NTNU – Trondheim
Norwegian University of
Science and Technology

Computational Materials: Experimental Platform

Ole Petter Skarre Lund

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Gunnar Tufte, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

In this research project we try to exploit computational properties of unconventional materials (materials usually not considered as a computational substrate). Such materials may offer computation at extreme low cost and may also enable us to do computation that is hard (or impossible) on a von Neumann stored program machine. Currently we explore possible computational properties of carbon nano tubes.

In 2010 a first version of a platform was made. This system consists of a PCB, including an Atmel microcontroller and a Xilinx FPGA, that acts as an interface between a PC and a micro electrode array. The array interfaces the material under investigation.

In this project the experimental platform will be extended. There are several possible directions. As such there is a possibility for several students pursuing different directions. Possible directions:

a) Extending the software, microcontroller and PC (mostly c-programming).

b) Extending the FPGA interface to the material (VHDL and c-programming).

c) Design of additional interface circuits between the FPGA and the micro electrode array (PCB-design, digital/analogue design).

Abstract

The field of *evolution in materio* is a branch of unconventional computing that uses artificial evolution to manipulate materials so that the emergent properties can be used for computation. Some materials may have very complex properties and this could be utilized to do computation faster and/or more energy efficient than today's computers. The purpose of this master's thesis is to extend an already existing prototype system, called Mecobo, that is used for evolution in materio. The prototype system is currently only able to apply digital signals to the materials, i.e. high and low voltage, but with this new extension it will also be able to apply an arbitrary waveform in addition to reading the response from the materials. The main components of the extension are digital-to-analog converters that together with an FPGA functions as a frequency synthesizer. An analog-to-digital converter is also used for sampling the response signal. Some initial experiments are presented, where the linearity/nonlinearity of the response signal is investigated when analog signals are applied to carbon nanotubes. The results shows that carbon nanotubes are able to influence the signals, as they pass through it.

Sammendrag

Forskningsfeltet *evolution in materio* er en gren av feltet ukonvensjonell databehandling som bruker kunstig evolusjon for å manipulere materialer slik at egenskapene som oppstår kan benyttes til å gjøre signal- og databehandling. Noen materialer kan ha veldig komplekse egenskaper og dette kan bli utnyttet til å gjøre beregninger raskere og/eller mer energieffektivt enn hva dagens datamaskiner kan. Hensikten med denne masteroppgaven er å utvide et allerede eksisterende prototype-system, kalt Mecobo, som blir brukt til *evolution in materio*. Prototype-systemet er foreløpig bare i stand til å anvende digital signaler på materialer, med andre ord høy og lav spenning, men med den nye utvidelsen vil det bli mulig å andvende vilkårlige bølgeformer i tillegg til å kunne lese responsen fra materialet. Hovedkomponentene i den nye utvidelsen er digital-til-analog konvertere som sammen med en FPGA fungerer som en frekvenssynthesizer. En analog-til-digital konverter blir brukt for å lese responssignalet fra materialet. Noen innledende eksperimenter presenteres, hvor lineæriteten/ikke-lineæriteten til responsen fra nanorør av karbon blir undersøkt, når analoge signaler blir andvendt. Resultatene viser at materialet er i stand til å påvirke signalene, idet de passerer gjennom det.

Acknowledgments

I would like to thank my supervisor Gunnar Tufte. His knowledge in signal processing, analog/digital electronics and evolvable systems has been very helpful during the process of designing and evaluating the system and in analyzing the results of the experiments. Also, thanks to Odd Rune Strømmen Lykkebø for helping me with the various problems that I encountered.

Contents

List of Figures

List of Tables

Acronyms

FPGA field programmable gate array

PCB printed circuit board

EA evolutionary algorithm

SPI serial peripheral interface

DDS direct digital synthesizer

NCO numerically controlled oscillator

INL integral nonlinearity

DNL differential nonlinearity

DAC digital-to-analog converter

ADC analog-to-digital converter

PA phase accumulator

PAC phase-to-amplitude converter

SNR signal-to-noise ratio

SFDR spurious-free dynamic range

MSB most significant bit

LSB least significant bit

FSM finite state machine

LUT look-up table

ROM read-only memory

1 Introduction

This master's thesis is a about unconventional computation and unconventional computational machines. It is a part of a Future and Emerging Technologies project called NAnoSCale Engineering for Novel Computation using Evolution (NASCENCE). This project is funded by EU's Seventh Framework Programme (FP7). From the projects website we can read that: *"The aim of this project is to model, understand and exploit the behavior of evolving nanosystems (e.g. networks of nanoparticles, carbon nanotubes or films of graphene) with the long term goal to build information processing devices exploiting these architectures without reproducing individual components. With an interface to a conventional digital computer we will use computer controlled manipulation of physical systems to evolve them towards doing useful computation."* [nas]

1.1 Unconventional Computation

As Moore's law is expected to collapse sometime in the future, we need a new way to do computation to be able to continue the increase in speed and energy efficiency that we see in computers today. In the past, most of the improvements came from new microarchitectures that exploited instruction-level parallelism and from higher clock speeds. Unfortunately, these improvements cannot continue indefinitely because of the serial nature of the uniprocessor and the heat it generates at high frequencies. So for the last ten years, the processor industry has shifted the focus from single-core processors towards multi-core processors to take advantage of data-level and thread-level parallelism [HP12]. Although this is an important step, it is not enough to overcome the ever-increasingly demand for more speed and energy efficiency. Projects like NASCENCE is hopefully

the start of a transition from conventional to unconventional computers in areas that needs the computational strength of a unconventional computer to solve difficult problems faster or where low energy consumption is crucial. The goal is not necessarily to replace all conventional computers, but instead be a supplement.

Today, the word computation is something that most people associate with the traditional electronic computer based on von Neumann's stored program machine, consisting of silicon transistors. However, a lot of physical processes can be viewed as a computational process. Maybe the most apparent one is the biological process where an organism is developed from a zygote to a fully grown individual [KB03]. Another example is the Belousov-Zhabotinsky reaction. This is a nonlinear oscillating chemical reaction that creates complex patterns. It is an example of non-equilibrium thermodynamics that can be used to manipulate and process information and it is a subject of study in the field of reaction-diffusion computing [JHJ10]. A common denominator for these physical processes is that there is no central control. The system where the computation takes place is distributed and consists of many small parts that interacts locally with the nearest neighbors [SLHR06]. The system is parallel by nature and it often exhibits complex and non-linear properties that emerges from this interaction. This could be very useful. For example, small perturbations may cause great effect on the system as opposed to a system with linear properties where the effect usually is proportional to the perturbation [JHJ10].

We want to exploit this computational capability to do computation much faster and/or more energy efficient than what is possible with a conventional computer. Problems that are considered impossible to solve on the traditional computer may be solvable using such an unconventional computer. These systems may also offer other desirable properties such as robustness and adaption because of its decentralized structure [Hey]. There is no central processor or control that needs to function correctly so that the rest of the system can work correctly. All the small parts contribute equally and if some of them are disturbed or destroyed, the rest of the system may continue to function properly together. This is an example of robustness. Adaption may help the system to function correctly in different environments, by reorganizing and stabilize in a new state while maintaining its original operation or function.

While a conventional computer can be controlled and programmed using a large number of different programming languages that works as digital ab-

straction layer for a processor that we know all about, this is not the case with unconventional computers. They are often distributed and consists of a vast number of processing elements that together creates the emergent computational properties. The implementation could be a number of different physical and distributed systems where there is no obvious way to specify the systems operation or function. In such a distributed system, the programming techniques can be separated into two categories [Sip02]. The first one is *direct programming* where the programmer completely specifies the interaction/connection between the parts and each parts function, etc. If we do not have any knowledge about the systems internal structure and we don't know how the desired properties emerges or if it is to complicated to use direct programming, we could use an *adaptive method* such as artificial evolution [Dow10]. In this method, the system is specified to a certain extent before the adaptive process of evolution develops or produces the functionality we want.

1.2 Evolution In Materio

Evolution in materio is a field of research that uses artificial evolution to control or manipulate materials to do some form of computation. The material can be solid, liquid or gas. The idea is to use artificial evolution to change the physical or electrical configuration of the material and exploit the emergent properties to do the computation. The physical system, in this case some type of material, may have very complex and unpredictable properties. It may be very hard or even impossible for humans to understand how the system works and we therefore need a way to overcome this problem. The power of evolution is that it does not have to know anything about the problem it tries to solve. We can therefore use artificial evolution as tool to solve complex problems we humans have little or no knowledge about [Har06]. By mimicking nature, we can configure the system without knowing how it works internally. We can view it as a black box. The material we are currently experimenting with is carbon nanotubes. We now little about the internal workings of the material and how its properties emerges and therefore we use artificial evolution to investigate this.

Figure 1.1 shows how one may visualize evolution in materio. The physical or electrical configuration of the material is changed and an input signal is processed by the material. The modified output signal is tested and its fitness is measured according to a fitness function. The fitness score

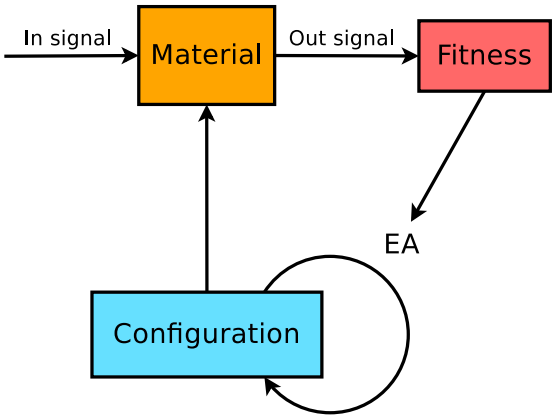

Figure 1.1: Visualization of evolution in materio.

is then used by the evolutionary algorithm to change the configuration using different genetic operators (such as mutation and crossover). The evolutionary algorithm moves through the solution space in search for a possible solution.

Evolution in materio has been successfully applied to several problems. For example, Simon Harding has used the technique to evolve a robot controller using liquid crystals as material [Har06]. Another example is Adrian Thompson's experiment where he evolved a tone discriminator circuit on a field programmable gate array (FPGA). The setup of the experiment is shown in figure 1.2. The FPGA acts as the material and the desktop PC runs the evolutionary algorithm and applies the different configurations to the FPGA. The tone generator is used to test the circuit and the integrator is for measurements that is used in the fitness calculation. The evolved circuit could discriminate between two different tones and it exploited the inherently analog nature of the silicon transistor [Tho96]. More about these experiments in chapter 2.

1.3 Thesis Outline

This master's thesis describes an extension to the evolution in materio prototype system developed by Odd Rune Strømmen Lykkebø, as part of his master's thesis [Lyk10]. The system is currently only able to apply digital signals to the material, but we also want it to have the ability to apply dynamic signals, e.g. a sine wave or triangle wave, and sample the output response signal. The reason for this is simple; we are searching for an

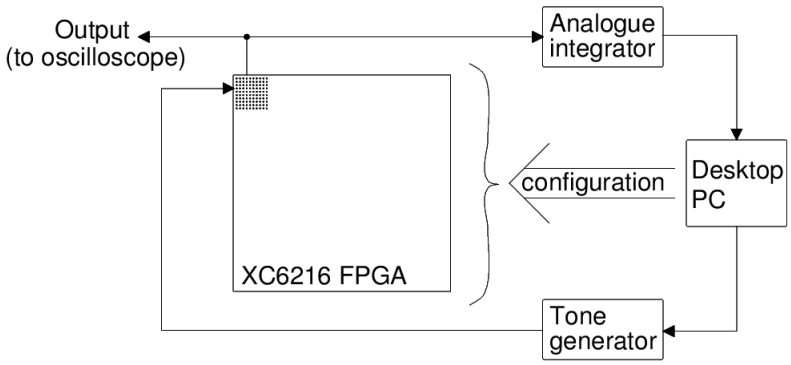

Figure 1.2: Adrian Thompson's experiment setup. Illustration taken from [Tho96].

electrical configuration(s) that manipulates the material so that computational properties emerges. By extending to analog signals we get a much larger set of possible configurations. The extension is therefore a printed circuit board with digital-to-analog and analog-to-digital converters that connects to the material, in addition to necessary software and hardware needed for the communication between the prototype system and the new extension.

Chapter 2 describes some previous work done in this field and a brief overview of the work done by Odd Rune Strømmen Lykkebø. Chapter 3 gives and overview of the system, both the original system and the new extension. In chapter 4 we dive into the details of the system and look how it really functions, its capabilities and its limitations. The testing and evaluation is described in chapter 5. Chapter 6 presents initial experiments where we test the response of the material. Chapter 7 ends this master's thesis with a conclusion.

2
Background

This chapter presents background information on the earlier work done in the field of evolution in materio. In addition, it gives an overview of the existing platform that this thesis is an extension of.

2.1 Previous Work

2.1.1 Pioneer Work

The English cybernetician and psychologist Gordon Pask was a pioneer in the fields of evolvable and self-organizing systems. In the 1950's he conducted experiments where he tried to evolve a complex system that was capable of perceiving sound or magnetic fields. The system's parts was not fully specified and the thought was that the system was able to create its own "relevance criteria", meaning that it would discover on its own the observables that was required to perform a given task [Car93]. This was a very different engineering approach at the time. Usually, each component in a system has its position and behavior specified.

Gordon Pask used an electrode array that was suspended in a dish that contained an acidic aqueous metal-salt solution. This type of solution has the potential to behave in a very complex way [Car93]. By applying current on the electrodes, wires can self-assemble in the metal-salt solution. Figure 2.1 illustrates the experimental system with wires forming between electrodes in the chemical solution at the bottom. A network of these wires forms and together they can perform signal processing. Gordon Pask's system did manage to discriminate between 50 Hz and 100 Hz tones.

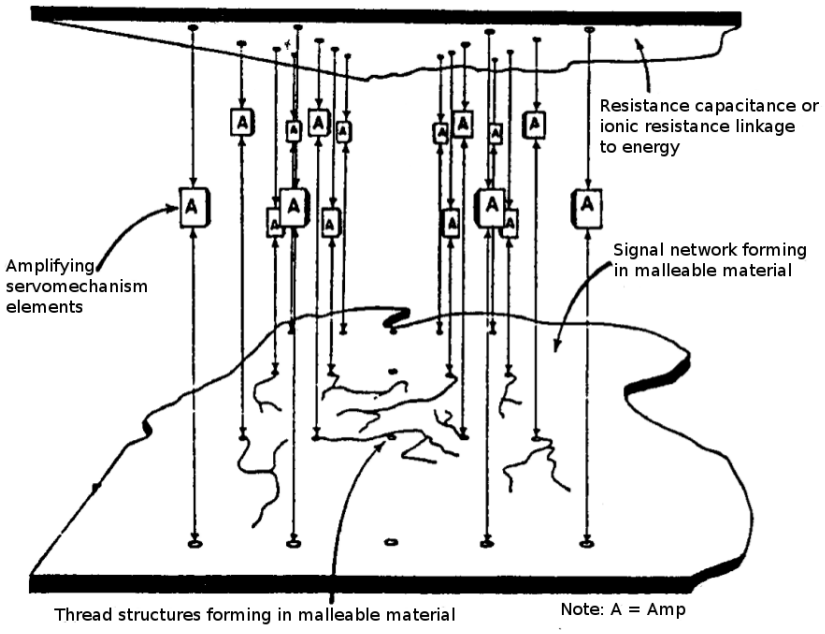

Figure 2.1: Schematic Gordon Pask's electrochemical system. Illustration taken from [Car93].

To train or program the system so that the desired behavior emerged, Pask used a set of resistors and changed their values using probabilities. This can be seen as a precursor to or a crude version of an evolutionary algorithm [Har06]. So in essence, what Gordon Pask did in his experiments is very similar to what we today call evolution in materio.

2.1.2 FPGA Tone Discriminator

Closely related to evolution in materio is the field of *intrinsic hardware evolution*. It uses artificial evolution to evolve electronic circuits. Adrian Thompson was in the mid 90's the first to use artificial evolution to evolve an FPGA configuration that could discriminate between two square waves of 1 kHz and 10 kHz without any clock source [Tho96]. The evolved circuit consisted of 10×10 logic cells that were connected together. A number of cells were removed without affecting the behavior of the circuit. Figure 2.2 shows the resulting functional part of the circuit. The gray boxes are cells that cannot be removed without affecting the behavior of the circuit. This is strange since these cells cannot influence the output via a connected path. These cells have some effect on the other cells, but

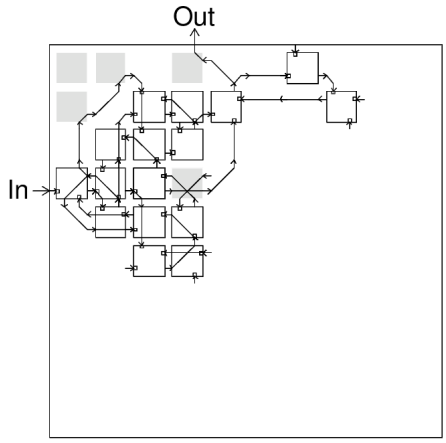

Figure 2.2: Functional part of the evolved circuit with gray cells that cannot be removed without affecting the behavior. Illustration taken from [Tho96].

not through wiring. Thompson suggested that this effect could come from power-supply wiring or electromagnetic coupling.

Even though we humans think of the FPGA as a digital device, under the hood it is analog, just as the rest of the world. The evolutionary algorithm don't care about this abstraction and exploits everything to find a candidate solution. In this way, Thompson's experiment can be viewed as evolution in materio since it exploited the analog properties of the silicon transistors in the FPGA to evolve a working circuit.

2.1.3 Liquid Crystals

Another more recent example of evolution in materio is the work done by Simon Harding where he used liquid crystals as material [Har06]. The hardware platform he used for these experiments was a liquid crystal evolvable motherboard. It was based on previous motherboards that has been used for intrinsic hardware evolution. An example of such a previous motherboard is shown in figure 2.3. This motherboard was constructed by Paul Layzell. He was motivated by Adrian Thompson's work on intrinsic hardware evolution, but he wanted to be able to monitor the whole system during the evolutionary process. Since you cannot monitor signals inside an FPGA he developed a system where you can do that. The figure shows many switch arrays where each crosspoint can be controlled individually. At the sides, discrete components can be connected and thus it is possible

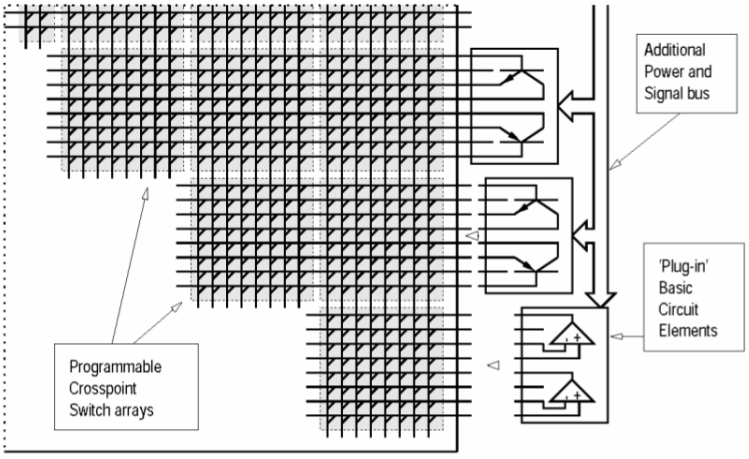

Figure 2.3: Evolvable motherboard constructed by Paul Layzell. Illustration taken from [Har06].

to monitor the evolutionary process. Several successful experiments were conducted.

Simon Harding created his evolvable motherboard by connecting an off-the-shelf liquid crystal display to switch arrays as shown in figure 2.4. The switches are controlled by a computer and this computer also reads the response from the liquid crystal display. By using an evolutionary algorithm that applied voltages to the connections he was able to evolve complex material behavior. Among the successful experiments was the evolved real-time robot controller that could navigate around in a simulated environment. Signals from two sensors was fed to the liquid crystal and the response was used to control the robot using two motors as shown in figure 2.5. The sensors measured the distance to an obstacle and was instructed to output a square wave signal with a frequency that was proportional to the straight line distance. Then, the liquid crystal did some form of signal processing with the square wave signals coming from the sensors and controlled the motors by setting the voltage to high or low. According to [Har06], this is the first time liquid crystal has been used to control a robot.

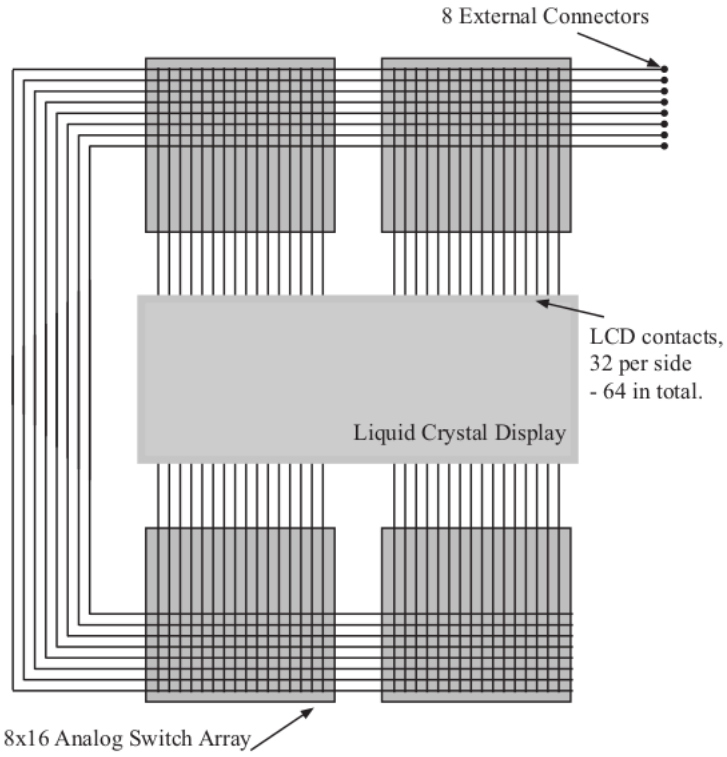

Figure 2.4: Liquid crystal evolvable motherboard constructed by Simon Harding. Illustration taken from [Har06].

2.2 Existing Platform

As stated in chapter 1, the work presented in this thesis is based on Odd Rune Strømmen Lykkebø's master's thesis. In his thesis, he designed and implemented a prototype system for evolution in materio. The overview of the system is shown in figure 2.6. It consists of a host computer, a microcontroller, an FPGA and a material bay for interfacing with the material. The main part in the system is called Mecobo, and this is where the microcontroller and the FPGA are located. It connects the host computer with the material bay. Figure 2.7 shows what Mecobo looks like. The laptop acts as the host computer where the evolutionary algorithm runs. On the right we have Mecobo and in the middle is the material bay that can contain different materials.

To communicate with Mecobo and set a voltage pattern on the electrodes in the material bay you have to use a C library specifically made for this system, called *libEMB*. This library contains different functions to

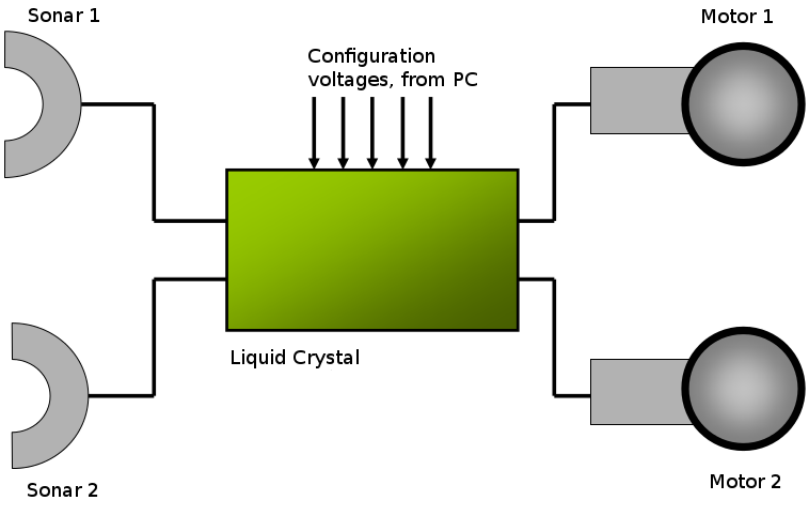

Figure 2.5: Setup for the evolved robot controller. Illustration taken from [Har06].

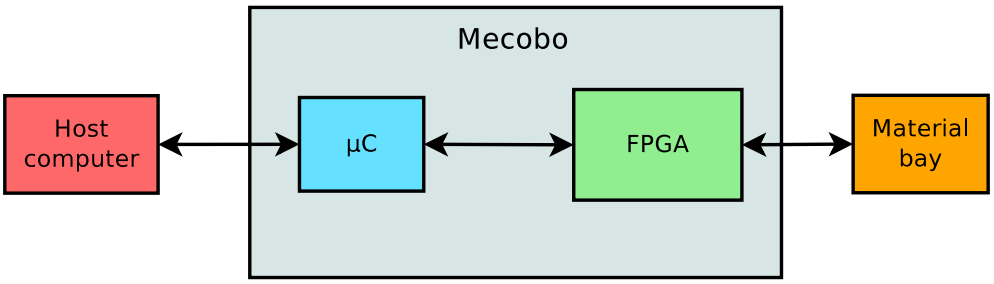

Figure 2.6: System overview without the new extension.

set and read the electrodes, in addition to other functions. Calling these functions will cause the host computer to communicate with the microcontroller on Mecobo which in turn will communicate with the FPGA. When Mecobo has done its work, the function that was called on the host computer returns. An example of how to use the `setPattern` function is shown in listing 2.1. The function has two arguments, where one (`pinconfig_t *config`) specifies which pins is output and the other argument (`pattern_t *pattern`) specifies the pattern to apply to the output pins. As we can see, the use of this function is fairly easy.

