Laboration Report
SSY011

Jacob Andersson, Ludvig Johansson

Summary

This lab rapport concludes the building and verification of an audio transmission system. The transmission is digital with 8 bits resolution with an aim of 20- 12000 Hz bandwidth, but in reality the bandwidth was much less. The project was divided in steps where an apparatus was built by several subsystems and by combining analog and digital electronic components. A standardized way of transmission made communication between different lab groups possible.

Contents

1 Introduction

This report will present the approach and calculations that has been made during the course Elektriska system. The purpose of this lab is to get deeper knowledge in analog and digital electronics, the programming language VHDL and simulations in the program LTSpice.

The assignment of this lab is to build a sound transmission system with specific specifications, these are 8-bit resolution and frequency range between 20 to 12000 Hz. The system should both be able to transfer and receive signals. The system will be built of various parts and components during six separate labs, each lab concentrated on one specific part of the system [3].

- Counter and parallel to serial converter
- D/A-converter
- A/D-converter and serial transmitter
- Serial transmission
- Audio amplifier
- Low-pass filter and functional test

These parts can be seen in the schematics of the subsystems shown in figure 1

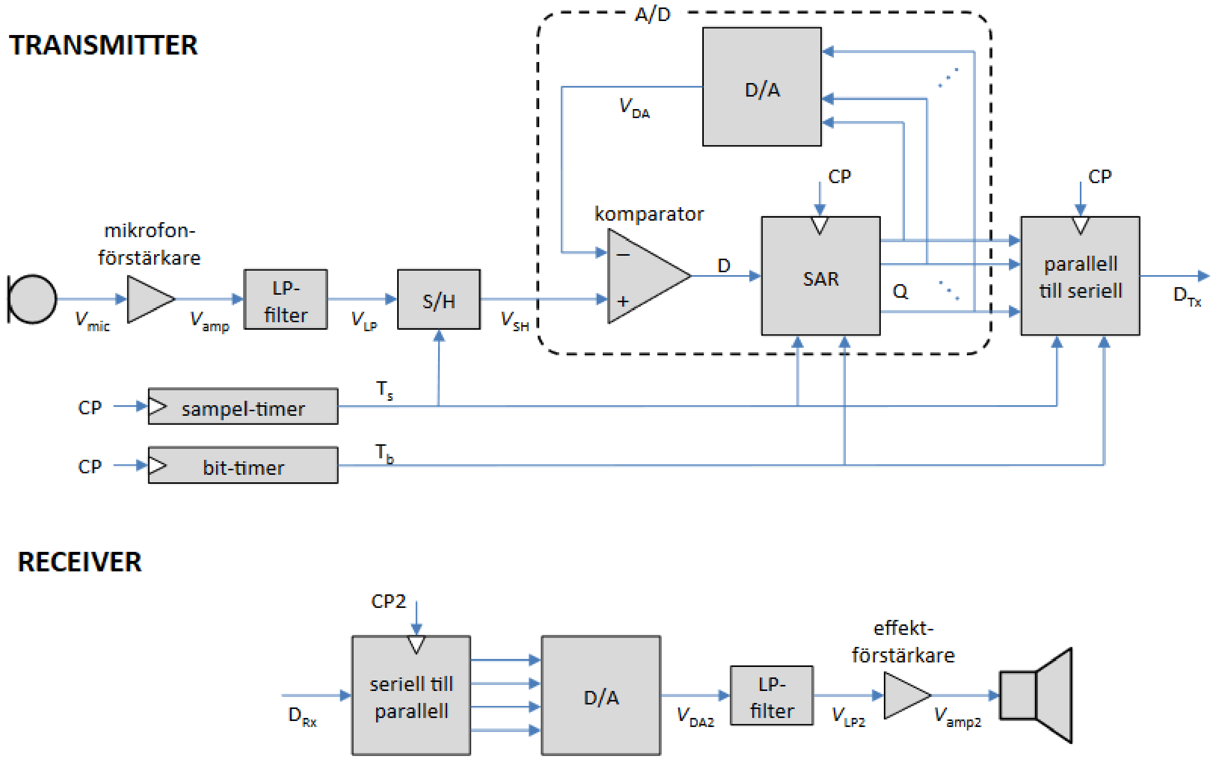


Figure 1: Schematics over the subsystem in the apparatus, from [3].

2 Subsystems

All the different subsystems will be presented below.

2.1 Counter

During this lab two counters were created to symbolize two different clocks. These clocks were designed so they had different frequency interval and in that way could be used as separate trigger signals. The two clocks were based on the 50 MHz clock that is included in the Development and Education board (DE1), in this board there is a integrated FPGA circuit, which stands for field-programmable gate array. This circuit consist of many digital components such as gates, memory, buses and other peripherals. The connection between these can be reconfigured. This board is assigned in the start of the lab. The two clocks that was prepared had frequencies of 960 Hz and 9600 Hz.

As mentioned, the two clocks should have functions as trigger signals. The 960 Hz clock should work as a sample trigger T_s with a duty-cycle of 5%, this can be seen in figure 2. It will create a pulse train that for each puls saves a bit pattern to a register. The second clock with 9600 Hz frequency will be a shift register trigger called T_b, the pulse width of T_b can be seen in figure 3 . For each pulse that the clock creates a register will be shifted one step to the right, example is shown in figure 4. There was as well a process for parallel to serial converting. This process should take a message of 8-bits and by shifting the register bit by bit it should create a serial transfered message

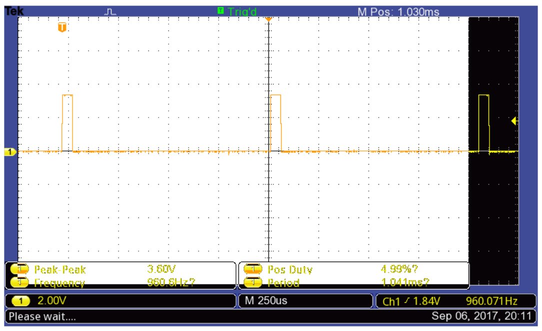


Figure 2: The signal T_s with a frequency of 960 Hz.

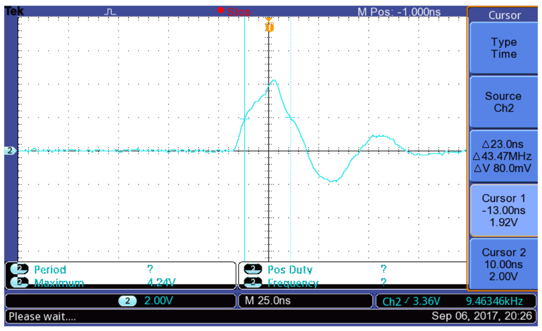


Figure 3: Signal T_b with a frequency of 9600 Hz.

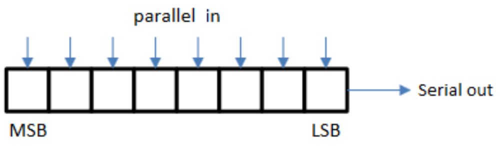


Figure 4: Principal for parallel to serial transmission, from [3].

The system is constructed by the use of VHDL coding. VHDL is is a hardware description language. It is for programming VHSIC, which stands for very high speed integrated circuit. The 50 MHz clock that is integrated in the DE1 board is in two different processes divided into each clock. A problem that arose was that the two clocks T_S and T_b went out of sync of each other. This problem occurred since the number of ticks of each counter where not a factor ten of the other. This was corrected by truncating the counter for T_S. When both the clocks were implemented a process for parallel to serial converter was designed. This converter was programmed in the way that 8 switches were used as in signals. When the process got a T_s pulse the signals where stored in a register. For every T_b pulse this register where shifted one step to the right and the bit that was in the first location of the register, which is the LSB, where transmitted out on the signal D_{Tx}. In figure 5 one may see a message that was sent.

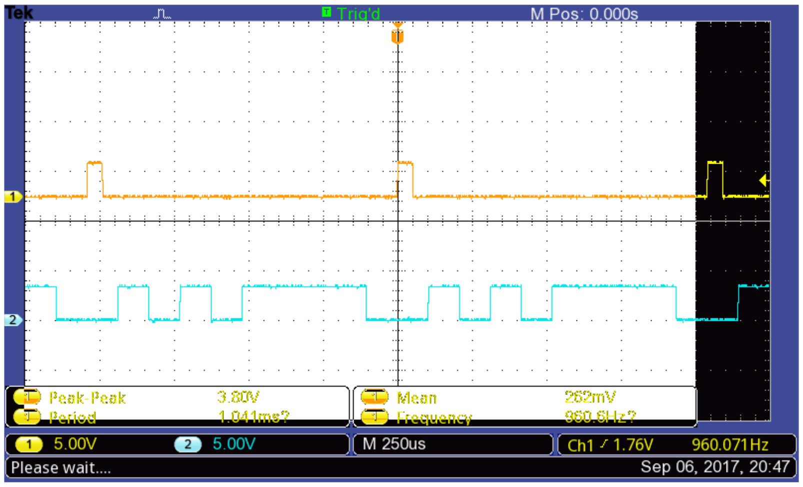


Figure 5: Message sent serial.

In table 1 the values of times and frequencies that was measured is shown.

Table 1: Table showing period time and frequency for signals T_s and T_b.

Signal	T_s	T_b
Amplitude	$\sim 3.3V$	$\sim 4.24V$
Period time	1.042 ms	104 μs
Frequency	960.071 Hz	9.615 kHz
Pulse width	52.4 μs	23 ns

2.2 D/A converter

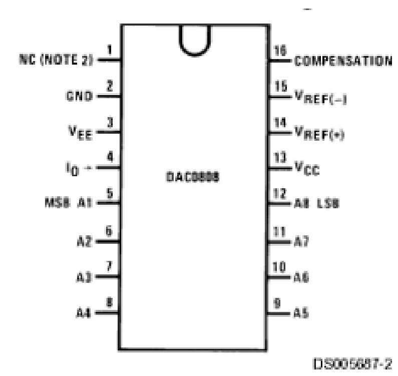
In this part a digital to analog converter, also known as DAC, was implemented and examined. The cornerstone of the circuit is the DAC0808, shown in figure 6, made by Texas Instrument. This is an 8-bit converter meaning that it has $2^8 = 256$ levels. There is also an operational amplifier of the type TL072, shown in figure 7, located in the circuit. The system is built as shown in figure 8.

The specifications for the circuit was specified in the Lab-PM [3] and is the following

- Current I_{14} shouldn't be higher then approximately 2 mA, subsequently limiting the maximum current of I_4 to the same value.
- Output voltage V_{DA} should lie in the interval of 0-10 V.
- The current through R_3 should not go above 20 mA, this to avoid overheating the resistance.
- The reference voltage V_{REF} is determined by the zenerdiode D1, where the reverse voltage of the zenerdiode is 5.6 volts. The current through the diode should be at least 10 mA to get a stable reference voltage $V_{REF} = 5.6$ V. The zenerdiode is marked with max 0.5 W, this limits the maximum current.
- The plotted line in figure 8 will later on feed the D/A-converters off-set resistance R_5 in figure 22 and microphone in in figure 30. These components draws a maximum of 2 mA.

