University of Innsbruck

Institute of Computer Science

Intelligent and Interactive Systems

Improved On-Board Communication for Low-Cost Mobile Robots

Alexander Hirsch

B.Sc. Thesis

Supervisor: Justus Piater
Simon Haller

20th June 2014

Abstract

This project seeks to improve the communication process between the host computer of an existing, low-cost robot, and some on-board peripherals. This robot is used by the department of Computer Science at University of Innsbruck for teaching students autonomous systems. The robot is driven by an Android phone which communicates with the robot's internal peripherals. The components in use for this communication process introduce various problems which are resolved by this project to ensure better user experience.

Possible prototypes have been evaluated to find the best possible solution for the task. Although the best solution still has some flaws, another prototype was created which brings some new benefits along. Among them are easy to use and easy to maintain interfaces. Together with the prototypes documentation and examples have been crafted to ensure easy access to the material for students. Modularity of the robot has been improved which enables other people to easily enhance the robots capabilities.

The component used in the final solution consists of a microcontroller which is programmed to read in data from an Android phone and controls the robot accordingly. Apart from hardware the microcontroller's firmware has been written together with and example Android application.

Acknowledgments

First one on the list has to be Simon Haller who provided a huge amount of support even while being busy.

Also huge thanks to Lukas Schöpf, setting up a PCB printing lab really helped with creating the prototypes and final solution.

Thank you to Senka Krivić, my fellow students, my family and the friendly members of the IIS Team which always provided support when needed.

Contents

List of Figures

Declaration

By my own signature I declare that I produced this work as the sole author, working independently, and that I did not use any sources and aids other than those referenced in the text. All passages borrowed from external sources, verbatim or by content, are explicitly identified as such.

Signed: . . . . . Date: . . . . . .

Chapter 1

Introduction

The department of Computer Science at the University of Innsbruck utilizes a set of relatively small, portable robots. These robots have been assembled by hand and are used for research as well as for teaching. In summer term 2013 students have taken advantage of the new robots in one course for the first time. They had to combine the robot with their Android phone to create an autonomous unit.

While the Android phone's camera is now the robot's new vision, the powerful Central Processing Unit (CPU) which comes with modern Android phones, like the 2.3 GHz Quad-core inside a LG Nexus 5, is the robot's new brain. Complex tasks can be broken down into smaller, simpler subtasks from a high level perspective using Java as programming language.

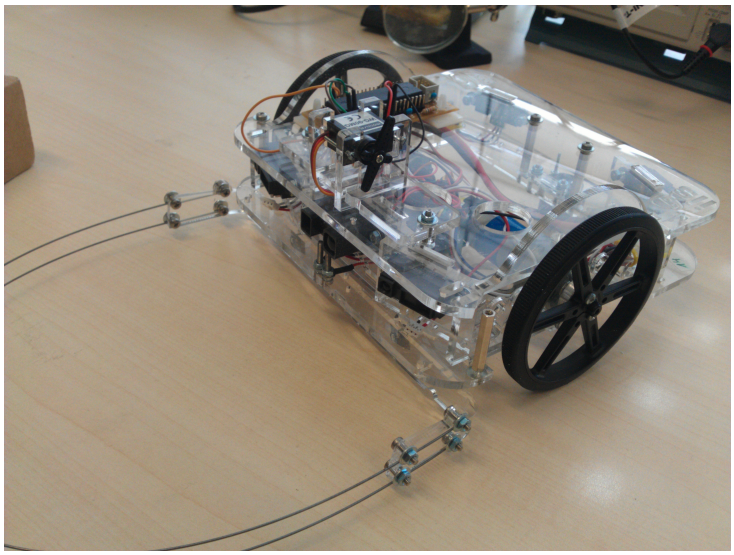


Figure 1: Front view of the robot

Figure 1 shows a front view of the robot. It is made up of two layers stacked on top of each other. While the lower (interior) layer holds the battery, motors, sensors as well as a *control board*, the upper (on top) layer holds a servo motor (section 3.3) together with the *communication board*.

1.1 Context and Motivation

As already mentioned, summer term 2013 was the first time students were able to work with these self created robots. It soon became clear that there were still issues in the design as well as in component interaction and programming. Probably the most frustrating issue for the students was the communication part between their Android application and the robot itself. The *IOIO* board (section 3.1) is used as *communication board* to fill the gap between high level Java programming and low level robot control. Although the *IOIO* board is in itself a very powerful board and can be used as *communication board* as well as *control board* its usage can lead to some very complex problems. This is due to its Application Programming Interface (API) desgin, which is highly Android specific and multi threaded. The high Android focus makes it hard to use in combination with different (embedded) systems, like a Raspberry Pi[1] or Beagle Bone Black[2]. Additionally the IOIO Android communication is highly dependant on the IOIO firmware version, IOIO Library version and the Android version. Multi threading on the other hand makes debugging certainly a whole lot more complex and might lead to some race conditions if not properly synchronized with other components.

