*MSc thesis*

# Point to point wireless audio with limited bandwidth and processing

Even Steen Brenden
*evenstee@stud.ntnu.no*

Supervisor: Lars Lundheim
*lundheim@iet.ntnu.no*

Department of Electronics and Telecommunications
Norwegian University of Science and Technology
June 7, 2012

**Abstract**

Designing an audio communication system for an embedded platform requires an analysis of both general audio processing techniques and the specific characteristics of the platform. Therefore this is a task that must be handled differently for different platforms, although many aspects of the design process can be generalized.

This report describes such an analysis. A general-purpose audio compression scheme is designed for optimizing audio quality given bounds on processing resources and bit rates. Essential system design aspects such as analog-to-digital and digital-to-analog conversion, scheduling for real-time performance, error resilience and number representation is considered. A low-cost, fully self-contained real-time wireless audio communication system is designed and implemented.

# Contents

# Introduction

The project covered by this report investigates the implementation of an audio communication system on a particular System-on-Chip (SoC) that can transmit data point-to-point over radio. Limited processing, storage and transmission resources and latency restrictions will pose bounds on application possibilities and audio quality. Focus will be on utilizing these resources by finding and implementing an audio communication system that operates in real-time. The resulting implementation will then function as an assessment on what the SoC in question is able to achieve in real-time. We will strive to make use of the features present on-chip without the need for external devices so that the SoC is fully self-contained, effectively lowering production costs.

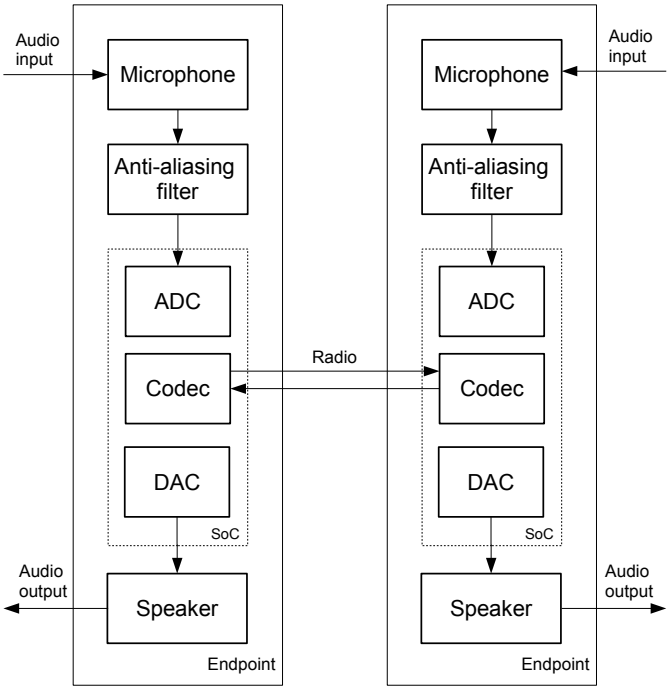


Figure 1: A typical application of the system described in this project.

Several applications for the system will be discussed. One of these is illustrated in figure 1. It is a full-duplex (both directions simultaneously) communication system using two identical endpoints with audio input and output and radio signal transmission and reception. The endpoints could for example be headsets with microphones.

An overview of the signal chain of the system is shown in figure 2. The

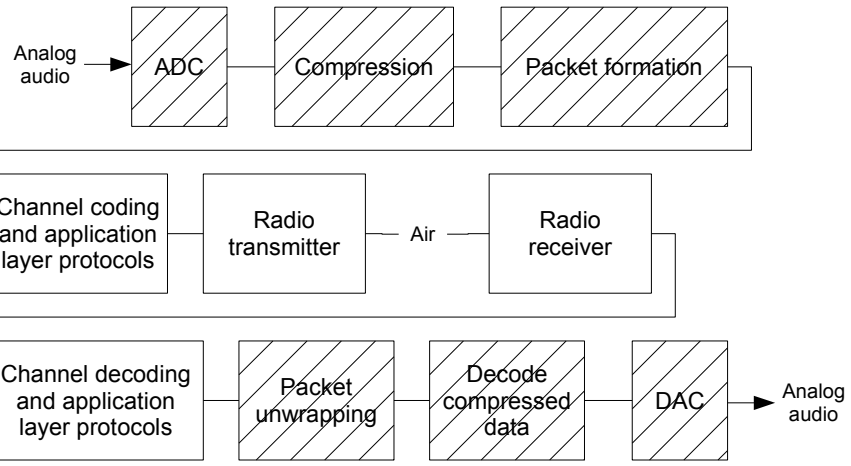


Figure 2: Signal path. This project will mainly be concerned with the shaded blocks.

shaded blocks are the ones this project will be concerned with: the quality of the analog-to-digital conversion (ADC), audio compression, packet formation for transmission over radio on the transmitter side, then unpacking, decoding and digital-to-analog conversion on the receiver side. Figure 2 shows half-duplex operation (one way at a time). For full-duplex operation both endpoints must be both a transmitter and receiver. The scope of the project is primarily in the software domain. No hardware development will be carried out.

A project going in-depth on the radio part of the SoC was done in parallel with this project and is found in [1]. Also, a project that analyzes what compression algorithms are possible for real-time operation on the SoC is found in [2].

This report is written so that it can be used as a guide for doing a similar project. It is organized as follows: First, the embedded system is presented. Then a compression scheme is chosen and explained. Number representation and arithmetics is discussed, followed by a chapter on how to conceal errors. Then the analog-to-digital converter is assessed and an approach to on-chip digital-to-analog conversion is discussed. This is followed by an presentation

and discussion of a system implementation and a performance assessment on this is carried out. Finally, a set of possible applications is discussed.

This report is accompanied by a ZIP file containing source code and audio files. Details on this is presented in appendix E and F, respectively.

# Chapter 1

# System-on-Chip

This chapter will present the main components and properties of the System-on-Chip in question [3]. Since this project ultimately concerns what applications are possible to implement on this specific SoC, the possibilities and constraints that are described here will set the tone for all following chapters. The main components are illustrated in figure 1.1.

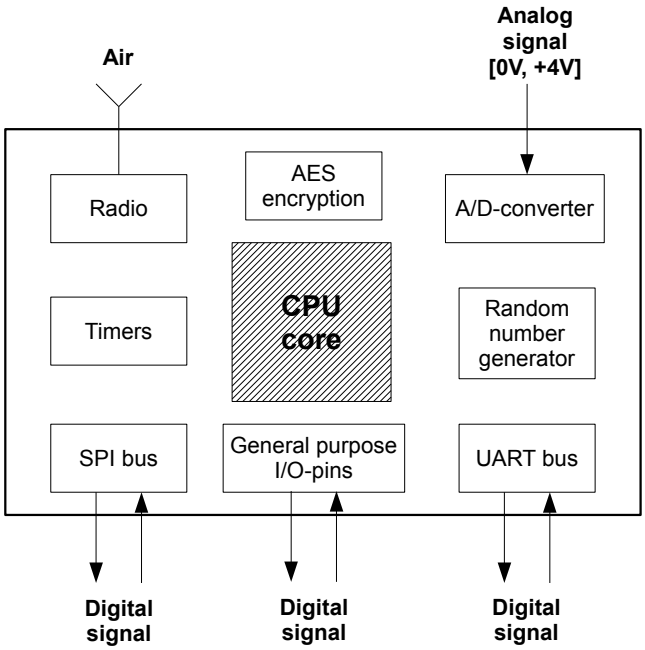


Figure 1.1: System-on-Chip main components.

## 1.1 SoC physical properties

The following is a list of the main characteristics of the SoC [3]. These will be discussed below.

- A CPU clocked at $f_C = 16$ MHz with an efficiency of $E_C = 0.9$ MIPS/MHz. This gives an average of $\overline{M} = f_C E_C = 14.4$ MIPS (Million Instructions Per Second).
- 32-bit fixed-point arithmetic.
- 1-cycle hardware multiplier.
- 256 kB flash program memory.
- 16 kB RAM.
- Delta-sigma analog-to-digital converter (ADC) with hardware dithering and four modes, as shown in table 1.1.
- 2.4 GHz half-duplex radio transmitter and receiver (RF) with three transmission rates $R_R$: 250 kbps, 1 Mbps or 2 Mbps.
- Both the radio (RF) and the CPU core are designed for very low power consumption.

Table 1.1: ADC modes in terms of bit depth and sample rate.

| **Mode** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|
| Bit depth $B_A$ [bits] | 8 | 9 | 10 | 12 |
| Sample rate $f_s$ [kHz] | 50 | 28 | 14.7 | 3.8 |
| Bit rate $R_A = B_A f_s$ [kbps] | 400 | 252 | 147 | 45.6 |
| Bandwidth $f_b = \frac{f_s}{2}$ [kHz] | 25 | 14 | 7.35 | 1.9 |
| Dynamic range [dB] | 48.16 | 54.18 | 60.2 | 72.24 |

At first sight, one of the main constraints of the SoC is the CPU with the given $f_C$ and fixed-point arithmetics only. Throughout this project we will attempt to find the what audio processing algorithms the processor can run in real-time for given bit rates. Arithmetic operations are normally executed in one cycle. The 1-cycle hardware multiplier is very helpful in achieving this. As will be shown in chapter 8, the flash program memory is more than large enough for our use. Therefore the compiler may be set to produce so-called branch-free code by repeating code segments. This is beneficial

because branching is one of the instructions that may use more than one cycle to execute [4]. An assessment on what algorithms may be run for each ADC mode given the constraint on CPU power is described in [2].

The issues when dealing with fixed-point arithmetic for signal processing is discussed in chapter 3. In short, they do not pose a problem for designing the system, but they do make the implementation a bit more complex.

A potential threat on audio quality is the ADC. It has four operating modes, which is shown in table 1.1. It is a general purpose ADC that is not specifically designed for audio sampling, although it may be used for such purposes. Chapter 5 will make an assessment on what audio applications it is suitable for. An analog anti-aliasing filter is not present on-board pre-ADC, so any input must be filtered so that its bandwidth is no larger than $f_b = \frac{f_s}{2}$. The dynamic ranges in table 1.1 is calculated as $6.02B_A + 1.76$ dB [5, p. 125]. For comparison, CD quality audio has a dynamic range of 98.08 dB [6, ch. 1].

The radio is half-duplex, meaning that at any time it can either transmit or receive, but never both at once. However, an overlaying protocol emulating full-duplex operation exists for this SoC [1]. That protocol is intended for infrequent and sporadic packet transmission, which is not the case for an audio transmission system. This project will mainly investigate the use of this protocol for half-duplex (one-way) audio transmission without any overlying application layer protocol, although some tests for operating in full-duplex will also be carried out. Protocol design will not be discussed. The reason for having different radio transmission rates $R_R$ to choose from is that a lower $R_R$ will produce a higher bit energy, which again makes for more robust transmission. Notice that all uncompressed bit rates $R_A$ in table 1.1 are less than all radio transmission rates $R_R$ except when $R_R = 250$ kbps. This may seem like a reason not to compress at all, but because other devices in the same RF band may use the same air space, it is likely that there will be a lot of packet losses accompanied by retransmissions. Therefore, these theoretical limits that are never fully reached in practice. Lowering the compressed rate $R_C$ will give more time to ensure reliable transmission [1].

The SoC has built-in routines for power management. This project will focus on the limited processing resources, not on energy consumption. Therefore power saving schemes will not be addressed. The CPU may execute power management routines that are transparent to the programmer, such as putting the system to sleep during idle times. However, since a real-time audio application has constant, frequent and periodic execution, there will probably not be much time to sleep.

## 1.2 Possible compression schemes

For all ADC modes except mode 4, [2] shows that most MPEG-standard codecs will not be able to run real-time on this device. This is because there are not sufficient processing resources for doing real-time operation using the sample rate $f_s$ of its respective mode. [2] further makes an assessment on what sort of general compression algorithms can be run full-duplex in real-time for the four modes. These are summarized in table 1.2.

Table 1.2: List of compression schemes and the ADC modes they can perform full-duplex real-time operation in using our SoC.

| **Compression scheme** | **Possible modes** |
|---|---|
| MPEG Layer-3 (MP3) | Mode 4 |
| MPEG-4 AAC | Mode 4 |
| MPEG-4 AAC-Low-Complexity | Modes 3 and 4 |
| ADPCM | All modes |
| DPCM | All modes |

# Chapter 2

# Compression and coding

The aim of compression is to reduce the bit rate for transmission while minimizing distortion. For this project there is no definite goal on which rate to achieve [7]. We merely want to utilize the given processing power and then make an assessment on rate versus distortion for a codec running on the SoC. The compression scheme should be of general purpose, since we are investigating several applications for sound sources with different statistical properties. With this in mind, [2] makes an assessment of the SoC and chooses DPCM as a basic compression scheme. This scheme will have optional adaption features that can be used to challenge the restrictions on computational resources with the benefit of enhancing audio quality for a fixed bit rate or vice versa. The theory behind DPCM and ADPCM will be presented in this chapter. Focus will be on the features that gave the best results with respect to audio quality in [2].

## 2.1 Symbols for rates, bits and bandwidths

Before diving into compression, the system symbols and their place in the signal path is shown in figure 2.1. These will be used throughout this report. $R_A$ is the bit rate from the ADC with bit depth $B_A$. $f_s$ and $f_b$ is the sample rate and signal bandwidth, respectively. These remain constant through the signal path. $R_C$ is the compressed bit rate with bit depth $B_C$. $R_R$ and is the rate at which the radio transmits. See [1] for a discussion on other metrics for the radio. $B_A$ will also be used when referring to general signal bit depths.

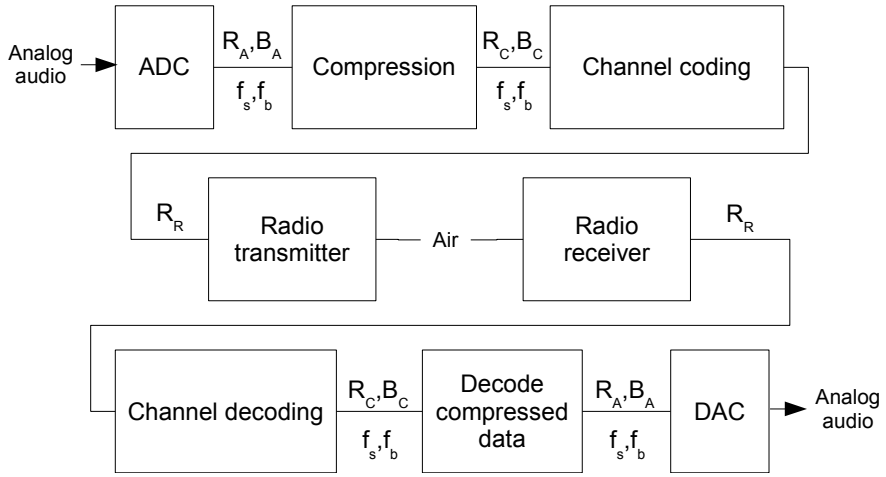


Figure 2.1: Signal path with symbols.

## 2.2 DPCM

Differential Pulse-Code Modulation, as described in [5, ch. 6], uses linear prediction to remove signal redundancy. A closed-loop DPCM encoder and decoder is illustrated in figures 2.2 and 2.3, respectively.

Equations 2.1-2.3 describes a DPCM system. It consists of the following signals: the input signal $x(n)$, the predicted signal $\hat{x}(n)$, the reconstructed signal $\tilde{x}(n)$, the error signal $d(n)$ and the quantized error signal $e(n)$, which is the signal that is to be transmitted from encoder to decoder.

$$d(n) = x(n) - \hat{x}(n) \tag{2.1}$$
$$e(n) = d(n) - q(n) \tag{2.2}$$
$$\tilde{x}(n) = \hat{x}(n) + e(n) \tag{2.3}$$

The concept is as follows. Let $x(n)$ be represented by $B_A$ bits. A good prediction will produce a $d(n) < x(n)$. Then $d(n)$ may be quantized with $B_C < B_A$ bits to produce $e(n) = d(n) + q(n)$, where $q(n)$ is the quantization error. An example of $x(n)$ versus $d(n)$ is shown in figure 2.4.

The decoder produces $\tilde{x}(n)$, which is a reconstruction of $x(n)$. This is the signal that is heard at the receiver side. Provided an error-free channel, the only error in this DPCM system is the one introduced by the quantizer, $q(n)$. Note that $q(n)$ need not necessarily be additive as in equation 2.2; the relation between $d(n)$ and $e(n)$ depends on the quantizer type. For a uniform

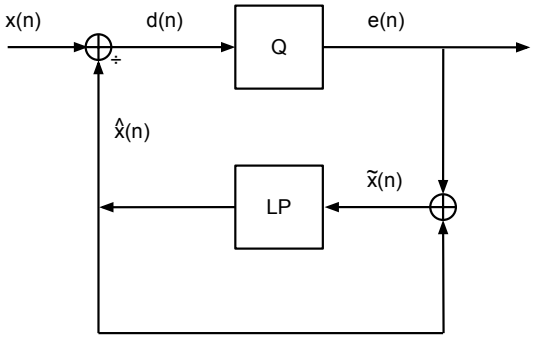


Figure 2.2: A DPCM encoder.

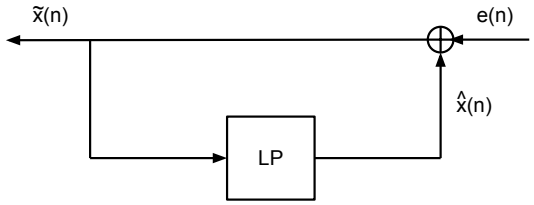


Figure 2.3: A DPCM decoder.

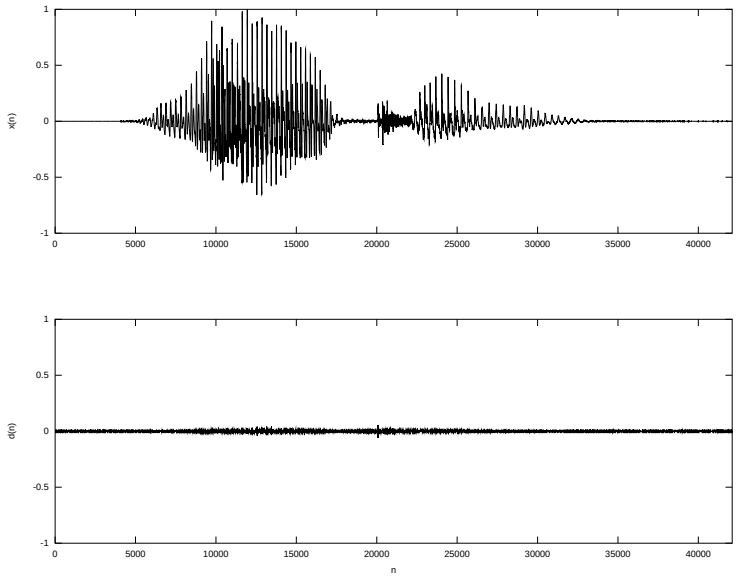


Figure 2.4: Example of input signal $x(n)$ versus residual signal $d(n)$.

quantizer with $B_C \geq 3$, however, it is common to treat it as additive [8, ch. 4].

### 2.2.1 Prediction

To design an efficient DPCM coder, we need a way of calculating good predictor $\hat{x}(n)$. Equation 2.4 defines $\hat{x}(n)$ as a linear predictor for $\tilde{x}(n)$ of oder $P$ with a set of prediction coefficients $\{\alpha_i\}_{n\in\{1,2,\ldots,P\}}$.

$$\hat{x}(n) = -\sum_{i=1}^{P} \alpha_i \tilde{x}(n-i) \tag{2.4}$$

Let $x(n)$ be a $M$-sample sequence for $n = 0, 1, \ldots, M-1$. A common way of measuring prediction error power is by defining

$$\sigma_d^2 = E[|d(n)|^2] = E[|x(n) - \hat{x}(n)|^2], \tag{2.5}$$

which is to minimized. $E$ is an expectation operator applied to $\{x(n)\}$. Using equations 2.4 and 2.5 one can find an equation for the optimal $\alpha_i$'s with respect to a minimal $\sigma_d^2$. These can be found by solving the Yule-Walker equations (2.6).

$$\sum_{i=0}^{P} \alpha_i R_{|j-i|} = \mathbf{M}\boldsymbol{\alpha}_P = \mathbf{0}, \tag{2.6}$$

where $\mathbf{M}$ and $\boldsymbol{\alpha}_P$ is

$$\mathbf{M} = \begin{bmatrix} R_1 & R_0 & R_1 & \ldots & R_{P-1} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ R_P & R_{P-1} & \ldots & \ldots & R_0 \end{bmatrix}, \boldsymbol{\alpha}_P = \begin{bmatrix} 1 \\ \alpha_1 \\ \ldots \\ \alpha_P \end{bmatrix} \tag{2.7}$$

This is merely a set of linear equations based on

$$R_k = \frac{1}{M} \sum_{n=-0}^{M-|k|-1} x(n)x(n+|k|), \tag{2.8}$$

the autocorrelation function of $x(n)$. A fast and elegant method for solving the Yule-Walker equations is Levinson-Durbin recursion, an inductive method that uses $\alpha_1, \ldots, \alpha_{k-1}$ to calculate $\alpha_k$. The Levinson-Durbin algorithm is presented in appendix A.

