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 20. 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.

Projektergebnis

Contents

Gendererklärung i
Kurzfassung/Abstract ii
Projektergebnis 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
3 Textadventure 17
3.1 General Implementation details 18
4 Erklärung der Eigenständigkeit der Arbeit 20
I List of Figures I
II List of Tables I
III Listings I
Anhang II

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[1] 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.

[1] 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 1 the layout of the Atari Parallel Bus Interface is shown as used on the Atari 800XL.

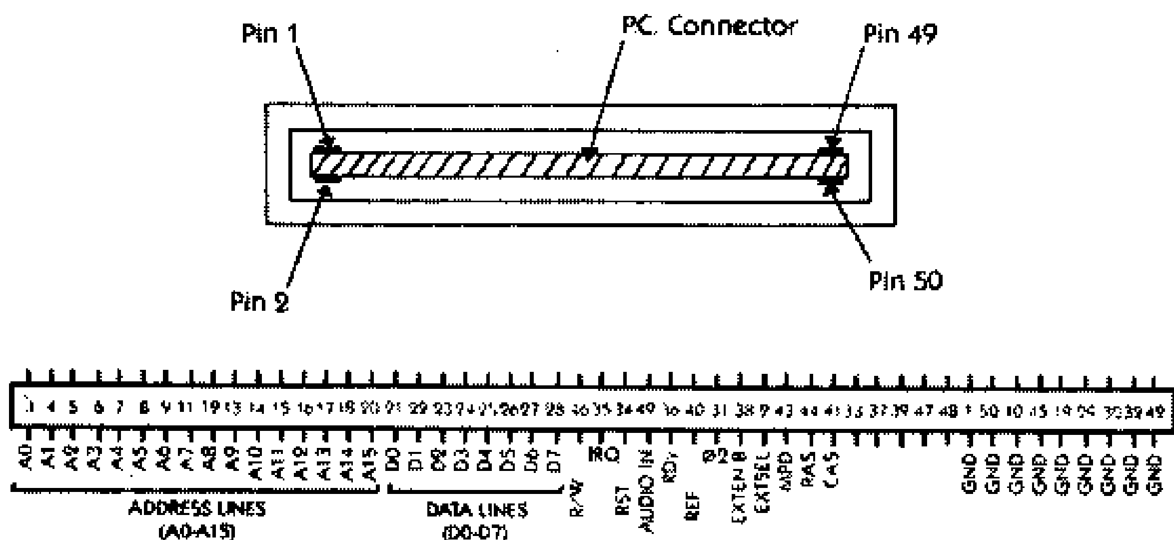

Figure 1: Atari PBI Pinout;Source: `https://www.atarimagazines.com`

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. These were for all nescessary measurements enough. Though due to the size and construction of the device, which can be seen in figure 2 errors wer encountered while performing the measurements. These are noted on occurance.

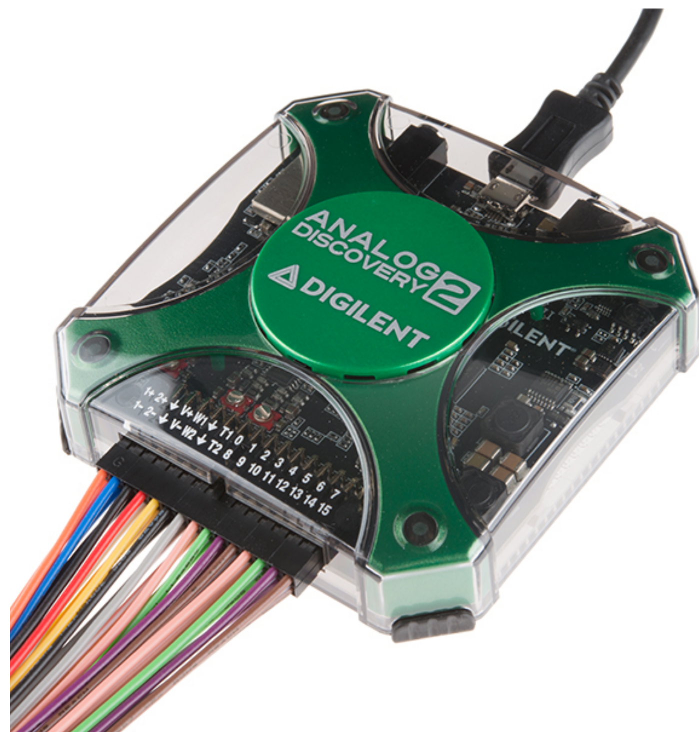

Figure 2: Digilent Analog Discovery 2;Source: `https://www.sparkfun.com/`

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. 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 3.

