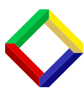

DIPLOMARBEIT

FPGA-basiertes RISC-V-Computersystem: YARM

Höhere Technische Bundeslehr- und Versuchsanstalt Anichstraße

Abteilung

Elektronik und technische Informatik

Ausgeführt im Schuljahr 2019/20 von:

Armin Brauns 5AHEL

Daniel Plank 5BHEL

Betreuer/Betreuerin:

Dipl.-Ing. Christoph Schönherr

Projektpartner: IT-Syndikat, Verein zur Förderung des freien Zugangs zu technischer Fort- und Weiterbildung jeglicher Art, Hackerspace Innsbruck

Ansprechpartner: Ing. David Oberhollenzer B.Sc.

Innsbruck, am 26. März 2020

Abgabevermerk:

Datum:

Betreuer/in:

Gendererklärung

Aus Gründen der besseren Lesbarkeit wird in dieser Diplomarbeit die Sprachform des generischen Maskulinums angewendet. Es wird an dieser Stelle darauf hingewiesen, dass die ausschließliche Verwendung der männlichen Form geschlechtsunabhängig verstanden werden soll.

This thesis is written in the language form if the generic masculin for improved readability. It is pointed out that all masculin-only uses may and should be interpreted as gender neutral.

Kurzfassung/Abstract

Diese Diplomarbeit beschäftigt sich mit der Arbeitsweise von Prozessoren und Prozessorperipherie in moderner und traditioneller Form. Sie versucht anschaulich den Aufbau eines Computersystems in Hard- und Software veranschaulichen sowie diesen erklären. Dafür wurde auf einem XILINX FPGA ein RISC-V32I Prozessor in VHDL implementiert sowie diverse Parallelbus gebundene Hardwareperipherie entwickelt und gebaut. Als Harwareperipherie wurde ein 8-Bit 2-Kanal DAC und eine serielle Schnittstelle mit TIA-/EIA-232 Pegeln gebaut. Der Prozessor implementiert das RISC-V32I base instruction set. Aufgrund der starken Verwendung von Englisch im Software- und Hardwarebereich wurde diese Diplomarbeit in Englisch verfasst, was ebenfalls die Lesbarkeit erhöhen soll. Die entstandene Dokumentation soll für Menschen mit einem Grundlegenden Verständnis von Elektronik sowie der Hardware-Beschreibungssprache VHDL verständlich sein.

This diploma thesis deals with the operation of processors and their orresponding peripherials in modern andd traditional forms. It attempts to illustrate the structure of a computersystem in hard- and software. To reach this goal a RISC-V32I processor has been implemented in VHDL on a XILINX FPGA as well as some peripherials bound to the parallel bus. These peripherials include a 2-channel 8-bit Digital to analog converter as well as a TIA-/EIA-232 compliant serial interface. Due to the common use of english in the hardware and software engineering field this thesis was written in english, which should enhance readability as well. The written documentation should be understandable for everyone with a basic understanding of electronics as well as the hardware description language VHDL.

Result

The project was fully implemented with all functionality originally targeted. The system has been tested and verified and all example code have been documented and tested as running. Implementations in hardware were made in open-source programs and the RISC-V processor can compile using an open source toolchain. The completed project can be found on the USB stick which accompanies this thesis, or in the git repositories at `https://git.it-syndikat.org/tyrolyean/dipl.git` and `https://gitlab.com/YARM-project/`.

Contents

Gendererklärung i
Kurzfassung/Abstract ii
Result iii
1 Task description 1
1.1 Hardware 1
2 Hardware peripherials 2
2.1 Parallel bus 2
2.1.1 Address Bus 2
2.2 Data Bus 3
2.3 Control Bus 3
2.3.1 Master Reset 3
2.3.2 Write Not 3
2.3.3 Read Not 3
2.3.4 Module Select 1 and 2 Not 3
2.4 Testing and Measurement 4
2.4.1 Measurements 4
2.4.2 Testing 4
2.5 Backplane 5
2.5.1 Termination resistors 5
2.6 Case 6
2.7 Serial Console 8
2.7.1 16550 UART 8
2.7.2 MAX-232 9
2.7.3 Schematics 9
2.7.4 Demonstration Software 13
2.8 Audio Digital-Analog-Converter 17
2.8.1 TLC 7528 Dual R2R Ladder DAC 18
2.8.2 IDT7201 CMOS FIFO Buffer 18
2.8.3 Theory verfication 19
2.8.4 Schematics 20
2.8.5 Demonstration Software 23
3 Addressing DACA and DACB 26
3.1 FPGA to Hardware interface 27
3.1.1 Measurement error 29
4 Textadventure 30
4.1 General Implementation details 30

4.1.1 General definitions and pinout of the AVR 30
4.1.2 Read and Write routines 32
4.1.3 UART and DAC update polling 32
4.2 DAC sound generation 33
4.2.1 DAC modes 33
4.2.2 Tones and Tracks 37
4.2.3 Track switching 42
4.3 User command interpretation 42
4.3.1 Command structure and parsing 42
4.3.2 Command parameters 43
4.4 Gameplay 45
4.5 Memory constraints 46

5 Erklärung der Eigenständigkeit der Arbeit 48

I List of Figures I

II List of Tables I

III Listings I

Anhang IV

1 Task description

1.1 Hardware

Due to the recurring questions in the environment of the Hackerspace Innsbruck about the internal workings of a computer system and the lack of material to demonstrate these, hardware should be developed for educational purposes. This hardware should not be to complex to understand but still demonstrate basic tasks of a computer system. The targeted computing tasks are human interface device controllers, under which a **D**igital to **A**nalog **C**onverter[A] and a serial console with TIA-/EIA-232 compliant voltage levels were chosen. For these peripherials schematics and a working implementation in the hardware building style of the hackerspace should be built. All nescessary hardware will be provided by the Hackerspace. If possible already present hardware should be used, if impossible new one will be ordered. All schematics should, whenether possible be written in open-source software such as Kicad or GNU-EDA.

If possible software-examples should be written as well, though the complexity of these was coupled to the time left to spend on the project. Software should be written in C, the coding convention is left to the implementer.

[A]From now on reffered to simply as DAC

2 Hardware peripherials

2.1 Parallel bus

The core part of the hardware is the interface between the microprocessor and the hardware peripherials. This bus is delivering data in parallel and is therefore named the “parallel bus“. This bus has 3 different sub-parts:

1. The address bus
2. The data bus
3. The control bus

This split is common in many computer architectures and bus systems used by various microprocessor manufacturers. In figure i the layout of the Atari Parallel Bus Interface is shown as used on the Atari 800XL.

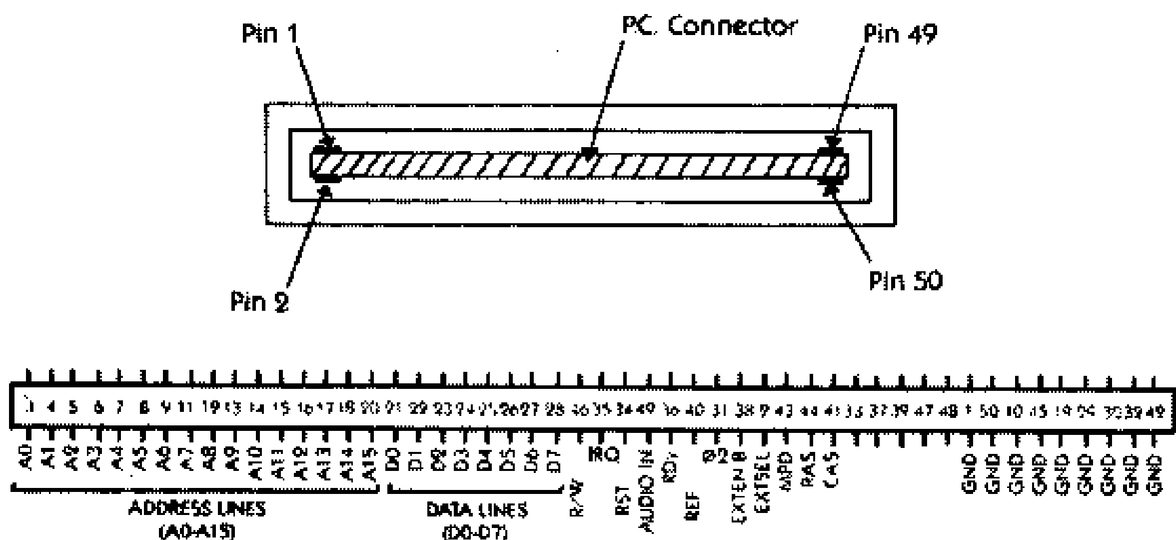

Figure i: Atari PBI Pinout;Source: `https://www.atarimagazines.com`

2.1.1 Address Bus

The address bus contains the nescessary data lines for addressing the individual registers of the Serial connection and the uart. On any modern system this bus is from 16 to 64 bits wide. For our implementation the bus size was chosen to be 8 bit, which is multiple times the amount of needed address space, but is the smallest addressable

unit on most microcontroller architectures and therefore easy to program with. The address bus is unidirectional.

2.2 Data Bus

The data bus contains the actual data to be stored to and read from registers. The data bus is, as well on most systems a multiple of 16 bits wide, but for the same reasons as the data bus, was shrunk down in our case to 8 bits. The data bus is bidirectional.

2.3 Control Bus

Control bus is a term which referes to any control lines (such as read and write lines or clock lines) which are neither address nor data bus. The control bus in our case needed to be 5 bits wide and consists of:

- MR ... Master Reset
- $\neg WR$ ... Write Not
- $\neg RD$ ... Read Not
- $\neg MS1$ ... Module Select 1 Not
- $\neg MS2$ ... Module Select 2 Not

2.3.1 Master Reset

A high level on the MR lane signals to the peripherials that a reset of all registers and states should occure. This is needed for the serial console and the dac.

2.3.2 Write Not

A low level on the $\neg WR$ lane signals the corresponding modules that the data on the data bus should be written to the register on the address specified from the address bus.

2.3.3 Read Not

A low level on the $\neg RD$ lane signals the corresponding modules that the data from the register specified by the address on the address bus should be written to the data bus.

2.3.4 Module Select 1 and 2 Not

A low level on one of these lines signals the corresponding module that the data on address data and the control lines is meant for it.

2.4 Testing and Measurement

For functional testing and verification of implementation goals, measurements needed to be performed invarious different ways and testing software was required.

2.4.1 Measurements

Measurements were performed, if not noted otherwise, with the Analog Discovery 2 from Digilent as it has 16bit digital I/O Pins as well a a Waveform generator and 2 differential oszilloscope inputs[1]. These were for all nescessary measurements enough. Though due to the size and construction of the device, which can be seen in figure ii errors wer encountered while performing the measurements. These are noted on occurance.

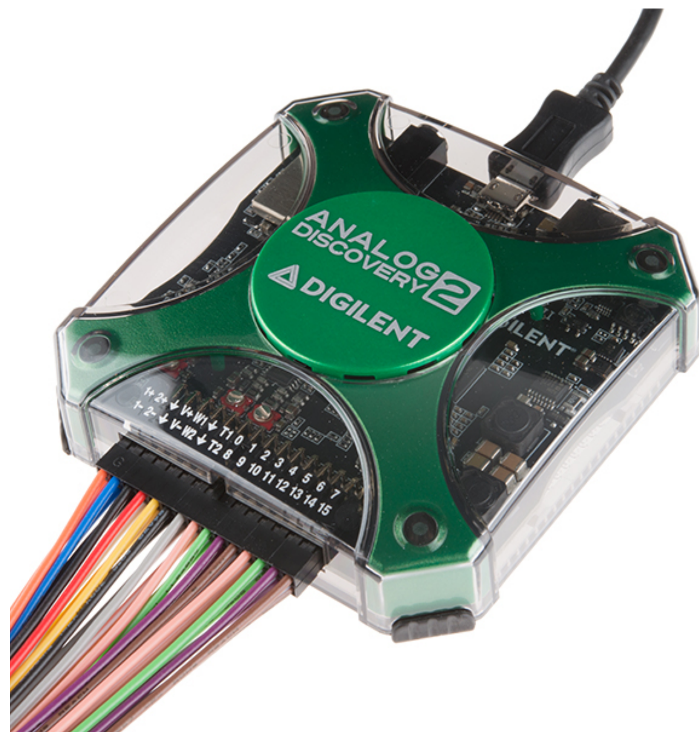

Figure ii: Digilent Analog Discovery 2;Source: `https://www.sparkfun.com/`

2.4.2 Testing

All testing was performed with an Atmel ATMega2560 due to it’s large amount of I/O pins, 5V I/O which is the more common voltage level on CMOS peripherials, way of addressing pins (8 at a time) and availability. [2] All testing software was written for this ATMega and compiled using the avr-gcc from the GNU-Project.

2.5 Backplane

To connect the modules to the microprocessor, many pins need to be connected straight through. For this purpose a backplane was chosen where DIN41612 connectors can be used. These connectors were chosen for their large pin count (96 pins) and their availability. The backplane connects all 96-pins straight through. With the 6 outer left and right pins connected for VCC and ground, as can be seen in figure iii.

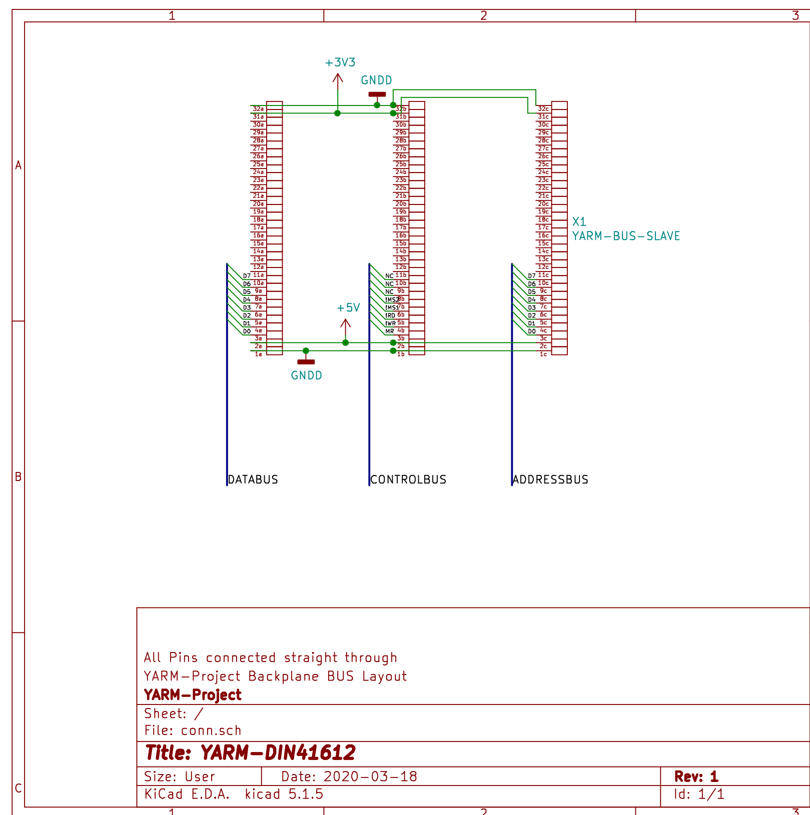

Figure iii: Layout of the DIN41612 Connectors on the Backplane

2.5.1 Termination resistors

In constrast to other systems using this backplane, no termination resistors were used. This makes the bus more prone to refelctions, however these were not a problem during development with the maximum transmission rate of 1MHz, as can be seen in the sample recording in figure iv

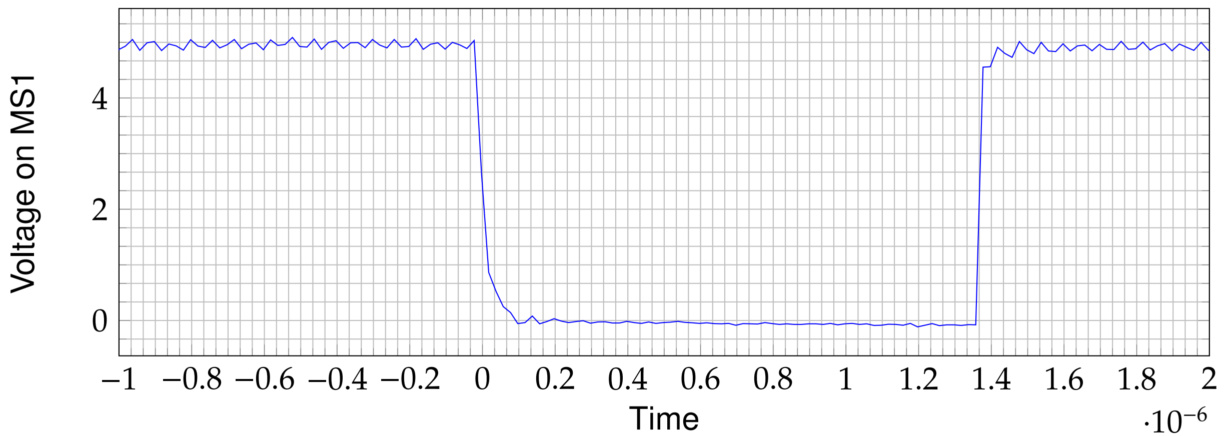

Figure iv: Measurement at around 1MHz bus clock on MS1

The ripple seen in figure iv are most likely due to the sample rate of the Oszilloscope, which is around 10Mhz after an average filter has been applied. The measurement was performed on the finished project, with all cards installed.