In the simplest form of DPCM, $P = 1$ and $\alpha_1 \in (0, 1)$, typically $\alpha_1 = 0.9$. Optimal $\alpha_i$ values may be found offline by analyzing larger data sets with

representative audio inputs $x(n)$, for instance with the method discussed above, or online using adaptive schemes. Adaptive schemes are more computationally demanding than static schemes. [2] found adaptive prediction to be substantially less effective than adaptive quantization (explained below) with respect to audio quality. Therefore adaptive prediction will not be discussed in this project.

## 2.3 Quantization, companding and entropy coding

A uniform quantizer with a fixed step size, as described in [5][ch. 4.3] is simple with respect to both implementation and theory. However, most signals are not uniformly distributed. More complex quantizers may make use of known statistical properties of the signal, typically by analyzing its probability density function (PDF). A common approach for PCM-coded signals is to remap amplitude values according to a companding curve so that the resulting PDF is closer to a uniform distribution and therefore well suited for uniform quantization with a fixed step size [5][ch. 4.5]. This is called companding or logarithmic quantization. A related approach is to design non-uniform quantizers tailored for specific PDFs, for instance a Gaussian distribution. These are called PDF-optimized quantizers.

Any transmission system with a quantizer that produces an output with a non-uniform distribution will benefit from entropy coding [5, ch. 4]. Coding techniques such as arithmetic coding can be used after compression if this is the case with the error signal [9, ch. 5]. This would produce a variable-length codebook, which could compromise error robustness when transmitting the signal, since it is harder to decode an array of samples without a priori knowledge of its length [1]. By designing the quantizer such that its output is uniformly distributed, entropy coding is avoided since all codewords will be equally probable. This is typically achieved with a PDF-optimized quantizer.

Another step in producing a uniform output from the encoder is remap the amplitudes of the input signal using a companding curve [6, ch. 3.3.2]. This mapping is done prior to quantization, so for DPCM it would be applied to $d(n)$. A typical companding curve called $\mu$-law is shown in figure 2.5. $\mu$-law is a well known standard used in commercial digital communications systems. It is defined by function $F(x)$ in equation 2.9. Typically, $\mu = 255$ for $B_C = 8$. After decoding at the receiver side, companding must be inversed

[1] This problem can be overcome with a protocol that enforces Automatic Repeat Query [1]. For simplicity, the implementation of this project will not enforce this.

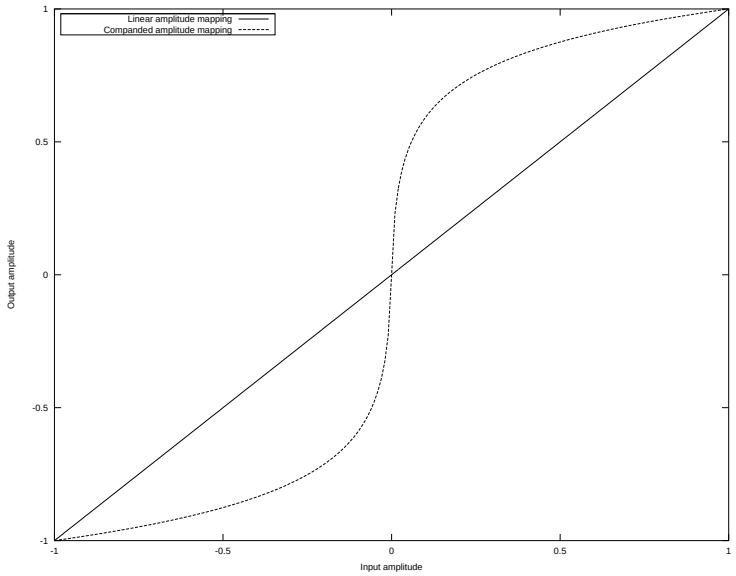


Figure 2.5: Companding curve example.

by applying $F^{-1}(x)$.

$$F(x) = \text{sgn}(x)\frac{\ln(1+\mu|x|)}{\ln(1+\mu)}, \ -1 \leq x \geq 1 \tag{2.9}$$

This project will not be concerned with companding as it is not common to compand $d(n)$ in DPCM [10]. Companding is mentioned here to emphasize that a companded signal prior to quantization may have an impact on audio quality and is therefore a source of improvement that is not computationally demanding.

## 2.4 ADPCM

DPCM can be adaptive in two ways: prediction adaptive, as mentioned in the previous section, and quantizer step size adaptive, which is what will be referred to as ADPCM throughout this report. As mentioned earlier, the quantizer is the only step in the signal chain where the signal is irreversibly degraded, not counting transmission losses. Therefore it makes sense to put an effort into designing it well. The approach discussed in this section is a uniform quantizer with a sample-by-sample-adaptive step size. The reasons for this is as follows:

- Step-adaptive quantizers tend to be less signal-specific than most other quantizer types, which is an important property since we will be investigating several types of audio sources and applications in this paper [5, ch. 4].
- They are known to produce a stable SNR [5][p. 195].
- They are easy to implement and can have a low memory overhead.
- Compared to a uniform quantizer with a static step size, a quantizer with an adaptive step size is one step closer to a PDF-optimized one [5, ch. 4].

Adaption schemes can be divided into two categories: forward adaption and backward adaption. Forward adaption works on sample blocks of size $M$ which is buffered up and analyzed to produce adaption parameters every $M$th sample. Adaption parameters is typically prediction coefficients or step sizes. This introduces a delay of $M$ samples and requires adaption parameters to be sent as side information and therefore adds to the transmission bit rate.

With backward adaption, the encoder and decoder simultaneously calculates the adaption parameters based on the quantized and transmitted samples, and therefore makes it possible to update prediction coefficients on a sample-by-sample basis. This also eliminates the additional side information bits and delays at the cost of suboptimality because we are analyzing the coded signal $e(n)$ and the decoded signal $\tilde{x}(n)$ instead of the input signal $x(n)$. Therefore AQB generally performs worse than AQF for coarse quantization[5][p. 303].

Forward-adaptive and backward-adaptive quantization is abbreviated AQF and AQB, respectively. AQF can be very straightforward: simply find the dynamic range of a block of $M$ samples and adjust the step size so that the quantizer operates within the same dynamic range as the signal. For example, $d(n)$ in figure 2.4 can be quantized with $B_C$ bits using a significantly smaller step size $\Delta$ than $x(n)$ because it has a significantly shorter dynamic range.

A lot of algorithms for AQB exists. The following scheme, called adaptive quantization with a one-word memory, is one of the least demanding AQB algorithms with respect to both computational complexity and memory usage [11]. The adaptive step size $\Delta(n)$ is calculated as

$$\Delta(n-1)M(|C(n)|), \tag{2.10}$$

where $C(n)$ is the current codeword and $M$ is a table of size $2^{B_C-1}$ that holds *step size multipliers* that has a distinct value for all $|C(n)|$. The basic

idea for $M$ to have values less then one in its lower half and values larger than one in the upper half, so that an increase in $C(n)$ compared to $C(n-1)$ yields an increase on $\Delta(n)$ and vice versa. An example for $B_C = 3$ using a midtread uniform quantizer is shown in figure 2.6, with example values for $M$.

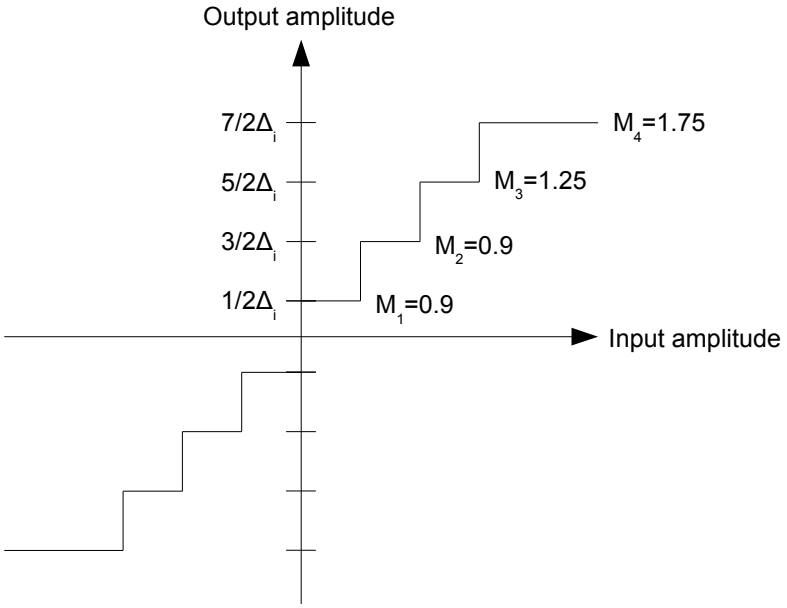


Figure 2.6: Example of one-word memory AQB.

Although the encoder must do both adaption, encoding and decoding for each sample, adaption is only one table lookup and one multiplication. The real work, which is of course computed offline, is to find the $M$-table that is best suited for the signal in question. Training a system to find the best fitting $M$-table will not be pursued in this project as there exists standard tables for typical values of $B_C$ that have taken typical signal statistics into account.

# Chapter 3

# Number representation

This chapter will discuss digital number representations and arithmetic operations. The focus will be on fixed-point arithmetic, since there is no on-board floating point unit (FPU), only an integer arithmetic logic unit (ALU) on the SoC.

## 3.1 A comparison of floating point and fixed-point representation

When designing a digital system, the inclusion of an FPU is a trade-off between cost and value range, both of which is higher with native floating point arithmetic. A comparison of typical 32-bit floating point and fixed-point integer properties is shown in table 3.1 [12, ch. 1.5]. Q15.16 is a signed 32-bit fixed-point number representation scaled with 15 bits for the integer part and 16 bits for the fractional part, which will be discussed in depth in section 3.3. The range is the span of numbers that a type can represent. The precision is the difference between successive values relative to scaling and is equal to the value of the least significant bit.

Table 3.1: Comparsion of 32-bit floating point and two types of fixed-point number representation.

| | IEEE 754 Float | Signed integer | Q15.16 |
|---|---|---|---|
| Value range | $\sim \pm 3.4 \times 10^{38}$ | $-2^{31}$, $2^{31}-1$ | −32678, 32767 |
| Precision (maximum) | $2^{-23}$ | 1 | $\sim 1.5 \times 10^{-5}$ |
| Overflow (definiton) | $\infty$ | $2^{31}-1$, $-2^{31}$ | −32678, 32767 |
| Underflow (definition) | $-\infty$ | 0 | 0 |

Emulating true floating point operation on a fixed-point ALU is known

to be costly in terms of computations [12, p. 29][1]. The precision, however, is initially the same for both floating point and fixed-point types if they are represented by the same number of bits. It is the scaling that makes the difference in the representative precisions in table 3.1. Therefore, it is only the scale and range that might be a problem when using fixed-point types. So for any algorithm, do an analysis to find the right scale for that particular algorithm. If no scale exists that satisfies the required precision and range, analyze the algorithm in question to locate the parts that needs different ranges and scales. Then one can scale the fixed-point number differently for each part, effectively achieving floating point arithmetic (although not true emulation). Even though such an analysis can be a tedious task, it will produce an implementation that is less computationally demanding than true floating point emulation.

This chapter will investigate what range and precision is needed for the algorithms described in chapter 2 and present a number type that is sufficient for these.

## 3.2 Fixed-point arithmetic

Having shed some light on the main issues in comparing floating- and fixed-point representations, the rest of this chapter will focus entirely on fixed-point arithmetic.

## 3.3 Qi.f. number format

The words 'integer' and 'fixed-point' have been used interchangeably so far. This is because from a processor view, they are the same. The processor has no knowledge of any point between the integer and fractional part of a number and treats all numbers as integers. A common way of denoting a fixed-point number is as Q$i.f$, where $i$ is the number of bits assigned for the integer part and $f$ for the fractional part, as seen in equation 3.1.

$$\mathrm{Q}i.f = \sum_{j=0}^{i} 2^j + \sum_{k=1}^{f} 2^{-k} \tag{3.1}$$

The sign bit is not counted, so $B_A = 1 + i + f$ is the total of available bits for a number. A full 32-bit integer, for example, can be denoted as Q31.0.

[1] True emulation involves storing the numbers in a floating point format and run subroutines for all operations on these numbers so that floating point operation is transparent to the programmer.

[13] describes how to carry out basic arithmetic on any fixed-point number Q$i.f$. For all cases where $f > 0$, special care needs to be taken to get correct results with fixed-point operations. In addition, operations where $f$ and $i$ are different for the operands requires even more care, since we need to align the points before carrying out the operation. Therefore it is desirable to use only one type of Q$i.f$ if possible.

Addition of two Q$i.f$ numbers needs no extra treatment, since overflow in the fractional part will flow over to the integer part and underflow in the integer part will flow over to the fractional part. When multiplying two numbers represented by $B_A$ bits, the intermediate result is a $2B_A$ size number, i.e. Q$2i.2f$. Also, when multiplying the $f$ fractional bits we need to post-scale by right-shifting $f$ bits, since the processor sees $\sum_{k=1}^{f} 2^{-f}$ as $\sum_{k=1}^{f} 2^{f}$. Algorithms for multiplication and division of Q$i.f$-numbers can be found in appendix D.

When choosing a suitable Q$i.f$ format, we must take into account all data variables of the system. Let's look at the ones needed for DPCM:

- The input samples $x(n)$ occupy no more than 12 bits due to the ADC.
- For perfect prediction, the predictions coefficents $\alpha_i$ can in theory take on the value from predicting the difference between two input samples: $x(n) - \alpha_i x(n-i) = 0 \ \forall i$. The extremes for this is when $|x(n)| = 1$ and $|x(n-i)| = 2^{-12}$ and vice versa. This means that $|\alpha_i| = 2^{12}$ at its largest and $|\alpha_i| = 2^{-12}$ at its smallest. Therefore $\alpha_i$ theoretically needs $12 + 12 = 24$ bits for full representation. This is, however, an extreme overrepresentation, since most values of $\alpha_i$ tends to be in the range of $-2.5 \leq \alpha_i \leq 2.5$ [10].
- For AQB and AQF, the step size $\Delta(n)$ should be limited to the same number of bits as $x(n)$, since it makes no sense to have a step size smaller than the LSB of $x(n)$.

These requirements are covered by the common format Q15.16, where we define $x(n) \in [-1, 1-2^{-12}]$. This is illustrated in figure 3.1. The figure shows that we have a total of seven spare bits to prevent under- and overflows when processing, three from the MSBs and 4 from the LSBs.

## 3.4 Saturation arithmetic

Saturation arithmetic is illustrated in figure 3.2. With this arithmetic, any number that is produced by an arithmetic operation is limited within a fixed

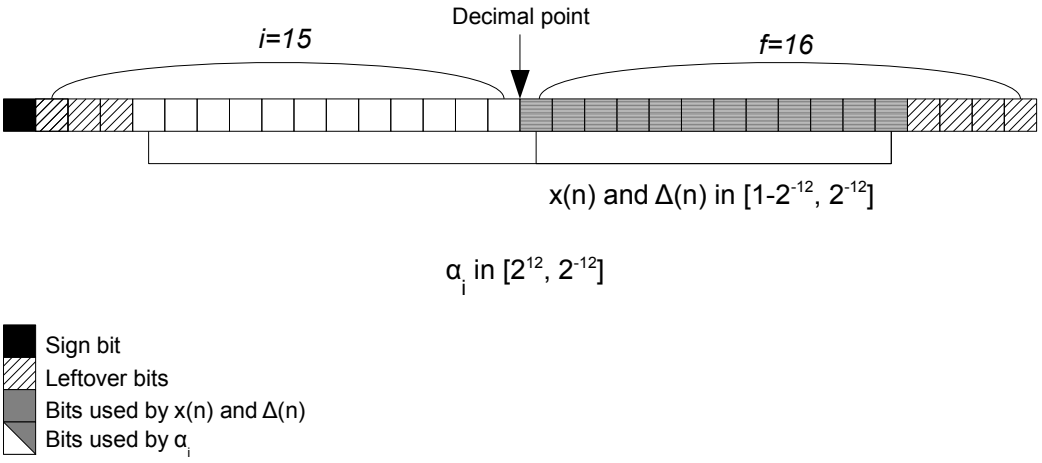


Figure 3.1: Q15.16 number format structure.

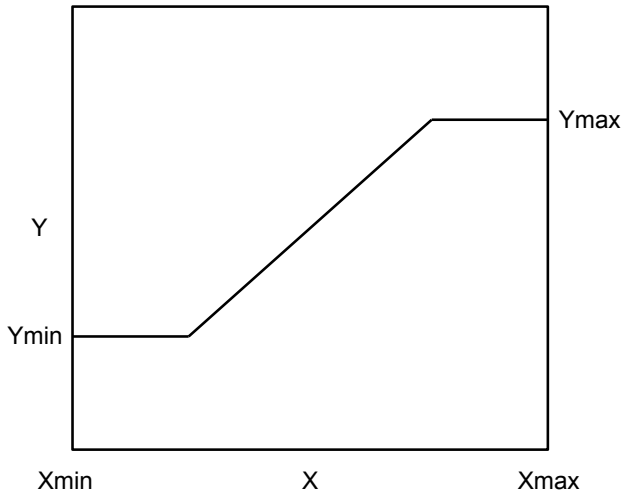


Figure 3.2: Saturation arithmetic with input and output numbers X and Y, respectively.

range of a maximum and minimum number. For signal processing, saturated integer arithmetic is commonly used. This is because other methods for handling (or not handling) overflow, the most common being binary modulo, may result in very large errors. Too see why, let $X$ and $Y$ be two numbers represented by $B_A$ bits. Denote addition with binary modulo and saturation arithmetic as $+_m$ and $+_s$, respectively. Then

$$X +_m Y = (X + Y) \bmod (2^{B_A - 1} - 1) \tag{3.2}$$

$$X +_s Y = \max((\min(X + Y, 2^{B_A - 1} - 1), -2^{B_A - 1}) \tag{3.3}$$

For instance, if $X = 32767$ and $Y = 1$, then $X +_m Y = -32768$, which produces the error $E = |(X +_m Y) - (X + Y)| = 65536$. On the other hand, $X +_s Y = 32767$, which produces $E = 1$. In this sense, saturation arithmetic is the digital counterpart to *clipping* or *limiting* [14] in the analog domain. Software saturation requires extra computations and memory resources since we need to calculate the resulting number before deciding whether it overflows or not. However, most current architectures, including our SoC, has the option of using registers with implicit saturation, which will handle saturation automatically without using any extra resources [3].

# Chapter 4

# Error concealment

Losing audio samples, either by packet loss or by late reception at the receiver is something that is likely to happen every now and then. Dealing with errors in on a real-time media transmission system can be divided into to three categories:

- Error resilience - making sensitive data more robust against errors.
- Error protection - detecting and correcting errors.
- Error concealment - detecting and concealing errors.

These are somewhat overlapping. Error resilience and protection is part of channel coding and will not be discussed in this project. This chapter will discuss error concealment.

Packet loss means losing a full packet of unrecoverable data that holds $N$ samples. Error concealment means finding the subjectively best sounding way of patching up the "hole" in the audio stream due to such a loss. Error concealment is carried out at the receiver side, although the sender may send extra information that would make concealment at the receiver perform better. This project will only be concerned with simple error concealment techniques that are carried out at the receiver. They will also be source coder independent, meaning they can be applied to any decoded stream. Common methods include [15]:

1. Padding with zeros and low-pass filtering to avoid abrupt changes in audio.
2. Add white or pink noise with same power as last packet.
3. Repeating the last packet or pattern matching using segments of earlier packets.

These techniques are only applicable to systems with small amounts of samples per packet and infrequent losses [15]. The methods are illustrated by exampled in figure 4.1.

