Sheet*. Line Drivers/Receivers.

[4] Analog Devices, Inc., *Using the UART Port Controller on SHARC Processors*, ee-296 ed.

[5] Maxim, Inc., *Determining Clock Accuracy Requirements for UART Communications*.

[6] Analog Devices, Inc., *AD1835A Data Sheet*, a ed., 2003.

[7] Analog Devices, Inc., *ADSP-2136(7/8/9) SHARC Processor Hardware Reference*, 1.0 ed., August 2006.

[8] Analog Devices, Inc., *Using the Expert DAI for SHARC Processors*, ee-243 ed., June 2010.

[9] P. A.Blume, *The LabVIEW Style Book*. Prentice Hall, 1 ed., March 2007.

[10] R. W. Larsen, *LabVIEW for Engineers*. Prentice Hall, 1 ed., February 2010.

[11] J. G. Proakis and V. K. Ingle, *Digital Signal Processing Using MATLAB*. Global Engineering: Christopher M. Shortt, 3 ed., August 2006.

[12] J. Chowning and D. Bristow, *FM Theory and Applications*. Shimomeguro Meguroku, Tokyo, Japan: Yamaha Music Foundation, 3 ed., 1986.

[13] K. Fujimoto, "How to make and play sound with matlab." http://www.h6.dion.ne.jp/ ff-f/old/technique/auditory/matlab.html.

[14] J. Thomas, "Making a studio pt.3," May 2009. http://en.audiofanzine.com/homestudio/editorial/articles/ma a-studio-pt3.html.

[15] T. Zuckerman, "Pitches, in hertz, produced by a keyboard instrument with 88 digitals, tuned to equal temperament." http://shakahara.com/pianopitch2.php.

[16] K. Eneman, "Coursebook on digital signal processing," 2011-2012.

[17] G. Read, "Synth secrets, part 10: Modulation," February 2000. http://www.soundonsound.com/sos/feb00/articles/synthsecrets.htm.

[18] G. Read, "Synth secrets, part 12: An introduction to frequency modulation," April 2000. http://www.soundonsound.com/sos/apr00/articles/synthsecrets.htm.

[19] G. Read, "Synth secrets, part 13: More on frequency modulation," May 2000. http://www.soundonsound.com/sos/may00/articles/synth.htm.

[20] A. D. Inc., "Digital signal processing 101- an introductory course in dsp system design." http://www.analog.com/en/processors-dsp/processors/beginners_guide_to_dsp/fca.html.

[21] E. W. Weisstein, "Fourier series - sawtooth wave." http://mathworld.wolfram.com/FourierSeriesSawtoothWave.html.

[22] E. W. Weisstein, "Fourier series - square wave." http://mathworld.wolfram.com/FourierSeriesSawtoothWave.

[23] E. W. Weisstein, "Fourier series - triangle wave." http://mathworld.wolfram.com/FourierSeriesSawtoothWave.html.

[24] E. W. Weisstein, "Bessel function of the first kind." http://mathworld.wolfram.com/BesselFunctionoftheFirstKind.html.

[25] P. Burk, L. Polansky, D. Repetto, M. Roberts, and D. Rockmore, *Music and Computers: A Theoretical and Historical Approach.* Key College Publishing. http://music.columbia.edu/cmc/MusicAndComputers/.

Appendix A

Bessel functions and spectrum examples

A.1 Modulation index I = 0

$J_0(0) = 1$ (carrier component) $\qquad$ $J_n(0) = 0$ for n ≥ 0 (sidebands)