- As logical levels 0 volts is used as logical zero and +3.3 volts is used as logical one.

The function of this system is to take a 8-bit message and convert it to an analog output signal. The process also needs a new clock with the frequency of 100Hz, why this frequency is chosen is explained further down in this section. A new counter consisting of an 8-bit register was also implemented. The counters function is to count up for every pulse from the 100 Hz clock. When enough pulses have been counted the register will overflow and restart. This creates a saw-tooth wave. It was also specified that the counter should be able to switch from counting up to counting down, which will then invert the saw-tooth wave.



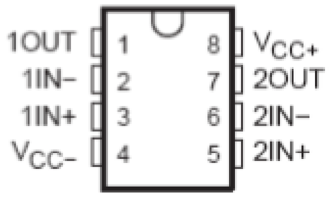


Figure 6: Pin configuration of the DAC0808, picture from [1]. *Figure 7: Pin configuration of the TL072 amplifier. Picture from [3].*

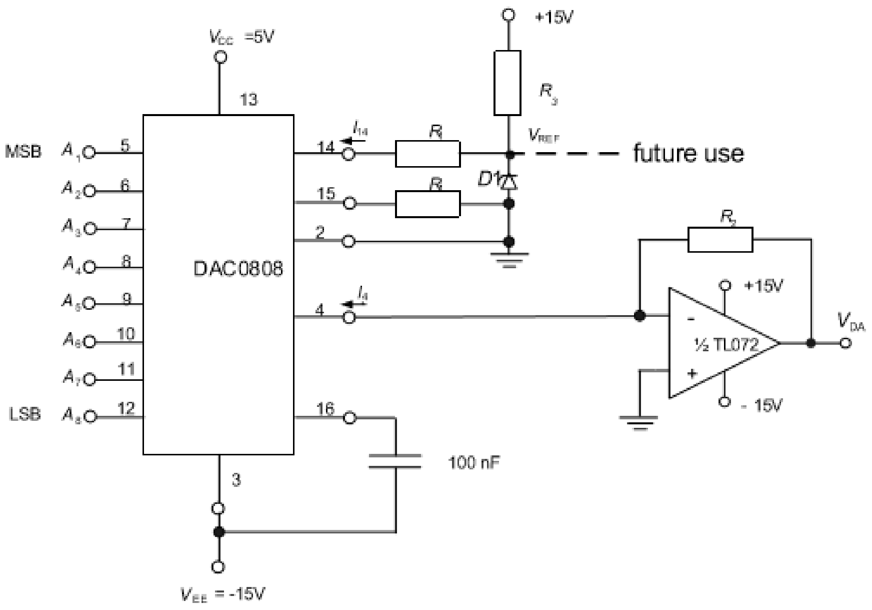


Figure 8: Schematics of the D/A-converter. Picture from [3].

To get the right resistances R_1, R_2 and R_3 in the circuit 8, the values for them was calculated using the mathematical formulas 1 and 4 from [3].

$$I_4 = I_{14} \cdot \frac{A}{256} = I_{14} \cdot (\frac{A_1}{2} + \frac{A_2}{4} + ... + \frac{A_8}{256}) \tag{1}$$

and since

$$I_{14} = \frac{V_{REF}}{R_1} \tag{2}$$

as well as

$$V_{DA} = I_4 R_2 \tag{3}$$

the following expression could be obtained

$$V_{DA} = \frac{V_{REF} \cdot R_2}{R_1} \cdot \frac{A}{256} \tag{4}$$

After calculations the resistances where rounded to match the E12 series, which are the actual resistances that can be used. It was also taken in consideration that the specified values of currents and effect was not compromised. The chosen resistances are shown in table 2.

Table 2: Table over chosen resistances.

R1	2700 Ω
R2	4700 Ω
R3	560 Ω

The system is then connected, following the schematics from figure 8 and with the actual resistances that's been calculated and rounded. The flat cable, which is connects the DE1 board to the breadboard, this cable handles the connection between theses two boards. The inputs on the DAC (A_1 to A_8) is the output signals from the DE1 board. A sequence of different measurements was then obtained on the analog output signal V_{DA}, this to examine if calculated values on the V_{DA} signal coincide with the measured values. In table 3 the measurements and calculations are submitted.

Table 3: Table showing calculated and measured values on output signal.

Digital control signal (Decimal)	Calculated analog output signal (V)	Measured analog output signal (V)
0	0	0.00
16	0.61	0.60
32	1.22	1.20
64	2.43	2.40
128	4.87	4.81
255	9.71	9.57

Table 3 shows that the calculated values for the output signal did very well match the measured values and no further measurements was done.

Next part was to write VHDL code for a 8-bit binary counter with the option to change if it should count up or down, the code can be seen in Appendix listing 6. The frequency should be set to count up or down 100 times a second i.e. 100 Hz. As before, the 50 MHz clock should be used to implement the new clock. The choice between up or down counting should be set by the switch SW9 on the DE1 board. All of the bits in the 8-bit binary counter should be available as output signals as well as T_b. As mention, a flat cable is connected to the DE1 board as well the breadboard in order to communicate between them. The message in the binary counter should as well be shown on light emitting diods (LEDs) on the DE1 board to be able to verify the message.

When the program was running, with the use of an oscilloscope, the signal could be displayed. The period time could then be measured using functions in the oscilloscope. It was measured to 2560 ms which matched the calculated period time that was calculated to 2560 ms.

The trigger signal was then changed to 9600 Hz to get a faster saw tooth wave, these can be shown in figure 9 and 10.

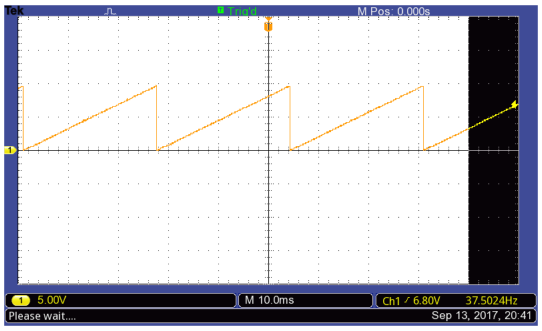


Figure 9: Picture from oscilloscope showing upward saw tooth wave.

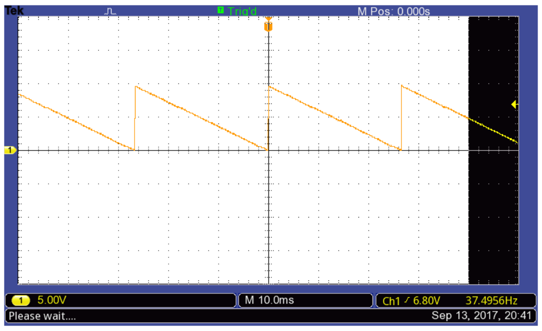


Figure 10: Picture from oscilloscope showing reverse saw tooth wave.

When zoomed in on the output signal when it goes from 00000000 to 11111111 one may see a slightly overshoot of the signal and "ringing", this can be seen in figure 11. To overcome this problem a capacitor C_2 was parallel connected to the resistance R_2 with a value of 33 pF which creates a HP-filter and removes the "ringing"', as shown in figure 12. Without the C_2 capacitor, the signal was stable after 3.48 μs. When the capacitor C_2 was parallel connected with resistance R_2 the time until it was stable was only 468 ns.

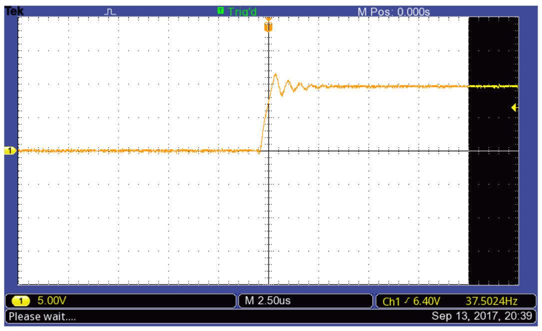


Figure 11: Picture showing the overshoot and "ringing" signal when the signal goes from 000... to 111...

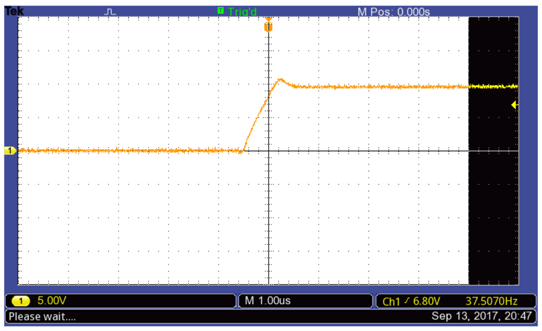


Figure 12: Picture from oscilloscope showing the signal after capacitor C_2 is connected parallel with resistance R_2.

The slew rate of an operational amplifier is the output signals maximum change per time unit. This can be measured by applying a voltage step on the input and then measure how fast the output voltages changes. The saw tooth wave have this step or "jump" when it goes from 00000000 to 11111111. This can then be used to measure the TL072 operational amplifiers slew rate. According the the amplifiers datasheet [4], the amplifier should have a slew rate of 13 V/μs. When measured through the oscilloscope, without the C_2 capacitor connected, the slew rate was 13.5 V/μs, with Δt = 400 ns and Δv = 5.4 V. When C_2 then was connected parallel to R_2, the slew rate was significantly better with 15.35 V/μs, where Δt = 560 ns and Δv = 8.6 V.

The slew rate limits how often the D/A-converter may be used. It's important that the output signal gets the time to converge to the actual voltage that corresponds to the digital signal before it's changing.

2.3 A/D converter

One important part for this transmission system is a way to make analog sound into digital signals. One says that a signal needs to be sampled and quntized. This part will cover how the conversion from analog voltage to digital parallel takes place, in other words, how it is being quantized and sampled. Therefore an Analog to Digital Converter (ADC) needs to be constructed. The construction of a DAC in the previous section is reused in this solution and a Successive Approximation- type of ADC is built.

This type of ADC uses three main parts, see figure 14. A D/A converter, a comparator and a Succesive Approximation Register (SAR). The SAR will be realized using VHDL and the FPGA, whilst the D/A converter and the comparator is hardware on the breadboard. The comparator used in this task is LM311.