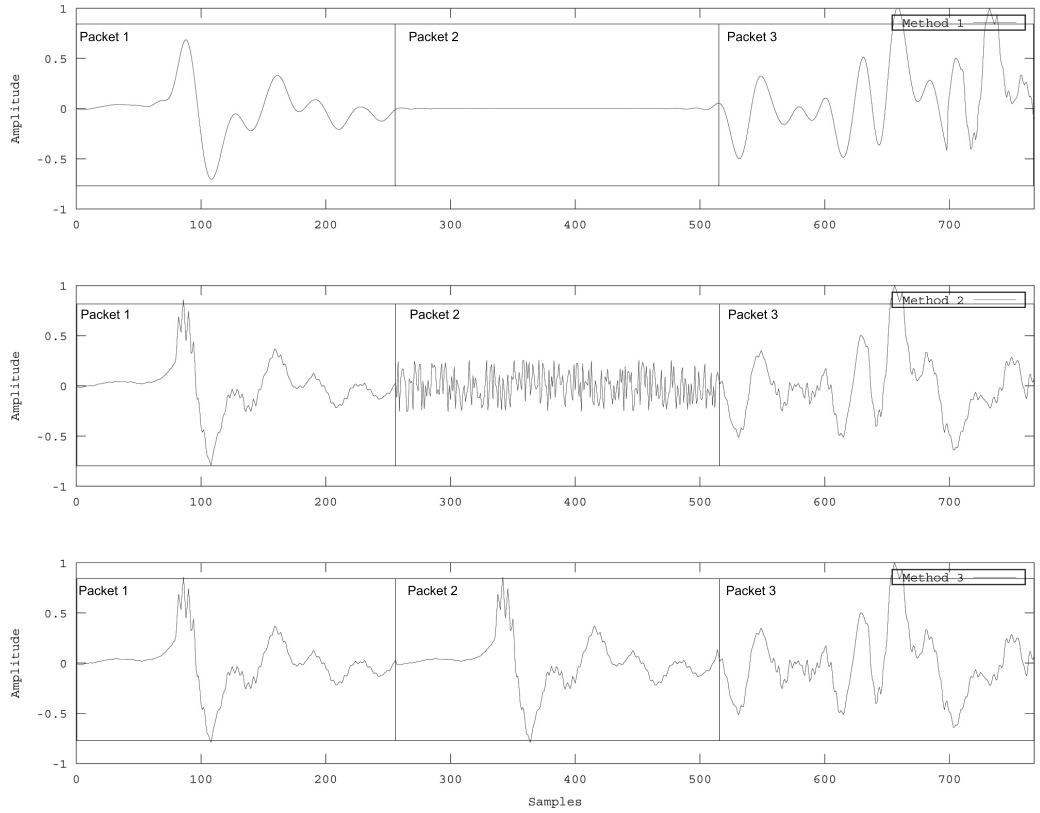


Figure 4.1: 3 methods for error concealment.

Although the SoC has an on-chip random number generator, number generation is considered more time consuming than producing zeros, so method 2 is considered the slower algorithm of the three. Method 3 is the least computationally demanding one, but it requires more memory overhead sicne the previous packet must be available. Which is the better sounding of the three methods is very dependent on the statistics of the signal in question and the packet sample size $N$. For instance, the loss of a large packet may sound better with a moment of silence as with method 1. On the other hand, for a very uncorrelated or noisy signal due to hard quantization, method 2 or 3 probably sounds better.

# Chapter 5

# Analog-to-digital conversion

This chapter will present and evaluate the on-chip analog-to-digital converter (ADC) and conclude with an assessment how it performs when used for digitizing audio signals. The ADC uses delta-sigma modulation for conversion [3]. It is considered a low-cost general-purpose converter that is typically used for digitizing voltages from mechanic input such as a rotary knob [7]. The audio files analyzed in this chapter is referenced in appendix F.

Table 5.1 (repeated from chapter 1) lists possible conversion modes in terms of bit resolution $B_A$ and sampling frequency $f_s$ in samples per second [SPS]. Mode 4 will not be evaluated as it was not available at the time of

Table 5.1: ADC modes.

| **Mode** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|
| Bit depth $B_A$ [bits] | 8 | 9 | 10 | 12 |
| Sample rate $f_s$ [kHz] | 50 | 28 | 14.7 | 3.8 |
| Bit rate $I = B_A f_s$ [KBPS] | 400 | 252 | 147 | 45.6 |
| Bandwidth [kHz] | 25 | 14 | 7.35 | 1.9 |

this project. The approximate conversion formula is shown in equation 5.1, where $V_{in}$ and $V_{max}$ is the analog signal voltage and its maximum value, respectively, so that $0 \leq V_{in} \leq V_{max}$.

$$x(n) = \frac{V_{in}}{V_{max}} 2^{B_A} \tag{5.1}$$

Figure 5.1 illustrates how the ADC can convert signals from up to 8 inputs. A conversion can not be done faster than $\frac{1}{f_s}$ seconds, though. This means that if $L$ is the number of inputs used, the effective sampling frequency is $f_s(L) = \frac{f_s}{L}$. For example, sampling a stereo signal ($L = 2$) in mode 3 would produce two signals $x_l(n)$ and $x_r(n)$ with bit resolutions $B_A = 10$ bits and

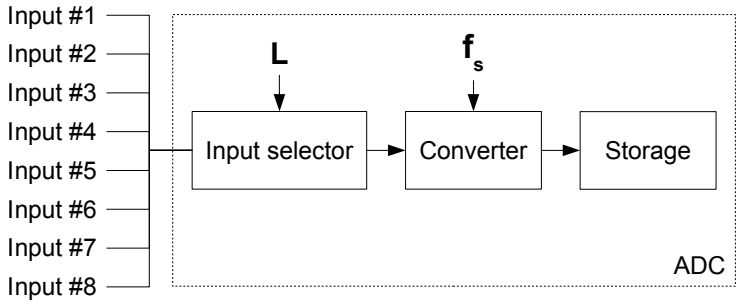


Figure 5.1: Input selection and conversion diagram for the ADC.

sample rate $f_s = 7.35$kHz each. This chapter will only be concerned with ADC operation using $L = 1$. Applications with $L > 1$ will be discussed in chapter 9.

The values of $f_s$ in table 5.1 are the *maximum* sample rates for their respective modes. Any mode may sample at a rate equal to or lower than $f_s$, but not higher.

## 5.1 Evaluation methods

An ideal AD converter will do perfectly uniform quantization[1] of the input signal $x_a(t)$ and will therefore introduce a quantization noise with a flat power spectral density (PSD) only [5, ch. 4]. Perceptually, broadband noise with close-to-flat PSDs tends to sound more comforting to the human ear than narrowband noise [6, ch. 5] and is therefore desirable. Sadly, no such ADC exists. All ADCs have some inherent distortion due to non-linearities in their conversion function in addition to the quantization noise, which is why equation 5.1 from the previous section is an approximation. Such distortions produce harmonics of the input signal. This section will present the theory and practicalities behind the test setup in figure 5.2 as described in [16] and [17].

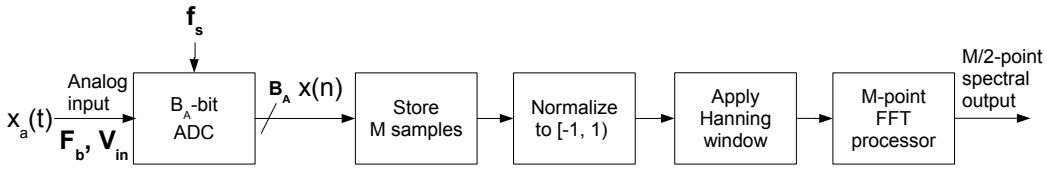


Figure 5.2: ADC performance test setup [17].

[1] Provided that $B_A \geq 3$ [8, ch. 9].

Signals on the form $x_a(t) = \frac{V_{max}}{2}(\sin(2\pi ft) + 1)$ as well as silence will be sampled an analyzed to make an assessment of the ADC performance. $x_a(t) \in [0, V_{max}]$ volts and $t \in [0, t_x]$, where $t_x$ seconds is the duration of $x_a(t)$. For the test we will look at power spectral density (PSD) estimates[2] denoted by $P_s(k)$ for any signal $s(n)$, where $k$ indexes frequency bins. We will also need the DFT-transform $S(k)$ applied to any signal $s(n)$, as defined in equation 5.2.

$$S(k) = \sum_{n=0}^{M-1} s(n)e^{-i2\pi\frac{k}{M}n} \tag{5.2}$$

$M$ is the length of the signal $s(n)$ in samples and also the size of the DFT transform. Entries of $S(k)$ will simply be referred to as *bins* later in the text. Further, we will use $E_S$ as a measure of the energy as defined in equation 5.3.

$$E_S = \sum_{k=0}^{M-1} |S(k)|^2 \tag{5.3}$$

The *signal-to-noise-and-distortion-ratio* (SINAD), as defined in equation 5.4, is a common measure of ADC performance.

$$\text{SINAD} = \frac{E_X}{E_Q} \tag{5.4}$$

Here, $E_X$ is the input signal energy and $E_Q$ is the quantization noise, including harmonic distortions of $f$ and excluding the DC component [17]. The SINAD measure provides a good indication of the the overall dynamic performance of the ADC since all noise and distortion is taken into account. Also, it can easily be translated to a measure called the *effective number of bits* (ENOB) as defined in equation 5.5 [16].

$$\text{ENOB} = \frac{1}{2}\log_2(\text{SINAD}) - \frac{1}{2}\log_2(1.5)[\text{bits}] \tag{5.5}$$

This measure tells us that the measured signal-to-noise ratio (SNR) for an ADC corresponds to that of an ideal ADC with $B_A = \text{ENOB}$ bits. Put another way, the ENOB tells us how many bits are effectively used to represent $x_a(t)$, since some fraction of the $B_A$ bits is used to represent the distortion introduced by the conversion and not $x_a(t)$.

Noise and distortion is measured by converting $x_a(t)$, then applying a Hanning window function[3] and a DFT $X(k)$ of size $M = t_x f_s$ points. Then

[2]Estimated using Octave function **spectral_xdf** with a triangle window.

[3]Too reduce the size of the side lobes of $X(k)$ [10].

the noise energy $E_Q$ is measured from $X(k)$, with $k \in [0, \frac{M}{2}]$ except for the bins that holds $f$ and the DC component. Similarly, one can measure the energy $E_X$ of $x_a(t)$ by calculating the energy in the bins that holds the $f$ component. This is illustrated by example in figure 5.3. The values of $f$ are

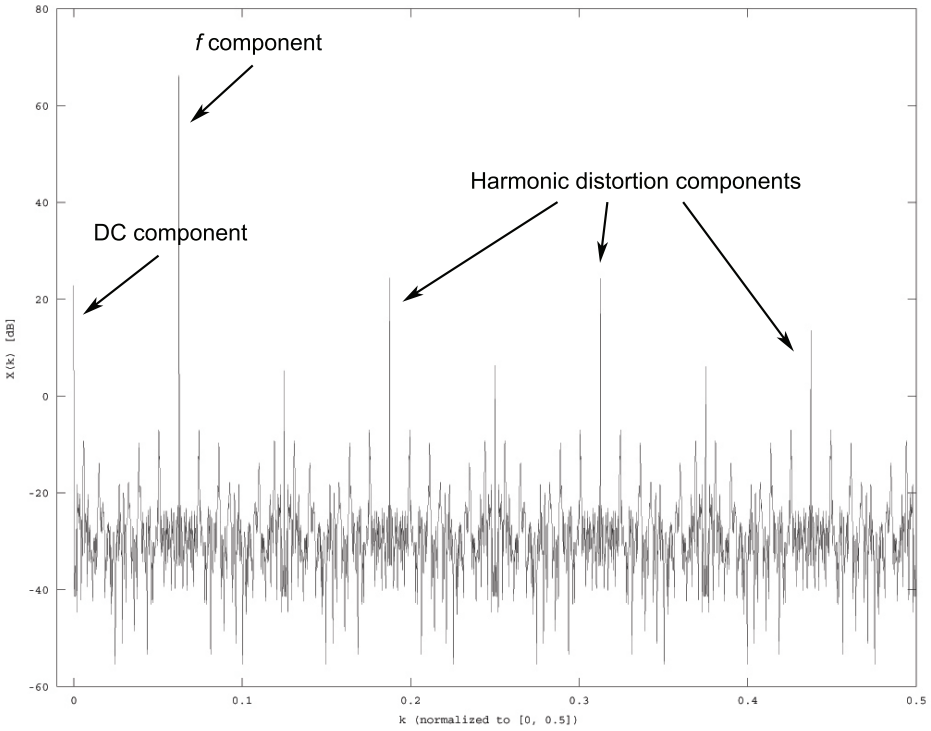


Figure 5.3: Example of $M$-point DFT plot showing DC, $f$ and harmonic distortion components.

chosen so that they are approximately logarithmically distributed in $[0, \frac{f_s}{2}]$ according to $f = f_s \frac{J}{M}$, where $J$ is some number that relatively prime to $M$[4].

## 5.2 Performance tests

Before presenting the SINAD results we will look at a PSD estimate for silence when sampled by the ADC to investigate how close to uniform the ADC is. Figure 5.4 shows the PSD estimate $P_X(k)$ using $M = 60000$ samples of $x_a(t) = \frac{V_{max}}{2}$ sampled with the ADC. $t_x = 5$ seconds, $V_{max} = 4.0$V and $f_s =$ 14kHz for all modes for easier calculation. These values are used throughout

[4]Being relatively primed means having no common factors.

all tests in this section unless stated otherwise. As stated earlier, modes 1, 2 and 3 uses $B_A = 8$, 9 and 10 bits, respectively.

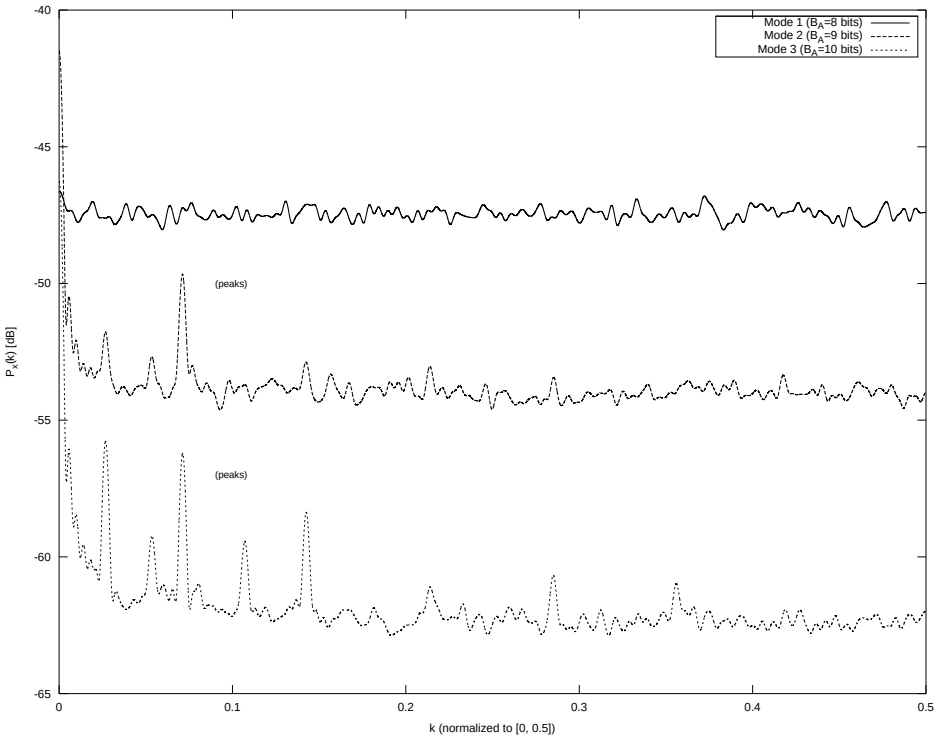


Figure 5.4: ADC noise and distortion PSD estimation on sampled silence.

The graph shows that $P_x(k)$ is relatively flat for $f \in [0, \frac{f_s}{2}]$, with the exception of some peaks in the lower end of the spectrum. The noise floors (the value at which the plots are nearly flat) are at about $-62$ dB, $-55$ dB and $-48$ dB for modes 3, 2 and 1, respectively. A uniform quantizer with $B_A \geq 3$ is known to have the same characteristic, and, as stated earlier, the ideal is a uniform quantizer [5, ch. 4]. The graph also shows that the noise power $P_x(k)$ of ADC modes 1, 2 and 3 differ by approximately 6 dB. Since an ideal uniform quantizer has an

$$\text{SNR} = 6.02B_A + 1.5\text{dB}, \tag{5.6}$$

this adds to the claim that the ADC performs close to that of an ideal uniform quantizer and that the noise floors are mainly made up of quantization noise [5, ch. 4].

Figure 5.5 shows the results of calculating SINAD and ENOB for $x(n)$ versus eight values of $f$. As when carrying out the PSD estimates above,

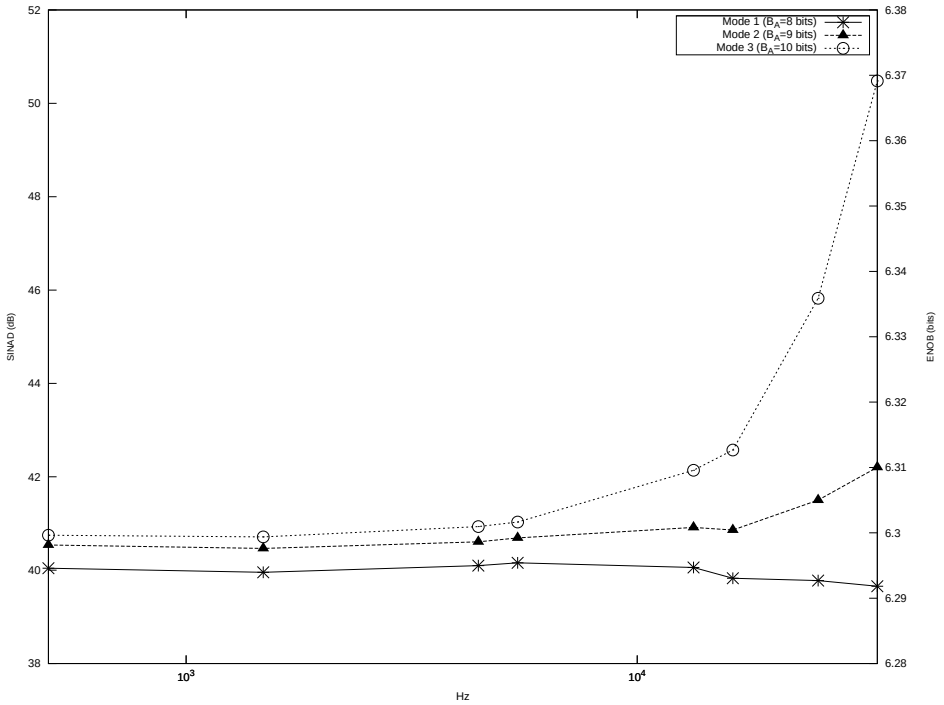


Figure 5.5: SINAD and ENOB for ADC modes 1, 2 and 3.

$t_x = 5$ seconds, $V_{max} = 4.0$V and $f_s = 14$kHz for all modes. $f \in \{99, 297, 891, 1089, 2673, 3267, 5049, 6831\}$.

The ENOB results are somewhat surprising. One would expect an increase of approximately 1 bit in ENOB for an increase of 1 bit in $B_A$ according to figure 5.4 and equation 5.6. As figure 5.5 shows, this is not the case. The differences between the ENOB calculations for all modes are much less than 1. The graph suggests that modes 2 and 3 only improves by a marginal amount compared to mode 1 in terms of the ENOB. This is contrary to the PSD estimates presented above.

An explanation for these ENOB calculations could be that the harmonic distortions have magnitudes large enough to outweigh the differences in the quantization noise of the modes. As mentioned earlier, figure 5.4 shows a number of peaks in the PSD, especially in the lower half of the spectrum, for mode 2 and 3. Since the peaks are similarly distributed for modes 2 and 3, it is feasible to assume that these peaks are present in all modes, and that they are merely masked by the noise floors for mode 1 and partly for mode 2. Now, figure 5.4 only shows the distortion when sampling silence, leaving out all potential harmonic distortions. Figure 5.6 shows $P_x(k)$ when $x_a(t) = \frac{V_{max}}{2}(\sin(2\pi ft) + 1)$ with $f = 99$ Hz and $M = 60000$ samples. As

expected, this graph shows even more peaks, especially some very prominent ones in the midband bins that are almost identical in magnitude across the three modes. These are about 24 dB above the noise floor of mode 3, meaning that they have magnitudes 16 times of that of that noise floor. It is feasable to assume that these, in addition to the other distortions, are the cause of the poor ENOB calculations.

