Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Sondre Ninive Andersen

Implementation of Delta-Sigma ADCs using the Microchip AVR-DB family of microcontrollers

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth, Martin Thomaz
June 2020

NTNU
Norwegian University of Science and Technology

MICROCHIP

Sondre Ninive Andersen

Implementation of Delta-Sigma ADCs using the Microchip AVR-DB family of microcontrollers

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth, Martin Thomaz
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Problem Description

The AVR-DB family of microcontrollers from Microchip, includes user-configurable op-amps as peripherals. This opens the door to creating many analog processing circuits, without active external components. One possible use case for this is analog-to-digital conversion, by implementing a Delta-Sigma ADC, which is not currently available in any AVR microcontroller. While the AVR-DB does include an ADC, it is optimized for general-purpose use, with a trade-off between resolution, sampling speed, power-consumption and other factors. A Delta-Sigma ADC might be able to outperform the existing ADC in resolution, by sacrificing other metrics.

The project consists of:

- Implementing a working Delta-Sigma ADC, using the op-amps available in the AVR-DB family MCUs
- Characterizing this ADC with regards to accuracy and speed, and comparing it to the on-chip SAR ADC
- Evaluating the advantages and disadvantages of the implementation

Abstract

AVR® MCU DB is a new family of microcontrollers from Microchip, which includes user-configurable op-amps as peripherals. These op-amps makes it possible to implement moderately complex op-amp circuits, without external active components, which opens the door for low-cost and small-size applications involving analog signal processing.

This project uses these op-amps, together with other available peripherals, to implement an integrating analog-to-digital converter (ADC), in pursuit of a higher-resolution alternative to the 12-bit on-chip ADC. Three variations of a Delta-Sigma ADC were implemented, as well as a dual-slope type ADC. These were then characterized to measure the linearity, and effective resolution.

Among the implementations, the best performing is a first-order Delta-Sigma type ADC, with a moving-average digital filter, implemented in a core-independent way, using timer/counter peripherals. This ADC achieved close to a 12-bit usable measurement. While this only matches the on-chip ADC, there are opportunities for improvement. Additionally, the technique could also be viable with other, lower-cost, microcontrollers, where a on-chip might be lower resolution, or absent.

Sammendrag

AVR® MCU DB er en ny mikrokontrollerfamilie, som inkludere bruker-konfigurerbare op-amper som periferifunksjoner. Disse op-ampene gjør det mulig å implementere op-amp kretser med moderat kompleksitet, uten eksterne aktive komponenter. Dette åpner muligheten for analog signalprosessering i applikasjoner med lav kostnad og liten størrelse.

I dette prosjektet benyttes disse op-ampene, sammen med andre tilgjengelige periferifunksjoner, til å implementere en integrerende AD-omsetter (ADC), for å forsøke å oppnå et alternativ til den integrerte 12-bit ADCen, med høyere oppløsning. Det ble implementert tre varianter av en Delta-Sigma ADC, og en dual-slope type ADC. Alle implementasjonene ble så karakterisert, for å måle linearitet og effektiv oppløsning.

Det beste ytelsen ble oppnådd med en førsteordens Delta-Sigma ADC med et digitalt glidende gjennomsnittsfilter, implementert uavhengig av CPUen i mikrokontrolleren. Denne ADCen oppnådde en nær 12-bit brukbar måling. Selv om dette bare tangerer den integrerte ADCens ytelse, er det muligheter for forbedring. I tillegg kan teknikken muligens også benyttes med andre mikrokontrollere, i situasjoner uten en integrert ADC, eller hvor denne har lavere oppløsning.

Table of Contents

Abbreviations

AC	=	Analog Comparator
ADC	=	Analog-to-Digital Converter
CCL	=	Configurable Custom Logic
CPU	=	Central Processing Unit
DAC	=	Digital-to-Analog Converter
DIP	=	Dual In-Line Package
DUT	=	Device Under Test
$\Delta\Sigma$	=	Delta Sigma
DSP	=	Digital Signal Processing
EVSYS	=	Event System Controller Peripheral
GPIO	=	General Purpose Input/Output
ISR	=	Interrupt Service Routine
LUT	=	Lookup Table
MISO	=	Master-In Slave-Out
MOSI	=	Master-Out Slave-In
MCU	=	Microcontroller Unit
OPAMP	=	Op-amp System Peripheral
PCB	=	Printed Circuit Board
SAR	=	Successive Approximation Register
SPI	=	Serial Peripheral Interface
TC	=	Timer/Counter
TCA	=	Timer/Counter Type A
TCB	=	Timer/Counter Type B
TCD	=	Timer/Counter Type D
TQFP	=	Thin Quad Flat Pack

Chapter 1

Introduction

The modern approach to signal processing typically relies on the use of the extensive toolkit of digital signal processing (DSP). However, in order to use these tools to process data from the physical world, it is necessary to convert the analog signals of the real world into digital signals, thus setting the requirement for high-performance analog-to-digital converters (ADCs). In the world of embedded systems, microcontrollers (MCUs) have long been used for varied tasks, and since MCUs normally include an ADC as an integrated peripheral, they are able to do both the analog-to-digital conversion, and the digital signal processing, in the one component.

Microchip's AVR® MCU DB (AVR-DB) family of microcontrollers features a 12-bit Successive Approximation Register (SAR) ADC [10]. This type of ADC is good for general purpose applications, offering a reasonable tradeoff between resolution, sampling rate, power requirement, and complexity. For some applications, however, this trade-off does not accurately reflect the prevailing priorities. For measuring thermal sensors, for example, sampling rate might not be as important as high resolution [5]. In these cases, a different approach might be worth exploring.

The AVR-DB family of microcontrollers also includes user-configurable operational amplifiers, or op-amps, as a peripheral. This allows the MCU to also do simple analog signal processing, with minimal external components. This addition, along with other common peripherals, makes it possible to implement alternative types of ADC, allowing trade-offs customized to the application.

There are many different types of ADCs, with different trade-offs. While SAR ADCs are a good general-purpose choice, for applications where resolution is the most important, Delta-Sigma ($\Delta\Sigma$) type ADCs are likely the best choice [11]. This project focuses on implementing a $\Delta\Sigma$-ADC, using no active external components, in an attempt to improve the resolution of the conversion result.

At the time of writing, the AVR-DB family has not yet been released. Throughout this project, information about the microcontrollers have been taken from the preliminary datasheet. Once released, the datasheet will be available from Microchip's website [10].

Collaboration with Microchip Technology Inc.

This project was a collaboration with Microchip Technology Inc. The microcontrollers used for development and characterization were engineering samples provided by Microchip, and the testing was carried out using lab-equipment from Microchip's Trondheim office.

Chapter 2

Background

2.1 Analog-to-Digital Converter Types

There exists a large variety of types of ADCs, which have different trade-offs between conversion speed, resolution, power consumption, and other performance metrics. This project mainly focuses on the Delta-Sigma type of ADC, but also considers a dual-slope implementation. The following section gives an overview of the operation of these two ADC types.

2.1.1 Delta-Sigma ADC

A delta-sigma ($\Delta\Sigma$) ADC is an oversampling converter. This means that it samples the input at a higher rate than the rate required by the Nyquist–Shannon sampling theorem, and trades away this excessive sampling rate for higher resolution. At the highest level, a $\Delta\Sigma$ ADC consists of a delta-sigma modulator feeding a low-pass filter. The low-pass filter is digital, while the modulator is implemented with an analog circuit. A benefit of this design is that the analog circuitry can tolerate quite imprecise components, without affecting the ADCs performance.

The $\Delta\Sigma$ modulator is a simple feed-back loop consisting of an integrator, an ADC, a sample and hold, and a DAC. shown in Figure 2.1. The sample and hold is clocked at the *sample rate*, f_s, which is distinct from the *data rate*, f_d. The quantizer is often as simple as an analog comparator, effectively a 1-bit ADC. The presence of this quantizer makes the sample and hold simple to implement, as it can be a purely digital component, often just a 1-bit D latch. The integrator will integrate the sum of the input signal, and the feedback term from the sample and hold [3, .p 922].

In the time-domain, the function of a 1-bit modulator can be understood as follows: If the output is low, while the input is at some positive voltage, v_{in}, the integrator will start rising at a rate proportional to v_{in}. For each clock edge, the quantizer will see if the integrator has reached the $\frac{V_{ref}}{2}$, and at some point, it will be above this threshold, and the output will go high. This will likely happen a bit after the integrator passed the threshold,

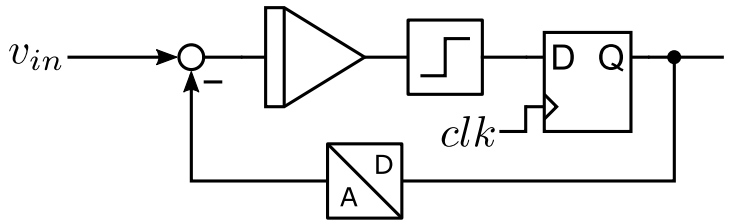


Figure 2.1: The basic block diagram for a $\Delta\Sigma$ modulator.

since the output is only updated on each clock edge. With the output high, the integrator will start falling, now with a downwards rate proportional to $V_{ref} - v_{in}$. After some number of clock cycles, the integrator output is again below $\frac{V_{ref}}{2}$, and the output goes low again.

The result is that the high-time and low-time of the integrator is determined by the rates Kv_{in} and $K(V_{ref} - v_{in})$, where K is some constant of the integrator, along with how much the value over- or undershoots the threshold each time. Since the amounts of over- and undershoot can be thought of as roughly randomly distributed, the average duty-cycle of the output will be proportional to v_{in}.

The output from the modulator can be interpreted in a few different ways. It can be thought of as an *encoding*, with the analog input voltage represented by the duty-cycle of the signal, but it can also be interpreted as an analog signal itself. With this interpretation, it can be modeled as a copy of the analog input voltage, mixed with a large amount of high-frequency quantization noise. This is exploited by the converter, as a low-pass filter is all that is needed to obtain the original signal. However, since the modulator-output can only have one of two voltages, and the edges are locked to the modulator clock, the low-pass filter can be implemented digitally, leveraging the power of DSP.

2.1.2 Dual-Slope ADC

One of the simplest types of integrating ADCs is the dual-slope ADC. The operation of this type of converter can be described as follows. First, v_{in} is multiplexed to the input of the integrator, and this voltage is integrated over some predetermined amount of time, Δt_1. This results in a voltage on the integrator proportional to the input voltage. Next, a known reference voltage of opposite polarity to v_{in} is applied to the integrator input, causing the output to fall back towards the starting point. An analog comparator triggers once this second slope is completed, and a timer captures the time, Δt_2, this took. Since the change in voltage for the second slope is proportional to v_{in}, and the rate is constant, Δt_2 is also proportional to v_{in} [3, .p 914].

Setting the equation for each slope equal to each other, with K being the constant of integration, results in the following equation

$$v_{max} = K\Delta t_1 v_{in} = K\Delta t_2 v_{ref},$$

which gives the proportional relationship between Δt_2 and v_{in} as

$$v_{in} = v_{ref} \frac{\Delta t_2}{\Delta t_1}.$$

One of the benefits of this method, is that the constant of integration, as well as v_{max}, is canceled out in the calculations, meaning that the accuracy of the conversion is not affected by such things as drift in the capacitance of the integrating capacitor. Also, both Δt_1 and Δt_2 would be measured as a certain number of clock cycles, but since only the ratio between these times are interesting, any drift in the clock frequency is also canceled out.

A weakness in this design is that it depends on the linearity of the integrator, as any non-linearity will be directly coupled to the ADC transfer function.

2.2 ADC Performance Metrics

In order to evaluate the performance of the ADCs implemented during this project, it is necessary to define some metrics. Among the most common, are offset and gain error, integral non-linearity (INL), differential non-linearity (DNL) [8], and the amount of input-referred noise.

2.2.1 Offset and Gain Error

An n-bit ADC has some full range of inputs, $V_{ref,n}$ to $V_{ref,p}$, and a range of output codes from 0 to $2^n - 1$. In the ideal ADC, an input of $V_{ref,n}$ produces a 0 output code, while the input $V_{ref,p}$ produces an output of $2^n - 1$, and each step in the transfer function is the same size, one least-significant-bit (LSB), which equals

$$\frac{V_{ref,p} - V_{ref,n}}{2^n - 1}.$$

The first and last steps are exceptions, and should be half the size of the ordinary steps, meaning that the first and last transitions should occur at $V_{in} = V_{ref,n} + 0.5\,\text{LSB}$ and $V_{in} = V_{ref,p} - 0.5\,\text{LSB}$ respectively.

Offset error refers to the deviation of the transition between output codes 0 and 1, from its ideal location. A positive offset error means that this transition happens at a lower voltage, meaning that the output of the ADC is generally too high, while a negative offset error means that the transition happens at a too high input voltage, resulting in the ADC output being generally too low.

If the offset voltage is corrected for, so that the 0-to-1 transition is at the right location, the last transition might not be in the correct location. This is called the gain-error, and is a measure of the error in the average step-size of the transfer function. It is measured as the deviation of the final transition of the ADC, from its ideal location, after the offset error has been corrected. Positive gain error means that this transition happens for a too low input voltage, while negative gain error means that the transition happens for a too high input voltage.

Both of these errors can generally be calibrated for, either by trimming the analog circuitry, or in software.

2.2.2 Non-Linearity

Even if both the first and last transition have been calibrated to be in the correct locations, there might be errors in the location of the remaining transitions. This represents a deviation from the ideal straight-line response of the ADC, also called nonlinearity.

There are two main measures of non-linearity: differential and integral. The differential non-linearity is a measure of the size of the steps in the transfer function, while the integral non-linearity measures the maximum deviation of the output code from the ideal straight line [9].

2.2.3 Input-Referred Noise

Input-referred noise refers to the noise present in the ADC, modeled as an analog noise-source on the input, when it repeatedly converts a stable input voltage. This noise is usually modeled as a Gaussian distribution, centered at the ideal output code. In some cases, a small amount of input-referred noise is actually desirable, as it enables additional resolution to be obtained through averaging multiple samples. Without any noise, multiple samples would, of course, give the same value, rendering averaging pointless.

An interesting metric which can be derived from the input-referred noise is the *noise-free code resolution*. This is the number of bits of the result which are stable, and not influenced by the input-referred noise, and represents the maximum accuracy of the ADC, if no extra processing is done [4].

2.3 Op-amp Integrator Circuit

The main idea for this project is to implement the $\Delta\Sigma$ modulator using the peripherals available in the microcontroller, and any required external components. As described in section 2.1.1 this requires an integrator, a comparator, some sample-and-hold functionality, and a 1-bit digital-to-analog converter (DAC). Most of these can be implemented using only the peripherals in the MCU, but the integrator requires a few external passive components.

The core of all the ADC implementations in this project, is the op-amp integrator. This circuit, shown in Figure 2.2, will integrate the input, with the time-constant being determined by the values of the capacitor and input resistors [3, p. 230]. Intuitively, the circuit can be understood by considering that the negative feedback-action of the op-amp will keep the voltage at the negating op-amp input equal to V_{ref}. This means that the current through each resistor is proportional to the input voltage (relative to V_{ref}), and this current must continue through the capacitor, charging it over time. In Figure 2.2, the circuit is drawn with two inputs, but in general, any number of inputs can be used, each with its own scaling, set by the input resistors.

In more rigorous terms, assuming the simplest model of an op-amp with infinite gain, the equation for the output can be derived from the equation for a capacitor:

$$v_{out}(t) = V_{ref} - v_c = V_{ref} - \frac{1}{C}\int i_c dt = V_{ref} - \frac{1}{C}\int \sum \frac{v_i - V_{ref}}{R_i} dt \qquad (2.1)$$

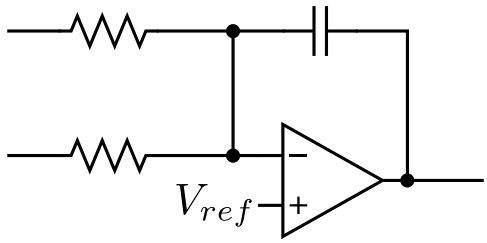


Figure 2.2: Circuit diagram for the op-amp implementation of a summing integrator.

For the circuit show in Figure 2.2, with both resistors having resistance R, and the capacitor having capacitance C, this equation simplifies to:

$$v_{out}(t) - V_{ref} = -\frac{1}{RC}\int (v_1 - V_{ref}) + (v_2 - V_{ref})dt \tag{2.2}$$

This equation shows a couple of important properties of this circuit. First, that it is inverting. This is important to remember as the circuit is used as a building block in more complex circuits. Second, the signal levels are all referred to V_{ref}. This means that V_{ref} can be set to any suitable voltage, and the circuit will behave identically around this voltage level.

One thing that is not modelled by this equation is the effects of the limited voltage rails. In microcontrollers, the voltages are typically limited to the range from around $0\,\mathrm{V}$, to around V_{DD}. This means that if the integrator moves too far in either direction, it will saturate close to one of these limits. Another problem with the limited voltage range comes when we want to supply an input to the integrator using the microcontroller itself. Since the inputs are referred to V_{ref}, and outputs from the microcontroller is limited to the $0\,\mathrm{V}$ to V_{DD} range, the range of input-signals is limited to $-V_{ref}$ to $V_{DD} - V_{ref}$. Which means that if V_{ref} were to be connected to $0\,\mathrm{V}$, as is typical for op-amp integrators, the microcontroller outputs would be unable to apply negative signals to the integrator. To mitigate this, and be able to control the integrator in both directions, V_{ref} will be set at $\frac{V_{DD}}{2}$ for the circuits used in this project.

2.4 Target Architecture

The target architecture is the AVR-DB family of MCUs from Microchip [10]. This family includes several variants of microcontrollers, but these differ only in pin count, memory-sizes, and the number of instances of various peripherals. Table 2.1 shows selected differences between the variants. All the ADCs described in chapter 4 are compatible with any of these, and the main development work was done using the 28-pin dual in-line package (DIP) variant. In order to minimize parasitic effects such as lead-inductance, the 48-pin thin quad flat pack (TQFP) was used for the characterization in chapter 5.

Table 2.1: Selected parameters of the available parts in the AVR-DB family of MCUs [10].

Part No.	Pin count	Op-amps	CCL LUTs
AVR128DB28	28	2	4
AVR128DB32	32	2	4
AVR128DB48	48	3	6
AVR128DB64	64	3	6

2.4.1 On-Chip ADC

The microcontroller which was used for this project already contains an on-chip ADC, and the $\Delta\Sigma$ ADC is compared to this. Since the specifications of this on-chip ADC is still being characterized at the time of writing, exact numbers are not available, but this section outlines the general specifications of this ADC, to provide a benchmark for comparison.

The on-chip ADC is a SAR ADC, with 12-bit resolution, and a maximum sampling rate of $100\,\mathrm{kHz}$, which is equivalent to a single conversion taking $10\,\mu\mathrm{s}$. The conversion can be configured to be either single-ended or differential, and reference voltage is selectable between internal or external references, or V_{DD}.

While only a single instance of the ADC is available in the MCU, the input to this can be multiplexed between 22 different inputs, including one channel internally connected to an internal temperature sensor. Additionally, the ADC peripheral implements several extra features such as window-compare interrupts, automatic accumulation of samples, and free-running and event triggered conversion.

2.5 Peripherals

The AVR DB includes a wide range of peripheral modules. Most are available in similar forms in earlier MCUs from Microchip, but the OPAMP peripheral is new to this family. This project was initiated as an idea for using this OPAMP peripheral for extending the analog-to-digital conversion capabilities of the microcontroller, so it is a central peripheral for the project. There are several other peripherals, however, that were used in the implementations described. The relevant peripherals are listed below, and these are described in the following sections.

- Operational Amplifier System (OPAMP)
- Event System (EVSYS)
- Timer/Counter type A and B (TCA and TCB)
- Custom Configurable Logic (CCL)
- Analog Comparator (AC)
- Serial Peripheral Interface (SPI)

2.5.1 Operational Amplifier System (OPAMP)

The OPAMP peripheral consists of up to three operational amplifiers, each with a configurable resistor-ladder, and configurable input multiplexers. These op-amps can be internally connected into many standard op-amp circuits, or connected to external pins.