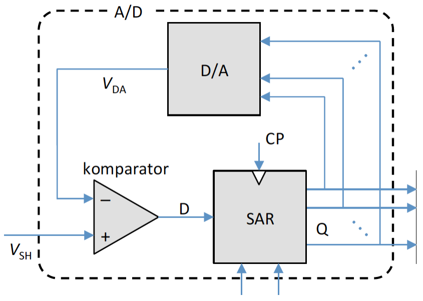


Figure 13: A schematic view of the function principals for the ADC

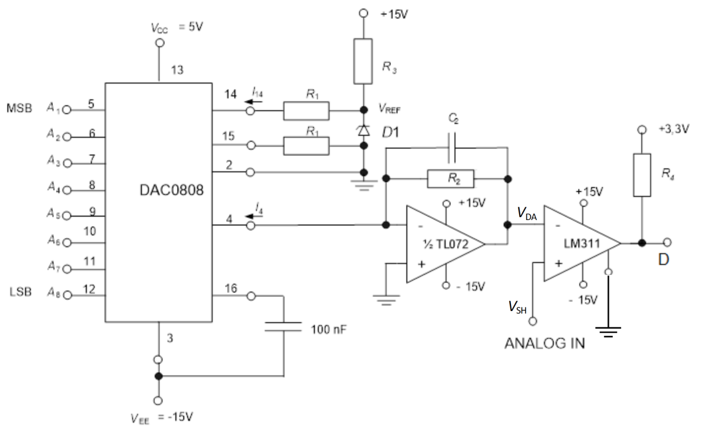


Figure 14: A circuitdiagram for the analog part of the ADC

Looking at figure 13 the function principle is revealed. The SAR will control the D/A to output a known voltage and the comparator will tell if this voltage is higher or lower than the analog voltage in, here called V_{SH}. Using this principle to "guess" what V_{SH} is by using an algorithm.

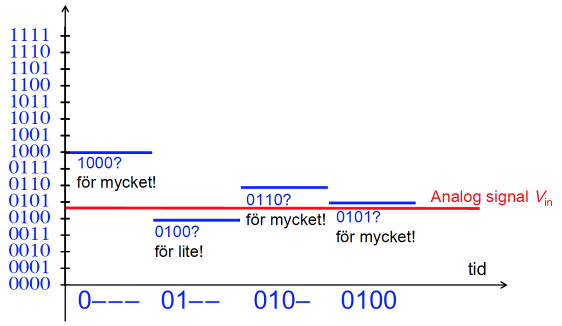


Figure 15: Example of 4 bit SAR-algorithm. Picture from slides in lecture 7.

In the final solution an 8 bit register is used but in order to demonstrate the SAR algorithm an example with 4 bits is used in figure 15. The y axis is voltage. The nibble put out by the SAR, which the DAC translates to a voltage, is shown to the left of the y-axis. The blue line is the actual voltage put out by the DAC and the red line is the analog signal examined. The x-axis is time and beneath is the output from the SAR.

V_{ref} is the maximum value the ADC will be able to quantize and it is 10 V. In this example there are 16 discrete levels but in a 8 bit ADC there will be 256 levels. As a first guess, the SAR will output half of V_{ref} from the DAC. The comparator will give back a 0 if the guess was to high and a 1 if the guess was to low. One could think of it as 0 equals 'go higher' and 1 equals 'go lower'. So then in the next guess in this example the SAR guesses lower and the comparator returns a 1. Note how the SAR keeps the output from the comparator between each cycle. By this successive process the best possible approximation of the analog voltage is achieved.

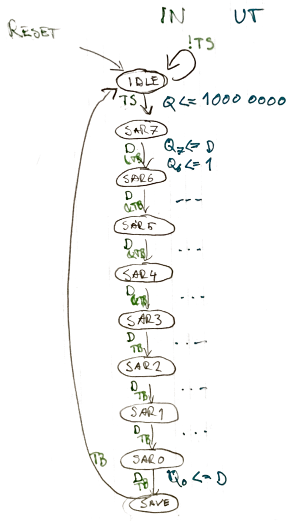


Figure 16: A Moore type diagram for the final solution, later realized in VHDL.

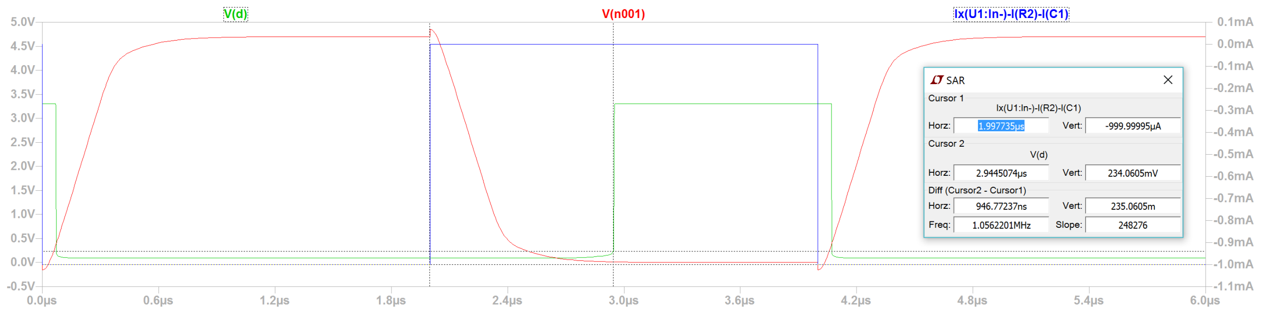
In order to realize this algorithm one good way is to use a state machine. The process in doing one contains writing a state diagram. Figure 16 show the solution. In green is all input signals and in blue are the outputs. The pulse T_S will bring the machine out of idle. It will put it's first guess out and will move to the next state with every T_B pulse. The actual source code can be found in listing 7.

There are however some limitations for this type of ADC. First of all, an analog continuous physical quantity is being described in a discrete quantity. In the example above, the description get very course. By adding bits the description gets finer and finer, but also time is lost in each cycle. This makes the response time from the output of the SAR until the output from the comparator is stable into an important factor since it is corresponding to one cycle of approximation.

In order to find this response time LTSpice is used to simulate this. The DAC0808 is not possible to simulate in LTSpice but, referring to figure 14, the response time between current I_4 and voltage D is possible. The test is to simulate the lowest voltage measurable as the input V_{SH} and see whats happens when a step is made between 0 and 1 mA on I_4, since this corresponds to a 0 to 5 volts step on V_{DA}, which is the largest step (guess) ever made. All other steps will be smaller and therefore faster. In figure 17 is a simulation of this response. In the worst case scenario the response time was about 1 μs. The greatest hold back in speed is the OP-amp TL072 which has a Slew Rate (SR) of about 13 V/μs but since the comparator reacts when $V_{SH} > V_D A$ or vice verse, and this transition happens in the slope (see figure), this adds even more time for the response.

As a first step of function test a DC source was used to try to simulate the byte 00110000. In figure 18 an theoretical time diagram for the approximation resulting in the mentioned byte. In figure 19 is the actual result plotted on an oscilloscope. This byte should theoretically correspond to 1.83 V and in the experiment the measured voltage was 2.04 V. In other words, it was close but some calibration might be needed and in big, the ADC is working as expected. An input of 5 V was also put in and it yielded the byte 1000 0101 (= 4,98 V) and 100 0011 was expected so the accuracy was much greater around the middle.

In the lab a test was made to measure the response. Shown in figure 20 is an oscilloscope picture showing



SAR

Cursor 1
Ix(U1:In-)-I(R2)-I(C1)
Horz: 1.997735µs Vert: -999.99995µA

Cursor 2
V(d)
Horz: 2.9445074µs Vert: 234.0605mV

Diff (Cursor2 - Cursor1)
Horz: 946.77237ns Vert: 235.0605m
Freq: 1.0562201MHz Slope: 248276

Figure 17: Simulating response time from a change in current I_4 (blue) until the binary signal D (green) is output. Also in graph is the output of TL072 where it is taking a step from approx 5 V down to 0 V, the greatest step ever expected.

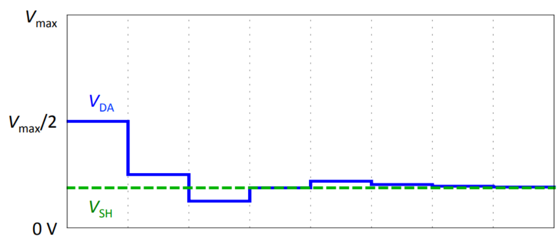


Figure 18: Timing diagram for successive approximations for the digital output 00110000.

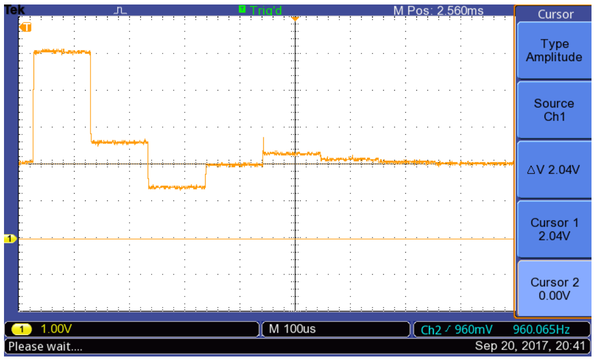


Figure 19: The circuit in the lab, reproducing the same word.

the delay Q to D when the input voltage is 3 V. In this situation the response time was about 0.3 μs. When the input voltage is changed to just below 5V the delay is greater, about 0.65 μs. This is because the SR gives a certain rate of change for the output of V_{DA} which leads to that the time until the comparator switches gets longer, since V_{DA} have to make a greater change before the transition $V_{SH} < V_{DA}$ is made. The greatest response time measured was about 0.65 μs. In the previous lab SR of the TL072 was measured to 15V/μs but in the LTSpice model it is 13V/μs. Unfortunately the step when V_{DA} goes from 5 to 0 was not possible to measure and it is a state that can never take place because if V_{SH} is 5V in one cycle of approximation, and in the next cycle V_{SH} is 0 V, then V_{DA} would be 5V in the first step, then 2,5V in the next and then smaller and smaller steps towards 0V. This is rendering the biggest step ever made to be 2,5 V which should be approximately 0.55 μs according to simulations, and was measured in the lab to 0.65 μs.

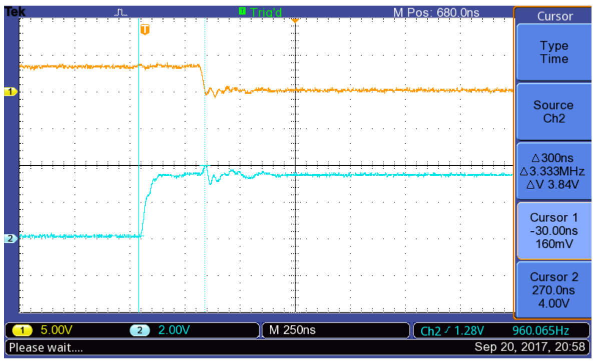


Figure 20: Channel 1: D output, Channel 2: Q_7 input, external trig on T_S, showing result when $V_SH = 3$ V

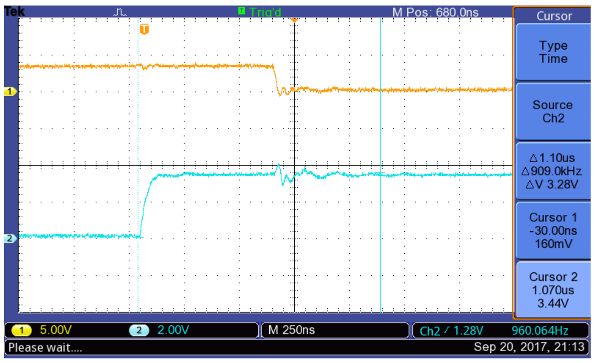


Figure 21: Channel 1: D output, Channel 2: Q_7 input, external trig on T_S. Showing a worst case.