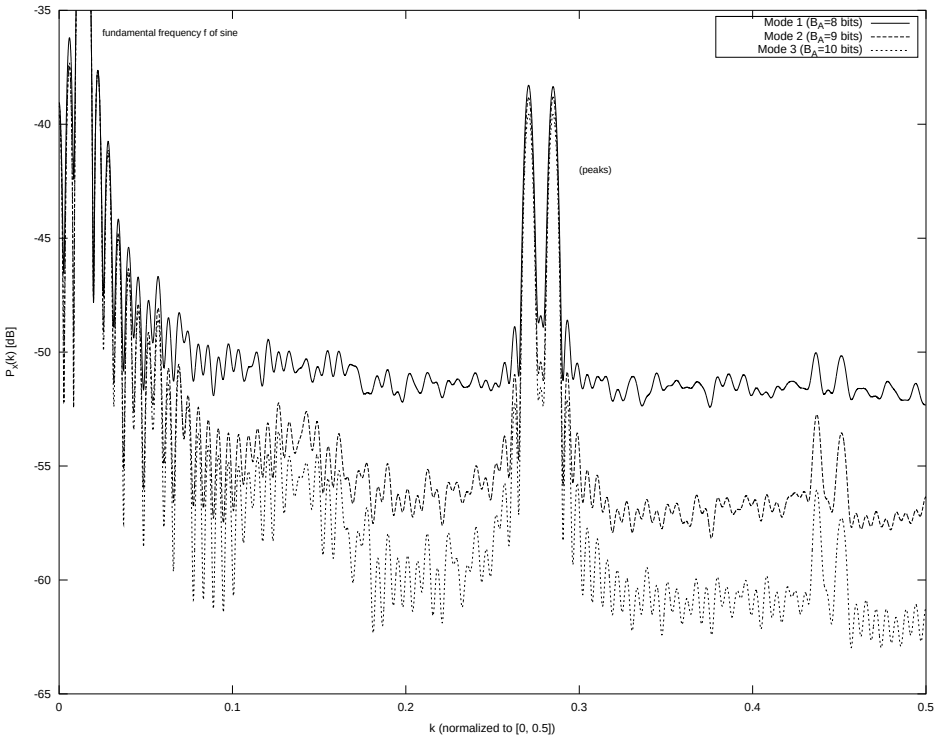


Figure 5.6: ADC noise and distortion PSD estimation on sampled sine with frequency $f = 99$ Hz.

Notice also that the ENOB calculations in figure 5.5 improves for higher values of $f$. This is because higher values of $f$ will produce less harmonics across the spectrum, effectively increasing the ENOB at the higher end of the spectrum [10].

Summing up the ENOB calculations, the modes are almost identical in terms of ENOB and modes 2 and 3 perform poorly compared to their $B_A$, although always marginally better than mode 1. We will not go into the exact impact of the distortions on the ENOB readings, nor the source of the various peaks[5], but merely conclude that the distortions are simply outweighing the

[5]In addition to harmonic distortions, non-harmonic distortions can arise from artifacts in the circuitry of the ADC [10]

quantization noise.

Informal listening tests was carried out by the author. They confirmed a broadband noise spectrum. A distinguishable difference between the noise from modes 1, 2 and 3 was heard, and the harmonic and non-harmonic distortions were almost non-audible. The perceived noise was reduced with an increasing $B_A$. In other words, the listening tests concur with the PSD estimates and the ADC performs almost like an ideal quantizer perceptually. This makes the important point that for our application, we will not be listening to pure sines, but to audio with complex spectra, and the ENOB calculations are not by far as important as the PSD estimates for this project.

## 5.3 Noise reduction

[2] presented, implemented and discussed several different approaches to reducing the noise introduced by the ADC. Now that we have characterized the noise produced by the ADC, it becomes clear that some digital noise reduction algorithm could improve on the signal post-conversion.

[2] suggests an low cost (in terms of computations) noise reduction scheme called a *noise gate* [14, ch. 9]. An example of this is shown in figure 5.7.

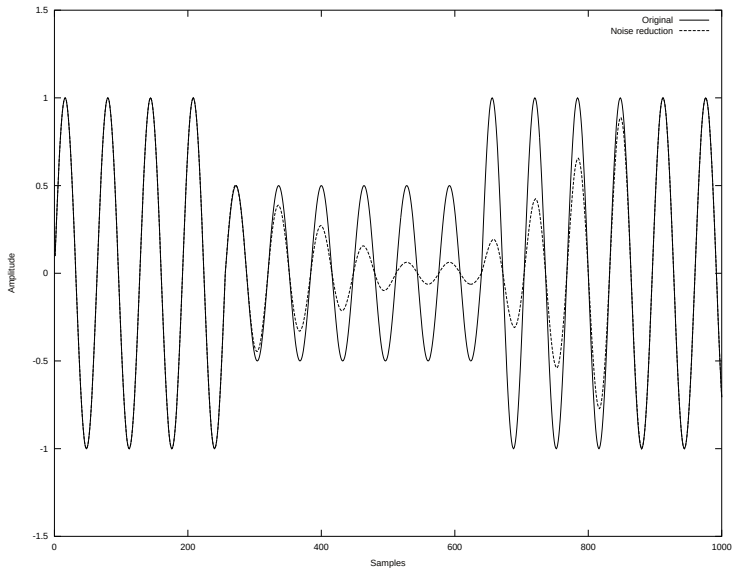


Figure 5.7: Example noise gate on a sine that starts in [-1, 1], drops to [-0.5, 0.5] and goes back up to [-1, 1]. $G_t = -6$ dB, $T_{gate} = 500$ samples and $G_a = 24$ dB.

The idea is simply to detect signals that have amplitudes under a certain threshold $G_t$ for a time period of $T_{gate}$ and attenuate it by $G_a$ dB, effectively removing or reducing the noise. Since we are primarily trying to remove the quantization noise from the ADC, $T_{gate}$ should be set to a value near the peak value of the quantization noise. This can be found by analyzing silent periods of the signal.

This scheme will of course only remove noise when no other significant signal is present. For example, if someone starts talking, no noise will be removed. This is justified by the fact that noise below a certain threshold may be masked by the clean signal, especially if the noise is white. Noise is, in other words, most audible during silence [18, ch. 31].

## 5.4 Conclusion

From table 1.1 it was already clear that CD-quality audio was not attainable, and expectations for the ADC to perform well with audio signals were low. However, the noise present turned out to relatively white, and the perceived quality when testing with pure speech audio was surprisingly good for all the evaluated modes. When dealing with speech, bandwidths larger than $f_b = 8$kHz are not interesting [19], so modes 2 and 3 are the most applicable for such an application. And as mode 2 produces almost double the bit rate of mode 3, mode 3 is the best choice for a speech source. A discussion on different applications for different modes is found in chapter 9.

# Chapter 6

# Digital-to-analog conversion by pulse-width modulation

The SoC does not have an on-board digital-to-analog converter (DAC) to output analog audio. Although one may use an external DAC through a digital bus like SPI, it is desirable to only use on-chip features. This chapter will present pulse-width modulation (PWM) as a technique for implementing an on-chip DAC [3]. Focus will be on the technique and we will not go deep into the mathematical theory behind PWM.

PWM is a means of outputting any digital signal on a single general-purpose I/O pin (GPIO) so that it functions as a 1-bit DAC. This is shown by example in figure 6.1, and the signal path is shown in 6.2. The idea is to modulate the pin state with the audio signal $\tilde{x}(n)$ using a counter signal $p(n)$ with period $\frac{1}{f_t}$ and a carrier signal $c(n)$ with period $\frac{1}{f_c}$. If $f_s$ is the sampling frequency of $\tilde{x}(n)$, we should have $f_s \ll f_c \ll f_t$ for reasons that will become clear below. PWM is carried out as follows.

- Let $\tilde{x}(n)$ be the audio signal used for modulation with sampling frequency $f_s$ and bit depth $B_A$.
- Generate carrier signal $c(n)$. This is a sawtooth wave with period $\frac{1}{f_c}$ that counts to $2^{B_A}$ once a period. The counter increments by 1 every tick, which is every $\frac{1}{f_t}$ seconds.
- Generate a pulse-width modulated signal $p(n) = \begin{cases} 1, & \tilde{x}(n) > c(n) \\ 0, & \text{otherwise} \end{cases}$ which is computed every $\frac{1}{f_t}$ seconds.
- Output $p(n)$ on the designated GPIO pin and apply an analog low-pass filter with cut-off frequency at $\frac{f_s}{2}$ to remove high-bandwidth artifacts

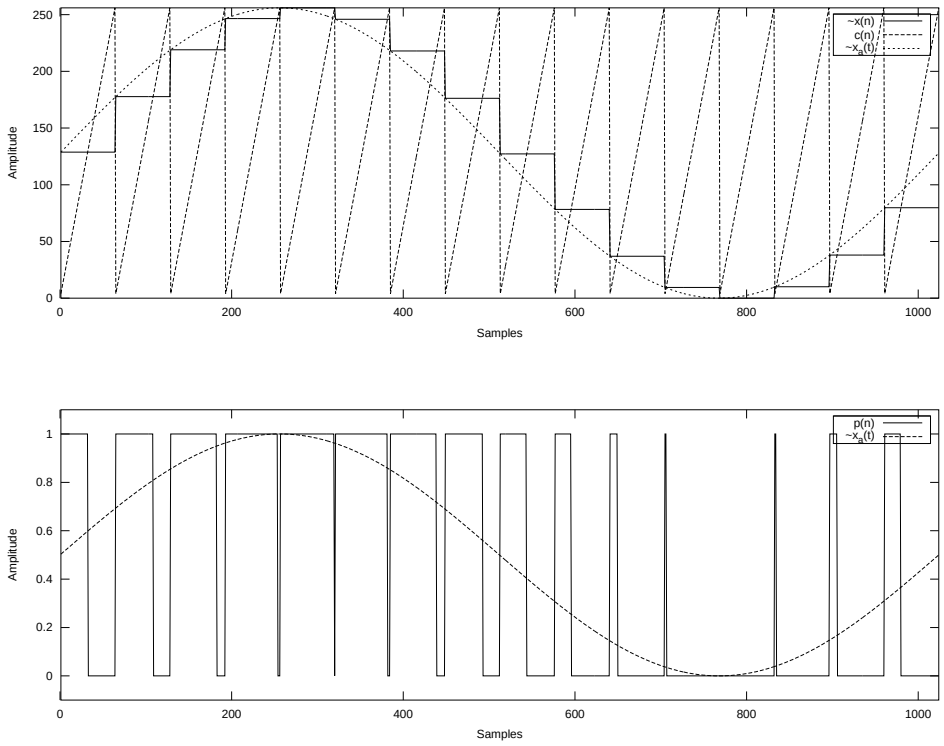


Figure 6.1: PWM digital simulation example with $B_A = 8$ bits, $f_s = 10$ kHz, $f_c = 120$ kHz and $f_t = 30.72$ MHz.

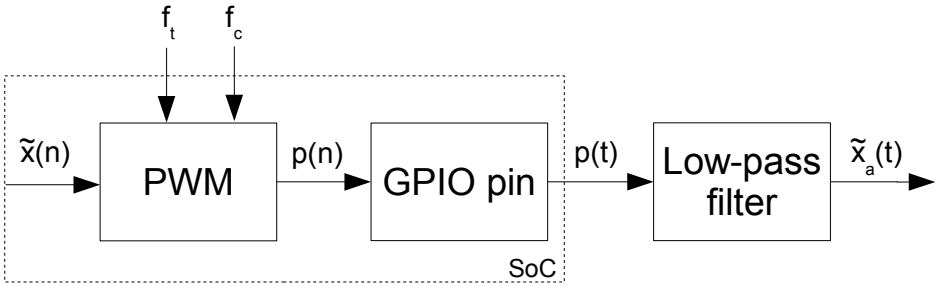


Figure 6.2: PWM signal path.

from the modulation. The resulting signal $\tilde{x}_a(t)$ is then the analog counterpart of $\tilde{x}(n)$.

The method is intuitively clear; the larger the value of $\tilde{x}(n)$, the longer the duty cycle of $p(n)$, which produces more energy on the analog output $p(t)$. Low-pass filtering $p(t)$ will then produce $\tilde{x}_a(t)$. This is illustrated by example in figure 6.1.

For carrying out PWM with the desired $B_A$, $f_c$ and $f_t$, the following constraint applies: since $c(n)$ must count to $2^{B_A}$ every $\frac{1}{f_t}$ seconds, $2^{B_A} f_c \leq f_t$. For our CPU, $f_C = 16$ MHz. Assuming that it is possible to generate one $p(n)$ sample using only one machine cycle (usually possible by using a hardware latch to bypass software processing), we can have $f_t \leq f_C$. Further, [20, p. 160] states that for high quality PWM output, one should have $f_s \leq 12 f_c$. So for high quality PWM using our SoC we have the constraints $2^{B_A} f_c \leq f_C$ and $f_s \leq 12 f_c$, which can be shortened to $f_s \leq \frac{f_C}{12 \cdot 2^{B_A}} = f_{max}$. This is checked for each ADC mode and summed up in table 6.1.

Table 6.1: PWM possibilites for the four ADC modes. For high quality PWM, we must have $f_s \leq f_{max}$.

| **Mode** | $B_A$ [bits] | $f_s$ [kHz] | $f_{max}$ [Hz] |
|---|---|---|---|
| 1 | 8 | 50 | 5209 |
| 2 | 9 | 28 | 2605 |
| 3 | 10 | 14.7 | 1302 |
| 4 | 12 | 3.8 | 353 |

As $f_s > f_{max}$ for all modes, the conclusion is that high quality PWM output is not possible in any of the SoC modes. Therefore, noisier PWM output which only obey the constraints $2^{B_A} f_c \leq f_C$ and $f_c < 20$ kHz, the threshold for human hearing, is the the only option. This does not not necessarily mean that the audio quality will be unintelligeble. It merely means that there one can expect some quality degradation compared to the high quality measure of [20, p. 160] as a result of PWM artifacts [10].

An approach to implementing PWM is presented in section 7.4.

# Chapter 7

# Implementation

In this chapter we will assemble the ideas presented in earlier chapters into a working system. The system components are illustrated in figure 7.1. Implementation details on radio transmission (TX) and reception (RX) will not be discussed.

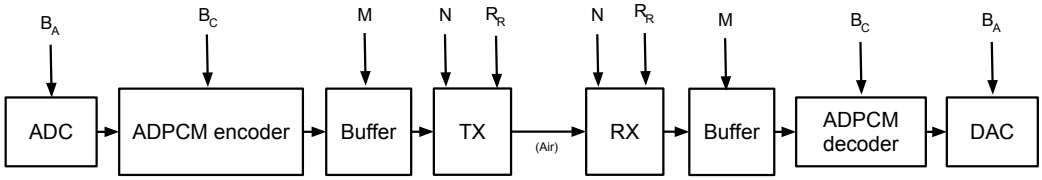


Figure 7.1: System implementation overview.

The following are the features of the system:

- ADPCM compression with AQB
  - Sample-by-sample processing - samples are encoded and decoded instantly upon arrival.
  - Fixed prediction order $P = 1$ and coefficient $\alpha_1 = 0.9$.
- Q15.16 32-bit fixed point number representation.
- Radio transmission with two options:
  - Full-duplex with Gazell protocol (non-optimized) [1].
  - Non-duplex (one side is TX and the other is RX).
- DAC by PWM.

The system has the following parameters:

- $N$ [samples] - the number of samples per packet for radio transmission.
- $M$ [samples] - buffer size.
- $B_A$ [bits] - fixed number of bits for representing each uncompressed sample.
- $B_C$ [bits] - fixed number of bits for representing each compressed sample.
- $R_R$ [kbps] - radio transmission rate.

The codec was simulated in Octave (a MATLAB-compatible language) using 64-bit floating point numbers. Then it was ported to C and rewritten using fixed point numbers. Drivers and other control flow was implemented in C and the codec was inserted into the signal chain. Finally, the code was compiled for the SoC core. Appendix E contains a list of all code files accompanied by a brief description.

## 7.1 Buffers

This section will discuss the design and operation of the two first-in-first-out (FIFO) cues/buffers used in the implementation, as illustrated in figure 7.2.

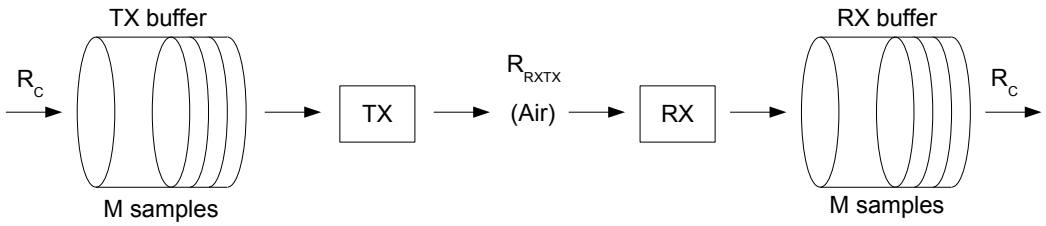


Figure 7.2: Buffer flows from sender to receiver [1].

The ADC is a strictly timed sample producer. Since $N$ samples are needed to build a packet for transmission, a buffer holding $N$ compressed samples as they are gathered is required. However, once the buffer is filled up and transmission is started, we need all of these $N$ samples until the packet is successfully received. While transmission is ongoing, new ADC samples may appear. These must also be stored somewhere, so we need an

additional buffer of size $N$ that is filled up while the former $N$ samples are being transmitted. Therefore the transmitter requires a total buffer size of minimum $M = 2N$.

Similarly, the DAC is a strictly timed sample consumer. Packets will arrive at arbitrary times and stored in a buffer of size $N$. The DAC will read from this buffer and convert sample by sample to its analog equivalent, but the RX should not overwrite that buffer while the DAC is in the middle of reading from it. Therefore we need another buffer of a minimum size of $M = 2N$ at the receiver.

Now, $M = 2N$ is merely a theoretical lower bound on the buffer size. Too see why, consider again figure 7.2. The rate $R_C$ that the TX buffer is being written to and the RX buffer is being read at is considered constant in time. Since the radio transmission rate $R_R$ is merely a theoretical limit, we need to introduce the throughput rate $R_{RXTX}$, which is the effective transmission bit rate. $R_{RXTX}$ is dependent on current channel conditions and so it will vary with time. Indeterministic incidents such as packet losses will call for retransmission, which will reduce $R_{RXTX}$. Therefore, $R_{RXTX} \leq R_R$. Whenever $R_{RXTX} < R_C$, the TX buffer will overflow and consequently the RX buffer will underflow. Conversely, when $R_{RXTX} > R_C$, the TX buffer will underflow and consequently the RX buffer will overflow. Therefore, in practice, $M \gg 2N$. [1, ch. 7] makes an assessment on buffer sizes $M$ versus channel condition parameters and suggests practical values of $M$.

A common way of implementing a FIFO buffer is illustrated by example in figure 7.3. Such a buffer is called a circular buffer as it is indexed modulo $M$. When a sample is ready to be read from the ADC at the transmitter side, it is copied over to the slot pointed at by $P_R$, which is incremented by 1. Similarly, at the receiver side, every time a packet of $N$ samples arrive, they are decoded and copied into the next $N$ slots starting at $P_W$, which is incremented by $N$.

Overflow occurs when $P_W$ catches up with $P_R$ so that $P_R = P_W$. As long as the buffer is under this condition, subsequent incoming samples must be discarded. Conversely, underflow occurs when $P_R$ catches up with $P_W$ so that $P_W = P_R$. As long as the buffer is under this condition, the consumer that reads from the buffer (TX or DAC) will not be fed. In both cases, error concealment, as described in chapter 4, can be applied to patch up the wholes in the sample stream that results from over- or underflow.

Throughout this report, the rate $f_s$ at which the ADC and DAC operates at has, for simplicity, been assumed to be equal. This is, however, not necessarily true in the case where the ADC and DAC are situated on separate physical devices. Because of physical inaccuracies on the SoCs, they are certain to fall out of synchronization and drift apart at some point in time

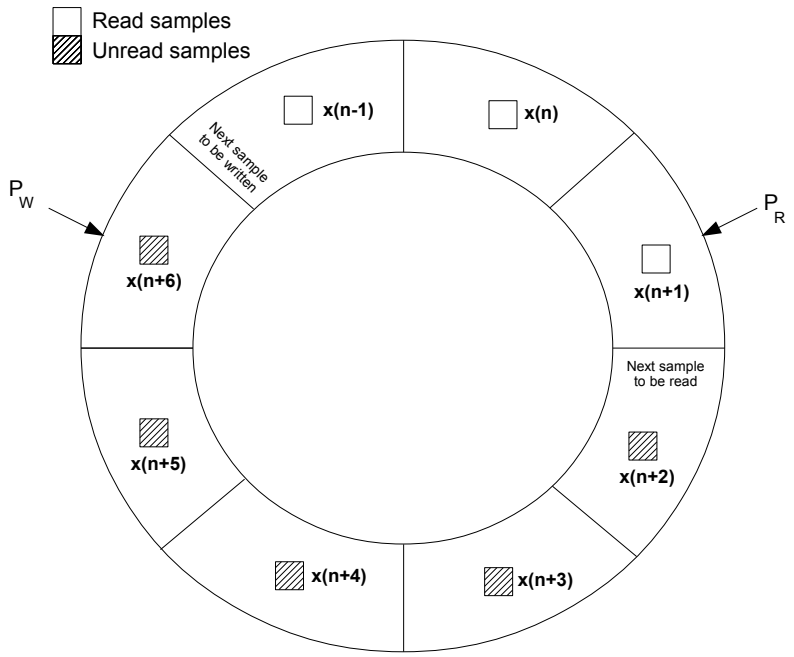


Figure 7.3: Example snapshot of a FIFO buffer state for $M = 8$.

even though their timers are set to the same resolution. This means that the RX buffer is sure to under- or overflow sooner or later, depending on whether the ADC or the DAC has the faster $f_s$. This problem can be overcome by some control logic that periodically senses drifting and adjusts the $f_s$ at the receiver side accordingly. There are numerous methods for solving this problem [21, ch. 5], none of which will not be addressed in this project.