Listing 2.1: setPattern code example

```
// Create a pin configuration with 2 pins
// Set both as output
pinconfig_t *config = init_config(2);
```

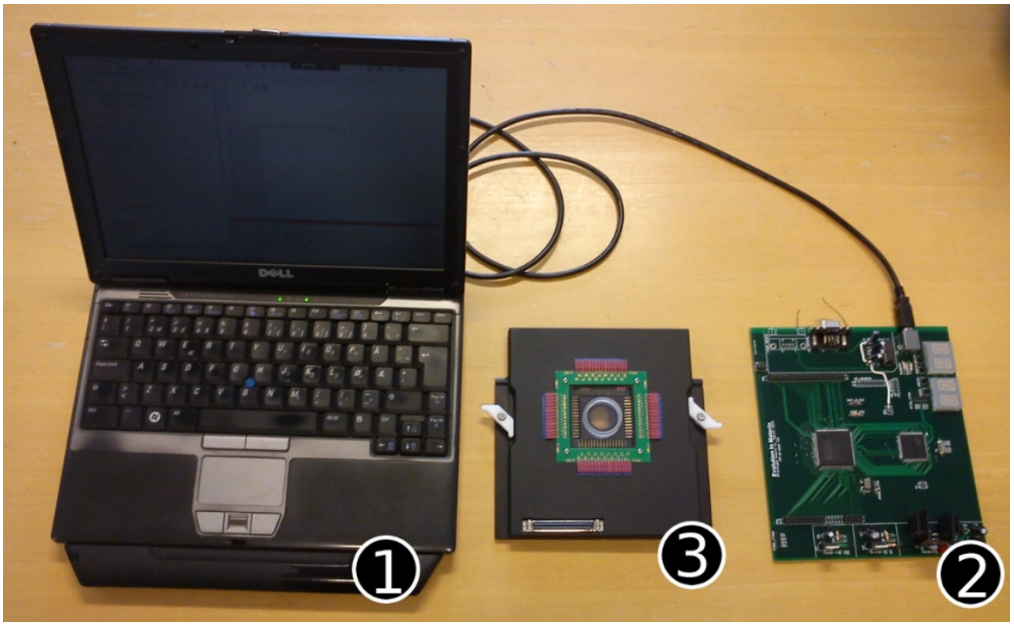

Figure 2.7: 1: Host computer. 2: Mecobo. 3: Material bay. Picture taken from [Lyk10].

```
set_pin_mode(0, PIN_OUT, config);
set_pin_mode(1, PIN_OUT, config);

// Generate a random pattern
pattern_t *pattern = generate_random_pattern(2);

// Set pattern
setPattern(pattern, config);
```

The material currently used in experiments is carbon nanotubes and it is shown in figure 2.8 This material was constructed by spreading out the nanotubes using a probability distribution. This causes the nanotube density to be nonuniform. The idea is that the nonuniform density will provoke nonlinear properties in the material. There are also plans to use the gold particles, shown in figure 2.9, sometime in the future.

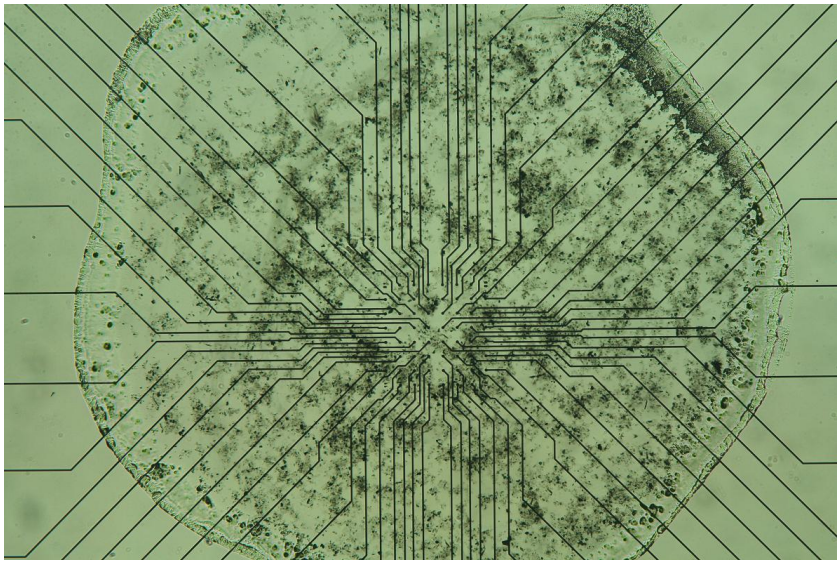

Figure 2.8: Nanotubes made of carbon. Electrodes for interfacing with the material is clearly visible as dark lines going towards the center.

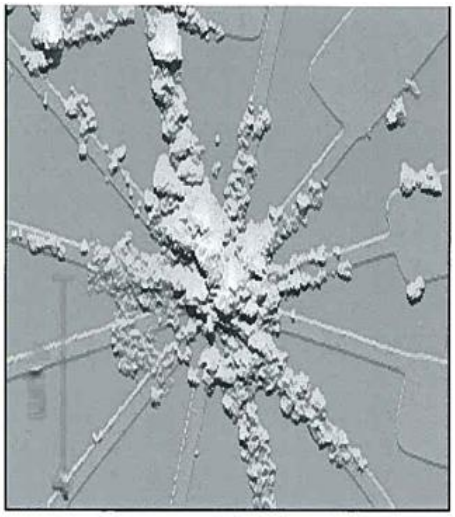

Figure 2.9: Picture of gold particles that will be used as material under test in the future.

3

System Overview

In this chapter we will explain the main parts of the original system and the new extension/daughterboard (just called analog board) and give an overview of its capabilities. Figure 3.1 shows where the analog board fits in the original system. As we can see from the figure, the analog board is connected to the FPGA on one side and to the material bay on the other side. This is illustrated in more detail in figure 3.2. The connectors shown in this figure is not necessarily what we actually use, but it gives us an idea of the connection between Mecobo and a daughterboard.

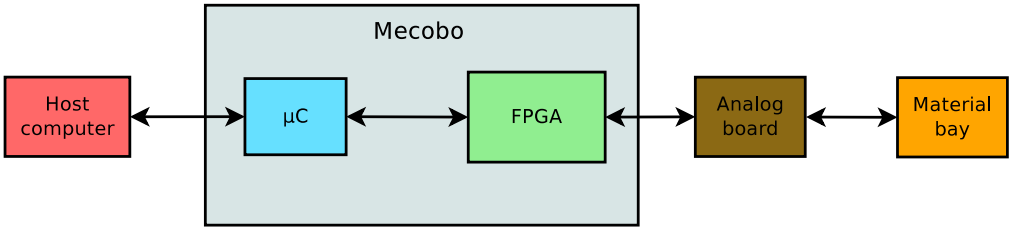

Figure 3.1: System overview, including the analog board.

3.1 The Experimental Hardware

The purpose of Mecobo is to be able to apply electric current to some kind of material. Since the FPGA is a digital device it can only set the voltage on the electrodes in the material bay to digital high or low, which decreases the search space for the evolutionary algorithm (EA). With the analog board however, we can choose between many more voltage levels. This will greatly increase the search space of the EA. Figure 3.3 shows the system we want and how the information flows. Here, we are using

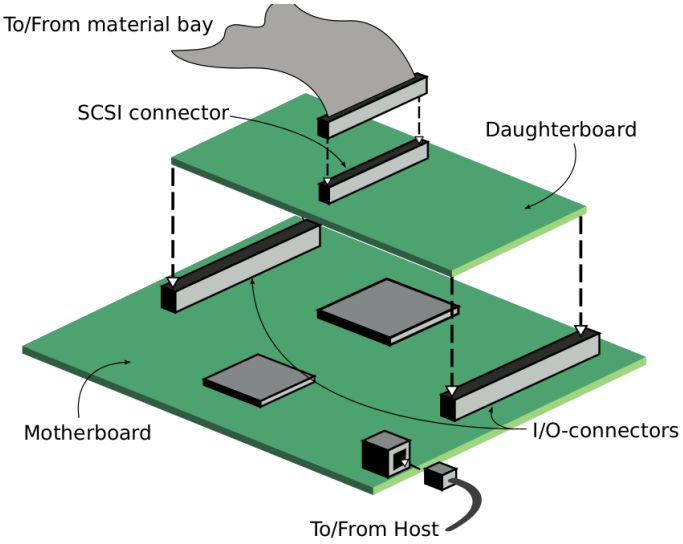

Figure 3.2: Mecobo and how it interfaces with the daughterboard. Picture taken from [Lyk10].

both analog and digital stimuli as the input signal or configuration. The stimuli is first set and then the response is read back and used as input to the fitness function in the EA that is running on the host computer. This is repeated until the EA terminates because of a predefined termination criterion that has been met, such as a fitness threshold, elapsed time or number of iterations or generations.

As a more detailed example, we can define two outputs from the analog board as input to the material and three outputs as a part of the configuration. We can also define one input to the analog board as the output from the material which we will use to measure the response from the material. Four pins on the FPGA might be define as the second part of the configuration. The goal could be to double the frequency of one of the input signals and sum the amplitude of the two input signals. Figure 3.4 illustrates this proposed experiment setup and its final configuration where the goal is met. We can see that the two input sine signals has the same fixed amplitude and frequency. The part of the configuration coming from the FPGA is expressed as a bit-pattern and in this case equal to 0100. The second part of the configuration is coming from the analog board. The two first signals is just static voltages. The third configuration signal is a regular sine wave. From the output signal we can see that the material has doubled the frequency of the input signals and summed their amplitude. To actually reach the goal function, we can use a (1+4)-ES (evolutionary

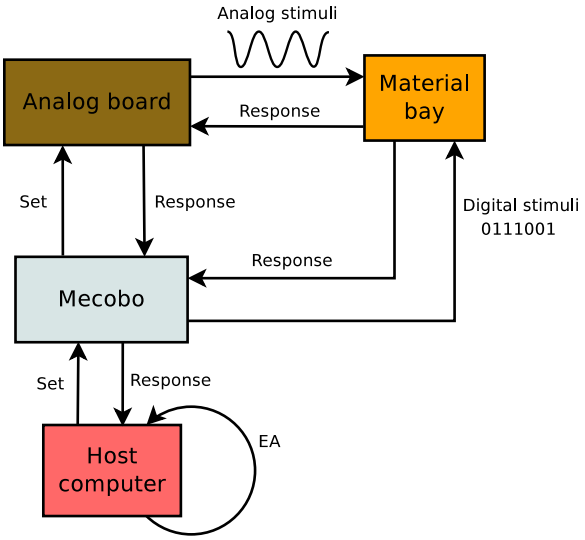

Figure 3.3: The prototype system and the main flow of information between the components.

strategy). This is a simple evolutionary algorithm with one parent and four children. The expanded view of the host computer in figure 3.4 shows the main steps in the EA. First, we generate a random population of 4 individuals. Each of these individuals has its own configuration that is applied to the the material. This configuration acts as the genome. An example of how it can be represented is shown in table 3.1. The configuration for the three analog inputs are expressed as the amplitude, frequency and phase of a sine wave. The configuration for the digital inputs are just a bit pattern. For each individual, the response from the material is read back and the fitness is calculated so that we can rank the individuals and select the best one to be the parent for the next generation. Choosing the right fitness function is often the hardest part in an evolutionary approach. It has to be defined properly so that it takes into account those parameters and properties that makes the algorithm converge to a good solution. After the selection process, the individual that was selected is now a parent and is used to generate four new individuals or children. This is done by using genetic operators to alter the parent's genome. The cycle continues until one of the individual's fitness value has reached a certain threshold.

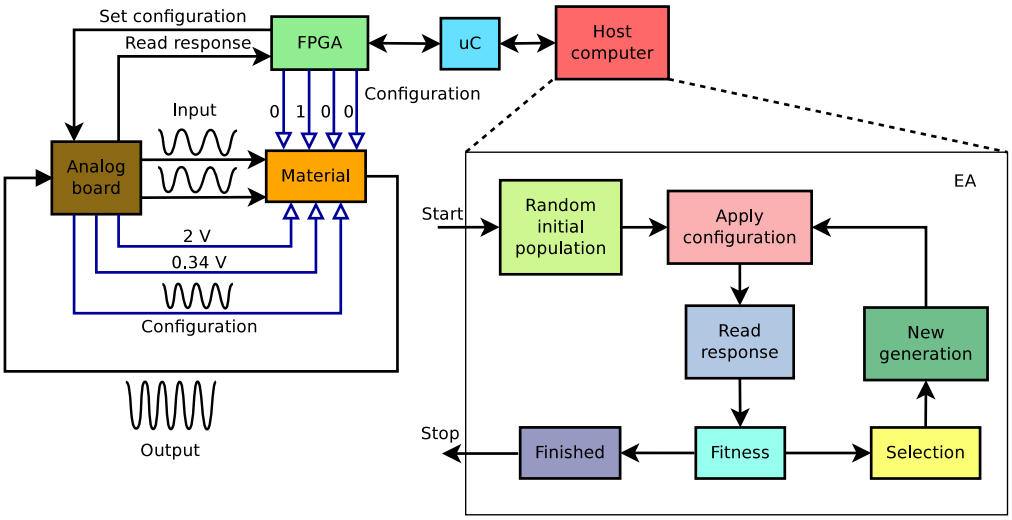

Figure 3.4: One possible experiment setup.

	Analog stimuli			Digital stimuli
	Amplitude (V)	Frequency (Hz)	Phase (°)	Bit pattern
1	1.23	100	0	0001
2	2.5	0	90	
3	0.34	587	245	

Table 3.1: Example genome for the evolutionary algorithm.

3.2 Extension

As already stated, we want to be able to apply multiple analog signals, both dynamic and static, and read the response from the material. We decided that 12 output signals together with 8 input signals are enough for now. We want four different wave types: sawtooth wave, sine wave, triangle wave and square wave. These waveforms are shown in figure 3.5. For these waves we want to control the frequency, amplitude and phase offset. This will greatly increase the search space since every new variable adds a new dimension to the search space. The frequency range requirement is quite loose, but a range from 0 Hz to several kilo hertz should be sufficient and the frequency resolution should be less than 10 Hz. The phase offset range should be from 0° to 360° with at least one degree resolution. The amplitude range should go from 0 V to a variable voltage level that we can change if we want to. The amplitude resolution should be high enough so that we get a smooth wave at high frequencies and amplitudes. Static

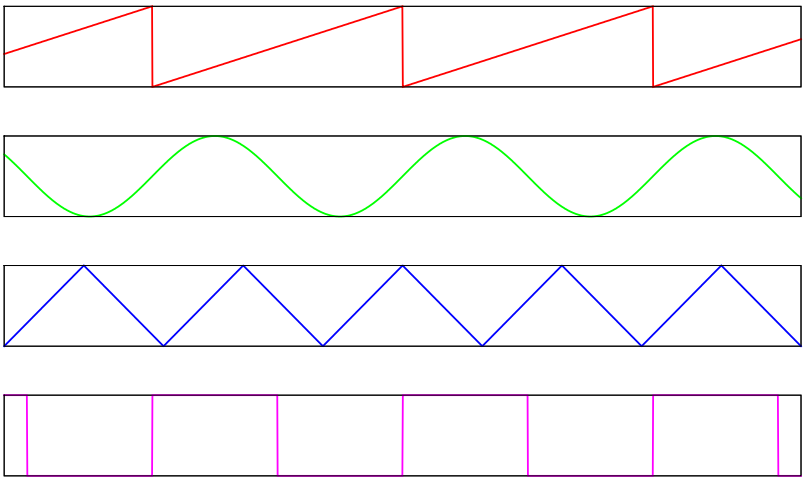

Figure 3.5: Waveforms we want the system to generate.

signals can be expressed as a sine wave of zero hertz and a phase offset. The sampling part of the system should be able to sample a dynamic signal with a frequency of several kilo hertz. Figure 3.6 illustrates the sampling of signals from the material. The response is converted from analog voltage to a digital code on request from the FPGA and then it is sent to the host computer via Mecobo. Because we don't know all the properties of the material or the configuration(s) needed to manipulate it, we define these somewhat loose requirements.

To reach the specified requirements, we need a suitable digital-to-analog converter (DAC) and a analog-to-digital converter (ADC). The converters are to be connected to the FPGA on Mecobo. Using the FPGA as a controlling unit gives us great flexibility in the design and it is easier to meet timing constraints. It also makes the pairing with the old design much easier. What we essentially want is an arbitrary waveform generator or a direct digital synthesizer (DDS).

3.2.1 Direct Digital Synthesizer

A DDS is a type of frequency synthesizer that can create arbitrary waveforms with a wide range of frequencies and phase offsets, using a single fixed-frequency oscillator [dds]. The DDS consists of a numerically controlled oscillator (NCO), a digital-to-analog converter and a low-pass filter, as shown in figure 3.7. The NCO is implemented on the FPGA while the

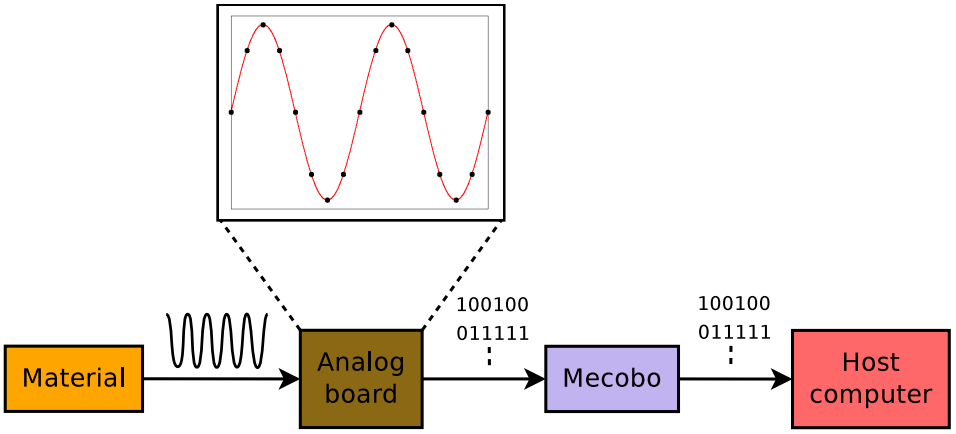

Figure 3.6: Sampling the response from the material.

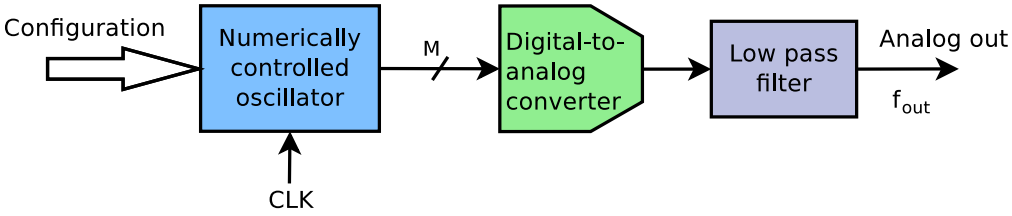

Figure 3.7: Main components of a direct digital synthesizer.

DAC is on the analog board. The low-pass filter is not on the analog board, but implemented as a separate part. By doing this, we can change the filter to another one with different characteristics. The next chapter will give a detailed explanation of the different components.

3.2.2 Signal Sampling

Generating waveforms is not enough. We also have to be able read the response from the material. So in addition to the DDS, we created a way to sample signals using an ADC. The design of such a system is less complicated than the DDS, but just as important. The FPGA controls the ADC and instructs it to send new samples, which the FPGA stores in a shared memory. The microcontroller can then read these samples and send them to the host computer.

4

Design and Implementation

This chapter explains in detail how the system works. The system must be able to set and read analog signals and the main parts on the analog board are therefore three digital-to-analog converters (DAC) and one analog-to-digital converter (ADC), where the DACs sets the analog signals and the ADC reads the response. To control the analog board we use an FPGA which gives us great control and flexibility when it comes to the design of the DAC and ADC controllers and the design of the NCO. Communication between the FPGA and the analog board happens constantly during operation since the NCO always has a new wave sample for the DAC(s) and because the FPGA instructs the ADC to sample the response signal when the user of the system requests it.

First, we will describe the DA/AD converters in section 4.1. In section 4.2 comes an explanation of the FPGA design and especially the design of the parts that together makes up the NCO. Section 4.3 describes the changes in the microcontroller design while the new functions in the libEMB library is described in section 4.4. Then comes the PCB design section 4.5. After that, in section 4.6, comes a short description of the low pass filter that is used during testing and experimentation. In section 4.7, we will describe the most important known error sources of the system.

4.1 DA/AD Converter

Figure 4.1 shows the analog board and its main components at a high level of abstraction. The exact chips used are AD5684R (DAC) and AD7888 (ADC) from Analog Devices. The block diagram of the DAC chip is shown in figure 4.2 and the block diagram of the ADC chip is shown in figure 4.3.

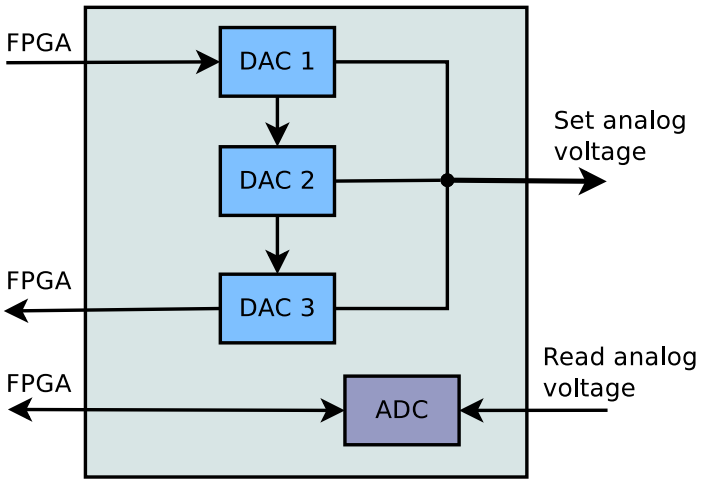

Figure 4.1: Analog board overview.

Each DAC has 4 channels and the board can therefore set 12 different signals. The ADC has 8 channels. Both DAC and ADC has a 12-bit resolution. Using 12-bit gives a sufficient resolution for our purposes. It means that the ADC can detect a voltage change of

$$\Delta V = \frac{V_{REF}}{4096} \tag{4.1}$$

where 1.2 V $\leq$ V_{REF} $\leq$ V_{DD}, when the reference source is external. Using the internal reference source gives V_{REF} = 2.5 V. The external voltage reference pin is connected to V_{DD}, but an optional resistor can be used to adjust the external reference to the desired voltage. Each DAC has an output amplifier which is controlled by the gain pin. When the gain pin is tied to ground, the four output voltages can span between 0 V and V_{REF} [1]. If it is tied to V_{DD}, the voltage span between 0 V and 2×V_{REF}. The (ideal) output voltage is calculated using

$$V_{OUT} = V_{REF} \times Gain \left[\frac{D}{4096}\right] \tag{4.2}$$

where 0 V $\leq$ V_{REF} $\leq$ V_{DD} with an external reference or V_{REF} = 2.5 V with the internal reference. As with the ADC, the external voltage reference pin on the DACs are connected to V_{DD} with an optional resistor for adjustment. Disabling of the internal voltage reference is done by writing to the configuration register, for both the DAC and ADC. Gain is 1 or 2 depending on whether it is tied to ground or V_{DD} (as stated above). The variable D is the base 10 converted binary value written to the DAC, and $0 \leq D \leq 4095$.

[1] Not the same V_{REF} as the ADC uses.

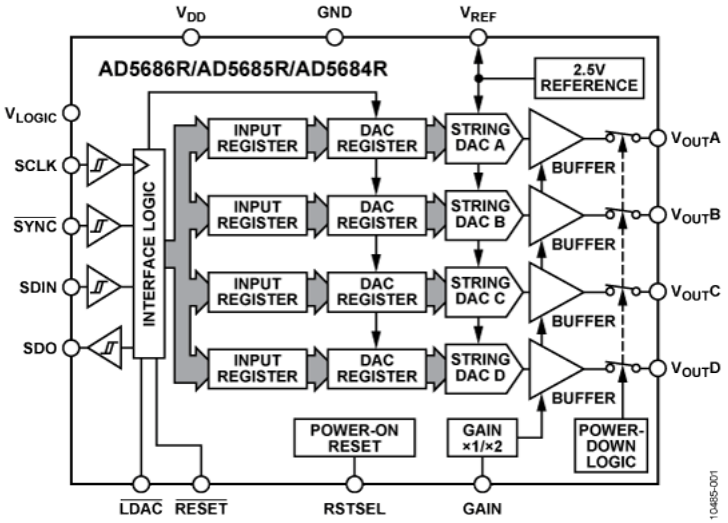

Figure 4.2: Block diagram of the digital-to-analog converter (AD5684R). Source: `http://www.analog.com/static/imported-files/images/functional_block_diagrams/AD5684R_fbl.png`.

Figure 4.4 shows the analog board at a more detailed level when it comes to the digital communication. Both the DACs and ADC uses serial peripheral interface (SPI) for communication. This is a full duplex synchronous serial communication interface, with only four signals. This means that it is relatively easy to implement an SPI controller on the FPGA that can communicate with these devices. The four signals are clock ($\text{SCLK}_{\text{D/A}}$), slave select (DAC: $\overline{\text{SYNC}}$, ADC: $\overline{\text{CS}}$), serial in (DAC: SDIN, ADC: DIN) and serial out (DAC: SDO, ADC: DOUT).

The DAC features a daisy-chain mode, which means that you can connect several DACs together by connecting the serial out pin on one DAC to the serial in pin on the next DAC. Data is first clocked into the first DAC. The command register is 24 bits and when more than 24 bits is clocked in, bits will be clocked out of the first DAC and into the second. So when there is three DACs the clock has to run for $3 \times 24 = 72$ clock cycles so that all three DACs receives their command. This makes it easy to scale up and add more DACs since you only need one serial line for input and one for output. It is especially useful when I/O-pins on the FPGA are scarce, since no matter how many DACs you add, only two I/O-pins is needed for data transfer (you also need clock and select of course). The serial out pin on the last DAC provides readback for the FPGA.