To establish this part was an important step to determine the greatest conversion time needed for one sample. Each bit generated needs in the worst case 0.65 μs, so the greatest bitrate possible is 1,54 Mbit/s or 154 k samples/s. Looking at figure 21 it is observable however that V_{DA} becomes a bit unstable when the transition takes place at the comparator and a bitrate period of 1.1 us would be needed to ensure that all parts of the system will take a stationary condition. Conclusion, for a safe and stable transmission a maximum bitrate of 0.91 Mbit/s should be used.

With this setup, the ADC was able to convert voltages between 0 to 10V. Since audio transmission uses sine waves, we must add an offset in order to convert -5 V to 5 V. (The very same solution was also used for the DAC when we use the board as a reciever). To compute the value of R_5 the current through R_5 is essential, here after called I_{R_5}. The design criteria told us in section 2.2 that $I_4 \in [0, 2]$ mA (which ended up being $I_4 \in [0, 2.07]$ mA with the chosen resistors) and by realizing that I_{R_5} will work as and subtraction to I_4's contribution to R_2 since $I_{R_2} = I_4 - I_{R_5}$ (Kirschoffs current law). Since the goal is to make V_{DA} symmetric around 0V, I_{R_5} must be 1 mA. Another way to see it, is by realizing that we want to offset the output -5V and that $I_{R_2} = V_DA/R_2 = -5/4700 \approx -1$ mA leading $I_{R_5} = 1$ mA. Since $R_5 = V_{ref}/I_{R_5}$ this yields $R_5 = 5600\Omega$

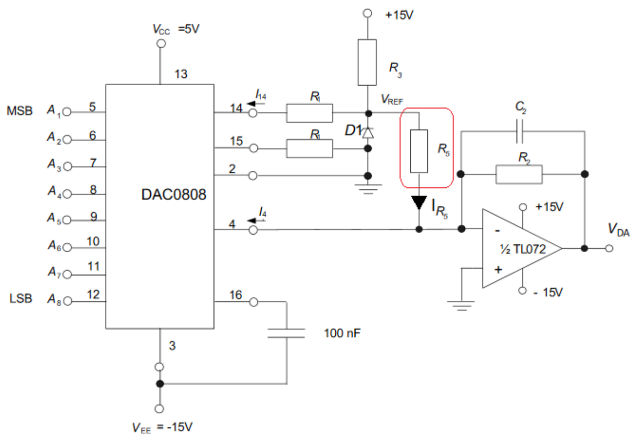


Figure 22: Introducing resistor R_5 to create an offset voltage

2.4 Sample and Hold

The signal being sampled by this system varies in time. In the previous section it was concluded that the ADC needs time in order to convert the analog signal to digital. Should the input voltage vary during the conversion cycle wouldn't the SAR be able to accurately approximate this signal. Therefore a component that can hold the input voltage at a steady level during this cycle needed.

A circuit solution for this problem can be done using a voltage follower, a switch and a capacitor. In figure 23 is the function principle of the sample and hold illustrated. The principle is that as long as the circuit breaker is closed the capacitor will charge or discharge, aiming to have the same potential as the input. Whenever the circuit breaker opens, the capacitor will hold the same voltage since it can't discharge, remember that the input to the OP-amp is high impedance. The OP-amp is connected as a voltage follower and the output of the OP will have the same potential as the voltage being held in the capacitor.

This system uses a ready-made component called LF398 for this function. The capacitor is connected externally and it is being fed with +/- 15 V and will sample (close the switch) when it's logic input is high and hold when the logic input is low. By connecting this logic input to our T_S trigger signal, and the output to the ADC input, it is ensured that the time-varying signal will not change during the AD conversion cycle.

To ensure the function a test was carried out where a signal generator was connected as a input and a oscilloscope was studying both the input and output signal. The sampling frequency of T_S is 960Hz and in figure 24 is a 100Hz triangle wave sampled. Figure 25 is an amazing example of the sampling theroem,

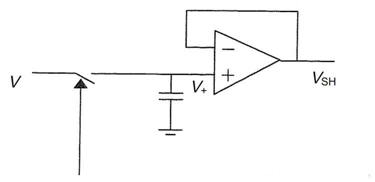


Figure 23: A simple sample and hold circuit

saying that the sampling frequency must be at least twice as high as the signal being sampled. The input triangle wave is 500Hz and it can clearly be seen that the sampled output signal does not at all look like the input. The test was being considered successful and the result motivated why the sampling frequency needs to be at least twice as high as the sampled, something that will be discussed more in later sections.

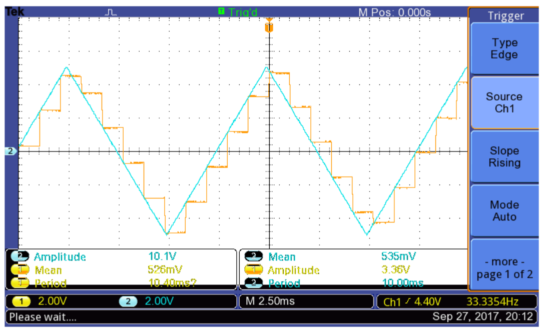


Figure 24: 100 Hz

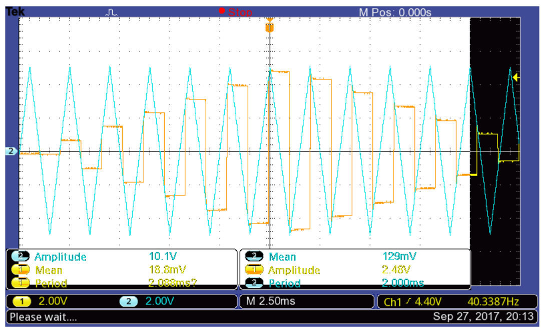


Figure 25

2.5 Serial Transmitter

Refering back to section 2.1 (counter) and 2.3 (ADC) also a serial transmitter needed to be constructed. The solutions for the parallel in serial out shift register in the first lab (counter) was combined with the ADC/SAR code and created and solution presented in listing 8. The important step here was to make sure that the previous sample is the one being transmitted while a present sample was being processed by the ADC, see figure 26. Also a stop bit (high) and start bit (low) was added. In total 10 bits per package.

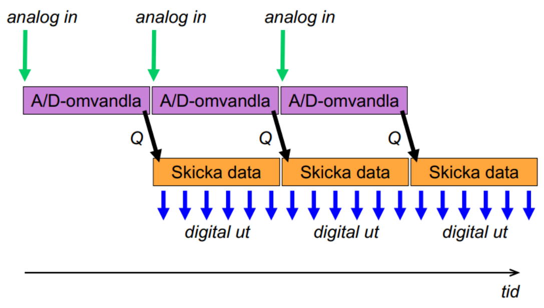
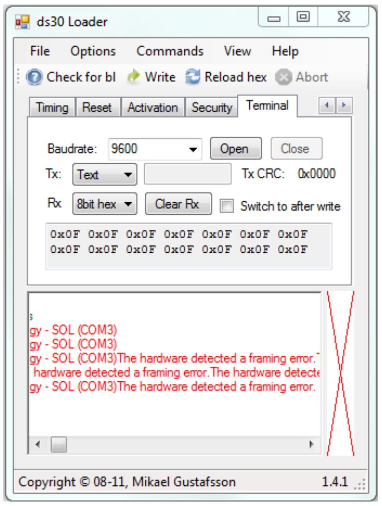


Figure 26: Continious sampling and transmission. omvandla = convert, skicka = send, ut = out, tid = time.

To prove the function of this system, a baud rate of 9600 was set in the hardware. A computer with a terminal prompt was used to receive a signal. In figure 27 is the recieved byte shown and in figure 28 is the actual output from the system shown.



Baudrate: 9600
Tx: Text
Tx CRC: 0x0000
Rx 8bit hex

```
0x0F 0x0F 0x0F 0x0F 0x0F 0x0F 0x0F
0x0F 0x0F 0x0F 0x0F 0x0F 0x0F 0x0F
```

```
gy - SOL (COM3)
gy - SOL (COM3)
gy - SOL (COM3)The hardware detected a framing error.T
 hardware detected a framing error.The hardware detecte
gy - SOL (COM3)The hardware detected a framing error.
```

Copyright © 08-11, Mikael Gustafsson 1.4.1

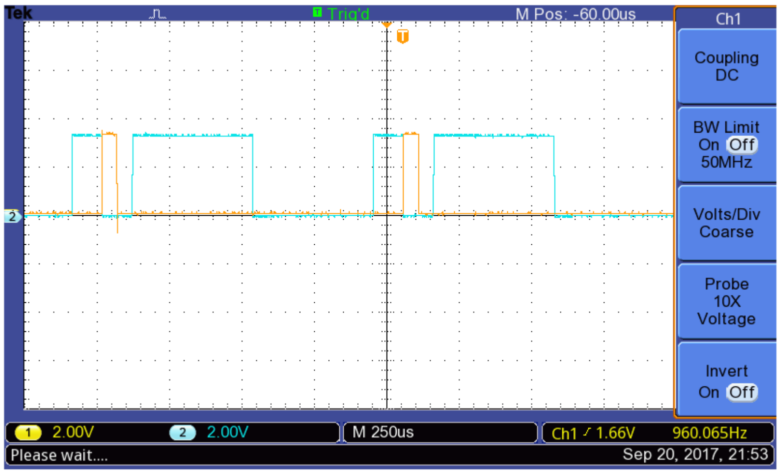


Figure 27: The byte 0x0F received in a terminal prompt.

Figure 28: The byte sent. Please observe that LSB is sent first, and there is start and a stop bit. In yellow T_S and serial out in blue.

2.6 Serial receiver

For the system to receive a serial transmitted signal it has to be synchronized with the transmitting side. To achieve this, a system where the receiver identifies a stop and start bit is built. The stop bit is a logical zero and when a start bit, that is a logical one, is noted the system starts a sequence where it interpret the incoming signal. The interpretation is done on the middle part of each bit period to safely identify the actual logical bit. The system acquires 8 bits and halts when the stop bit occurs. Whit these 8 bits, the system can then transform it into an analog signal. The logical zeroes and ones are represented as -8 and +8 volts on the DE1 card, this is handled by the cards driver though so it won't be taken in to consideration during this lab.