The 48- and 64- pin variants of AVR-DB includes all three op-amps, while the lower pin count variants only contain two. For the setups described in chapter 4, the most important circuit configuration is the summing integrator configuration, shown in Figure 2.2.

Since the op-amp peripheral includes a configurable resistor-ladder, it is attractive to try to use these resistors in the integrator circuit, in order to avoid having to add them externally. However, a limitation in the peripheral is that the inputs to the op-amp can only *either* be connected to the internal resistor ladder, *or* to the pad, allowing connections to external components. This means that if it is connected to the resistor-ladder, there is no way to connect the capacitor, so external resistors are required.

The op-amps in the peripheral are powered from the same power-supply as the MCU, which is a single-rail supply. However, the non-inverting input of each op-amp can be internally connected to a $\frac{V_{DD}}{2}$ reference. This effectively allows the circuit shown in Figure 2.2 to function with positive and negative signal values, but with a $\frac{V_{DD}}{2}$ offset.

2.5.2 Event System (EVSYS)

The event system (EVSYS) can be used to directly connect peripherals, and make them work together without intervention by the Central Processing Unit (CPU). The system consists of ten event-channels, each of which can be assigned to any of the event-generators in the MCU. Any General Purpose I/O (GPIO) pin can be assigned to a channel, and most of the peripherals provide some events. Peripherals can also be configured to listen to any of the channels, and trigger certain actions based on the event. Event actions can either be edge-triggered, or they can be level-triggered. For example, the ADC can be configured to start a conversion on the rising edge of an event, or TCA can be configured to run as long as an event input is high.

The following descriptions of peripherals includes more examples of what can be connected to the event system, both as event producers and users.

2.5.3 Timer/Counters

A Timer/Counter (TC) is a peripheral that provides several different functionalities, based around a simple binary counter. If the counter is clocked from a periodic clock, such as the system clock or a prescaled version of it, or an external clock, it is called a *timer*. If the timer is clocked by a sporadic signal, such as an external input, or an internal event, it is called a *counter*.

As a *timer*, a TC can provide periodic interrupts with cycle-accurate timing, it can generate a pulse-width modulated signal on an output pin, it can generate a variable-frequency square-wave, or it can be stopped by an internal or external trigger, functioning as a time-to-digital converter. As a *counter*, it can count internal or external events, and provide a running tally for the CPU to read, or give an interrupt when a predetermined number of events have occurred.

In AVR-DB, several variants of TCs are available. For this project, types A and B are used. Type A (TCA) is a 16-bit counter, intended to be an as flexible as possible general-purpose counter. Type B (TCB) is also a 16-bit counter, but with more features for input-capture, such as counting or timestamping external events.

Many of the basic functions however, such as providing a periodic interrupt or event signal, can be implemented with both types of TCs. This is relevant in the lower pin count variants of AVR-DB, where only a limited number of instances of each type are available.

2.5.4 Custom Configurable Logic (CCL)

The CCL is a peripheral that can function as "glue logic" between the MCU and external components, or between peripherals in the MCU itself. The CCL consists of up to six look-up tables (LUTs), and a sequential block for each pair of LUTs, which can be configured as one of several kinds of flip-flop. Each LUT has a single output, and three inputs which can be connected to many different internal signals, or to external pins. The peripheral has additional functionality for synchronizing asynchronous signals, filtering short glitches, and edge-detection, but none of these features are used in this project. These features do however require a clock, which can be selected between multiple sources, including one of the inputs of the LUT itself.

Each sequential block takes one input each from the even- and odd-numbered LUTs, as well as the selected clock signal from the even-numbered LUT. Depending on the configuration, the block then behaves as one of several flip-flop types. The available configurations are a gated D flip-flop, a JK flip-flop, a D latch, and a RS latch. The two flip-flop configurations are clocked by the clock source from the even-numbered LUT, while the latch configurations are not clocked, and acts instantaneously to changes on the inputs.

2.5.5 Analog Comparator (AC)

The AC is an analog comparator, which compares two analog voltages and gives a digital output corresponding to the sign of their difference. Both inputs can be connected to external pins, but the negating input can alternatively be connected to either a internal bandgap reference voltage, or to an internal DAC, allowing fine control over the threshold voltage.

The AC is not clocked, meaning it continuously compares the inputs, and outputs a continuous result. Additionally, the comparator has a programmable amount of hysteresis, and the output can be inverted. Finally the output can be routed into the event system, and can drive a GPIO pin.

2.5.6 Serial Peripheral Interface (SPI)

The SPI is a synchronous data transfer interface, normally used for full duplex communication between the microcontroller and other devices. The interface is based around a shift-register, which is loaded with the data word to transmit. Then, when operating as a bus master, the word is shifted sequentially out on the Master-Out-Slave-In (MOSI) line, while another word is shifted into the shift-register from the Master-In-Slave-Out (MISO)

line. This simultaneous shifting is clocked using the Serial Clock (SCK) line, which is driven by the bus master.

While this functionality is normally used for communication, the shift register can be leveraged to read any bitstream, such as the one produced by a delta-sigma modulator. This use-case is explored in chapter 4.

Chapter 3

Design Process

This chapter describes the process of implementing the ADCs. The finished implementations, which are used in the characterization are described in more detail in chapter 4.

3.1 Initial Considerations

The first step of the design process was to understand the different types of $\Delta\Sigma$ ADCs, and how they could be realized. Additionally, the resources available in the AVR-DB MCU had to be understood, so that decisions could be made about what seemed feasible, and what could be ruled out. It quickly became clear that the op-amp integrator would be the main op-amp circuit required. This circuit was the first to be built and tested, giving promising results as shown in Figure 3.1.

During this initial testing, the analog comparator peripheral (AC) was also tested, to see how it could be connected to the op-amp output. In the pinout of the microcontroller, it can be seen that several of the AC inputs share pins with the OPAMP outputs. Luckily, it is possible to enable both functions on these pins, allowing the AC to be directly connected to the OPAMP output without an external jumper. In Figure 3.1, channel 3 shows the AC output when comparing the integrator output with $\frac{V_{DD}}{2}$.

After this initial testing, it was decided to attempt implementations of both first- and second-order $\Delta\Sigma$ ADCs, as well as a traditional integrating multi-slope ADC, since this would have a more-or-less identical hardware configuration as a first-order $\Delta\Sigma$ modulator, consisting of a single op-amp integrator, with the output connected to an analog comparator.

3.2 Prototyping

The prototyping of the implementations was done with an AVR128DB28 device in a DIP-28 package on a breadboard, as shown in Figure 3.2. This device was selected simply because it was the first engineering sample available, and it was convenient to use on a

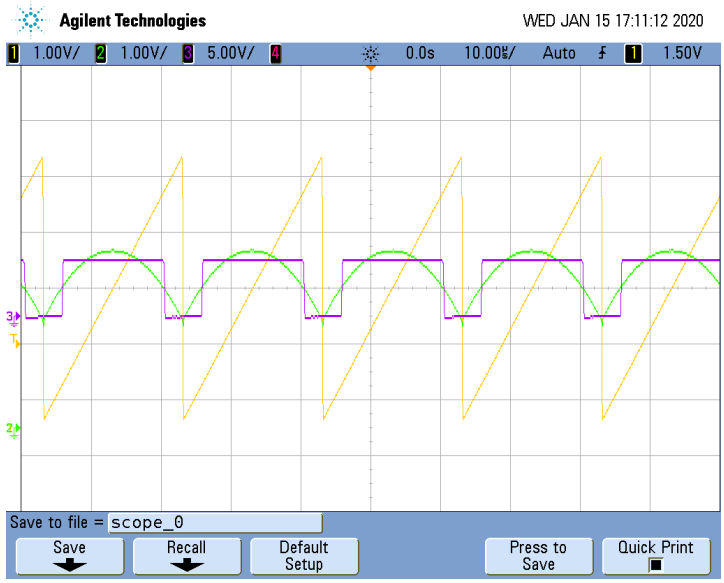


Figure 3.1: Oscilloscope plot of the operation of an op-amp integrator. Channel 1 shows the input, generated by a signal generator, channel 2 shows the output of the integrator, and channel 3 shows the output from an analog comparator comparing the integrator output with $\frac{V_{DD}}{2}$.

breadboard. One downside was that the 28-pin version of the AVR-DB only has two op-amps in the OPAMP peripheral, potentially limiting the possible designs. This turned out not to be a problem, however, as all the desired implementations was realizable with only two op-amps.

3.2.1 First-Order Modulator

After the initial testing of a simple integrator, a complete first-order $\Delta\Sigma$ modulator was implemented. Initially this was done with the CPU periodically reading the AC result and applying a correcting signal to the summing integrator using a digital output pin. While this consumed nearly all the CPU time, it verified the concept, and allowed the CPU to also count the high and low AC readings, giving a rudimentary measurement result.

To make the modulator core-independent, the CPU reading the AC output was replaced with a D-latch, implemented in the CCL. This allowed the modulator to run without the CPU, but it still needed some way of counting or recording the clock cycles. This was implemented by using two timer-counters, along with the event system.

Once this was working, the breadboard prototype was connected to the characterization-setup to see what performance could be expected. This testing revealed one potential issue with the input. In order to eliminate any effects of source impedance, a second op-amp was used in unity-gain configuration to buffer the input signal. This turned out to limit the usable range of the input, causing nonlinearities for input voltages close to the rails. This was caused by the fact that the integrated op-amps actually saturate a bit inside the rails.

After verifying that the voltage source in the characterization-setup had negligible output impedance ($0.2\,\Omega$ [2]), this buffer was removed, which eliminated the saturation-issues close to the rails.

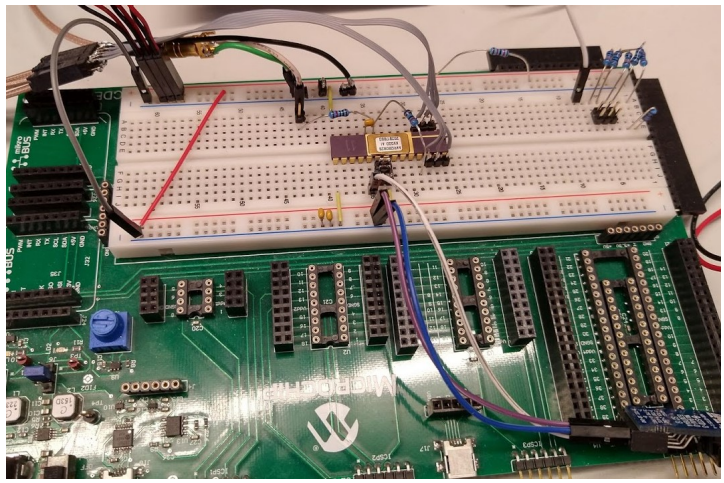


Figure 3.2: The breadboard setup for prototyping op-amp circuits and ADC implementations. Note that the circuit-board the breadboard is mounted to is unrelated to the project, and not electrically connected to anything on the breadboard itself.

Apart from this issue, the testing on this implementation indicated that it was working well, so the focus shifted to continuing implementing alternatives. The timer-counter based averaging was limited by the fact that it required 2^n clock cycles to be able to produce a n-bit result, giving very long conversion times. In order to use a general Finite Impulse Response (FIR) filter, each bit in the bitstream must be handled individually. To avoid having to do this in real-time, the SPI was used to record the bitstream, so that a filter-kernel could be applied afterwards.

3.2.2 Second-Order Modulator

Next, a second-order modulator was implemented. This started to push the available space in the pinout of the 28-pin device, and required some care to select compatible instances of the peripherals. For instance, the output of `OP0` (instance 0 of the OPAMP peripheral) shares its pin with input 2 of `LUT2` in the CCL, so these signals cannot be used together. However, a compatible selection of peripherals was found, and the second-order modulator was implemented successfully.

3.2.3 Multi-Slope ADC

Finally, the multi-slope ADC was implemented. This used the same integrator and comparator as the first-order $\Delta\Sigma$ configurations, but re-introduced the second op-amp to buffer the input. This was necessary in order to multiplex in either the voltage input (for the first slope), or a software-controllable DAC-voltage (for the return-slopes).

While the hardware-configuration was mostly the same, or even a bit simpler, than for the $\Delta\Sigma$ ADCs, the firmware turned out to be more challenging. Since this type of ADC performs the conversion in several stages, which are not easily implemented in a core-independent way, the CPU has to run code to control this process, which makes the firmware more complex. This also meant that implementing multi-slope rundown proved very difficult, as the initial return-slope would inevitably overshoot the threshold by much more than a single clock cycle, due to the delay between the interrupt-trigger and the CPU changing the input to the integrator. Multi-slope rundown was ultimately abandoned, with only a single rundown slope being used for the characterized ADC. However, it might be possible with more careful design, and this is discussed further in chapter 6.

3.3 Characterization PCB Design

After all the desired designs had been implemented and shown to be working, a Printed Circuit Board (PCB) was designed in order to connect the implementations to the characterization-setup in a reliable way, shown in Figure 3.3. Although the different implementations require different external passive components and connections, the circuits were similar enough that a single PCB could support all configurations by providing extra pads and jumpers. The PCB was designed for the same interface as Microchip normally uses for ADC characterization, allowing mostly plug-and-play operation with the lab-setup.

By this time in the project, engineering samples of the AVR128DB48 in a TQFP-48 package was available, so this was used for the characterization.

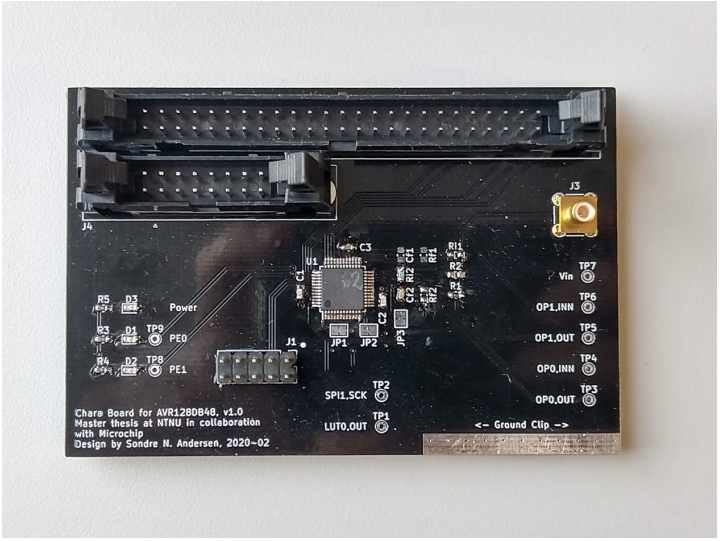


Figure 3.3: The PCB designed for characterizing the implemented ADCs.

Chapter 4

Implementation

This chapter describes the implementation details for each of the implemented ADCs. Two different types of ADCs were implemented and characterized: $\Delta\Sigma$ and dual-slope. The $\Delta\Sigma$ type was additionally implemented in three different variants, resulting in a total of four different configurations. The performance of the ADC implementations is described in chapter 5. Since the various implementations share several subcomponents, such as the op-amp integrator, each component is only described in detail along with the first circuit in which they are used.

The code used for configuring and running the ADCs are listed in appendices 7.5.1 to 7.5.5. Each of the four ADC implementations share a common interface, consisting of an initialization-function, a function to start the conversion, a check for if the conversion is complete, and a getter-function to read back the result. In addition, the code implements Interrupt Service Routines (ISRs) for various interrupts.

The full circuit diagrams in this chapter (figures 4.1, 4.5, 4.6, and 4.8) show both the external components, as well as the internal peripherals used for the circuit. The dashed line indicates the boundary between the MCU and the external components. The occupied MCU pins are shown with the name of the relevant signal, the pin name (e.g. `PD3` indicating that this is the third pin on `PORTD`), and in parentheses, the pin number on the 28-pin DIP package.

4.1 First-Order $\Delta\Sigma$ ADC with a Rectangular Moving Average Filter (ADC1)

The first implementation consists of a first-order $\Delta\Sigma$ modulator, connected to a pair of timer/counter-modules, acting as a rectangular moving average filter.

The modulator is built using one op-amp (`OP0`), one analog comparator (`AC0`), and a pair of LUTs from the CCL (`LUT0` and `LUT1`). The op-amp, along with the three external components, is configured as a summing integrator. Internally in the MCU, the output of this integrator is quantized by the analog comparator, followed by a sampling by `LUT0`

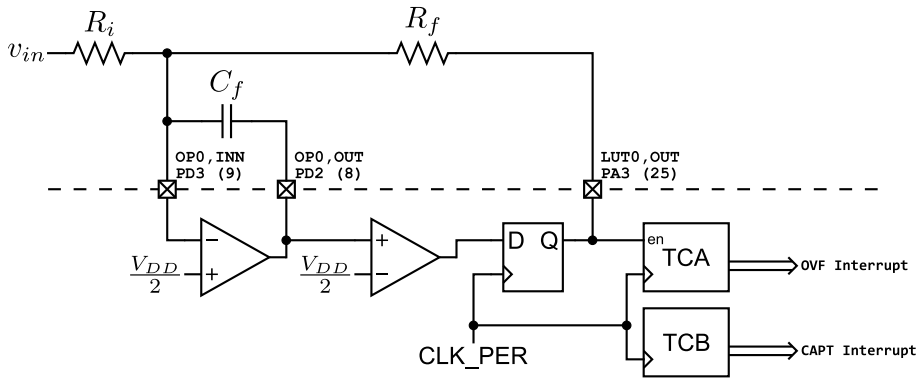


Figure 4.1: The circuit making up the $\Delta\Sigma$ ADC, with a rectangular moving average filter.

and `LUT1` from the CCL, configured as a D flip-flop. The output of this flip-flop is the output bitstream, and it is fed back to the integrator through resistor R_f.

The bitstream is also passed, through the event-system, to timer/counter `TCA0`. This timer/counter is configured to count the clock-edges of the peripheral clock, `CLK_PER`, while the bitstream is high. Meanwhile, `TCB0` is configured to count every clock edge, and provide an interrupt after a predetermined number of clock cycles. The result is that after this interrupt occurs, the value in the count-register of `TCA0` (`TCA0.CNT`) is proportional to the average duty-cycle of the bitstream.

4.1.1 Integrator

The integrator is built as a standard summing, inverting integrator-circuit, with a feedback-capacitor C_f, and input resistors R_i and R_f. The non-negating input of the op-amp is connected to $\frac{V_{DD}}{2}$, to allow the circuit to integrate in both directions, with inputs in the range GND to V_{DD}.

The function of the integrator is described in section 2.3. With two input-resistors, R_i and R_f, feedback-capacitor C_f, and V_{ref} set to $\frac{V_{DD}}{2}$, (2.1) can be written:

$$v_{\texttt{OP0,OUT}} = \frac{V_{DD}}{2} - \frac{1}{C_f}\int (\frac{1}{R_i}(v_{in} - \frac{V_{DD}}{2}) + \frac{1}{R_f}(v_{\texttt{LUT0,OUT}} - \frac{V_{DD}}{2}))dt. \tag{4.1}$$

One important design-decision for this circuit is choosing the values of the resistors and the capacitor. These values must be selected large enough so that the integrator does not saturate over the course of a clock-cycle, but small enough to give a good signal-to-noise ratio on the output. If the input- and feedback-resistors are selected with the same value ($R_f = R_i = R$), (4.1) simplifies to

$$v_{\texttt{OP0,OUT}} = \frac{V_{DD}}{2} - \frac{1}{RC_f}\int (v_{in} + v_{\texttt{LUT0,OUT}} - V_{DD})dt. \tag{4.2}$$

The steepest positive slope of the output results from the lowest value integrand, which occurs when $v_{in} = v_{\texttt{LUT0,OUT}} = 0\,\text{V}$. At the beginning of the clock-cycle, $v_{\texttt{LUT0,OUT}} =$

0 V can only happen if $v_{\texttt{OP0,OUT}} \leq \frac{V_{DD}}{2}$, so this can be taken as the worst-case initial value. The condition that the integrator should not saturate at the end of one clock-cycle, from this worst-case initial state, can then be expressed as

$$v_{\texttt{LUT0,OUT}} = \frac{V_{DD}}{2} + \frac{1}{RC_f}\frac{1}{f_{\texttt{CLK\_PER}}}V_{DD} \leq V_{DD}, \tag{4.3}$$

which simplifies to

$$RC_f \geq \frac{2}{f_{\texttt{CLK\_PER}}}. \tag{4.4}$$

This gives a simple constraint which must be met by the selection of clock frequency, capacitance, and feedback resistance. The same derivation can be done for the symmetric worst-case state with the steepest negative slope, occurring for $v_{in} = v_{\texttt{LUT0,OUT}} = V_{DD}$, which gives the same result.