1.2 Contribution

This work is dedicated to finding a better solution for communication between an Android phone and the robot. This means evaluating possible replacements for the IOIO board as well as fully develop and manufacture the best solution.

There are no special constraints, the solution should be practical in terms of cost, dimension, complexity, portability, etc. A good all-round solution is targeted.

Apart from a printed circuit board (PCB) and its firmware, good documentation including examples should be created for easy access. Students should have no problems using this example as basis for their projects.

1.3 Thesis Outline

The entire thesis consists of 7 chapters. The current chapter introduces and motivates this thesis. The next chapter provides a detailed overview of the robot itself, followed by some background information needed to fully understand this work. In chapter 4 the evaluation process is described. The fully developed prototypes are documented in chapter 5 and 6. Chapter 7 covers the Android test application.

[1] see http://www.raspberrypi.org/
[2] see http://beagleboard.org/Products/BeagleBone+Black

Chapter 2

The Robot

Introduction

This chapter will introduce the robot together with its components. After describing the 2 layer layout interaction between the components will be explained.

2.1 Robot Interior

The robot's chassis is made up of 4 acrylic glass plates snapped together with screws. Two plates on the sides, on the bottom the other on top. The top and bottom plate create enough room between them so the battery, some sensors, 2 DC motors as well as the control board fit in there.

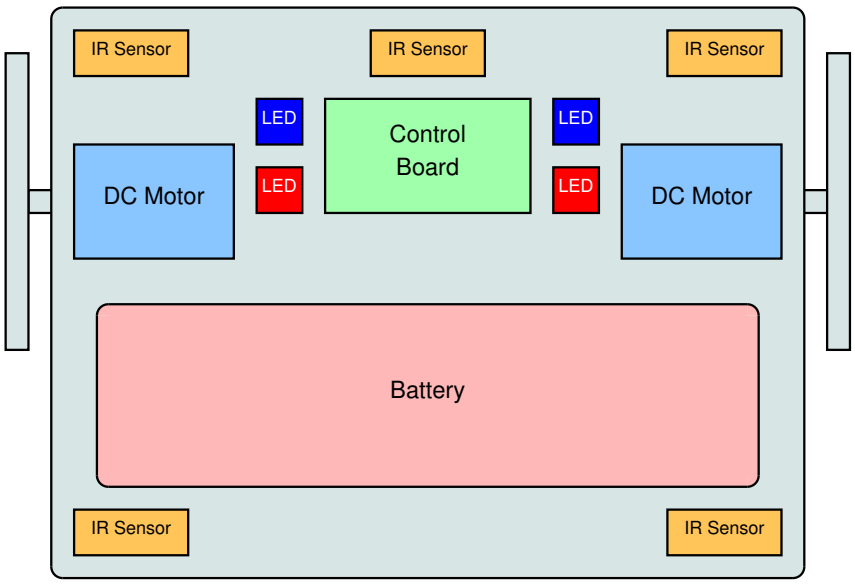


Figure 2: Robot interior outline

Figure 2 provides an overview of the interior components. The colors in this figure match the colors in fig. 4 for consistency. The battery provides power to all components on the inside as well as the servo motor on top (section 2.2).

The *control board* holds all required electronic parts, distributes power from the battery to where it is needed, holds two H-Bridge (section 3.4) motor drivers (one for each DC motor) and an ATmega644 AVR microcontroller (section 3.7) to control the robot, hence the designation “control board”.

All 5 sensors are Infra Red (IR) proximity sensors (section 3.6) which are used to grant the robot more situational awareness. Apart from these sensors bumpers can be added if necessary. The robot also features 2 red and 2 blue Light Emitting Diodes (LEDs) which can be controlled separately.

Also the control board has an on/off button located on the underside which can be accessed through a hole in the acrylic glass plate.