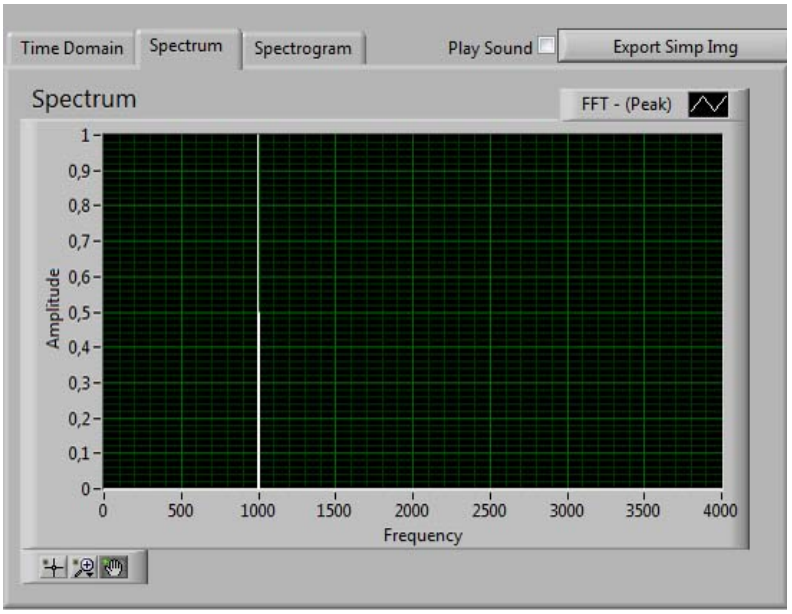

FIGURE A.1: Spectral diagram of a pure 1000Hz signal. Naturally there are no sidebands.

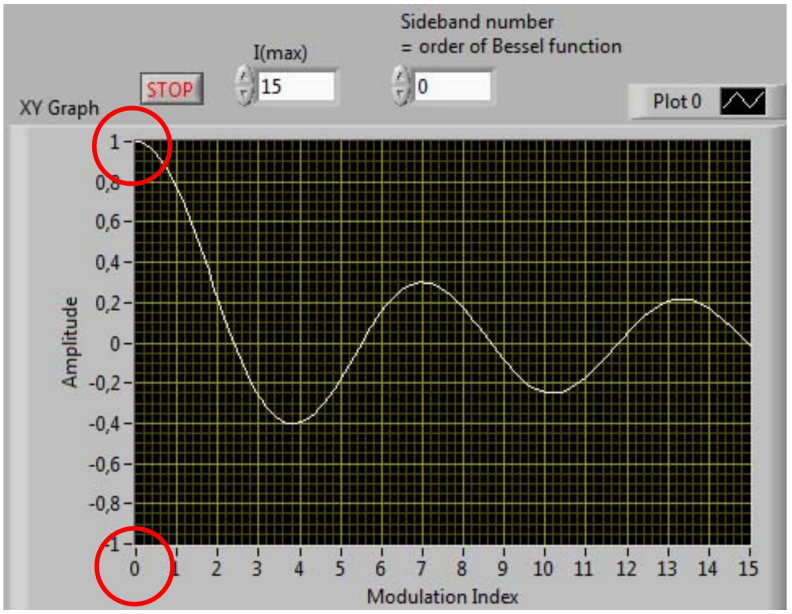

FIGURE A.2: Bessel function of the carrier component (order 0) Remark the amplitude is equal to 1 for I = 0: $J_0(0) = 1$