4.1.2 Analog Comparator

The analog comparator performs the quantizer-function of the modulator, and in this way acts as a 1-bit ADC. The comparator is connected to the integrator output by the fact that the two signals, `OP0,OUT` and `AC0,AINP0` share the same pin on the MCU, allowing them to be connected with no external routing. The voltage at the negating input of the comparator is set by an internal 8-bit DAC, configured to $\frac{V_{DD}}{2}$.

The peripheral operates continuously, since there is no clocked mode available. The peripheral has settings for hysteresis and for trading higher response time for lower power, but both these settings were only found to decrease performance. The comparator is therefore set to its most responsive configuration, with no hysteresis. There is also an option to output the comparison result to an I/O pin. This is not required in operation, as the signal can be routed through internal channels, but can be valuable for debugging purposes.

4.1.3 D Flip-Flop

The D flip-flop performs the sample-and-hold functionality of the modulator. It is implemented with the CCL, using `LUT0` and `LUT1`. Within the CCL, each LUT does not have support for sequential logic by itself, but each pair of adjacent LUTs share a sequential block, which can be configured in several different ways. The relevant configuration here is the D flip-flop configuration, where the even LUT (`LUT0` in this case), provides the data-value, while the odd LUT (`LUT1`), provides a gate-value. The flip-flop only operates while the gate-input is high, so `LUT1` is configured to always output a "1".

Each input of the LUT can be multiplexed between a variety of signal sources, including the outputs of the analog comparators. The even LUT can therefore simply be configured to pass the analog comparator output to the sequential block. The clock for the sequential block can also be multiplexed between several different sources. In this case, it should be clocked by the same signal that clocks the counters used in the filter, which is the system-wide `CLK_PER`.

4.1.4 Feedback and Modulator Operation

In order to close the feedback-loop, and complete the $\Delta\Sigma$ modulator, the output bitstream is fed back to the integrator through the feedback-resistor R_f. In the ideal case, for DC-inputs, the average duty-cycle of the output is exactly proportional to the input voltage v_{in}, which relies on the input- and feedback-resistors being exactly equal. However, this might not always be the case.

If R_i is lower than R_f, the input component becomes stronger than the feedback. This means that if the input gets too close to either supply-rail, the feedback will be unable to prevent the integrator from drifting away from $\frac{V_{DD}}{2}$, even with $0\,\%$ or $100\,\%$ duty-cycle. This effectively limits the range of the ADC to a reduced region, centered around $\frac{V_{DD}}{2}$.

If R_f is lower than R_i, the feedback component becomes stronger than the input, resulting in less extreme duty-cycles for the rail voltages. This means that $0\,\%$ and $100\,\%$ duty-cycle falls outside of the GND to V_{DD} input range, resulting in lower effective resolution in this range.

In specific applications, these effects could be leveraged to either focus the available resolution around $\frac{V_{DD}}{2}$, or extend the input range well beyond the MCU supply. This possibility is discussed further in section 6.4.1.

Obtaining Full Resolution Within the Supply-Range

For this project, the goal is to get the highest possible resolution across the full supply-range. However, even if a trimmer-potentiometer is used to set R_i as close as possible to R_f, it is difficult to prevent integrator drift when the input voltage is very close to the rails. In order to avoid this, the modulator can be trimmed with R_f slightly higher than R_i, resulting in a small non-zero duty-cycle for $0\,\mathrm{V}$ inputs, and a slightly less than $100\,\%$ duty-cycle for $v_{in} = V_{DD}$, effectively introducing a controlled gain-error. This keeps the duty-cycle of the bitstream away from the saturation regions, where slight variations can cause instability or other issues. The gain-error can then be corrected for in the software filter. This is described further in the next section.

4.1.5 Rectangular Moving Average Filter

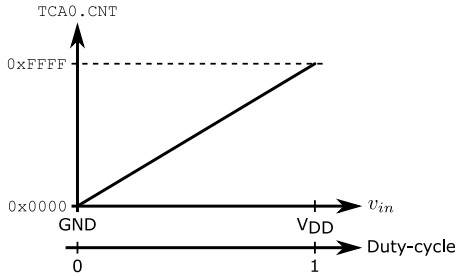
This first ADC implementation uses a rectangular moving average filter, implemented using two timer/counter peripherals. `TCA0` counts the high pulses of the modulator output, while `TCB0` keeps track of how long `TCA0` has been counting for. In the simplest setup, `TCB0` gives an interrupt after $2^{16} - 1$ clock cycles, and `TCA0` has then counted to between 0 and $2^{16} - 1$, giving a 16-bit measurement of the average duty-cycle.

One disadvantage of this filter is that it requires approximately 2^n clock cycles to provide n bits of resolution, giving a quite low sample rate if high resolution is required. However, during these clock-cycles, the filter operates completely independent of the CPU.

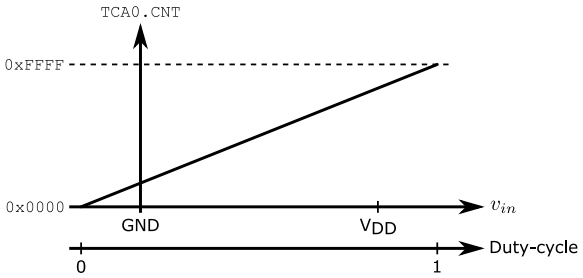
Correcting the Gain-Error Introduced by Mismatched R_i and R_f

In order to correct for the gain-error described in section 4.1.4 without losing resolution, a small modification is necessary. The idea is to allow `TCB0` to count for more than

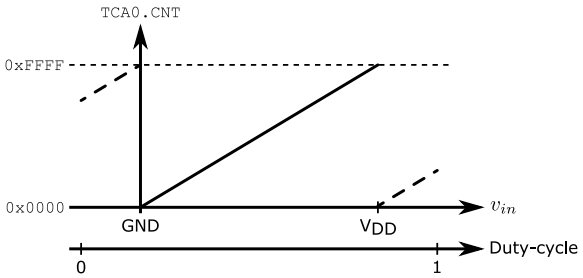
$2^{16} - 1$ clock cycles, which increases the gain. This also introduces an offset, which can be canceled by starting `TCA0` a bit "below" zero, meaning at a value close to the maximum.



(a) Original transfer function between v_{in} and duty-cycle, and `TCA0.CNT`.



(b) Transfer-function when $R_i > R_f$.



(c) Transfer-function once the correction has been implemented.

Figure 4.2: Correcting the gain-error introduced by mismatch between R_i and R_f by manipulating the start- and end-points of the timer/counters used for filtering.

Figure 4.2 shows the steps required. In Figure 4.2a, the input- and feedback-resistances are perfectly balanced, giving a zero duty-cycle at $v_{in} = 0\,\mathrm{V}$, and a $100\,\%$ duty-cycle at $v_{in} = V_{DD}$. After R_i is reduced, the desired input-range covers a smaller portion of the range of duty-cycles, as shown in Figure 4.2b. Note that `TCA0.CNT` is still proportional to the duty-cycle, meaning a gain-error has been introduced, centered around $\frac{V_{DD}}{2}$. In the figure, the gain-error is exaggerated, to about $50\,\%$. Figure 4.2c show the situation after the correction has been introduced. `TCB0` is allowed to count for longer, $50\,\%$ extra, in

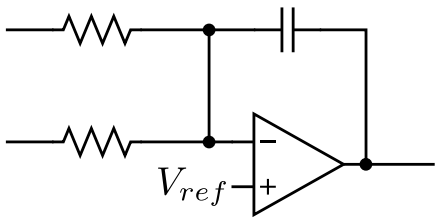
this case, and TCA0 is started at a value $25\,\%$ below the maximum, which cancels the introduced offset.

To avoid having to deal with multiple overflows of TCB0, since it now needs to count for more than 2^{16} clock-cycles, its prescaler can be used to slow the counting down by a factor of two. With this slower clock, TCB0 only needs to count approximately 2^{15} cycles, plus the additional time needed to account for the gain-error.

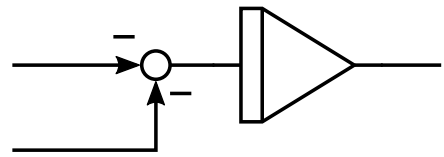
An additional benefit of this solution is that the overflow-interrupt for TCA0 can be handled, giving a way to detect out-of-range inputs. With in-range inputs, TCA0 should overflow exactly once. If TCA0 overflows twice, this indicates that v_{in} was higher than the calibrated range. If the duty-cycle was low enough that TCA0 does not overflow, v_{in} was below the calibrated range. This does make the ADC no longer fully core-independent, but this interrupt handler only needs to increment a variable, and will therefore only require a few clock-cycles, once or twice per conversion.

4.1.6 Dealing with the Inverting Integrator

The block-diagram for a first-order $\Delta\Sigma$ ADC shown in Figure 4.4a uses non-inverting integrators, but the op-amp integrator described in section 4.1.1 is inverting. This means that some block-diagram manipulation is required to assemble a correctly functioning circuit. The integrator circuit and its equivalent block-diagram are shown in Figure 4.3.



(a) Circuit diagram for the op-amp implementation of a summing integrator.



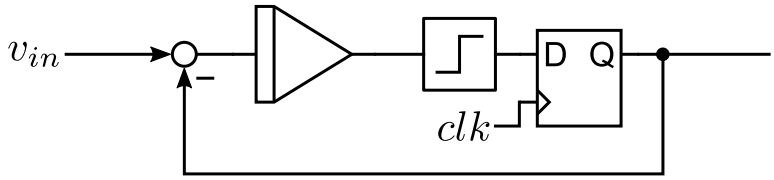
(b) Equivalent block diagram. Note that both inputs to the summing node are inverting.

Figure 4.3: Integrator circuit and its equivalent block-diagram

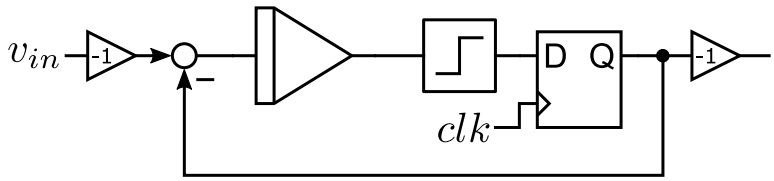
For this first ADC implementation, the simplest option is to insert an inverter at the input and output of the entire ADC (Figure 4.4b). The output-inverter is easily implemented in software, while the input-inverter, together with the integrator and summing node in the original block-diagram, can be replaced by the op-amp implementation of an integrator (Figure 4.4c). Section 4.1.6 describes the more complex solution required when two integrators are cascaded.

4.1.7 Conversion Time

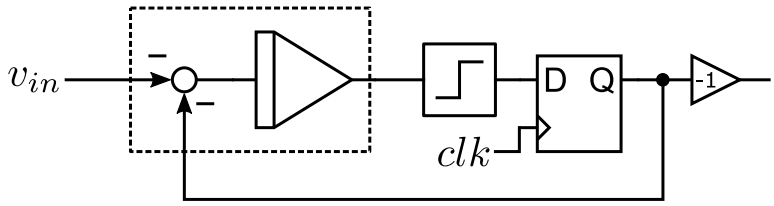
Since this ADC needs to be able to accumulate up to 2^{16} counts, the clock rate needs to be as high as possible to avoid the conversion taking too long. The AVR-DB can run up to $24\,\mathrm{MHz}$, but at this clock rate, the modulator does not perform well, resulting in a large amount of noise on the measurements. This is probably in part caused by the limited bandwidth of the internal op-amp, which is around $1\,\mathrm{MHz}$.



(a) Original block diagram, with standard integrator. Note that the summing node has one inverting and one non-inverting input.



(b) Equivalent block diagram, with inverters added to the input and output.



(c) Final block diagram, with the inverting integrator in the dashed rectangle. Note that both inputs to the summing node are inverting.

Figure 4.4: Block diagram manipulation needed to use the inverting integrator circuit. Note that the digital low-pass filter is not shown.

The best results were obtained with a clock frequency of $4\,\mathrm{MHz}$. At this clock speed, each conversion takes approximately $\frac{2^{16}}{4\,\mathrm{MHz}} = 16.38\,\mathrm{ms}$.

4.2 First-Order $\Delta\Sigma$ ADC with FIR Filter (ADC2)

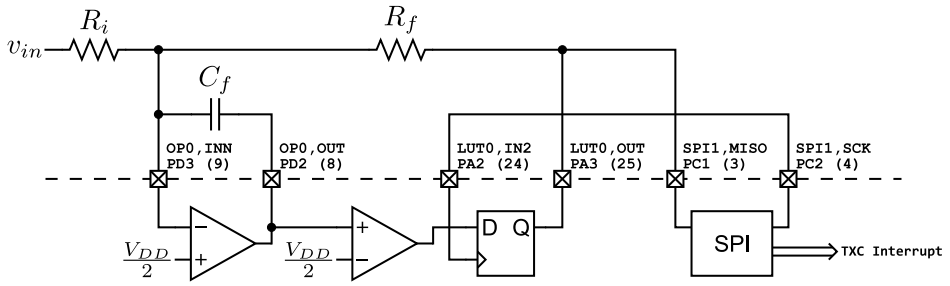


Figure 4.5: The circuit making up the second $\Delta\Sigma$ ADC, with a sinc-in-time FIR-filter.

The second implementation is very similar to the first, but replaces the simple rectangular filter with an FIR filter, with arbitrary length and coefficients. This requires the MCU to record the output of the modulator at each clock-cycle, so that the filtered value can be obtained through a convolution with a filter kernel. One possibility would be to let the CPU handle an interrupt, and manually read the output of the modulator for each bit, but this would require a lot of CPU-intervention. Using the SPI peripheral as a shift-register, the number of times the CPU must intervene can be reduced by approximately a factor of eight, while also avoiding having to have the CPU handle individual bits.

4.2.1 Reading the Bitstream with the SPI

In this implementation, the modulator is controlled by the SPI peripheral. The SPI is instructed by the CPU to transmit a dummy-byte, which causes the `SCK` line to produce eight clock pulses, which in turn are used to clock the modulator. Since the SPI protocol specifies full-duplex communication, the SPI expects data on the Master-In-Slave-Out, or `MISO` line, so the output of the modulator is connected there, allowing the SPI to read the bitstream into its shift register.

As the shift-register in the SPI only holds one byte of data, the CPU has to intervene every eight bits. This is done by handling the "Data Register Empty" (`DRE`) interrupt, which triggers when the SPI has loaded the dummy-data into its shift-register. The reason to handle this interrupt, as opposed to the more intuitive "Receive Complete" (`RXC`) interrupt, is that the `RXC` interrupt only triggers after the shift-register has shifted out all its data, generating the eight clock-edges, which means there will be a significant delay until new dummy-data can be sent. Handling the `DRE` interrupt instead allows new dummy-data to be ready immediately once the shift-register is empty. There still seems to be a two CPU-cycle delay between bytes, presumably as the SPI moves data between internal registers. However, since a relatively low oversampling ratio is used, the SPI clock can

be slowed down to 128 times relative to the CPU clock, making the two CPU-cycle delay very short compared to the total clock period of the SPI clock.

```
ISR(SPI1_INT_vect)
{
  // Read received bitstream data
  uint8_t data = SPI1.DATA;
  // Set new dummy-data to continue the "transmission"
  SPI1.DATA = 0xFF;

  // Fill the bitstream-array until the required number of
  // bytes has been captured
  if (byte_index < BITSTREAM_LENGTH_BYTES) {
    bitstream[byte_index++] = data;
  }

  // Since the filter-kernel is symmetric, we only store
  // one half of it in memory. To use it for both halfs of
  // the filter, the second half of the bitstream must be
  // reversed. To avoid having to flip all the bytes later,
  // we can use the Data Order bit in the SPI to flip the
  // bits for us.
  if (byte_index >= BITSTREAM_LENGTH_BYTES / 2) {
    SPI1.CTRLA &= ~SPI_DORD_bm;
  }

  // Finally, clear the interrupt flag
  SPI1.INTFLAGS = SPI_DREIF_bm;
}
```

Listing 4.1: Interrupt handler for reading the ΔΣ-modulator bitstream using the SPI.

Once triggered, the interrupt handler can read the previously received data, store it in an array, and load another dummy-byte into the data-register. The full ISR is shown in listing 4.1.

In order to clock the modulator from the SPI, one modification must be made to the configuration of the CCL. In the first-order modulator, the CCL was clocked by `CLK_PER`, which is an internal signal. Unfortunately, there is no option to select the SPI-clock as a clock source for the CCL, but there is an option to use input 2 (`LUTn,IN2`) as a clock. This signal is therefore routed externally, from `SPI1,SCK` to `LUT0,IN2`.

4.2.2 FIR Filter

Once the `DRE` interrupt has been handled enough times to receive as many bits as the oversampling ratio, the data acquisition is complete. Now, the CPU can perform the convolution with the filter kernel, giving the single sample of the filtered and decimated signal.

One simplifying factor in this convolution is that the output from the modulator is a binary signal. This means that each kernel coefficient should simply either be added, or not added to the accumulator, meaning no multiplications are necessary. As the filter kernel is symmetric, another optimization can be implemented by only storing the first half of it. However, this means that the second half of the bitstream must be mirrored. To avoid having to mirror each byte in software, they can be mirrored by flipping the Data Order

bit in the SPI halfway through the acquisition, which causes the bits to be read into the shift-register in the reverse order.

The filter was chosen as a simple sinc-in-frequency filter, with 256 coefficients, a cutoff-frequency of 0.004, and a cosine window function. The kernel coefficients were computed on the device itself, using the code shown in listing 4.2.

```
#define FIR_FILTER_KERNEL_SIZE 256
#define FIR_FILTER_KERNEL_SCALE 65535

#define SINC(x) ((x) ? (sin(x)/(x)) : (1))
#define WINDOW(x) (0.5 - 0.5 * cos(2.0 * M_PI * (x)))

void fir_init_kernel(const float f_c)
{
  for(int i = 0; i < FIR_FILTER_KERNEL_SIZE / 2; i++) {
    int index = i - FIR_FILTER_KERNEL_SIZE / 2;
    float ideal_kernel = 2.0 * f_c
      * SINC(2.0 * M_PI * f_c * index);
    float window =
      WINDOW((float)i / FIR_FILTER_KERNEL_SIZE);
    kernel[i] = (int16_t)(FIR_FILTER_KERNEL_SCALE
      * window * ideal_kernel);
  }
}
```

Listing 4.2: Function for calculating the coefficient for the filter-kernel.

4.2.3 Conversion Time

Since this ADC only records enough samples to be able to preform a convolution with the filter kernel, it requires much fewer cycles to complete one conversion. Additionally, the modulator is running at a clock frequency 128 times slower than the main clock, so the main CPU clock can be set to the maximum 24 MHz, without degrading the modulator performance.

With a filter-kernel of 256 coefficients, meaning the ADC requires 256 samples to complete one conversion, this results in a conversion time of $\frac{128 \times 256}{24\,\mathrm{MHz}} = 1.365\,\mathrm{ms}$. This does not include the time required for the convolution, since this would depend on the acquired data, and on how well the assembly is optimized. A rough estimate of ten cycles on average per bit gives an additional time of 107 µs.

4.3 Second Order $\Delta\Sigma$ ADC with FIR (ADC3)

The third configuration is similar to the second, but implements a second-order $\Delta\Sigma$ modulator. The digital filter is implemented exactly the same as for the previous version.