To be able to receive the value of one bit in the middle part of the bit period a counter must be applied. This counter will use the T_b signal to count up one step, this is also to synchronize the receiver with the transmitter with a baud rate of 9600 bit/s. When the counter has done 2604 ticks, which is half a period, the information of the bit as well as the next state is stored. When the counter has done 5207 ticks, a full period, the counter is reset. When it has reached the final state, the bit sequence is shown on the LEDs on the DE1 board. A state diagram, shown in figure 29, represent the sequence in a good way. The VHDL code that was used is represented in listing 9.

To test the receiving system, the DE1 card and the circuit was connected to the computer with a RS232 cable. The system was tested by sending different ASCII signs from the computer to the circuit and analyze what the receiving sequence of bits was as well as the measured analog output voltage, the results is shown in table 4.

Table 4: Received bit sequence as well as measured output voltage.

ASCII-sign	Received bit sequence	Measured analog output (V)	Calculated bit sequence
P	01010000	-1.6	01010000
?	00111111	-2.24	00111111
å		Don't know	11100101
!	00100001	-3.44	00100001
=	00111101	-2.32	00111101
A	01000001	-2.16	01000001

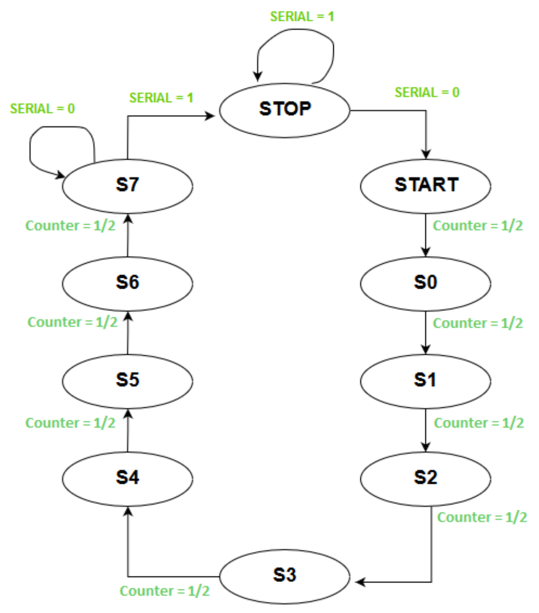


Figure 29: State diagram that shows the sequence for the receiver.

When the letter å was sent, the signal was undefined, or the same as ?, showing that the system don't know how to interpret the signal. This is because the letter å is not included in the standard ASCII table. If the signal instead was sent as hex, which would then be 0xE5, the sign was properly received. What can be noted is that the letter å is the only letter in table 4 where the MSB is a 1, shown in the calculated bit sequence column.

2.7 Audio amplifier

To be able to send and as well receive sound, a microphone amplifier and a power amplifier, for driving a headphone or speaker, will have to be installed. The microphones function is to convert audio signals into electrical signals, and is proportional to the sound pressure that the system can translate and later transmit. The power amplifiers function is to take the received signal and amplify it to suit the actual speaker or headphone.

The microphone amplifier, shown in figure 30, works as a voltage source and a JFET, the N-channel JFET got the same current-voltage characteristics as a MOS transistor, i.e. negative threshold voltage [3]. The JFET is a kind of field effect transistor which means that the current is driven by electric fields. The field effect transistor got three inputs, source, gain and drain. Source is where the charge enters the transistor, drain is where the charge goes out from the transistor and gate is what controls the flow between source and drain. JFETs are made of P and N doped substrate. This means that the P substrate contains a lot of positive charges, and N doped substrate is when it contains a lot of negative charges. The JFET has a channel that is controlled by the voltage over reversed bias PN-junction [2].

The power amplifier, as shown in figure 31, is made of an operational amplifier circuit and two CMOS amplifiers, this is also known as a class B type power amplifier. This means that the transistors leads half of the time each. The operational amplifier, which is of the sort TL072, as in figure 7, can't generate the current that is needed to drive a headphone or speaker. That is why the CMOS amplifiers are added. A headphone normally needs around 10-20 mW and with a input impedance of 20 Ω the current that is needed is around 35 mA_{eff}. The input for the power amplifier is supposed to come from the D/A-converter after the signals has been filtered, which will be discussed in the next section. The function of the resistance R_6 that is connected after the operational amplifier and connected to the output lead is to remove crossover distortion, this occur when the transistors goes from negative to positive, and vice versa.

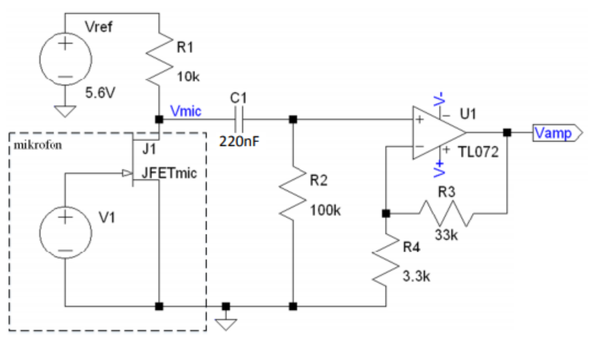


Figure 30: Schematics over the microphone amplifier circuit, from [3].

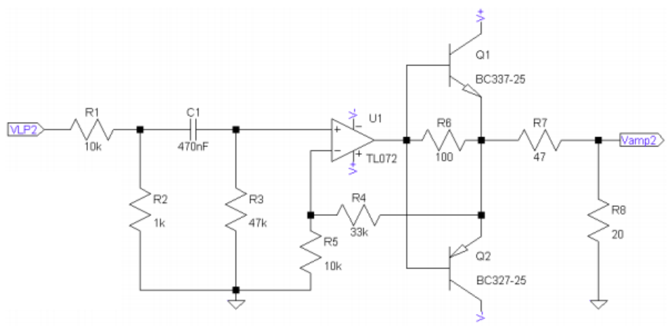


Figure 31: Schematics over the power amplification circuit, from [3].

The task for this part of the lab was to connect both of the circuits and perform measurements. For the microphone amplifier, the voltage of 5.6 V was taken from the reference voltage V_{REF}, as can be seen in figure 8. Then the DC-voltage was measured on the microphones input to 3.36 volts. This was then used to calculate the resting current for the microphone element using equation 5.

$$I_{REST} = \frac{V_{REF} - 3.36}{10k\Omega} \tag{5}$$

The resting current I_{REST} turned out to be 0.22 mA.

The power amplifier was then examined to see which frequency is the lower cut-off frequency, which is when the amplitude of the signal is lowered with 3 dB. A function generator was used to produce a sinus signal on the input V_{LP2}. The amplitude of the sinus signal was set to 5 V and the frequency range was between 1 Hz to around 20 kHz. The measurement data was then collected with three points per decade. Table 5 shows the acquired results.

Table 5: Measurements for the power amplifier showing amplification compared to the input frequency.

Frequency	Input signal (V)	Output signal (V)	Amplification (dB)
1	10	0.14	-37
2.15	10	0.30	-30.46
4.6	10	0.58	-24.72
10	10	0.92	-20.72
21.5	10	1.04	-19.65
46.4	10	1.08	-19.33
100	10	1.10	-19.17
215	10	1.10	-19.17
464	10	1.10	-19.17
1000	10	1.10	-19.17
2154	10	1.10	-19.17
4649	10	1.10	-19.17
10000	10	1.10	-19.17
21544	10.6	1.18	-19.23

2.8 LP filter

During the final part the task was to build a low-pass filter. The low-pass filters function is to filter out frequencies that is not wanted. For example, the sampling frequency for the sound transmission system that is built is 24 kHz, this because the system should be able to manage audio signals with a frequency range from 20 Hz to 12 kHz. Because of the frequency that the system is sampling with, all frequencies above

12 kHz is unwanted due to the Nyquist criteria. To address this problem a low-pass filter is implemented on both the transmitting side of the circuit, which prevent aliasing of the signal when sampling, as well as the receiving side, which smooths the signal from the D/A-converter. The cut-off frequency, which is the frequency where the filter, after reaching this point, attenuates signals with higher frequencies than 12 kHz. For a non ideal filter, the cut-off frequency is when the signal is damped 3 dB. A visualization of how the low-pass filter attenuates the signal can be shown in figure 33.

The low-pass filter can be represented in a complex plane,shown in figure 32. In this plane, real and complex numbers represent zeros and poles. Zeros are where the transfer function becomes zero and poles are when the transfer function go against infinity. Both zeros and poles must be either complete real or appear in complex conjugate pairs [5]. The filter that is constructed during this task is a Butterworth using two cascaded second degree Sallen-key filter. Using following table 6, the transfer function can be calculated.

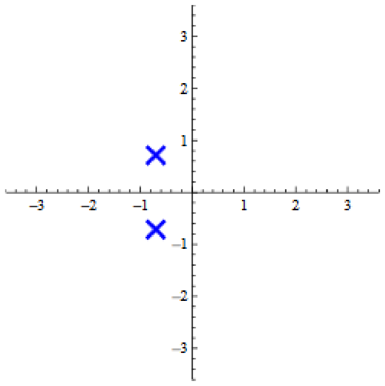


Figure 32: Example showing poles plot for a low-pass filter without zeros. from lecture slides.

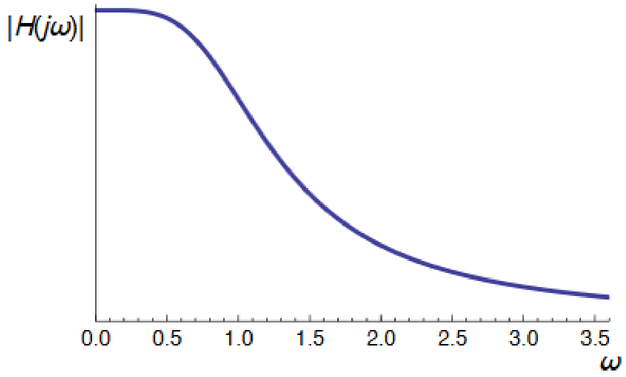


Figure 33: Example of bode plot for a low-pass filter without zeros, from lecture slides.

Table 6: Table showing filter polynomials, from [].