2.6 Case

The case for the backplane was provided by the hackerspace, and is meant for installation in a rack. The case is meant for installation of cards in the EUROCARD format, therfore all modules were built by this formfactor.

Figure v: The case with installed backplane

2.7 Serial Console

One core part of any computer systems is it's way to get human input. On older systems, and even today on server machines, this is done via a serial console. On this serial console, characters are transmitted in serial, which means bit by bit over the same line. The voltage levels used in these systems vary from 5V to 3.3V or +-10V. The most common standart for these voltage levels is the former RS-232[B] or as it should be called now TIA-[C]/EIA-[D]232. Voltage- levels as per TIA-/EIA Standard are not practical to handle over short distances to handle however, so other voltages are used on most interface chips and need to be converted.

2.7.1 16550 UART

The 16550 UART[E] is a very common interface chip for serial communications. It produces 5V logic levels as output on TX and needs the same as input on RX. Thoug common for a UART, these voltage levels need to be converted to TIA-/EIA-232 levels for a more common interface.

The 16550 UART is, in it's core a 16450 UART, but has been given a FIFO [F] buffer. It needs three address lines, and 8 data lines, which can be seen in figure vi

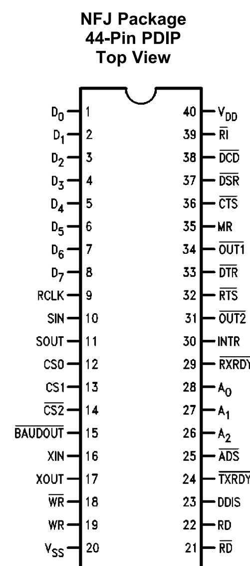

Figure vi: PC-16550D Pinout[3]

In figure vi the most important lanes are the SIN and sout lanes, as they contain the serial data to and from the 16550 UART.

[B]RS... Recommended Standard
[C]TIA...Telecommunications Industry Association
[D]EIA.. Electronic Industries Alliance
[E]Uinversal Asynchronous Receiver and Transmitter
[F]First-In First-Out

2.7.2 MAX-232

To convert the voltage levels of the 16550 UART to levels compliant wit TIA-/EIA-232 levels, the MAX-232 is used. It has two transmitters and two receivers side and generates the needed voltage levels via an internal voltage pump[4].

2.7.3 Schematics

Based on the descriptions in the datasheets the schematic in figure vii was developed.

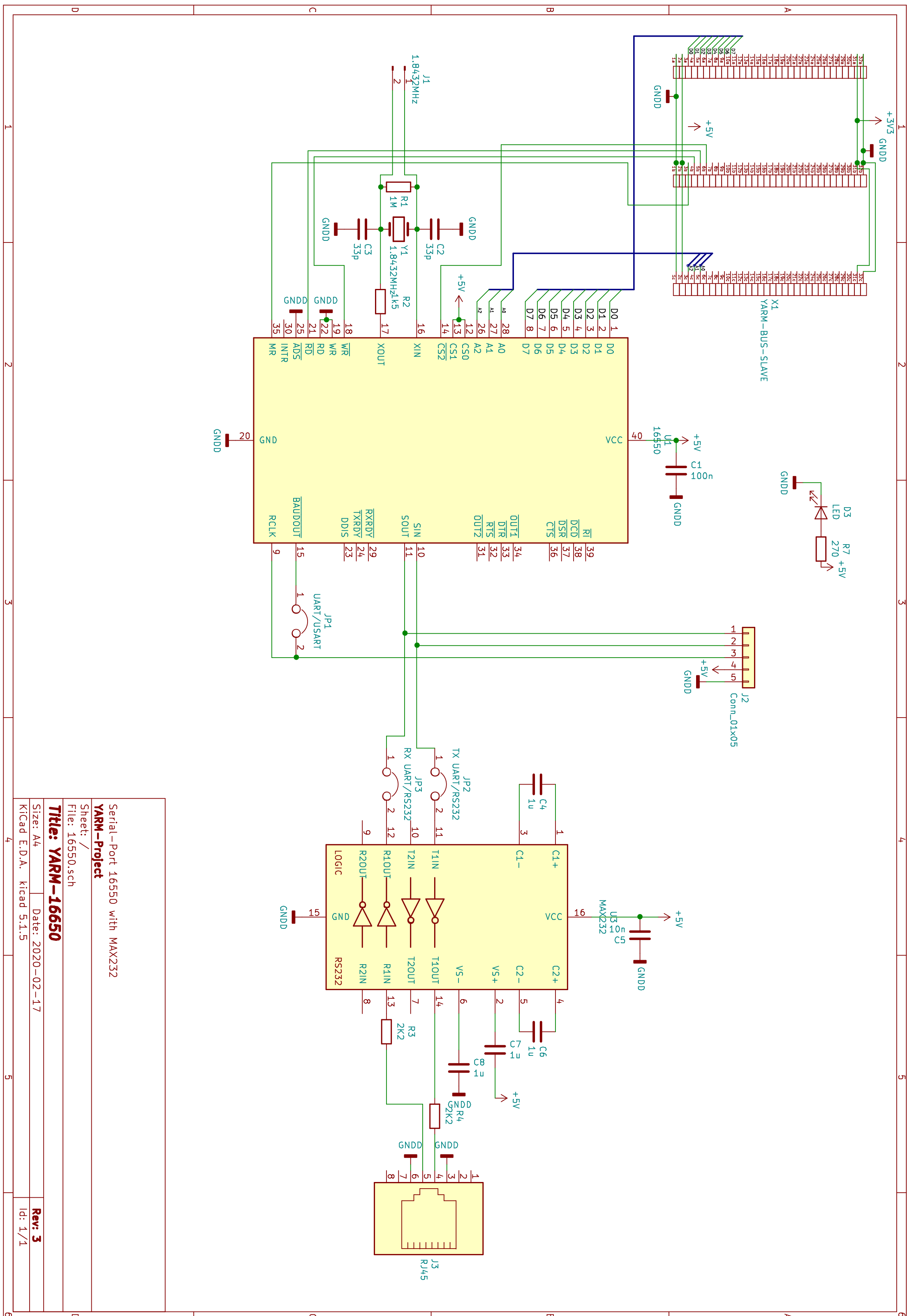

Figure vii: The schematic of the UART Module

Element Description The quartz oszillator Y1 is the clock source for the Baud Rate generation and was chosen with 1.8432 MHz for availability reasons and because it is the lowest ozillator from which all common baud rates can still be derived from [3]. Resistors R1 and R2 are for stability and functionality of the Oszillator nescessary as per datasheet. The resulting frequency can bemeasured via J1, the measurement can be seen in viii. C1 is used to stabilize the voltage for the 16550 UART and is common practice. Via JP1 the UART can be transformed into a USRT where the receiver is synchronized to the transmitter via a clock line. This mode has, however, not been tested, and the clock needs to be 16 times the receiver clock rate[3]. The final output of the 16550 UART can be used and measured via J2, as shown in figure ix . Before the UART on J2 can be use however, the Jumpers JP2 and JP3 need to be removed as otherwise the MAX-232 will short out with the incoming signal. capacitors C4, C6, C7, C7 and C8 are for the voltage pump as defined in the datasheet[4]. R4 and R5 have been suggested by the supervisor in order toavoid damage to the MAX-232. The RJ-45 plug is used to transmit the TIA-/EIA-232 signal, rather than the more common D-SUB connector, because the RJ-45 connector fits on a 2.54mm grid. The Pinout onthe RJ-45 plug can be seen in figure x. C5 has the same functionality for the MAX-232 as the C1 has to the 16550-UART.