4.3.1 Second-Order $\Delta\Sigma$ Modulator

To build a second-order $\Delta\Sigma$ modulator, a second integrator is needed. This is implemented with `OP1`, and requires a second set of resistors and a capacitor, bringing the total number

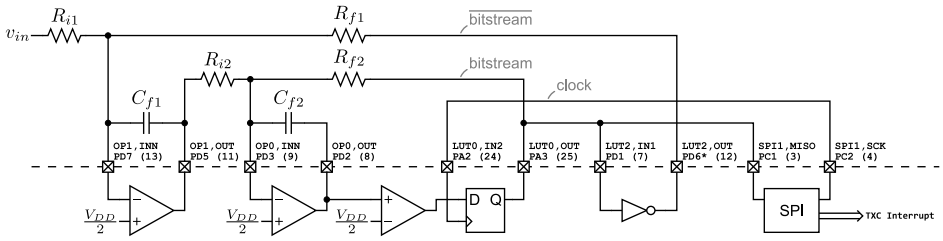


Figure 4.6: The circuit making up the third ΔΣ ADC, with a second-order modulator.

of external components to six. Functionally, the second-order modulator is very similar to the first-order modulator. The digital filter is identical, using the SPI to read the bitstream produced by the D flip-flop.

4.3.2 Dealing with the Inverting Integrator

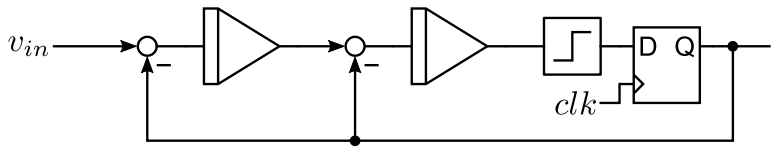
As described in section 4.1.6, the fact that the integrator circuit built around the op-amp is inverting means that the block-diagrams for the ADC must be modified. Figure 4.7 shows the modifications needed for the second-order ADC. For the first-order ADC, the required modifications only demanded an inversion to be added at the final output of the ADC, which could easily be implemented in software. For the second-order ADC, the simplest required modifications add an inverter in the feedback-path, which has to be implemented in hardware.

4.3.3 Pin Allocation for the Added Inverter

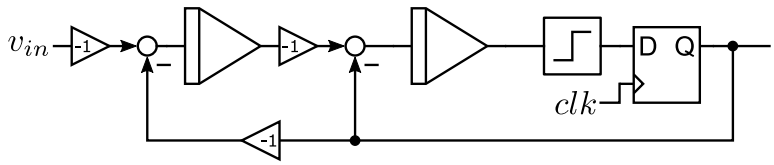
Since the added inverter only needs to process the binary feedback-signal, it can be implemented using the CCL, by configuring a LUT to take a single input, and output the inverted result to a pin. The CCL has the possibility for `LUTn` to select the output of `LUT[n+1]` as an input, so this connection could in theory be made internally. Unfortunately, as this third ADC implementation occupies a significant portion of the available pins on the 28-pin package, the options for how to allocate pins and peripherals becomes limited. In order to fit this circuit within the 28-pin package, the connections to this inverter needs to be made externally, including routing the output of `LUT2` to its alternate pin. On the higher pin-count packages, more peripheral options and pins are available, so this implementation could likely be simplified.

4.3.4 Conversion Time

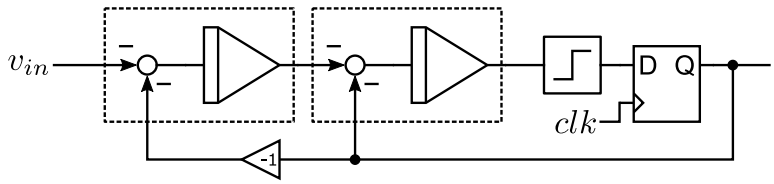
Apart from the added complexity in the analog circuit, and initialization code, this ADC works identically to the first-order ΔΣ ADC with and FIR filter. Therefore, the conversion time is the same, at $1.365\,\text{ms}$, plus time for the convolution.



(a) Original block diagram, with standard integrators. Note that the summing nodes has one inverting and one non-inverting input.



(b) Equivalent block diagram, with inverters added to both inputs and the output of the first integrator.



(c) Final block diagram, with the inverting integrators in dashed rectangles. Note that both inputs to both the summing nodes are inverting. The remaining inverter in the feedback to the first integrator is implemented using the CCL.

Figure 4.7: Block diagram manipulation needed to use the inverting integrator circuit. This only shows the modulator, not the digital low-pass filter.

4.4 Dual Slope ADC (ADC4)

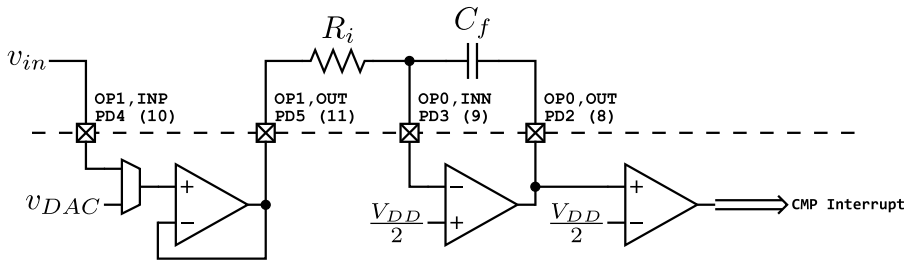


Figure 4.8: The circuit making up the dual-slope ADC.

The final implementation considered is a dual-slope type ADC. This implementation is much simpler in terms of internal and external circuitry, requiring only two external components, and no CCL or SPI. However, it is more involved on the software side, as the conversion happens in several stages, with different configurations.

The idea is that `OP1` is used to buffer the input, and functions as a multiplexer. This allows either the input voltage v_{in}, or the output from the DAC to be passed to the input of the integrator. The sequence is that the input is integrated, controlled by `TCA0` configured to give an interrupt after a fixed amount of time, then a known voltage from the DAC is integrated, and the time needed to cross a threshold is measured, again using `TCA0`.

Typically, a dual-slope ADC is illustrated as first integrating up from zero, then integrating back down to ground. This is impractical in this implementation because the integrator saturates at zero, meaning that it is difficult to measure when it "crosses" this threshold. Instead, the integrator is configured with the same $\frac{V_{DD}}{2}$ offset as has been used in the previous implementations. This means that if $v_{in} > \frac{V_{DD}}{2}$, the inverting integrator will integrate downwards, and the run-down slope must be positive, while if $v_{in} < \frac{V_{DD}}{2}$ the integrator will integrate upwards, meaning the run-down slope must be negative. This is controlled in software, after the initial integration of the input, by looking at the output of the comparator to see which direction the run-down should happen in. This comparator also serves to detect when the run-down is completed, at which point the conversion is complete.

The result captured in `TCA0.CNT` will not reflect which direction the run-down happened in, and in fact, the direction of the run-down represents a 17th bit of the result. Due to the fact that the characterization-setup is only capable of capturing 16-bit results, this fact is ignored in the firmware, the 16-bit value in `TCA0.CNT` is returned directly, and some post-processing is done to extract the final 17-bit result. This is further discussed in section 5.3.4.

4.4.1 Input Buffer and Multiplexer

The OPAMP-peripheral consists of two or more op-amps, each with a controller around it, allowing it to be configured by software. Parts of this controller are multiplexers on both inputs of the op-amp, allowing them to be connected to a range of signal sources. The dual-slope type ADC requires this ability in order to multiplex the input to the integrator between the signal input and a fixed reference. The op-amp itself is configured for unity-gain, but if gain is desirable, the resistor-ladder in the peripheral can be used to configure up to 16 times gain.

An additional benefit of routing the input through an op-amp before the integrator is that it buffers the signal, massively increasing the input impedance. The previous configurations have all connected the ADC-input to the integrator directly, which means that the input impedance is limited to however large the input resistor is. If higher input impedance is needed, these configurations can be extended to also buffer the input through a unity-gain-configured op-amp.

One disadvantage of buffering the input through an op-amp is that it limits the input range, as the op-amps in the peripheral are not capable of passing the full range from GND to V_{DD}. The previous configurations could all be configured for a much wider input range, by increasing the input-resistance, but the dual-slope configuration is limited to the linear operating range of the op-amps.

4.4.2 Possibility of Multi-Slope Run-Down

One possible improvement to this implementation would be to include a multi-slope run-down. With this method the initial integration of v_{in} works the same, but the following return-integration happens with a much steeper slope. Since the steep slope causes a significant change in voltage within each clock-edge, this will result in some overshoot. A second return-slope can then measure this overshoot by integrating back towards the middle, now with a gentler slope. In theory, this can be repeated several times, allowing an accurate measurement to be taken in a much shorter time.

Unfortunately, this is challenging to make work on the AVR-DB. It can be implemented, by handling the `AC0` interrupt in the same way as with the simple implementation described above, but in this case the delay between the comparator interrupt, and the interrupt handler becomes more of a problem. For the method to work, the integrand needs to change very shortly after the analog comparator triggers, as it is trying to measure the overshoot that occurred within a single clock cycle. Since there is no way of changing the DAC value with direct peripheral-to-peripheral signaling, there will inevitably be a multiple clock cycle delay, during which the integrator will keep integrating past the threshold, washing out the overshoot.

Figure 4.9 illustrates this issue in an attempted implementation of a two-slope run-down ADC. The traces show the two run-down slopes, with the first having a steeper absolute slope than the second. Channel 2 (in green) shows the immediate output from the analog comparator, and channel 3 (in purple) shows the integrator output. Here it can be seen that the integrator continues downward well beyond the falling edge of the comparator. Note that this was run with a $4\,\mathrm{MHz}$ clock speed, meaning that the approximately $20\,\mu\mathrm{s}$ delay between the comparator falling edge and the integrator changing direction cor-

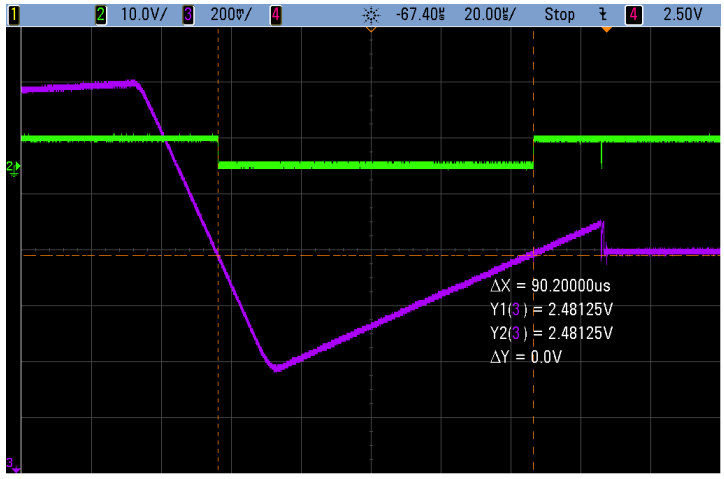


Figure 4.9: Oscilloscope trace illustrating the delay between the analog comparator falling edge, and the integrator changing direction. Channel 2 (in green) shows the immediate output from the analog comparator, and channel 3 (in purple) shows the integrator output. Channel 3 has an offset of $2.5\,\mathrm{V}$

responds to approximately 80 clock-cycles. While this implementation could be improved, and it might be possible to work around this issue, to somehow obtain better results with a dual-slope-type implementation, no further iteration was done on this method.

4.4.3 Conversion Time

Since this ADC uses the duration of the run-down slope for the measurement, this ADC has a varying conversion time. The duration of the two slopes depends on the time-constant of the integrator used, the input-voltage, and the voltage used for the run-down slope.

In the ideal case, the CPU should be able to count any number between 0 and 2^{16} samples during the run-down slope. Since the integrator is not clocked, the CPU clock can be set to the maximum possible $24\,\mathrm{MHz}$, meaning that the run-down slope should ideally take between zero and $\frac{2^{16}}{24\,\mathrm{MHz}} = 2.731\,\mathrm{ms}$. This can be tuned in after the integrator time-constant has been set, by adjusting the DAC voltage applied during this slope.

The duration of the first slope, depends on how long the integrator needs to reach close to saturation, which depends on the time-constant of the integrator. Using the values $R_i = 10\,\mathrm{k\Omega}$ and $C_f = 10\,\mathrm{nF}$, an integrating time of $133.3\,\mu\mathrm{s}$ was found to be suitable, corresponding to 3200 cycles of the $24\,\mathrm{MHz}$ CPU clock.

This results in a conversion time of between $133.3\,\mu\mathrm{s}$ and $2.864\,\mathrm{ms}$. This estimate neglects the processing time, meaning there is a few extra cycles of delay between the slopes and at the start and end of the conversion.

Chapter 5

Characterization

To measure the performance of the implemented ADCs, each circuit was characterized. This is a process where known signals are applied to the ADC input, and the measurements it provides are recorded and analyzed. From this, statistical properties about the ADC can be derived. This chapter details the process for capturing these properties, and gives the results for the four implementations.

Although this chapter only presents one set of characterization results of each ADC, several adjustments were made to each in order to see what effects this had on the performance. The numbers presented in this chapter represents the best achieved results. In addition, for the second-order $\Delta\Sigma$ ADC, three different FIR-filters were tested. Chapter 6 includes further discussion about these adjustments, and how they affected the ADCs performance.

5.1 Characterization Setup

The characterization was done in the Microsoft Trondheim office, using the equipment normally used there to characterize the built-in ADCs in Microchip microcontrollers. This is a customized Data Acquisition (DAQ) system developed by Microchip, using hardware from the National Instrument PXI System [1], shown in Figure 5.1, and a LabView Virtual Instrument (VI). The system communicates with the device-under-test (DUT) using a serial communication interface, and custom firmware is running on the DUT to implement this interface, and follow commands. The DAQ hardware includes a 16-bit DAC [2].

While the DAQ system contains a lot of functionality for analyzing the results, these analyses are designed for lower bit-width results than the 16-bit measurements produced by all the ADCs in this project. Instead of relying on this functionality, minor changes were made to the LabView VI in order to record the full 16-bit results from the DUT, and the data was then exported for later analysis using MATLAB. This allowed the data to be gathered over a few days, limiting the time needed in the lab.

The characterization process starts by powering up the device, then reading back the V_{DD} voltage, to compensate for any voltage drop in the cabling. All tests were done with

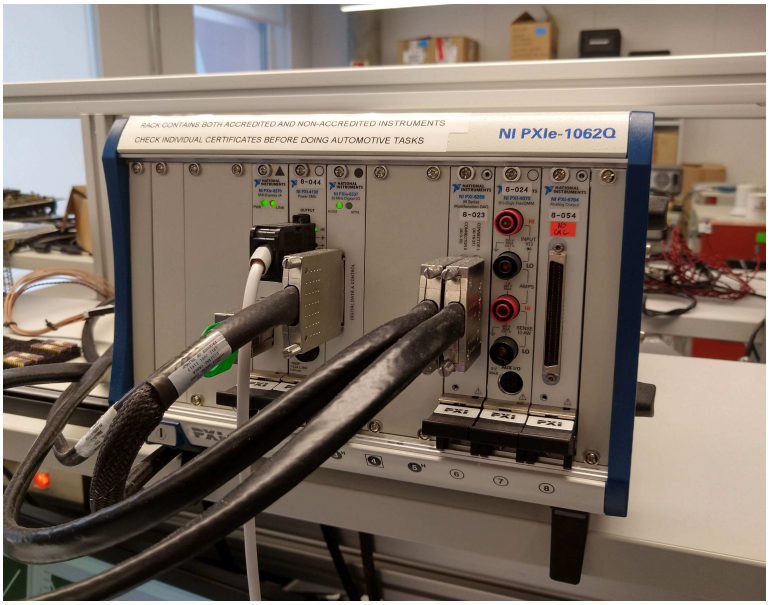


Figure 5.1: National Instrument PXI Instrument.

V_{DD} configured to $5\,\mathrm{V}$. Next, a stepped voltage ramp is applied, and at each step the device is asked to take a number of measurements. These measurements are sent back to the DAQ system, and recorded.

The two parameters of this process is the number of voltage steps, and the number of samples to take for each voltage. Both of these parameters are specified before a characterization-test is run. As each conversion takes some amount of time, a tradeoff must be made between the number of samples taken, and the time required to complete the testing. For each $\Delta\Sigma$ ADC implementations in this project, two tests were done. One with 8096 distinct voltage steps, and only a single sample for each step, and another test with 16 distinct voltages, and 255 measurements per voltage. The exact numbers of steps and repeated measurements are somewhat arbitrary, but the first test was mainly intended to get enough different samples to accurately measure the non-linearity of the transfer function, while the second test was intended to measure the noise of repeated measurements.

5.1.1 Characterization PCB

In order to connect the DUT, and the required external components for the ADC implementation, to the DAQ hardware, a custom PCB was designed. This PCB contains the same interface and power-supply headers as the characterization PCBs usually used with the DAQ system. Figure 5.2 shows the PCB connected to the Microchip-designed interface board, which is connected to the PXI Instrument.

The PCB also contains footprints for the external components, so that all four ADC implementations can be configured by soldering or not soldering some of the components.

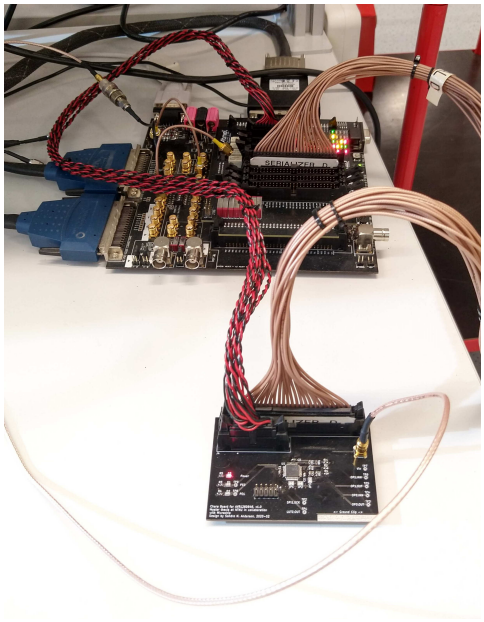


Figure 5.2: Characterization PCB connected to the interface board.

Both the full schematics and the layout of the PCB is given in the Appendix.

5.1.2 DUT Firmware

The firmware running on the DUT during the testing was the same firmware used by Microchip, but modified to work with the $\Delta\Sigma$-ADCs implemented in this project. At its core, the firmware simply consists of the code required to setup and run the ADC implementation, an implementation of the serial protocol for communicating with the DAQ system, and a loop to receive a command to start a conversion, and return the results.

5.2 Plots

In the following section, the measurement results are presented for the implemented ADCs. For the most part, three different plot types are used, and to avoid repetition, they are explained here.

5.2.1 Transfer Function Plots

Figures 5.3, 5.6, 5.8, 5.11, 5.12, 5.15b, 5.16b, 5.17b, and 5.19, simply show the transfer functions of the ADCs. These are scatterplots of pairs of input-voltages and output-codes, showing the correlation between the two. These plots show results from the 8096-step run, with only a single measurement per step.

5.2.2 Noise Plots

Figures 5.4, 5.10, 5.14, and 5.22, show the results of the second run of each ADC, with 255 measurements per step, which is intended to measure the span of codes the ADC can return for a fixed voltage. The dashed line at $y = 0$ represents the average of all 255 measurements in a given voltage step, while the blue and orange lines are the maximum and minimum deviations from this average. Thus, the distance from the blue to the orange line represents the maximum peak-to-peak noise which can be expected.

Upon analyzing the data, it was observed that the first few measurement of each voltage-step occasionally gave very different measurements from the remaining. Since this only affected the first one or two samples, it was assumed to be caused by initial settling of the $\Delta\Sigma$ modulator, and the outliers were removed.

5.2.3 INL Plots

Figures 5.5, 5.7, 5.9, 5.13, 5.20, 5.21a, and 5.21b, show the measured INL. This is derived from the transfer function plots, by computing a best-fit regression line, and then measuring the vertical distance from each sample to this line.