Order	Polynomial (Pa)
1	1+a
2	1+1.414a+a^2
3	1+2a+2a$^2 + a^3$ =(1+a)(1+a+a^2)
4	1+2.613+3.414a$^2 + 2.613a^3 + a^4$ =(1+0.765a+a^2)$(1 + 1.848a + a^2)$

The filter, as mentioned before, was of the Butterworth type. The circuit that was created can be shown in figure 34. The transfer function was calculated as a fourth degree Butterworth, by the fourth element in table 6, and by knowing that the cut-off frequency should be 12 kHz the following calculations where done.

$$a = \frac{1}{\omega_u} = \frac{1}{2\pi 12000} = 13.263 \cdot 10^{-6} \tag{6}$$

$$0.765 \cdot 13.263 \cdot 10^{-6} = 10.146 \cdot 10^{-6}$$

$$10.146 \cdot 10^{-6} = \frac{1}{(R_1 + R_2)C_2} \tag{7}$$

If R_1 and R_2 is chosen to 5500 then

$$C_2 = \frac{10.146 \cdot 10^{-6}}{5500 + 5500} = 0.922nF \approx 1nF$$

the last capacitance is calculated following

$$a^2 = \frac{1}{\omega_u^2} = \frac{1}{(2\pi 12000)^2} = 175.905 \cdot 10^{-12} \tag{8}$$

$$175.905 \cdot 10^{-12} = \frac{1}{R_1 R_2 C_1 C_2} \tag{9}$$

$$C_1 = \frac{175.905 \cdot 10^{-12}}{R_1 R_2 C_2} \approx 5.8nF$$

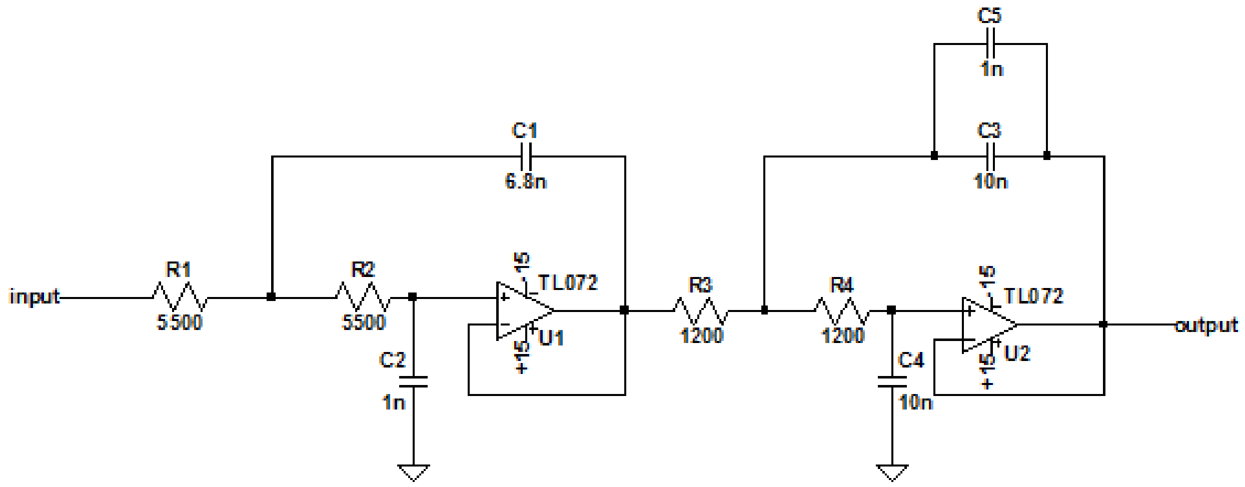
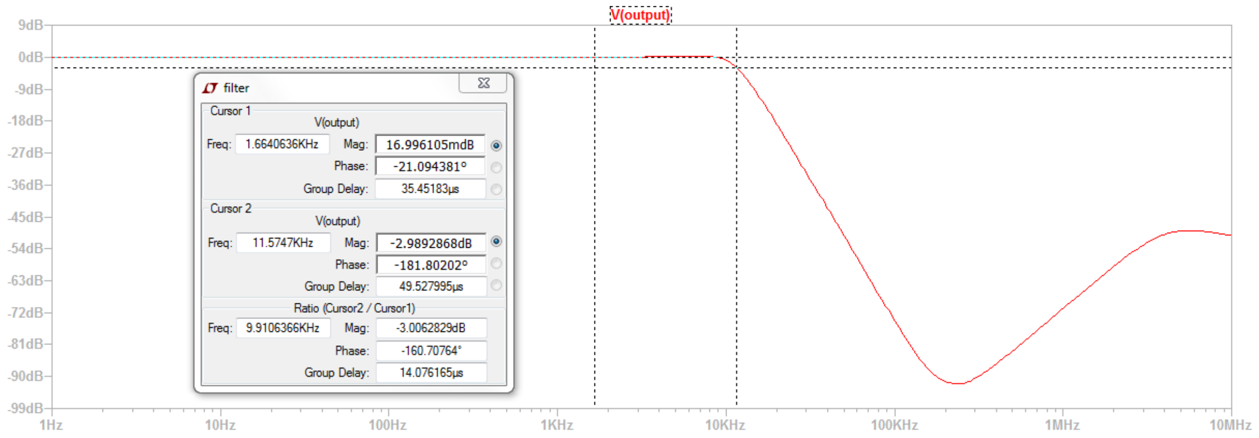


Figure 34: Schematics over 4th degree Butterworth filter made out of two cascaded 2nd degree Sallen-key filters.

The filter was simulated in the program LTSpice to see how the characteristics of the filter was. As can be seen in figure 35, the cut-off frequency, at the -3 dB mark was approximately 11575 Hz which is well enough, since it is close but not more than 12 kHz. Also seen in the plot of the filter is that after approximately 237 kHz, the signal starts to rise in gain again. After further investigations the conclusion of this phenomena was determined to be because of parasite properties in the components.



```
Cursor 1
                 V(output)
Freq:  1.6640636KHz   Mag:          16.996105mdB
                      Phase:        -21.094381°
                      Group Delay:  35.45183µs
Cursor 2
                 V(output)
Freq:  11.5747KHz     Mag:          -2.9892868dB
                      Phase:        -181.80202°
                      Group Delay:  49.527995µs
           Ratio (Cursor2 / Cursor1)
Freq:  9.9106366KHz   Mag:          -3.0062829dB
                      Phase:        -160.70764°
                      Group Delay:  14.076165µs
```

Figure 35: Showing the characteristics for the low-pass filter simulated in the program LTSpice.

After simulation, the circuit was connected on the breadboard. The first measurement that was done was with a sinus wave with amplitude 10 V_{pp} generated by a function generator. The task was to measure, at different frequencies, the output signal to determine what the actual cut-off frequency is. in table 7 all measurements and calculations is shown.

Table 7: Table showing measurements and calculations for attenuation regarding the low-pass filter.

Frequency (kHz)	Input signal (V)	Output signal (V)	Attenuation (dB)
1	10.0	10.2	0.172
2.15	10.0	10.2	0.172
4.65	10.2	10.4	0.167
7.94	10.3	10.4	0.084
8.91	10.2	9.92	-0.242
10	10.2	8.88	-1.204
11.2	10.2	7.28	-2.929
12.6	10.2	5.36	-5.589
14.1	10.1	3.84	-8.400
15.8	10.0	2.56	-11.835
17.8	10.1	1.76	-15.176
22.4	10.0	0.65	-23.742
28.2	10.1	0.28	-31.143

Measured cut-off frequency was 11.2 kHz.

The input signal was then changed to a square wave with the frequency of 6 kHz. The output signal was examined using fourier transformation on the oscilloscope, shown in figure 36. Seen on the oscilloscope is the first peak that is at 6 kHz, the second peak at 18 kHz and the third peak at 30 kHz. What can be seen on the output of the filter is that it turns the square wave into a sinus wave. This occurs because a square wave is made out of several (infinite) sinus waves. The filter rejects the frequencies that is above it's cut-off frequency and the components that is left generates a sinus wave. with an amplitude of 12.8 V.

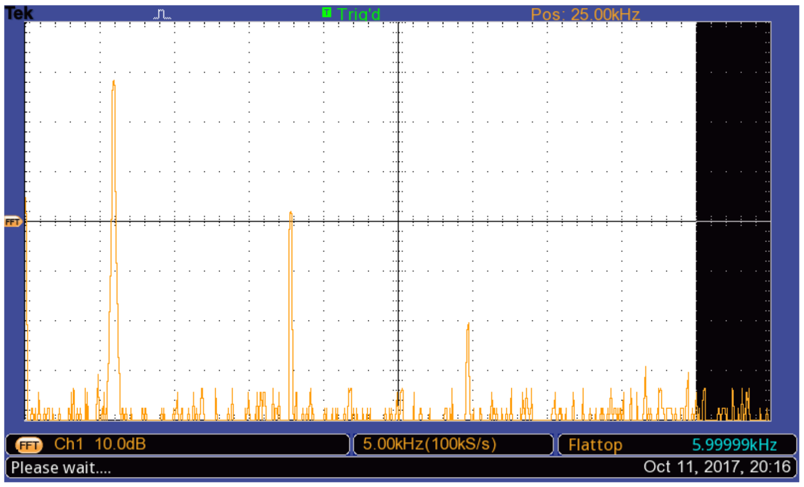


Figure 36: Fourier transform of a sinus wave with the frequency of 6 kHz

3 Test and verification

The final process of this lab was then to cooperate with another group where one created a transmitter and the other one created a receiver. Code for the SAR was downloaded to the DE1 board, the filter and S/H-circuit was connected to the A/D-converter. Our group was the transmitting part. Both the transmitting DE1 board and the receiving DE1 board was then connected using the RS232 contacts. The low-pass filter was then fed with a changeable DC-voltage, the voltage was randomly changed to synchronize the receiver with the transmitter. The transferred DC-voltage was studied in the oscilloscope and the LEDs where compared between the transmitter and receiver.

The system was also tested with a sinus wave with a frequency of 1 kHz as input signal. The amplitude of the signal was then changed and the smallest amplitude that could be detected in the receiver was 200 mV_{pp}, to be noted is that the function generator couldn't go lower. The maximum amplitude of the signal was as well studied to see when the output signal got distorted, this happened at 10 V_{pp}. The dynamics of the maximum and minimum amplitude was then calculated to 34 dB.

The signals where then examined after the filter on both the receiving side (V_{LP2}) and the transmitting side (V_{LP}) to see what difference both the signals had. What could be noted was that the signals appeared similar. As can be seen from figure 37 and 38, the filter on the transmitting side "smoothens" the signal before sending it to the receiver. The signal that is then received is a sinus wave with maximum frequency of 12 kHz and will therefore look the same after the receivers filter.

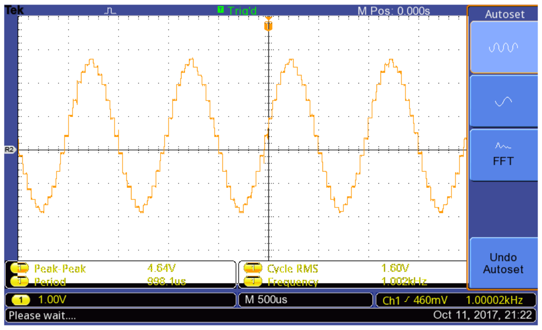


Figure 37: Signal going in to the low-pass filter at the receiving part.