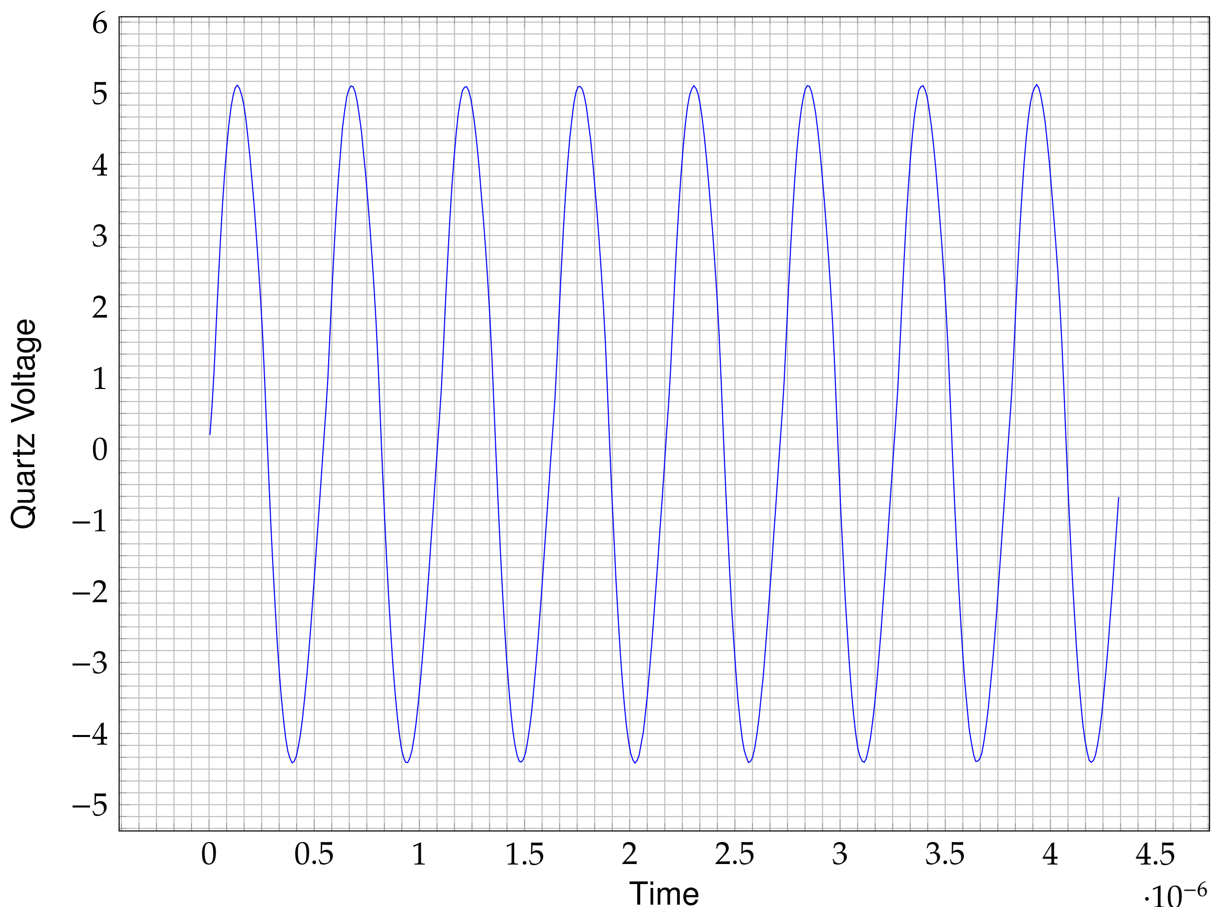

Figure viii: Measurement of the 1.8432 MHz Output on J1

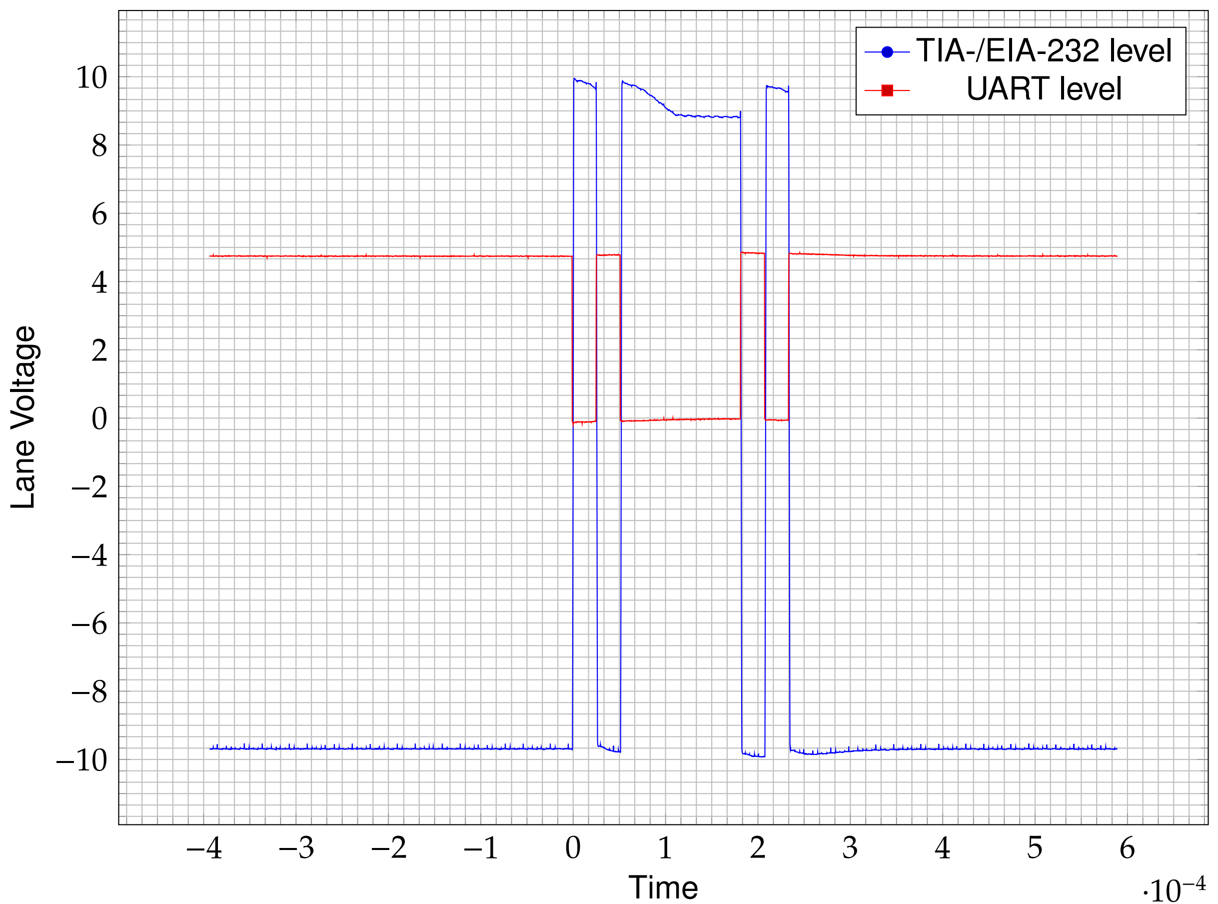

Figure ix: Measurement of a character transmission before and after MAX-232

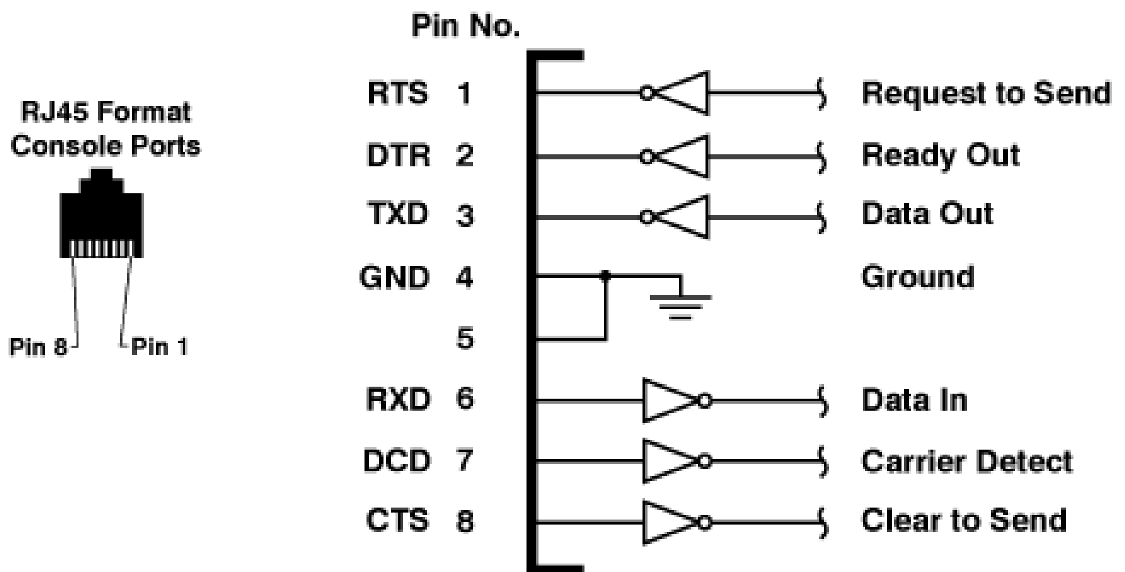

Figure x: Pinout of the RJ-45 Plug; Src: `https://www.wti.com/`

2.7.4 Demonstration Software

To demonstrate the functionality and prove, that the schematic has no underlying error, a program which regularly transmits a character was written as well as a simple echo program, which transmits all received characters. Both programs transmit 8 bit characters without parity at 38400 Baud. The output for program one can be seen in figure ix and the output for program two in figure xi.

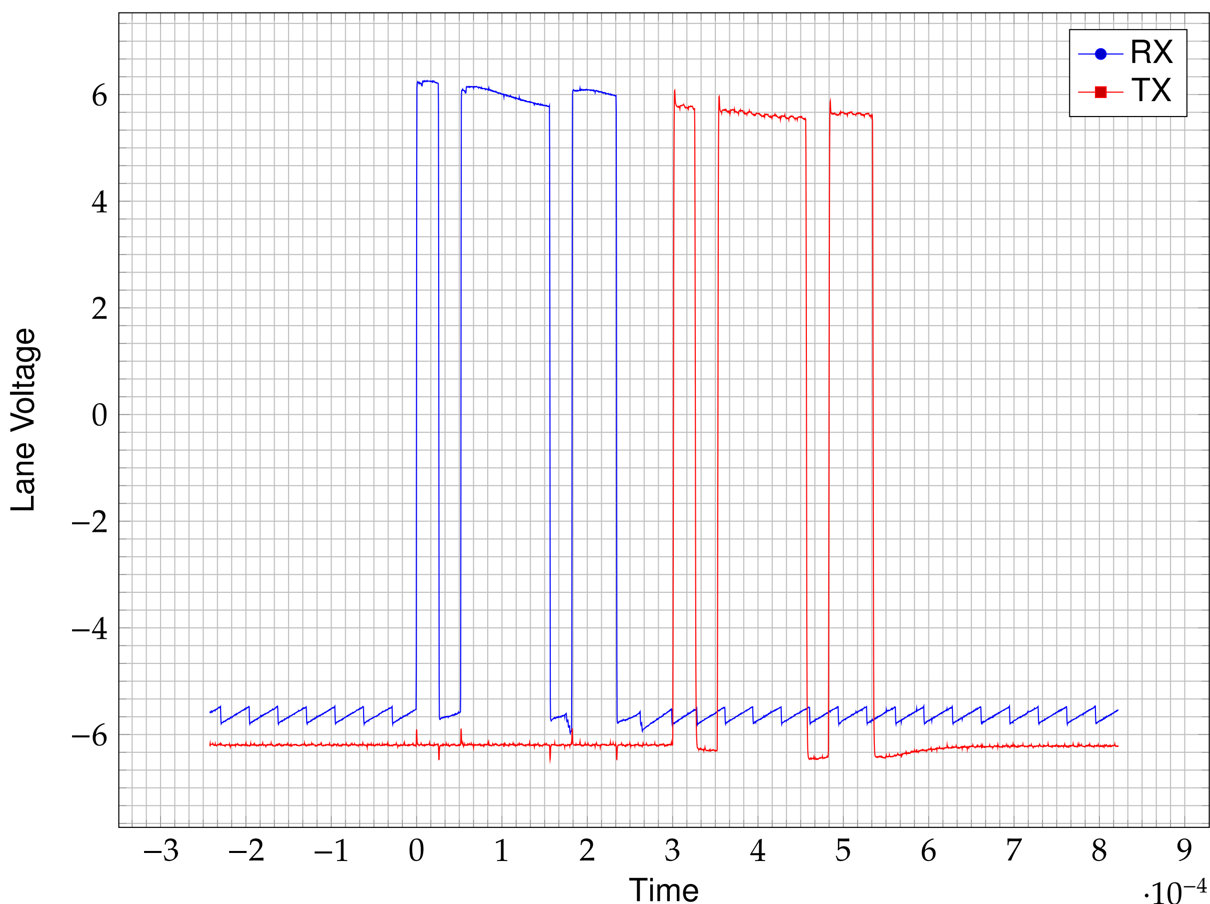

Figure xi: Measurement of a character echo

Transmit code The transmit code regularly transmits the letter capital A via the 16550 UART, but before it can do this it needs to perform some initialisations. The functions shown in listing I are the read and write routines for accessing the 16550 UART. These routines also apply to the echo code.

```
#define F_CPU 16000000UL

#include <stdint.h>
#include <util/delay.h>

#define BUS_HOLD_US 1

/* Shift values inside the PORTL Register */
```

```
#define WR_SHIFT 1
#define RD_SHIFT 2
#define MR_SHIFT 0
#define CS_SHIFT 3
#define CS_ADC_SHIFT 4

/* Registers in the 16550 UART */

#define UART_REG_DLLS   0
#define UART_REG_DLMS   1
#define UART_REG_TXRX   0
#define UART_REG_IER    1
#define UART_REG_IIR    2
#define UART_REG_LCR    3
#define UART_REG_MCR    4
#define UART_REG_LSR    5
#define UART_REG_MSR    6
#define UART_REG_SCR    7

void set_addr(uint8_t addr){

    PORTK = addr;
    return;
}

void write_to_16550(uint8_t addr, uint8_t data){


    set_addr(addr);
    DDRF = 0xFF;
    PORTL &= ~(1<<WR_SHIFT);
    PORTF = data;
    PORTL &= ~(1<<CS_SHIFT);

    _delay_us(BUS_HOLD_US);

    PORTL |= 1<<CS_SHIFT;
    set_addr(0x00);
    PORTL |= 1<<WR_SHIFT;
    PORTF = 0x00;
    return;
}

uint8_t read_from_16550(uint8_t addr){

    uint8_t data = 0x00;
    set_addr(addr);
```

```c
    DDRF = 0x00;
    PORTF = 0x00;
    PORTL &= ~(1<<RD_SHIFT);
    PORTL &= ~(1<<CS_SHIFT);
    _delay_us(BUS_HOLD_US);
    data = PINF;
    PORTL |= 1<<CS_SHIFT;
    set_addr(0x00);
    PORTL |= 1<<RD_SHIFT;
    DDRF = 0xFF;
    PORTF = 0x00;
    _delay_us(BUS_HOLD_US); /*Wait for the data and signal lanes to become
   stable*/
    return data;
}
```

Listing I: Read and write routines for the 16550 UART

To write to the 16550 UART, you need to perform some setup tasks. After startup, it requires a MR for at least 5ţs[3]. The baud rate divisor latch needs to be set to the specified divisor for the desired baud rate, and the character width and parity control needs to be set. The MR signal is beeing generated by the AVR on bootup. To access the divisor latch, the divisor latch access bit needs to be set and after setting up the baud rate divisor latch, it nees to be cleared to allow a regular transmission. This process can be seen in listing II

```c
int main(){

    /* Disable interrupts during initialisation phase */
    cli();

    /* Setup Data Direction Registers and populate with sane default
       values */
    DDRF = 0xFF; /* Data Bus */
    DDRK = 0xFF; /* Address Bus */
    DDRL = 0xFF; /* Control Bus */
    PORTF = 0x00;
    PORTK = 0x00;
    PORTL = 0x00;

    /* Cleanly reset the 16550 uart */
    PORTL |= (1<<WR_SHIFT);
    PORTL |= (1<<RD_SHIFT);
    PORTL |= (1<<CS_SHIFT);
    PORTL |= (1<<MR_SHIFT);
    _delay_us(100);
    PORTL &= ~(1<<MR_SHIFT);
```

```
22      _delay_us(1000);
23
24      sei();
25
26      for(;;){
27          write_to_16550(UART_REG_LCR,0x83);
28          write_to_16550(UART_REG_DLLS,0x03);
29          write_to_16550(UART_REG_DLMS,0x00);
30          write_to_16550(UART_REG_LCR,0x03);
31          write_to_16550(UART_REG_TXRX,'A');
32          _delay_us(10000);
33          }
34
35          return 0;
36  }
```

Listing II: 16550 INIT routines and single char transmission

The output of this code on the address, data and control bus as well as on the SOUT lane of the 16550 UART can be seen in figure xii

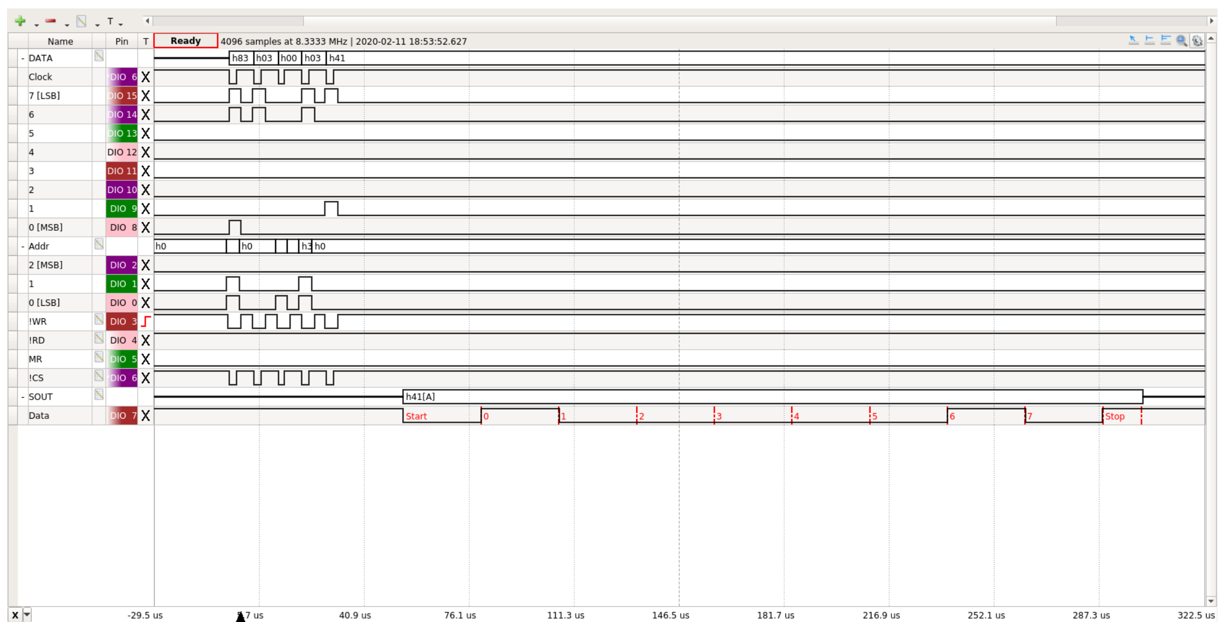

Figure xii: Transmission of character A via the 16550 UART

Echo code The echo code permanently polls the 16550 UART wether a character has been received, and if yes, reads it from the receiver holding register andwrites it back to the tx holding register. The output of this code can be seen in figure xi. The initialisation is practically the same as for the transmission code, as well as the read and write routines in listing I.

```
1  int main(){
```

```
    /* Disable interrupts during initialisation phase */
    cli();

    /* Setup Data Direction Registers and populate with sane default
       values */
    DDRF = 0xFF; /* Data Bus */
    DDRK = 0xFF; /* Address Bus */
    DDRL = 0xFF; /* Control Bus */

    /* Cleanly reset the 16550 uart */
    PORTL |= (1<<WR_SHIFT);
    PORTL |= (1<<RD_SHIFT);
    PORTL |= (1<<CS_SHIFT);
    PORTL |= (1<<CS_ADC_SHIFT);
    PORTL |= (1<<MR_SHIFT);
    _delay_us(100);
    PORTL &= ~(1<<MR_SHIFT);
    _delay_us(1000);

    write_to_16550(UART_REG_LCR,0x83);
    write_to_16550(UART_REG_DLLS,0x03);
    write_to_16550(UART_REG_DLMS,0x00);
    write_to_16550(UART_REG_LCR,0x03);
    for(;;){
        if(read_from_16550(UART_REG_LSR) & 0x01){
            write_to_16550(UART_REG_TXRX,
                read_from_16550(UART_REG_TXRX));
        }
    }

        return 0;
}
```

Listing III: 16550 character echo

2.8 Audio Digital-Analog-Converter

A digital to analog converter takes a digital number and converts it to a analog signal. The output of one such conversion is called a sample. With enough samples per second various different waveforms can be produced which, when amplified and put onto a speaker, can be heared by the human ear as a tone. With various tones in series a melody can be produced, which is what the DAC in this implementation does.