Some of the plots are zoomed in on one part of the input-range. When this is done, the x-axis reflects the relevant range, and it is also mentioned in the text. For INL plots the best-fit line is then computed only for the visible part of the data, in order to obtain an overall horizontal plot.

5.2.4 MATLAB Code

The MATLAB code used to generate the plots in this chapter can be found in the appendix. Appendix 7.4.3 lists the code used to generate the noise plots, and appendix 7.4.1 lists the code used to generate the transfer function and INL plots. Since the dual-slope ADC required additional processing to decode the measurement result, the transfer function and INL plots for this ADC were generated with a separate script, shown in appendix 7.4.2.

5.3 Results

5.3.1 First-Order $\Delta\Sigma$ ADC with Moving Average Filter (ADC1)

Figure 5.3 shows the raw transfer function of ADC1. This ADC was tuned to get close to the maximum range, without saturating the $\Delta\Sigma$ modulator, so the output codes almost span the full range from 0 to $2^{16} - 1$.

Figure 5.4 shows the maximum and minimum noise measured across the voltage range. For the highest voltage in this test, all 255 samples read exactly $2^{16} - 1$, resulting in zero noise. This is most likely caused by the measurement saturating, meaning the noise result for that voltage is meaningless. Overall, the peak-to-peak amplitude of lies between -8.43 and 7.22 LSB.

Figure 5.5 shows the computed INL. This plot shows that voltages close to $\frac{V_{DD}}{2}$ result in slightly too low measurements, compared to the voltages closer to the rails. Additionally

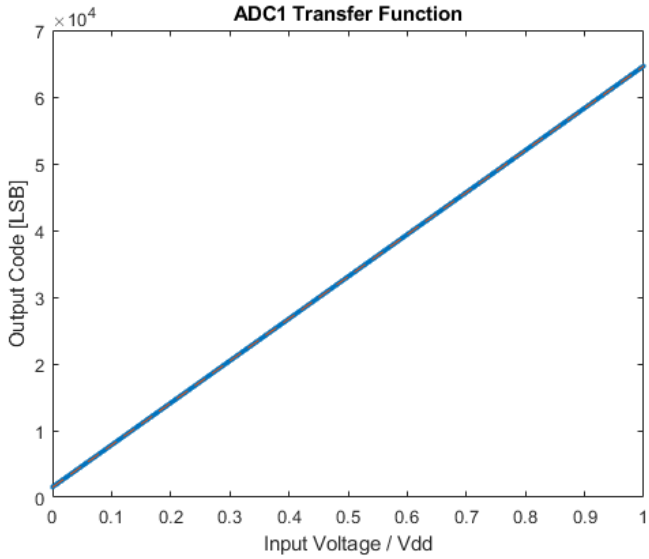


Figure 5.3: Transfer function for $\Delta\Sigma$ ADC 1. The (nearly invisible) dashed orange line is the best-fit regression line used to calculate INL.

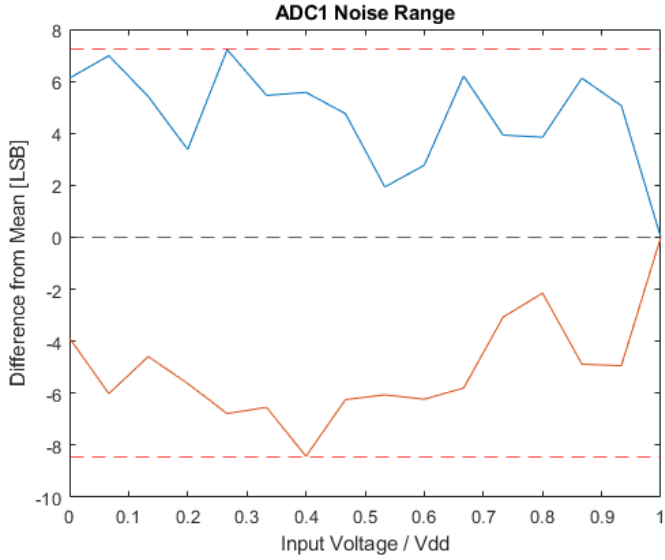


Figure 5.4: Plot of the noise for $\Delta\Sigma$ ADC 1. The red dashed lines shows the lowest deviation at -8.43 LSB, and the highest deviation at 7.22 LSB.

the noise can be seen to be consistent with Figure 5.4, being approximately equal to 15 LSB across the entire range.

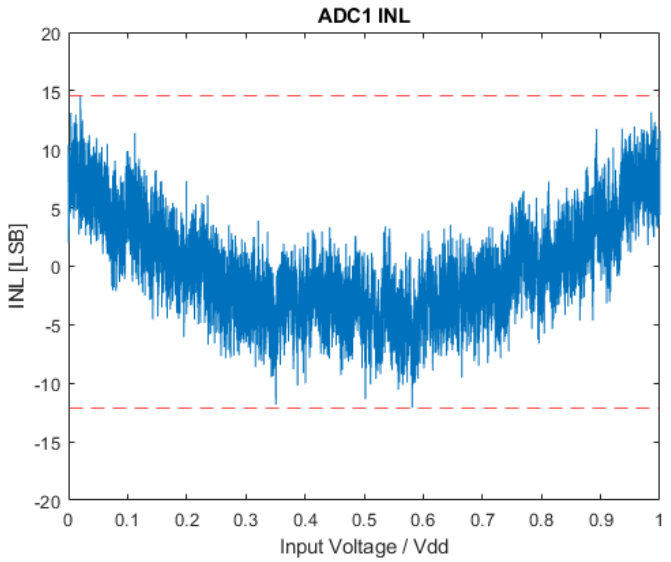


Figure 5.5: Plot of the INL for $\Delta\Sigma$ ADC 1. The red dashed lines shows the range of the INL from -12.10 LSB, to 14.58 LSB.

5.3.2 First-Order $\Delta\Sigma$ ADC with FIR-Filter (ADC2)

Figure 5.6 shows the raw transfer function of ADC2. Already, severe flat-spots can be seen in regular intervals along the input-range.

Figure 5.7 shows the INL computed across the entire range. The regular flat spots cause large swings in the INL.

In order to more closely examine the transfer function between each flat-spot, Figure 5.8 shows the transfer function of only for the data between the third and fourth flat-spot, and Figure 5.9 show the INL computed only on this data. This shows of a triangular pattern of four or five different linear segments, as well as some larger deviations.

Figure 5.10 shows the peak-to-peak noise of ADC2. The highest deviations in both directions happened at the third voltage step, which lines up with the first flat spot in the transfer function.

5.3.3 Second-Order $\Delta\Sigma$ ADC with FIR-Filter (ADC3)

Figure 5.11 shows the transfer function of ADC3. This ADC was implemented with equal input- and feedback-resistors, meaning the duty-cycle goes to zero and $100\,\%$ when the input voltage hits the rails. This causes some edge-effects close to the range limits, which can be seen as deviations from the linear relationship at both ends of the graph.

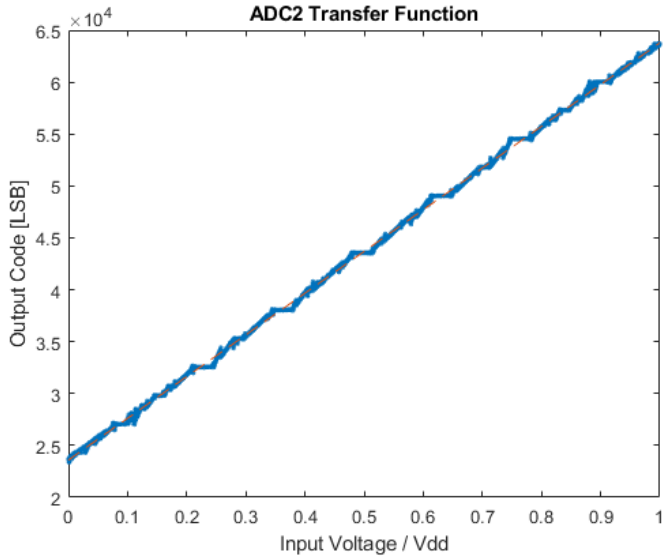


Figure 5.6: Transfer function for $\Delta\Sigma$ ADC 2. The dashed orange line is the best-fit regression line used to calculate INL.

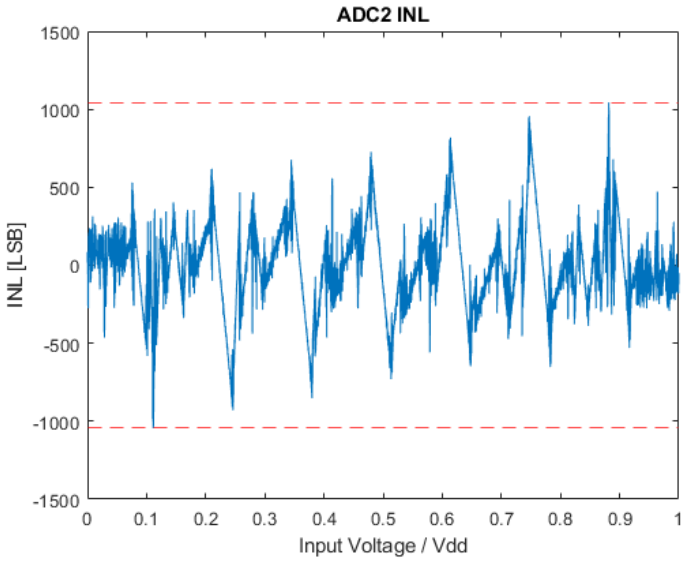


Figure 5.7: Plot of the INL for $\Delta\Sigma$ ADC 2. The red dashed lines show the range of the INL from -1044.2 LSB, to 1043.7 LSB.

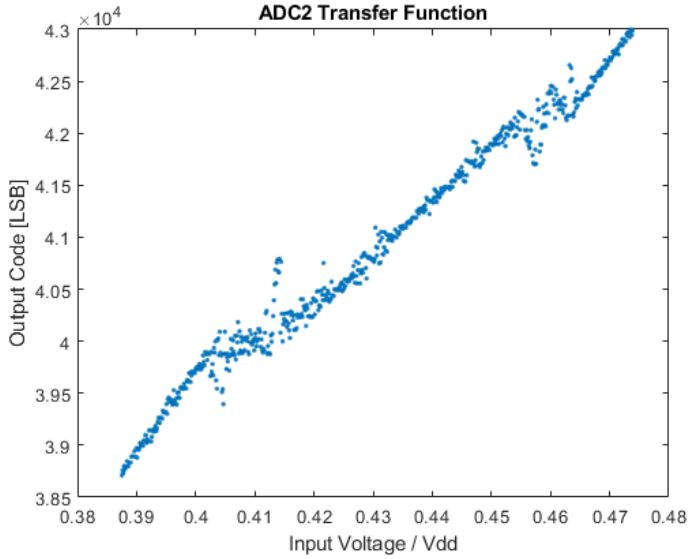


Figure 5.8: Transfer function for $\Delta\Sigma$ ADC 2 between the third and fourth flat-spot.

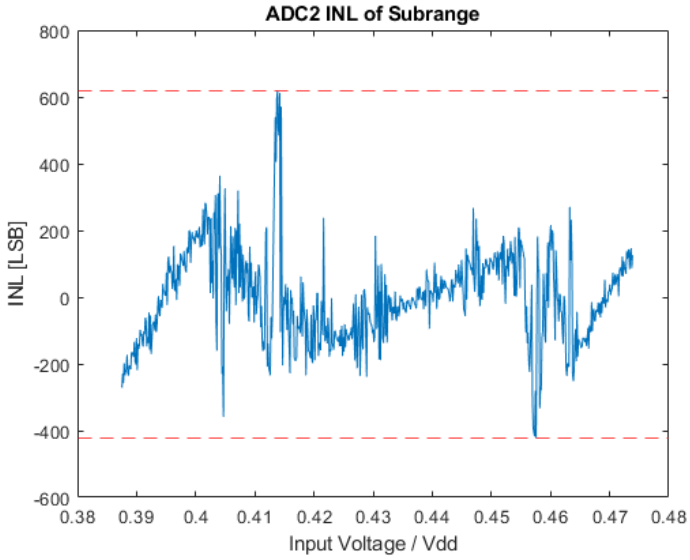


Figure 5.9: Plot of the INL for $\Delta\Sigma$ ADC 2, computed for the subrange of inputs shown in Figure 5.8. The red dashed lines show the range of the INL from -423.12 LSB, to 618.52 LSB.

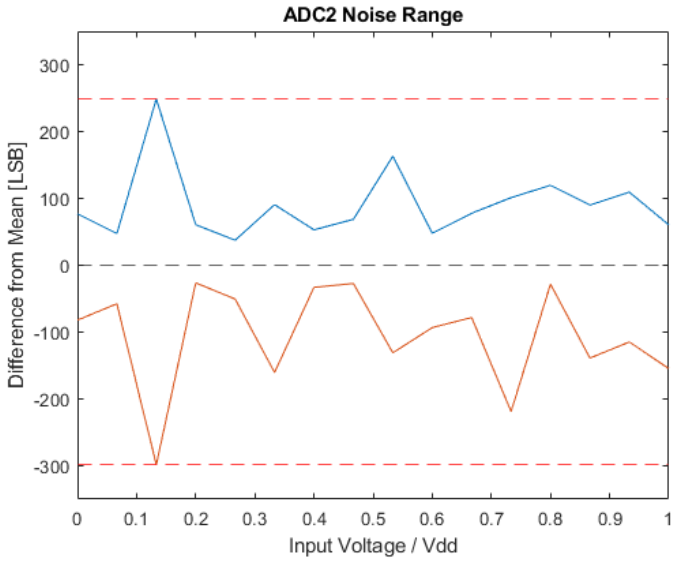


Figure 5.10: Plot of the noise for $\Delta\Sigma$ ADC 2. The red dashed lines show the lowest deviation at -298.35 LSB, and the highest deviation at 248.65 LSB.

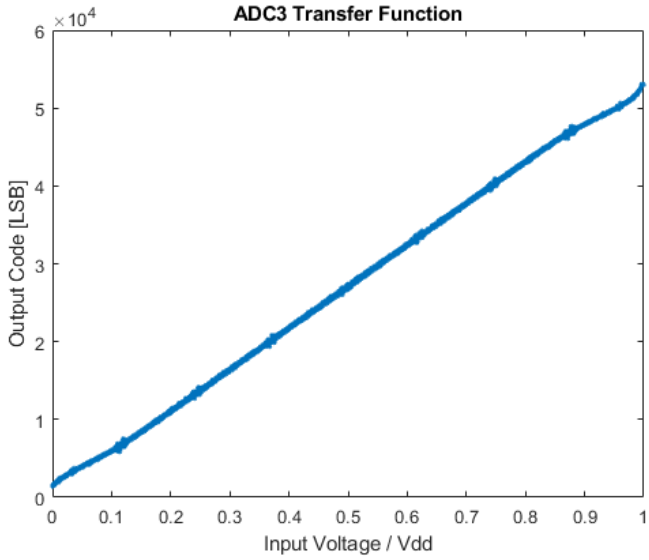


Figure 5.11: Transfer function for $\Delta\Sigma$ ADC 3.

Figure 5.12 shows the same transfer function, but now trimmed to avoid the problematic saturation-regions. The resulting plot shows what looks like a good linear correlation, but similarly to ADC2, there are regular disturbances along the range. In this case however, the disturbances do not seem to be as bad as the flat-spots in ADC2, and the transfer function between them looks better.

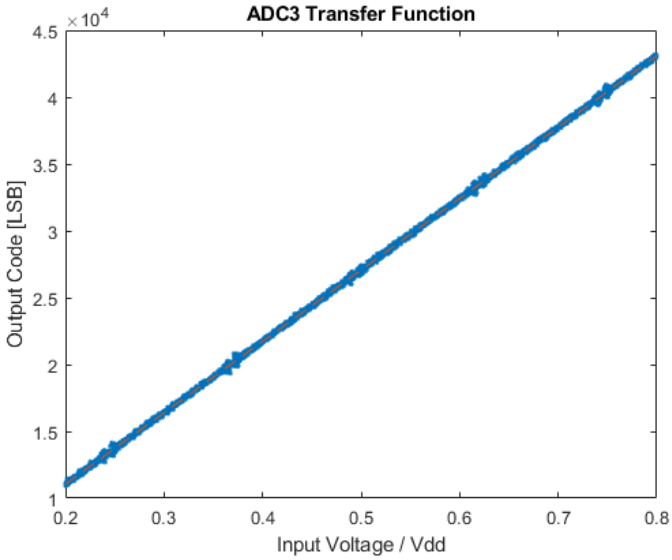


Figure 5.12: Transfer function for $\Delta\Sigma$ ADC 3 with the saturation-effects close to the rails trimmed off. The dashed orange line is the best-fit regression line used to calculate INL.

Figure 5.13 shows the INL of ADC3, trimmed to avoid the saturation regions, and Figure 5.14 shows the noise level for repeated sampling of a fixed voltage. Again, it seems that the noise has a peak-to-peak amplitude of 400 to 500 LSB, washing out any observable non-linearity. In addition, the regular disturbances can be seen in Figure 5.13, causing the maximum deviation from the ideal transfer function to be around ± 650 LSB.

In order to get some data on the effect of the FIR-filter used in this ADC, three runs were done with different filter cut-offs. Figures 5.15a, 5.16a, and 5.17a shows the three different filter-kernels tested, with cut-offs of $f_c = 0.001$, $f_c = 0.004$, and $f_c = 0.015$ respectively, and each with a cosine window function applied. The resulting transfer functions are shown next to these plots.

5.3.4 Dual-Slope ADC (ADC4)

Finally, Figure 5.18 shows the transfer function of ADC4. Due to the method used here, the raw data does not reflect the true result, so some amount of post-processing is required to extract the result. In order to do this, first the outliers at $V_{in} = 0$ and $V_{in} = \frac{V_{DD}}{2}$ were removed, and then the slope in the first half of the data was inverted. Finally, an offset was added to this slope to compensate for the non-zero measurement at $V_{in} = \frac{V_{DD}}{2}$. This

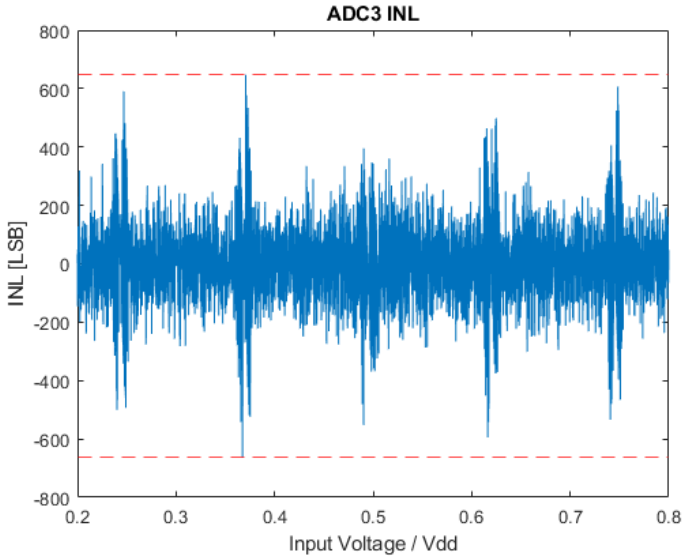


Figure 5.13: Plot of the INL for $\Delta\Sigma$ ADC 3, computed on the trimmed input range. The red dashed lines show the range of the INL from -663.47 LSB, to 647.05 LSB.