All the DACs share the same select signal called $\overline{\text{SYNC}}$. This signal is low

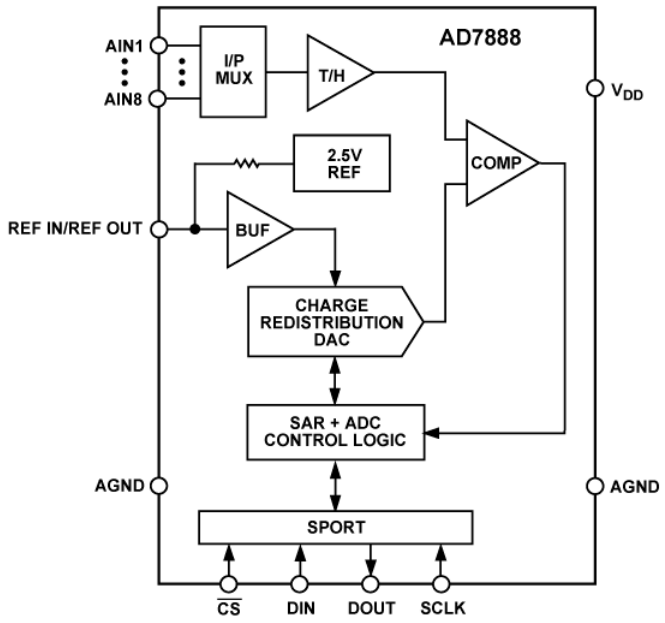

Figure 4.3: Block diagram of the analog-to-digital converter (AD7888). Source: `http://www.analog.com/static/imported-files/images/functional_block_diagrams/AD7888_fbl.png`.

when bits are clocked into the configuration register, and then it is taken high to signal that a new command is in the register. This command is then executed. The $\overline{\text{LDAC}}$ signal is also shared. When you want to set a new output voltage on the DAC channels, you can use the $\overline{\text{LDAC}}$ signal to specify when the channels should be updated. Each DAC channel has two registers where the digital voltage code is stored. One is called *input register* and the other is called *DAC register*. The output voltage is only update when the *DAC register* is updated. Holding $\overline{\text{LDAC}}$ high during transfer will only update the addressed *input register*, while holding it low will update both registers. Holding it high during transfer, but taking it low at the end of the transfer ($\overline{\text{LDAC}}$ is pulsed low) will update all channels asynchronously regardless of which channel was addressed. So, for example writing a new voltage code to three of the DAC channels while holding $\overline{\text{LDAC}}$ high and then write to the last channel and pulse $\overline{\text{LDAC}}$ low when the transfer is finished will update all channels asynchronously. All three DACs also share the same reset signal. More information about the specifications can be found in the datasheets [dac, adc].

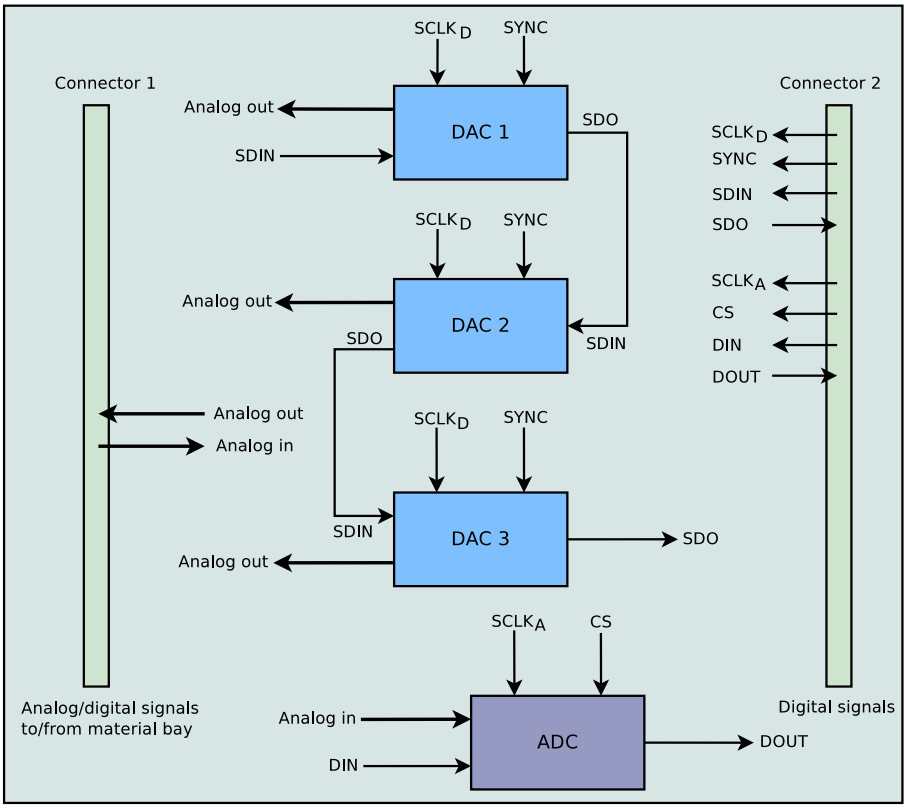

Figure 4.4: Block diagram of the analog board with communication lines.

4.2 FPGA

To be able to use the analog board we have to extend the FPGA design. Figure 4.5 shows the original FPGA design. In figure 4.6, we have added two SPI controllers, a wave module and a multiplexer. We have also extended the user module. Next, we will explain the new parts in detail. We will also give a brief explanation of the parts in the original design. Also, the new design is expecting an FPGA frequency of 50 MHz.

4.2.1 Original Design

The memory controller controls the access to the shared memory for both the microcontroller and the user module. The user module is implemented as a finite state machine (FSM) and its purpose is poll the shared memory to see if the microcontroller has written a new command to the command

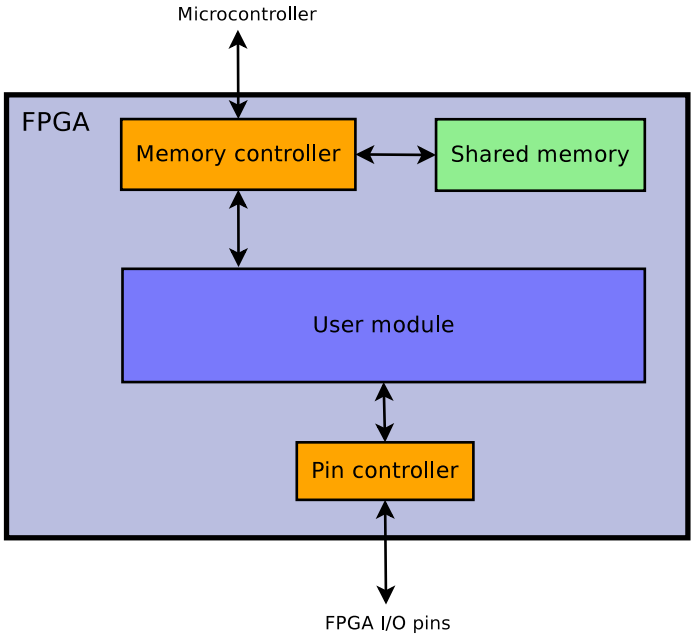

Figure 4.5: FPGA design overview.

memory location. The command is either to configure the I/O pins (decide which one is input or output and set output pins to high or low) or read back the response from the material for those pins that are designated as input. More information on the original design can be found in [Lyk10].

4.2.2 DAC SPI Controller

The DAC SPI controller takes care of the communication with the three DACs. Figure 4.7 shows the block diagram. The controllers FPGA interface includes a clock (`CLK`), reset (`RST`), enable (`EN`) and busy (`BUSY`) signal. In addition we have a command signal that tells the controller which command it should execute when `EN` is asserted. The `DATA` in signal contains the data that is written to the DACs and the `LDAC` signal determines if the $\overline{\text{LDAC}}$ pin should be high or low when writing to the DACs.

The DAC interface contains a serial clock (`SCLK`), slave select (`SS`), DAC reset (`RESET),`LDAC (`LDAC`), serial in (`MISO`) and serial out (`MOSI`). The serial in or MISO signal is currently not in use. The DAC supports a readback command which clocks the register content out on MISO, but the DAC SPI controller does not support this since it's a feature we actually don't need.

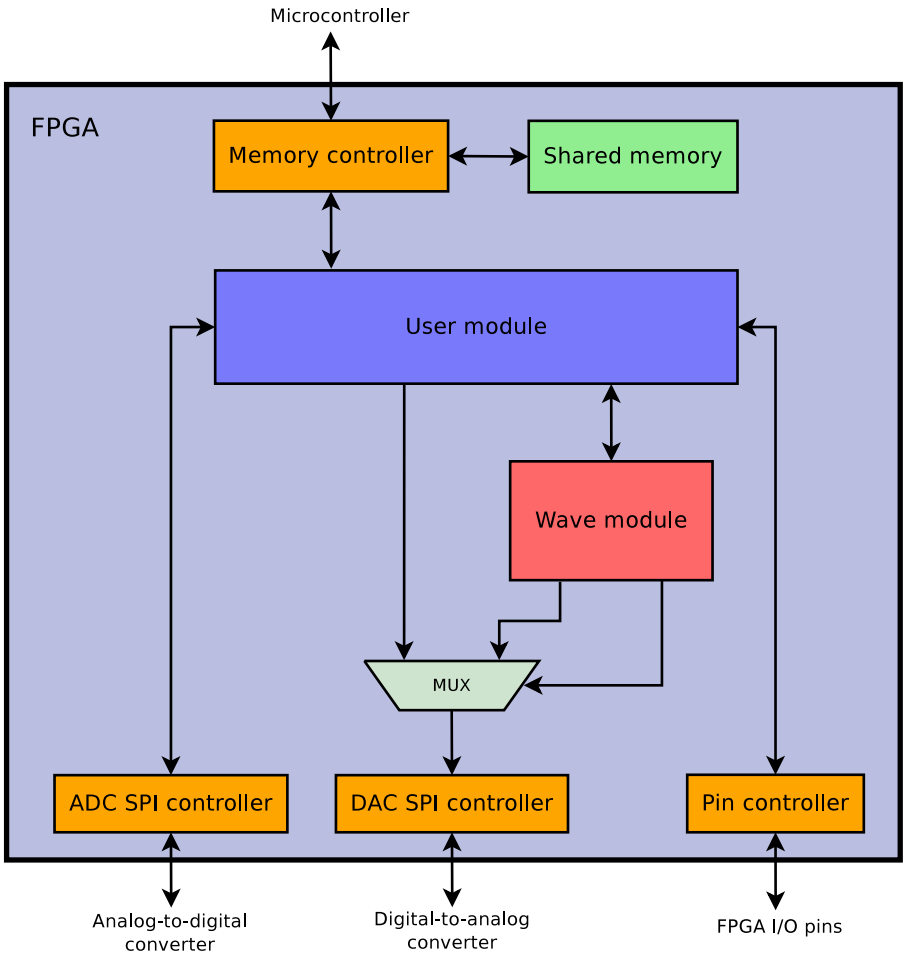

Figure 4.6: The extended FPGA design overview.

To prevent signal glitches, we added registers that works as buffers for the SPI output signals. The combinatorial logic in the finite state machine may cause glitches, but adding the registers (flip-flops) at the output prevents this glitches to propagate further. Adding these registers causes the signals to have a delay of one clock cycle.

Looking at table 4.1 we see that the controller has three write commands. It also has a daisy-chain enable implemented in hardware, that writes the daisy-chain command to the first two DACs. The last command is a reset command that resets the digital-to-analog converters. Their output voltage after a reset depends on whether `RSTSEL` is connected to ground or V_{DD}. The serial clock (`SCLK`) is running at a frequency of 25 MHz. The controllers finite state machine can be found in appendix B.

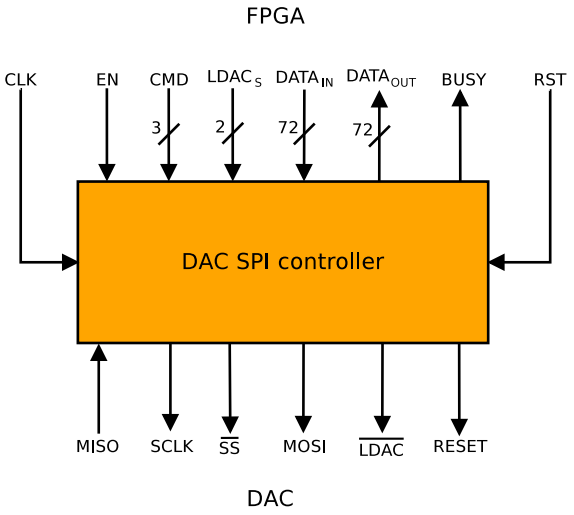

Figure 4.7: Block diagram of the DAC SPI controller.

Command	Description
`CMD_SPI_NOOP`	No operation
`CMD_SPI_WRITE_1`	Write to DAC 1
`CMD_SPI_WRITE_2`	Write to DAC 1 and 2
`CMD_SPI_WRITE_3`	Write to DAC 1, 2 and 3
`CMD_SPI_ENABLE_DC`	Enable daisy-chaining
`CMD_SPI_RESET`	Reset DACs

Table 4.1: Overview of DAC SPI commands.

4.2.3 ADC SPI Controller

The ADC SPI controller is less complicated than the DAC SPI controller. Figure 4.8 shows the input and output signals of the module. Just as the DAC SPI controller it has a clock (`CLK`), reset (`RST`), enable (`EN`) and busy (`BUSY`) signal. No command signal is needed since it only performs one command and that is to clock out 8 bits and clock in 16 bits. The `DATA` in signal provides the 8 output bits while the `DATA` out signal has the response ready when the controller goes idle and the `BUSY` signal is low. The 8 data bits being clocked out specifies the ADC channel to read next, enabling or disabling the internal reference voltage and power management. Since the voltage conversion is ready and clocked out on the next transfer, the first read after power up will always be zero. The ADC interface has the four standard SPI signals (serial clock, slave select, data in and data

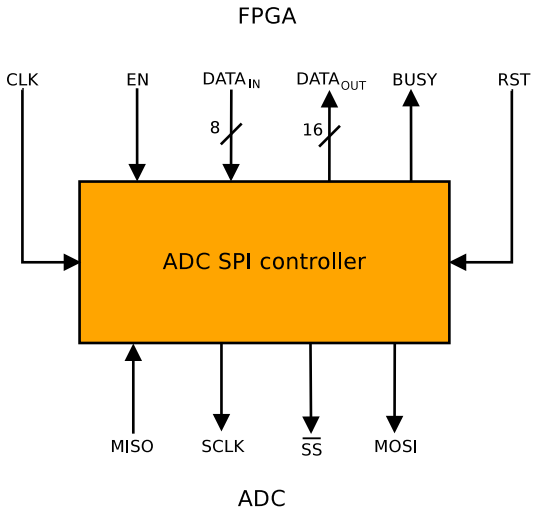

Figure 4.8: Block diagram of the ADC SPI controller.

out) and the serial clock runs at frequency of 2 MHz. As with the DAC SPI controller, registers were added at the output to prevent glitches in the communication. The controllers finite state machine can be found in appendix B.

4.2.4 Numerically Controlled Oscillator

The NCO is the component that provides the DAC with samples. The NCO can be configured digitally, hence the name *numerically controlled*. The wave module shown in figure 4.9 consists of a wave control, several wave generators, a sine look-up table (LUT) and a sample register that communicates with the DAC SPI controller. The wave generators and the sine LUT is what actually makes up the NCO and there is one wave generator for each DAC channel. Figure 4.10 shows the block diagram of the NCO. This a very basic implementation of an NCO and it is based on the design in [Van96].

The first part is the phase accumulator (PA). At each clock cycle the phase increment (also called frequency control word) is added to the accumulator register. This produces a linearly increasing digital value or a sawtooth wave. Additionally, we can add a phase offset. Note that the phase offset is not accumulated like the phase increment. The accumulated value or phase, is then truncated and used in the phase-to-amplitude converter (PAC). The PAC outputs the phase value as it is for sawtooth waveforms

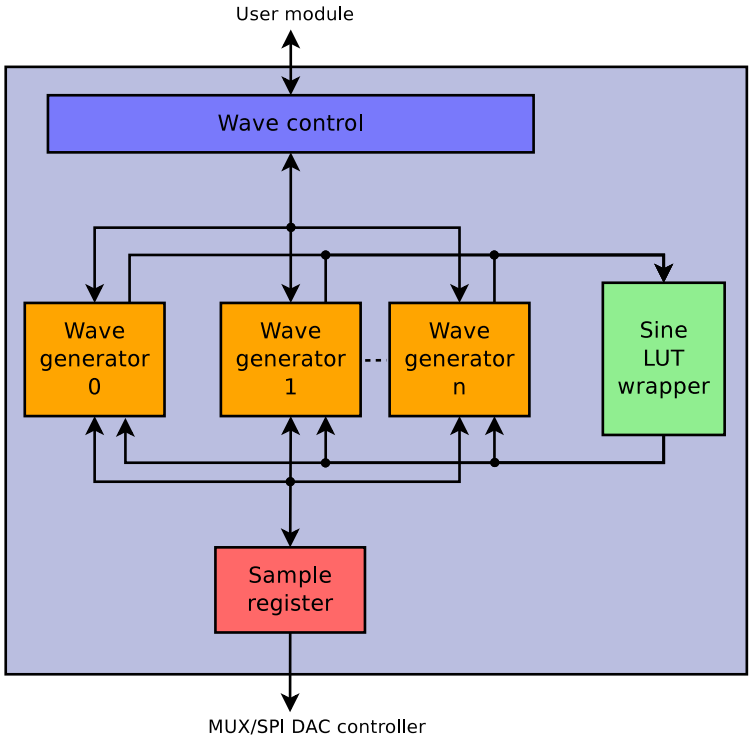

Figure 4.9: Block diagram of the wave module.

or converts the phase value into other waveforms by either using a look-up table or doing some calculations. The output value (amplitude) from the PAC is sent to the DAC, which converts this digital value/code to an analog voltage.

Phase accumulator

The accumulator register is N bits wide. At each clock cycle, the phase increment ΔP is added to the accumulator register. When the register reaches $2^N - 1$, it overflows and starts from the beginning. Equation 4.3 gives the rate of overflow and thus also the frequency of the desired wave

$$f_{out} = \frac{f_{clk}}{2^N}\Delta P \quad \forall \quad f_{out} \leq \frac{f_{clk}}{2} \tag{4.3}$$

where f_{clk} is the frequency of how often the FPGA is able to update the DACs. Remember that the DACs uses serial communication and the FPGA has to transfer $3 \times 24 = 72$ bits when communicating with the three DACs. The FPGA will also add some overhead (e.g. starting and stopping

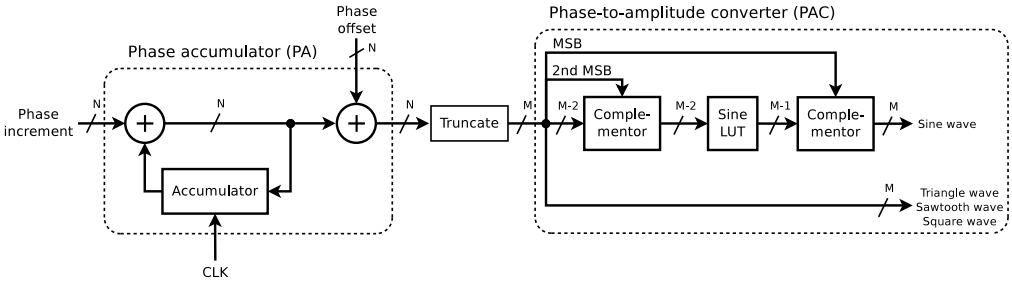

Figure 4.10: Block diagram of the numerically controlled oscillator.

a serial transfer), but the serial communication will be the limiting factor of f_{clk}. While the DAC SPI controller is busy transferring data, the wave generators will calculate the next sample and as long as the calculations take less time than the serial transfer, they will not limit f_{clk}. The limitation in equation 4.3 says that the equation is true for all f_{out} below or equal to the Nyquist frequency. From equation 4.3 we can derive

$$\Delta P = \frac{2^N}{f_{clk}} f_{out} \tag{4.4}$$

which is used to calculate the phase increment value based on the desired output frequency. By setting $\Delta P = 1$ in equation 4.3, we get the minimum possible change in frequency, also called the frequency resolution

$$\Delta f_{out} = \frac{f_{clk}}{2^N} \tag{4.5}$$

The phase accumulator width in our system is set to $N = 16$ bits which is the same as the shared memory word size. Higher N-value will give us better frequency resolution, but with $N = 16$ we get around 5 Hz which is sufficient.

When the accumulator register overflows, the remainder R_n is stored in the register and the next cycle will then start at R_n. This is shown in figure 4.11. At sample seven, the accumulator register has almost reached the its highest value $(2^N - 1)$ and at sample eight it overflows. The remainder R_n is now the starting value. Overflows may also occur when adding the offset, but the remainder R_n is not stored. After a certain number of cycles the initial remainder value R_0 will be reached. The number of cycles it takes to reach R_0 is called the *numerical period* or *grand repetition rate*. It is given by

$$Pe = \frac{2^N}{\mathrm{GCD}(\Delta P, 2^N)} \tag{4.6}$$

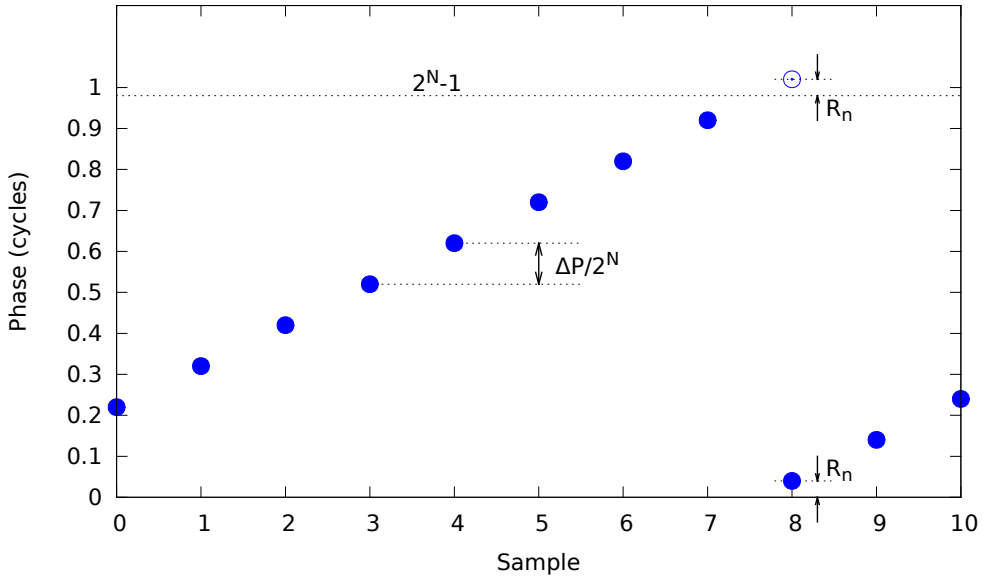

Figure 4.11: Normalized phase accumulator output. Illustration inspired by `http://en.wikipedia.org/wiki/File:Phase_Accum_Graph.png`.

where $\mathrm{GCD}(\Delta P, 2^N)$ is the greatest common divisor of ΔP and 2^N. Since it does not take the exact same number of cycles each time to overflow the accumulator register for a given ΔP, we are more interested in average overflow rate and this is what equation 4.3 tells us.

Phase-to-Amplitude Converter

While the PA determines the phase of the waveform, PAC determines the amplitude. This is done differently for different waveforms, but the truncation between the PA and PAC is common for them all. The width of the phase accumulator register is usually too large to be used directly as an index to a look-up table or directly as the amplitude of e.g. a sawtooth wave, because it may require too much memory space to store the look-up table in the first case and because it may be wider than the DAC resolution in the second case. Instead we use a fraction of the most significant bits. This means that the last $W = N - M$ bits of the PA output has to be removed. We are then left with an M bit wide value as input to the PAC.

For the sine wave we use a LUT to find the right amplitude. This sine LUT is located on the FPGA and is implemented as a single port read-only memory (ROM). The memory requests from the wave generators are

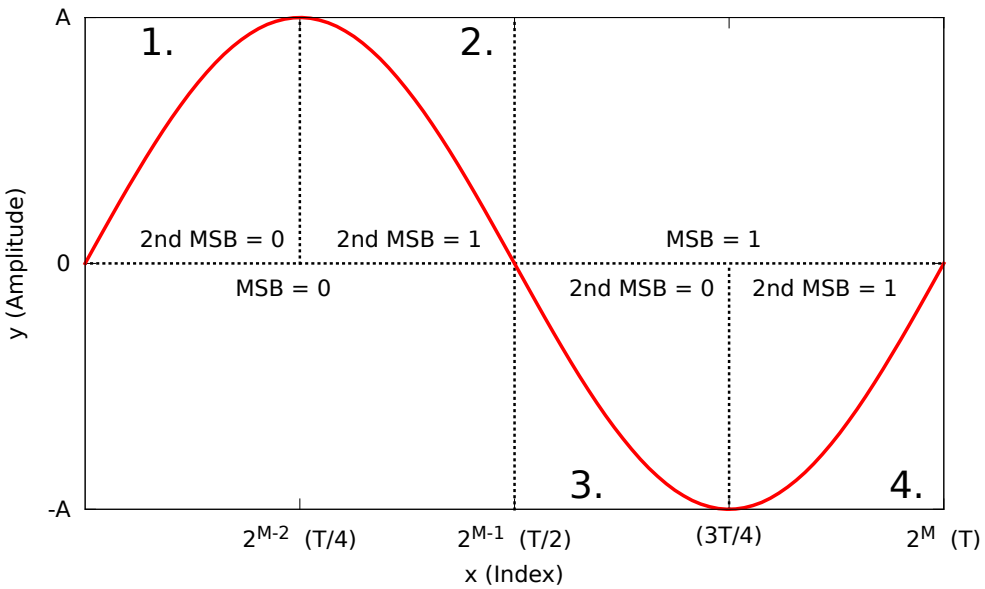

Figure 4.12: The four quadrants of a sine wave showing how the symmetry can be exploited.

processed sequentially and when all requests are handled, the values are presented to all wave generators at the same time. To save space we just store the first quadrant of the sine wave and exploit the sine wave symmetry to get the amplitudes for the second, third and fourth quadrant. Storing the whole period requires $2^M \times K$ bits where M is the width of the index or address and K is the width of the amplitude. By storing only the first quadrant we will use $2^{M-2} \times K$ bits which is four times less. We will have to add more logic to compensate for this, but smaller ROM means lower access time which is important since all the reads are handled sequentially.