2.8.1 TLC 7528 Dual R2R Ladder DAC

The TLC 7528 is a Dual output Parallel input R2R Ladder DAC with a maximum sample rate of 10MHz[5] and which (should be) is monotonic over the entire D/A Conversion Range. The TLC-7528 was the only component chosen, where availability was not a factor, but rather due to it's design.It is the cheapest dual R2R Ladder dac which takes **PARALLEL** input, which was an important feature, because the backbone of the project is its parallel bus. Further the DAC was developed for audio aplications[5] obvious and the TLC-7528 was the only IC available as DIP [G], of which the pinout can be seen in figure xiii

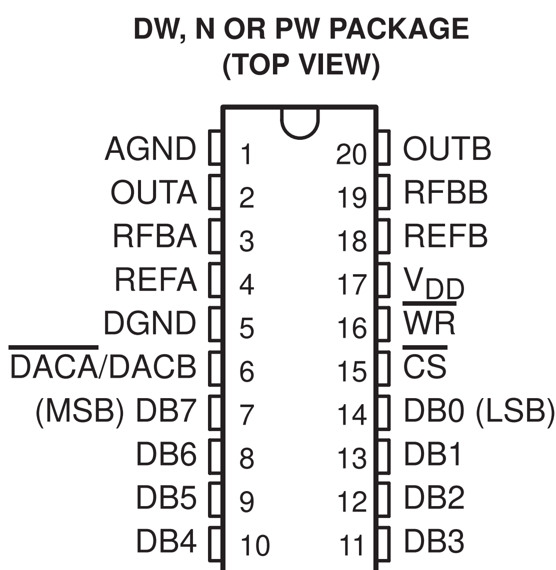

Figure xiii: TLC-7528 Pinout[5]

2.8.2 IDT7201 CMOS FIFO Buffer

The IDT7201 is an asychronous CMOS FIFO, which means that it can be read with a completely independant speed from which it is written and vice versa. It has 9 bit words, which can be seen in figure xiv, and can store up to 256 words[6]. It is used as a buffer to store data describing the targeted waveform in order to free time on the parallel bus for interaction with the 16550 UART.

[G]DIP... Dual Inline Package

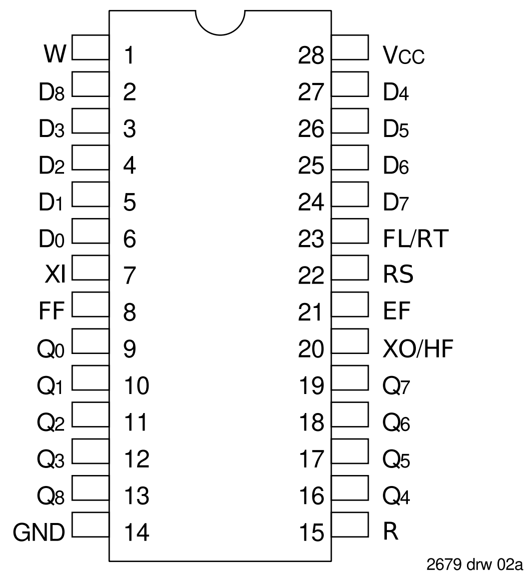

Figure xiv: IDT-7201 Pinout[6]

2.8.3 Theory verfication

Before tests of the complete unit were conducted, the functionality of the device and the validity of the knowledge of operations were performed. For that the DAC was directl connected to the ATMega without the FIFO infront of it. A saw was generated on only the DACA channel, which was put into voltage mode as described in the datasheet[5] and seen in figure xv. After the result seen in xvi was found a lot of effort was put in to determine the source of the heavy noise, however no obvious conclusion can be made, execpt that it comes from the DAC itself and is consistant over whatever frequency used. A damaged IC could be the reason or a sloppy production progress. Filters can be used to reduce the noise, however this was not done in this thesis, as the generated audio does not seem to suffer from these non-linearities as badly as when measured standalone.

voltage-mode operation

It is possible to operate the current multiplying D/A converter of these devices in a voltage mode. In the voltage mode, a fixed voltage is placed on the current output terminal. The analog output voltage is then available at the reference voltage terminal. Figure 11 is an example of a current multiplying D/A that operates in the voltage mode.

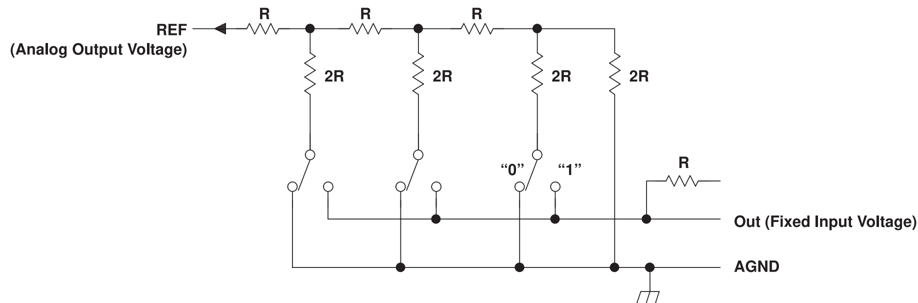

Figure 11. Voltage-Mode Operation

The following equation shows the relationship between the fixed input voltage and the analog output voltage:

$V_O = V_I (D/256)$

Figure xv: TLC-7528 in voltage modet[5]

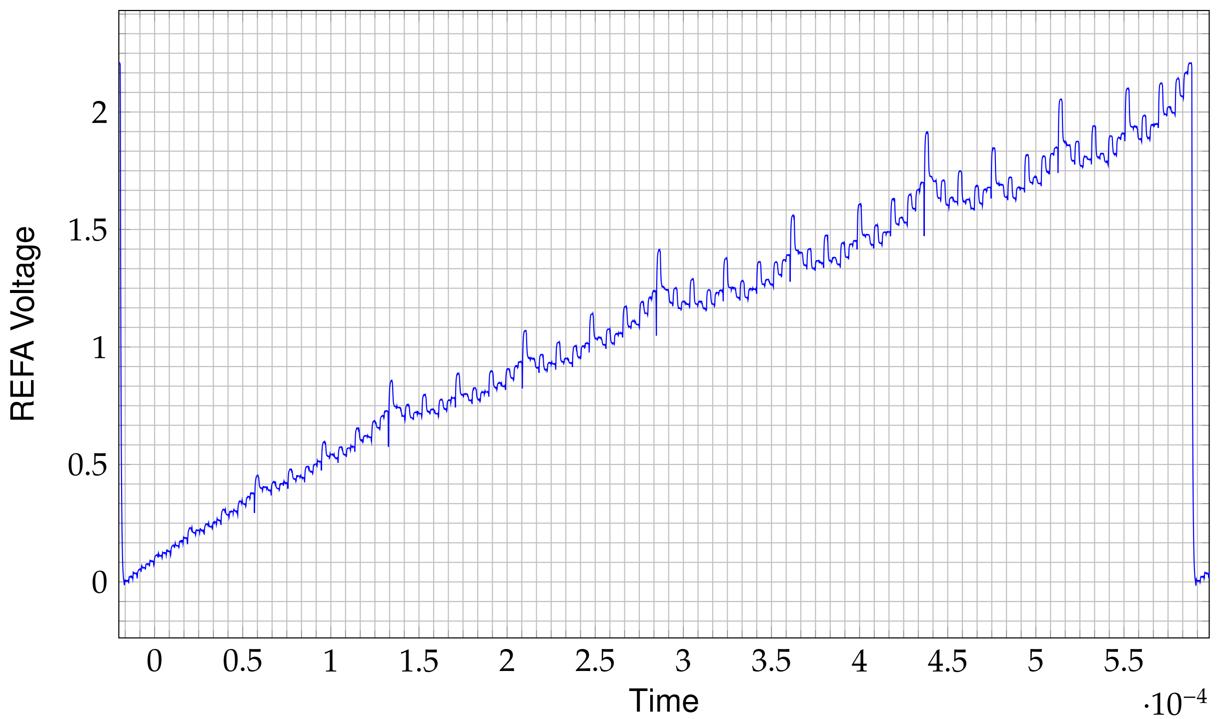

Figure xvi: Measurement of a generated SAW signal via the TLC7528

2.8.4 Schematics

Based on the descriptions in the datasheets the schematic in figure xvii was developed.

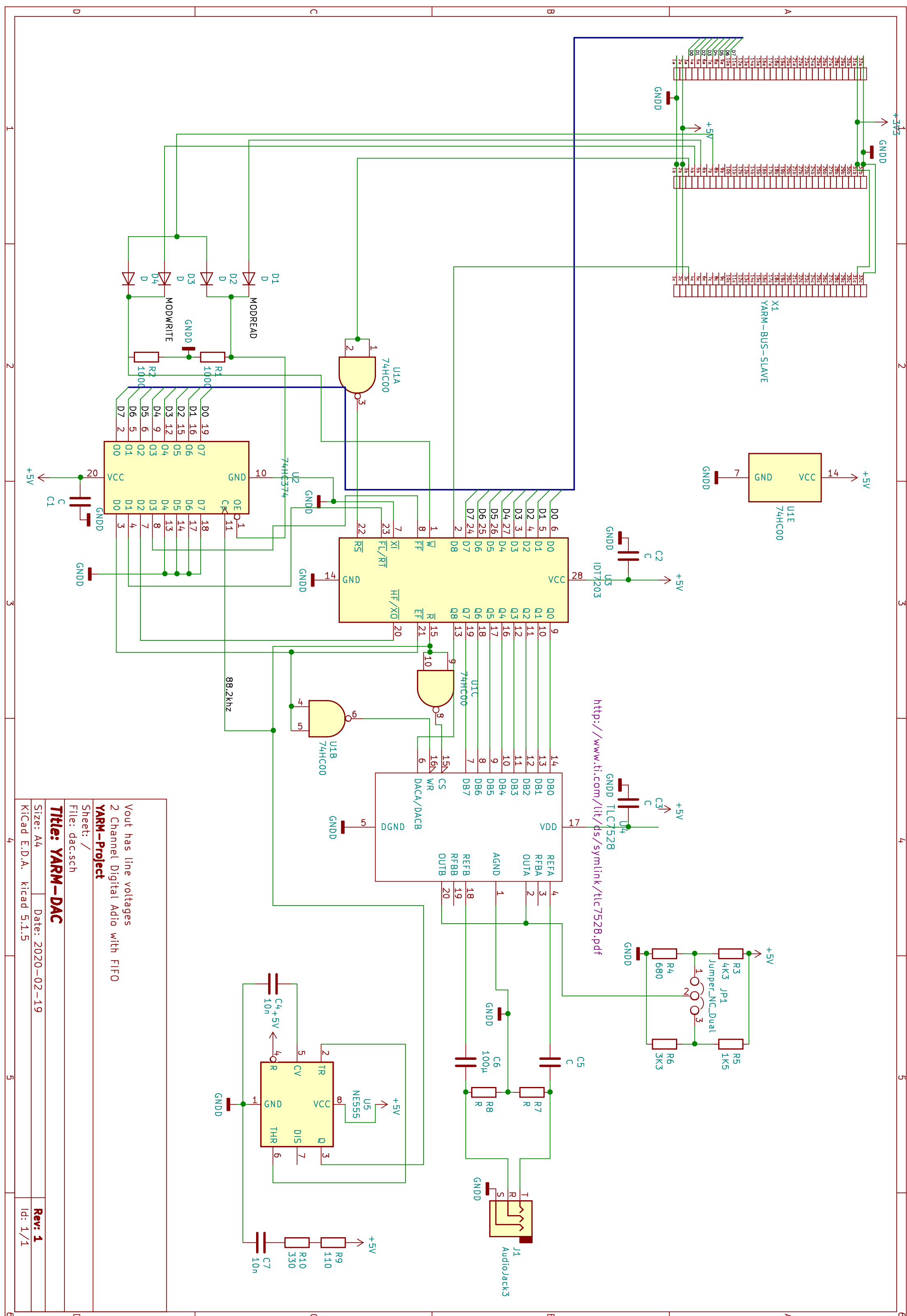

Figure xvii: The schematic of the DAC Module

Element Description Diodes D1 through D4 are used as OR-Gates in conjunction with R1 and R2 to generate the $\neg MODRD$ and $\neg MODWR$ signals for the D Flip-Flop [H] and FIFO respectively, by these formulas:

$\neg MODRD = \neg RD \vee \neg MS2$

$\neg MODWR = \neg WR \vee \neg MS2$

On a read access, the output enable of the D-Latch becomes low, which writes the status bits of the FIFO onto the data bus. C1, C2 and C3 are for stability reasons and are good practice, similar to the UART module. 74HC00 is a quad NAND-Gate[8] which is only used for inversion, chosen, like the 74HC374, for availability reasons. The A part of the NAND-Gate inverts the MR signal from the bus to a $\not M R$ signal as the FIFOs reset is low active. The B part of the NAND-Gate inverts the FIFO Empty flag. It's output is connected to the $\neg WR$ input of the DAC, which means that the DAC doesn't convert the input anymore, if the FIFO Empty flag is set to low.

The NE555 generates the audio clock signal, which should be the double of 44.1kHz[I] as 44.1kHz is the standard samling rate of CD-Audio[9]. Resistors R9 and R10 togehter with C7 form the Oscillator part of the NE55. C4 is for stability reasons and doesn't define the frequency of the oscillator.

The generated clock is used for the $\neg R$ of the FIFO and inverted on the DAC, which makes the data available on the output before being stored into the DAC as it receives the signal to store the data after the FIFO makes it available on the bus.

The DAC is operated in voltage mode as described in xv, with it's voltage source beeing available at either 3.472Vpp for professional audio or 0.894Vpp for consumer audio, as defined per convention.[10] The voltage source can be controlled via Jumper JP1.

C5 and C6 together with the load resistance on the audio jack form a high pass with a cutoff frequency of

$f_C = \frac{1}{2\pi RC} = \frac{1}{2\times\pi\times10K\Omega\times100\mu F} = 0.159154943Hz$

which should cover the hearable spectrum. The high pass was needed to generate a positive and negative half of the wave form, as the DC-Offset with a frequency of 0Hz is orders of magnitudes lower than the f_C of the highpass gets filtered away.

R7 and R8 have been installed in order to unload the capacitors after device poweroff.

NE55 Clock Source Though used as a clovk source, the NE555 is a bad clock source if a stable clock is needed, because it varies widely with temperature, preasure and aging elements. A better solution would have been a quartz which is divided down to the desired frequency, whichwas what CD Drives used to do, but more commonly in

[H] 74HC374[7]

[I] Because we have 2 output channels

modern CD Drives, an ASIC with internal PLL is used, thus the required quartz can no longer be sourced.