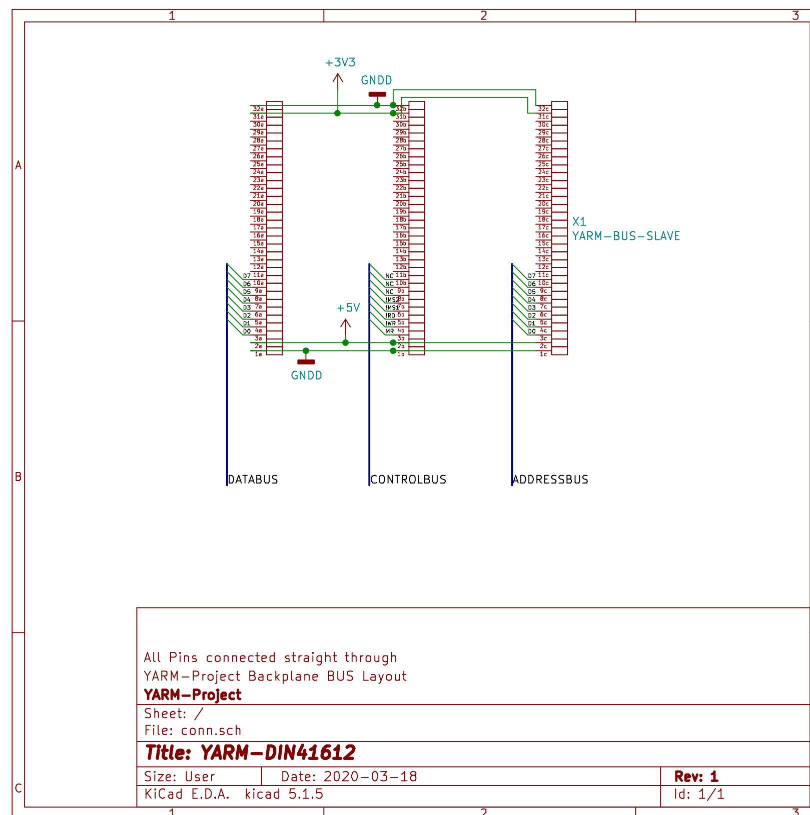

Figure 3: Layout of the DIN41612 Connectors on the Backplane

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 4

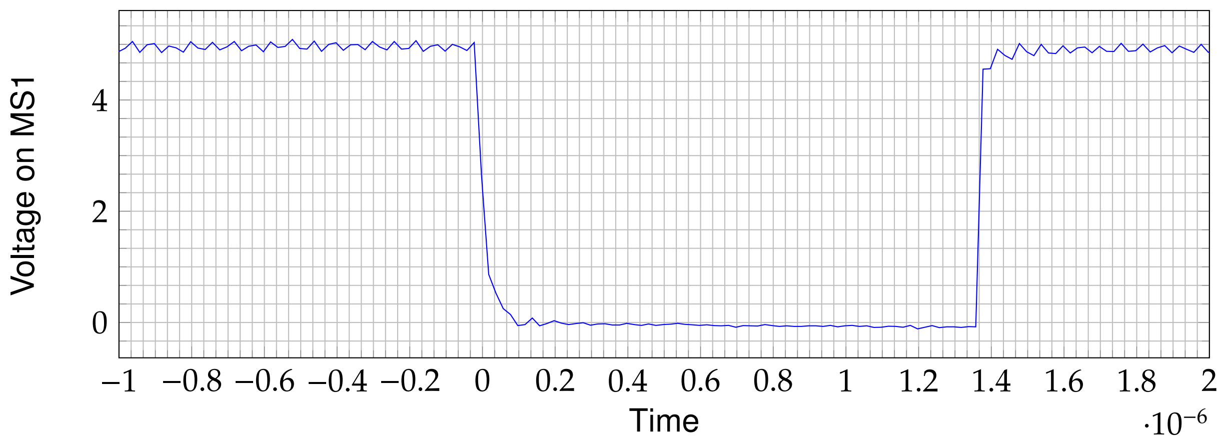

Figure 4: Measurement at around 1MHz bus clock on MS1

The ripple seen in figure 4 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 5: 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[2] or as it should be called now TIA-[3]/EIA-[4]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[5] 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 [6] buffer. It needs three address lines, and 8 data lines, which can be seen in figure 6

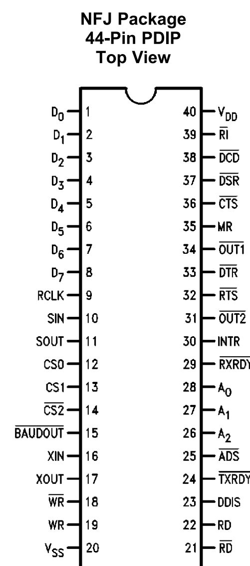

Figure 6: PC-16550D Pinout[1]

In figure 6 the most important lanes are the SIN and sout lanes, as they contain the serial data to and from the 16550 UART.

[2] RS... Recommended Standard
[3] TIA...Telecommunications Industry Association
[4] EIA.. Electronic Industries Alliance
[5] Uinversal Asynchronous Receiver and Transmitter
[6] 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[2].

2.7.3 Schematics

Based on the descriptions in the datasheets the schematic in figure 7 was developed.