2.2 Robot On Top

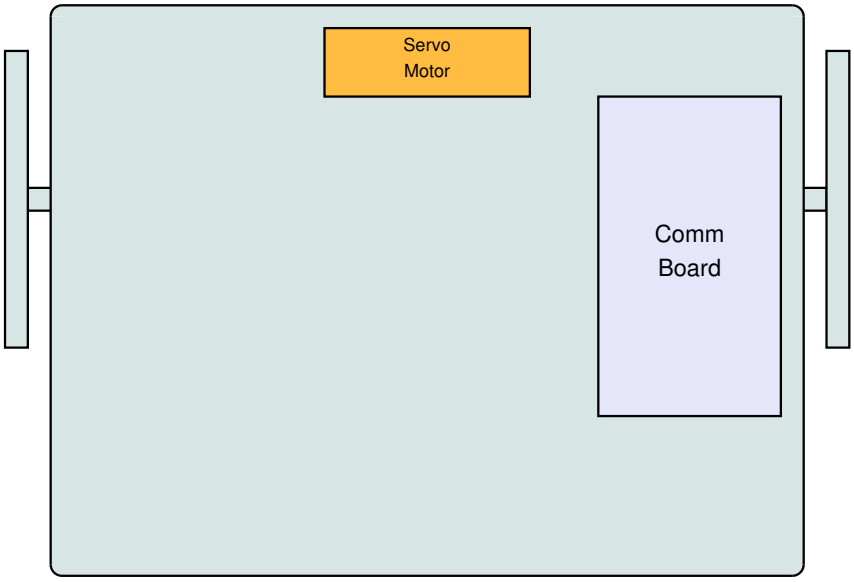


Figure 3: Robot on top outline

Figure 3 shows the top layer. Top and interior layer are connected with wires through holes in the top plate. The servo motor is connected to a bar in front of the robot (fig. 1) via a fabric thread. By altering the servo motor’s position the bar can be lowered to ensnare some object (like a ball) and be raised afterwards to release it.

The communication board is placed on top as well to provide easy access since an Android phone has to be connected to it via the Universal Serial Bus (USB).

2.3 Component Interaction

Communication between the two boards happens over an Inter-Integrated Circuit(I2C) bus (section 3.5). The communication board receives commands from the Android phones, and forwards them to the control board accordingly. For some reason I do not know, the servo motor has been connected to the communication board instead of the control board by design, hence the communication board has to control the servo motor as well.

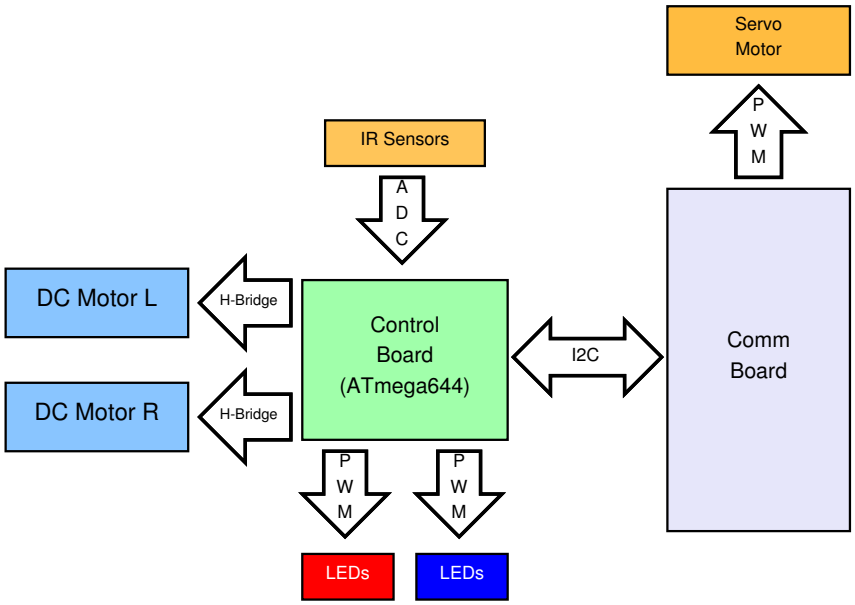


Figure 4: Robot component interaction

Controlling the servo motor is done via a pulse width modulation (PWM) signal (section 3.2). Where the pulse width is proportional to the servo motor's position.

IR sensors output a voltage signal which correlates to the distance between the sensor and an object in front of the sensor. Do note that this correlation is *not* linear (section 3.6). The provided voltage output is read via an analog to digital converter (ADC) and is then available as digital value which can be passed to the Android application.

Brightness of the LEDs can be controlled separately via PWMs. Here the PWM signal's mean value corresponds to the LEDs brightness.