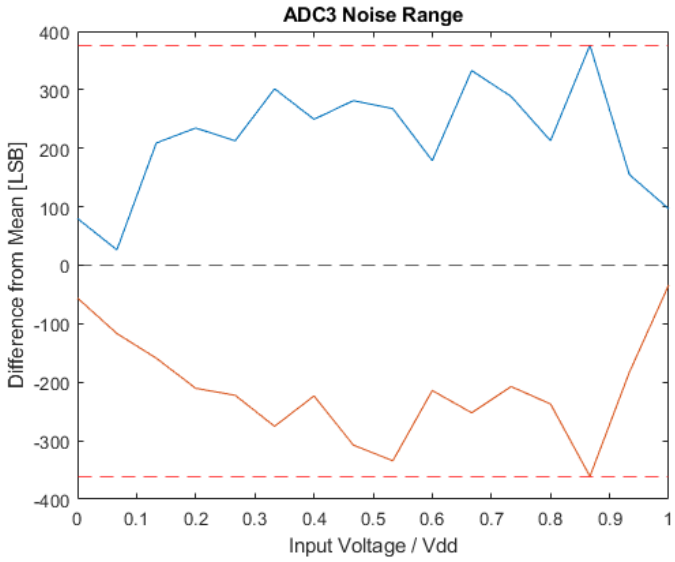
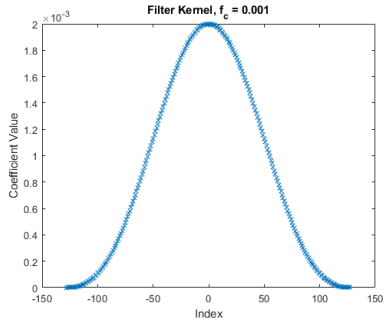
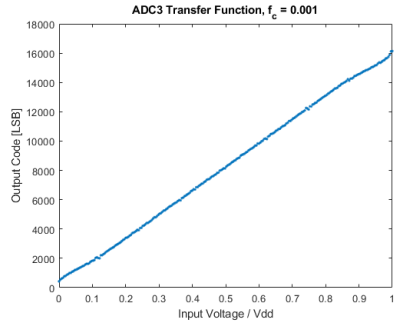


Figure 5.14: Plot of the noise for $\Delta\Sigma$ ADC 3. The red dashed lines show the lowest deviation at -360.96 LSB, and the highest deviation at 376.04 LSB.

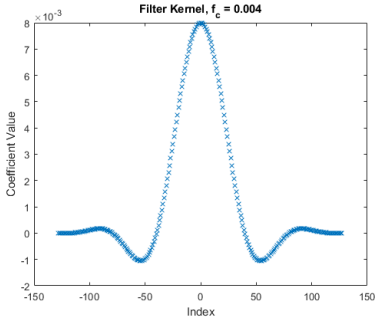


(a) Filter-Coefficients

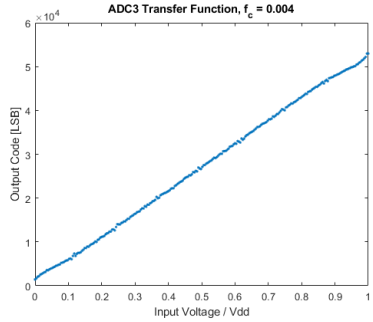


(b) Transfer function

Figure 5.15: Filter Kernel and Transfer Function of ADC3 with $f_c = 0.001$.

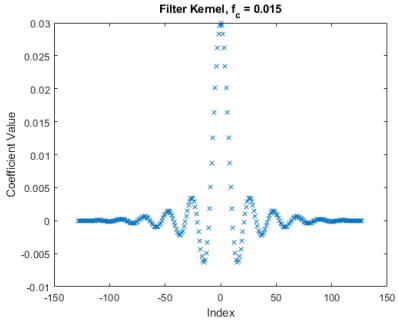


(a) Filter-Coefficients

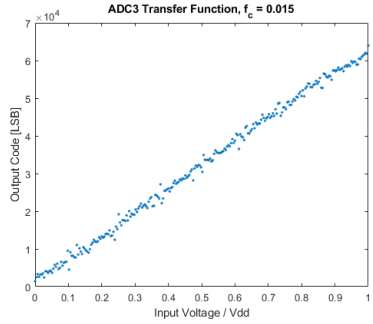


(b) Transfer function

Figure 5.16: Filter Kernel and Transfer Function of ADC3 with $f_c = 0.004$.



(a) Filter-Coefficients



(b) Transfer function

Figure 5.17: Filter Kernel and Transfer Function of ADC3 with $f_c = 0.015$.

offset was calibrated to obtain the smallest possible step in the resulting transfer function. Figure 5.19 shows this transfer function.

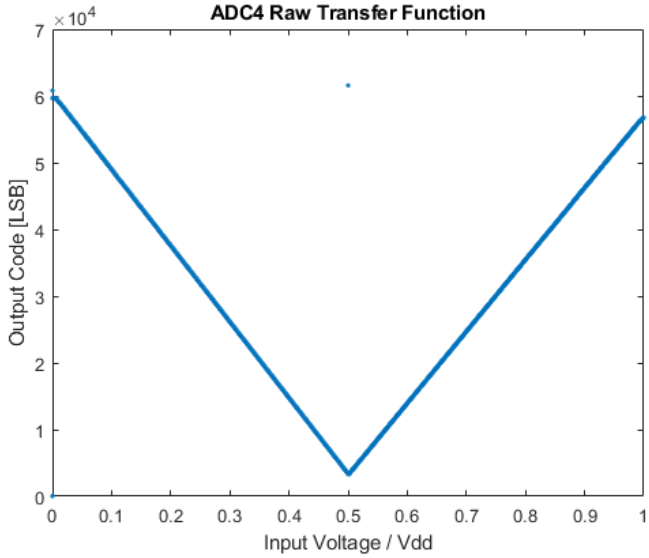


Figure 5.18: Raw transfer function for ΔΣ ADC 4.

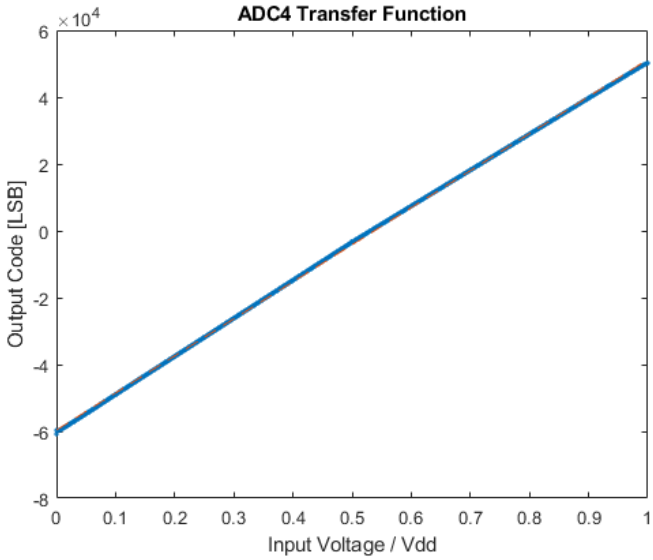


Figure 5.19: Decoded transfer function for ΔΣ ADC 4. The dashed orange line is the best-fit regression line used to calculate INL.

Figure 5.20 shows the INL of the post-processed transfer function of ADC4. This plot shows pretty clearly that there is a significant difference in gain between the bottom and top half of the input range. It can also be seen that there are some minor saturation-effects in the outer limits of the input range, which would be expected as the input-voltage is buffered by an op-amp with limited range.

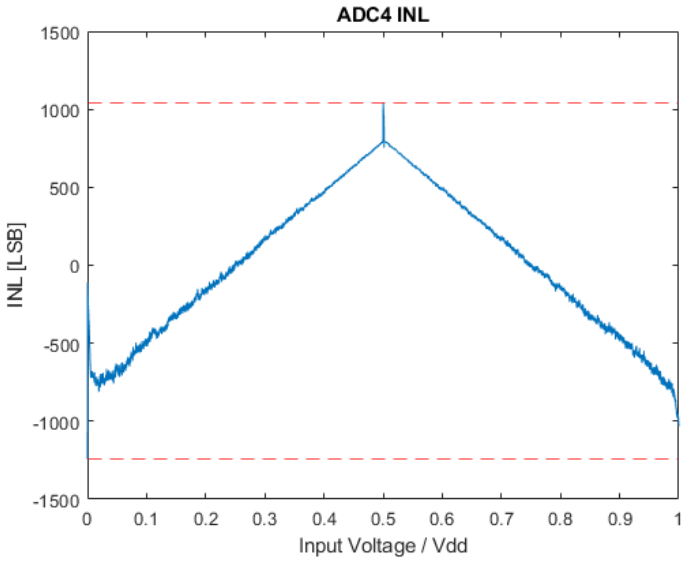
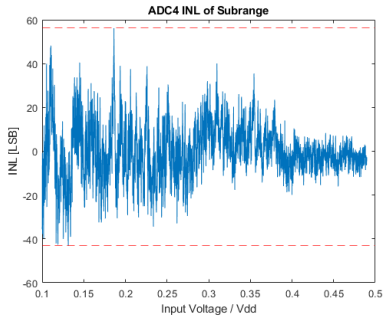


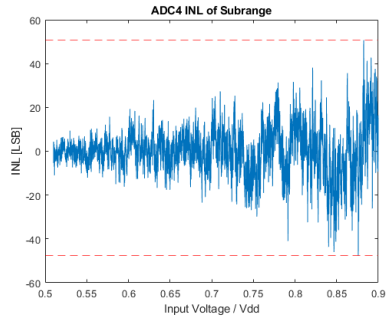
Figure 5.20: Plot of the INL of $\Delta\Sigma$ ADC 4.

To get a better idea of the linearity of each half of the input range, Figures 5.21a and 5.21b shows the INL computed for only the bottom, and top half respectively. The top and bottom of the input range is also trimmed away, as well as the small problematic area just around $\frac{V_{DD}}{2}$. The result is a clear picture of the errors growing bigger as the input voltage moves away from $\frac{V_{DD}}{2}$.

The last plot, Figure 5.22 shows the noise for ADC4. This plot supports the observation that the noise increases as the input moves away from $\frac{V_{DD}}{2}$, increasing from a peak-to-peak of around 30 LSB at the minimum to at most about 140 LSB.



(a) $V_{in} \in (0.1, 0.49)V_{DD}$



(b) $V_{in} \in (0.51, 0.9)V_{DD}$

Figure 5.21: INL of bottom and top ranges of ADC4 computed separately. The red dashed lines show the range of the INL, -43.10 to 56.16 for the low range, and -47.67 to 50.68 for the high range.

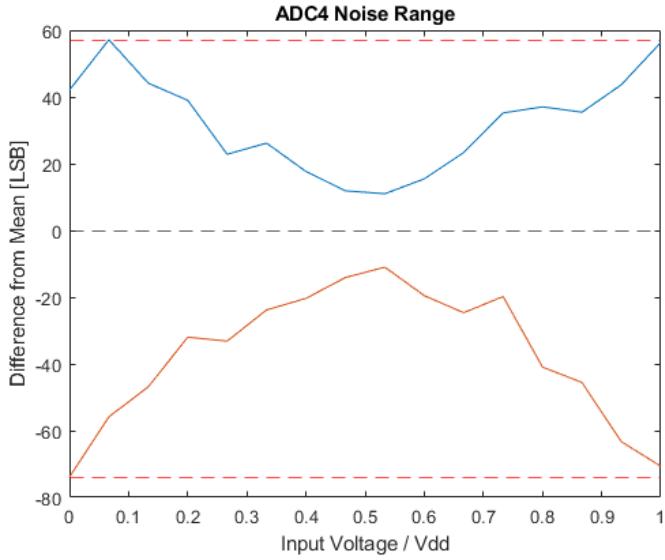


Figure 5.22: Plot of the noise of $\Delta\Sigma$ ADC 4. The red dashed lines show the lowest deviation at -73.93 LSB, and the highest deviation at 57.14 LSB.

Chapter 6

Discussion

6.1 General Performance of ADCs

6.1.1 ADC1

Among the four implemented ADCs, the first-order $\Delta\Sigma$ ADC with a moving average filter was by far the most accurate. The ADC was calibrated to reach almost the full supply-range of the MCU, and the maximum deviation in Figure 5.5 was below 15 LSB. This means that if the four least significant bits of the 16-bit result are discarded, the remaining 12 bits should be correct, giving a 12-bit accurate resolution, matching the on-chip ADC.

6.1.2 ADC2

The first-order $\Delta\Sigma$ ADC with an FIR-filter was the worst performing ADC. As implemented, the maximum deviation in INL was beyond 1000 LSB, meaning that without post-processing, at least the lower 10 bits of the result must be discarded, leaving only 6 accurate bits.

Flat-Spots at Regular Intervals

The major deviance from the ideal transfer function that can be seen in Figure 5.6 is the seven flat-spots that occur at regular intervals. These seem to be related to the fact that the SPI, which is used to capture the bitstream from the $\Delta\Sigma$ modulator, introduces some amount of clock jitter, as described in section 4.2.1. In these tests, the clock jitter only amounted to every eighth clock cycle being one 128th longer, so it was considered probably negligible.

This theory was tested by reducing the prescaler on the SPI, while decreasing the CPU clock to maintain the same modulator frequency. The effect is that the delays between bytes are extended, amplifying the clock jitter. This change dramatically amplified this flat-spot pattern, supporting the hypothesis that this is indeed the cause. Unfortunately,

there is no way of operating the SPI which avoids the clock jitter altogether, meaning the issue is hard to circumvent.

It might be possible to use another serial communication protocol, like the USART, to achieve the same bitstream-capture, but without the clock jitter. This is discussed in section 6.2.4. Alternatively, it might be possible to use the CPU to capture each bit separately, using pin-interrupts and a very lightweight ISR, but this would mean using a large proportion of the CPU-time.

6.1.3 ADC3

Compared to the results from ADC2, the second-order $\Delta\Sigma$ ADC, again with an FIR-filter, performed a bit better. The maximum deviation of the INL would require 9-10 bits to be discarded, but most of this inaccuracy is centered around a few "bad-spots". Away from these spots, it is probably possible to extract an 8-bit result from the ADC.

6.1.4 ADC4

Finally, the dual-slope ADC preformed surprisingly well, though again, not as well as ADC1. Due to the directional method, the ADC returns a 17-bit result, but from the raw INL plot, ten of these bits must be discarded in order to get a monotonic linear measurement in the remaining seven bits.

However, compared to the previous INL-plots, there is an obvious structure, and if this is calibrated to account for the difference in gain between the two halves of the input range, only five or six bits would need to be discarded, as indicated by Figures 5.21a and 5.21b, leaving an ten or eleven bit result.

6.2 Possible Improvements to ADC Implementations

6.2.1 Improved Selection of Resistances, Capacitors, and Operating Frequency

The work in this project focused mainly on implementing proof-of-concept versions of the selected ADC methods, not fine-tuning any single one. While some time was taken to test different parameters, the priority was to complete implementation and testing of all four implementations, so that possibilities and challenges with the methods could be exposed.

If more time is invested into selecting values for the passive components, and selecting clock-frequency, thresholds, settings for the comparators, and other such parameters, better performance can probably be obtained. This could be done by trial-and-error in the lab, but it might also be valuable to create a model of the ADCs in a simulation software, and run optimizations there. This approach would also allow analysis of the effects of component tolerances and drift in the values over time.

6.2.2 Improved FIR-Filter Design

The FIR-filter used in ADC2 and ADC3 was not optimized very well. A few variants were tried, and one that worked was selected, but it was not made a focus of iteration and improvement, other than examining the effects of varying the cutoff-frequency. More attention to this component might extract more performance from ADCs employing the bitstream-capture method.

6.2.3 Decreasing Latency of Software

All the firmware for this project was written in C. While most of the code is simply needed to configure peripherals before they operate without CPU-intervention, there are certain situations where the delay caused by code execution could impact the performance.

One example of this is the ISR which triggers at the end of both slopes of the dual-slope ADC. While this ISR runs, the integrator is still integrating, and for the run-down, the timer is still counting cycles until it is disabled by the code. This causes offsets in both the integrated voltage and the timer result.

Implementing critical parts of the code in assembly would allow exact control of cycle-accurate timing, allowing precise compensation to be applied. This would for example allow a timer to be stopped a few cycles into the ISR, but subsequently reset to the exact value it had when the ISR triggered, which would eliminate this source of error.

6.2.4 USART Peripheral in SPI Mode

Throughout this project, the SPI peripheral was used to capture the bitstream produced by the $\Delta\Sigma$ modulator, but an issue was discovered in the fact that the SPI is unable to perform two transmissions completely back-to-back, but it inserts a single CPU clock cycle in between. This was assumed to be negligible during development, but it turned out to be a probable cause of major error in the ADCs which used this technique.

The microcontroller contains several other serial communication peripherals, which were initially considered, but the SPI was selected as the most suitable. However, the USART module includes a master SPI mode, where it would be capable of performing the same function as the SPI module for these ADC implementations. Basic testing indicates that this peripheral *is* capable of back-to-back transmission, which might be a way of solving this issue.

6.2.5 Using CPU Sleep-Modes to Reduce Noise During Conversion

A common method for reducing ADC noise is to stop the CPU execution while the conversion is taking place [6]. This reduces the noise introduced by the rapidly switching digital logic, providing a calmer environment for the analog electronics. Once conversion is completed, an interrupt can wake up the CPU again, and normal operation can be resumed.

This method could also be utilized for ADC1 and ADC4 in this project. It could possibly also be applicable for ADC2 and ADC3, but the need to keep emptying the `SPI.DATA` register would require the CPU to keep going into and out of the sleep mode. For ADC4,

the CPU only needs to wake up once during the conversion, to prepare for the run-down slope, and for ADC1, the entire conversion can be completed with the CPU in sleep mode.

A requirement for this method is an interrupt to wake the CPU once the conversion is complete. Luckily, all four methods in this project uses interrupts to control the flow of the conversion already, so placing the CPU in sleep mode should not require any major changes. One minor change is that for time-critical interrupts, the wake-up time of the CPU must be compensated for, in addition to the response time of the ISR.

6.2.6 Multi-Bit $\Delta\Sigma$-Modulator

One method which was considered, but not chosen for implementation in this project is the multi-bit $\Delta\Sigma$ ADC. Instead of using a single comparator, producing a 1-bit data stream, two more comparators can be added, along with a CCL-based thermometer-to-binary decoder, producing a 2-bit data stream. The required 2-bit DAC could be made by providing two, differently weighted feedback paths to the integrator, one from each bit.

6.3 Possible Improvements to Characterization Procedure

6.3.1 Measuring Dynamic Signals

An issue with the way the characterization was done during this project is that the ADCs were only tested with static signals. One of the major selling-points of the $\Delta\Sigma$ ADC topology is the ability to move the quantization noise to higher frequencies, and then filter it out, so it seems likely that these ADCs would be more competitive in a frequency-domain test, where signal-to-noise ratio and Effective Number of Bits (ENOB) were measured.

6.3.2 Characterizing Long-Term Drift

DSADC1 was calibrated to give a reasonably low gain- and offset-error. No attempts were made to characterize the drift of this calibration, nor how any other factors, might impact the long-term stability of the ADC.

6.3.3 Characterizing Temperature Dependence

The passive components used in the ADC implementations all have some amount of temperature dependence, which might cause a temperature dependence of the ADC itself. In theory, none of the ADCs in this project depend on the exact component values, as all of them are designed so that the absolute component values cancel out. All the $\Delta\Sigma$ ADCs do however depend on the ratio of the input- and feedback-resistors, and any change to this ratio would directly impact the gain of the ADC. Another source of error is that the op-amp or comparators might change with temperature.

6.4 Alternate Use-Cases

Since none of the ADCs implemented in this project could provide better performance than the on-chip ADC in the AVR128DB MCUs, it is questionable if it would be worthwhile to implement any of them with this microcontroller. However, there might be situations where the techniques used in this project could be more applicable.

6.4.1 Direct Measurement of Higher-Voltage Inputs

All the $\Delta\Sigma$ ADCs in this project feed the input-voltage into an input-resistor, and effectively measures the resulting current. This means that if the value of the input-resistor is increased, the input-range of the ADC can be extended almost indefinitely, only limited by the voltage-rating of the resistor used.

This means that for example, a $12\,\mathrm{V}$ signal could be directly measured by the ADC, without worrying about exceeding the absolute maximum ratings of the chip. With even higher resistances, it might even be possible to directly measure a $230\,\mathrm{V}$ signal. The only requirement is that the current in the input-resistor can be balanced by the current in the feedback-resistor, which is produced by the $\frac{V_{DD}}{2}$ voltage difference across it.