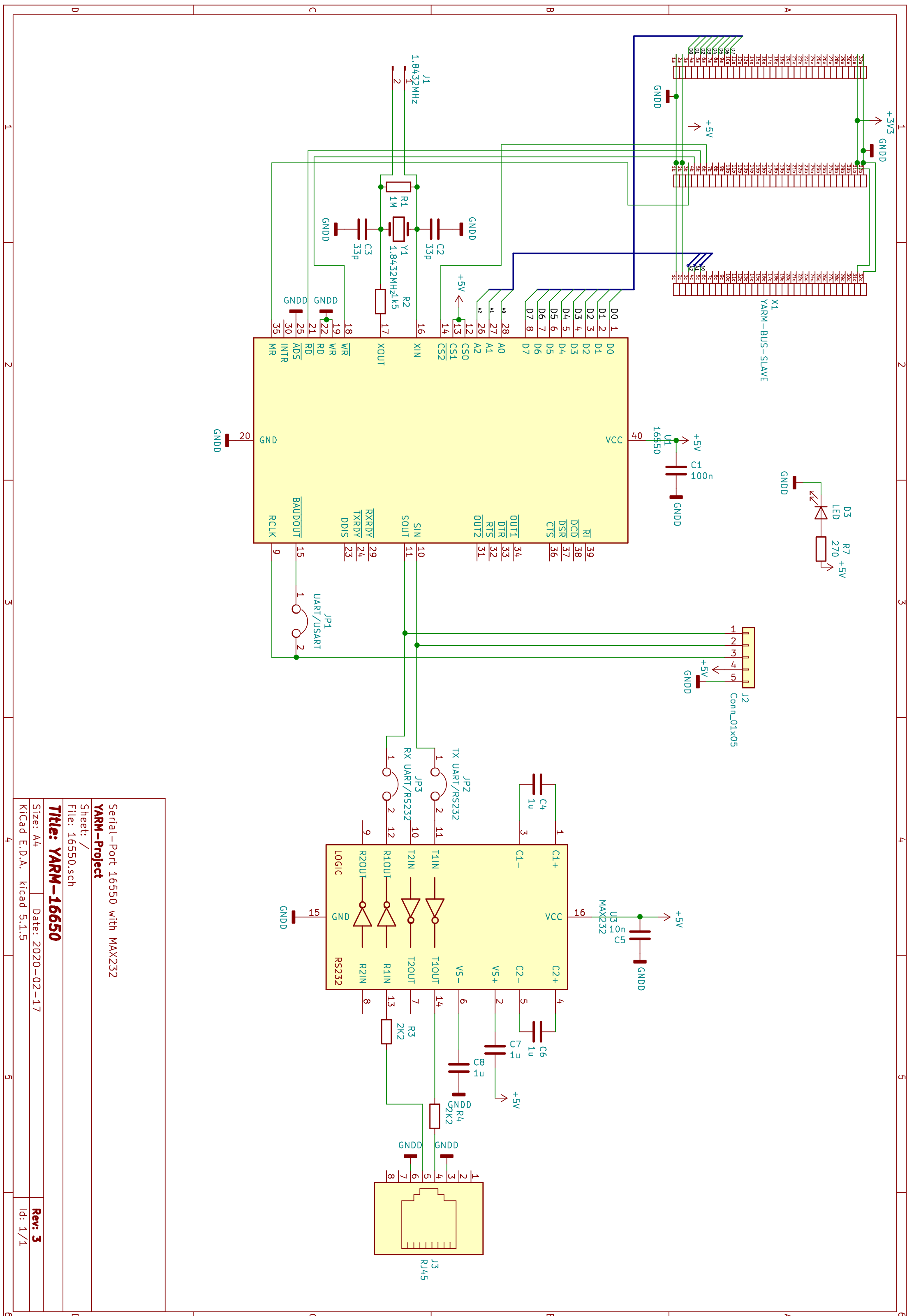

Figure 7: The schematic of the UART Module

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 [1]. 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 8. 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[1]. The final output of the 16550 UART can be used and measured via J2, as shown in figure 9 . 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[2]. 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 10. C5 has the same functionality for the MAX-232 as the C1 has to the 16550-UART.