If we look at figure 4.12 we see the four quadrants and the value of the most significant bit (MSB) and the 2nd most significant bit of the index when indexing the different quadrants. Looking at figure 4.10 we see that the first complementor uses the 2nd MSB to decide the index. For the first and third quadrant it will go through the indexes from 0 to $2^{M-2}-1$, while for the second and fourth quadrant it will go from $2^{M-2}-1$ to 0. This gives us a full rectified sine wave. The second complementor uses the MSB to decide the sign of the amplitude and invert the third and fourth quadrant to get a normal sine wave. In our implementation, the value of both M and K is set to 12 as this is the same as the DAC resolution. This gives us 4096 sine amplitude values that varies from 0 to 4095. Any wave will

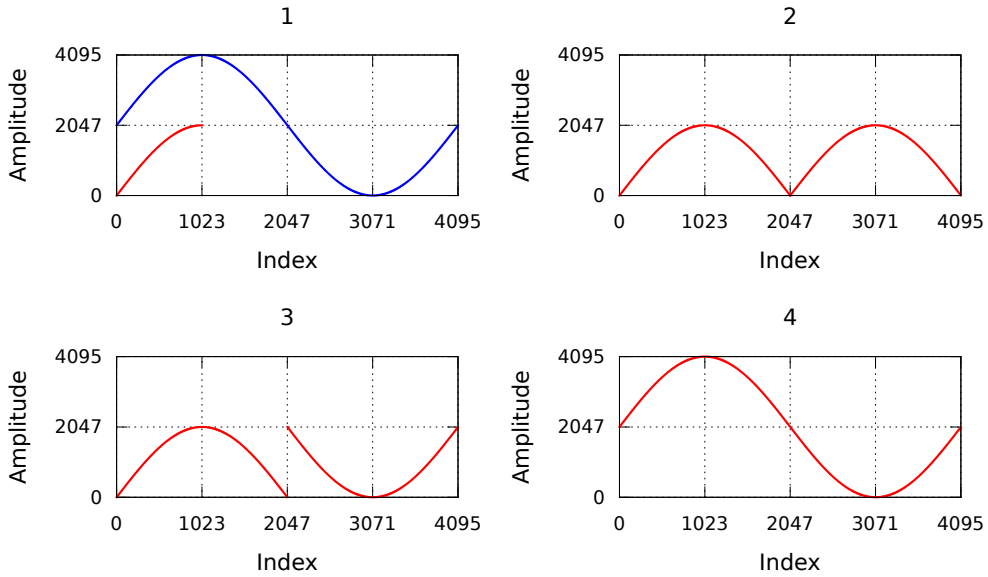

Figure 4.13: 1: Stored quadrant and full period sine wave. 2: Full rectified sine wave out from the sine LUT. 3: Third and fourth quadrant inverted. 4: DC offset added to the first and second quadrant.

have a DC offset of 2048 and a peak-to-peak amplitude of $2^{12} - 1 = 4095$. So to further reduce the size of the sine LUT, we store the sine amplitude from 0 to 2047, instead of 2048 to 4095, using

$$2047 \times \sin\left(\frac{\pi}{2048}x\right) \tag{4.7}$$

where x goes from 0 to 1023 ($2^{M-2} - 1$). Numbers from 0 to 2047 takes 11 bits to represent, while 12 bits is needed when representing numbers from 2048 to 4095. This means that we can save 1 bit of space for each LUT entry. Figure 4.13, graph 1, shows the stored quarter sine wave and the full period sine wave that we want. Since we store the amplitude from 0 to 2047, we have to add the DC offset of 2048 to the amplitude for the first and second quadrant to get the correct wave form. This is done by the last complementor (in figure 4.10). Graph 2 shows the full rectified sine wave we get from the sine LUT. In graph 3, the third and fourth quadrant has been inverted and in graph 4 the DC offset has been added to the first and second quadrant. With this last storage reduction the sine LUT will use $2^{M-2} \times (K - 1)$ bits, and with $M = K = 12$ we get $2^{10} \times 11 = 11264$ bits.

For the sawtooth wave we just truncate the PA value and use it as output from the PAC. The triangle wave is also quite easy to make since it's

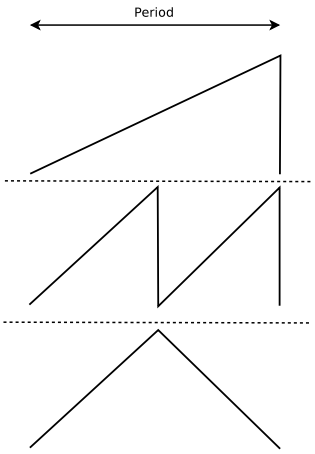

Figure 4.14: Making a triangle wave from a sawtooth wave by frequency doubling and inverting the second sawtooth.

basically a sawtooth wave with double frequency and an inverted second sawtooth. This is illustrated in figure 4.14.

For the square wave, the amplitude depends on the value of the MSB as this indicates half a period. Equation 4.8 shows this.

$$A_{Square} = \begin{cases} 2^M - 1 & \text{if MSB} = 1 \\ 0 & \text{if MSB} = 0 \end{cases} \tag{4.8}$$

Up till now we have assumed that the waveforms have a fixed peak-to-peak amplitude of $2^M - 1$. To change the amplitude of the sine wave we just shift the output value of the PAC n bits to the right. This is equivalent to dividing the value by 2^n and it's the easiest way to change the amplitude since the values in the sine LUT are set to the highest possible amplitude and there is no division unit on the FPGA. There are of course downsides with this method. First of all, we get very coarse amplitude levels since we are halving the amplitude for each shift to the right. Secondly, we loose accuracy since we are doing division of integers. The quotient will be an integer only if the dividend is greater or equal to the divisor and if it is even.

Changing the amplitude of the square wave is just as easy. We just need to output the desired amplitude a when MSB equals 1, instead of $2^M - 1$ as is done in equation 4.8.

For the sawtooth and triangle waves, it is a little bit different. Here we use the truncated PA value directly as output (amplitude) from the PAC. So

to get the correct amplitude we have to substitute 2^N in equation 4.3 and 4.4 with the desired amplitude $a + 1$. For the triangle wave and sawtooth wave, we have to make sure that the offset is less than or equal to the amplitude. If not we will get an incorrect waveform since PA-value will overflow all the time. This check is done in software by the host computer. Note that this limitation does not apply to the square wave or the sine wave, since they map the PA value to amplitude and don't use the value directly as amplitude.

For more detail on how the NCO is implemented in VHDL, see figure B.3 in appendix B. This figure shows the finite state machine.

Channel Grouping

Since there are 12 DAC channel in total we have grouped them into 4 groups. Figure 4.15 shows the DAC channels and their identification. In table 4.2 we can see how the channels are grouped. The number of bits transferred for each group is

$$b = 3 \times 24 \times (g + 1) \tag{4.9}$$

where g is the group number. So higher group number means lower f_{clk} which in turn means lower wave frequency (f_{out}). To always send data to all three DACs even though just one or two of them is in use may seem like a waste, but it is easier to implement. The maximum output frequency we get is high enough for our use, so that is not a problem.

To get the exact frequencies for each group we simulated the wave module design and counted how many FPGA clock cycles it took between each time the FPGA was ready to transfer a new set of samples. Using this information, equation 4.10 and $f_{FPGA} = 50$ MHz we can calculate f_{clk} and maximum f_{out} (Nyquist frequency).

Group	**Channels**
0	A1, A2, A3
1	A1, A2, A3, B1, B2, B3
2	A1, A2, A3, B1, B2, B3, C1, C2, C3
3	A1, A2, A3, B1, B2, B3, C1, C2, C3, D1, D2, D3

Table 4.2: DAC channel grouping.

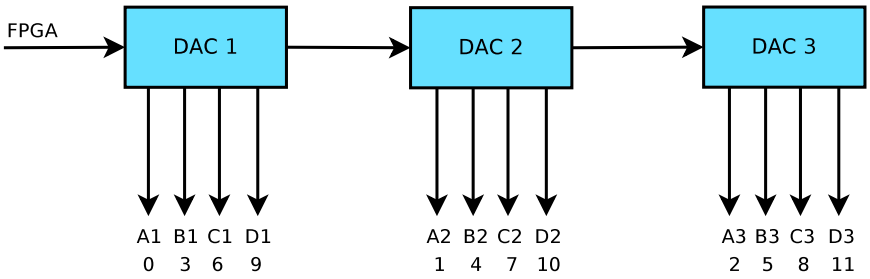

Figure 4.15: Overview of the channel identification numbers.

$$f_{clk} = \frac{1}{\frac{1}{f_{FPGA}} \times CC_{FPGA}} \tag{4.10}$$

The results is shown in table 4.3 and with channel group 0 we can output a waveform with a frequency of approximately 160 kHz which is the highest frequency, but wave frequencies over 70 kHz will probably look very distorted.

Group	$\mathbf{CC_{FPGA}}$	$\mathbf{f_{clk}}$	**Max** $\mathbf{f_{out}}$
0	154	324675, 3247 Hz	162337 Hz
1	302	165562, 9139 Hz	82781 Hz
2	450	111111, 1111 Hz	55555 Hz
3	598	83612, 04013 Hz	41806 Hz

Table 4.3: DAC channel group sample frequencies.

4.2.5 Wave Control

The wave control in figure 4.9 has three tasks to do. The first is to enable the whole wave module. When the user module wants to enable the wave module it talks to the wave control which in turn forwards the enable signal to the rest of the modules inside the wave module. The second task is to store the wave group which is needed by the sample register to know how many channels it should send new samples to. The third task is to control the select line on the multiplexer (see figure 4.6). When the wave module is enabled, the multiplexer should select the signals coming from the wave module. Otherwise it should select the signals from the user module.

4.2.6 Sample Register

When the wave generators are ready with a new sample, the sample register reads the new samples and initiates a transfer to the DACs. While the sample register and DAC SPI controller are transferring data, the wave generators are calculating new sample values. The number of bits transferred depends on the selected channel group and even though not all channels may be in use, the wave generators corresponding to unused channels will still calculate new samples, but the sample register just don't transfer these samples to the DACs. To make sure that the DAC channels updates at the same time (but asynchronously), the sample register instructs the SPI DAC controller to pulse $\overline{\text{LDAC}}$, as described in section 4.1. Figure B.4 in appendix B shows the FSM of the sample register.

4.2.7 User Module Changes

To be able to use the new modules, we expanded the FSM of the user module. The new commands are shown in table 4.4. Except for the three

Command	Description
Write 1	Write to DAC 1.
Write 2	Write to DAC 1 and 2.
Write 3	Write to DAC 1, 2 and 3.
Read	Read ADC voltage.
DCen	Enable daisy-chain
Reset	Reset DACs.
Enable wave	Enable the wave module.
Disable wave	Disable the wave module.
Wave conf	Write wave configuration to the corresponding wave generator.
Wave group	Write wave group to the wave controller.

Table 4.4: New user module commands.

write commands, there are a one-to-one correspondence between the user module commands and the commands/functions in the microcontroller and libEMB software.

When one of the write commands are executed, the FSM will first read the DAC data to transfer, from the shared memory. Then it enables the

DAC SPI controller and waits until the transferring is finished, before it clears the command register and returns to idle state. The DCen and reset commands will immediately enable the DAC SPI controller and execute the corresponding command. All four wave commands are similar to the write commands, except that the enable and disable wave commands does not involve reading from shared memory first. Also, none of the wave commands involves external communication. The read command will read the ADC data bits from memory in addition to the number of samples. Then it will repeatedly enable the ADC SPI controller and write the response to memory, until it has the number of sample we want. With an ADC serial clock frequency of 2 MHz and the current FSM, we get a sample frequency of 111111.111 Hz.

4.3 Microcontroller

The microcontroller software has been extended to be able to use the new FPGA commands. Since there is a one-to-one correspondence between the microcontroller functions and the libEMB functions, we will not explain the functions here, but instead explain other microcontroller changes and extensions. Section 4.4 will address the software changes.

4.3.1 Address Room

The original FPGA address space had $2^{12} = 4096$ addresses. Since we want to be able to get a lot of samples from the ADC, we expanded the address space by two bits. This gives us $2^{14} = 16384$, where the first 384 addresses are reserved for command and configuration data and the last 16000 addresses are for ADC samples. The organization of the address space is shown in figure 4.16.

4.3.2 Busy Line to Microcontroller

When we want the ADC to convert the analog voltage on one of its channels, we have to have some way of knowing when the ADC and the FPGA are finished executing the command so that we can read back the voltage samples from memory. The time it takes may vary depending on how many samples we want. In the original Mecobo design there is no easy

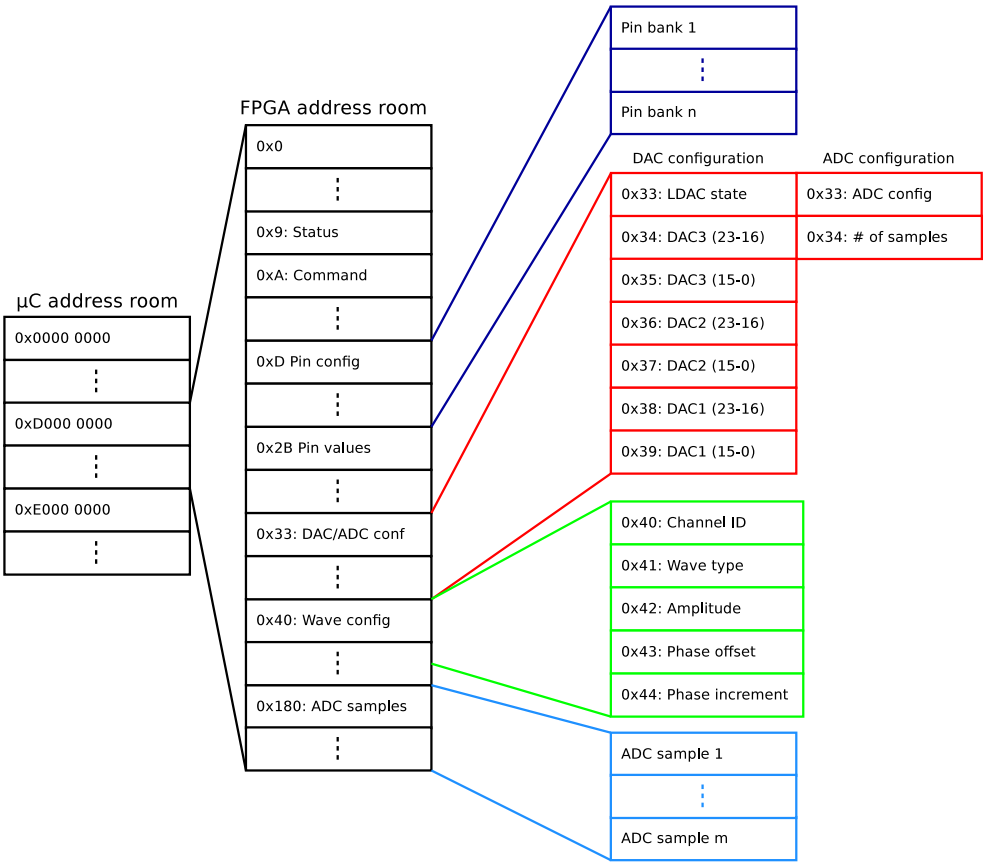

Figure 4.16: Microcontroller and FPGA address rooms

way to signal the microcontroller that the FPGA is finished. We could for example let the FPGA write to a specific memory address and let the microcontroller read this memory address to see if the FPGA is finished, but the problem is that the shared memory controller gives priority to the microcontroller. This means that we cannot let the microcontroller poll the memory repeatedly as this would limit and in worst case stop the FPGA from accessing the memory (writing samples), depending on the clock frequency of the microcontroller and the FPGA. Adding a delay in the polling loop or just let the microcontroller wait some amount of time until we are certain that the FPGA is finished could work, but it is not a good solution. So what we did was to take the RS232 transmit pin on the FPGA and connect it to pin 0 on the microcontroller's PB header (PBH0). This works now as a busy line that signals when the FPGA is busy executing a command. Since neither the RS232 pin nor the PB header were in use, it causes no problem to use these pins for this purpose. The microcon-

troller can now poll this busy line as much as it wants without interrupting the FPGA's memory accesses.

4.4 libEMB

To use the new functionality, we had to extend the libEMB software library. The new functions are listed in table 4.5. The first function is for reading the voltage level on one of the ADC channels. The number of samples can be between 1 and 16000. During testing we discovered some problems that causes the `read_voltage` function to hang. The problem seems to depend on the number of samples and it may be related to the USB communication between the microcontroller and the host computer, since the debug communication (RS232) between the microcontroller and the host computer tells us that the FPGA has finished executing and the microcontroller reaches the point where it starts to send back data to the host computer. When it happens we have to use the reset-button on Mecobo. To avoid this problem altogether, it is best to set the number of samples to 16000 since we know that this works.

The `read_voltage` function also has two arguments called *ref* and *pm*. The first one is used to enable ($ref = 0$) and disable ($ref = 1$) the internal reference. When using an external reference voltage, the internal reference should be disabled to obtain best performance [adc]. The second argument is used for power management. There are four different power modes and these are called *normal operation*, *full shutdown*, *autoshutdown* and *autostandby*. Every time a conversion is completed, the ADC enters the specified power mode and waits for the next command. The shutdown and standby modes needs some time to wake up and this is not taken into account when it comes to the FPGA implementation of the ADC SPI controller. Therefore, one should use the normal operation mode ($pm = 0$) to make sure that the system behaves correctly. The energy consumption of the prototype system is not our first concern so it should not be a problem that the ADC cannot enter power saving modes. An example of how to use the `read_voltage` function is shown in listing 4.1.

Listing 4.1: Read voltage example

```
1 #include "emb.h"
2
3 void get_voltage(void)
4 {
```

Function	Arguments	Description

`read_voltage`	`uint16_t channel`
	`uint16_t samples`
	`uint16_t * buf`
	`uint8_t ref`
	`uint8_t pm`
		Gets *samples* samples from ADC channel *channel*. Stores them in *buf*. *ref* turns on or off the internal reference while *pm* is the power management.

`write_dac`	`dac_config_t * conf` `uint16_t ldac`	Write *conf* to the DACs and use the LDAC-state *ldac*.
`enable_daisy_chain`	`void`	Enable DAC daisy-chaining.
`reset_dac`	`void`	Reset all DACs.
`enable_wave`	`void`	Enable the wave module.
`disable_wave`	`void`	Disable the wave module.

`set_wave_config`	`uint16_t channel_id`
	`uint16_t wave_type`
	`uint16_t amplitude`
	`uint16_t offset`
	`float freq`
	`int channel_group`
		Configure wave channel *channel_id* using *wave_type*, *amplitude*, *offset* and *freq*. *channel_group* is used when calculating the phase increment.

`set_channel_group`	`uint8_t group`	Set DAC channel group to *group*. Default is group 0.

Table 4.5: New libEMB functions implemented for the analog board.

```
5    // Set configuration variables
6    uint16_t channel = ADC_CHANNEL_1;
7    uint16_t samples = 16000;
8    uint16_t * buffer = malloc(sizeof(uint16_t) * samples);
9    uint8_t ref = 0; // Internal reference
10   uint8_t pm = 0; // Normal operation mode
11
12   // Read voltage
13   read_voltage(channel, samples, buffer, ref, pm);
14
15   // Use the samples
16   do_some_calculations(buffer, samples);
17
18   free(buffer);
19 }
```

The `write_dac` function is used to write all kinds of commands to the

DAC(s), including setting the voltage level on one or more channels. The first argument is a pointer to a structure called `dac_config` which again points to three different structures, called `dac_reg_content`, that holds the register content to be written to each DAC. It contains the command, the channel and data. Note that not all commands require channel or data information. More about this can be found in the datasheet [dac]. Two auxiliary functions called `create_dac_config` and `create_dac_reg_content` are used to create the structures. Listing 4.2 shows an example on how to use `write_dac`. In this example we set the output voltage on channel B and D on DAC 1 to 1.25 V. Note how we specify the channels by using the bitwise OR operator in line 7. In line 8, we shift the data bits four bits to the left since the four least significant bits are not in use. Also note that the configuration for DAC 2 and 3 is a `NULL` pointer. The `write_dac` function will detect the pointer that are `NULL` and choose the right write command (write 1, write 2 or write 3) on the FPGA. When writing to just e.g. DAC 1 and DAC 3, the pointer to the DAC2 configuration cannot be NULL. It has to point to a valid configuration since the FPGA will transfer 72 bits to configure DAC 3.

The argument called *ldac* is used to specify the $\overline{\text{LDAC}}$-signal during transfer. It could be set to *low*, *high*, *pulse* or *don't care*. Setting it to *don't care* is the same as setting it to *high*. Note that the *ldac* argument only applies to one command (0x0001), where you set the output voltage.

Listing 4.2: Write DAC example

```
1  #include "emb.h"
2
3  void static_voltage(void)
4  {
5    // Set configuration variables
6    uint8_t command = DAC_CMD_UPDATE_CHANNEL;
7    uint8_t channel = DAC_CHANNEL_B | DAC_CHANNEL_D;
8    uint16_t data = (voltage_to_data_conv(1.25) << 4); // 4
       LSB are not in use
9
10   // Create a DAC config structure
11   dac_config_t * c = create_dac_config();
12
13   // Create the register content
14   c->dac1_reg = create_dac_reg_content(command, channel,
       data);
15   c->dac2_reg = NULL;
16   c->dac3_reg = NULL;
17
18   // Apply command
```

```
19    write_dac(c, DAC_LDAC_PULSE);
20
21    free_dac_config(c);
22  }
```

Listing 4.3 shows how to initialize the analog board and the wave module. It is wise to always make sure that the wave module is disabled, since `reset_dac` and `enable_daisy_chain` won't be executed when it is enabled. Disabling the wave module also resets all the registers in the wave generators, so that when the module is enabled next time, the waves starts at the beginning. This is useful when two or more equal waves has a phase difference. By disabling and enabling the wave module, they will become synchronized. The wave configurations will not disappear when the wave module is disabled.

Listing 4.3: Analog board setup example

```
1  #include "emb.h"
2
3  void analog_board_setup(void)
4  {
5    disable_wave(); // Make sure the wave module is disabled.
6    reset_dac(); // Reset the DACs.
7    enable_daisy_chain(); // Enable daisy chain.
8  }
```

The last code example in listing 4.4 shows how to set the wave configuration. Line 8 sets the channel group. It is not necessary to call this function if you only want to use group 0, since this is the default group. The call to `enable_wave` in line 12 could be moved to after the calls to `set_wave_config` to synchronize the waves. Table 4.6 shows the constraints on the wave configuration arguments.

Listing 4.4: Wave configuration example

```
1  #include "emb.h"
2
3  void generate_wave(void)
4  {
5    // Set channel group.
6    // The default group is 0.
7    int channel_group = CHANNEL_GROUP_1;
8    set_channel_group(channel_group);
9
10   // Enable the wave module. This function could also
11   // be called after set_wave_config().
12   enable_wave();
```

```
13
14    // Set configuration variables
15    uint16_t channel = 0; // Channel A, on DAC 1
16    uint16_t wave_type = SINE_WAVE;
17    uint16_t amplitude = 0; // Zero shifts to the right
18    uint16_t offset = 1024; // Set phase offset to T/4
19    float frequency = 500.0;
20
21    // Apply first configuration
22    set_wave_config(channel, wave_type, amplitude, offset,
          frequency, channel_group);
23
24    // Set new configuration
25    channel = 4; // Channel B, on DAC 2
26    wave_type = SQUARE_WAVE;
27    amplitude = 4095 // Full amplitude
28    offset = 0;
29    frequency = 1000.0;
30
31    // Apply second configuration
32    set_wave_config(channel, wave_type, amplitude, offset,
          frequency, channel_group);
33 }
```

Argument	Constraint
`channel_id`	$0 \leq$ `channel_id` ≤ 11
`wave_type`	`SAWTOOTH_WAVE`, `TRIANGLE_WAVE`, `SINE_WAVE` or `SQUARE_WAVE`
`amplitude`	$0 \leq$ `amplitude` ≤ 12 for sine wave, else $0 \leq$ `amplitude` ≤ 4095
`offset`	$0 \leq$ `offset` ≤ 4095 for sine and square wave, else $0 \leq$ `offset` $\leq$ `amplitude`
`freq`	$0 \leq$ `freq` $\leq$ max f_{out} (see table 4.3)
`channel_group`	$0 \leq$ `channel_group` ≤ 3

Table 4.6: Argument constraints for the wave configuration.

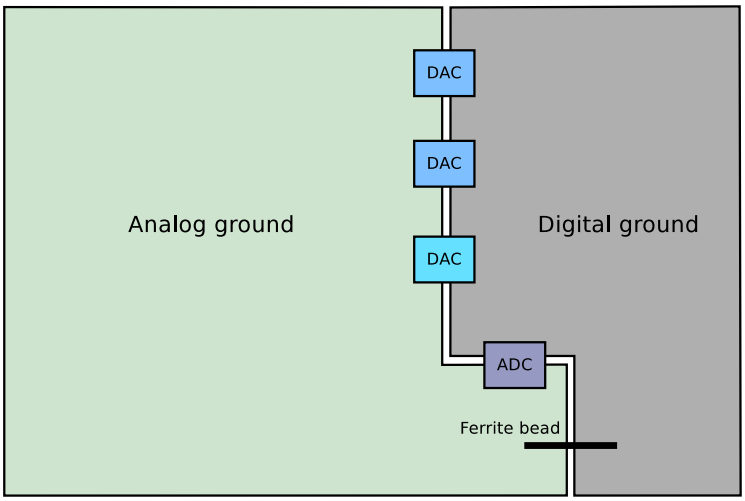

Figure 4.17: Splitting of analog and digital ground.

4.5 PCB

When designing a mixed signal PCB we have to take into consideration that noise and interference between analog and digital signals may occur. The analog board is a mixed signal PCB with four layers where the signals are separated. The top layer is the digital and analog ground, in addition to analog signals (also some short digital traces). The first internal layer is for digital signals in the horizontal direction. Second internal layer is the power plane. The bottom layer is for digital signals in the vertical direction. The reason for dividing the digital signals is to make the routing easier. Separation of the digital and analog signals gives less interference in the analog signals, and thus makes them more stable. This is especially important when the digital signals have a high frequency.

As stated above, the ground is split into analog ground (AGND) and digital ground (GND), as illustrated in figure 4.17. The reason for splitting the ground into an analog and digital part is to try to reduce the digital signals interference on the more fragile analog signals. They are connected in one place with a ferrite bead that suppresses high frequency noise which may come from the digital ground. The DACs and ADC has been placed on the board so that most of the analog pins are soldered on the analog ground side, while the digital pins are soldered on the digital ground side.