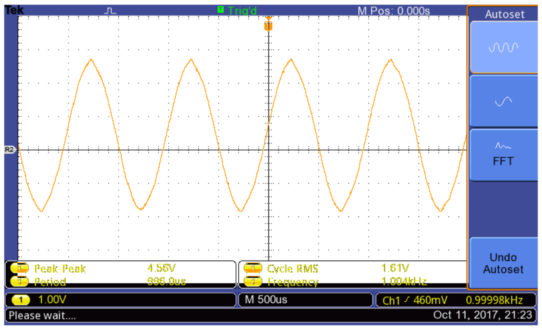


Figure 38: Signal coming out from the low-pass filter at the receiving part.

The signal was then set to the maximum amplitude where it didn't distort. The transfer 3dB bandwidth was then observed. It turned out to be from 0 to 3.3 kHz.

In the next step a microphone was connected to the transmitting side of the apparatus. The sound that then came from the receiving side can be described as noisy (weak noise). In this stage the system quantifies the audio signal with 8-bits. By grounding n number of bits A_1 to A_8, the sound quality was tested. It turned out that with only two to three bits the sound quality was good enough that one could hear what a person says.

4 Discussion

This project ended up in a functional audio transmission system. It did however not achieve the exact specifications, mostly due to that the bandwidth of the transmission was only 3.3 kHz. We think that this was caused due to under sampling. Even though we for-filled the Nyquist criteria, the sampling "chopped" of the amplitude of the signal, so in a next version we would adjust to sampling with a higher sampling rate.

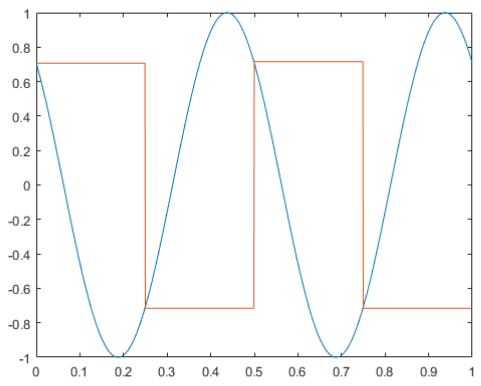
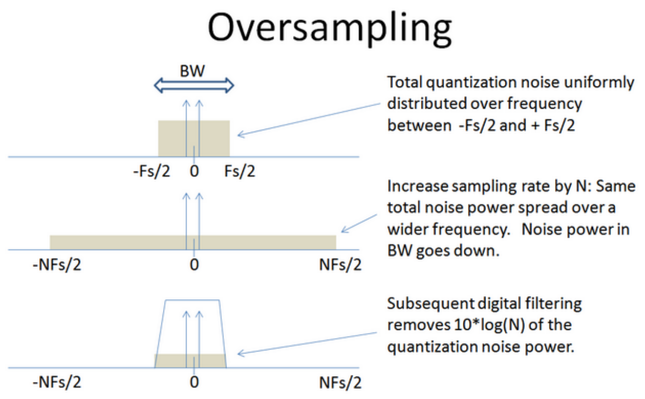


Figure 39: When the sampling frequency is exactly 2 times the sampled sine, a sine with correct frequency can be reproduced but not necessary correct amplitude.

Please see figure 39 for further explanation.

When we looked at the received signal in frequency spectrum, we could also detect a 12 kHz component, which was also herd during the test. A solution to this problem could be to apply a notch filter that removes this particular frequency, or raise the sampling frequency in order to move this component outside the filter bandwidth. Another benefit of "oversampling" is that we spread out the noise over a greater span, and consequently improves the Signal to Noise Ratio (SNR).



Oversampling
BW
Total quantization noise uniformly distributed over frequency between -Fs/2 and + Fs/2
-Fs/2
0
Fs/2
Increase sampling rate by N: Same total noise power spread over a wider frequency. Noise power in BW goes down.
-NFs/2
0
NFs/2
Subsequent digital filtering removes 10*log(N) of the quantization noise power.
-NFs/2
0
NFs/2

Figure 40: By spreading out the noise, we improve SNR.

Another function that would be nice to have would be a volume control. The easiest way to implement this would be to replace R_4 in the power amplifier, see figure 31, with a potentiometer. This would facilitate evaluating the system, since the user can adjust the volume to a satisfying level.

Unfortunately we never had time to setup our system as a receiver and test this. Since all of our subsystems have been proved functional in earlier test, we can confidentially assume that we would also be able to configure our solution into a receiver.

5 Reflection

As a start the laborations have been very educational and we have been able to combine skills from earlier courses. The level and tempo have been very reasonable. By this, we mean that the preparation tasks have been just enough challenging so we would both learn and comprehend what we've been taught. Also the time scheduled for laborations have been resoanable; sometimes we could leave early, when everything was working as it should be, but when we had problems, we had time to troubleshoot and correct some mistakes (or sometimes find cables not being fully inserted in the breadboard...)

One interesting option for next years students would be to have an optional lab, where some improvements or ideas of improvements could be tested and implemented, such as volume control and oversampling as discussed earlier. In order to motivate the student to do this, a grade system for the laborative part of the course could be applied for next year. This would for you have the benefit of getting feedback and ideas for future years courses.

We appreciated the increase of teaching assistants being made mid-course and also the system where the ques where divided between help and check.

References

[1] *DAC0808*. Datasheet. 1999.

[2] Bengt Molin. *Analog elektronik*. Studentlitteratur, 2013.

[3] *SSY011 Elektriska system – Laborationer ht 2017*. PDF. Aug. 2017.

[4] *TL07xx Low-Noise JFET-Input Operational Amplifiers*. 2017.

[5] *Understanding Poles and Zero*. PDF. URL: `http://web.mit.edu/2.14/www/Handouts/PoleZero.pdf`.

Appendix

In this section follows all the source code used in the lab. The names refers to which preparation task they were made for. For easy understanding and indexing, please see Lab-PM.

Listing 1: Source code for preparation task 1.1

```
library ieee;
use ieee.std_logic_1164.all;

entity bin_to_dec is port(
        sw0,sw1,sw2,sw3: in std_logic;
        ledr0, ledr1, ledr2, ledr3, ledr4,
        ledr5, ledr6, ledr7, ledr8, ledr9,
        ledg7: out std_logic);
end entity;


architecture struct of bin_to_dec is
        signal sw: std_logic_vector(3 downto 0);
begin
        sw <= sw3 & sw2 & sw1 &sw0;

        ledr0 <= '1' when sw = "0000" else '0';
        ledr1 <= '1' when sw = "0001" else '0';
        ledr2 <= '1' when sw = "0010" else '0';
        ledr3 <= '1' when sw = "0011" else '0';
        ledr4 <= '1' when sw = "0100" else '0';
        ledr5 <= '1' when sw = "0101" else '0';
        ledr6 <= '1' when sw = "0110" else '0';
        ledr7 <= '1' when sw = "0111" else '0';
        ledr8 <= '1' when sw = "1000" else '0';
        ledr9 <= '1' when sw = "1001" else '0';
        ledg7 <= '1' when sw > "1001" else '0';


end struct;
```

Listing 2: Source code for preparation task 1.2

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

-- Entity declared in DEL_1_3.vhd

architecture behaviour of timer is
        signal counter: std_logic_vector (26 downto 0);
begin
        process(clk50, reset)
        begin
                if reset = '1' then
                        counter <= (others => '0');
                        ts <= '1';
                elsif rising_edge(clk50) then
                        if counter = 52082 then
                                ts<= '1';
                                counter <= (others => '0');
                        else
                                counter <=counter +1;
                                if counter <= 2604 then
                                        ts <= '1';
                                else
                                        ts <='0';
                                end if;
                        end if;
                end if;
        end process;
end architecture;
```

Listing 3: Source code for preparation task 1.3

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity timer is port(
        clk50, reset: in std_logic;
        tb,ts: out std_logic);
end entity;

architecture behavior of timer is
        signal counter1: std_logic_vector (15 downto 0);
        signal counter2: std_logic_vector (15 downto 0);
begin
        ---- 960 Hz -----
        process(clk50, reset)
        begin
                if reset = '1' then
                        counter1 <= (others => '0');
                        ts <= '1';
                elsif rising_edge(clk50) then
                        if counter1 = 52079 then
                                ts <= '1';
                                counter1 <= (others => '0');
                                --counter2 <= (others => '0'); -- reset 9600 Hz timer to,
                        else
                                counter1 <=counter1 +1;
                                if counter1 <= 2603 then
                                        ts <= '1';
                                else
                                        ts <='0';
                                end if;
                        end if;
                end if;
        end process;
        ---- 9600 Hz ----
        process(clk50, reset)
        begin
```

```
                if reset = '1' then
                        counter2 <= (others => '0');
                        tb <= '1';
                elsif rising_edge(clk50) then
                        if counter2 = 5207 then
                                tb <= '1';
                                counter2 <= (others => '0');
                        else
                                counter2 <= counter2 +1;
                                tb <='0';
                        end if;
                end if;
        end process;

end architecture;
```

Listing 4: Source code for preparation task 1.4

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity parallel_to_serial is port(
        clk50, reset: in std_logic;
        b: in std_logic_vector (7 downto 0);
        So: out std_logic);
end entity;

architecture behavior of parallel_to_serial is
        signal counter1: std_logic_vector (15 downto 0);
        signal counter2: std_logic_vector (15 downto 0);
        signal sample: std_logic_vector (7 downto 0);
        signal tb,ts: std_logic;
begin
        ---- 960 Hz -----
        process(clk50, reset)
        begin
                if reset = '1' then
                        counter1 <= (others => '0');
                        ts <= '1';
                elsif rising_edge(clk50) then
                        if counter1 = 52079 then
                                ts <= '1';
                                counter1 <= (others => '0');
                                --counter2 <= (others => '0'); -- reset 9600 Hz timer to,
                        else
                                counter1 <=counter1 +1;
                                if counter1 <= 2603 then
                                        ts <= '1';
                                else
                                        ts <='0';
                                end if;
                        end if;
                end if;
        end process;
        ---- 9600 Hz ----
        process(clk50, reset)
        begin
                if reset = '1' then
                        counter2 <= (others => '0');
                        tb <= '1';
                elsif rising_edge(clk50) then
                        if counter2 = 5207 then
                                tb <= '1';
                                counter2 <= (others => '0');
                        else
                                counter2 <= counter2 +1;
                                tb <='0';
                        end if;
                end if;
        end process;
        ---- Shift register (Parallell in, serial out) ----
        process(tb, ts, b)
        begin
                if reset = '1' then
                        So <= '0';
                elsif ts = '1' then
                        sample <= b;
                        So <= '0';
                elsif tb = '1' then
                        So <= sample(0);
                        sample <= '1' & sample(7 downto 1);
                end if;



        end process;

end architecture;
```

Listing 5: Source code for preparation task 2.4

```
library ieee;
use ieee.std_logic_1164.all;

entity switch_led_n_port is port(
        sw: in std_logic_vector (7 downto 0);
        ledr,
        A :out std_logic_vector (7 downto 0));
end entity;


architecture struct of switch_led_n_port is

begin

        ledr <= sw;
        A <= sw;
```

```
end struct;
```

Listing 6: Source code for preparation task 2.6

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity AD_ctrl is port(

        clk50, reset: in std_logic;
        tb: out std_logic;
        sw9: in std_logic;
        ledr,
        A :out std_logic_vector (7 downto 0));

end entity;

architecture behavior of AD_ctrl is
        signal counter1: std_logic_vector (19 downto 0);
        signal direction: std_logic;
        signal tb_internal: std_logic;
        signal u8int : std_logic_vector(7 downto 0);

begin
        ---- Counter ----
        process(clk50, tb_internal, reset)
        begin
                if reset = '0' then
                        u8int <= (others => '0');
                elsif rising_edge(clk50) then
                        if tb_internal = '1' then
                                if sw9 = '1' then
                                        u8int <= u8int +1; -- Create saw tooth. The vector
                                else
                                        u8int <= u8int -1;
                                end if;
                        end if;
                end if;
        end process;
        ---- 100 Hz ----
        process(clk50, reset)
        begin
                if reset = '0' then
                        counter1 <= (others => '0');
                        tb_internal <= '1';
                        tb <= '1';
                elsif rising_edge(clk50) then
                        if counter1 = 499999 then
                                tb <= '1';
                                tb_internal <= '1';
                                counter1 <= (others => '0');
                        else
                                counter1 <= counter1 +1;
                                tb <='0';
                                tb_internal <= '0';
                        end if;
                end if;
        end process;
        ---- Set outputs ----
        ledr <= u8int;
        A <= u8int;

end architecture;
```