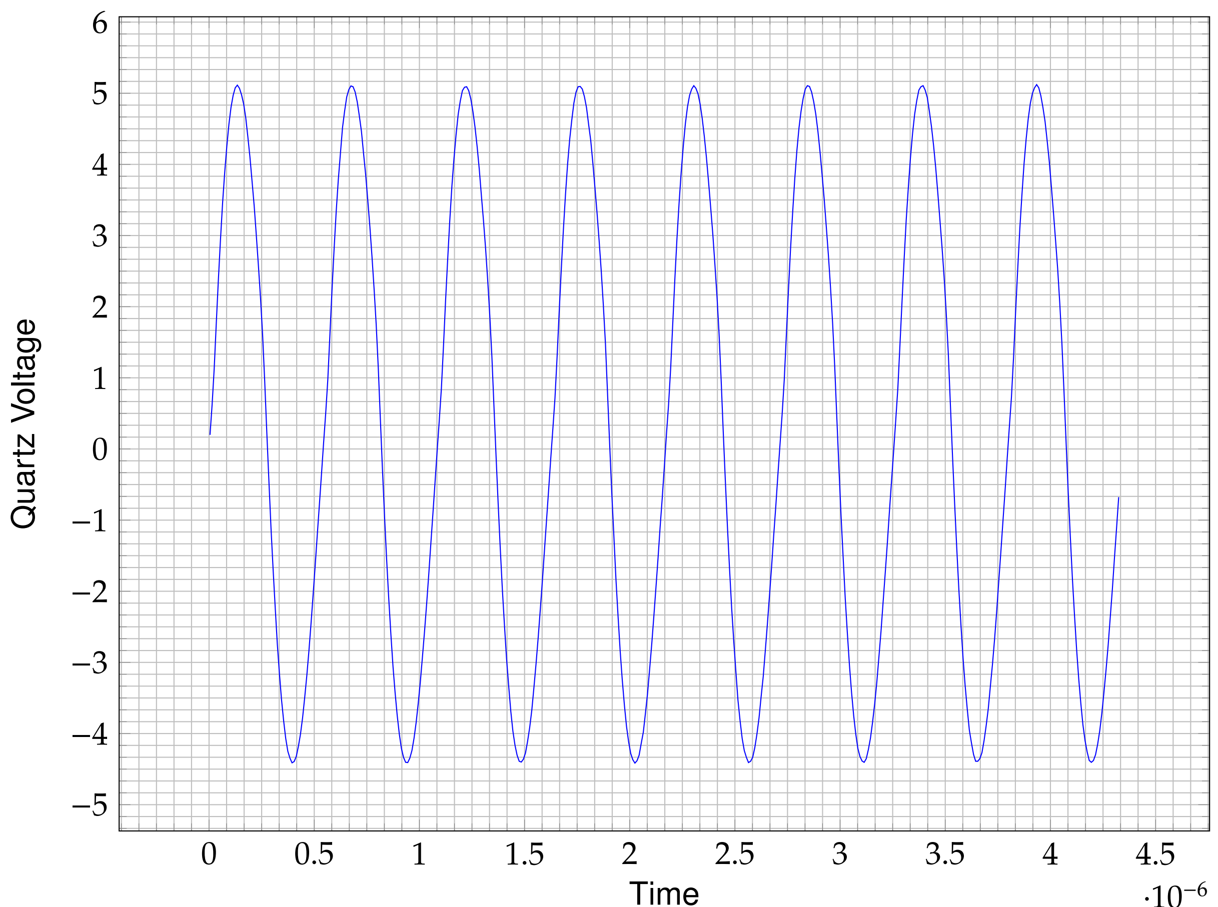

Figure 8: Measurement of the 1.8432 MHz Output on J1

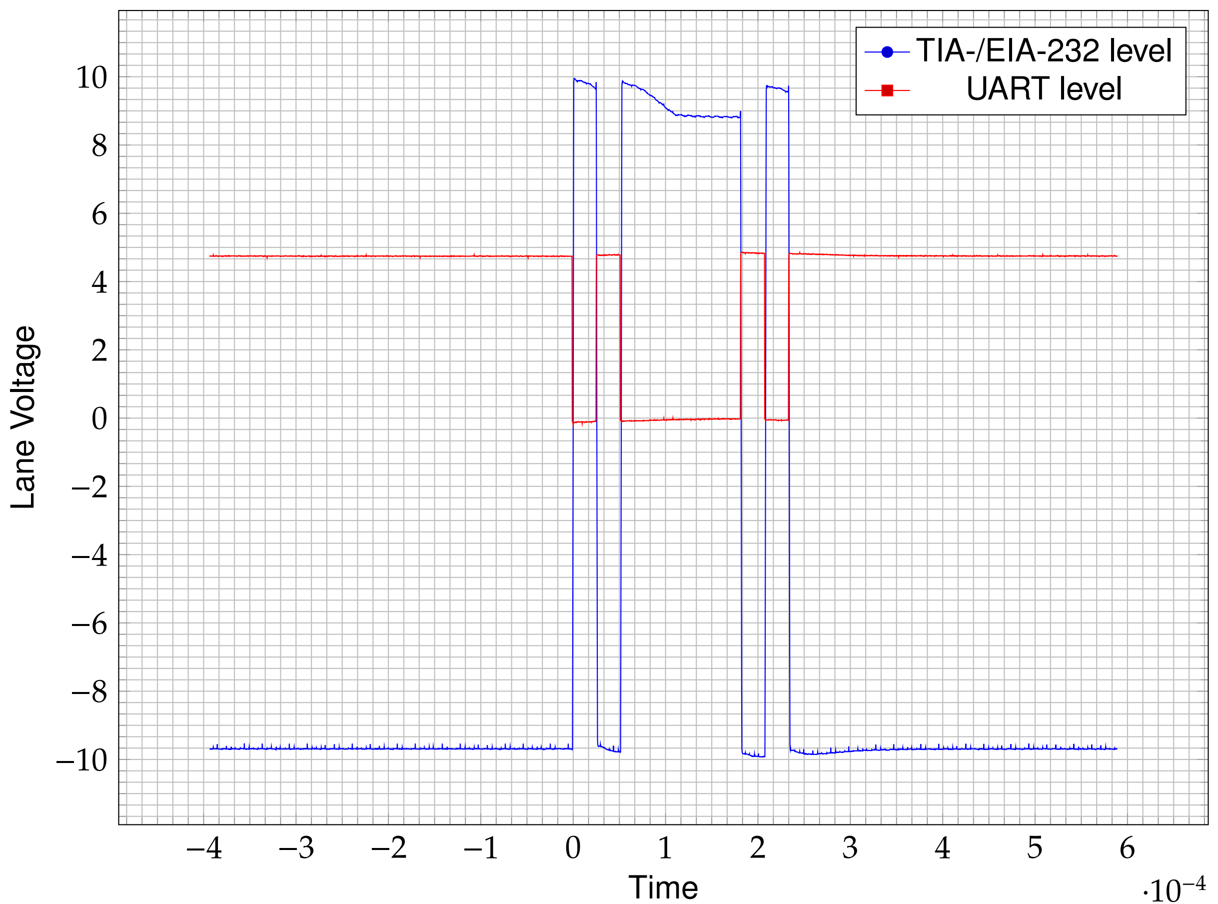