Figure 4.18 and 4.19 shows the schematics of the DAC and ADC, respectively. V_{DD} is the regular power supply while V_{logic} is the power supply for

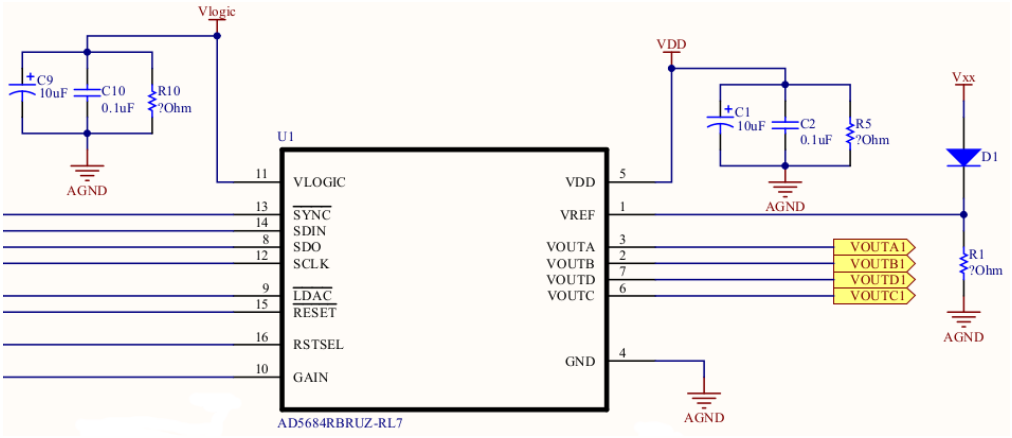

Figure 4.18: DAC schematic.

the digital logic inside the DAC. V_{xx} is connected to the external reference pin on both the DACs and ADC. All three power supply inputs V_{DD}, V_{xx} and V_{logic} are connected to the internal power plane.

Every power supply input on the DACs and ADC has two capacitors connected in parallel. Their purpose, together with a parallel resistor, is to smooth out any voltage spikes that may occur with both high and low frequencies. The diodes connected to the V_{REF} pins are used as current generators. After a certain voltage threshold, they give a constant current which together with the resistor creates a fairly good voltage regulator. By using V_{REF} as the voltage reference, we can trade range for resolution and vice versa depending on what we need. Figure 4.20 shows the final board with most of the components soldered on. The jumpers on the board can be used to select the gain and the value on the DAC channels after a reset. Table 4.7 lists the different PCB design rules for the analog board.

4.6 Low-pass Filter

The last part of the direct digital synthesizer is the low-pass filter. As the name suggests, this filter lets the low frequencies pass through, but reduces the high frequency noise above the *cutoff frequency*. The easiest filter to create is a first order passive RC filter and its circuit drawing is shown in figure 4.21. This is the type of filter we have used during testing and evaluation of the system. Equation 4.11 gives us the *cutoff frequency* for

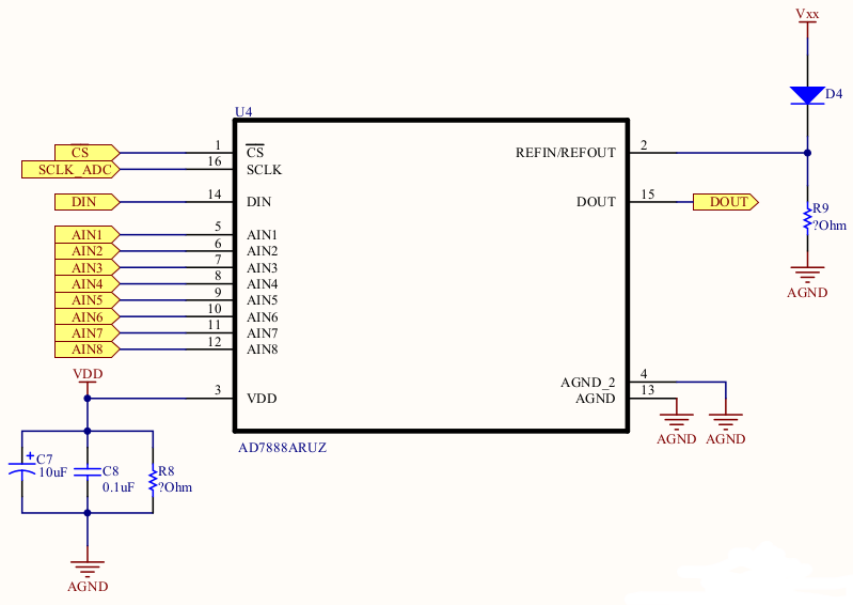

Figure 4.19: DAC schematic.

the low-pass filter

$$f_{cutoff} = \frac{1}{2\pi RC} \tag{4.11}$$

where we use $R = 4.7$ kΩ and $C = 470$ pF. This gives $f_{cutoff} = 72048$ Hz. These values were chosen based on how good the the output waveform looked on the oscilloscope.

4.7 Error sources

Throughout the system, errors in the form of distortion and noise will be introduced and it will degrade the generated wave(s). We will in this section explain the main errors introduced in the NCO and the DAC/ADC.

4.7.1 Phase Truncation

The first error is the phase truncation distortion or spur. This happens when we, as the name suggests, truncate the phase accumulator value. Figure 4.22 illustrates the effects of this truncation. It shows a phase wheel with the outer circle representing the values of a 5 bit phase accumulator

Rule	Value	Description
Current requirement	4.6 mA (minimum)	Trace current
Trace width	0.254 mm	≈1.55 A tolerance [pcb]
Trace thickness	0.2 mm	
Via 1	Inner diameter: 1 mm. Outer diameter: 0.2 mm	Power via. Based on [BG].
Via 2	Inner diameter: 0.254 mm. Outer diameter: 0.2 mm	Digital signals. Based on [BG].
Via 3	Inner diameter: 0.4 mm. Outer diameter: 0.2 mm	Connecting analog ground islands. Based on [BG].

Table 4.7: PCB design rules.

($N = 5$) and the inner circle represents the 3 bit truncated value ($M = 3$) we get from removing the 2 least significant bits from the phase accumulator value ($W = 2$). One revolution of the phase wheel is equivalent to one period of the wave. In the figure, the phase increment is 3 ($\Delta P = 3$) and the first four phase accumulator steps are marked. The first step adds 3 to the phase accumulator, but since the value does not reach the red dot representing the value 1 for the truncated value, we get a phase error E as shown in the figure. This phase error is the number of dots between the phase accumulator value and the truncated value and in the first step it is equal to 3 or $\frac{3}{16}\pi$ radians. The second step gives us a phase accumulator value of 6. It has moved past the first red dot and therefore the truncated value equals 1. The phase error is now 2 or $\frac{\pi}{8}$ radians. So in the third step we get a phase error of 1 or $\frac{\pi}{16}$ radians. The fourth step on the other hand, hits both the blue and the red dot meaning that there is no phase error. This error pattern will repeat itself and since the truncated value is converted to amplitude (or used directly), the phase errors will cause errors in the amplitude. These errors are periodic in the time domain, which means that it shows up as distortion or spurs in the frequency domain.

The magnitude and distribution of the distortion is determined by the number of bits in the phase accumulator (N), the number of bits in the truncated phase value (M) and the phase increment (ΔP). Some phase increment values gives no phase errors and therefore no distortion in the

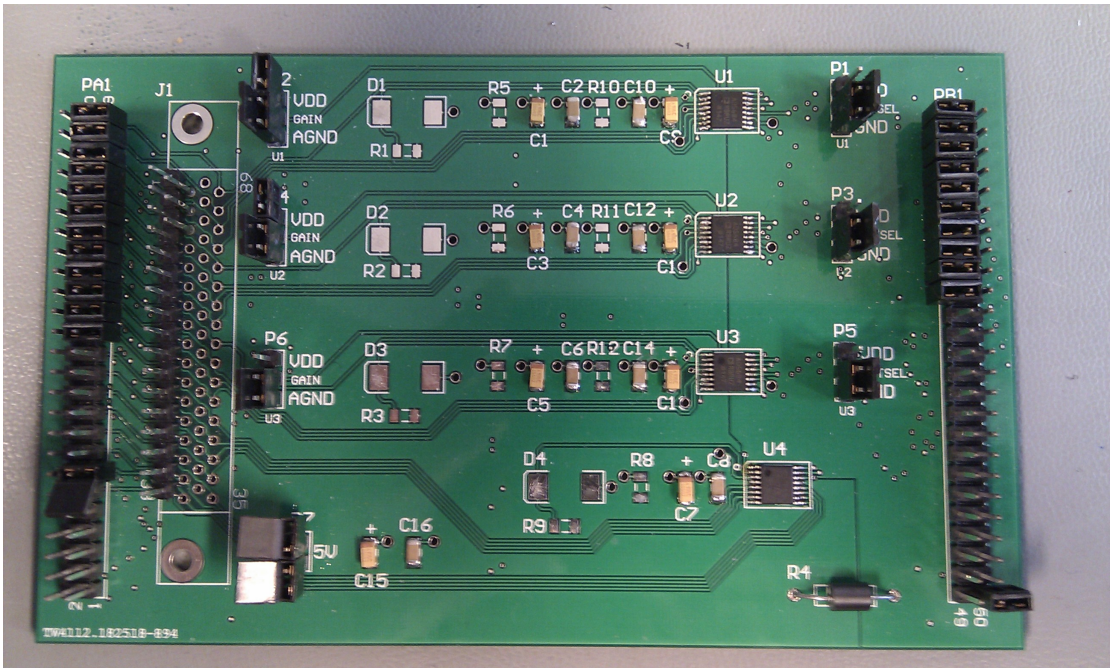

Figure 4.20: The analog board with most of the components soldered on.

frequency domain of the signal. These phase increment values all satisfies

$$\mathrm{GCD}(\Delta P, 2^W) = 2^W \tag{4.12}$$

At the opposite end we have phase increment values that gives the highest distortion level and they satisfy

$$\mathrm{GCD}(\Delta P, 2^W) = 2^{W-1} \tag{4.13}$$

When the number of truncated bits $W \geq 4$, which is true for our system,

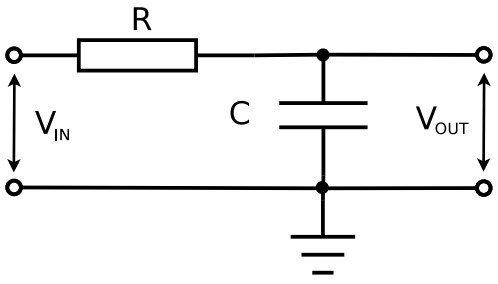

Figure 4.21: First order passive low-pass filter used at the DAC outputs.

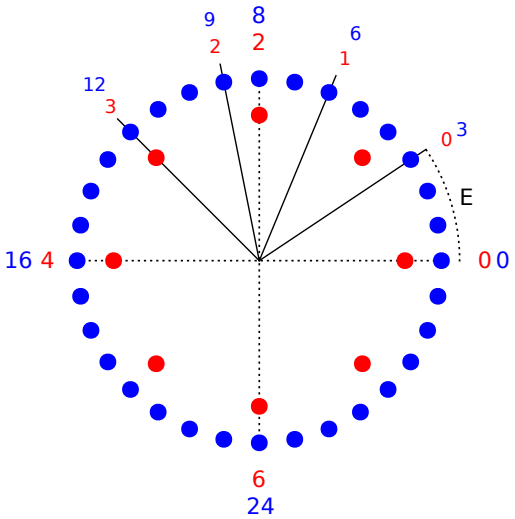

Figure 4.22: Illustration of phase truncation error. Blue dots represents the phase accumulator value and the red dots represents the truncated value. Illustration inspired by [dds].

the maximum distortion level could be estimated with

$$\zeta_{max} = -6.02M \tag{4.14}$$

where the answer is given in dBc, which is the power ratio of the distortion and the carrier signal [wik]. So for our system with $M = 12$, the maximum phase truncation distortion will be approximately -72 dBc. For more information about phase truncation error, see [dds], which is the source used in this subsection.

4.7.2 Quantization

The DA/AD converters in our system has a resolution of 12 bits which means that they can differentiate between 4096 voltage levels. Together with the internal voltage reference of 2.5 V, we get the lowest possible voltage change of $V_{LSB} = 600\ \mu$V. This means that if the signal to be converted by the ADC is changing with less than V_{LSB}, the ADC will not be able to detect this change and there will be a difference between the actual voltage level and the converted voltage level. For the DAC, we get the same effect when we want to create a sine wave signal with

voltage changes less than V_{LSB}, but have to round the the values so that the voltage changes is equal to or larger than V_{LSB}. This difference is called the quantization error and happens because we are using rounding to map a large set of input values to a smaller set. Since the sine samples stored in the LUT is rounded to the nearest integer, the stored sine wave will therefore deviate from a perfect sine wave and the error will manifest itself as distortion or spurs in the frequency domain [dds].

5

Testing and Evaluation

To test and evaluate the new extended system, we conducted several different tests. These can be categorized into system tests, FPGA tests and analog/digital converter tests.

5.1 System Tests

To test all the new functions in libEMB, we created different C programs that called the function(s) under test and verified the result of the call. This was either a visual verification with an oscilloscope or an automatic one where the test program measured for example voltage via a multimeter. The system tests verifies that the system as a whole works correctly and is basically black box testing. Some of the functions under test were tested together, such as `enable_daisy_chain` and `write_dac`, because their functionality is closely related. The tests and their results can be found in appendix A.1.

5.2 FPGA Tests

Testing the FPGA modules was mostly done by using automated functional simulation. The exception was the the tests where we generated waves, which were verified visually. This was done by writing every wave sample to file and then use Gnuplot to plot the samples. For the automated functional simulation we created random stimuli using built-in VHDL functions or we used fixed stimuli. The output of the module was

then compared to the expected output and the simulation stopped or gave a warning if the comparison failed.

5.3 Digital/Analog Converter Evaluation

To test and evaluate the DA/AD converters we did several measurements. These measurements can be categorized into *static* and *dynamic* tests or measurements. The first tests are called integral nonlinearity (INL) and differential nonlinearity (DNL) and they fall into the category of *static* tests. We also did frequency domain analysis which is *dynamic* testing. All measurements were done using channel A on the DAC 1 and channel 1 on the ADC. The test equipment used can be found in appendix C.

To test the DAC, we wrote a C program that iterates through all the digital voltage codes, i.e. from 0 to 4095. Each code gets written to the DAC which outputs the corresponding voltage. This voltage is then measured by a digital multimeter, that communicates with the computer running the program. The measured value is then converted back to digital code. The result of this measurement is shown in figure 5.1, where the code written to the DAC is along the x-axis and the measured value is along the y-axis. Here, the measured values are overlapping the ideal values. The INL and DNL can now easily be calculated using the data plotted in this figure.

A similar approach was used for measuring the ADC. Here, we used a DC power supply to output the voltage corresponding to the voltage code and the ADC converted this voltage back to digital code. The power supply we used had an accuracy of 10 mV, while the ADC has an accuracy of 0.6 mV. This means that the measurements gets noisy and inaccurate. To decrease the error introduced by this problem, we used the average value of several samples. Equation 5.1 shows the exact formula used

$$AVG_c = \frac{\sum_{r=1}^{20}\left(\frac{1}{2000}\sum_{s=1}^{2000} Sample_s\right)}{20} \tag{5.1}$$

where the subscript c denotes the decimal code of the voltage, i.e. $0 \leq c \leq 4095$. As we can see from figure 5.2, the measured values is not that far away from the ideal ones.

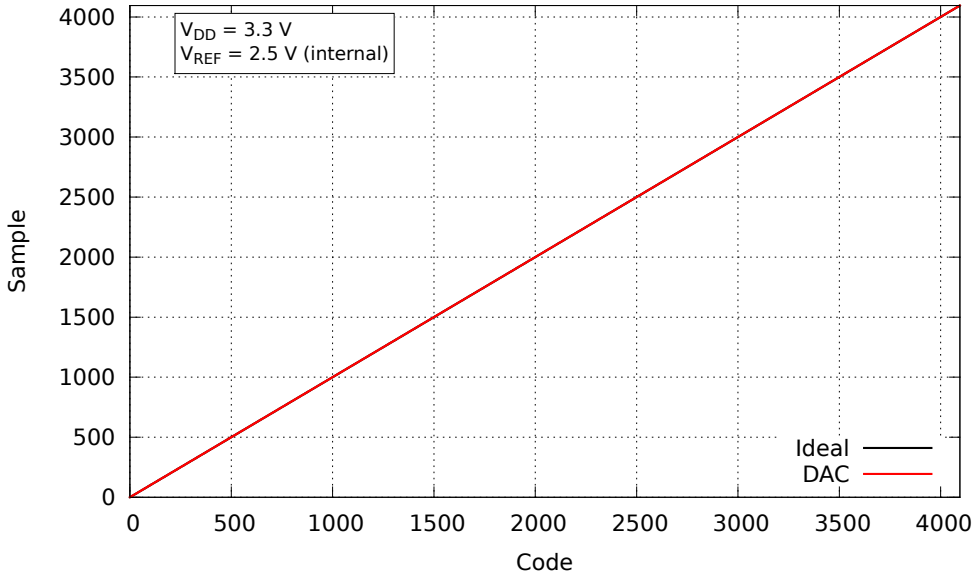

Figure 5.1: Plot of the ideal vs. measured DAC value.

5.3.1 Integral Nonlinearity

Integral nonlinearity, also called relative accuracy, tells us how much the measured transfer function deviates in least significant bits (LSBs) from the ideal transfer function of a DAC or ADC. The INL is calculated after the gain error and offset error has been removed. The best straight-line INL approach is used in this case, because it usually gives better results. Details of the method are presented in [max].

Figure 5.3 shows the INL plot for the DAC. The x-axis is the digital voltage code sent to the DAC and the y-axis is the deviation from ideal output. As we can see, the deviation is very small, mostly between 0 and -0.9 LSB. The average deviation is -0.479 LSB, which means that the measured voltage is on average approximately 300 μV lower than the ideal output voltage (this is after gain and offset error has been removed). This is a very low voltage difference and the measurement may therefore not be completely accurate, but it gives us a picture of the characteristics of the DAC. The reason why the value drops may be due to the fact that the multimeter changes the voltage reference and thus changes the accuracy [mul]. We do not know the reason why the curve is generally increasing, but we have had some electromagnetic interference with an unknown source(s) that may affect the measurement. Luckily, since the curve only increases

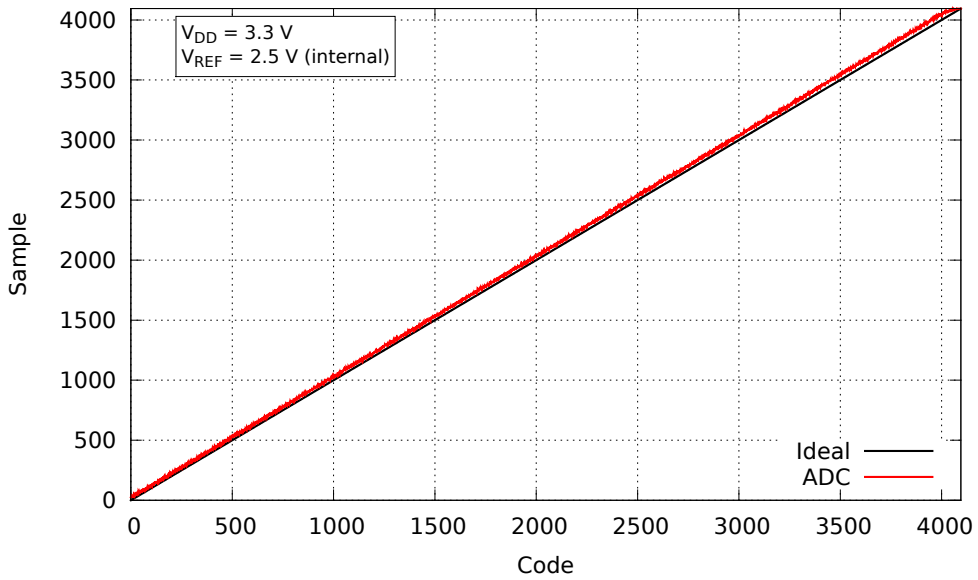

Figure 5.2: Plot of the ideal vs. measured ADC value.

by approximately 1 LSB over the interval from 0 to 4095, it can be ignored and we can conclude that the nonlinearity in DAC's transfer function is close to zero when it is compared to the ideal transfer function.

In figure 5.4, we can see the INL plot of the ADC. The x-axis is the digital voltage code that was converted and sent to the power supply and the y-axis is the difference between the measured voltage and the ideal voltage. Here the deviation is much greater. As mentioned above, the power supply we used when measuring the ADC performance had lower accuracy than the ADC and this will show up as noise in the measurement. It is in general more difficult to get an accurate ADC than an accurate DAC since the ADC has to convert a potentially noisy analog voltage to digital code, while the DAC converts an unambiguous digital code to an analog voltage. Additional noise could of course in both cases be introduced due to e.g. noisy ground or reference voltage. So even though the ADC measurements is not completely accurate, we get a picture of how it performs.

The average deviation is 22.5 LSB which means that the measured voltage is on average approximately 14 mV greater than the ideal voltage. The deviation is also generally increasing. Over the interval from 0 to around 4000, the INL goes from approximately 0 to 60 LSB which means a voltage increase of 36 mV. This tells us that the nonlinearity in the ADC's transfer function is quite high when compared to the ideal transfer function. The

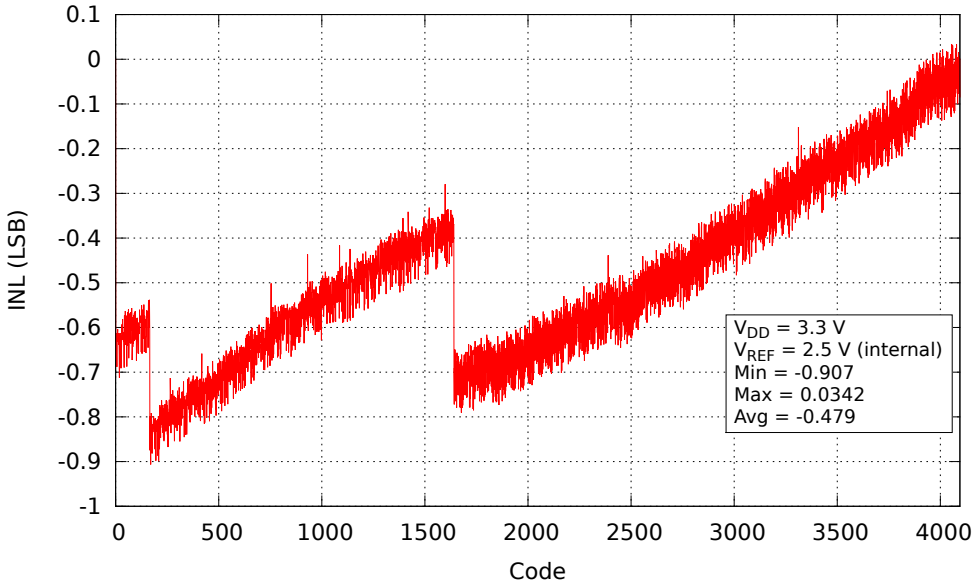

Figure 5.3: Plot of the DAC integral nonlinearity.

reason for this increase is unknown, but it might be a combination of the power supply accuracy and, as mentioned earlier, some electromagnetic interference.

5.3.2 Differential Nonlinearity

The DNL is a measurement of how much difference there is between the measured step width and the ideal step width of 1 LSB ($\frac{2.5}{4096} \approx 600\ \mu$V), between two contiguous codes. Ideally, the DNL should be equal 0 LSB as this is when the step width equals 1 LSB. Gain error has been removed before calculating the DNL. The exact formula used can be found in [max].

Looking at figure 5.5, we can see the DNL of the DAC. The x-axis is the digital voltage code sent to the DAC and the y-axis is the difference between the measured voltage step and the ideal voltage step. The largest difference is approximately -0.6 LSB, which is very small. The fact that the largest difference is less than or equal to 1 LSB (in absolute value) is no surprise since this means that the DAC has a monotonic transfer function with no missing codes, which is guaranteed by the design according to [dac, p. 18].

Figure 5.6 shows the DNL of the ADC. The x-axis is the digital voltage

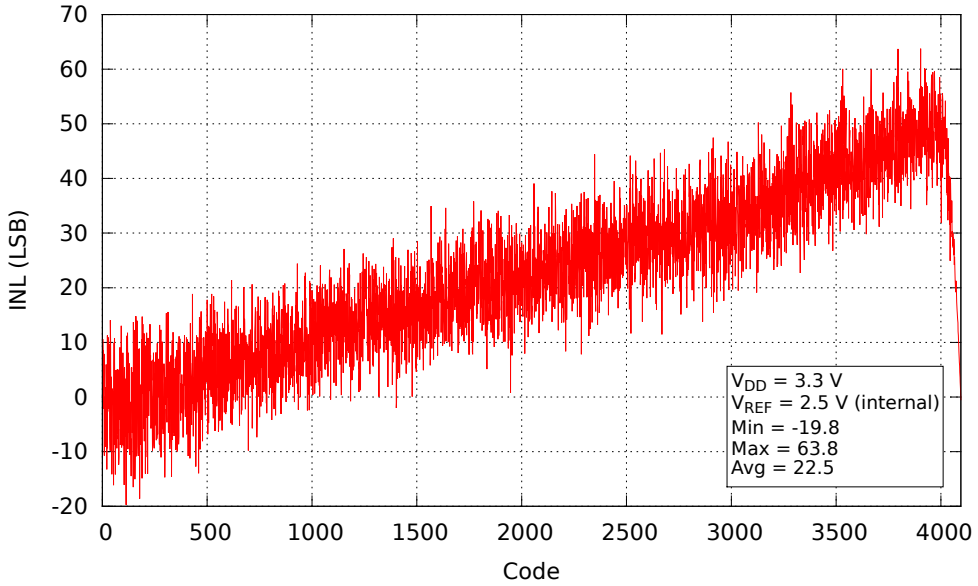

Figure 5.4: Plot of the ADC integral nonlinearity.

code that is converted to voltage and sent to the power supply and the y-axis is the difference between the measured voltage step and the ideal voltage step. According to [max], the transfer function of an ADC is monotonic with no missing when the DNL value is between -1 and +1 LSB. Just a monotonic tranfser function is guaranteed when the digital output code is non-decreasing when the input voltage is increasing, i.e. the DNL value is greater than -1 LSB. This prevents changes of the sign of the transfer curve's slope. A DNL value of 0 means that the voltage step equals exactly 1 LSB, while a value greater than 0 equals a voltage step that is greater than 1 LSB. If the value is -1 it means that the voltage step is zero, i.e. the measured voltage is the same as the previously measured voltage. A value less than -1 LSB on the other hand means that the voltage step is less than 0 LSB or a decrease in the measured voltage and this violates the requirement for monotonicity. As we can see from the figure, the ADC's transfer function is clearly non-monotonic and is missing codes since a lot of the DNL values are less than -1 LSB and greater than +1 LSB. A non-monotonic ADC with missing codes implies lower effective resolution. High DNL values will in general limit the performance by reducing the signal-to-noise ratio (SNR) and spurious-free dynamic range (SFDR) [max].