## 7.2 Scheduling for real-time operation

A real-time audio transmission system is a so-called *firm* real-time system, which is defined according to the following property:

> Infrequent deadline misses are tolerable, but may degrade the system's quality of service. The usefulness of a result is zero after its deadline [22, ch. 1].

This is as apposed to a *hard* real-time system, where deadlines result in a total system failure, and *soft* real-time systems, where system performance is merely degraded and whatever is lost is still usable to a certain degree. In our context, missing a deadline means that a sample does not reach the

DAC at its playback time. It is then unusable and should be discarded. This chapter is concerned with organizing the system flow so that losses can be prevented. To minimize the number of deadlines missed we need to use a scheduling policy. This is described next.

## 7.3 Scheduling for half-duplex operation

Half-duplex operation for transmitter and receiver is shown in the left and right flow chart of figure 7.4, respectively.

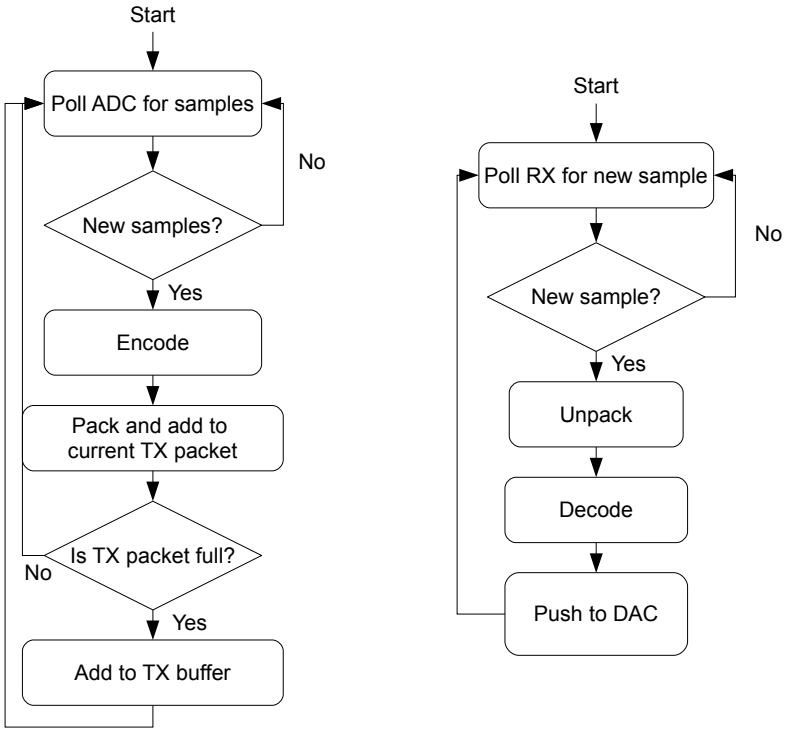


Figure 7.4: Flow chart for transmitter (left) and receiver (right) in half-duplex operation.

Packing and unpacking is the process of assembling and extracting the bits that make the compressed samples, respectively. For real-time operation, we must have that the sum of all processing per sample runs in less than $\frac{1}{f_s}$ seconds [2]. As long as this is enforced, half-duplex operation as in figure 7.4 will execute without missing deadlines.

### 7.3.1 Scheduling for full-duplex operation

The flow chart in figure 7.5 shows how full-duplex operation can be carried out. This is essentially the same as in flow charts in figure 7.4, except that we are toggling between encoding and decoding. It is the main loop of the program, meaning that it is the point of return after the execution of any other process that might interrupt it.

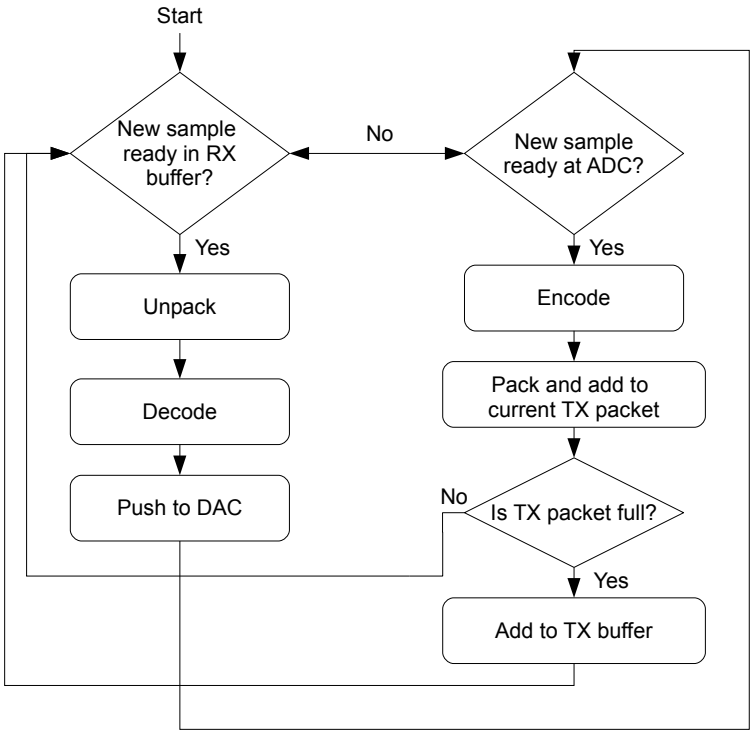


Figure 7.5: Flow chart for both transmitter and receiver in full-duplex operation. This is referred to as the main loop.

A process that interrupts the main loop is commonly called an interrupt service routine (ISR). An ISR has a start time $T_S$, an execution time $T_E$ and may also have a period $T_P$. Besides processes transparent to the programmer, such as radio operation, there are three ISRs that can interrupt the main loop. These are:

- ISR1: A process that runs when a new sample is converted with the ADC. ISR3 sets a flag for the main loop to read and determine if a new sample is ready from the ADC. The process is called every $T_P = \frac{1}{f_s}$ seconds.

- ISR2: A process that runs when a new sample is needed by the DAC to calculate the next duty cycle for PWM. ISR2 is also called every $T_P = \frac{1}{f_s}$ seconds.
- ISR3: A process that runs when a new packet is received at the RX. This process copies the packet to the RX buffer and sets a flag for the main loop to read and determine if new samples is ready to be decoded. ISR3 is a sporadic process and does not have a fixed $T_P$, but is called every $\frac{N}{f_s}$ seconds on average.

It is important to keep the execution time $T_E$ of any ISRs as brief as possible to reduce the probability of deadlines being missed, since the main loop will not be able to execute while an ISR is running. For instance, if the main loop is interrupted for too long, a sample $x(n)$ available at the ADC may be lost because the main loop was not able to fetch it before it was overwritten by a subsequent sample $x(n+1)$ available the ADC. This could be avoided using a buffer for incoming ADC samples, but that will introduce an unnecessary delay. An approach that avoids this is to keep $T_E$ short enough to avoid such problems for all ISRs. Assigning higher priorities to ISRs with shorter values of $T_E$ may also help reduce the probability of missing a deadline [22, ch. 13]. For instance, if ISR3 runs slower than ISR2 and ISR1, assign higher priorities to ISR1 and ISR2 so that they may interrupt ISR3, but not the other way around [1].

Also, consider the scenario where a call to an ISR will be blocked because the previous call to the same ISR is still executing. Because ISR3 is called $N$ times as seldom as ISR1 and ISR2 on average, this scenario is less likely to happen for ISR3 than for ISR1 and ISR2. This is another reason to give ISR1 and ISR2 higher priorities than ISR3.

As stated earlier, real-time operation requires that the sum of all periodic processing runs in less than $\frac{1}{f_s}$ seconds [2]. Since encoding and decoding is taking up most of this time, every ISR must have an execution time $T_E \ll \frac{1}{f_s}$. With rate-monotonic scheduling, one can find an optimal priority ordering if all processes have a fixed start time $T_S$, execution time $T_E$ and period $T_P$. However, ISR3 does not have a deterministic start time $T_S$ as we do not know when to expect a packet to appear at the RX. Therefore we cannot use the schedulability tests associated with the rate-monotonic scheduling policy to arrive at priority ordering that guarantees that all deadlines will be held. We will have to settle with the reasoning above. System processes and their respective priorities are shown table 7.1.

[1] This type of static priority scheduling policy is called rate-monotonic scheduling [22, ch. 13].

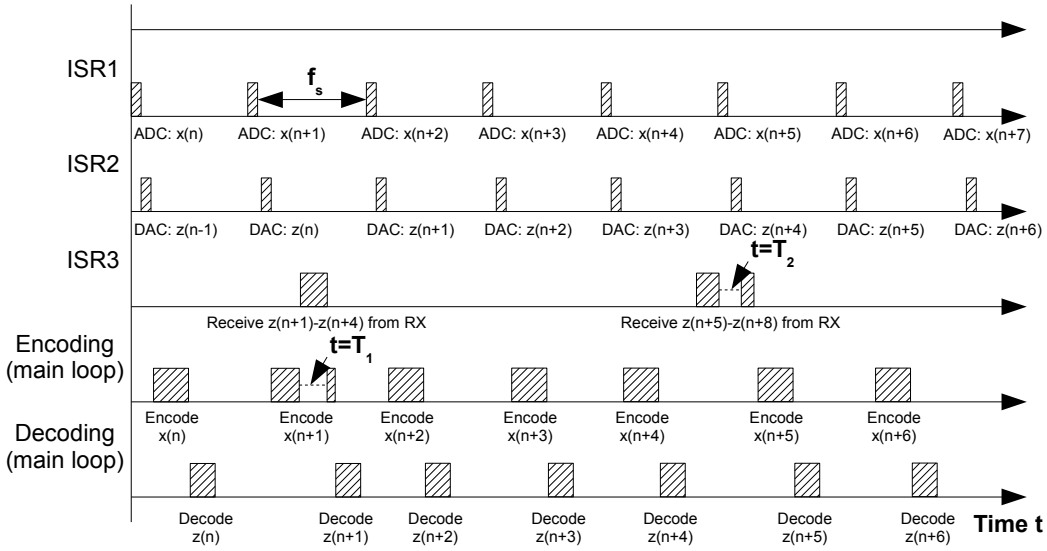


Figure 7.6: Scheduling example.

Table 7.1: System processes and their respective priorities.

| **Process** | **Priority** |
|---|---|
| Main loop | 0 |
| ISR3 | 1 |
| ISR1 | 2 |
| ISR2 | 2 |
| Other | 3 |

Figure 7.6 illustrates an example of full-duplex scheduling on an endpoint using the priorities in table 7.1 and is explained as follows. The shaded blocks indicate that an ISR or a main loop part is executing, so the length of these represent $T_E$. $x(n)$ is the signal from the ADC and $z(n)$ is the received signal from the RX. ISR1 and ISR2 is called every $T_P = \frac{1}{f_s}$ and are succeeded by encoding and decoding, respectively. As $N = 4$ samples in this example, ISR3 is called every $T_P = \frac{4}{f_s}$ on average. A dotted line represents a process that has been interrupted and not yet completed. ISR3 can only interrupt the main loop. This is seen at time $t = T_1$, where encoding is interrupted. ISR1 and ISR2 can interrupt any other process in the figure. This is seen at time $t = T_2$, where ISR3 is interrupted.

## 7.4 PWM

This section will describe an approach to implementing PWM as described in chapter 6. This is achieved with a counter running at $f_t$ and the three counter compare registers CC0, CC1 and CC2 [3]. The operation of the implementation is illustrated by example in figure 7.7.

A counter is started and its value $T_{counter}$ is incremented by 1 every $\frac{1}{f_t}$ seconds. All CC registers are set to toggle the selected GPIO pin for PWM output whenever $T_{counter} = \text{CC}$. Initially, $\text{CC0} = 2^{B_A}$. Then, when $T_{counter} = \text{CC0}$, algorithm 1 is run, where the next sample to be output is $\tilde{x}(n) \in \{0, 2^{B_A} - 1\}$. Variable $V$ holds the name of the CC register that was set in the previous call to algorithm 1.

**Algorithm 1** ISR2. Called when $T_{counter} = \text{CC0}$.

$\text{CC0} \leftarrow \text{CC0} + 2^{B_A}$
**if** $V = \text{CC1}$ **then**
  $\text{CC2} \leftarrow \text{CC0} + \tilde{x}(n)$
**else**
  $\text{CC1} \leftarrow \text{CC0} + \tilde{x}(n)$
**end if**

As described in chapter 6, $2^{B_A} f_c = f_t$. Therefore algorithm 1 is called every $\frac{1}{f_c} = \frac{2^{B_A}}{f_t}$ seconds to pick up a new sample of $\tilde{x}(n)$, and CC1 and CC2 sets the correct duty cycle for the current pulse. This is essentially how this PWM-based DAC implementation operates. The example in figure 7.7 illustrates this for $f_s = f_c$, which, as discussed in chapter 6, violates the high quality PWM output criteria of $f_s \leq 12 f_c$.

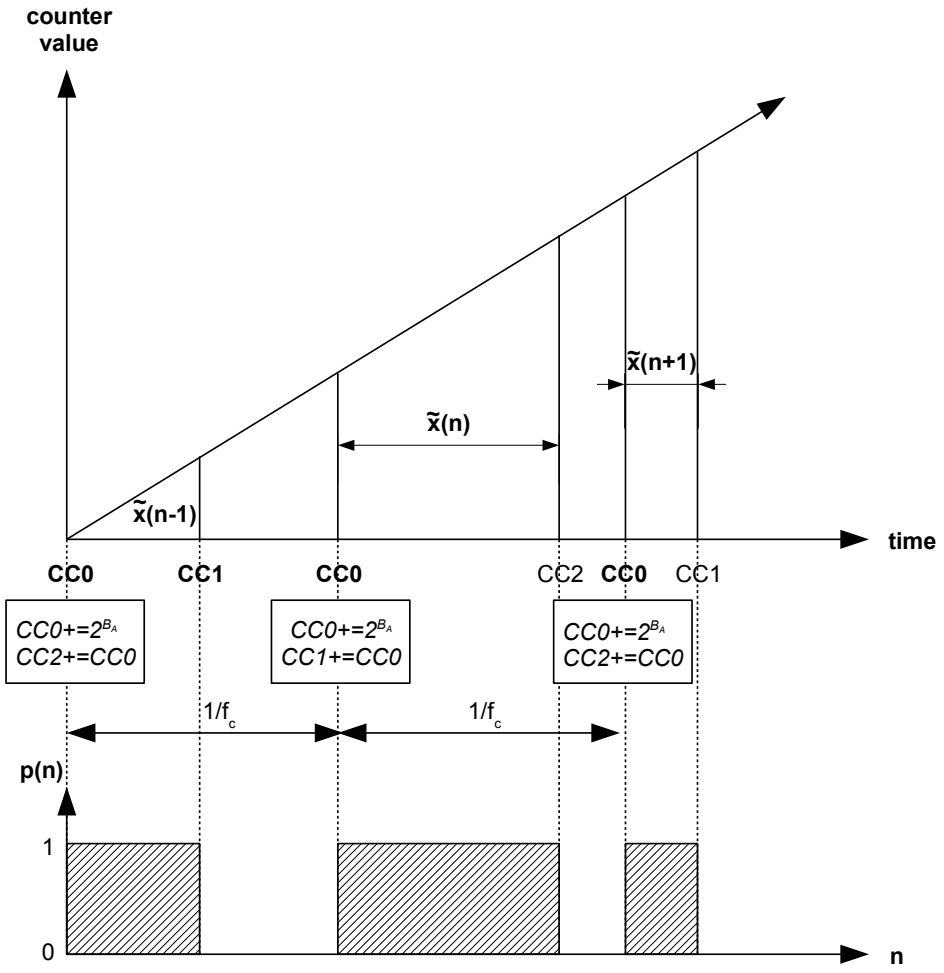


Figure 7.7: PWM operation from counter perspective. The boxes under their respective CC0 are the instructions that are carried out at the indicated time. In this example, $f_s = f_c$

The reason we need both CC1 and CC2 is because of real-time demands. Too see this, consider the following: if we only used CC1, there would be a problem running algorithm 1 if the current sample $\tilde{x}(n)$ was close to $2^{B_A}$ since algorithm 1 needs time to execute. In this case, $T_{counter}$ may have passed the values of CC1 and CC2 before they are set. Then the current GPIO pin toggles would be skipped. PWM operation is illustrated by another example in figure 7.8, where each of the pulse widths of CC0 shows the execution time $T_E$ of algorithm 1. Although fast, running algorithm 1 requires some time to

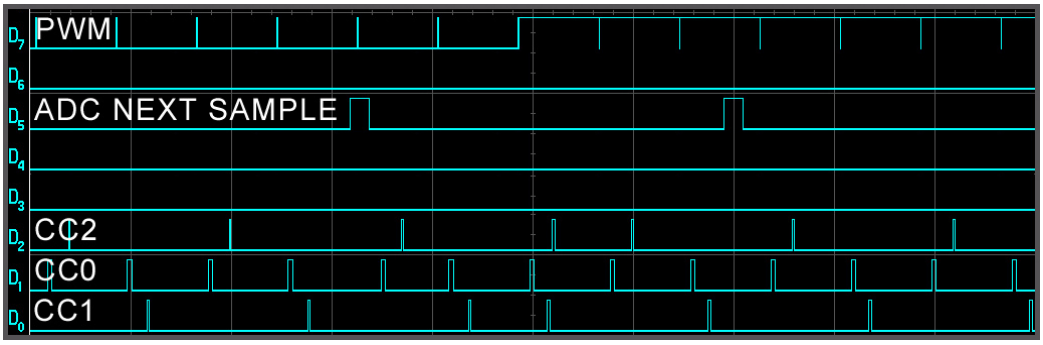


Figure 7.8: PWM timers and output scope snapshot.

run and therefore three, not two, compare registers are needed for real-time operation.

The SoC has hardware logic for toggling an output GPIO pin when a counter reaches a CC value [3]. Therefore no software processing is used for comparing values of CC1 and CC2 with $T_{counter}$.

## 7.5 Delays

Figure 7.9 sums up all system delays. The sum of these is the system sample latency $T_L$ in time from ADC input to DAC output. As the sum of all processing delays is required to be less than $\frac{1}{f_s}$ for real-time operation, it cannot take longer time to process a sample than to play it back. Therefore we must have $T_{Encoding} + T_{Packing} + T_{Unpacking} + T_{Decoding} < \frac{1}{f_s}$. As discussed in section 7.1, we need an $M \geq 2N$ at both the transmitter and receiver end to prevent buffer over- and underflow. There is also small transmission delay of about $T_{Radio} = 3$ ms [1]. $T_M$ is the delay resulting from the TX and RX buffer size $M$ because the samples needs to move through the entire buffer as with any FIFO buffer. For full-duplex communication, we should have $T_L \leq T_{max} = 150$ ms point-to-point [1].

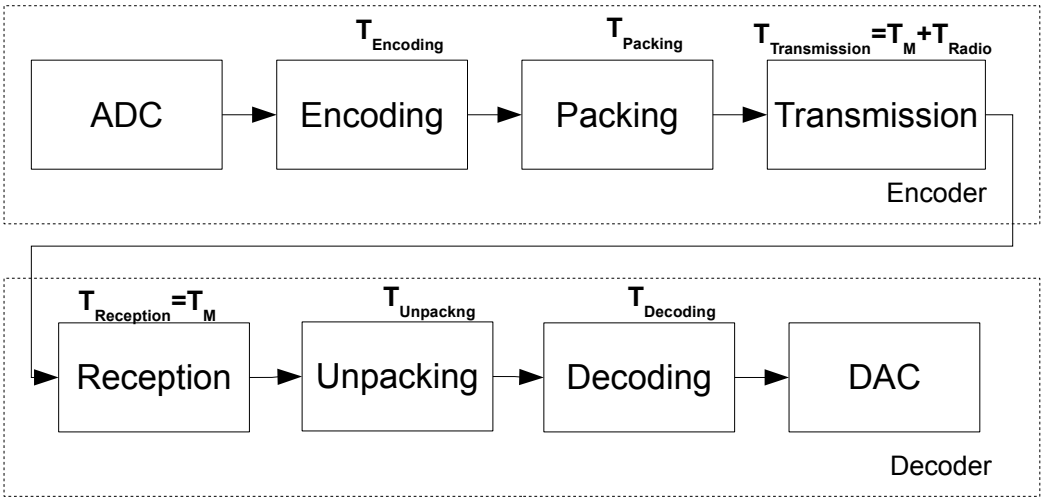


Figure 7.9: System delays.