Figure 9: Measurement of a character transmission before and after MAX-232

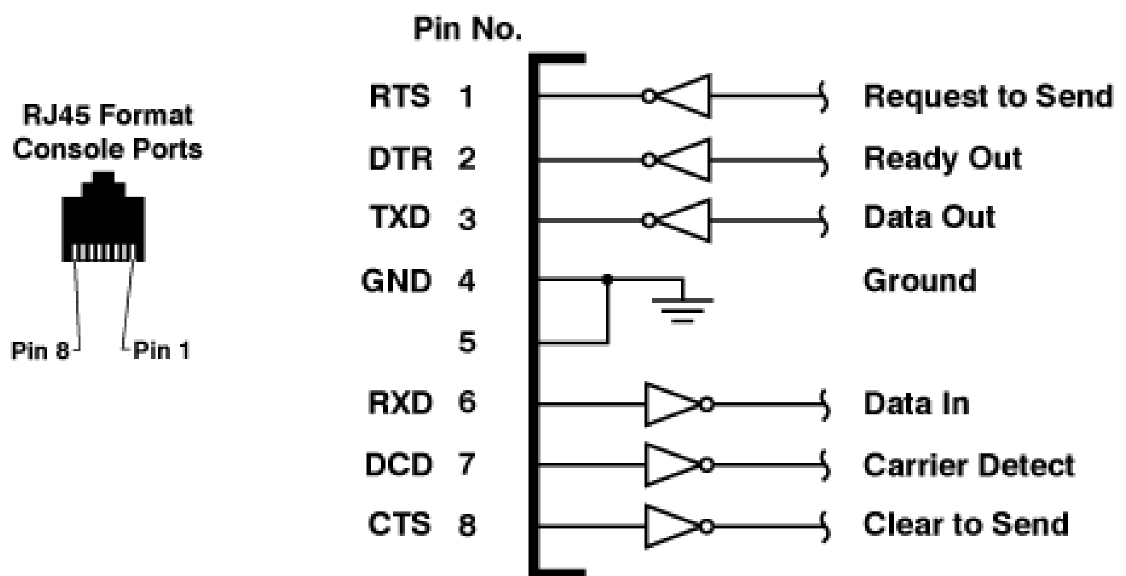

Figure 10: Pinout of the RJ-45 Plug; Src: `https://www.wti.com/`

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 9 and the output for program two in figure 11.