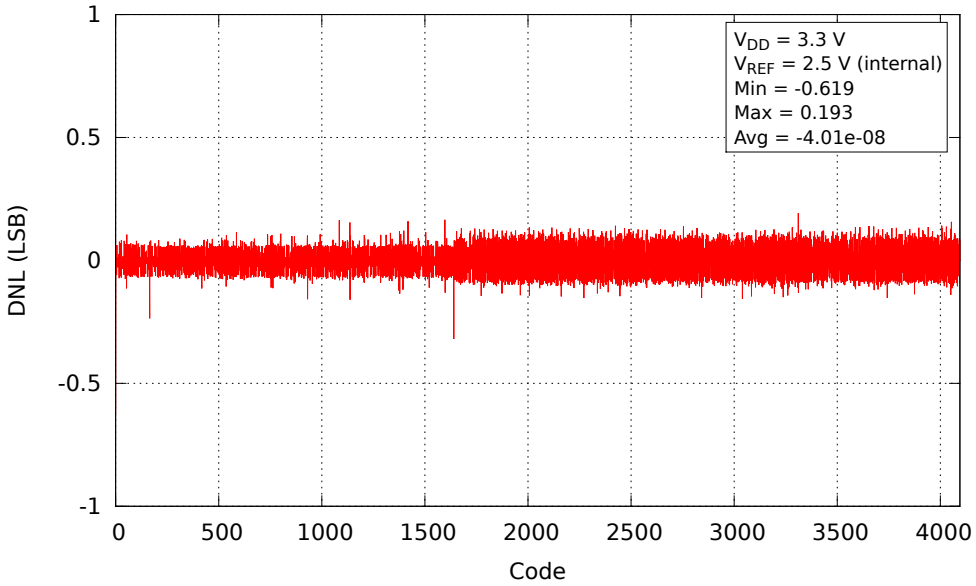

Figure 5.5: Plot of the DAC differential nonlinearity.

5.3.3 Frequency Domain Analysis

In addition to static measurements like INL and DNL, it is also important to do frequency domain analysis of the system. Frequency domain analysis can tell us how the energy of our signal is distributed over the different frequencies and the amount and distribution of unwanted noise and distortion. Since both the ADC and DAC may introduce noise and distortion, we measured them separately in addition to closed-loop measurement (the whole system). By measuring separately, we can more easily find out where the noise gets introduced into the system and thus it's easier to track down a noise source.

Two types of graphs are presented. The first is the dynamic performance of the component when the input/output sine wave signal has a frequency of 10 kHz and the second is the signal-to-noise ratio of the component as a function of frequency. This was also a measured using sine wave signal. We used the low pass filter described in section 4.6 for all the measurements. To calculate the frequency spectrum, we used a C library called FFTW (Fastest Fourier Transform in the West) [fftb]. This a well known open-source library that is licensed under GNU General Public License. The discrete fourier transform (DFT) requires a periodic signal (continuous endpoints) as input or else we will get spectral leakage where the power

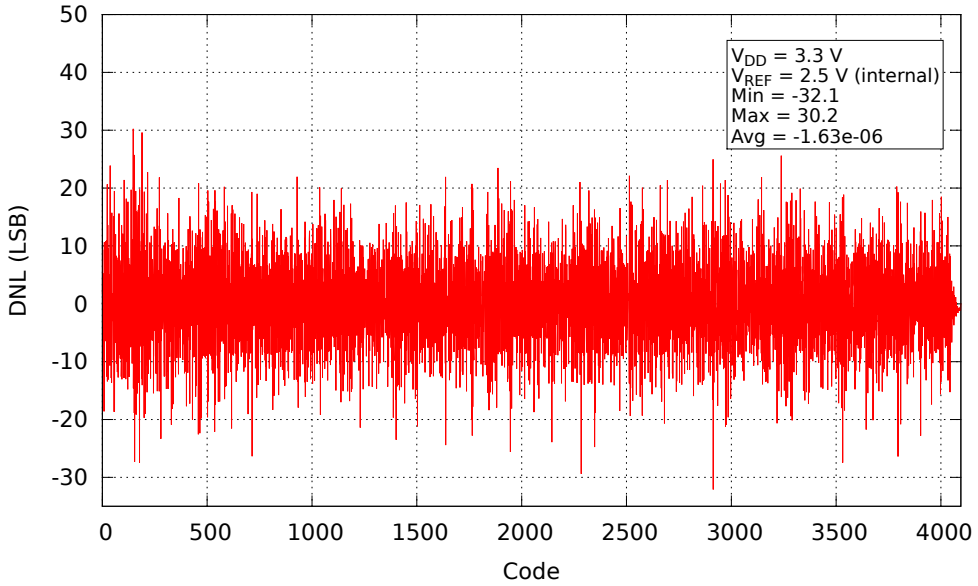

Figure 5.6: Plot of the ADC differential nonlinearity.

of the input signal will be spread over all the frequencies. Since the FFT algorithm is an implementation of DFT, we have to make sure that our input signal is periodic. We don't necessarily know if the acquired signal has an integral number of periods and we therefore have to apply a window function to the data points. There are several different window functions to choose from, all of the them with different pros and cons. We used the Hanning window since it is fairly easy to implement and it has good frequency resolution, spectral leakage is low and the amplitude accuracy is sufficient according to [ffta].

After the FFT was calculated, we divided the power spectrum into four different frequency components. These were DC, the fundamental frequency, harmonics and noise. Due to windowing and other reasons, the power of a frequency component is spread across several FFT bins. Because of this we used a frequency range to determine the power of the components. For example, if the fundamental frequency is 100 Hz, we might classify the frequencies from 95 Hz to 105 Hz as the fundamental frequency and use the sum of their powers as the fundamental frequency's power. The range was set manually, depending on the frequency resolution of the FFT. The frequency of all four components are defined pretty easily. The DC component, which is the average of the input, has a frequency of 0 Hz. The fundamental frequency f is the frequency we want. The harmonics

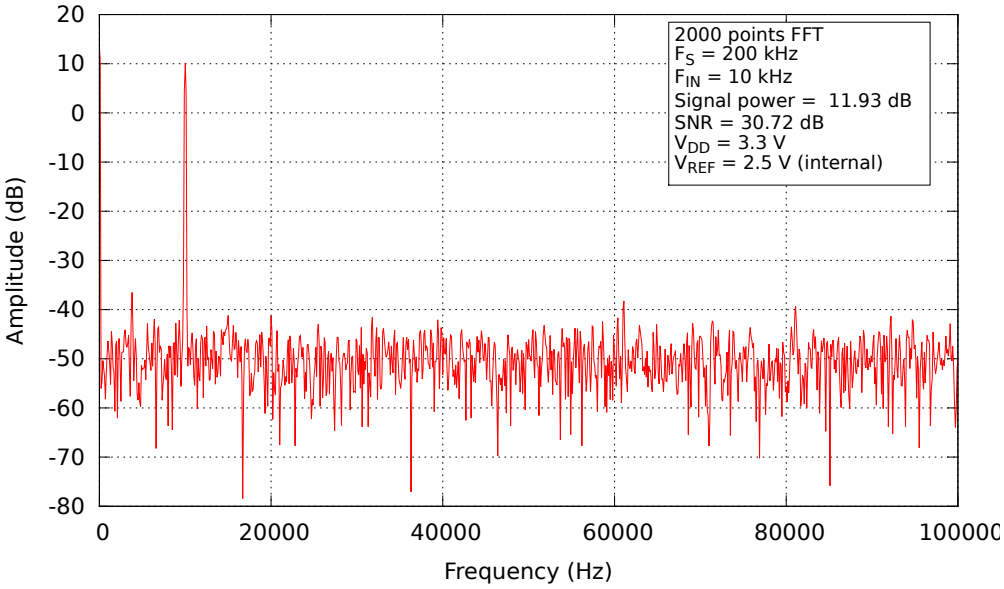

Figure 5.7: Dynamic performance of the DAC.

are the integer multiples of the fundamental frequency (the fundamental frequency is sometimes called the first harmonic). We only used the first four harmonics, i.e. $2f, 3f, 4f$ and $5f$. Lastly, the frequencies that was not a part of the DC, fundamental or harmonics components were classified as noise. The SNR value is defined as the power ratio between the fundamental frequency and the noise. A high SNR value means that it is easier to distinguish between the fundamental frequency and the noise compared to when the SNR value is low.

Digital-to-analog Converter

In figure 5.7, we can see the typical dynamic performance of the DAC. The measurements was taken using an oscilloscope with a maximum of 2000 datapoints and a sample frequency of 200 kHz. This gives a low frequency resolution (100 Hz), but it is sufficient. As we can see from the figure, the SFDR is around 50 dB and the SNR is almost 31 dB which is quite good. The noise looks like white noise and there is no obvious harmonic distortion which is good since this indicates that the DAC's transfer function is linear.

If we look at figure 5.8, we can see the SNR of the DAC as a function of the frequency. We only measured until the input frequency was 50 kHz since

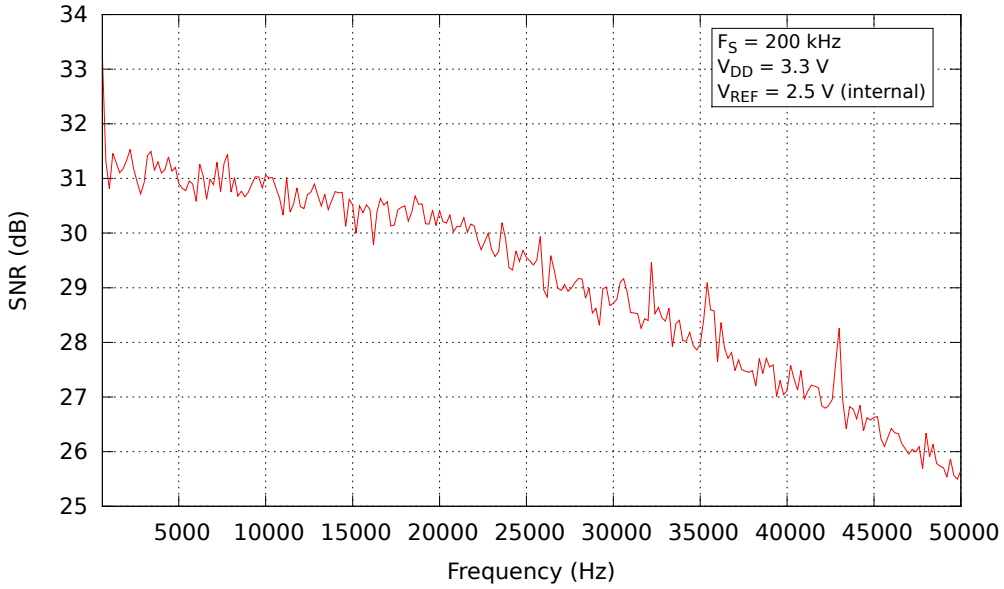

Figure 5.8: SNR vs. input frequency of the DAC.

this the around the Nyquist frequency of the ADC (which is 55.555 kHz). The SNR is decreasing as the frequency is increasing, which is probably due to the fact that the sine wave gets more distorted as the frequency increases and this causes the peak-to-peak amplitude to drop. The result is that the power of the fundamental frequency decreases towards the noise power and we get an SNR value that becomes smaller and smaller.

Analog-to-digital Converter

Testing the dynamic performance of the ADC was done by using a waveform generator connected to the ADC. The peak-to-peak amplitude was set to 2.5 V, the same as the ADC voltage reference. The maximum number of samples the Mecobo system can get is 16000 and the sample frequency is 111111.111 Hz. This gives us an FFT frequency resolution of approximately 7 Hz.

Figure 5.9 shows the the typical dynamic performance of the ADC. The SFDR is around 70 dB and the SNR is almost 55 dB, which is very good. As we can see, there are some periodic distortion or spurs. The distance between the spurs seems to be $\frac{1}{100}$ of the sample frequency, i.e. 1111 Hz. This might be deterministic jitter where the most likely causes either are cross-coupling or electromagnetic interference, or asymmetrical clock

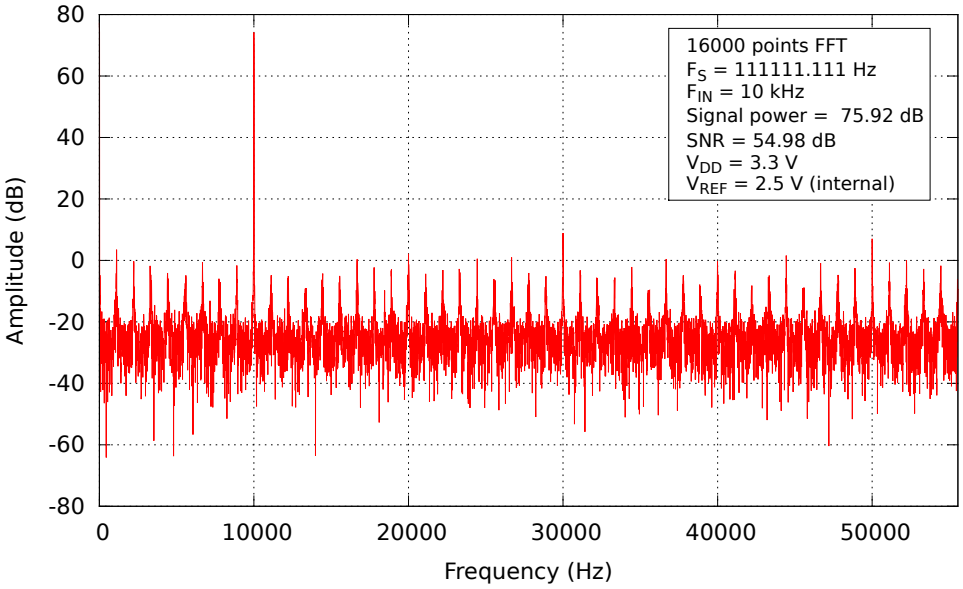

Figure 5.9: Dynamic performance of the ADC.

cycles.

Figure 5.10 shows the SNR of the ADC as a function of frequency. The SNR is quite high and stable, with only a small decrease as the frequency increases. The source of the fluctuations from 0 to 8 kHz is unknown.

Closed-loop

In the last measurements we tested the whole system by connecting the DAC to the ADC via a low pass filter. Figure 5.11 shows the dynamic performance at 10 kHz. The SFDR is around 55 dB and the SNR is 31 dB which is the same SNR value as the DAC has in figure 5.7. The noise floor has now increased by approximately 20 dB compared to figure 5.9 and the periodic distortion is more or less drowning in random noise. The INL and DNL measurements of the DAC (figure 5.3 and 5.5) shows that it is pretty accurate so it may look like the NCO is the limiting factor in the closed-loop.

The SNR versus frequency of the closed-loop is shown in figure 5.12. The fluctuations seems to be around 5-6 dB with some occasional spikes with larger values. It looks like some kind of period noise or distortion that reduces the signal-to-noise ratio. The general decrease is probably caused by

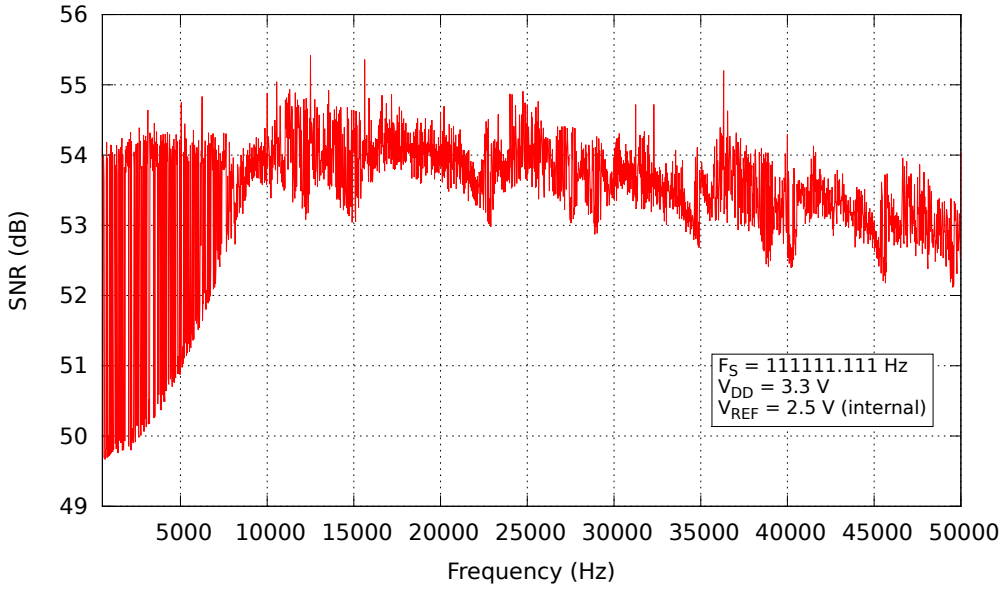

Figure 5.10: SNR vs. input frequency of the ADC.

the fact that the sine wave gets more and more distorted as the frequency increases, as mentioned in section 5.3.3.

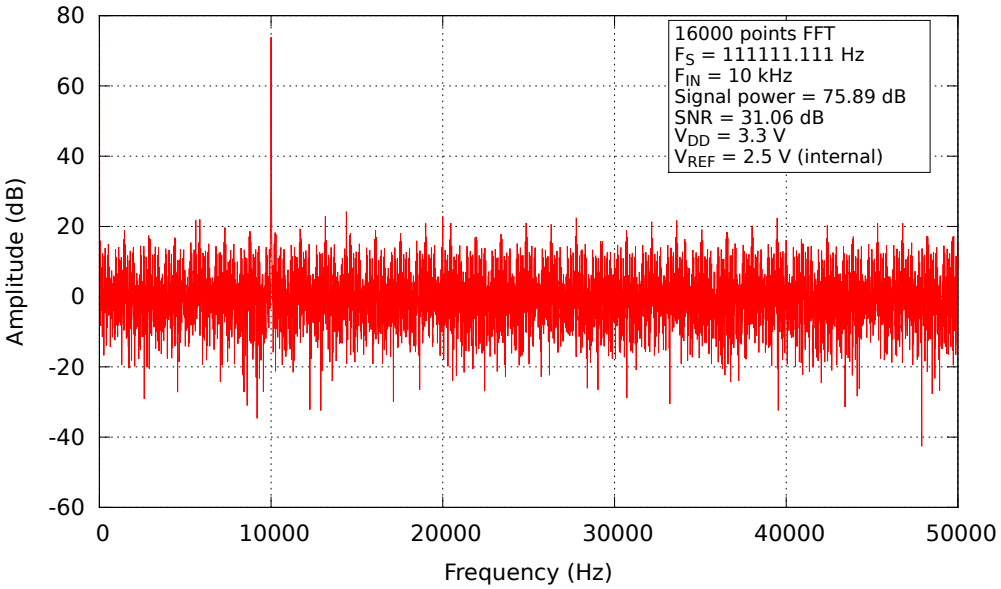

Figure 5.11: Dynamic performance of the closed-loop.

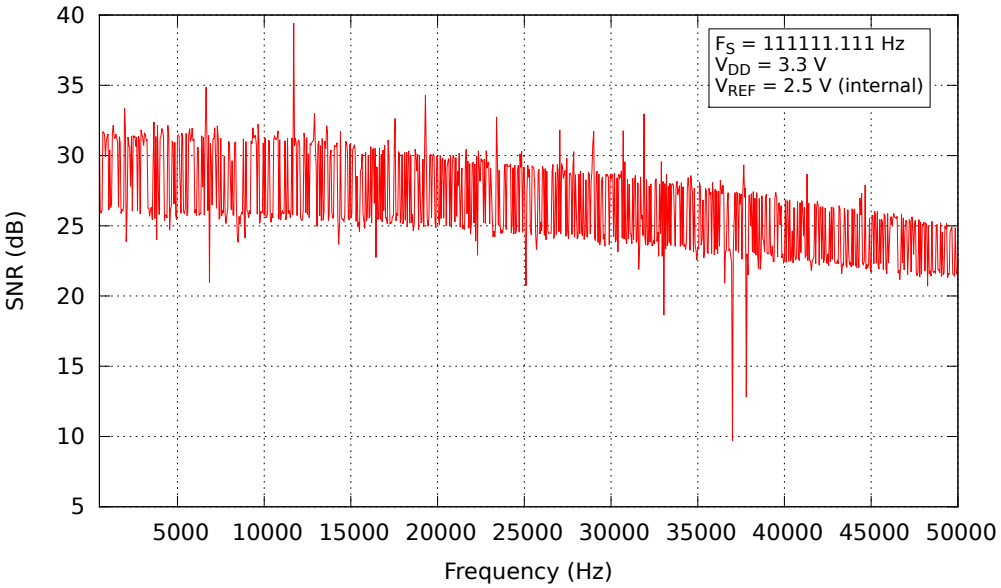

Figure 5.12: SNR vs. input frequency of the closed-loop.

6 Experiments

This chapter deals with initial experiments, where we look at the linearity/nonlinearity in the response from the material when applying analog signals to it. By applying different analog signals and changing their parameters, e.g. frequency, we can get an impression of the basic properties and behavior of the material. The material we used in the experiments was the carbon nanotubes, shown on the left side in figure 6.1. To the right is the multielectrode array that used to interface with the material. By identifying the main characteristics of the material, we might be able to tell what kind of functions the material is usable for and we might use it to narrow down the search space of the evolutionary algorithm by using this knowledge to set constraints on the evolution.

6.1 Trial and Error

One of the first things we did was to generate arbitrary signals with the Mecobo system and play around with the material by using different electrodes as input and output. The setup is shown in figure 6.2. The green circle illustrates the material and all the sides of the multielectrode array has electrode connections that may be defined as either input or output. By conducting this experiment, we got a picture of which electrode combinations that gave the best output, considering both amplitude and shape of the output signal.

The results we got was quite interesting. Looking at figure 6.3, we can see a screenshot of the oscilloscope when applying a 1000 Hz sine wave with a peak-to-peak amplitude of 2.5 V to the material. The signal at the bottom is the input and the signal at the top is the output from the

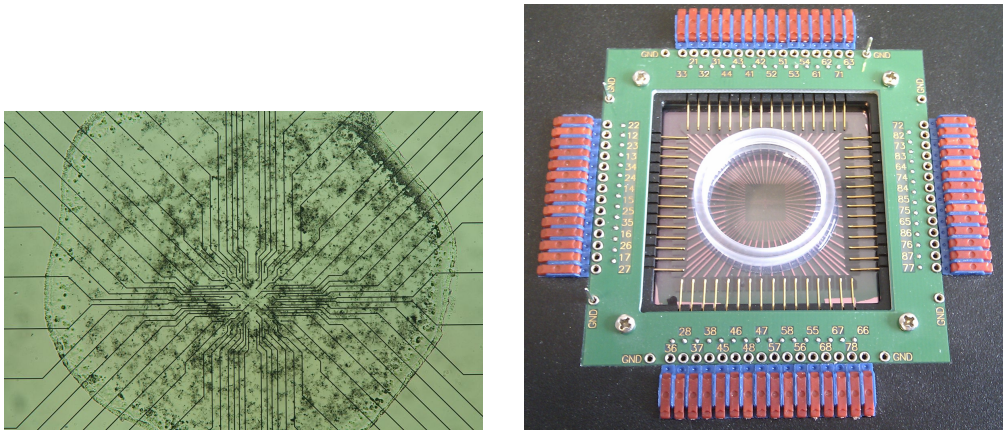

Figure 6.1: Carbon nanotubes and multielectrode array used in experiments.

material. The frequency of the output signal is very close to the frequency of the input signal and the peak-to-peak amplitude is 869 mV. As we can see, the shape of the output signal is different from the input signal. It resembles the input sine wave, but it has a quite steep incline from the bottom which decreases when it comes closer to the top.

We also tried two input signals where one was a 1000 Hz sine wave and the other was a 800 Hz sine wave, both with a peak-to-peak amplitude of 2.5 V. Figure 6.4 shows the response from the material. Using two input signals with different frequencies results in what looks like a weighted sum of the two inputs as shown in figure 6.5. More specifically, the response signal is close to the function

$$Out = c \times (x + 0.4y) \tag{6.1}$$

where x is the 1000 Hz input signal, y is the 800 Hz input signal and c is the gain. This functions is plotted in the right panel of the figure. Looking closer, we can see something that looks like a "swelling" on the graph. This is probably due to an oscillating effect in the material.

6.2 ADC Adding Noise To The Output

The next step was to use the ADC to sample the output signal instead of the oscilloscope. Figure 6.6 shows a screenshot of the oscilloscope before connecting the ADC. The input sine wave signal is at the bottom and the

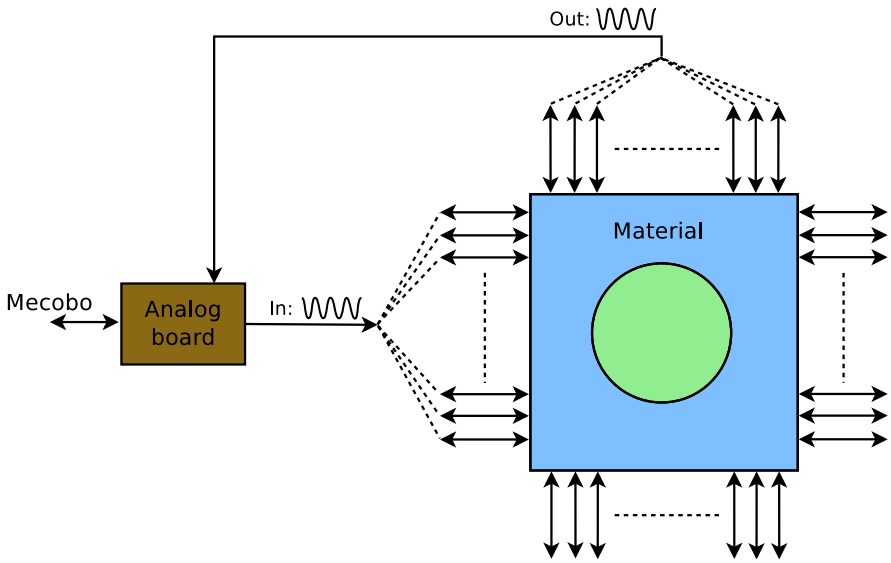

Figure 6.2: Setup used for trail and error experiments where the goal is to find the best input/output electrodes.

output signal is at the top. The output signal is close to the input signal in regards to frequency and shape. When connecting the ADC to the output signal, we get a high frequency noise signal as shown in figure 6.7. It may look like the ADC connection is enabling an oscillator circuit in the material that is sensitive to impedance.