[1] has conducted simulations on the SoC radio to find a lower bound on the TX and RX buffer size $M$ that should provide safety against buffer over- and underflows. The simulations were carried out with for a typical radio configuration[2] using a rate of $R_C = 196$ kbps. The maximum delay due to $M$ reported in this simulation is $2T_M = 48$ ms. As this project will only be concerned with $R_C < 196$ kbps, we will use $T_M = \frac{48}{2} = 24$ ms for the remainder of this report.

Consider a worst-case example. As $T_M = 24$ ms,

$$T_{Transmission} + T_{Reception} \approx 48\text{ms} \tag{7.1}$$

If total processing delay is $\frac{1}{f_s}$ seconds, then using mode 4, which has the lowest sampling frequency $f_s$,

$$T_{Processing} = T_{Encoding} + T_{Packing} + T_{Unpacking} + T_{Decoding} \approx 0.3\text{ms} \tag{7.2}$$

Then the total system latency is

$$T_L = T_{Processing} + T_{Transmission} + T_{Reception} + T_{Radio} \approx 51.3 < T_{max}\text{ms} \tag{7.3}$$

As this is a worst case, latency is generally not considered a real problem even in the most demanding configuration in terms of system delays.

[2] A packet size of 50 bytes and a bit error rate (BER) of $10^{-3}$.

## 7.6 Error concealment

The value of $M$ proposed by [1] in the previous section guarantees that no over- and underflows will occur. This means that all packets will arrive in time at the receiver, which again means that there will be no need for error concealment.

Error resilience does however stand as an important feature when designing an audio communication system, as applications might exists which finds that the $M$s suggested in [1] will result in unacceptable delays. Then a smaller $M$ must be chosen, which would no longer hold a guarantee against over- and underflows. Also, the radio may use a different protocol than the one tested in [1], which may not do retransmission at all. This means that if a packet is lost, no action is taken to retrieve it, and consequently some error concealment method should be applied to minimize the resulting error.

For these reasons error concealment has not been implemented for this project.

# Chapter 8

# Performance tests

This chapter will analyze the performance of the system in chapter 7. The system has many degrees of freedom. The ones relating to coder efficiency will be examined here. Performance is given by a measure of the error that is produced by the different stages in the signal path. The audio files analyzed in this chapter is referenced in appendix F.

Figure 8.1 shows all error sources of the system and where they appear in the signal chain. The sampling error is the quantization noise that is

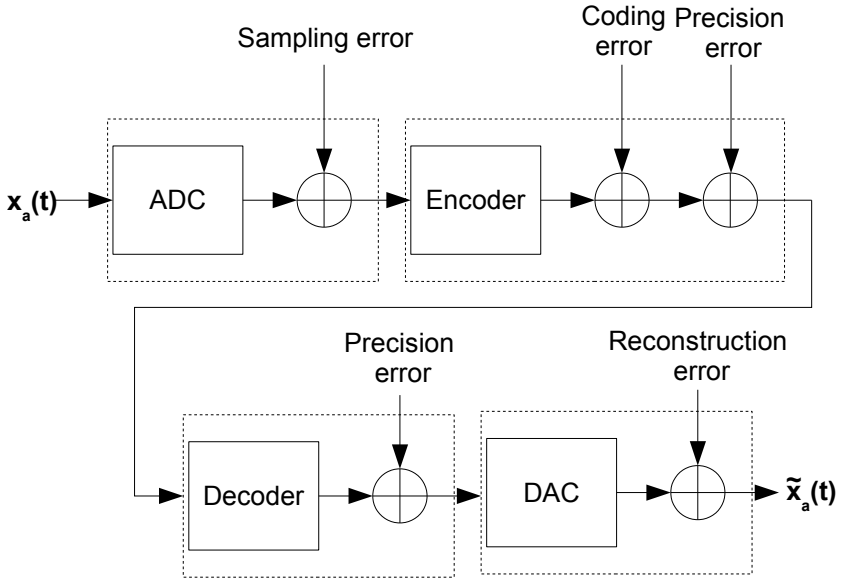


Figure 8.1: System error sources.

introduced when digitizing the analog signal $x_a(t)$ with the on-chip ADC. Compression yields a coding error and all processing with finite precision may introduce a rounding error. If packets gets lost or corrupted, we get

a transmission error (which of course overrides all other errors). Finally the mechanics of the DAC produces a reconstruction error when generating $\hat{x}(n)$, the reconstruction of $x(n)$. Note that these errors are not necessarily additive, even though they are depicted this way in figure 8.1. This chapter will examine the impact the errors have on performance by experiment. They will be tested both individually and together.

The mean squared error (MSE) metric for any $M$-sample sequences $u(n)$ and $v(n)$, $n = 0, 1, \ldots, M-1$, as defined in equation 8.1 will be used as a measure of the error between any signals $u(n)$ and $v(n)$.

$$\|\mathbf{u} - \mathbf{v}\|^2 = \frac{1}{M} \sum_{n=0}^{M-1} |u(n) - v(n)|^2 \tag{8.1}$$

$\mathbf{u} = [u(0), u(1), \ldots, u(M-1)]^T$ and $\mathbf{v} = [v(0), v(1), \ldots, v(M-1)]^T$ is used as a vector notation for $u(n)$ and $v(n)$, respectively.

Figure 8.2 shows the measurement points for the MSE measurement that will be discussed throughout this chapter. The source signal $x_a(t)$ is a 16-bit

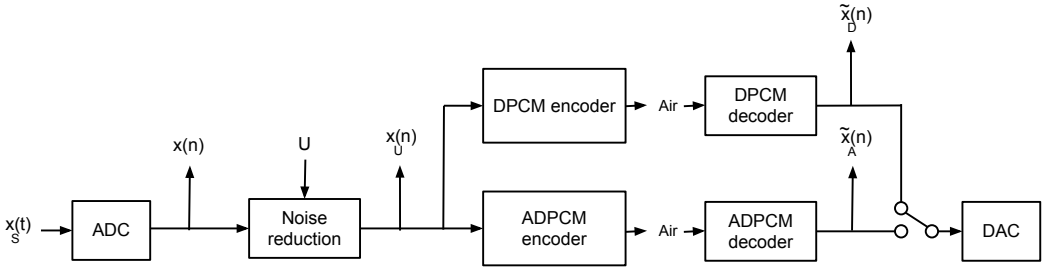


Figure 8.2: Reference taps for error measurements.

signal sampled at 44.1kHz (CD quality) played back through an industry standard PC DAC, ready to be sampled be the SoC ADC to produce $x(n)$. Parameter $U \in \{0 = \text{bypass}, 1 = \text{on}\}$ indicates whether noise reduction is applied to the signal prior to compression or bypassed so that $x_U(n) = x(n)$. Unless stated otherwise, $U = 0$. $\tilde{x}_D(n)$ and $\tilde{x}_A(n)$ represents the signal after DPCM and ADPCM decoding, respectively.

For all performance tests, a sound file containing Norwegian speech over 8 seconds was used. The resulting sampled signal $x(n)$ consisted of $M = 8f_s$ samples, $f_s$ being dependent on the ADC mode. To compare $x_a(t)$ with the other signal taps in figure 8.2, $x_a(t)$ was decimated to $f_s$ for each mode and all files were normalized to $[-1, 1)$. This was achieved using PC software. The original sound file as well as processed versions of it is included in the ZIP file accompanying this report.

## 8.1 A measure for operation count

Before proceeding, we will define a common measure for the number of arithmetic operations, $O$. The SoC core is known to be able to do one operation in almost one instruction, and almost one instruction every clock cycle $\frac{1}{f_C}$ on average [3]. $O$ can therefore be used as an approximate measure on the runtime of any algorithm on the SoC by inspection of that algorithm. An operation can be a comparison, a shift, addition or even a multiplication since the SoC core has a 1-cycle multiplier. Memory loads and stores will not be included when calculating $O$ since the caching policies of the SOC core has not been investigated. An operation is of course a simplified measure, since the compiler has the final say in the number of instructions needed for an algorithm. The $O$ measure will be used throughout the remainder of this report for comparison of algorithms.

## 8.2 Comparison of compression schemes

Table 8.1 shows the rates $R_C = f_s B_C$ kbps associated with each ADC mode for a selected range of $B_C$ bits per compressed sample. $R_C < R_R = 250$ kbps, the lowest radio transmission rate configuration, for every $R_C$ in this table[1].

Table 8.1: Bit rates $R_C$ with compression for a select range of $B_C$ per ADC mode.

| Mode | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $f_s$ [kHz] | 50 | 28 | 14.7 | 3.8 |
| $R_C$ ($B_C = 2$) [kbps] | 7.6 | 29.4 | 56 | 100 |
| $R_C$ ($B_C = 3$) [kbps] | 11.4 | 44.1 | 84 | 150 |
| $R_C$ ($B_C = 4$) [kbps] | 15.2 | 58.8 | 112 | 200 |

Figure 8.3 shows the performance of DPCM and ADPCM. As expected, DPCM performs worse than ADPCM. What is not so intuitive, is that $\|\mathbf{x} - \tilde{\mathbf{x}}_\mathbf{A}\|^2 < \|\mathbf{x_a} - \tilde{\mathbf{x}}_\mathbf{A}\|^2$ for all values of $B_C$. The reason for this is because in the chosen ADPCM scheme there is an inherent noise reduction feature that is in practice similar to the one presented in chapter 5. This was discovered by inspection of $\tilde{x}_A(n)$. Therefore, as seen in figure 8.3, $\tilde{x}_A$ is more similar to $x_a(t)$ than $x(n)$, which is ultimately what we want.

[1] In practice we must have $R_C + R_S \leq R_R$, where $R_S$ is some additional rate due to the protocol adding redundant information for more robust transmission. $R_S$ will be considered negligible in this report.

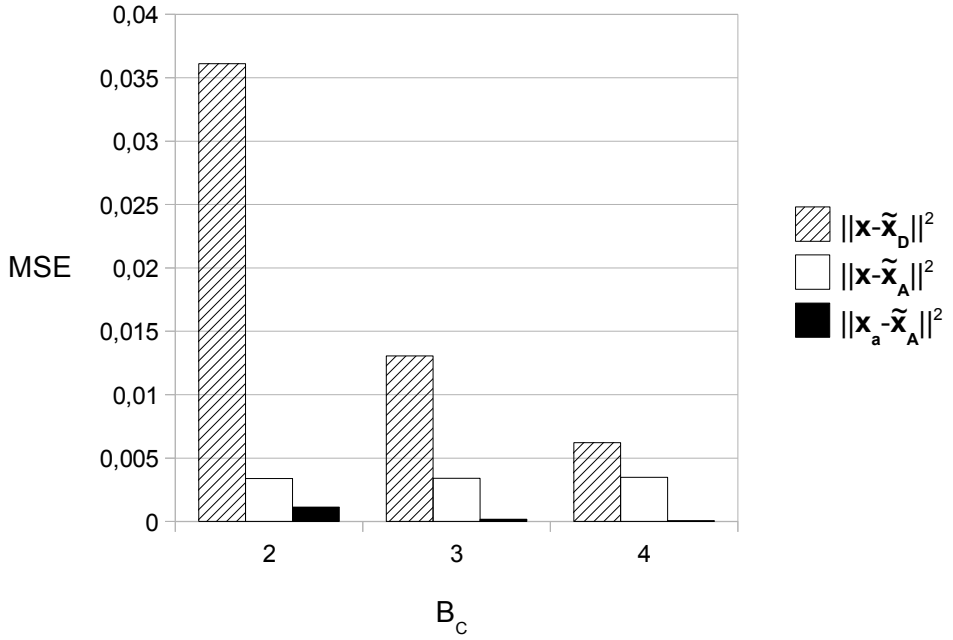


Figure 8.3: Comparison of compression schemes for different values of $B_C$.

The inherent noise reduction is explained as follows. For $B = 3$ and up, $M(|C(n)|)$ is more likely to fall into the lower half of the M-table, where the step size multipliers are less than 1. Therefore, codewords $C$ gradually fall to $C = 0$ for silent periods, where only ADC noise and distortion is present, effectively removing these. This means that a type of noise reduction very similar to the one described in section 5.3 comes for free with this specific ADPCM codec implementation. This works best for $B_C = 3$ and $B_C$=4 since for $B = 2$, the table is too coarse for step size $\Delta(n)$ to reach its minimal value.

The noise reduction feature also gives a pointer to why $||\mathbf{x} - \tilde{\mathbf{x}}_\mathbf{A}||^2$ is close to equal for $B_C = 2$, $B_C = 3$ and $B_C = 4$: the MSE measure is more dominated by the noise produced by the ADC, which is partially removed in $\tilde{x}_A$, than the noise produced by the codec.

## 8.3 Impact of noise reduction

The purpose of this analysis is to examine whether the chosen noise reduction scheme makes $||\mathbf{x_a} - \mathbf{x_U}||^2 < ||\mathbf{x} - \mathbf{x_U}||^2$ with $U = 1$, effectively reducing the noise introduced by sampling with the SoC ADC. The noise reduction algorithm used parameters $T_{gate} = 20$ ms, $G_t = -48$ dB and $G_a = 48$ dB. As figure 8.4 illustrates, noise reduction does reduce the ADC noise, but only by

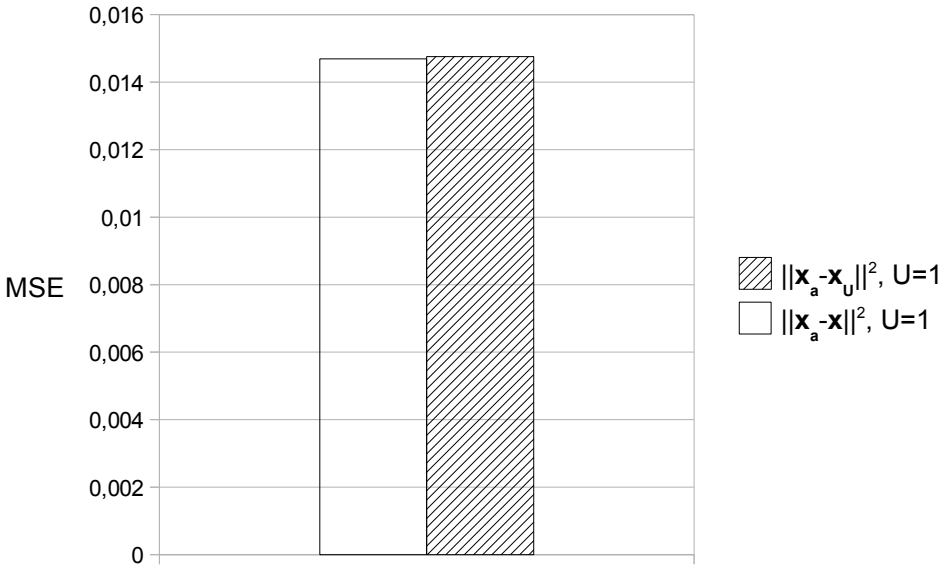


Figure 8.4: Impact of noise reduction without compression.

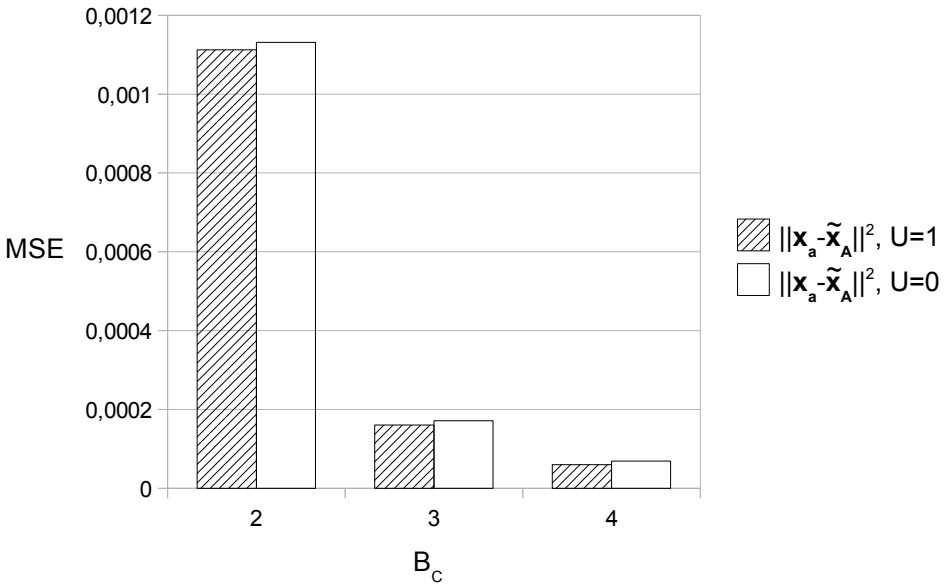


Figure 8.5: Impact of noise reduction with compression.

a small margin. Also, as figure 8.5 shows, the gain in using noise reduction with ADPCM is less than what is gained by the inherent noise reduction property of the ADPCM that was seen in figure 8.3. Therefore, when using this specific ADPCM, noise reduction is unnecessary.

## 8.4 Fixed point versus floating point performance

Figure 8.6 shows $||\mathbf{x} - \tilde{\mathbf{x}}_\mathbf{A}||^2$ for the MATLAB-simulated ADPCM codec and the Q16 SoC implementation.The difference is relatively small, and comes

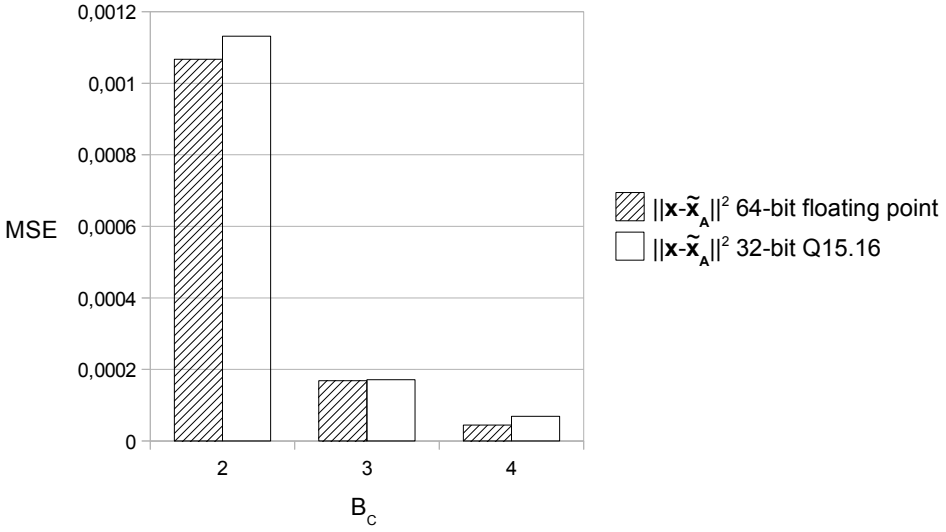


Figure 8.6: Floating point versus fixed point performance for ADPCM codec.

from the fixed resolution of the adaptive step size $\Delta(n)$. While 64-bit floating point precision can represent a step size as small as $\Delta(n) = 2^{-53}$ (see section 3.1), the 32-bit Q16 implementation can only represent $\Delta(n) = 2^{-16}$ since the point is fixed. Step sizes this small was observed by inspection. This could call for a different Q$i.f$ format to represent the step size. However, informal listening tests carried out by the author concluded that the audible difference between the 64-bit floating point $\tilde{x}_A(n)$ and the 32-bit Q16 $\tilde{x}_A(n)$ is small enough to say that there is too little to gain from altering the chosen Q$i.f$ format. As stated above, the MSE in figure 8.6 also suggests this.

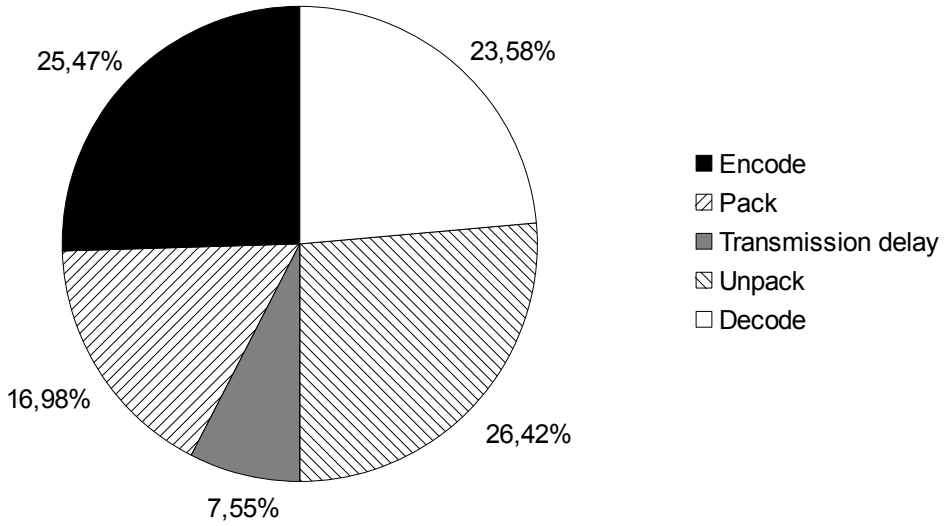


Figure 8.7: Relative time usage for different codec stages of DPCM scheme.