Listing 7: Source code for preparation task 3.4

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity SAR is port(
        D, clk50, reset: in std_logic;
        Qout, LEDR :out std_logic_vector (7 downto 0));
end entity;

architecture struct of SAR is
        type StateType is (U,IDLE, SAR0, SAR1, SAR2, SAR3, SAR4, SAR5, SAR6, SAR7);
        signal state: StateType;

        signal counter1: std_logic_vector (15 downto 0);
        signal counter2: std_logic_vector (15 downto 0);
        signal ts: std_logic;
        signal tb: std_logic;
        signal Q: std_logic_vector (7 downto 0);

begin
        ---- SAR statemachine ---
        process(clk50, reset, ts, tb)
        begin
                if reset =   0   then
                        state <= IDLE;
                elsif rising_edge(clk50) then
                        if tb = '1' then
                                case state is
                                        when IDLE =>
                                                if ts = '1' then
                                                        Q <= "10000000";
                                                        state <= SAR7;
                                                else
                                                        state <= IDLE;
                                                end if;
                                        when SAR7 =>
                                                Q(7) <= D;
                                                Q(6) <= '1';
```

```
                    state <= SAR6;
                when SAR6 =>
                    Q(6) <= D;
                    Q(5) <= '1';
                    state <= SAR5;
                when SAR5 =>
                    Q(5) <= D;
                    Q(4) <= '1';
                    state <= SAR4;
                when SAR4 =>
                    Q(4) <= D;
                    Q(3) <= '1';
                    state <= SAR3;
                when SAR3 =>
                    Q(3) <= D;
                    Q(2) <= '1';
                    state <= SAR2;
                when SAR2 =>
                    Q(2) <= D;
                    Q(1) <= '1';
                    state <= SAR1;
                when SAR1 =>
                    Q(1) <= D;
                    Q(0) <= '1';
                    state <= SAR0;
                when SAR0 =>
                    Q(0) <= D;
                    state <= IDLE;
                when others =>  --- undefined state, shouldn't be 
                    state <=IDLE;
            end case;
        end if;
    end if;
end process;

---- 960 Hz ------
process(clk50, reset)
begin
    if reset =   0   then
        counter1 <= (others => '0');
        ts <= '1';
    elsif rising_edge(clk50) then
        if counter1 = 52079 then
            ts <= '1';
            counter1 <= (others => '0');
            --counter2 <= (others => '0'); -- reset 9600 Hz timer to,
        else
            counter1 <=counter1 +1;
            if counter1 <= 2603 then
                ts <= '1';
            else
                ts <='0';
            end if;
        end if;
```

```
                end if;
        end process;

        ---- 9600 Hz ----
        process(clk50, reset)
        begin
                if reset =    0    then
                        counter2 <= (others => '0');
                        tb <= '1';
                elsif rising_edge(clk50) then
                        if counter2 = 5207 then
                                tb <= '1';
                                counter2 <= (others => '0');
                        else
                                counter2 <= counter2 +1;
                                tb  <='0';
                        end if;
                end if;
        end process;

        ---- Put out outputs ----
        ledr <= Q;
        Qout <= Q;

end struct;
```

Listing 8: Source code for preparation task 3.7

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity SAR_PS is port(
        D, clk50, reset: in std_logic;
        So: out std_logic;
        Qout:out std_logic_vector (7 downto 0));
end entity;

architecture struct of SAR_PS is
        type StateType is (U,IDLE, SAR0, SAR1, SAR2, SAR3, SAR4, SAR5, SAR6, SAR7, SAVE);
        signal state: StateType;

        signal counter1: std_logic_vector (15 downto 0);
        signal counter2: std_logic_vector (15 downto 0);
        signal ts: std_logic;
        signal tb: std_logic;
        signal Q: std_logic_vector (7 downto 0);
        signal midsample, sample: std_logic_vector (7 downto 0);

begin
        ---- SAR statemachine ---
        process(clk50, reset, ts, tb)
        begin
                if reset = '0' then
                        state <= IDLE;
        elsif rising_edge(clk50) then
                if tb = '1' then
                        case state is
                                when IDLE =>
                                        if ts = '1' then
                                                Q <= "10000000";
                                                state <= SAR7;
                                        else
                                                state <= IDLE;
                                        end if;
                                when SAR7 =>
                                        Q(7) <= D;
                                        Q(6) <= '1';
                                        state <= SAR6;
                                when SAR6 =>
                                        Q(6) <= D;
                                        Q(5) <= '1';
                                        state <= SAR5;
                                when SAR5 =>
                                        Q(5) <= D;
                                        Q(4) <= '1';
                                        state <= SAR4;
                                when SAR4 =>
                                        Q(4) <= D;
                                        Q(3) <= '1';
                                        state <= SAR3;
                                when SAR3 =>
                                        Q(3) <= D;
                                        Q(2) <= '1';
                                        state <= SAR2;
                                when SAR2 =>
                                        Q(2) <= D;
                                        Q(1) <= '1';
                                        state <= SAR1;
                                when SAR1 =>
                                        Q(1) <= D;
                                        Q(0) <= '1';
                                        state <= SAR0;
                                when SAR0 =>
                                        Q(0) <= D;
                                        state <= SAVE;
                                when SAVE =>
                                        midsample <= Q;
                                        state <= IDLE;
                                when others => -- undefined state, shouldn't be
                                        state <=IDLE;
                        end case;
                end if;
        end if;
end process;

---- 960 Hz ------
process(clk50, reset)
begin
```

```
        if reset ='0' then
                counter1 <= (others => '0');
                ts <= '1';
        elsif rising_edge(clk50) then
                if counter1 = 52079 then
                        ts <= '1';
                        counter1 <= (others => '0');
                else
                        counter1 <=counter1 +1;
                        if counter1 <= 2603 then
                                ts <= '1';
                        else
                                ts <='0';
                        end if;
                end if;
        end if;
end process;

---- 9600 Hz ----
process(clk50, reset)
begin
        if reset = '0' then
                counter2 <= (others => '0');
                tb <= '1';
        elsif rising_edge(clk50) then
                if counter2 = 5207 then
                        tb <= '1';
                        counter2 <= (others => '0');
                else
                        counter2 <= counter2 +1;
                        tb <='0';
                end if;
        end if;
end process;

---- Shift register (Parallell in, serial out) ----
process(tb, ts, midsample, clk50)
begin
        if reset = '0' then
                So <= '0';
        elsif rising_edge(clk50) then
                if ts = '1' then
                        sample <= midsample;
                        So <= '0';
                elsif tb = '1' then
                        So <= sample(0);
                        sample <= '1' & sample(7 downto 1);
                end if;
        end if;
end process;

---- Put out outputs ----

Qout <= Q;
```

```
end struct;
```

Listing 9: Source code for preparation task 4.5

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity serial_in_parallel_out is port(
        clk50, serial_in, reset: in std_logic;
        parallel, LEDR: out std_logic_vector (7 downto 0));
end entity;

architecture behaviour of serial_in_parallel_out is
        type StateType is (U, STOP, START, S0, S1, S2, S3, S4, S5, S6, S7);
        signal state, next_state : StateType;

        signal counter1 : std_logic_vector (13 downto 0);
        signal Q, next_Q, next_parallel: std_logic_vector (7 downto 0);
        signal serial_semisync, serial : std_logic;
begin
        ---- Internal clock, for baudrate 9600 ----
        process(clk50, reset)
        begin
                if reset = '0' then
                        counter1 <= (others => '0');
                        state <= STOP;
                        Q <= (others => '0');
                elsif rising_edge(clk50) then
                        counter1 <= counter1 + 1;
                        if state = STOP and serial = '1' then
                                counter1 <= (others => '0');
                        elsif counter1 = 2603 then -- 2604 ticks -> half period
                                state <= next_state;
                                Q <= next_Q;
                                parallel <= next_parallel;
                                LEDR <= next_parallel;
                        elsif counter1 = 5207 then
                                counter1 <= (others => '0');
                        end if;
                end if;
        end process;
        ----
        ---- Statemachine ----
        process(clk50, state, serial, Q)
        begin
                next_Q <= Q;
                case state is
                        when STOP =>
                                next_state <= START;
                        when START =>
                                next_state <= S0;
                                next_Q(0) <= serial;
                        when S0 =>
                                next_state <= S1;
                                next_Q(1) <= serial;
                        when S1 =>
                                next_state <= S2;
                                next_Q(2) <= serial;
                        when S2 =>
                                next_state <= S3;
                                next_Q(3) <= serial;
                        when S3 =>
                                next_state <= S4;
                                next_Q(4) <= serial;
                        when S4 =>
                                next_state <= S5;
                                next_Q(5) <= serial;
                        when S5 =>
                                next_state <= S6;
                                next_Q(6) <= serial;
                        when S6 =>
                                next_state <= S7;
                                next_Q(7) <= serial;
                        when S7 =>
                                next_parallel <= Q;
                                if serial = '0' then
                                        next_state <= S7;
                                else
                                        next_state <= STOP;
                                end if;
                        when others =>          -- Undefined state
                                next_state <= STOP;
                end case;
        end process;

        process(clk50, serial_in, serial, serial_semisync)
        begin
                if rising_edge(clk50) then
                        Serial_semisync <= Serial_in;
                        serial <= Serial_semisync;
                end if;
        end process;
        ----
        ---- Parallell code----
                -- none
end architecture;
```