One issue is that before the MCU has configured the modulator, the pin connected to the summing node of the integrator will be high-impedance, which means no current would flow through the input-resistor. This, in turn, means there is no voltage drop, which might cause the pin to exceed the absolute maximum rating, damaging the MCU. However, this can probably be solved by connecting a Zener-diode, or some other protection to this pin.

6.4.2 Implementation in Lower-Cost Applications

While the AVR-DB family of microcontrollers have a 12-bit ADC, there exists other MCUs with integrated op-amps, where the on-chip ADC is not as capable (for example the PIC16F527 [7]). In applications where cost, or other factors, limits the choice and quality of on-chip ADCs, the techniques used in this project might be a good option. It could also be an option to use an external op-amp, in order to implement a $\Delta\Sigma$ ADC with a MCU with very limited peripherals.

Implementation Without Op-Amp

If cost is the main concern, to the extent that neither an on-chip ADC or an internal or external op-amp is available, it might even be possible to implement any of the ADCs in this project without an op-amp integrator. A simple low-pass filter approximates an integrator for small amplitudes [3, p. 26], so with careful design it might be possible to implement a $\Delta\Sigma$ or dual-slope ADC with only a capacitor, two resistors, and a very basic microcontroller.

6.4.3 Enabling Differential Input

The $\Delta\Sigma$ ADCs all use a summing-integrator as the first stage of the circuit. Extending this integrator with an input-resistor connected to the non-negating input, could allow

conversion of differential signals.

Chapter 7

Conclusion

This project has proposed, designed, and tested four methods of implementing ADCs using the OPAMP peripheral present in the AVR-DB family of microcontrollers. Three of these were $\Delta\Sigma$ ADCs, and one was a dual-slope ADC.

Overall, the alternate ADCs in this project were not able to match the on-chip SAR ADC in any of the measured metrics. The first-order $\Delta\Sigma$ ADC with a moving average filter came closest, giving 12 accurate bits, but the conversion time of this ADC is slightly more than $2^{16} = 65536$ clock cycles, while the on-chip ADC typically only uses around 16 cycles.

The technique of using the SPI peripheral to capture the bitstream produced by a $\Delta\Sigma$ modulator did not produce any better results than a simple moving average filter, although the conversion time was much lower. However, as the performance was only tested for static input-voltages, there might be an unobserved advantage with this technique for dynamic signals.

Without further development, these implementations do not satisfy the original goal of implementing an ADC with higher resolution than the on-chip SAR ADC. There may however be applications for the techniques used in this project in other niches, such as if very high input-voltages need to be measured, or if an on-chip ADC is not available, or less capable than the 12-bit ADC present in the AVR-DB family of MCUs.

Finally, compared to the on-chip ADC, the implementations described in this project has the disadvantage that they introduce a significant amount of complexity, and use system resources, both in terms of CPU time and peripherals. However, in some applications, the customizability of these techniques could prove advantageous, possibly making them a viable alternative.

7.1 Future Work

7.1.1 Implementation on Less Capable MCUs

Implementing a dual-slope or $\Delta\Sigma$ ADC using the techniques in this project might make more sense with less capable microcontrollers. Therefore, it would be interesting to see what the minimum required capabilities are, and if this could be a viable alternative for low-cost applications.

7.1.2 Improvement of the Bitstream-Capture Technique

In the AVR-DB, the bitstream-capture method could work better by using the USART in SPI mode instead of the SPI peripheral. This could improve the two implementations in this project which uses this technique. Additionally, the frequency-response of these ADCs could also be characterized, to see if they have more strengths for dynamic signals.

Bibliography

[1] National Instruments Corporation. Ni pxie-1062q user manual, 2012. Available at https://www.ni.com/pdf/manuals/371843d.pdf.

[2] National Instruments Corporation. Ni 6289 device specifications, 2016. Available at https://www.ni.com/pdf/manuals/375222c.pdf.

[3] Paul Horowitz and Winfield Hill. *The art of electronics; 3rd ed.* Cambridge University Press, Cambridge, 2015.

[4] Analog Devices Inc. Adc input noise: The good, the bad, and the ugly. is no noise good noise?, 2006. Available at https://www.analog.com/en/analog-dialogue/articles/adc-input-noise.html.

[5] Analog Devices Inc. Adc requirements for rtc temperature measurement systems, 2015. Available at https://www.analog.com/en/technical-articles/adc-requirements-for-rtc-temperature-measurement-systems.html.

[6] Microchip Technology Inc. Avr® adc noise reduction mode. Available at https://microchipdeveloper.com/8avr:adcnoisereduce.

[7] Microchip Technology Inc. Pic16f527 datasheet, 2016. Available at http://ww1.microchip.com/downloads/en/DeviceDoc/40001652D.pdf.

[8] Microchip Technology Inc. Adc dc specifications, 2020. Available at https://microchipdeveloper.com/adc:adc-dc-spec.

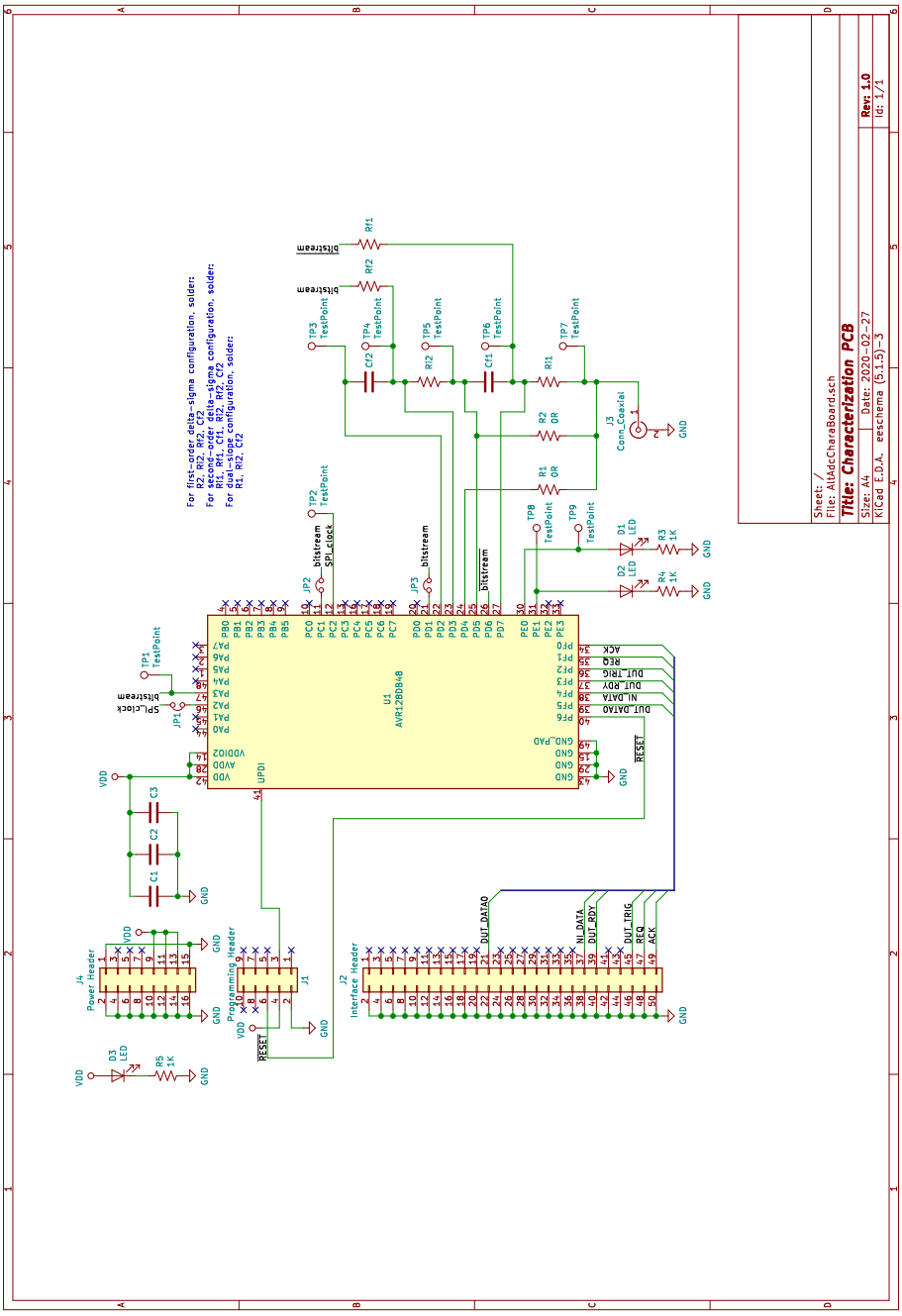
[9] Microchip Technology Inc. Adc integral non-linearity (inl), 2020. Available at https://microchipdeveloper.com/adc:adc-inl.

[10] Microchip Technology Inc. Avr128db48 datasheet, 2020. Available from https://www.microchip.com/wwwproducts/en/AVR128DB48.

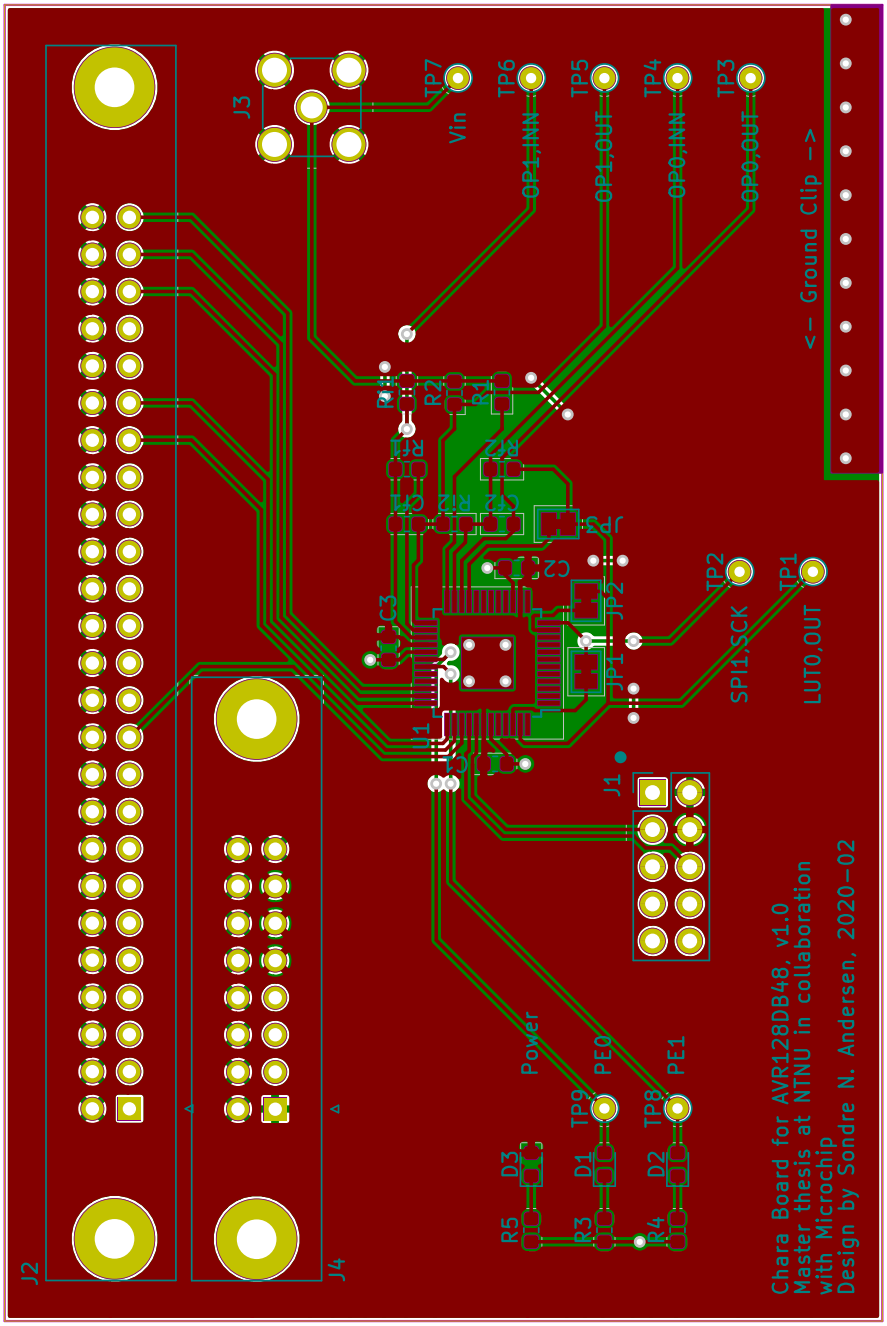
[11] W. Kester. Which adc architecture is right for your application?, jun 2005. Available at https://www.analog.com/en/analog-dialogue/articles/the-right-adc-architecture.html.

Appendix

7.2 Appendix A



7.3 Appendix B



7.4 Appendix C

7.4.1 Matlab Code for Analyzing Transfer Function

```
% Analyze transfer function
close all;
clear;

% Read chara data
filename = "RawData/DSADC1_8096x1.csv";
T = readmatrix(filename);
Steps = length(T(:,1));
inputs = T(:,1) / (Steps - 1);
outputs = T(:,2);

% Trim ends if neseccary
range_start = 0.0 * Steps;
range_end = 1.0 * Steps;
inputs = inputs(range_start+1:range_end);
outputs = outputs(range_start+1:range_end);
Steps = length(inputs);

% Compute linear regression best fit line
X = [ones(Steps,1) inputs];
B = X\outputs;

% Compute INL as difference between data and best-fit line
best_fit_line = X * B;
INL = outputs - best_fit_line;

% Generate plots
figure;
plot(inputs, outputs, '.');
hold on;
plot(inputs, best_fit_line, '--');
title('ADC1 Transfer Function');
xlabel('Input Voltage / Vdd');
ylabel('Output Code [LSB]');

figure;
plot(inputs, INL);
yline(max(INL), 'r--');
yline(min(INL), 'r--');
title('ADC1 INL');
xlabel('Input Voltage / Vdd');
ylabel('INL [LSB]');
```

7.4.2 Matlab Code for Analyzing Transfer Function of ADC 4

```
% Analyze transfer function of ADC4
close all;
clear;

% Read chara data
filename = "RawData/DSADC4_8096x1.csv";
T = readmatrix(filename);
```

```
Steps = length(T(:,1));
inputs = T(:,1) / (Steps - 1);
outputs = T(:,2);

% Generate plot of raw data
figure;
plot(inputs, outputs, '.');
title('ADC4 Raw Transfer Function');
xlabel('Input Voltage / Vdd');
ylabel('Output Code [LSB]');

% Decode directionality
HighRangeOffset = -6540;
outputs_low = -outputs(1:Steps/2);
inputs_low = inputs(1:Steps/2);
outputs_high = outputs(Steps/2+1:end) + HighRangeOffset;
inputs_high = inputs(Steps/2+1:end);

% Remove outliers
outputs_low(1) = [];
inputs_low(1) = [];
outputs_high(2) = [];
inputs_high(2) = [];
Steps = Steps - 2;

outputs = [outputs_low; outputs_high];
inputs = [inputs_low; inputs_high];

% Trim ends
range_start = floor(0.1 * Steps);
range_end = floor(0.49 * Steps);
inputs = inputs(range_start+1:range_end);
outputs = outputs(range_start+1:range_end);
Steps = length(inputs);

% Compute linear regression best fit line
X = [ones(Steps,1) inputs];
B = X\outputs;

% Compute INL as difference between data and best-fit line
best_fit_line = X * B;
INL = outputs - best_fit_line;

% Generate plots
figure;
plot(inputs, outputs, '.');
hold on;
plot(inputs, best_fit_line, '--');
title('ADC4 Transfer Function');
xlabel('Input Voltage / Vdd');
ylabel('Output Code [LSB]');

figure;
plot(inputs, INL);
yline(max(INL), 'r--');
yline(min(INL), 'r--');
title('ADC4 INL of Subrange');
```

```
xlabel('Input Voltage / Vdd');
ylabel('INL [LSB]');
```

7.4.3 Matlab Code for Analyzing Noise Range

```
% Analyze noise
close all;
clear;

% Parameters for the data acquisition
Steps = 16;
SamplesPrStep = 255;
StartSample = 3;

% Read chara data
filename = "RawData/DSADC1_16x255.csv";
T = readmatrix(filename);
data_length = length(T(:,1));
inputs = (0:15) / 15;
outputs = T(:,2);
outputs = reshape(outputs, SamplesPrStep, Steps);

% Compute statistics for each step
max_deviation = zeros(Steps,1);
min_deviation = zeros(Steps,1);
midpoint = zeros(Steps,1);
for i = 1:Steps
    midpoint(i) = mean(outputs(StartSample:end,i));
    max_deviation(i) = max(outputs(StartSample:end,i)) - midpoint(i);
    min_deviation(i) = min(outputs(StartSample:end,i)) - midpoint(i);
end

% Generate plot
plot(inputs, max_deviation);
hold on;
plot(inputs, min_deviation);
yline(0, '--');
yline(max(max_deviation), 'r--');
yline(min(min_deviation), 'r--');
title('ADC1 Noise Range');
xlabel('Input Voltage / Vdd');
ylabel('Difference from Mean [LSB]');
```

7.5 Appendix D

7.5.1 C-Code for ADC1

```
/*
 * adc1.c
 * Delta-Sigma ADC implementation with rectangular moving average filter.
 */

#include "adc1.h"
#include <avr/io.h>
#include <avr/interrupt.h>

volatile static uint16_t last_result;
volatile static bool conversion_complete = false;
volatile static uint8_t tca_overflows;

#define OVERCOUNTING_N 3000

void dsadc_init_modulator(void)
{
  // Configure OPAMP0 to integrator
  //   Using OP0,INN (PD3, pin 9) as the virtual zero node
  //   Occupying OP0,OUT (PD2, pin 8) on same pin as AC0,AINP0
  OPAMP.OP0CTRLA = OPAMP_OUTMODE_NORMAL_gc | OPAMP_ALWAYSON_bm;
  OPAMP.OP0INMUX = OPAMP_MUXPOS_VDDDIV2_gc | OPAMP_MUXNEG_INN_gc;
  OPAMP.OP0CAL = 143;

  // Enable op-amps
  OPAMP.CTRLA = OPAMP_ENABLE_bm;

  // Configure AC0 to threshold output from OP0 against Vdd/2
  //   Occupying AC0,AINP0 (PD2, pin 8) on same pin as OP0,OUT
  VREF.ACREF = VREF_REFSEL_VDD_gc;
  AC0.MUXCTRL = AC_MUXPOS_AINP0_gc | AC_MUXNEG_DACREF_gc;
  AC0.DACREF = 0x80;
  AC0.CTRLA = AC_HYSMODE_NONE_gc | AC_ENABLE_bm;

  // Configure LUT0 to sample-and-hold the output from AC0 with with
    CLK_PER as clock source
  //   Using LUT0,OUT (PA3, pin 25) as the output
  CCL.TRUTH0 = 0b00000010;
  CCL.LUT0CTRLC = CCL_INSEL1_MASK_gc;
  CCL.LUT0CTRLB = CCL_INSEL1_MASK_gc | CCL_INSEL0_AC0_gc;
  CCL.LUT0CTRLA = CCL_OUTEN_bm | CCL_CLKSRC_CLKPER_gc |
    CCL_FILTSEL_FILTER_gc;

  // Configure LUT1 to always output "1"
  CCL.TRUTH1 = 0xFF;
  // Configure sequencer for LUT0 and LUT1
  CCL.SEQCTRL0 = CCL_SEQSEL0_DFF_gc;
  // Configure LUT0,OUT to output
  PORTA.DIRSET = (1 << 3);
  // Enable CCL
  CCL.LUT1CTRLA |= CCL_ENABLE_bm;
  CCL.LUT0CTRLA |= CCL_ENABLE_bm;
  CCL.CTRLA = CCL_ENABLE_bm;
```

```
52
53    // Configure event channel 0 to carry the output from LUT0 to TCA
54    EVSYS.CHANNEL0 = EVSYS_CHANNEL2_CCL_LUT0_gc;
55    EVSYS.USERTCA0CNTA = EVSYS_USER_CHANNEL0_gc;
56
57    // Configure TCA to count clock cycles while EVENTA is high, handle the
        overflow
58    // interrupt to detect overflows
59    TCA0.SINGLE.PER = UINT16_MAX;
60    TCA0.SINGLE.EVCTRL = TCA_SINGLE_EVACTA_CNT_HIGHLVL_gc |
        TCA_SINGLE_CNTAEI_bm;
61    TCA0.SINGLE.INTCTRL = TCA_SINGLE_OVF_bm;
62
63    // Configure TCB to give an interrupt after about 2^16 cycles
64    // The -61 is because the interrupt handler takes some time to launch
65    TCB0.CTRLA = TCB_CLKSEL_DIV2_gc;
66    TCB0.CCMP = 0x7FFFU - 61 + OVERCOUNTING_N;
67    TCB0.INTCTRL = TCB_CAPT_bm;
68  }
69
70  void dsadc_start_conversion(void)
71  {
72    // Reset state variables, start TCA "below" 0 to allow a non-zero duty-
        cycle at 0V input
73    tca_overflows = 0;
74    conversion_complete = false;
75    TCA0.SINGLE.CNT = 0xFFFF - OVERCOUNTING_N;
76    TCB0.CNT = 0;
77
78    // Enable timer counters to start conversion
79    TCB0.CTRLA |= TCB_ENABLE_bm;
80    TCA0.SINGLE.CTRLA |= TCA_SINGLE_ENABLE_bm;
81  }
82
83  bool dsadc_conversion_complete(void)
84  {
85    return conversion_complete;
86  }
87
88  uint16_t dsadc_get_result(void)
89  {
90    // In normal cases, TCA should have overflowed exactly once. If it has
        overflowed 0 or 2 times,
91    // this indicates an out-of-range input
92    if (tca_overflows < 1) {
93      return 0xFFFF;
94    }
95    if (tca_overflows > 1) {
96      return 0x0;
97    }
98
99    return UINT16_MAX - last_result;
100 }
101
102 ISR(TCA0_OVF_vect) {
103   // Count overflows of TCA
104   tca_overflows++;
```

```
  TCA0.SINGLE.INTFLAGS = TCA_SINGLE_OVF_bm;
}

ISR(TCB0_INT_vect)
{
  asm volatile("nop");
  asm volatile("nop");
  asm volatile("nop");

  // Get result
  last_result = TCA0.SINGLE.CNT;
  conversion_complete = true;

  // Stop timer/counters
  TCB0.CTRLA &= ~TCB_ENABLE_bm;
  TCA0.SINGLE.CTRLA &= ~TCA_SINGLE_ENABLE_bm;

  TCB0.INTFLAGS = TCB_CAPT_bm;
}
```