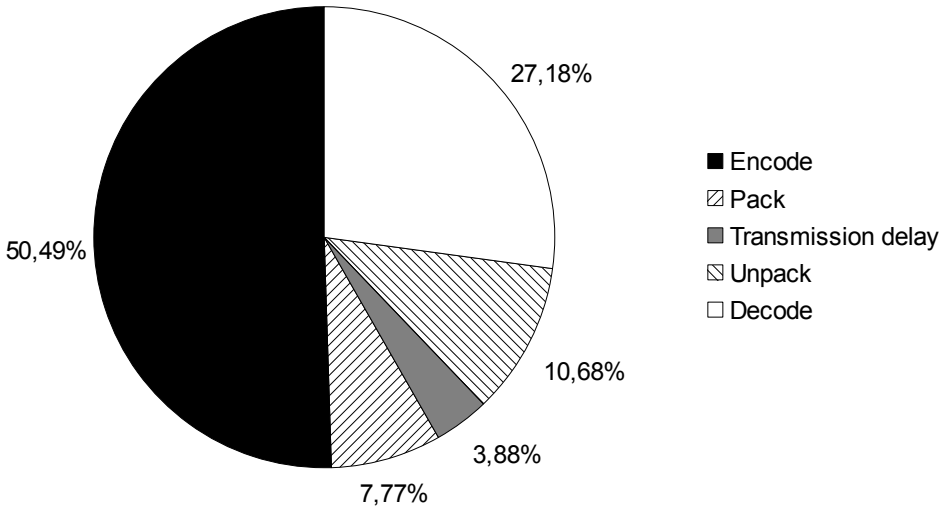


Figure 8.8: Relative time usage for different codec stages of ADPCM scheme.

## 8.5 Profiling the implementation

Figures 8.7 and 8.8 are pie charts showing the relative time use for the main stages of the DPCM and APDCM codecs, respectively. The data was derived from inspection of oscilloscope snapshots where the start and end times of encoding, packing, unpacking, decoding and transmission delay was tracked. The point of this analysis is to get an idea of where one should start optimizing the codec and also improve on the scheduling policy. For DPCM, it is seen that encoding and decoding time is fairly equal. This is because the midrise quantizer, which is the principal difference between the DPCM encoder and decoder, has a simple and fast implementation. Packing and unpacking is almost as time consuming as encoding and decoding, which suggests that they are open for optimization. For ADPCM, however, the encoder is by far the most complex stage since it needs to calculate the next step size, encode the current sample and also decode it for use in the next step size adaption, while decoding only calculates the next step size and decodes the current sample (as discussed in section 2.4).

Profiling the system also gave a measure on how much time one the system uses for processing a sample though the entire signal chain: from the ADC, through the encoder, over the radio and into the decoder before it is finally present at the DAC on the receiver side. From this, an approximation on the maximal sample rate that the ADC and DAC may operate at for each codec is produced. To accompany this the number of operations per sample is found by code inspection. The data is presented in table 8.2. The basic DPCM codec may carry out full-duplex real-time operation far beyond the sample rate for ADC mode 3 ($f_s = 14.7$kHz) because of its low complexity. ADPCM will only manage this for half-duplex (one way at a time) operation, but has an objective quality far superior of that of the DPCM coder, as seen in section 8.2. Lastly, the complex ADPCM that was briefly discussed in chapter 2 was also implemented and profiled to get a measure on the time consumption of a significantly heavier compression scheme.

Table 8.2 holds the most important results of this chapter since it provides data that can be used to measure against any other compression scheme.

Table 8.2: Maximum sample rate for real-time operation per compression scheme.

| | DPCM | ADPCM | Complex ADPCM |
|---|---|---|---|
| Max rate half-duplex [kHz] | 37 | 15 | 6 |
| Max rate full-duplex [kHz] | 18 | 9 | 3 |
| [O/sample] | 53 | 231 | 809 |

As mentioned in the introduction of this report, the goal of this project is not to develop a unique, optimized codec but to get a measure on what bounds on real-time operation the SoC presents in terms of complexity. For example, table 8.2 states that an algorithm that needs $O = 231$ operations for processing one sample can be expected to run at approximately $f_s = 10$ kHz if both encoding and decoding needs to be carried out simultaneously. This is of course a crude measure, as different architectures have different properties and their compilers will compile the code base differently, but this is never the less a point of reference. Also, from section 8.2 we have a measure on how well such an algorithm can perform in terms of MSE, although this is neither an upper nor a lower bound. It is also merely a point of reference.

This project has mainly been concerned with testing half-duplex operation without retransmission features as a proof-of-concept, while [1] has investigated the accompanying full-duplex protocol called Gazell. As mentioned in chapter 1, this protocol is designed and optimized for short, infrequent and sporadic transmissions from wireless keyboards and mouse peripherals, which is the opposite of the requirements for this project. The protocol was tested with the simple DPCM codec, but could not deliver full-duplex transmission faster than approximately $f_s = 244$ Hz due to limitations in the protocol. As will be seen in chapter 9, there are exists at least one audio applications for such a low bandwidth.

## 8.6 Memory requirements

As stated in chapter 1, the SoC allows program sizes up to 256 kB. Using an optimized compiler, the total program size is approximately 6 kB.

The SoC has 16 kb RAM available. RAM requirements for the implementation of this project are about 2 kB, not counting RX and TX buffers. Assuming worst-case buffer sizes of 1200 bytes [1, ch. 7] to prevent over- and underflows, which there must be two of (TX and RX buffer) for full-duplex operation, we arrive at a total of approxiamtely 4.5 kb of RAM.

Summing up, the memory resources provided by the SoC are sufficient for this implementation.

# Chapter 9

# Applications

With the preceding chapters in mind, this chapter will discuss some applications for the SoC. These will focus mainly on using the on-board ADC, ADPCM compression and PWM for audio output. All applications will use DAC by $B_A$-bit PWM unless otherwise stated.

## 9.1 Rates and delays

Before presenting the applications, the ADC modes with compressed and uncompressed bit rates and their respective delays is summarized in table 9.1 as a reference.

## 9.2 Full-duplex real-time speech communication system

A stand-alone speech communication system using our SoC is interesting because of the price range of the SoC. Such a system would be private, wireless and for two users at a time. It is illustrated in figure 9.1. The two apposing sides could be, for example, headsets with microphones. This is by far the most demanding application for the device since we need to do decoding and encoding simultaneously.

For high-bandwidth speech ($f_b < 7$ kHz [19]) we need to use ADC mode 3 or higher. A suggested configuration for this application is therefore mode 3 ($f_s = 14.7$ kHz and $f_b = 7.35$ Hz) and $B_C = 4$. According to table 9.1, this will produce a compressed bit rate of $R_C = 58.8$ kbps.

The device also have built-in support for AES-encryption which could be used for security against eavesdropping.

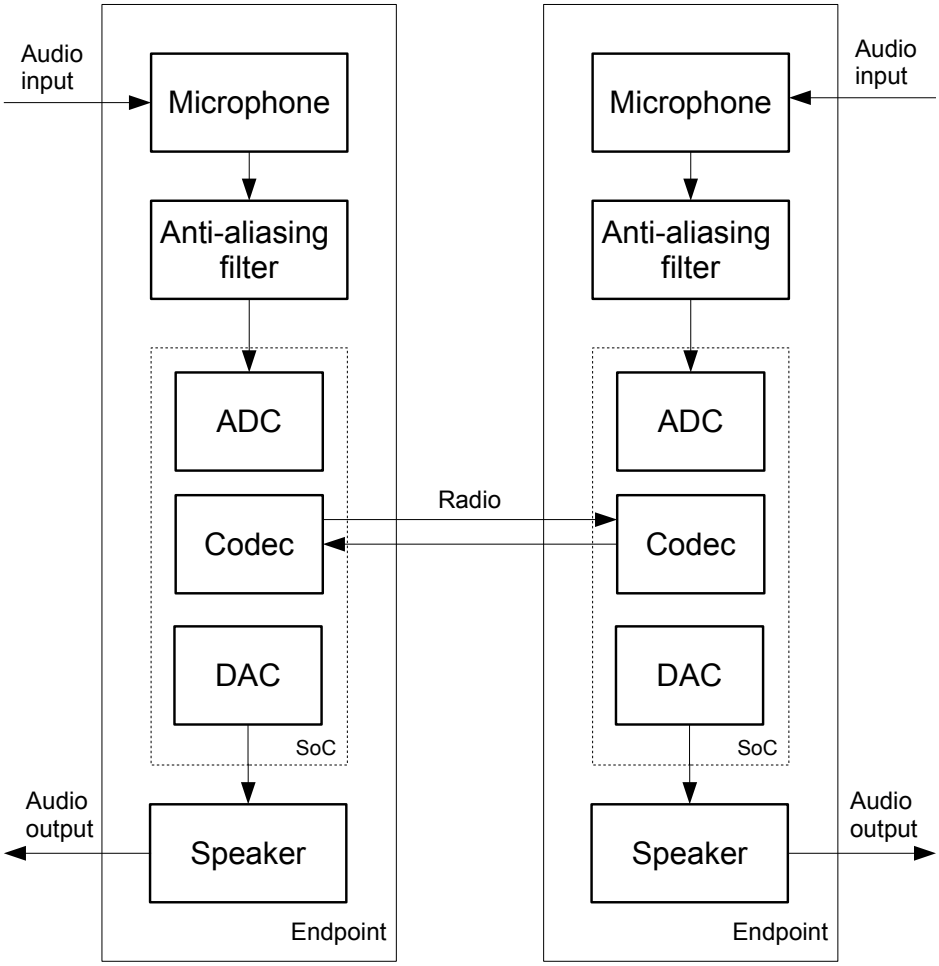


Figure 9.1: Full-duplex real-time audio communication system.

Table 9.1: ADC modes versus bit rates and delays.

| **Mode** | $f_s$ [kHz] | $B_C/B_A$ [bits] | Bit rate $R_C$ [kbps] |
|---|---|---|---|
| 1 | 50 | 2 | 100 |
| | | 3 | 150 |
| | | 4 | 200 |
| | | 8 | 400 |
| 2 | 28 | 2 | 56 |
| | | 3 | 84 |
| | | 4 | 112 |
| | | 9 | 252 |
| 3 | 14.7 | 2 | 29.4 |
| | | 3 | 44.1 |
| | | 4 | 58.8 |
| | | 10 | 147 |
| 4 | 3.8 | 2 | 7.6 |
| | | 3 | 11.4 |
| | | 4 | 15.2 |
| | | 12 | 45.6 |

## 9.3 ITU-T G.722 wideband speech codec

The speech communication system above may be adapted to fulfill a ITU-T codec standard, the G.722. This can be achieved using some extra analog components at both sender and receiver and two input simultaneously at the ADC ($L = 2$). The system is illustrated in figure 9.2. With an input signal bandwidth of $f_b$, the QMF (quadrature mirror filter) divides the input signal into two separate audio streams, one in the subband $[0,\ \frac{f_b}{2}]$ and one in the subband $[\frac{f_b}{2},\ f_b]$. The two signals are then both decimated by a factor of two so that the total bit rate $R_A$ is the same as for the input audio. Then each of these streams are coded independently using ADPCM. This is called subband coding. The point of this is to allocate a different number of bits $B_1$ and $B_2$ for the high-pass and low-pass stream, respectively. This is beneficial because a the input signal typically has more energy in one stream than the other, so one can expect a better SNR compared to $B_C = B_1 + B_2$ for a full-band ADPCM. The signal is then multiplexed and the reverse process is carried out at the receiver side.

Notice that since this application requires two analog inputs. Referring to chapter 5, we have $L = 2$. This means that $f_b \leq \frac{f_s}{2L}$ for any mode because of aliasing and ADC sample rate constraints. Alternatively, the QMF filters may be implemented in the digital domain, but this would require extra

processing and it is not clear whether the SoC would handle this or not.

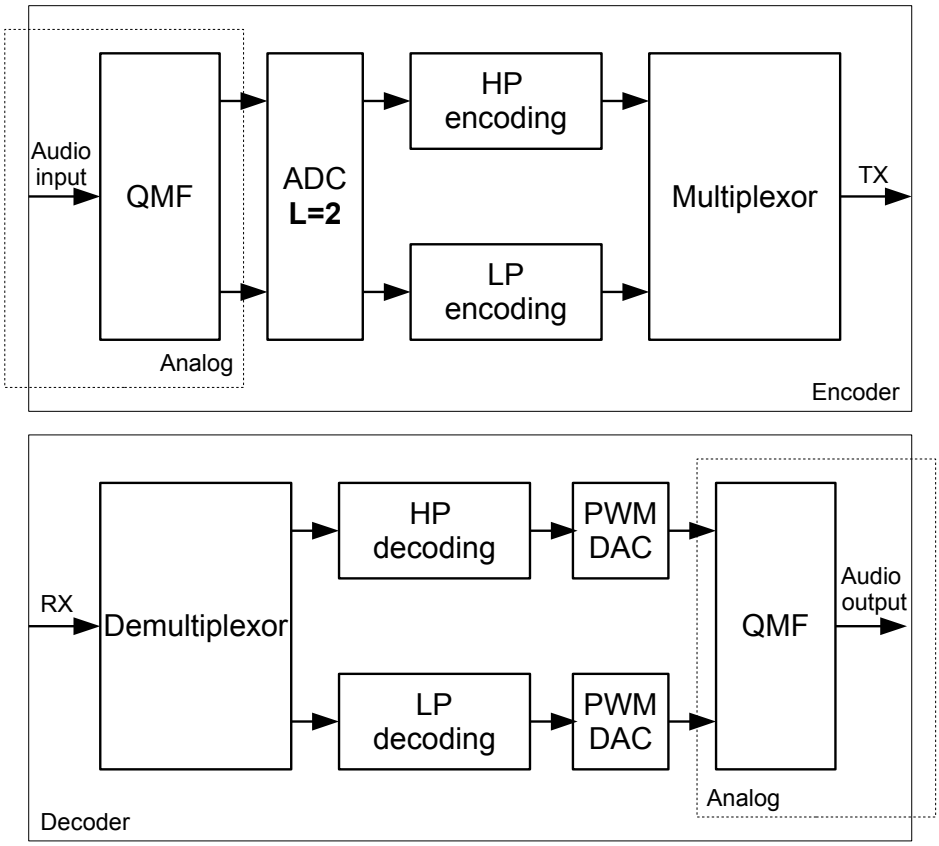


Figure 9.2: ITU-T G.722 wideband speech codec implementation using the device.

A suggested configuration for this application is mode 3 ($f_s = 14.7$ kHz) and $B_C = 4$ bits with $B_1 = 3$ bits and $B_2 = 1$ bit. According to table 9.1, this will produce a compressed bit rate of $R_C = 58.8$ kbps.

## 9.4 Wireless transmission of sound for subwoofer

This is a relatively low-complex application where compression might not be needed since the in-air bit rate is very low. Also, transmission is one-way. The system is illustrated in figure 9.3.

[6, p. 268] defines the bandwidth of a subwoofer as less than $f_b = 150$ Hz. Clearly, mode 4 ($B_A = 12$, $f_s = 3800$Hz) is the right ADC mode for this

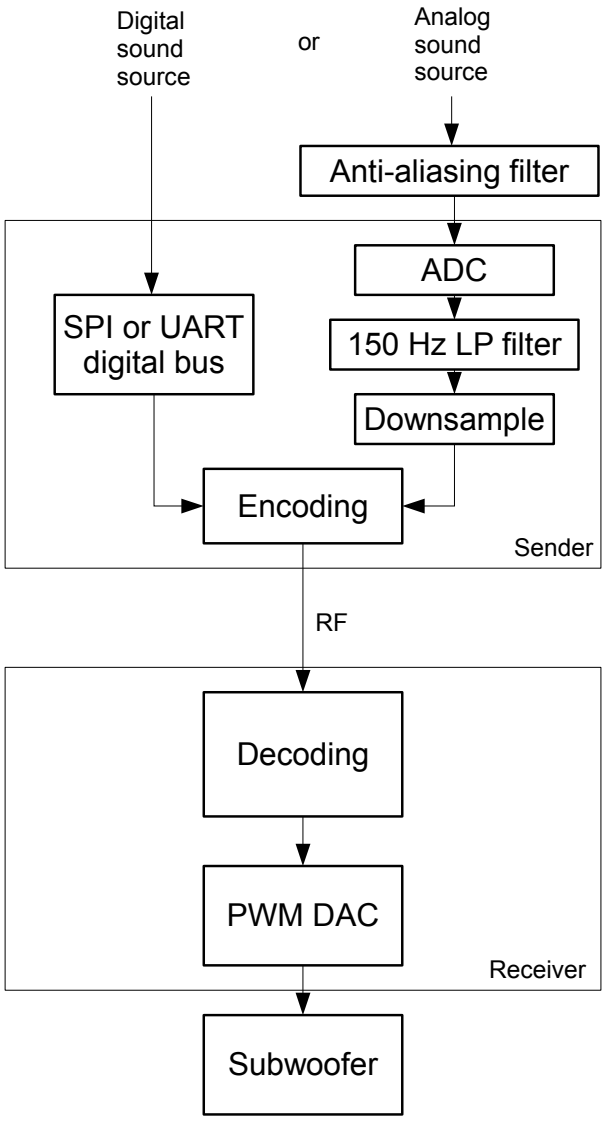


Figure 9.3: Wireless transmission of sound for subwoofer.

application. Since the bandwidth is low, we can downsample the signal by from $f_s = 3800$ Hz to $f_s = 300$ Hz, reducing the bandwidth to $f_b = 150$ Hz. Then, since $B_A = 12$ bits for mode 4, the resulting uncompressed bit rate is $R_A = 2 \cdot 150 \cdot B_A = 3.6$ kbps. Since this $R_A$ is only about 1.5% the size of the lowest possible radio rate $R_R = 250$ kbps, it can be argued that compression is uneccessary.

## 9.5 A source for voice recognition

Figure 9.4 shows how the device can be used to transmit audio wireless to a third party device that performs voice recognition. This is similar to the subwoofer application above, except that we should have bandwidth of at least $f_b = 7$ kHz for decent voice recognition, as mentioned in section 9.2. This is fulfilled with all ADC modes except mode 4 . The performance of the speech recognizer, no matter how good, is of course also dependent on the quality of the ADC. But since no person will actually listen to the sampled signal, we can take means to reduce whatever noise is produced by the ADC without considering the subjective quality of the signal.

Since one is typically not using the voice recognition feature continuously, the remote device that is being spoken into can be designed so that the user actively engages and disengages voice recognition, for example with a single button. This means that the SoC can be put to sleep when it is not used which will significantly reduce power consumption compared to the other appliances.

A suggested configuration for this application is mode 3 ($f_s = 14.7$ kHz and $f_b = 7.35$ Hz) and $B_C = 4$ bits. According to table 9.1, this will produce a compressed bit rate of $R_C = 58.8$ kbps.

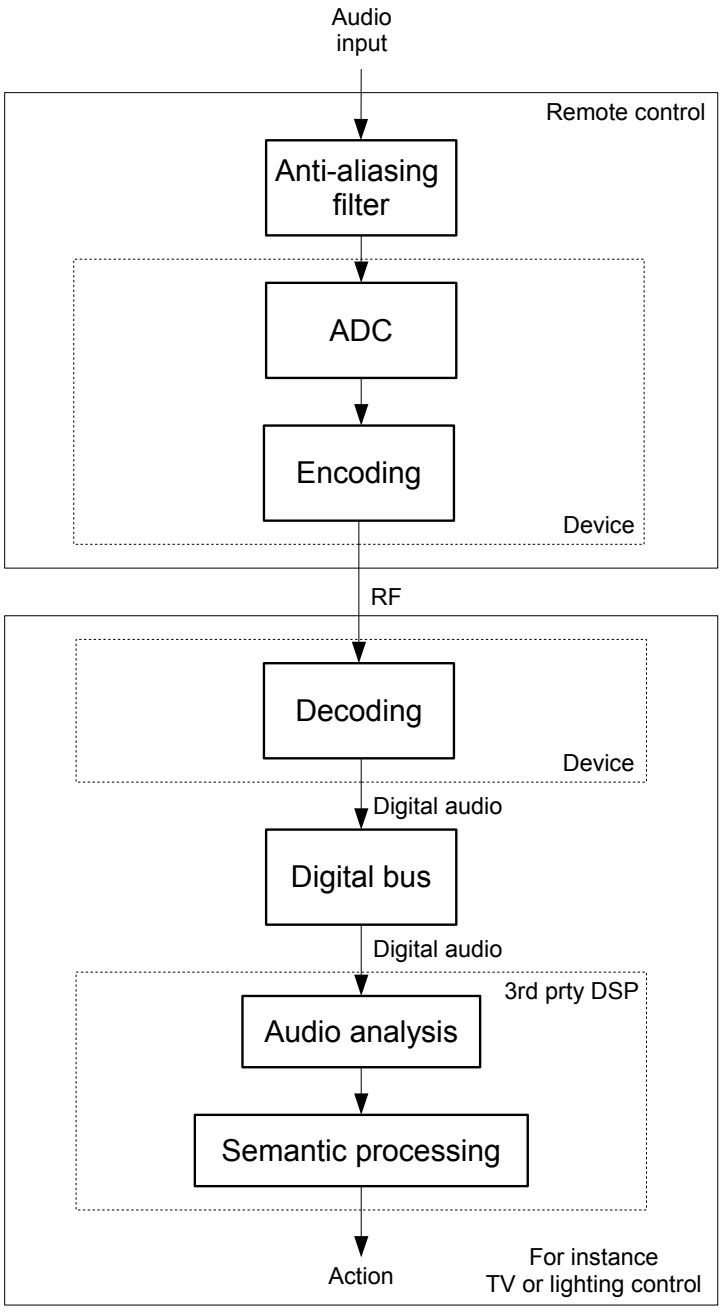


Figure 9.4: Voice recognition system.