2.8.5 Demonstration Software

SAW Generator To prove read and write access from the D Flip-Flop and the FIFO are working, the same saw signal has been generated as in figure xvi , however the signal was put into the FIFO and not the DAC directly. The resulting saw wave can be seen in figure xviii together with the FIFO Empty flag. The FIFO Empty flag, as explained before, is inverted and starts/ends the complete D/A conversion, until further data is received.

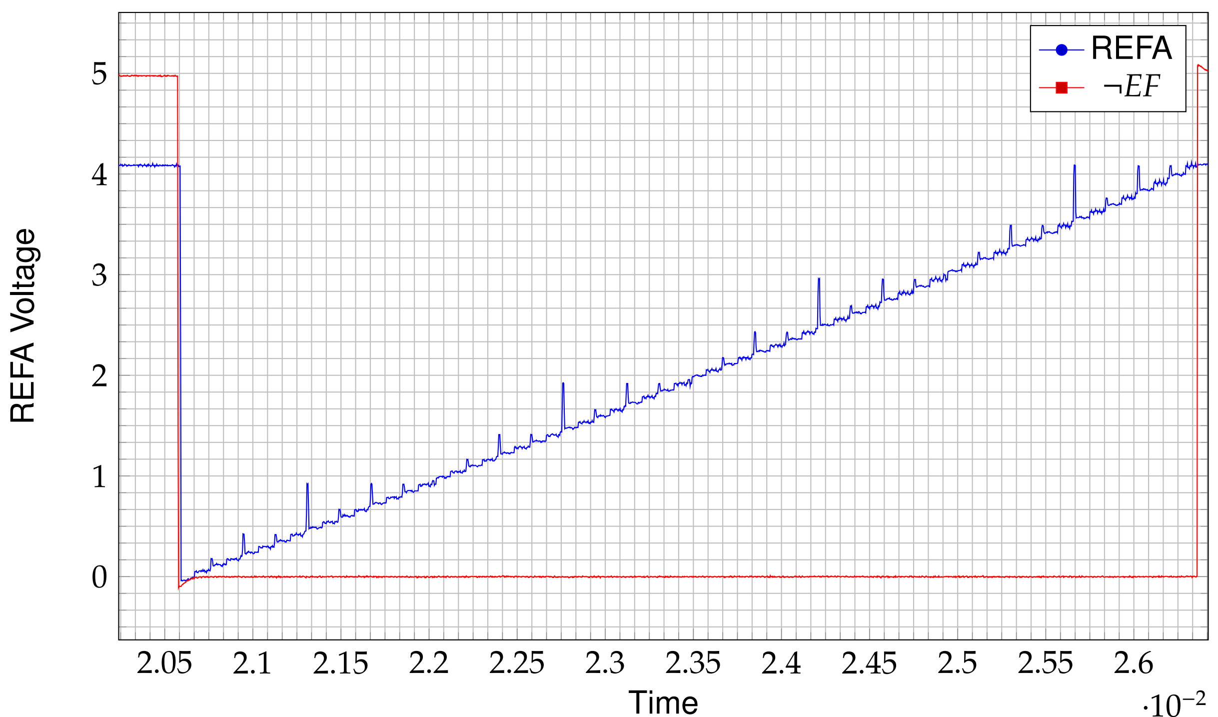

Figure xviii: Measurement of a generated SAW signal with the FIFO Empty flag

The time difference betwen a stor and complete write cycle can be seen in figure xx, while the figure xix shows the transmission between dac and fifo in more detail.

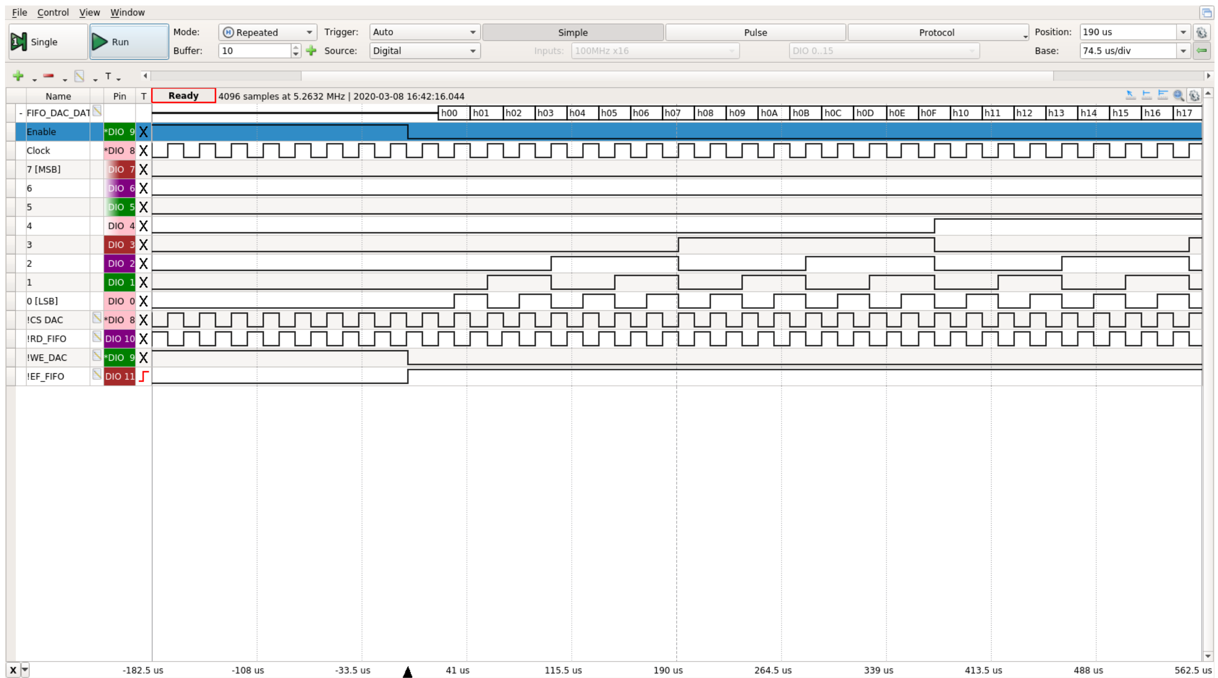

Figure xix: A transmission between the FIFO and the DAC

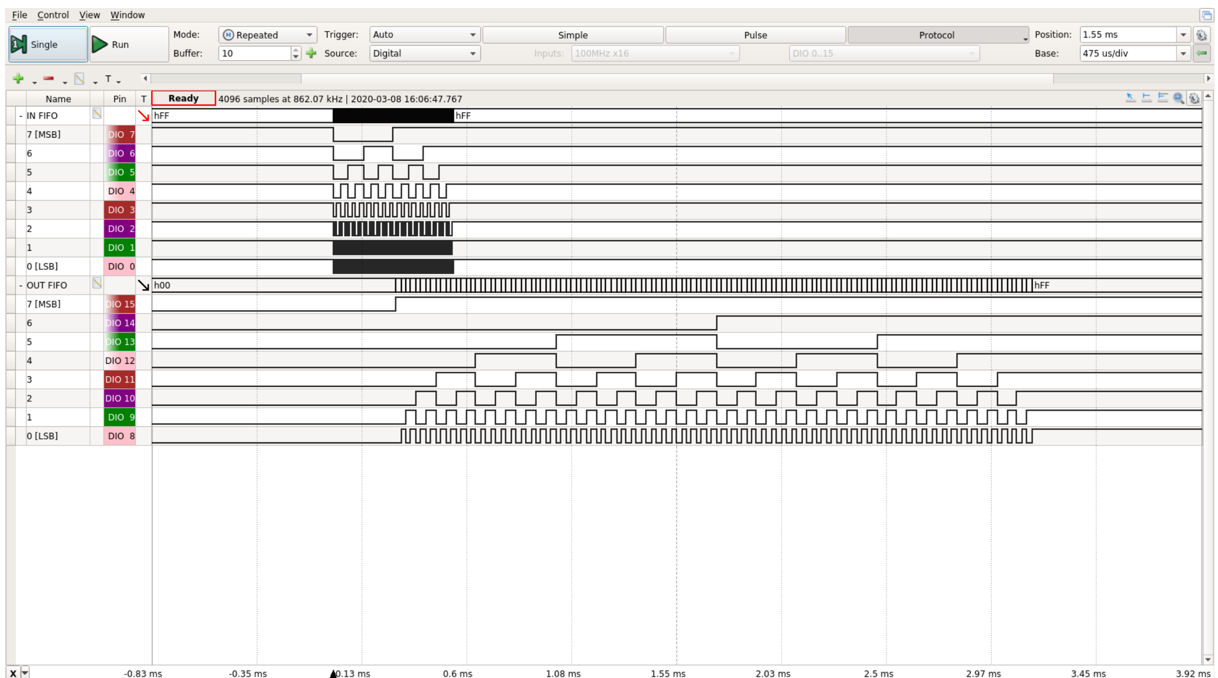

Figure xx: A fifo store operation in contrast to the load operation

The initialisation routines and read/write operations for the DAC module are basically the same as for the UART module, and have thus been ommitted. They can be seen in listing II and partially in listing II.

```
int routine(){

    for(uint8_t i = 0; i  < 0xFF; i++){
```

```
        write_to_dac(0x00, i);
    }

    write_to_dac(0x00, 0x00);

    _delay_ms(10);
        return 0;
}
```

Listing IV: SAW Generation for the DAC with FIFO

Sine Generator As a further example a sine was generated and played on the DAC. The ATMega itself is not powerful enough to generate the sine on the fly, therefore a lookup-table had to be generated, which can be seen in listing V. How the data is transmitted to the FIFO can be seen in listing VI and figure xxi and the resulting sine on both output channels can be seen in figure xxii.

```
    /* Generate sine table */
    uint8_t sine_table[256];
    for(size_t i = 0; i < 256; i++){
        sine_table[i] = 0xFF&((int)((sin(i/((double)255)*(3.141592*2))*
            127.5+127.5)));
    }
```

Listing V: Sine LUT Generation

The look-up table has a size of 256, which is the maximum value an 8 bit integer can take. This size was chosen to make operation faster as it only takes one cycle to load an array value into a register and another one to store it into the GPIO register. The sine table in further examples was pre-genrated on the compiling host to reduce startup time. The mothod shown in listing V is not fast due to the lack of a floating point unit on the AVR. [2]

```
int routine(){

    for(uint8_t i = 0; i  < 0xFF; i++){
        write_to_dac(i%2, sine_table[i]);
    }

    write_to_dac(0x00, 0x00);
    write_to_dac(0x01, 0x00);

    _delay_ms(10);
        return 0;
}
```

Listing VI: DAC Sine Generation

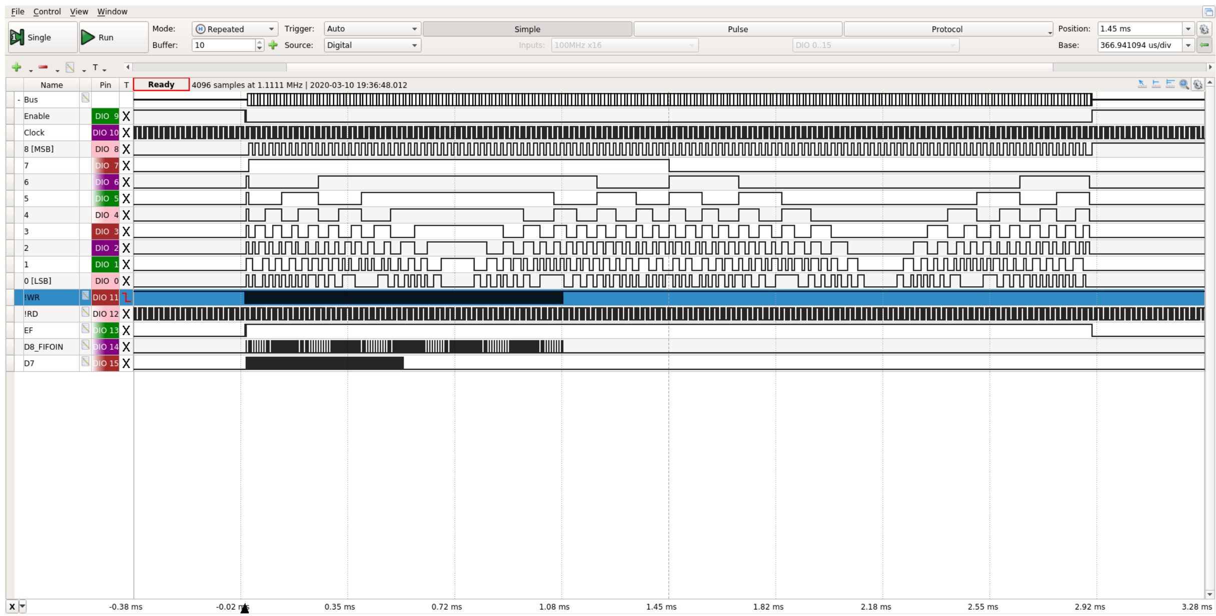

Figure xxi: Storage and retrieval of a sine to and from the FIFO

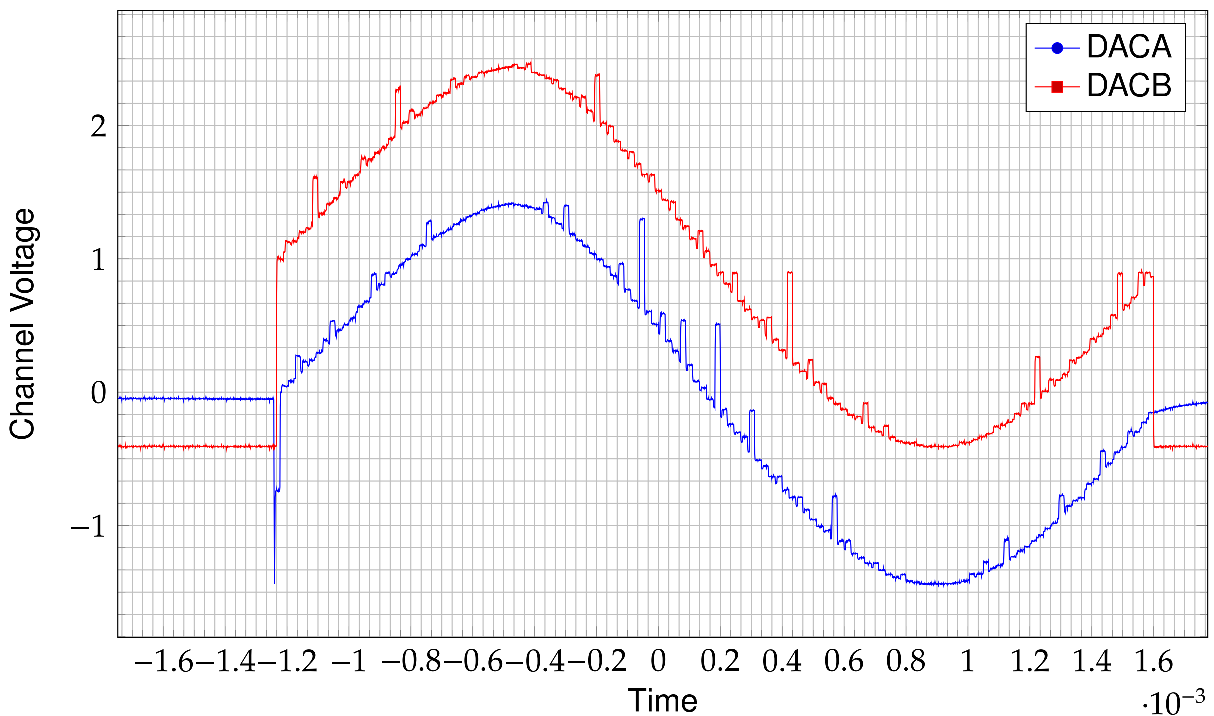

Figure xxii: Measuremet of the generated sine from the sine LUT on DACA and DACB

3 Addressing DACA and DACB

The DAC used has 2 output channels which can be selected by the $\neg DACA/DACB$ pin as seen in figure xiii. This pin was mapped to bit 0 of the address bus in order to make use of it. Bit 8 on the fifo was used to store the bit. It was not implemented with half the bus clock to make both channels independent of each other. This however uses more

time on the backend because it means the fifo is used up at twice the speed. No current example makes use of this, but it may be used in future examples and implementations on this unit.