7.5.2 C-Code for ADC2

```
/*
 * adc2.c
 * Second order Delta-Sigma ADC implementation with rectangular moving
    average
 * filter.
 */

#include "adc2.h"
#include "fir_filter.h"
#include <avr/io.h>
#include <avr/interrupt.h>

#define FILTER_KERNEL_Fc 0.004

#define BITSTREAM_LENGTH_BYTES (FIR_FILTER_KERNEL_SIZE / 8)

static volatile uint8_t bitstream[BITSTREAM_LENGTH_BYTES] = {0};
static volatile uint8_t byte_index = 0;

void dsadc2_init_modulator(void)
{
  // Initialize the filter kernel
  fir_init_kernel(FILTER_KERNEL_Fc);

  // Configure OPAMP0 to integrator
  OPAMP.OP0CTRLA = OPAMP_OUTMODE_NORMAL_gc | OPAMP_ALWAYSON_bm;
  OPAMP.OP0INMUX = OPAMP_MUXPOS_VDDDIV2_gc | OPAMP_MUXNEG_INN_gc;
  OPAMP.OP0CAL = 143;

  // Enable op-amp
  OPAMP.CTRLA = OPAMP_ENABLE_bm;

  // Configure AC0 as quantizer
  VREF.ACREF = VREF_REFSEL_VDD_gc;
  AC0.MUXCTRL = AC_MUXPOS_AINP0_gc | AC_MUXNEG_DACREF_gc;
```

```
35   AC0.DACREF = 0x80;
36   AC0.CTRLA = AC_HYSMODE_NONE_gc | AC_ENABLE_bm;
37
38   // Configure LUT0/1 as DFF
39   PORTA.DIRSET = PIN3_bm;
40   CCL.TRUTH0 = 0b00100010;
41   CCL.LUT0CTRLC = CCL_INSEL2_IN2_gc;
42   CCL.LUT0CTRLB = CCL_INSEL1_MASK_gc | CCL_INSEL0_AC0_gc;
43   CCL.LUT0CTRLA = CCL_OUTEN_bm | CCL_CLKSRC_IN2_gc |
       CCL_FILTSEL_DISABLE_gc;
44   CCL.TRUTH1 = 0xFF;
45   CCL.SEQCTRL0 = CCL_SEQSEL0_DFF_gc;
46   CCL.LUT0CTRLA |= CCL_ENABLE_bm;
47   CCL.LUT1CTRLA |= CCL_ENABLE_bm;
48   CCL.CTRLA = CCL_ENABLE_bm;
49
50   // Configure SPI for bitstream capture
51   PORTC.DIRSET = PIN2_bm;
52   SPI1.CTRLA = SPI_MASTER_bm | SPI_DORD_bm | SPI_PRESC_DIV128_gc;
53   SPI1.CTRLB = SPI_BUFEN_bm | SPI_SSD_bm | SPI_MODE_1_gc;
54   SPI1.INTCTRL = SPI_RXCIE_bm;
55   SPI1.CTRLA |= SPI_ENABLE_bm;
56
57   // Set byte-index past the end of the array, indicating that no
       conversion is in progress
58   byte_index = BITSTREAM_LENGTH_BYTES;
59 }
60
61 void dsadc2_start_conversion(void)
62 {
63   // Set byte_index to the start of the array to start the conversion
64   byte_index = 0;
65 }
66
67 bool dsadc2_conversion_complete(void)
68 {
69   // Conversion is done once the byte_index reaches the end of the array
70   return (byte_index >= BITSTREAM_LENGTH_BYTES);
71 }
72
73 uint16_t dsadc2_get_result(void)
74 {
75   return fir_filter_bitstream(bitstream);
76 }
77
78 ISR(SPI1_INT_vect)
79 {
80   uint8_t data = SPI1.DATA;
81   SPI1.DATA = 0xFF;
82
83   if (byte_index < BITSTREAM_LENGTH_BYTES) {
84     bitstream[byte_index++] = data;
85   }
86
87   if (byte_index >= BITSTREAM_LENGTH_BYTES / 2) {
88     // Take in bits in the second half of the bitstream in reverse order
89     SPI1.CTRLA &=  ~SPI_DORD_bm;
```

```
  }

  SPI1.INTFLAGS = SPI_RXCIF_bm;
}
```

7.5.3 C-Code for ADC3

```
/*
 * adc3.c
 * Second order Delta-Sigma ADC implementation with FIR filter.
 */

#include "adc3.h"
#include "fir_filter.h"
#include <avr/io.h>
#include <avr/interrupt.h>

#define FILTER_KERNEL_Fc 0.004

#define BITSTREAM_LENGTH_BYTES (FIR_FILTER_KERNEL_SIZE / 8)

static volatile uint8_t bitstream[BITSTREAM_LENGTH_BYTES] = {0};
static volatile uint8_t byte_index = 0;

void dsadc3_init_modulator(void)
{
  // Initialize the filter kernel
  fir_init_kernel(FILTER_KERNEL_Fc);

  // Set byte-index past the end of the array, indicating that no
    conversion is in progress
  byte_index = BITSTREAM_LENGTH_BYTES;

  // Configure OPAMP1 as integrator
  OPAMP.OP1CTRLA = OPAMP_OUTMODE_NORMAL_gc | OPAMP_ALWAYSON_bm;
  OPAMP.OP1INMUX = OPAMP_MUXPOS_VDDDIV2_gc | OPAMP_MUXNEG_INN_gc;

  // Configure OPAMP0 as integrator
  OPAMP.OP0CTRLA = OPAMP_OUTMODE_NORMAL_gc | OPAMP_ALWAYSON_bm;
  OPAMP.OP0INMUX = OPAMP_MUXPOS_VDDDIV2_gc | OPAMP_MUXNEG_INN_gc;

  // Enable op-amps
  OPAMP.CTRLA = OPAMP_ENABLE_bm;

  // Configure AC0 as quantizer
  VREF.ACREF = VREF_REFSEL_VDD_gc;
  AC0.MUXCTRL = AC_MUXPOS_AINP0_gc | AC_MUXNEG_DACREF_gc;
  AC0.DACREF = 0x80;
  AC0.CTRLA = AC_HYSMODE_NONE_gc | AC_ENABLE_bm;

  // Configure LUT0/1 as DFF and LUT2 as inverter
  PORTA.DIRSET = PIN3_bm;
  PORTMUX.CCLROUTEA |= PORTMUX_LUT2_ALT1_gc;
  PORTD.DIRSET = PIN6_bm;
  CCL.TRUTH0 = 0b00100010;
  CCL.LUT0CTRLC = CCL_INSEL2_IN2_gc;
  CCL.LUT0CTRLB = CCL_INSEL1_MASK_gc | CCL_INSEL0_AC0_gc;
```

```
50    CCL.LUT0CTRLA = CCL_OUTEN_bm | CCL_CLKSRC_IN2_gc |
        CCL_FILTSEL_DISABLE_gc;
51    CCL.TRUTH1 = 0b11111111;
52    CCL.SEQCTRL0 = CCL_SEQSEL0_DFF_gc;
53    CCL.TRUTH2 = 0b00000001;
54    CCL.LUT2CTRLC = CCL_INSEL2_MASK_gc;
55    CCL.LUT2CTRLB = CCL_INSEL1_IN1_gc | CCL_INSEL0_MASK_gc;
56    CCL.LUT2CTRLA = CCL_OUTEN_bm;
57    CCL.LUT0CTRLA |= CCL_ENABLE_bm;
58    CCL.LUT1CTRLA |= CCL_ENABLE_bm;
59    CCL.LUT2CTRLA |= CCL_ENABLE_bm;
60    CCL.CTRLA = CCL_ENABLE_bm;
61
62    // Configure SPI for bitstream capture
63    PORTC.DIRSET = PIN2_bm;
64    SPI1.CTRLA = SPI_MASTER_bm | SPI_DORD_bm | SPI_PRESC_DIV128_gc;
65    SPI1.CTRLB = SPI_BUFEN_bm | SPI_SSD_bm | SPI_MODE_1_gc;
66    SPI1.INTCTRL = SPI_DREIE_bm;
67    SPI1.CTRLA |= SPI_ENABLE_bm;
68
69    // Send data to start modulator
70    //SPI1.DATA = 0xFF;
71  }
72
73  void dsadc3_start_conversion(void)
74  {
75    // Set byte_index to the start of the array to start the conversion
76    byte_index = 0;
77  }
78
79
80  bool dsadc3_conversion_complete(void)
81  {
82    // Conversion is done once the byte_index reaches the end of the array
83    return (byte_index >= BITSTREAM_LENGTH_BYTES);
84  }
85
86
87  uint16_t dsadc3_get_result(void)
88  {
89    return fir_filter_bitstream(bitstream);
90  }
91
92  ISR(SPI1_INT_vect)
93  {
94    uint8_t data = SPI1.DATA;
95    SPI1.DATA = 0xFF;
96
97    if (byte_index < BITSTREAM_LENGTH_BYTES) {
98      bitstream[byte_index++] = data;
99    }
100
101   if (byte_index >= BITSTREAM_LENGTH_BYTES / 2) {
102     // Take in bits in the second half of the bitstream in reverse order
103     SPI1.CTRLA &=  ~SPI_DORD_bm;
104   }
105
```

```
  SPI1.INTFLAGS = SPI_DREIF_bm;
}
```

7.5.4 C-Code for ADC4

```
/*
 * adc4.c
 * Dual slope ADC implementation.
 */

#include "adc4.h"
#include <avr/io.h>
#include <avr/interrupt.h>

volatile bool integral_direction_up;
volatile bool conversion_done = false;
volatile uint16_t conversion_result = 0;

void dual_slope_init_modulator(void)
{
  // Enable DAC
  VREF.DAC0REF = VREF_REFSEL_VDD_gc;
  DAC0.CTRLA = DAC_OUTEN_bm;
  DAC0.DATA = 0x200 << DAC_DATA_gp; // Vdd/2
  DAC0.CTRLA |= DAC_ENABLE_bm;

  // Configure OPAMP1 to buffer
  //   Using OP1,INP (PD4, pin 10) as the input
  //   Using OP1,OUT (PD5, pin 11) as the output
  OPAMP.OP1CTRLA = OPAMP_OUTMODE_NORMAL_gc | OPAMP_ALWAYSON_bm;
  OPAMP.OP1INMUX = OPAMP_MUXPOS_VDDDIV2_gc | OPAMP_MUXNEG_OUT_gc;

  // Configure OPAMP0 to integrator (shunted in the idle configuration)
  //   Using OP0,INN (PD3, pin 9) as the virtual zero node
  //   Occupying OP0,OUT (PD2, pin 8) on same pin as AC0,AINP0
  OPAMP.OP0CTRLA = OPAMP_OUTMODE_NORMAL_gc | OPAMP_ALWAYSON_bm;
  OPAMP.OP0INMUX = OPAMP_MUXPOS_VDDDIV2_gc | OPAMP_MUXNEG_OUT_gc;

  // Enable OPAMPs
  OPAMP.CTRLA |= OPAMP_ENABLE_bm;


  // Configure AC0 to threshold OP0,OUT against DACREF
  VREF.ACREF = VREF_REFSEL_VDD_gc;
  AC0.MUXCTRL = AC_MUXPOS_AINP0_gc | AC_MUXNEG_DACREF_gc;
  AC0.CTRLA |= AC_ENABLE_bm | AC_OUTEN_bm;


  // Configure TCA
  TCA0.SINGLE.CTRLA = TCA_SINGLE_CLKSEL_DIV1_gc;
  TCA0.SINGLE.CTRLB = TCA_SINGLE_CMP0EN_bm;
  TCA0.SINGLE.CMP0 = 3200;


  // Enable PD6 as output (for debugging)
  PORTA.DIRSET = PIN6_bm;
}
```

```
53
54 void dual_slope_start_conversion(void)
55 {
56   PORTA.OUTSET = PIN6_bm;
57   conversion_done = false;
58
59   // MUX in the input
60   OPAMP.OP1INMUX = OPAMP_MUXPOS_INP_gc | OPAMP_MUXNEG_OUT_gc;
61   // Enable integrator
62   OPAMP.OP0INMUX = OPAMP_MUXPOS_VDDDIV2_gc | OPAMP_MUXNEG_INN_gc;
63
64   // Start TCA
65   TCA0.SINGLE.CNT = 0;
66   TCA0.SINGLE.INTCTRL = TCA_SINGLE_CMP0_bm;
67   TCA0.SINGLE.CTRLA |= TCA_SINGLE_ENABLE_bm;
68
69   // Set DACREF to Vdd/2 to enable the AC to see the direction
70   // in which the integration occurred
71   AC0.DACREF = 0x80;
72 }
73
74 bool dual_slope_conversion_complete(void)
75 {
76   return conversion_done;
77 }
78
79 uint16_t dual_slope_get_result(void)
80 {
81   return conversion_result;
82 }
83
84 ISR(TCA0_CMP0_vect)
85 {
86   // "integral_direction_up" indicates wether the v_in is above or below
       Vdd/2
87   integral_direction_up = (AC0.STATUS & AC_CMPSTATE_bm);
88
89   // Configure the integrator to integrate towards the middle, with AC0
       thresholding
90   // against a value slightly beyond the middle
91   if (integral_direction_up) {
92     AC0.DACREF = 0x80;
93     // Mux in high voltage
94     DAC0.DATA = (0x3FF) << DAC_DATA_gp;
95     OPAMP.OP1INMUX = OPAMP_MUXPOS_DAC_gc | OPAMP_MUXNEG_OUT_gc;
96     // Enable AC0 interrupt on falling edge
97     AC0.INTCTRL = (0x2 << AC_INTMODE_gp); // AC_INTMODE_NEGEDGE
98   }
99   else {
100    AC0.DACREF = 0x80;
101    // Mux in low voltage
102    DAC0.DATA = (0x000) << DAC_DATA_gp;
103    OPAMP.OP1INMUX = OPAMP_MUXPOS_DAC_gc | OPAMP_MUXNEG_OUT_gc;
104    // Enable AC0 interrupt on rising edge
105    AC0.INTCTRL = (0x3 << AC_INTMODE_gp); // AC_INTMODE_POSEDGE
106  }
107
```

```
108   // Enable AC0 interrupt (after clearing the flag so that the interrupt
        does
109   // not immediately fire)
110   AC0.STATUS = AC_CMPIF_bm;
111   AC0.INTCTRL |= AC_CMP_bm;
112
113   // Clear the interrupt flag
114   TCA0.SINGLE.INTFLAGS = TCA_SINGLE_CMP0_bm;
115 }
116
117 ISR(AC0_AC_vect)
118 {
119   // Configure the integrator to integrate back towards the middle
120   if (integral_direction_up) {
121     AC0.DACREF = 0x80;
122     // Mux in slightly low voltage
123     DAC0.DATA = (0x200 - 100) << DAC_DATA_gp;
124     // Enable AC0 interrupt on rising edge
125     AC0.INTCTRL = (0x3 << AC_INTMODE_gp); // AC_INTMODE_POSEDGE
126   }
127   else {
128     AC0.DACREF = 0x80;
129     // Mux in slightly high voltage
130     DAC0.DATA = (0x200 + 100) << DAC_DATA_gp;
131     // Enable AC0 interrupt on falling edge
132     AC0.INTCTRL = (0x2 << AC_INTMODE_gp); // AC_INTMODE_NEGEDGE
133   }
134
135   // Enable AC0 interrupt (after clearing the flag so that the interrupt
        does
136   // not immediately fire)
137   AC0.STATUS = AC_CMPIF_bm;
138   AC0.INTCTRL |= AC_CMP_bm;
139
140   // Clear the interrupt flag
141   AC0.STATUS = AC_CMPIF_bm;
142 }
```

7.5.5 C-Code for the FIR-Filter

```
1  #include "fir_filter.h"
2  #include <math.h>
3
4  static int16_t kernel[FIR_FILTER_KERNEL_SIZE / 2]; // Only store half the
       kernel
5
6  #define SINC(x) ((x) ? (sin(x)/(x)) : (1))
7  #define WINDOW(x) (0.5 - 0.5 * cos(2.0 * M_PI * (x)))
8
9  void fir_init_kernel(const float f_c)
10 {
11   for(int i = 0; i < FIR_FILTER_KERNEL_SIZE / 2; i++) {
12     int index = i - FIR_FILTER_KERNEL_SIZE / 2;
13     float ideal_kernel = 2.0 * f_c * SINC(2.0 * M_PI * f_c * index);
14     float window = WINDOW((float)i / FIR_FILTER_KERNEL_SIZE);
15     kernel[i] = (int16_t)(FIR_FILTER_KERNEL_SCALE * window * ideal_kernel)
       ;
```

```
16    }
17  }
18
19  uint16_t fir_filter_bitstream(volatile uint8_t* bitstream)
20  {
21    volatile uint16_t accumulator = 0;
22
23    for(int i = 0; i < FIR_FILTER_KERNEL_SIZE; i++) {
24      if (bitstream[i / 8] & (1 << (i % 8))) {
25        if (i < FIR_FILTER_KERNEL_SIZE / 2) {
26          accumulator += kernel[i];
27          } else {
28          accumulator += kernel[FIR_FILTER_KERNEL_SIZE - 1 - i];
29        }
30      }
31    }
32
33    return accumulator;
34  }
```

NTNU
Norwegian University of
Science and Technology

MICROCHIP