6.3 Frequency Response

For the first experiment we wanted to see the response from the material when two sine waves was applied to it and we did a frequency sweep on one of them. Since the ADC added too much noise to the output signal, we used an oscilloscope instead to sample and measure the frequency. One input signal was set to a fixed frequency of 1000 Hz while the other input varied from 500 Hz to 1500 Hz. The setup is shown in figure 6.8. The number after the arrow is the electrode number.

Figure 6.9 shows the second frequency plotted against the output frequency. At 500, 1000 and 1500 Hz, the output frequency is what we expect when superpositioning two sine waves with these frequencies. Figure 6.10, 6.11 and 6.12 shows the output waveforms when channel 1 has a frequency of 500, 1000 and 1500 Hz, respectively. Again, the waves looks

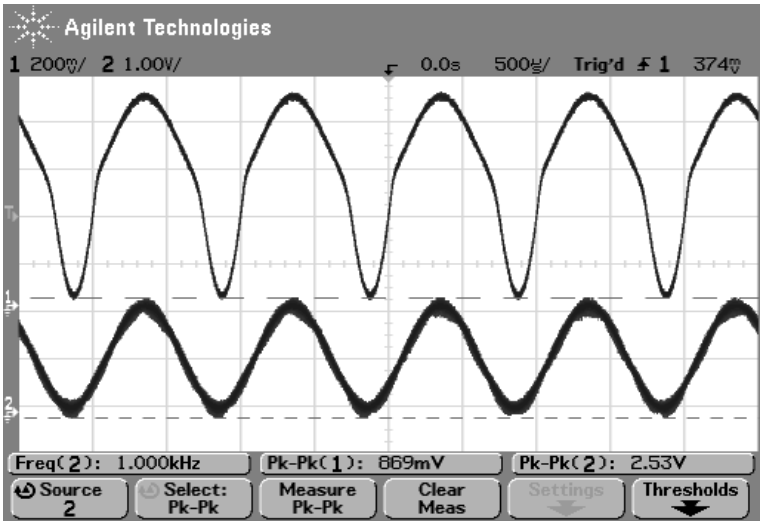

Figure 6.3: Oscilloscope screenshot showing the response of the material when applying a 1000 Hz sine wave as input.

like a weighted sum of the amplitude of the two input wave. The output frequency at 800 Hz is more interesting. Here, it suddenly increases to approximately 800 Hz, but same input frequencies was used in figure 6.4 where the output frequency was 200 Hz. There could be a lot of reason for this, e.g. different electrodes used, temperature etc. In the ranges from 500 Hz to 900 Hz and 1100 Hz and 1500 Hz, the output frequency is unstable with a lot of rapid fluctuations. The cause of this is unknown.

If we look closely at figure 6.10, 6.11 and 6.12 we can see that the small peaks are increasing and decreasing in amplitude. For example, in figure 6.12, the amplitude of left peak is increasing while the right peak is decreasing. This is probably the same effect as we saw in figure 6.4.

6.4 Phase Response

In the second experiment we wanted to see how the phase of the output signal changed when we changed the phase of one of the input signals. We did two experiments to see if the results from the first experiment changed if we changed input and output electrodes.

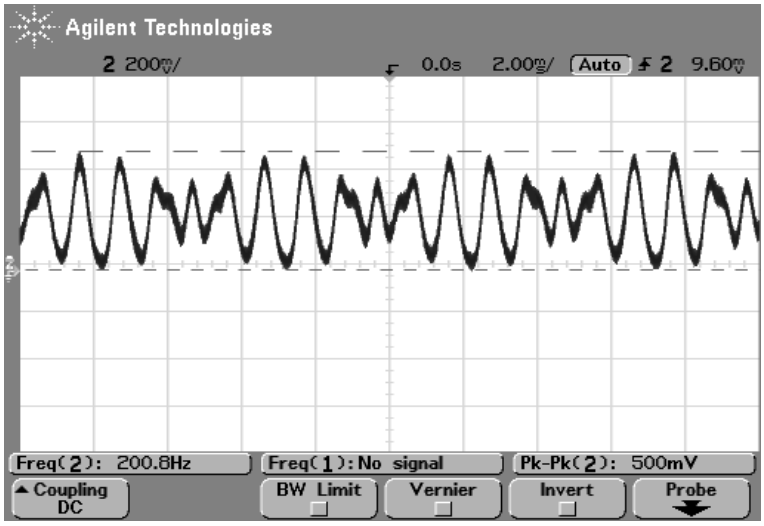

Figure 6.4: Oscilloscope screenshot showing the response from the material when applying a 1000 Hz sine wave and a 800 Hz sine wave as input.

6.4.1 Experiment 1

For the first experiment, we used the same electrodes as in the frequency response experiment (figure 6.8). Both channels had a fixed frequency of 1 kHz and channel 0 had no phase offset. On channel 1 we did a phase sweep from 0° to 360° . Figure 6.13 shows the results. The phase offset on channel 1 is on the x-axis and the phase difference between channel 1 and the output is on the y-axis. The phase difference increases almost linearly until approximately 170° and from approximately 190° it increases in the

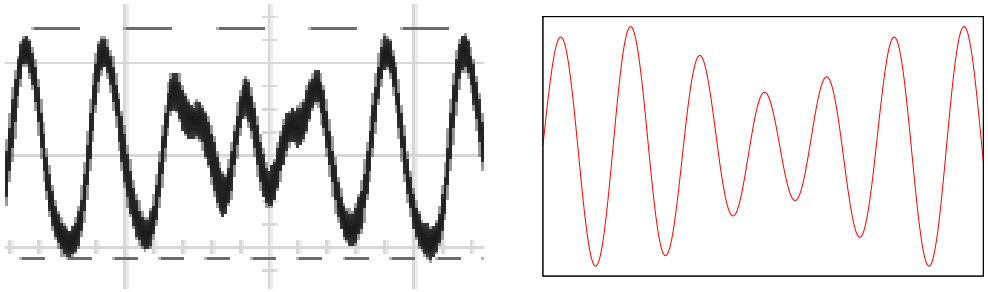

Figure 6.5: Left: response signal when applying 1000 Hz and 800 Hz sine waves to the material. Right: the function $(x + 0.4 * y)$ with x being the 1000 Hz input signal and y being the 800 Hz input signal. Illustration taken from [Tuf].

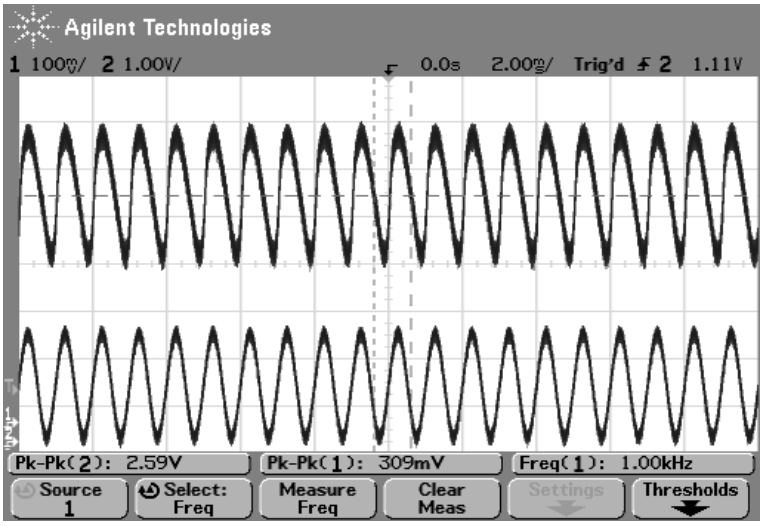

Figure 6.6: Oscilloscope screenshot showing material response without noise. The wave at the bottom is the input and the wave at the top is the output from the material.

same linear way. The phase difference is probably caused by a changing delay in the material or equivalently, different paths that the current takes. In the short interval between 170° and 190° , the phase difference goes from +160° to -160° with a lot of fluctuations in the middle. These fluctuations are probably caused by instability or an oscillating effect in the material.

6.4.2 Experiment 2

In the second phase response experiment, we changed the electrodes to see if we could get the same results as in experiment 1, but with different electrodes (channel 0 → 41, channel 1 → 44 and out → 43). The results can be found in figure 6.15. Here, the phase difference is much closer to a horizontal line which suggests that the properties of the material is nonuniform. Around 180° , we can see that the phase difference is very large, almost -1000° . The phase difference is calculated using the following equation [osc]:

$$\text{Phase difference} = \frac{\text{Delay}}{\text{Channel 1 period}} \times 360 \tag{6.2}$$

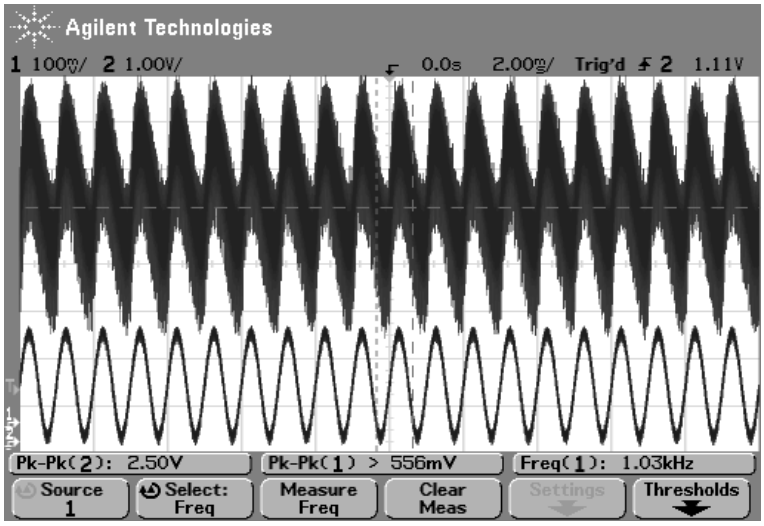

Figure 6.7: Oscilloscope screenshot showing material response with noise. The wave at the bottom is the input and the wave at the top is the output from the material with the ADC channel attached to it.

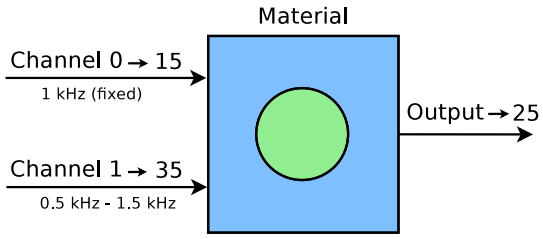

Figure 6.8: Frequency response experiment setup.

where the period and delay is measured according to figure 6.14. The delay can be calculated by rearranging equation 6.2:

$$\text{Delay} = \frac{\text{Phase difference}}{360} \times \text{Channel 1 period} \tag{6.3}$$

and with a phase difference of -1000° and a channel 1 period of $\frac{1}{1000\ \text{Hz}} = 1$ ms, this gives us a delay of -2.77 ms. A negative delay means that rising edge of the output occurred before the rising edge of channel 1. A delay of almost 3 ms is quite big and it is interesting to note that the maximum positive phase difference is only 400° .

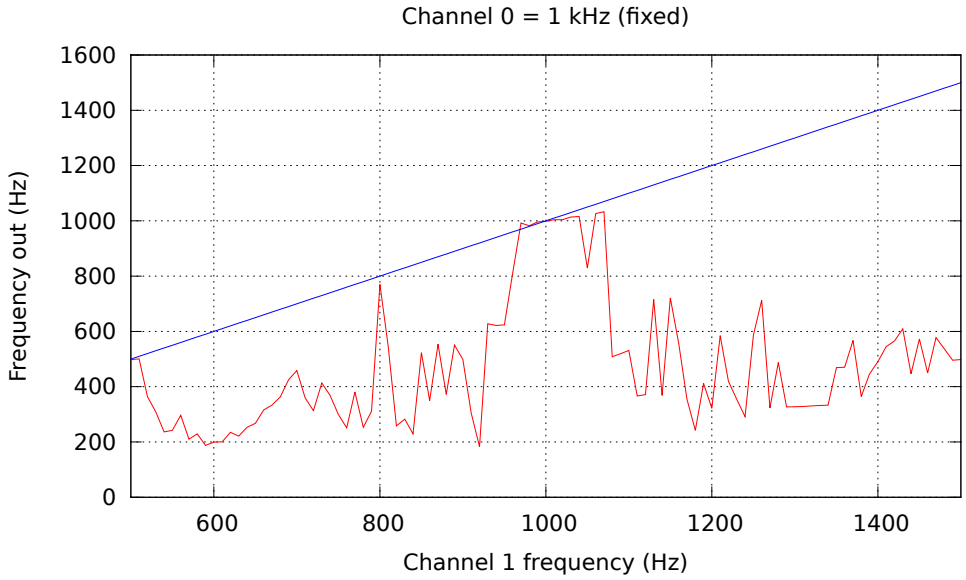

Figure 6.9: Frequency in vs. frequency out. One input was a fixed 1 kHz sine wave. The second sine waves input frequency is shown on the x-axis. The response from the material is the y-axis. Both input waves have a peak-to-peak amplitude of 2.5 V.

6.5 Amplitude Response

In the last experiment, we wanted to see how the peak-to-peak amplitude changes as a function of frequency. For this experiment we used a waveform generator from HP (see appendix C) because the signal from our waveform generator gets too distorted when the frequency gets above 60-70 kHz. We defined one input (84) and one output (74) on the material and did two frequency sweeps. The first one was from 1 kHz to 1 MHz and is shown in figure 6.16. We can see that the peak-to-peak amplitude decreases as the frequency increases.

The second frequency sweep was from 10 Hz to 2000 Hz and is shown in figure 6.17. Here, the curve increases rapidly at the start before it starts to flatten out. The material reduces the amplitude of the lower frequencies, just like conventional high-pass filters.

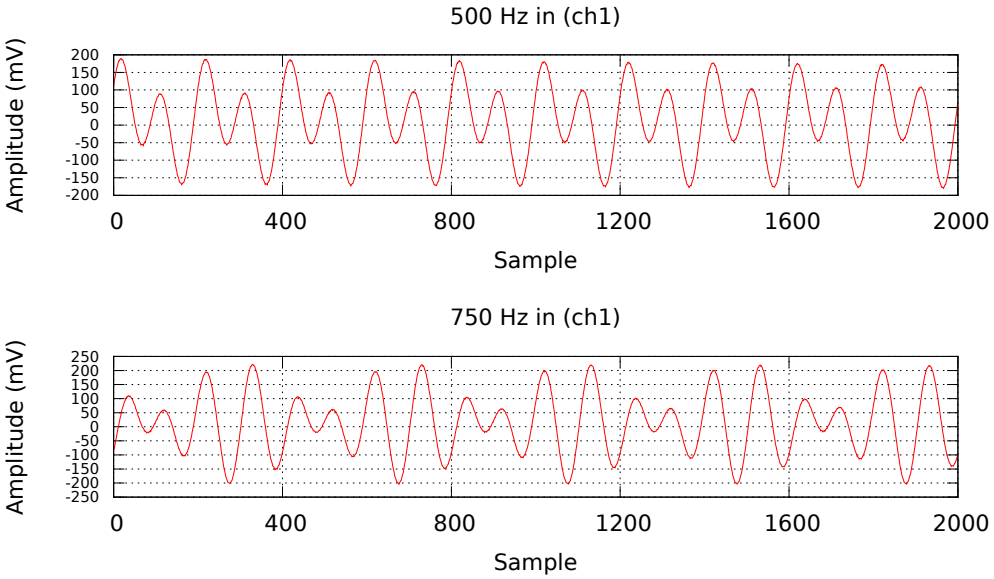

Figure 6.10: Top: output waveform when channel 1 has a frequency of 500 Hz. Bottom: output waveform when the frequency is 750 Hz.

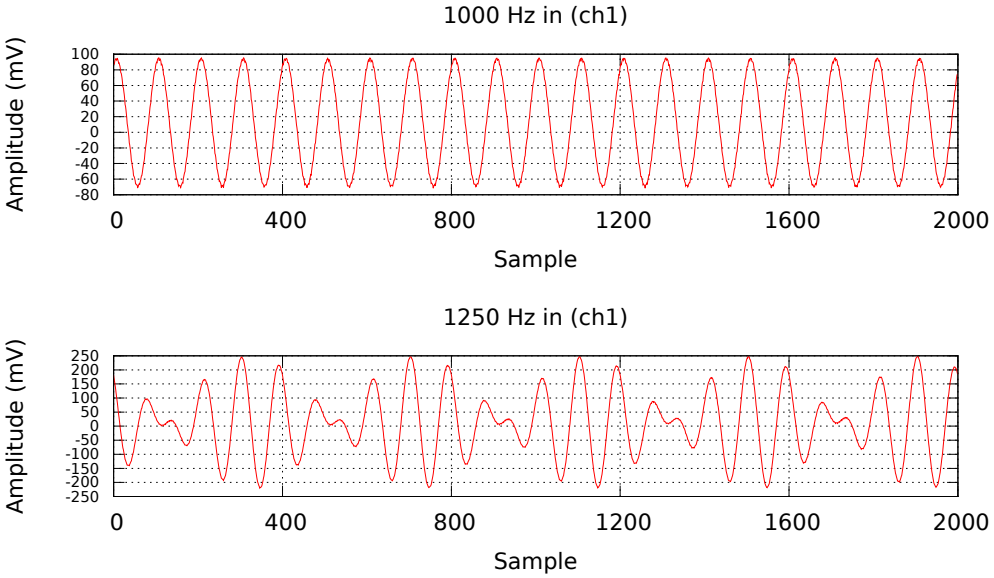

Figure 6.11: Top: output waveform when channel 1 has a frequency of 1000 Hz. Bottom: output waveform when the frequency is 1250 Hz.

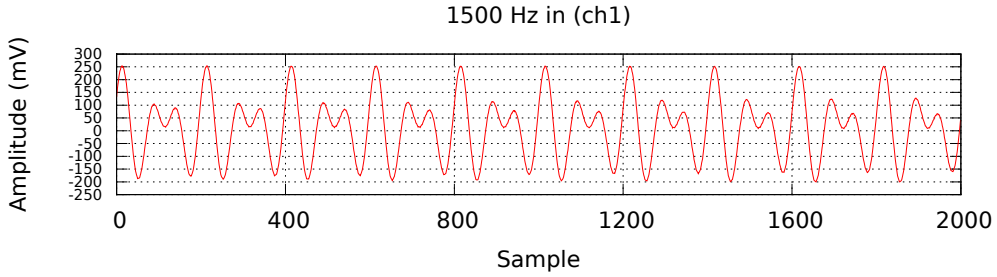

Figure 6.12: Output waveform when channel 1 has a frequency of 1500 Hz.

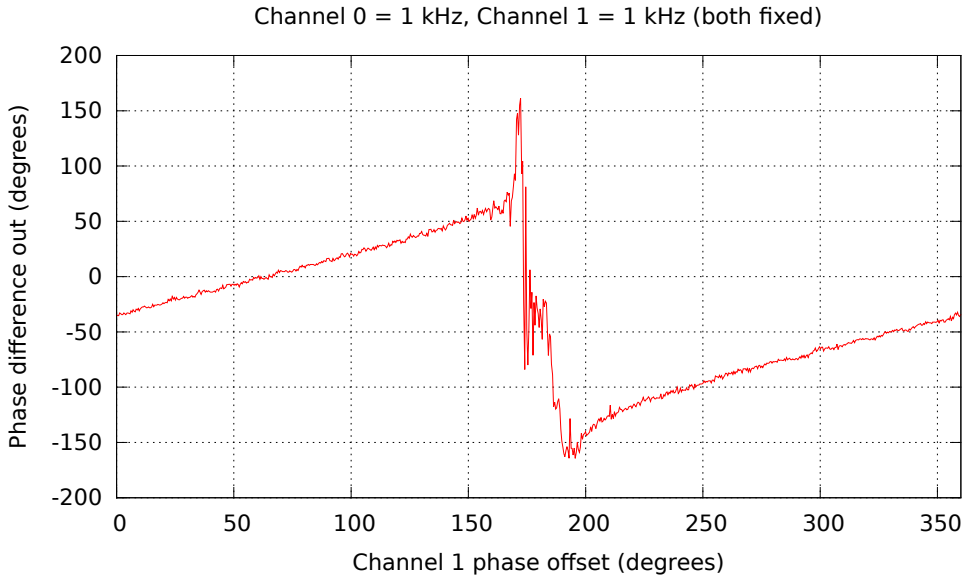

Figure 6.13: Phase response, experiment 1: channel 1 phase offset vs. the phase difference between channel 1 and the output.

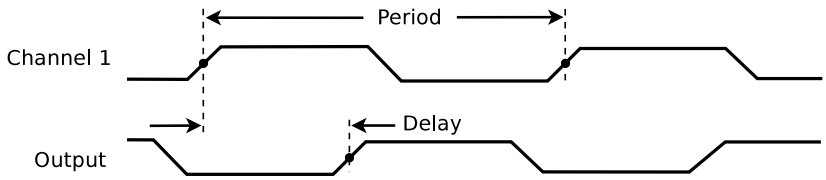

Figure 6.14: Period and delay measurements used for calculating the phase difference. Illustration taken from [osc].

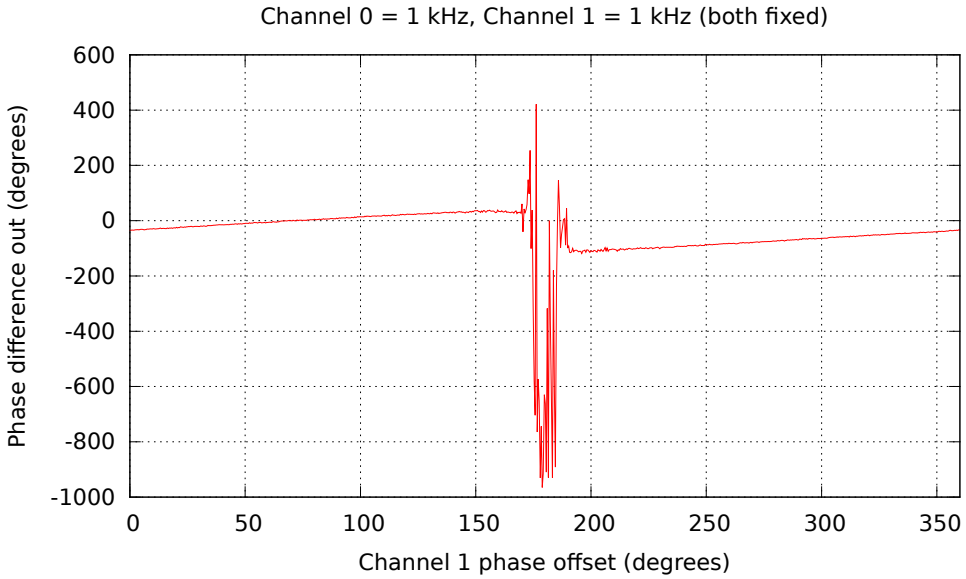

Figure 6.15: Phase response, experiment 2: channel 1 phase offset vs. the phase difference between channel 1 and the output.

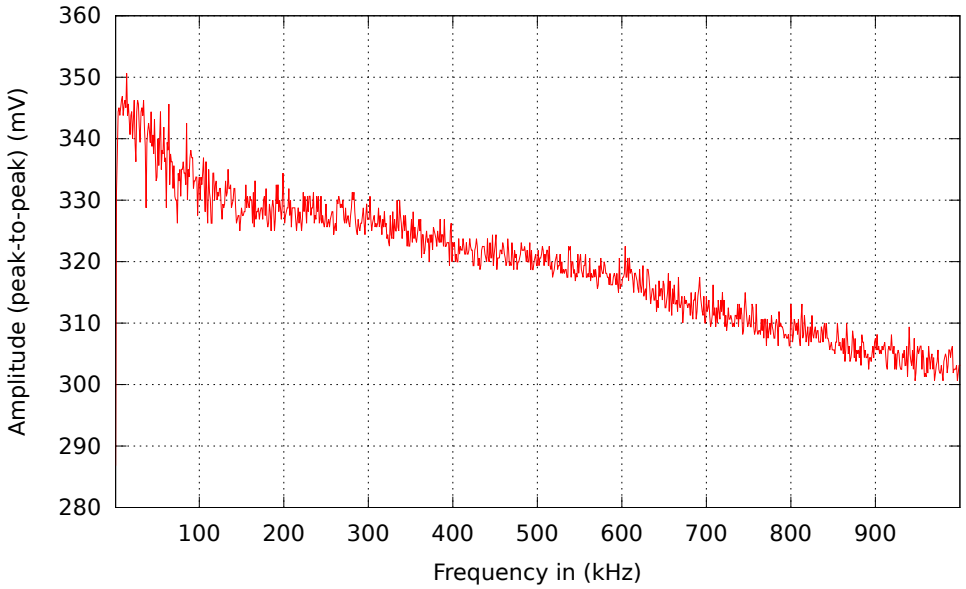

Figure 6.16: Input frequency vs. output peak-to-peak amplitude. The frequency sweep goes from 1 kHz to 1 MHz.