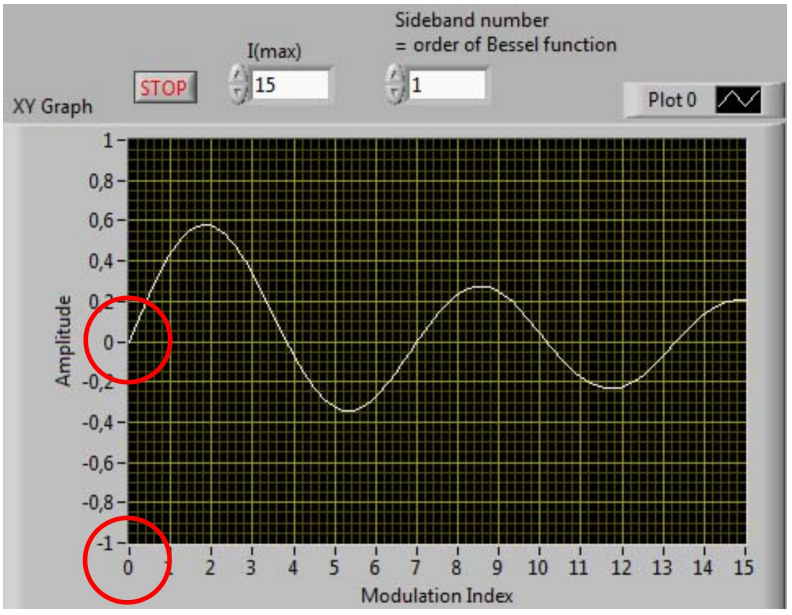

FIGURE A.3: Bessel function of the first sidebands component (order 1) Remark the amplitude is equal to 0 for I = 0: $J_1(0) = 0$

A.2 Modulation index I = 1

$J_0(1) < 1$ (carrier component begins to decline)

$J_n(1) \geq 0$ (sidebands begin to increase)

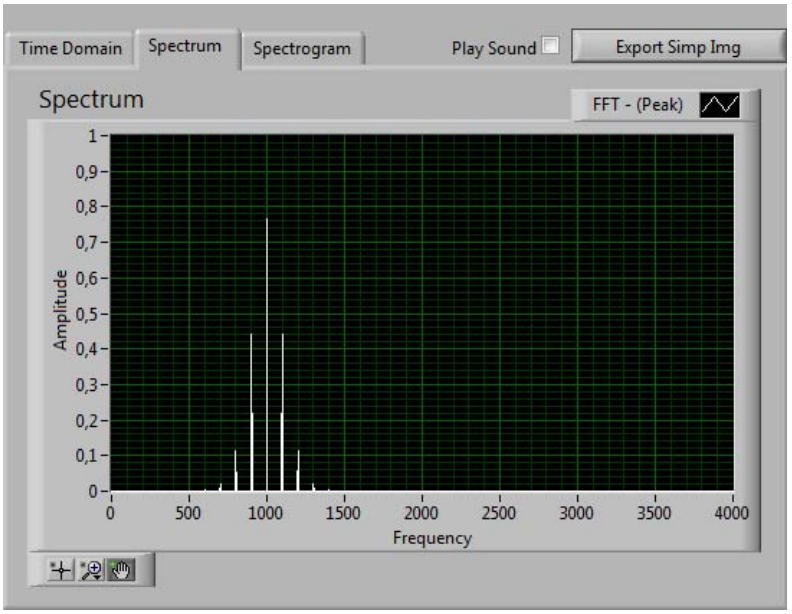

FIGURE A.4: Spectral diagram of a 1000Hz signal modulated with a 100Hz signal and I = 1.

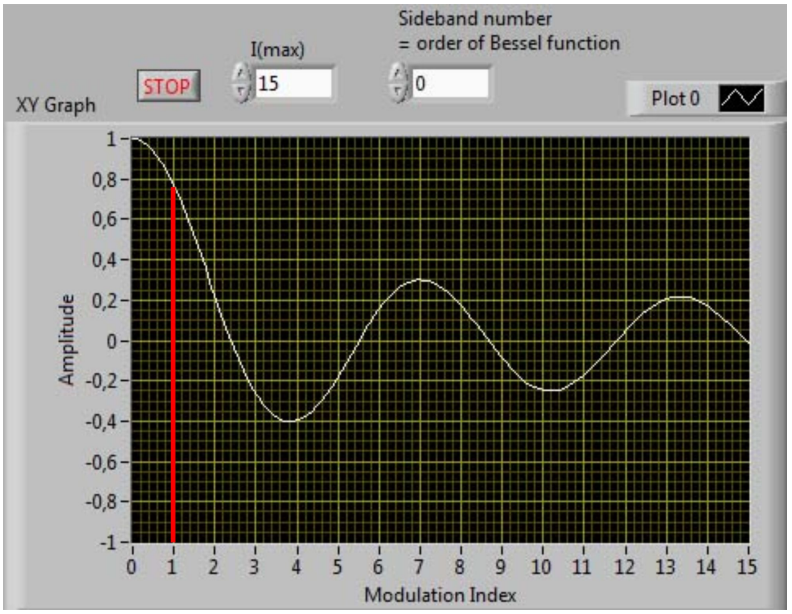

FIGURE A.5: Bessel function for the carrier (order 0) and I = 1 shows that the amplitude of the carrier component decreases.