On the audio jack DACA is mapped to the right channel and DACB to the left channel.

3.1 FPGA to Hardware interface

To make the Hardware work with the FPGA's 3.3V I/O, level shifter have been installed and a FPGA module was built. This module maps the IO/Pins in a similar way to the ATMega 2560 used in examples before. The bidirectional 5V<->3.3V logic level converters have been obtained on amazon, and have not been well documented. Their functionality has been tested and verified in both directions, which can be seen in figures xxiii and xxiv. The schematic has also been determined through measurements with a multimeter and the schematic in figure xxv shows similar resistor values in the same configuration [11].

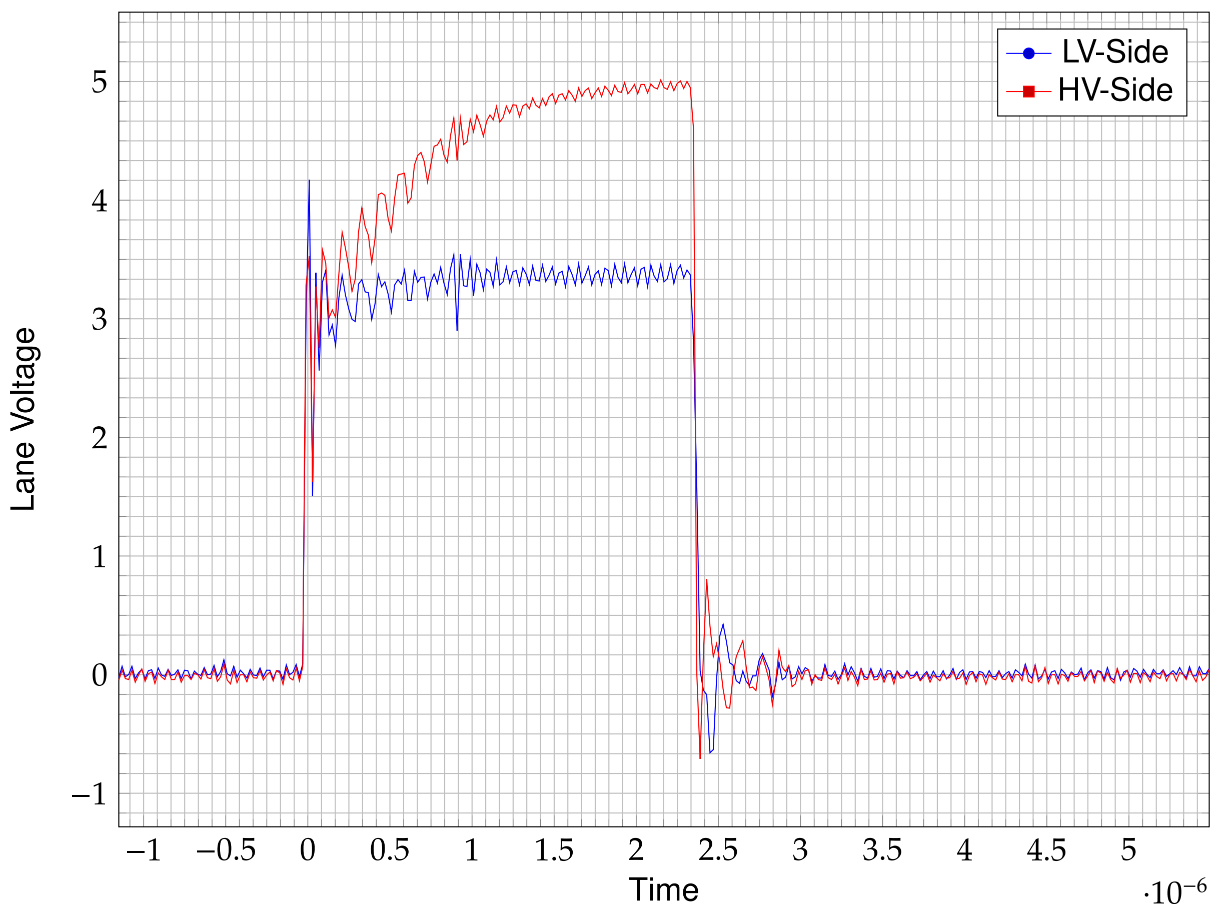

Figure xxiii: 3.3V to 5V conversion using the level shifter

The in figure xxiii shown output on the HV side, corresponds with the schematics in figure xxv where it can be seen that the resistor R2 is loading the bus capacitance to a

5V high state.

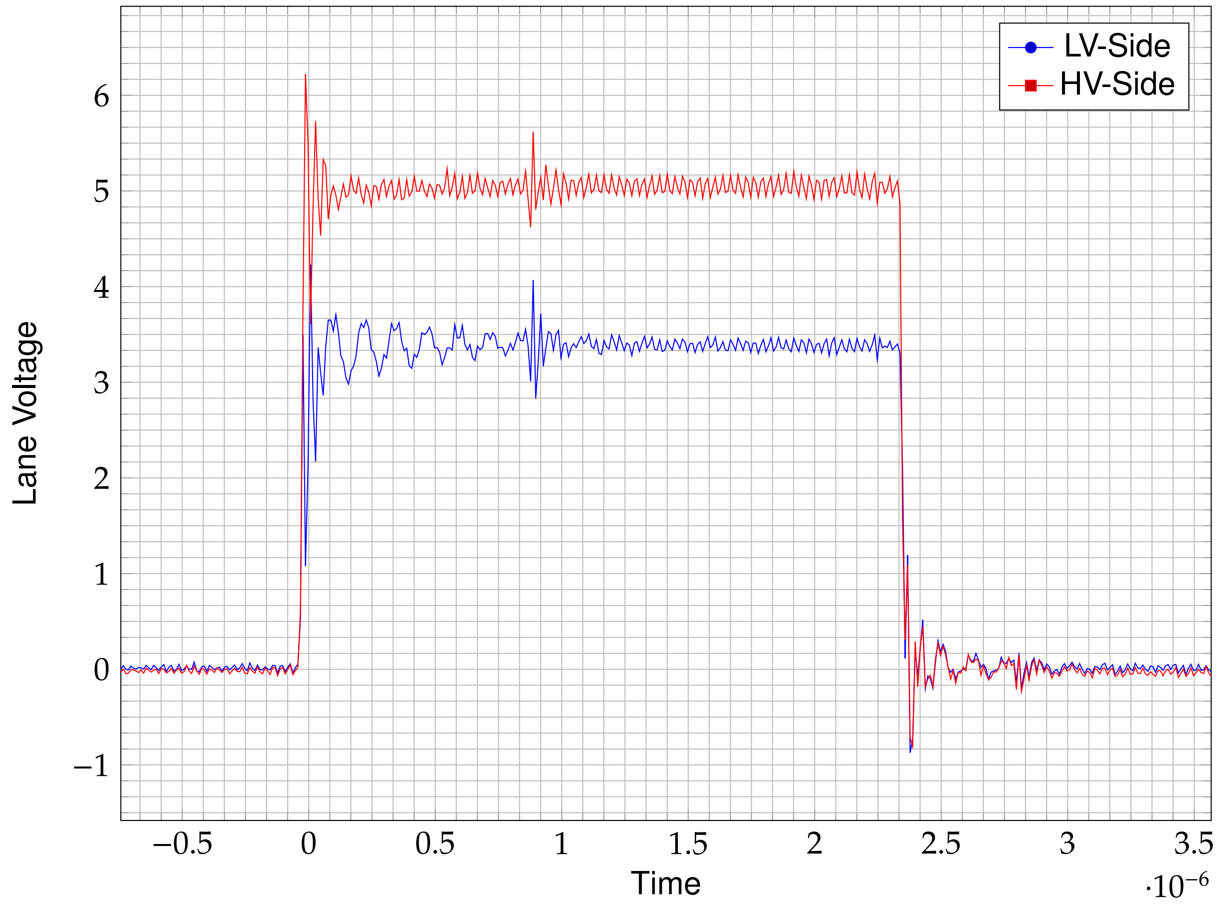

Figure xxiv: 5V to 3.3V conversion using the level shifter

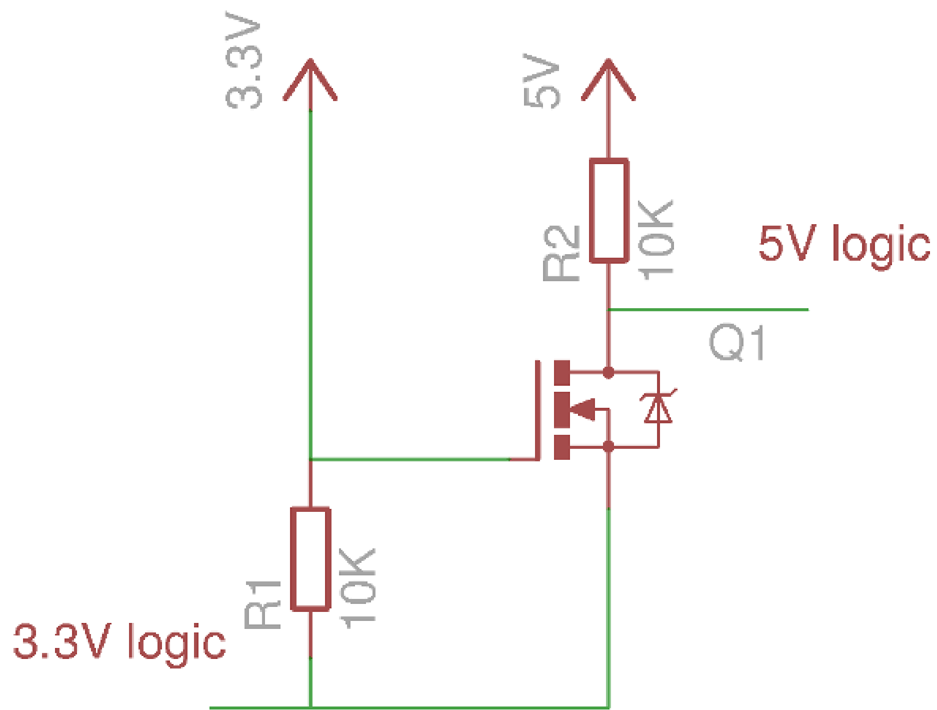

Figure xxv: The internal schematics of the level shifter[11]

3.1.1 Measurement error

During an attempt to measure wether the level shifters in the final module were working, a measurement between the LV and the HV side showed only a difference of 0.7V. After some troubleshooting, it was found that the Analog Discovery has clamping diodes against the 3.3V rail shown in figure xxvi. These diodes produce the 0.7V offset and prevent the parallel bus from rising to 5V when a digial I/O pin of the Analog Discovery 2 is connected to the bus. [12].

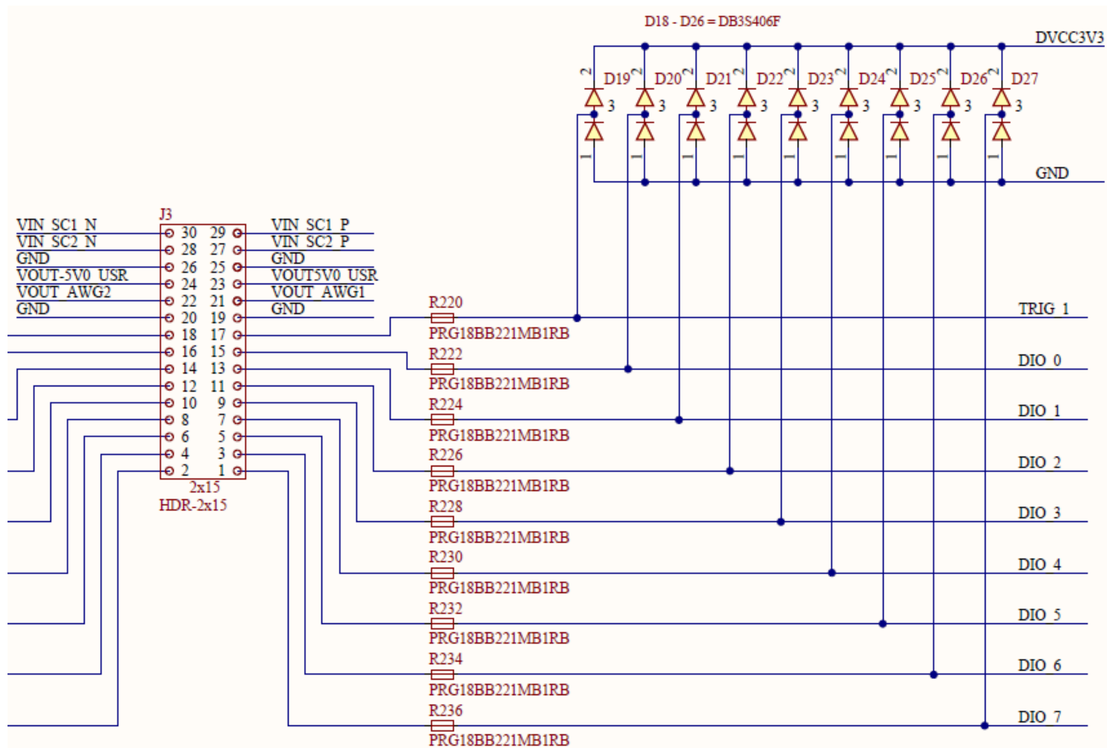

Figure xxvi: The internal clamping diodes of the Analog Discovery 2[1]

4 Textadventure

To illustrate how the components work together and can be used in various different applications, a small text-adventure with audio effects was written in C. The main goal was to show the capabilities of even small systems like the one developed.