# Chapter 10

# Conclusion

The SoC was proven to have the capacity to be used for point-to-point audio transmission in terms of processing power, memory and I/O capabilities. This chapter summarizes the results of this project.

Initially, the processing power and the ADC of the SoC was considered bottlenecks when designing an audio transmission system. The well-known ADPCM scheme was implemented, benchmarked and proven to run in real-time at sample rates up to about $f_s = 15$kHz in half-duplex. By inspecting the code we got a measure on what computational complexity the SoC was able to do for two of the four ADC modes ($f_s = 14.7$ and 28kHz). The ADC was tested in terms of THD+N and gave surprisingly good results considering that it is a general purpose converter that is not designed with audio sampling in mind. Noise reduction was briefly discussed and assessed as a means of reducing quantization noise and distortion from the ADC.

This project has been about finding new appliances for a device that was originally intended for something different - a typical engineer's problem. The SoC and its accompanying full-duplex protocol is designed for infrequent, low bit rate packet transmission like wireless PC keyboard strokes. The proprietary full-duplex protocol analyzed in [1] was found to be unable to carry out full-duplex real-time operation for any of the ADC modes' sample rates $f_s$. [1] also arrives at this conclusion and suggests several other full-duplex protocols that are designed for audio transmission, which means periodic and frequent high-rate packet transmission.

An low cost approach to performing analog output on-chip was successfully implemented by applying the pulse-width modulation principle. Since the ADC proved sufficient for several applications in terms of audio quality, and since everything else needed for point-to-point audio transmission was present on-chip, this result was the last piece of the puzzle for designing a fully self-contained audio communication system for the SoC. Without the

need for any external, third-party peripherals, production costs are severely reduced, which was the point of investigating this appliance for the SoC in the first place.

In addition to presenting an assessment of the SoC, this report was written as a guide for anyone who is in the business of doing a similar task for any embedded SoC. It explores the main theoretical and practical issues when designing an audio transmission system, and may also be generalized for numerous other signal transmission applications.

# Chapter 11

# Further work

This section will present and briefly discuss further work and potential ideas that could contribute to the project.

## 11.1 Synchronizing transmitter and receiver

As mentioned briefly in section 7.1, an issue with ADC and DAC clock synchronization will arise when they are situated on separate physical devices, no matter how similar in design these may be [21, ch. 5]. This is a common problem that occurs because of physical inaccuracies. Choosing a method for overcoming this is highly dependent on the coding scheme; if the transmitted information is straightforward PCM audio, discarding or filling in samples whenever the receiver buffer over- or underflows may suffice, but if more complex coding schemes is used, the loss of a sample or packet may be equally complex to disguise. The implementation that accompanies this project did not cover the subject of synchronizing transmitter and receiver. The ADC and the DAC were merely tuned to not fall out of sync immediately. For production-level quality, this issue needs to be addressed.

## 11.2 Code base optimized specifically for this SoC core

As mentioned earlier, table 8.2 in section 8.5 is a crude measure on what the SoC core can achieve for the given sample rates. Each architecture has its own pros and cons, and exploiting these is what makes a difference. All implementation, including Q16 operations, have been written as a proof of concept and has not been tailored to the SoC architecture. Therefore there

is likely to be room for improvement in computational complexity. Carrying out this implementation would produce a more accurate measure on what order of computations per sample this specific SoC can carry out for a given sample rate.

## 11.3 Full-duplex operation protocol

Full-duplex operation with the protocol investigated in [1] was proven to bve unsuitable for full-duplex audio transmission, mainly because it is tailored for reliable, loss-free spurious transmission of mouse and keyboard peripheral output. A protocol written for full-duplex audio transmission was considered a time consuming task since the radio is natively half-duplex. However, since half-duplex operation was proven to be effective enough for the applications discussed in this project, and since section 8.5 showed that bandwidths large enough to hold speech is attainable with full-duplex operation with respect to computational complexity, it is reasonable to assume that such a protocol could be implemented. [1] proposes several solutions for suchs protocols.

## 11.4 Joint source and channel coding

Source and channel coding have been investigated and designed separately for this project, the latter part being handled by another student [1]. The source-channel separation theorem [9][p. 187] states that source and channel coding can be optimal and independent if the data set is infinitely large. So if buffers were infinitely large, one need not consider any properties of the channel when designing the source coder. Obviously this is never the case, and investigations on how to jointly design source and channel coding has been investigated [23].

## 11.5 Other possible points for further work

- Error resilience, although presented and discussed, was not implemented. Tests on this with simple variants on channel conditions could then be carried out.
- Explore alternative audio processing algorithms for the applications discussed in chapter 9 and apply subjective listening tests for audio quality assessment in addition to the technical MSE measurement.

- Advance on noise reduction for minimizing ADC noise and distortion. This is a rather large field of study with numerous schemes which one can choose from.
- Look into echo cancellation to reduce potential feedback noise for full-duplex operation.

# Appendix A

# Levinson-Durbin recursion

The following is an implementation of Levinson-Durbin recursion. It is derived from [24]. ACF($s(n)$, $l$) calculates the autocorrelation function for the given signal $s(n)$ with lag $l$.

**Algorithm 2** Levinson-Durbin recursion

**for** $l = 0 \rightarrow P$ **do**
  $R_l = \text{ACF}(x(n), l)$
**end for**
$\alpha_1 = -\frac{R_1}{R_0}$
$E_1 = R_0 + R_1\alpha_1$
**for** $k = 1 \rightarrow P$ **do**
  $\lambda = \frac{-\sum_{j=0}^{k} \alpha_j R_{k+1-j}}{E_k}$

$$A_{k+1} = \begin{bmatrix} \alpha_0 \\ \vdots \\ \alpha_k \\ 0 \end{bmatrix} + \lambda \begin{bmatrix} 0 \\ \alpha_k \\ \vdots \\ \alpha_0 \end{bmatrix}$$

  $E_{k+1} = (1 - \lambda^2)E_k$
**end for**

# Appendix B

# M-tables for one-word-memory AQB

The following are suggested step size multipliers (M-tables) for DPCM signals with $B_C \in \{2, 3, 4\}$ [11].

| $B_C$ | Step size multipliers |
|---|---|
| 2 | 0.8, 1.6 |
| 3 | 0.9, 0.9, 1.25, 1.75 |
| 4 | 0.9, 0.9, 0.9, 0.9, 1.3, 1.6, 2.0, 2.4 |

# Appendix C

# Complex ADPCM

This appendix will describe a more complex AQB algorithm than the one-word-memory AQB described in section 2.4. It was originally presented in [5, ch. 4]. The following method calculates the step size $\Delta(n)$ as proportional to the sample standard deviation $\sigma_s(n)$ of any input signal $s(n)$ with a scaling factor $\eta$:

$$\Delta(n) = \eta\sigma_s(n) \tag{C.1}$$

A good choice for $\eta$ is

$$\eta = \frac{\Delta}{\sigma_s} \tag{C.2}$$

where $\Delta$ is the step size of the non-adaptive uniform quantizer and $\sigma_s$ is the long-term standard deviation of $s(n)$. Equation C.1 derived in [5, ch. 4], and can be used as a basis for a iterative adaptive algorithm as follows. Introducing $\beta \in \{0, 1\}$ as factor for how fast the step size will adapt, we can estimate $\sigma_s$ recursively with $\hat{\sigma}_s$:

$$\hat{\sigma}_s^2(n) = \beta\hat{\sigma}_s^2(n-1) + (1-\beta)s^2(n-1) \tag{C.3}$$

This idea behind this is to low-pass filter the changes input variance with $\beta$ as a coefficient for adaption speed [5, ch. 6.4]. Combining equations C.3 and C.1, we get a recursive formula for $\Delta(n)$:

$$
\begin{aligned}
\Delta(n) &= \eta\hat{\sigma}_s \\
&= \eta\sqrt{\beta\hat{\sigma_s}^2(n-1) + (1-\beta)s^2(n-1)} \\
&= \eta\hat{\sigma}_s(n-1)\sqrt{\beta + (1-\beta)\frac{\eta^2 s^2(n-1)}{\eta^2\hat{\sigma_s}^2(n-1)}} \\
&= \Delta(n-1)\sqrt{\beta + (1-\beta)\eta^2\left(\frac{s(n-1)}{\Delta(n-1)}\right)^2}
\end{aligned}
\tag{C.4}
$$

Equation C.4 is fairly intuitive. An increase in amplitude when moving from $s(n-1)$ to $s(n)$ increases $\Delta(n)$ from $\Delta(n-1)$ and vice versa, with $\beta$ controlling the abruptness of the change over time.

# Appendix D

# Algorithms for multiplication and division of Q15.16 numbers

Multiplication and division of Q$i.f$-numbers $X$ and $Y$ represented by $B_A$ bits can be carried out as in algorithm 3 and 4, respectively [13].

**Algorithm 3** Q$i.f$ multiplication

Allocate $2B_A$ bits for $XY_{ii}$, $XY_{if}$, $XY_{ff}$ and $XY_{result}$
$XY_{sign} \leftarrow$ resulting sign of $X \cdot Y$
$X_i \leftarrow$ integer part of $X$ and $Y_i \leftarrow$ integer part of $Y$
$X_i \leftarrow$ fractional part of $X$ and $Y_i \leftarrow$ fractional part of $Y$
$XY_{ii} \leftarrow X_i \cdot Y_i$ {Integer only part.}
$XY_{if} \leftarrow X_i \cdot Y_f + X_f \cdot Y_i$ {Middle part.}
$XY_{ff} \leftarrow X_f \cdot Y_f$ {Fractional only part.}
$XY_{ii} \leftarrow XY_{ii} \cdot 2^{-f}$ {Prescale before adding it all up.}
$XY_{ff} \leftarrow XY_{ff} \cdot 2^{f}$ {Prescale before adding it all up.}
$XY_{result} \leftarrow XY_{ii} + XY_{if} + XY_{ff}$
$XY_{result} \leftarrow$ truncate $XY_{result}$ to $B_A$ bits
$XY_{result} \leftarrow XY_{result} \cdot XY_{sign}$

**Algorithm 4** $Qi.f$ division

$XY_{sign} \leftarrow$ resulting sign of $\frac{X}{Y}$
Allocate $2B_A$ bits for $XY_{result}$
Allocate $B_A$ bits for remainder $XY_r$ and divisor $XY_d$
$XY_r \leftarrow X \cdot 2^{-B_A/2}$ {Initialize $XY_r$ witt prescaled $X$.}
$XY_d \leftarrow Y \cdot 2^{B_A/2}$ {Initialize $XY_d$ witt prescaled $Y$.}
{Calculate integer part bit-by-bit.}
**for** $n = 0 \rightarrow B_A$ **do**
  $XY_d \leftarrow XY_d \cdot 2$
  **if** $XY_r \geq XY_d$ **then**
    $XY_r \leftarrow XY_r - XY_d$
    $XY_a \leftarrow XY_a + 2^{B_A-1-n}$
  **end if**
**end for**
{Calculate fractional part bit-by-bit.}
**for** $n = 0 \rightarrow B_A$ **do**
  $XY_r \leftarrow XY_r \cdot 2$
  **if** $XY_r \geq XY_d$ **then**
    $XY_r \leftarrow XY_r - XY_d$
    $XY_a \leftarrow XY_a + 2^{B_A/2-1-n}$
  **end if**
**end for**
$XY_{result} \leftarrow XY_{result} \cdot XY_{sign}$

# Appendix E

# Code references

This appendix lists the files modules of the C implementation and MATLAB simulation. The files are located in the */code* folder in the ZIP file accompanying this report.

## E.1 Octave/MATLAB code

*acf.m* - Calculates one-sided autocorrelation function.
*adc_analysis_script.m* - Calculates ENOB and SINAD from ADC tests.
*caqb_calc_const.m* - Script for calculating complex AQB constants offline for faster C implementation.
*caqb_decoder.m* - Complex AQB decoder.
*caqb_delta_calc.m* - Complex AQB step size calculation.
*caqb_encoder.m* - Complex AQB encoder.
*caqb_test_script.m* - Complex AQB test script.
*double2Q16.m* - Converts 64-bit floating point number to Q15.16.
*dpcm_dec.m* - DPCM decoder.
*dpcm_enc_q.m* - DPCM encoder.
*double2Q16.m* - 64-bit floating point to 32-bit Q16 converter.
*draw_adc_noise_psd.m* - Script for calculating and drawing PSD estimation from ADC test results.
*draw_companding_curve.m* - Script for drawing $\mu$-law companding curve.
*draw_error_concealment.m* - Script for drawing examples of error concealment.
*draw_nr.m* - Script for drawing example figure for noise reduction.
*draw_pwm_ex.m* - Script for drawing PWM example in chapter 6.
*ENOB.m* - ENOB calculator.
*expander.m* - Noise gate.

*gen_sines_for_adc_test.m* - Script that generates sines for ADC test.
*lpc_direct.m* - LPC calculation by solving linear equations.
*maqb_decoder.m* - One-word-memory AQB decoder.
*maqb_delta_calc.m* - One-word-memory AQB step size calculation.
*maqb_encoder.m* - One-word-memory AQB encoder.
*maqb_test_script.m* - One-word-memory AQB test script.
*midrise_q.m* - Midrise quantizer.
*Q162double.m* - Converts Q15.16 to 64-bit floating point number.

## E.2 C x86 code (Q16 and float)

*adpcm.c/.h* - ADPCM encoder and decoder, not including LPC calculation.
*caqb.c/.h* - Complex AQB.
*coding.c/.h* - All aspects of encoding and decoding.
*config.h* - System configuration.
*expander.c/.h* - Noise gate.
*l-d.c/.h* - Levinson-Durbin algorithm. Rewritten from [24].
*maqb.c/.h* - One-word-memory AQB.
*maths.c/.h* - Custom math operations.
*midrise.c/.h* - Midrise quantizer.
*Q16.c/.h* - Q15.16 fixed-point type.
*system.c/.h* - System simulation for scenario 1. Analogous to *system_sim_1.m*.
*tester.c/.h* - Testing "framework" for all C code.
*Makefile* - Makefile for all C code.

## E.3 C SoC code

*adc.c/.h* - ADC driver and buffer logic.
*adpcm.c/.h* - ADPCM encoder and decoder, not including LPC calculation.
*config.h* - System configuration.
*dac.c/.h* - DAC driver and buffer logic.
*gazell.c/.h* - Gazell protocol.
*l-d.c/.h* - Levinson-Durbin algorithm. Rewritten from [24].
*main.c/.h* - Entry point.
*maqb.c/.h* - One-word-memory AQB.
*maths.c/.h* - Custom math operations.
*Q16.c/.h* - Q15.16 fixed-point type.

*radio_config.c/.h* - Radio configuration.
*rx.c/.h* - RX logic.
*tx.c/.h* - TX logic.
*uart.c/.h* - UART driver for testing.

# Appendix F

# Audio file references

This appendix lists folders with audio files that were used for performance tests thoughout the project. The files are located in the */audio* folder in the ZIP file accompanying this report. All files are named by content, $B_A$ and $f_s$.

## F.1 ADC test sources

*door_*[$f_s$]*_*[$B_A$]*bit.wav* - Squeaking door.
*lydbok_1_*[$f_s$]*_*[$B_A$]*bit.wav* - Norwegian speech #1.
*lydbok_2_*[$f_s$]*_*[$B_A$]*bit.wav* - Norwegian speech #2.
*prime_sines_1_*[$f_s$]*_*[$B_A$]*bit.wav* - Pure sines. #1.
*sub_*[$f_s$]*_*[$B_A$]*bit.wav* - Subwoofer.

## F.2 ADC test results

*door_*[$f_s$]*_*[$B_A$]*bit.wav* - Squeaking door.
*lydbok_1_*[$f_s$]*_*[$B_A$]*bit.wav* - Norwegian speech #1.
*lydbok_2_*[$f_s$]*_*[$B_A$]*bit.wav* - Norwegian speech #2.
*prime_sines_1_*[$f_s$]*_*[$B_A$]*bit.wav* - Pure sines. #1.
*sub_*[$f_s$]*_*[$B_A$]*bit.wav* - Subwoofer.
*n+d_*[$f_s$]*_*[$B_A$]*bit.wav* - ADC noise+distortion.

## F.3 Performance test sources

*lydbok_1_*$[f_s]$*_*$[B_A]$*bit.wav* - Norwegian speech #1.

## F.4 Performance test results

These files are named by the following conventions.

CAQB - ADPCM with complex AQB was used.
DBITS$[B_C]$ - $B_C$ bits.
DOUBLE - MATLAB 64-bit floating point was used instead of 32-bit Q16.
DPCM - DPCM was used.
MAQB - ADPCM with one-word memory AQB was used.
NR - Noise reduction was used.
NOCOMP - Compression was bypassed.

# Bibliography


[1] J. M. Lloveras. Point to point wireless audio with limited bandwidth and processing (report). 2012. Channel coding and RF Part of the same project that is described in this report.

[2] E. S. Brenden. Point to point wireless audio with limited bandwidth and processing (report). 2011. A previous project on this device, focusing on investigating what algoithms will run real-time on the device.

[3] Nordic Semiconductor. nRF51 SOS. Datasheet for the SoC (article).

[4] C. F. Goss. Machine code optimization - improving executable object code (article). 1986.

[5] N. S. Jayant and P. Noll. *Digital Coding of Waveforms*. Prentice Hall, 1984.

[6] V. Atti A. Spanias, T. Painter. *Audio Signal Processing and Coding*. John Wiley and Sons, Inc., first edition, 2007.

[7] R. Brandtsegg, 2012. Personal communication with second supervisor Rune Brandtsegg at Nordic Semiconductor.

[8] D. K. Manolakis J. G. Proakis. *Digital Signal Processing - Principles, Algorithms and Applications*. Pearson Prentice Hall, fourth edition, 2007.

[9] J. A. Thomas T. M. Cover. *Elements of Information Theory*. John Wiley and Sons, second edition, 2006.

[10] L. Lundheim, 2012. Personal communication with supervisor Lars Lundheim.

[11] N. S. Jayant. Adaptive quantization with a one-word memory (article). 1973.

[12] L. M. Bishop J. M. Van Verth. *Essential Mathematics for Games and Interactive Applications - A Programmer's Guide*. Morgan Kaufmann Publishers, second edition, 2008.

[13] J. Lauha. The neglected art of fixed point arithmetic (article). Seminar for ASSEMBLY event in Helsinki the summer of 2006.

[14] M. Talbot-Smith. *Sound Engineering Explained*. Focal Press, second edition, 2002.

[15] D. Lin B. W. Wah, X. Su. A survey of error-concealment schemes for real-time audio and video transmissions over the internet (article). 1986.

[16] A. Schaefer. The Effective Number of Bits (ENOB) of my R and S Digital Oscilloscope (article). 2011.

[17] W. Kester. Understand SINAD, ENOB, SNR, THD, THD+N, and SFDR so You Don't Get Lost in the Noise Floor (article). 2009.

[18] P. A. Wheeler T. D. Rossing, F. R. Moore. *The Science of Sound*. Addison Wesley Longmain, third edition, 2002.

[19] J. Rodman. The effect of bandwidth on speech intelligibility (article). 2006.

[20] R. Gentile J. MD. J. Katz. *Embedded Media Processing*. Elsevier, first edition, 2006.

[21] S. Fechtel H. Meyr, M. Moeneclaey. *Digital Communication Receivers: Synchronization, Channel Estimation, and Signal Processing*. John Wiley and Sons, Inc., 1998.

[22] A. Wellings A. Burns. *Real-Time Systems and Programming Languages*. Addison Wesley Longmain, fourth edition, 2009.

[23] F. Hekland. Review of joint source-channel coding (article). 2004.

[24] J. Degener. lpc.c, 1994. Levinson-Durbin C implemenation (C code).