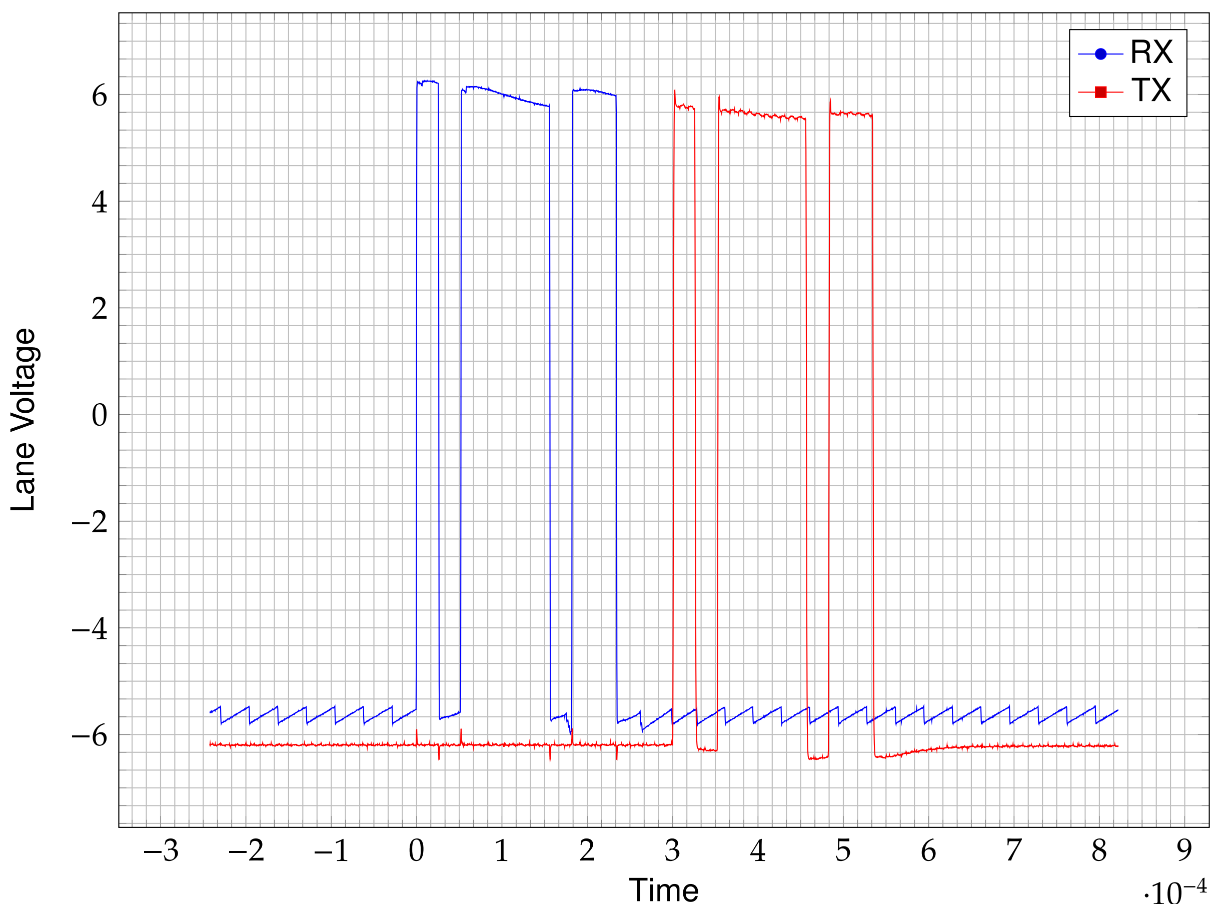

Figure 11: Measurement of a character echo

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 1 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);
```

```
    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 1: 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[1]. 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 2

```
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);
```

```
    _delay_us(1000);

    sei();

    for(;;){
        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);
        write_to_16550(UART_REG_TXRX,'A');
        _delay_us(10000);
        }

        return 0;
}
```