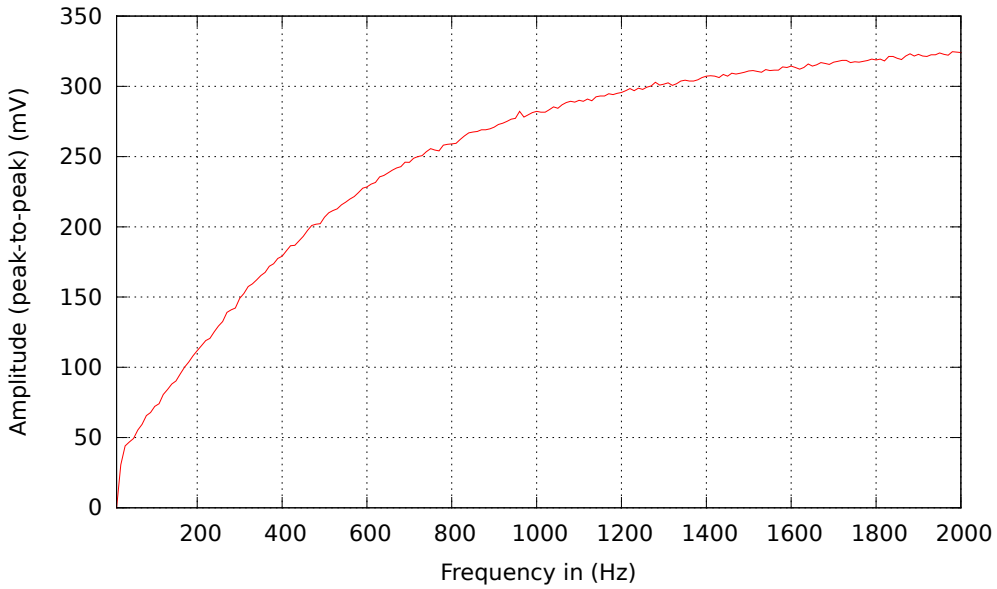

Figure 6.17: Input frequency vs. output peak-to-peak amplitude. The frequency sweep goes from 10 Hz to 1 kHz.

7 Conclusion

The purpose of this master's thesis was to create an extension to Mecobo that was able to generate dynamic signals and read the response from the material. This was successfully achieved by using digital-to-analog converters and a analog-to-digital converter together with the FPGA on Mecobo to create a direct digital synthesizer that works as an arbitrary waveform generator and to create a sampling system. The already existing C library called libEMB was also extended with new functionality to be able to control and change the wave parameters. The system requirements in section 3.2 have been met and the system performs reasonable well according to the measurements in chapter 5. The measurements gives us an indication on when experimental results becomes uncertain due to the limited accuracy of the system. Some noise problems with an unknown source(s) seems to be present, but this should not be a problem as long as we are aware of it.

We did some initial experiments to investigate the response from the carbon nanotubes. By applying analog signals to the material, we were able to get some interesting responses showing that the material is capable of letting the signals propagate almost unchanged through it and capable of influencing the signal (see figure 6.6 and 6.3 for examples). The experiments also show that it is possible to influence the signal externally. The experiments does not give an answer on whether the carbon nanotubes can be programmed or not.

The material may be equivalent to a configurable network of resistors and capacitors. If this is true, the increase and decrease in the amplitude of dynamic signals causes the capacitors to remain discharged and this puts the network in a sort of high impedance state. If this hypothesis is true, it means that response we measure from the material actually is the voltage

change in the input(s) signal, i.e. the measured response is $V_{out} = dV_{in}/dt$ [Tuf].

7.1 Further Work

There are several improvements that can be done to the extension, especially to the NCO. One improvement could be to add a dither to the NCO, between the phase accumulator and the truncation. The dither adds white noise to the $W+1$ least significant bits of the PA value, to reduce the phase truncation spurs. Over time the average of the truncated value will be close to the PA dither value and the error will be random (or pseudorandom) [dds].

Another improvement is to find a new method for changing the sine wave amplitude. The current method only shifts the amplitude to the right to reduce it, but this is not the best way since the amplitude is halved for each shift. This gives few amplitude levels. The best way would be to use an embedded division core, but this requires a new FPGA.

The way we sample the ADC could be more parallel. Now, the ADC SPI controller has to wait while the user module is writing a newly acquired sample to memory. Instead it could start a new sample request while the user module are doing memory operations. This way, the ADC could run at full speed with a 125 kSPS throughput rate instead of the current 111 kSPS.

The constructed NCO uses a straight forward architecture to create the waves, with a sine look-up table that is shared between the wave generators. To make the generators more independent of each other and not be constrained by a shared resource, a new architecture like modified Sunderland architecture or Taylor series approximation could be used to reduce the needed look-up table size [Van96]. With a reduced look-up table size each wave generator can have its own look-up table. The penalty is of course more logic to compensate for the smaller tables, but this is a trade-off that one has to consider.

The DA/AD converters clock signal is currently generated by the FPGA by using a counter to reduce the FPGA clock frequency before it is sent to the DA/AD converters. This can be a source of jitter in the DA/AD converters performance because of phase distortion in the generated clock

signal. Instead, one could use an external clock signal that generates a more reliable clock signal.

Bibliography

[adc] AD7888 Datasheet. http://www.analog.com/static/imported-files/data_sheets/AD7888.pdf.

[BG] Doug Brooks and Dave Graves. Current Carrying Capacity of Vias. http://www.pcbcic.com/docs/viacurrents1.pdf.

[Car93] Peter Cariani. To Evolve an Ear: Epistemological Implications of Gordon Pask's Electrochemical Devices. 1993.

[dac] AD5686R/AD5685R/AD5684R Datasheet. http://www.analog.com/static/imported-files/data_sheets/AD5686R_AD5685R_AD5684R.pdf.

[dds] A Technical Tutorial on Digital Signal Synthesis. http://www.ieee.li/pdf/essay/dds.pdf.

[Dow10] Keith L. Downing. Introduction to Evolutionary Algorithms. 2010.

[ffta] Application Note - Understanding FFT Windows. http://www.physik.uni-wuerzburg.de/~praktiku/Anleitung/Fremde/AN014.pdf.

[fftb] Fastest Fourier Transform in the West (FFTW). http://www.fftw.org.

[Har06] Simon Harding. *Evolution in Materio*. PhD thesis, University of York, Feb 2006.

[Hey] Francis Heylighen. The Science of Self-Organization and Adaptivity.

[HP12] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Elsevier, Fifth edition, 2012.

[JHJ10] Larry Bull Ben De Lacy Costello Julian Holley, Andrew Adamatzky and Ishrat Jahan. Computational Modalities of Belousov-Zhabotinsky Encapsulated Vesicles. 2010.

[KB03] Sanjeev Kumar and Peter Bentley, editors. *On Growth, Form and Computers*. Elsevier Limited Oxford UK, 2003.

[Lyk10] Odd Rune Strømmen Lykkebø. Design and implementation of a prototype platform for evolution in materio. Master's thesis, NTNU, Jul 2010.

[max] INL/DNL Measurements for High-Speed Analog-to-Digital Converters (ADCs). http://www.maximintegrated.com/app-notes/index.mvp/id/283.

[mul] Agilent 34410A/11A user's Guide. http://cp.literature.agilent.com/litweb/pdf/34410-90001.pdf.

[nas] NAnoSCale Engineering for Novel Computation using Evolution (website). http://nascenceproject.blogspot.no/p/background.html.

[osc] Agilent 54621D/22D/41D/42D Mixed-Signal Oscilloscopes - User's Guide. http://cp.literature.agilent.com/litweb/pdf/54622-97036.pdf.

[pcb] Pcb trace width calculator. http://www.circuitcalculator.com/wordpress/2006/01/31/pcb-trace-width-calculator/.

[Sip02] Moshe Sipper. The Emergence of Cellular Computing. *Computer*, 32(7):18–26, 2002.

[SLHR06] Julian F. Miller Simon L. Harding and Edward A. Rietman. Evolution in Materio: Exploiting the Physics of Materials for Computation. 2006.

[Tho96] Adrian Thompson. An evolved circuit, intrinsic in silicon, entwined with physics. 1996.

[Tuf] Gunnar Tufte. NASCENCE Draft Experimental Work. Draft, NTNU.

[Van96] Jouko Vankka. Methods of Mapping from Phase to Sine Amplitude in Direct Digital Synthesis. In *IEEE International Frequency Control Symposium*, 1996.

[wik] Wikipedia - dbc. http://en.wikipedia.org/wiki/DBc.

Appendices

A

Test Plans

A.1 System Tests

A.1.1 libEMB

read_voltage

Generate 1000 random numbers D, where $0 \leq D \leq 4095$. Convert them to voltage ($V = \frac{2.5}{4096}D$). Use a DC power supply to output the voltages, then read back the voltage using `read_voltage`. For each random number, take 16000 samples. Calculate the average of the 16000 samples and check if it is between ±100 of the original value (due to the lower resolution of the power supply used and ADC measuring error).

enable_daisy_chain, write_dac

Call `enable_daisy_chain` . Then write 1000 random voltages to DAC 1 using command `DAC_CMD_WRITE_INPUT_REG` (0001) and channel A. The LDAC state should also be random (0,1 or 2). Verify with a voltmeter that the output on the channel changes depending on the LDAC state and that the measured voltage is between ±5 of the original voltage. Repeat for DAC 2 and DAC 3. If this test succeeds, it's assumed that all other DAC commands will also succeed.

reset_dac

Write and output 2.5 V ($D = 4095$) on channel A on all three DACs. Connect the `RSTSEL` pins to ground. Verify with a voltmeter that the voltage on all three DACs falls down to 0 V.

enable_wave, set_wave_config

Call `enable_wave` and write an arbitrary configuration to all three

channels (channel group 0: A1, A2 and A3). Verify with an oscilloscope that a waveform is present on each channel, and that it has the correct frequency and amplitude.

set_channel_group

Do the same as in the test above, but also change the channel group. Channel group 0 gives output on A1, A2 and A3. Channel group 1 gives output on channel A1, A2, A3, B1, B2 and B3. Channel group 2 adds the C channels and lastly, group 3 adds the D channels.

disable_wave

Use `enable_wave` and `set_wave_config` to output an arbitrary waveform. Use an oscilloscope to verify that the waveform disappears when `disable_wave` is called.

Function	Passed	Comment
`read_voltage`	Yes	None
`enable_daisy_chain`	Yes	None
`write_dac`		
`reset_dac`	Yes	None
`enable_wave`	Yes	None
`set_wave_config`	Yes	None
`set_channel_group`	Yes	None
`disable_wave`	Yes	None

Table A.1: libEMB function tests

A.2 FPGA Design Tests

A.2.1 SPI DAC Controller

Test Description	Passed	Comment
Write to 1, 2 and 3 DACs with 100 randomly generated DAC configurations each and verify that the correct output is clocked out on the serial output line (MOSI). Also verify that $\overline{\text{LDAC}}$ has the correct value and that $\overline{\text{SS}}$ stays low during transmission.	Yes	None
Execute daisy-chain command and verify that the serial output line outputs correct command. Also verify $\overline{\text{LDAC}}$ and $\overline{\text{SS}}$	Yes	None
Execute the reset DACs command and verify that the reset line goes low.	Yes	None

Table A.2: SPI DAC controller tests

A.2.2 SPI ADC Controller

Test Description	Passed	Comment
Generate 100 random ADC configurations (MOSI) and 100 random ADC output values (MISO) and verify that the controller transmits and receives the values correctly. Also, verify that $\overline{\text{SS}}$ goes low during transmission.	Yes	None

Table A.3: SPI ADC controller test

A.2.3 Sine LUT

Test Description	Passed	Comment
Iterate through all addresses (0 $\rightarrow$ 1023) and verify that the correct sine value is presented at the data out line.	Yes	None

Table A.4: Sine look-up table test

A.2.4 Sine LUT Wrapper

Test Description	Passed	Comment
Generate 1000×12 random addresses, and verify that the correct sine value is presented on the 12 different data out lines.	Yes	None

Table A.5: Sine look-up table wrapper test

A.2.5 Configuration Register

Test Description	Passed	Comment
Write 1000 random DAC/ADC configurations and check the output.	Yes	None

Table A.6: Configuration register test

A.2.6 Wave Configuration Register

Test Description	Passed	Comment
Write 1000 random wave configurations and check the output.	Yes	None

Table A.7: Wave configuration register test

A.2.7 Wave Memory

Test Description	Passed	Comment
Write 1000 random wave configurations and check the output.	Yes	None

Table A.8: Wave memory test

A.2.8 Wave Generator

Test Description	Passed	Comment
For sawtooth, triangle and square waveforms, generate waves with the following amplitude and offset: 1. Amp: 0, Off: 0 2. Amp: 2048, Off: 0 3. Amp: 2048, Off: 2048 4. Amp: 4095, Off: 0 5. Amp: 4095, Off: 2048 6. Amp: 4095, Off: 4095 For the sine waveform: 1. Amp: 12, Off: 0 2. Amp: 1, Off: 0 3. Amp: 1, Off: 2048 4. Amp: 0, Off: 0 5. Amp: 0, Off: 2048 6. Amp: 0, Off: 4095 Use a frequency of 10 kHz. Write the output to files and plot it using a plotting program. Verify visually that the waveforms are not deformed.	Yes	No deformation other than what one would expect (due to the way the waves are made).

Table A.9: Wave generator test

A.2.9 Sample Register

Test Description	Passed	Comment
Set enable line to high and low and verify that the busy line changes.	Yes	None
Enable the module and write the values $0 \rightarrow 11$ to *s_data_0* $\rightarrow$ *s_data_11*, i.e. write the id to the corresponding sample data line. When channel group is 0, verify that *spi_data* outputs: 1. $\overbrace{\underbrace{1}_{cmd}\ \underbrace{1}_{ch}\ \underbrace{002}_{id}\ \underbrace{0}_{padding}}^{DAC3}\ \overbrace{110010}^{DAC2}\ \overbrace{110000_{hex}}^{DAC1}$ Channel group 1 should give: 1. $1100201100101100 00_{hex}$ 2. $1200501200401200 30_{hex}$ Channel group 2 should give: 1. $1100201100101100 00_{hex}$ 2. $1200501200401200 30_{hex}$ 3. $1400801400701400 60_{hex}$ Lastly, channel group 3 should give: 1. $1100201100101100 00_{hex}$ 2. $1200501200401200 30_{hex}$ 3. $1400801400701400 60_{hex}$ 4. $1800B01800A01800 90_{hex}$ The first 8 bits are command and channel, then comes 12 bits of id/data and the last 4 bits are padding.	Yes	None

Table A.10: Sample register tests

A.2.10 Wave Controller

Test Description	Passed	Comment
Write 1 and 0 to the enable register and verify that the enable output line changes to the value in the register.	Yes	None
Enable the module and toggle the busy input line. Verify that the multiplexer select line has the correct value.	Yes	None
Write 100 random values to the channel group register and verify that the channel group out line has the same value.	Yes	None

Table A.11: Wave controller tests

A.2.11 Wave Module

Test Description	Passed	Comment
Enable the module. Verify that the busy line line goes high. Disabling the module should make them go low.	Yes	None
Set the channel group to 3. Configure every wave generator with an arbitrary waveform. Split up the the value on the SPI data line into three values and write the output to file. Plot the data and verify visually that the waveforms are not deformed.	Yes	None

Table A.12: Wave module tests

A.2.12 Toplevel

To test the toplevel module (called `mecobo_with_analog_board`) the tests for testing individual modules were used. When testing the toplevel the user module and memory was also tested. This was done by writing data and commands to memory and verify that the user module executed the commands correctly.

Test Description	Passed	Comment
Command CMD_WRITE_1	Yes	None
Command CMD_WRITE_2	Yes	None
Command CMD_WRITE_3	Yes	None
Command CMD_READ	Yes	None
Command CMD_DCEN	Yes	None
Command CMD_RESET	Yes	None
Command CMD_ENABLE_WAVE	Yes	None
Command CMD_DISABLE_WAVE	Yes	None
Command CMD_WAVE_CONF	Yes	None
Command CMD_WAVE_GROUP	Yes	None

Table A.13: Toplevel tests

B

Finite State Machines

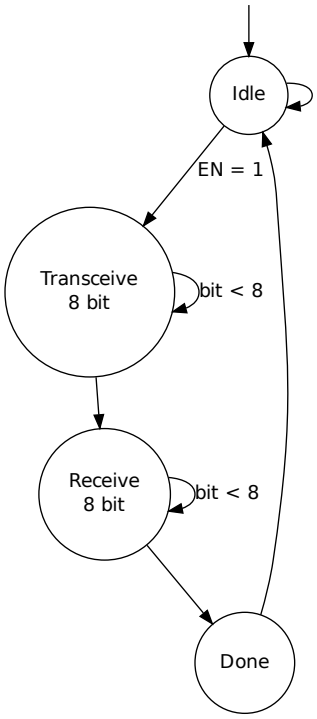

Figure B.1: FSM of the ADC SPI controller.

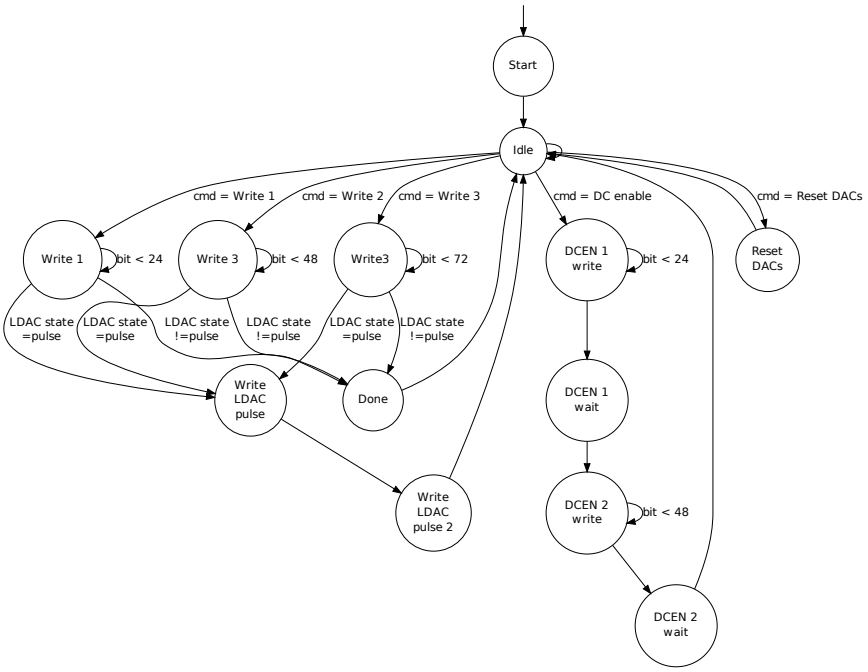

Figure B.2: FSM of the DAC SPI controller.

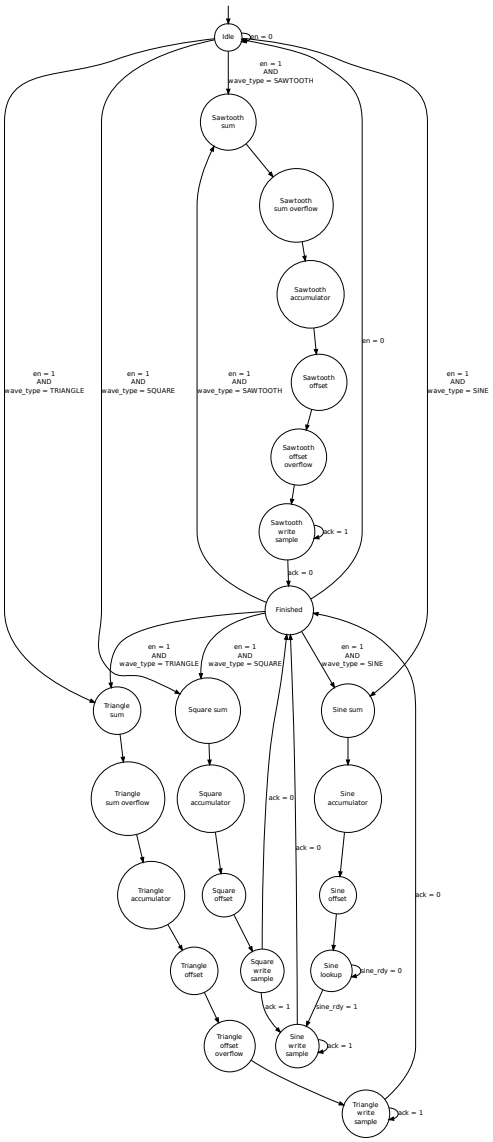

Figure B.3: FSM of the NCO.

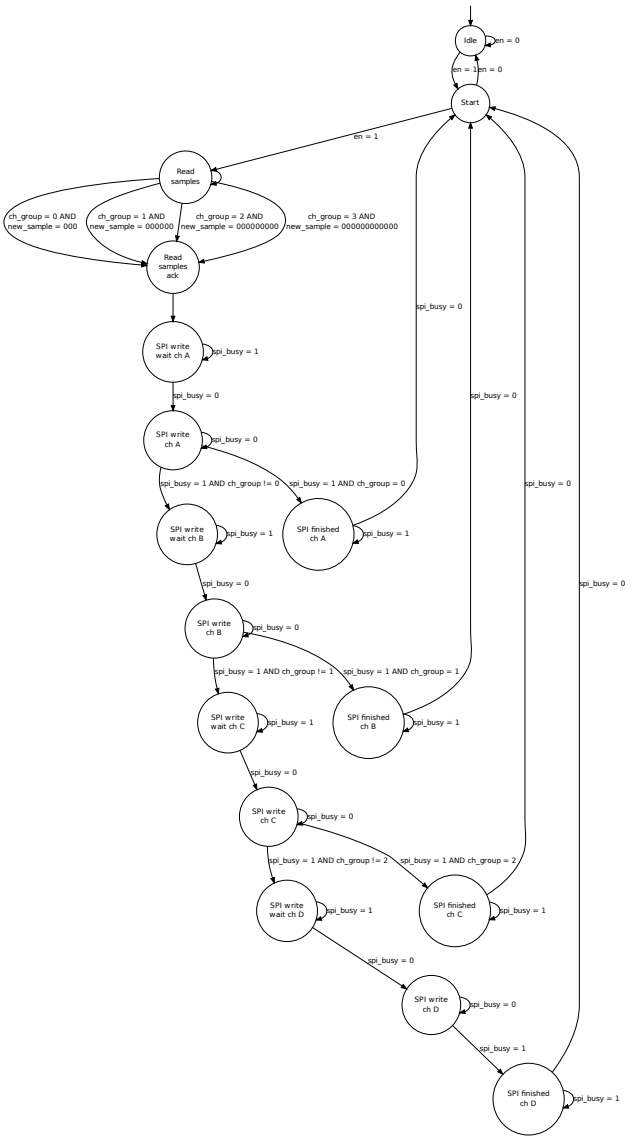

Figure B.4: FSM of the sample register.

C

Test Equipment

Type	Manufacturer	Model
DC power supply	Agilent	6612C
Multimeter	Agilent	34410A
Waveform generator	Hewlett Packard	33120A
Mixed signal oscilloscope	Agilent	54622D

Table C.1: Test equipment

D

Bill of Materials

RefDes	Description	Value	Qty.	MPN
U1/U2/U3	Digital-to-analog converter		3	AD5684RBRUZ
U4	Analog-to-digital converter		1	AD7888ARUZ
C1/C3/C5 C7/C9/C11 C13/C15	Capacitor	10μF	8	TCJA106M016R0200
C2/C4/C6 C8/C10/C12 C14/C16	Capacitor	100 nF	8	MCCA000416
R1	Resistor		1	
R2	Resistor		1	
R3	Resistor		1	
R4	Ferrite bead		1	
R5/R6 R7/R8	Resistor		4	
R9	Resistor		1	
R10/R11/R12	Resistor		3	

Table D.1: Bill of materials for the analog board

E

Schematics

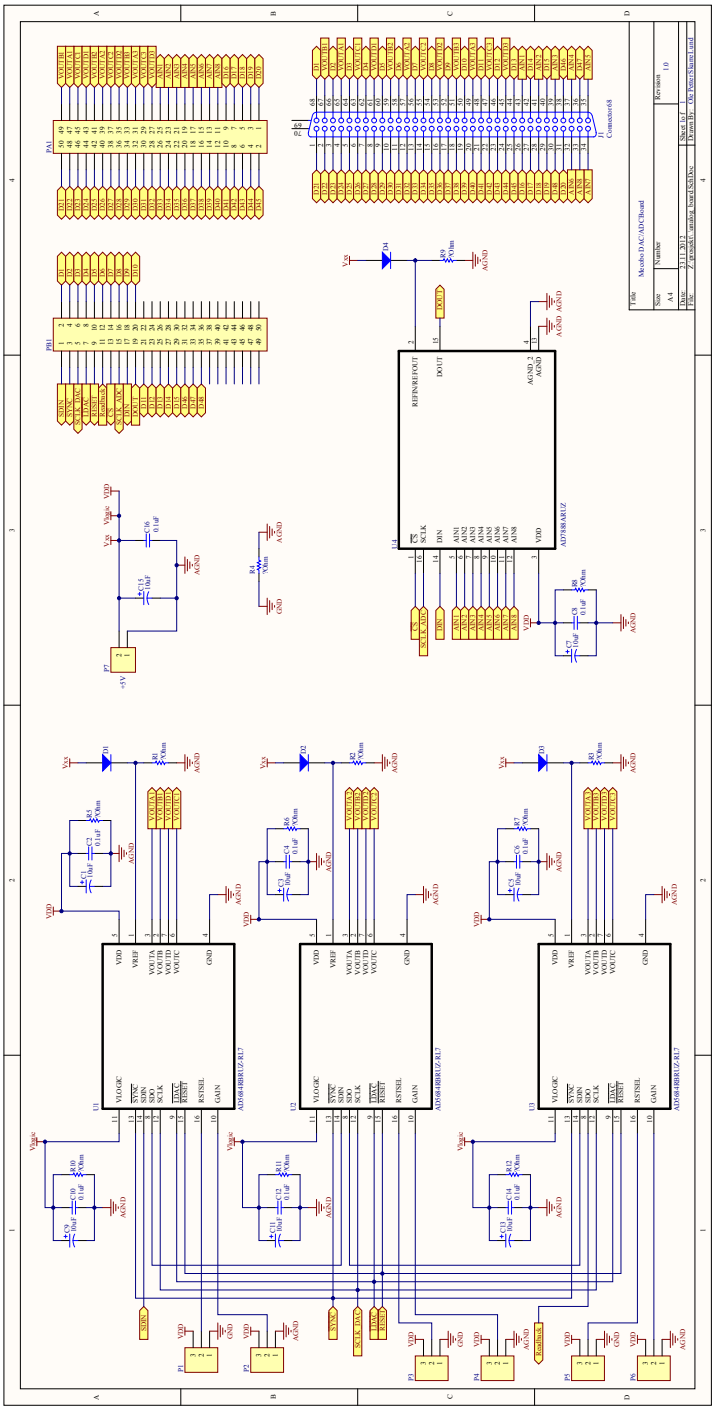

Figure E.1: Analog board PCB schematic.