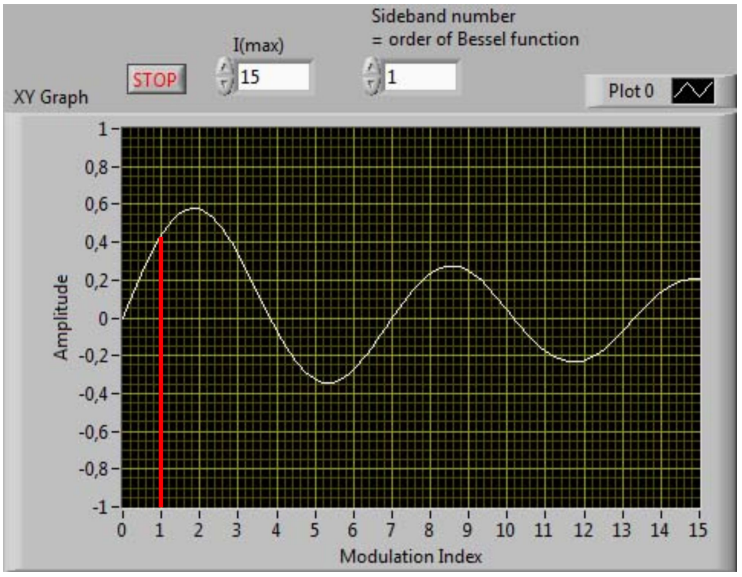

FIGURE A.6: Bessel function of the first sideband component (order 1) returns an amplitude of 0.4 if I = 1. Compare this value with figure A.4

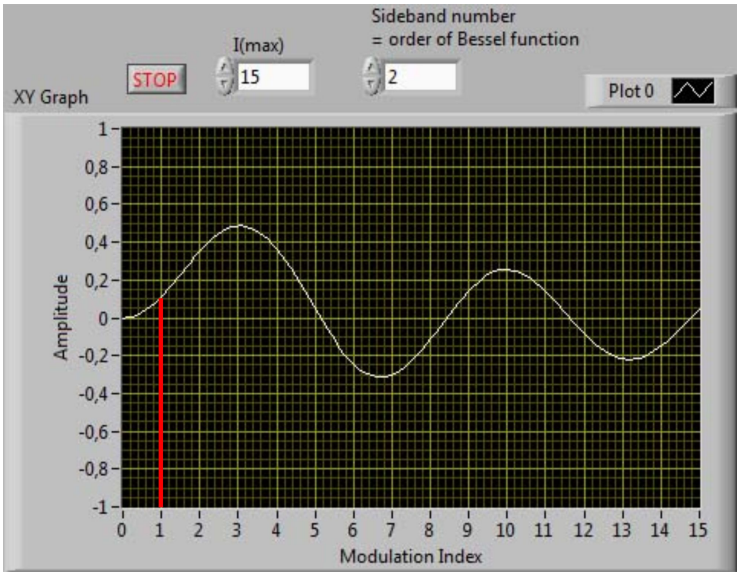

FIGURE A.7: Bessel function of the second sideband component (order 2) returns an amplitude of 0.1 if I = 1. Compare this value with figure A.4

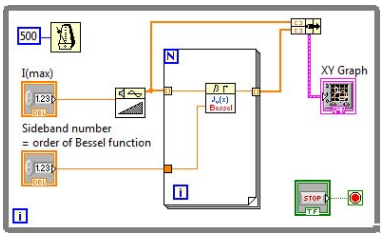

FIGURE A.8: Simple labview program to display Bessel functions of the first kind