4.1 General Implementation details

4.1.1 General definitions and pinout of the AVR

Like the before examples, the textadventure was implemented on an ATMega2560 and uses 3 different Registers for transmission: PORTF, PORTK and PORTL for address bus, data bus and control bus respectively, as can be seen in listing VII

```
1  /* Copyright (C) 2020 tyrolyean
2   *
3   * This program is free software: you can redistribute it and/or modify
4   * it under the terms of the GNU General Public License as published by
5   * the Free Software Foundation, either version 3 of the License, or
6   * (at your option) any later version.
7   *
8   * This program is distributed in the hope that it will be useful,
9   * but WITHOUT ANY WARRANTY; without even the implied warranty of
10  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
```

```
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

#ifndef _AVR_H_TEXT
#define _AVR_H_TEXT



#define F_CPU 16000000UL
#include <avr/io.h>

/* Shift values for the peripherials on the control bus PORTL */

#define MR_SHIFT      0
#define WR_SHIFT      1
#define RD_SHIFT      2
#define CS_UART_SHIFT 3
#define CS_DAC_SHIFT  4

#define ADDR_REG      PORTK
#define DATA_REG      PORTF
#define CTRL_REG      PORTL

#define ADDR_DDR_REG  DDRK
#define DATA_DDR_REG  DDRF
#define CTRL_DDR_REG  DDRL

/* Included here to prevent accidental redefinition of F_CPU */
#include <util/delay.h>

/* Time it takes for the bus lanes to become stable for read and write
   access */
#define BUS_HOLD_US 1

void set_addr(uint8_t addr);

#endif
```

Listing VII: The avr.h header file

The in listing VII shown preprocessor macros MR_SHIFT, WR_SHIFT, RD_SHIFT, CS_UART_SHIFT and CS_DAC_SHIFT are used to indicate the position of the corresponding control lines inside the control bus register. All other shift values are the same bitordering in input as in output.

The BUS_HOLD_US is used to tell the avr how many microsecons it takes for the data bus to be latched into input register of the devices on write or how long it takes for the data bus to become stable on read. A delay of less than 1 microsecond is not possible due to limitations of the AVR and the bus capacity, which increases the BER[J] to a level which effects regular operation.

4.1.2 Read and Write routines

The set_addr function is the same as in the UART example code in listing I and has therefore been omitted, execept for its definiton in the avr.h file in listing VII. The read and write functions for the UART module and the DAC module are the same as in the example code for the modules and have been ommited therefore as well.

4.1.3 UART and DAC update polling

The AVR constantly polls the DAC and UART modules for updates as can be seen in listing VIII. The routine_MODULE functions poll their respective modules for updates as can be seen in listings IX and X. When a character is received, it is stored inside a bufer array and regular operation continues. If the $\neg EF$ status bit is set in a read from the dac, the feed_dac function is called which stores 256 bytes into the DAC and regular operation continues.

```
1
2 int routine(){
3     routine_dac();
4     routine_uart();
5     routine_game();
6         return 0;
```

Listing VIII: The routine function looped by the main

```
1  void routine_uart(){
2
3      uint8_t received = read_from_uart(UART_REG_LSR);
4      if(received & 0x01){
5          received = read_from_uart(UART_REG_TXRX);
6          ingest_user_char(received);
7          if(received == '\r'){
8              writechar_16550('\n');
9          }
10         writechar_16550(received); /* Echo back */
11     }
12
13     return;
```

[J] BER...Bit Error Ratio

```
}
```

Listing IX: The routine function for the UART

```
void routine_dac(){

    uint8_t received = read_from_dac(0x00);
    if(!(received & (0x01<<0))){
        feed_dac();
    }
    return;
}
```

Listing X: The routine function for the DAC

4.2 DAC sound generation

4.2.1 DAC modes

The DAC can produce any waveform described by 8 bit unsigned PCM code. Though possible to feed predefined waveforms into the DAC, the AVR doesn't have enough onboard memory to store more than a few seconds of these waveforms.

For example to store one second of 8 bit unsigned PCM Code at 2 times 44.1KHz sampling rate of the DAC, the AVR would have to store $s = 2 \times 44100 \frac{Bytes}{s} * 1s = 2 \times 44100 Bytes = 88.2KB$, but it has only a total of 256KB of onboard flash[2] which makes for a total track lengh of $t = \frac{256KB}{88.2\frac{KB}{s}} = 2.9s$ with only one track.

Therefore the AVR generates the audio on runtime. To do that it has 6 builtin modes in which it can run, as can be seen in listing XI:

1. silent mode: The DAC produces no output at all and is completely silent.
2. sine mode: The DAC produces a sine with a specific frequency and an amplitude of 255.
3. square mode: The DAC produces a square wave with a specific frequency and an amplitude of 255.
4. saw mode: The DAC produces a saw wave with a specific frequency and an amplitude of 255.
5. noise mode: The DAC produces a pseudo-random white-noise with a maximum amplitude of 255.
6. triangle mode: The DAC produces a triangle wave with a specific frequency and an amplitude of 255.

To perform these tasks the DAC takes two parameters, again seen in listing XI:

- A frequency deviation: Used to tell the dac how much the desired frequency deviates from the base frequency of each waveform.
- A mode: Used to tell it which waveform to generate

```
/* The operation modes of the dac used for generation of different tones */
#define DAC_MODE_SILENT      0
#define DAC_MODE_SINE        1
#define DAC_MODE_SQUARE      2
#define DAC_MODE_SAW         3
#define DAC_MODE_NOISE       4
#define DAC_MODE_TRIANGLE    5

extern uint8_t dac_mode;
/* This variable is used to deviate the frequency from the baseline
   frequency
 * of around 1kHz. If this integer is positive it makes the produced
   waveform
 * longer, if it is negative the produced waveform becomes less sharp, but
   the
 * frequency goes up. 0 is the baseline */
extern int16_t dac_frequency_deviation;
```

Listing XI: The DAC operation modes

```
void feed_dac(){
    /* Internal counter for positioning inside the currently playing
     * waveform */
    static uint8_t threash = 0x00;
    /* Used to generate the desired frequency offset if the waveform should
     * be made "longer" --> the frequency made lower from baseline
     */
    static int16_t freq_delay_cnt = 0x00;
    switch(dac_mode){

        default:
        case DAC_MODE_SILENT:
            for(uint8_t i = 0; i  < 0xFF; i++){
                write_to_dac(i%2, 0);
            }

        break;

        case DAC_MODE_SINE:
        /* Generates a sine from a predetermined sine table in program
```

```
21            * space */
22               for(uint8_t i = 0; i  < (0xFF/2); i++){
23                   write_to_dac(1,
24                       pgm_read_byte(&sine_table[threash]));
25                   write_to_dac(0,
26                       pgm_read_byte(&sine_table[threash]));
27
28                   if(dac_frequency_deviation >=0){
29                       freq_delay_cnt++;
30                       if(freq_delay_cnt >=
31                           dac_frequency_deviation){
32                           freq_delay_cnt = 0;
33                           threash++;
34
35                       }
36
37                   }else{
38                       threash -= dac_frequency_deviation;
39                   }
40
41               }
42                   break;
43           case DAC_MODE_SQUARE:
44           /* Generates a square wave tone */
45               for(uint8_t i = 0; i  < (0xFF/2); i++){
46                   if(threash > (0xFF/2)){
47                       write_to_dac(0, 0xFF);
48                       write_to_dac(1, 0xFF);
49                   }else{
50                       write_to_dac(0, 0);
51                       write_to_dac(1, 0);
52                   }
53                   if(dac_frequency_deviation >=0){
54                       freq_delay_cnt++;
55                       if(freq_delay_cnt >=
56                           dac_frequency_deviation){
57                           freq_delay_cnt = 0;
58                           threash++;
59
60                       }
61
62                   }else{
63                       threash -= dac_frequency_deviation;
64                   }
65               }
66                   break;
67           case DAC_MODE_SAW:
```

```
68              /* Generates a saw wave tone */
69              for(uint8_t i = 0; i  < (0xFF/2); i++){
70                  write_to_dac(0, threash);
71                  write_to_dac(1, threash);
72                  if(dac_frequency_deviation >=0){
73                      freq_delay_cnt++;
74                      if(freq_delay_cnt >=
75                          dac_frequency_deviation){
76                          freq_delay_cnt = 0;
77                          threash++;
78 
79                      }
80 
81                  }else{
82                      threash -= dac_frequency_deviation;
83                  }
84              }
85          break;
86          case DAC_MODE_NOISE:
87          /* Generates white noise from a predetermined LUT
88           */
89              for(uint8_t i = 0; i  < (0xFF/2); i++){
90                  static uint16_t noise_cnt = 0;
91                  write_to_dac(1,
92                      pgm_read_byte(&noise_table[noise_cnt]));
93                  write_to_dac(0,
94                      pgm_read_byte(&noise_table[noise_cnt]));
95 
96                  noise_cnt++; /* Doesn't have frequency diversion
97                              */
98                  if(noise_cnt >= 1024){
99                      noise_cnt = 0;
100                 }
101 
102             }
103                 break;
104         case DAC_MODE_TRIANGLE:
105             /* Generates a triangle wave tone */
106             for(uint8_t i = 0; i  < (0xFF/2); i++){
107                 static int8_t direction = 1;
108                 if((threash == 0xFF) | !threash){
109                     direction = -direction;
110                 }
111                 write_to_dac(0, threash);
112                 write_to_dac(1, threash);
113                 if(dac_frequency_deviation >=0){
114                     freq_delay_cnt++;
```

```
                    if(freq_delay_cnt >=
                        dac_frequency_deviation){
                        freq_delay_cnt = 0;

                        threash += direction;

                    }

                }else{
                    if((dac_frequency_deviation *
                                                direction) >
                        (0xFF - threash)){
                        threash = 0xFF;
                        continue;
                    }
                    threash = (dac_frequency_deviation *
                        direction);
                }
            }
        break;
    }

    return;
}
```

Listing XII: The DAC waveform generation code

4.2.2 Tones and Tracks

A sound track inside the textadventure consists of independent tones. A tone is a waveform at a specific frequency played for a specific time. To perform the specific time functionality independant of DAC speed, an ISR [K] on the AVR was used to change to the next tone every millisecond. A track is an array of tones with an end marker tone at the end which is a tone with a length of 0ms. The end marker tone tells the ISR to reset to the initial tone. The ISR can be seen in listing XIII and the sound update function, which actually updates the current tone and is responsible for playing a track in listing XIV. The output of an example track can be seen in figures xxvii and xxviii.

```
ISR(TIMER0_COMPA_vect)
{
    update_sound();
}
```

Listing XIII: The ISR which fires every millisecond

[K]ISR...Interrupt Service Routine

```c
/* Loops a track indefinitely and changes voices according to predefined
   tables.
 * A new track resets the internal state. A voice with a length of 0ms is
   used
 * to mark the end of a track and continue at the beginning
 */
void update_sound(){

    static uint16_t audio_time = 0;
    static size_t tone_pointer = 0x00;
    static struct tone_t current_tone = {DAC_MODE_SILENT, 0,0};
    if(current_track == NULL){
        /* ABORT */
        audio_time = 0x00;
        return;
    }
    audio_time++;
    static const struct tone_t * old_track = NULL;

    if(audio_time >= current_tone.length ||
        current_track != old_track){

        if(old_track != current_track){
            tone_pointer = 0;
            audio_time = 0x00;
            old_track = current_track;
        }
        memcpy_P(&current_tone,&(current_track[tone_pointer]),
            sizeof(current_tone));

        if(current_tone.length == 0){
            tone_pointer = 0;
            memcpy_P(&current_tone,&(current_track[tone_pointer]),
                sizeof(current_tone));

        }

        dac_mode = current_tone.waveform;
        dac_frequency_deviation = current_tone.frequency_deviation +
            global_frequency_offset;
        audio_time = 0x00;
        tone_pointer++;
    }
    return;
}
```