Listing 2: 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 12

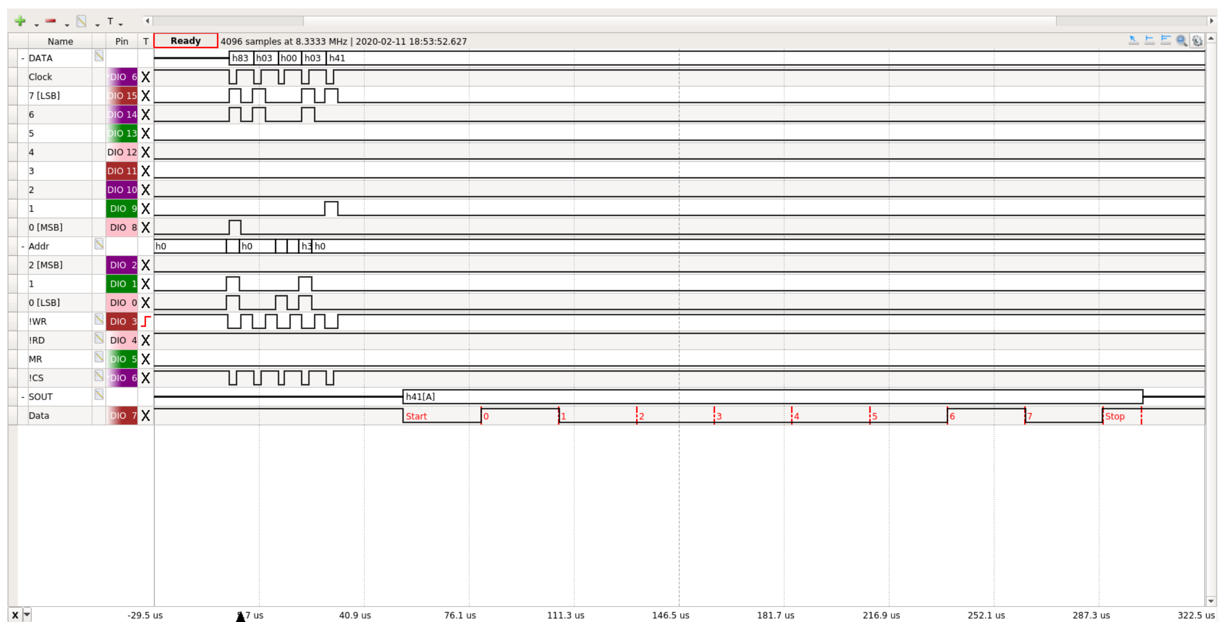

Figure 12: Transmission of character A via the 16550 UART

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 11. The initialisation is practically the same as for the transmission code, as well as the read and write routines in listing 1.

```
int main(){
```

```
2
3     /* Disable interrupts during initialisation phase */
4     cli();
5
6     /* Setup Data Direction Registers and populate with sane default
7        values */
8     DDRF = 0xFF; /* Data Bus */
9     DDRK = 0xFF; /* Address Bus */
10    DDRL = 0xFF; /* Control Bus */
11
12    /* Cleanly reset the 16550 uart */
13    PORTL |= (1<<WR_SHIFT);
14    PORTL |= (1<<RD_SHIFT);
15    PORTL |= (1<<CS_SHIFT);
16    PORTL |= (1<<CS_ADC_SHIFT);
17    PORTL |= (1<<MR_SHIFT);
18    _delay_us(100);
19    PORTL &= ~(1<<MR_SHIFT);
20    _delay_us(1000);
21
22    write_to_16550(UART_REG_LCR,0x83);
23    write_to_16550(UART_REG_DLLS,0x03);
24    write_to_16550(UART_REG_DLMS,0x00);
25    write_to_16550(UART_REG_LCR,0x03);
26    for(;;){
27        if(read_from_16550(UART_REG_LSR) & 0x01){
28            write_to_16550(UART_REG_TXRX,
29                read_from_16550(UART_REG_TXRX));
30        }
31    }
32
33        return 0;
34 }
```

Listing 3: 16550 character echo

3 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.

3.1 General Implementation details

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 4

```
/* Copyright (C) 2020 tyrolyean
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * 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 */
```

```
42 #include <util/delay.h>
43
44 /* Time it takes for the bus lanes to become stable for read and write
      access */
45 #define BUS_HOLD_US 1
46
47 void set_addr(uint8_t addr);
48
49 #endif
```

Listing 4: The avr.h header file

The in listing 4 shown defines 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.

4 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

1	Atari PBI Pinout;Source: https://www.atarimagazines.com	2
2	Digilent Analog Discovery 2;Source: https://www.sparkfun.com/	4
3	Layout of the DIN41612 Connectors on the Backplane	5
4	Measurement at around 1MHz bus clock on MS1	6
5	The case with installed backplane	7
6	PC-16550D Pinout[1]	8
7	The schematic of the UART Module	10
8	Measurement of the 1.8432 MHz Output on J1	11
9	Measurement of a character transmission before and after MAX-232	12
10	Pinout of the RJ-45 Plug; Src: https://www.wti.com/	12
11	Measurement of a character echo	13
12	Transmission of character A via the 16550 UART	16

II List of Tables

III Listings

1	Read and write routines for the 16550 UART	13
2	16550 INIT routines and single char transmission	15
3	16550 character echo	16
4	The avr.h header file	18

Literaturverzeichnis

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

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

ANHANG