Listing XIV: The sound update function

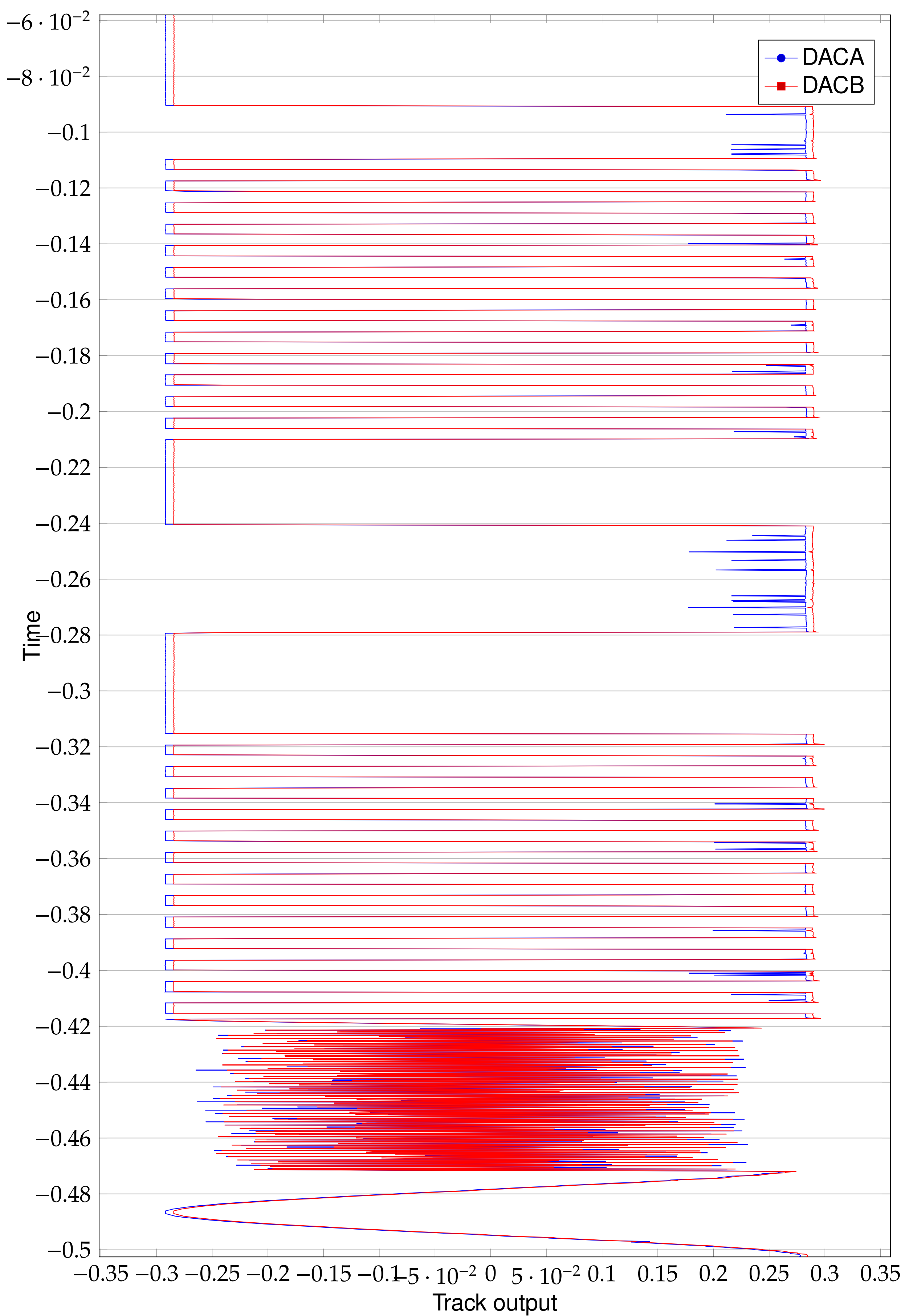

Figure xxvii: The output of an example track part 1

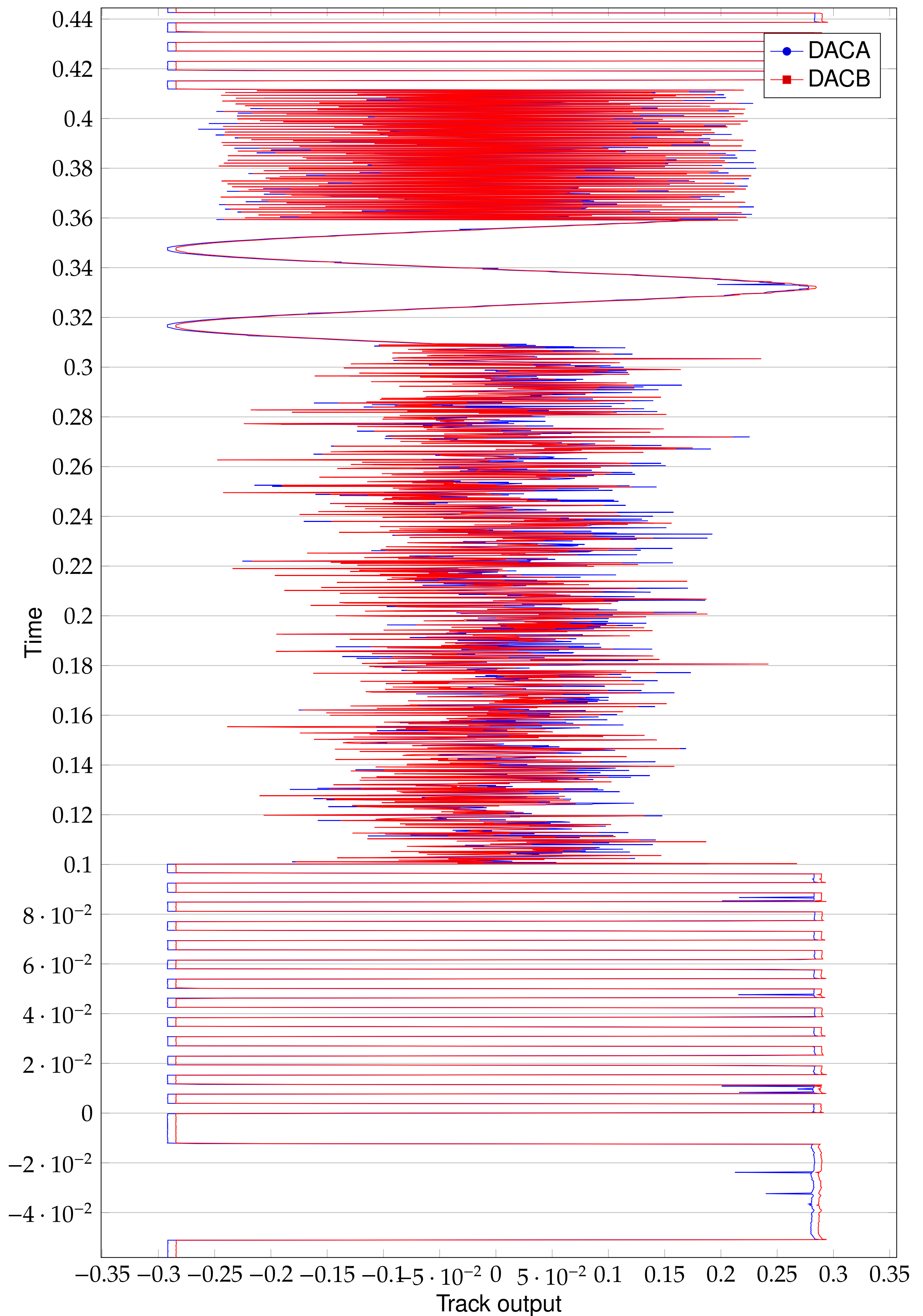

Figure xxviii: The output of an example track part 2

4.2.3 Track switching

To switch tracks on different actions, there is a map of tracks associated with rooms. Every room has an associated track, where the association can change on actions performed, which allows for a game atmosphere change. Track changes are performed outside the ISR, which could theoretically result in a race condition where the ISR would load a faulty track for 1ms if the track change was not performed fast enough, but this is prevented by disabling global interrupts during a track change.

4.3 User command interpretation

4.3.1 Command structure and parsing

As in other text adventures [13] a command consits of one line of input terminated by a newline or line feed character \n. The carriage return character which is sometimes transmitted with a line feed character is not parsed in this text adventure. Incoming character parsing can be seen in listings IX and XV.

As one command is parsed each part is required to be separated by an empty space character which is ascii code 32 [14]. The first part of the given input is then compared to an array of actions a user can perform, for example use or search, as can be seen in listing XVI

In listing IX the comment echo back can be seen. The write_char function before it writes the last received character back to the terminal which sent it. This is done to write what the user typed out to the terminal as otherwise it would not be seen what has been typed on any VT100 compatiable terminal[15] or terminal emulator.

```
void ingest_user_char(char in){
    if(in == 0x7F /* DELETE CHAR */){
        command_buffer[command_buffer_pointer--] = 0x00;

    }else{
        command_buffer[command_buffer_pointer++] = in;
    }
    return;
}
```

Listing XV: The character ingest function

The in listing XV shown branch overrides the last received character with 0x00 which is ascii NUL and decrements the buffer pointer by one if the received character was 0x7F. 0x7F is the ADCII DELETE character [14] which instructs the receiving end that the last received character was a mistake and should be purged. This is also what a vt100 compiant terminal emulator sends when the backspace or delete key is pressed [15].

```
void routine_game(){

    if(command_buffer_pointer >= sizeof(command_buffer)){

        command_buffer_pointer = 0x00;
        memset(command_buffer, 0, sizeof(command_buffer));

        println("\nToo much input!");
        return;
    }

    if(command_buffer[command_buffer_pointer-1] == '\n' ||
            command_buffer[command_buffer_pointer-1] == '\r'){
        /* A command from the user has been received, we are ready to
         * do something!*/

        int8_t action_id = -1;
        for(size_t i = 0; i < sizeof(action_table)/sizeof(const char*);
            i++){
            if(strncasecmp(action_table[i], command_buffer,
                strlen(action_table[i])) == 0){
                action_id = i;
                break;
            }

        }
        if(action_id < 0){
            println(info_table[1]);
        }else{
            perform_action(action_id);

        }

        command_buffer_pointer = 0x00;
        memset(command_buffer, 0, sizeof(command_buffer));
    }

    return;
}
```

Listing XVI: The command parsing function

4.3.2 Command parameters

Command paramters are interpreted as the string that follows the action and the space behind it. As can be seen in the case for ACTION_USE in listing XVII the use item

function is passed the command buffer[L] plus the length of the entered command plus one for the space. So the string starting at the passed address should match the start address of the parameter. If no parameter is supplied, the address should point to a character containing ASCII NUL, which marks the end of a string, bcause after comand parsing the string is overwritten with zeros as seen in listing XVI.

```
void perform_action(uint8_t action_id){
    putchar_16550('\n', NULL);
    switch(action_id){
        default:
        case ACTION_HELP:
            println("You can:");
            for(size_t i = 0; i < NUM_ACTIONS; i++){
                println("    %s",action_table[i]);
            }
            break;

        case ACTION_DESCRIBE:
            describe_room(current_room, false);
            break;

        case ACTION_NORTH:
        case ACTION_SOUTH:
        case ACTION_WEST:
        case ACTION_EAST:
            move_direction(action_id -1);
            break;
        case ACTION_INVENTORY:
            print_inventory();
            break;
        case ACTION_SEARCH:
            print_room_item();
            break;
        case ACTION_TAKE:
            consume_room_item(command_buffer+
                strlen(action_table[ACTION_TAKE])+1);
            break;
        case ACTION_USE:
            use_item(command_buffer+
                strlen(action_table[ACTION_USE])+1);
            break;

    };
    println(info_table[3]);

```

[L]which is an address in memory

```
    return;
}
```

Listing XVII: The command execution routine

4.4 Gameplay

The game itself plays like a regular game with limtations set in direction. Playeras can search for items in each room and grab the found items as can be seen in figure xxix. The general gamplay is perfomred via altering the map data and the strings output to the user.

```
INIT
LONELY ROAD

You are on the dead end of a lonely road. You look right and left ofyou, but
you cannot remember why you are here... You are terrified.
▒INIT
LONELY ROAD

You are on the dead end of a lonely road. You look right and left ofyou, but
you cannot remember why you are here... You are terrified.
help

You can:
    help
    north
    south
    west
    east
    describe
    use
    inventory
    search
    take
What are you going to do?
dearch
Invalid command!
search

You found a PISTOL
What are you going to do?
take pistol

You took the PISTOL
What are you going to do?
north

Moving towards north
S/N DIRT ROAD

You travel a bit towards the moon, you think that's the way to go. You find a
bear in the middle of the road sleeping seemingly in peace.
What are you going to do?
use pistol

You can't use that!
What are you going to do?
use sausage

You can't use that!
What are you going to do?
search sauasage

You found a SAUSAGE
What are you going to do?
take sausage

You took the SAUSAGE
What are you going to do?
use sausage

it ran away...
What are you going to do?
```

Figure xxix: A regular beginning of the game

4.5 Memory constraints

The AVR has 8kB of internal SRAM which are used for stack and heap [2]. During the build of the program an ELF file can be obtained which contains infromation on the

programs structure and memory usage on boot. Strings and variables are contained within the .data section of the elf file, but loaded into the .bss section during boot[16]. This is done for integer variables, as well as for strings, which makes the use of strings limited not to the flash size but to the RAM size of the AVR. To save memory, sound tracks as well as the sine and noise table have been put into program space with the PROGMEM attribute as described by the avr-libc documentation[17]. In listing XII a read from program memory can be seen in the noise and sine modes.

5 Erklärung der Eigenständigkeit der Arbeit

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe. Meine Arbeit darf öffentlich zugänglich gemacht werden, wenn kein Sperrvermerk vorliegt.

Ort, Datum	Armin Brauns
Ort, Datum	Daniel Plank

I List of Figures

i	Atari PBI Pinout;Source: https://www.atarimagazines.com . . .	2
ii	Digilent Analog Discovery 2;Source: https://www.sparkfun.com/	4
iii	Layout of the DIN41612 Connectors on the Backplane	5
iv	Measurement at around 1MHz bus clock on MS1	6
v	The case with installed backplane	7
vi	PC-16550D Pinout[3]	8
vii	The schematic of the UART Module	10
viii	Measurement of the 1.8432 MHz Output on J1	11
ix	Measurement of a character transmission before and after MAX-232 . .	12
x	Pinout of the RJ-45 Plug; Src: https://www.wti.com/	12
xi	Measurement of a character echo	13
xii	Transmission of character A via the 16550 UART	16
xiii	TLC-7528 Pinout[5]	18
xiv	IDT-7201 Pinout[6]	19
xv	TLC-7528 in voltage modet[5]	20
xvi	Measurement of a generated SAW signal via the TLC7528	20
xvii	The schematic of the DAC Module	21
xviii	Measurement of a generated SAW signal with the FIFO Empty flag . . .	23
xix	A transmission between the FIFO and the DAC	24
xx	A fifo store operation in contrast to the load operation	24
xxi	Storage and retrieval of a sine to and from the FIFO	26
xxii	Measuremet of the generated sine from the sine LUT on DACA and DACB	26
xxiii	3.3V to 5V conversion using the level shifter	27
xxiv	5V to 3.3V conversion using the level shifter	28
xxv	The internal schematics of the level shifter[11]	29
xxvi	The internal clamping diodes of the Analog Discovery 2[1]	30
xxvii	The output of an example track part 1	40
xxviii	The output of an example track part 2	41
xxix	A regular beginning of the game	46

II List of Tables

III Listings

I	Read and write routines for the 16550 UART	13

II	16550 INIT routines and single char transmission	15
III	16550 character echo	16
IV	SAW Generation for the DAC with FIFO	24
V	Sine LUT Generation	25
VI	DAC Sine Generation	25
VII	The avr.h header file	30
VIII	The routine function looped by the main	32
IX	The routine function for the UART	32
X	The routine function for the DAC	33
XI	The DAC operation modes	34
XII	The DAC waveform generation code	34
XIII	The ISR which fires every millisecond	37
XIV	The sound update function	38
XV	The character ingest function	42
XVI	The command parsing function	43
XVII	The command execution routine	44

LITERATURVERZEICHNIS

[1] Analog Discovery 2 Reference Manual. Digilent, Inc. Sept. 2015. URL: https://reference.digilentinc.com/_media/reference/instrumentation/analog-discovery-2/ad2_rm.pdf.

[2] Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V. Atmel Corporation. Feb. 2014. URL: https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf.

[3] PC16550D Universal Asynchronous Receiver/Transmitter With FIFOs. Texas Instruments Inc. 1995. URL: https://www.scs.stanford.edu/10wi-cs140/pintos/specs/pc16550d.pdf.

[4] MAX232x Dual EIA-232 Drivers/Receivers. Texas Instruments Inc. Feb. 1989. URL: https://www.ti.com/lit/ds/symlink/max232.pdf.

[5] DUAL 8-BIT MUTLIPLYING DIGITAL-TO-ANALOG CONVERTERS. Texas Instruments Inc. 1987. URL: https://www.ti.com/lit/ds/symlink/tlc7528.pdf.

[6] Integrated Device Technology, Inc.: CMOS ASYNCHRONOUS FIFO. RENESAS. 2002. URL: http://www.komponenten.es.aau.dk/fileadmin/komponenten/Data_Sheet/Memory/IDT7201.pdf.

[7] High-Speed CMOS Logic Octal D-Type Flip-Flop, 3-State Positive-Edge Triggered. Texas Instruments Inc. Feb. 1998. URL: https://www.ti.com/lit/ds/schs183c/schs183c.pdf.

[8] SNx4HC00 Quadruple 2-Input Positive-NAND Gates. Texas Instruments Inc. Dec. 1982. URL: https://www.ti.com/lit/ds/symlink/sn74hc00.pdf.

[9] Compact disc digital audio system. Standard. International Electrotechnical Commission, Sept. 1987.

[10] Ethan Winer: The Audio Expert: Everything You Need to Know About Audio. Focal Press, 2013. URL: https://books.google.com/books?id=TIfOAwAAQBAJ&pg=PA107#v=onepage&q=-%2010%20dbv&f=false.

[11] Jenny List: „Taking It To Another Level: Making 3.3V Speak With 5V“. In: (Dec. 2016). URL: https://hackaday.com/2016/12/05/taking-it-to-another-level-making-3-3v-and-5v-logic-communicate-with-level-shifters/.

[12] Schottky Barrier Diode DB3S406F0L Silicon epitaxial planar type. Panasonic. Mar. 2010. URL: https://industrial.panasonic.com/content/data/SC/ds/ds4/DB3S406F0L_E.pdf.

[13] Ron Schnell: Dunnet Source Code. Emacs. 1982. URL: https://github.com/jwiegley/emacs-release/blob/master/lisp/play/dunnet.el.

[14] ASCII Format for Network Interchange. Standard. Network Working Group, Oct. 1969. URL: https://tools.ietf.org/pdf/rfc20.pdf.

[15] VT100 SERIES TECHNICAL MANUAL. Digital Equipment Corporation. 1979. URL: https://vt100.net/docs/vt100-tm/ek-vt100-tm-002.pdf.

[16] Tool Interface Standard (TIS)Executable and Linking Format (ELF) Specification. Standard. TIS Committee, May 1995. URL: https://refspecs.linuxbase.org/elf/elf.pdf.

[17] Unknown Author: Data in Program Space. avr-libc 2.0.0 Standard C library for AVR-GCC. 2016. URL: https://www.nongnu.org/avr-libc/user-manual/pgmspace.html.

Anhang