Both DC motors can be controlled individually using H-Bridge drivers. The motors also provide feedback about how much they have turned, which is read by the microcontroller. This way telemetry can be used to control the robot as well.

Chapter 3

Background Information

Introduction

This chapter will state information needed in the upcoming chapters. The reader can skip this part for now and come back later when in need. This chapter is aimed at computer scientists not that familiar with electronic concepts and designs.

3.1 IOIO Board

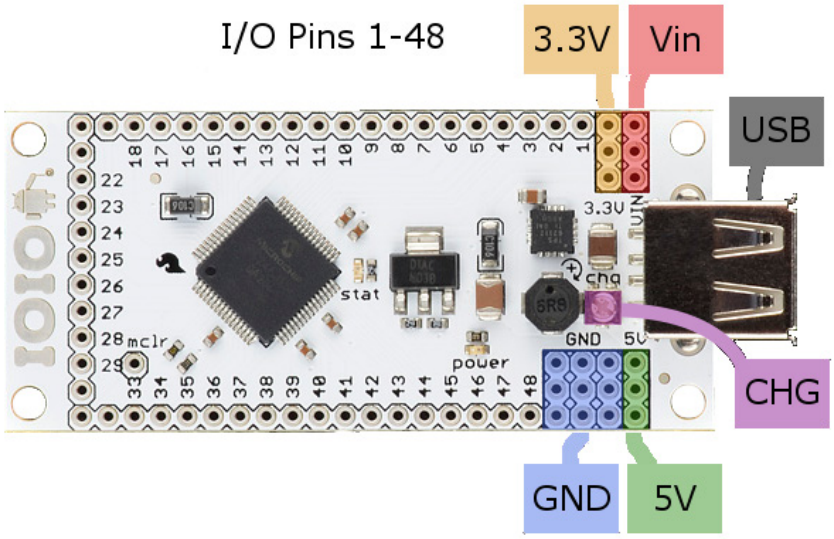


Figure 5: IOIO Board outline [6, /Getting-To-Know-The-Board]

Figure 5 shows an outline of the IOIO Board including annotation on some important connectors. The board operates at 3.3 V and features pin headers on the side for easy access. A PIC24F[1] is used as control unit.

[1] Peripheral Interface Controller

Pins 47 and 48 are used for I2C communication, pin 10 is used as PWM output to control the servo motor.

The IOIO board comes with USB capability including a driver and a multi threaded API for Android. The USB interface can also be used for a bluetooth adapter which enables wireless communication to the IOIO board.

3.2 Pulse Width Modulation

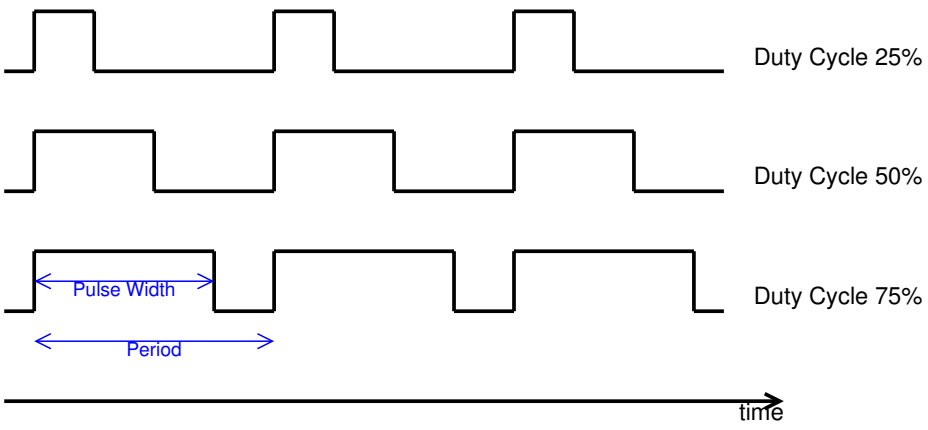


Figure 6: 3 PWM signals with different duty cycle

A PWM signal is a periodic signal switched between logic high (commonly $5\,\mathrm{V}$) and logic low (commonly $0\,\mathrm{V}$). The *logic high time* duration is called pulse width and the ratio between pulse width and period is called duty cycle (stated in percent). Figure 6 provides 3 example PWM signals with different duty cycle. By varying the duty cycle the signal's mean value changes accordingly. This can be used to control a LED's brightness if the frequency is high enough. [1, /en/Tutorial/PWM]

$$\text{Duty Cycle } [\%] = \frac{\text{Pulse Width} \times 100}{\text{Period}} \tag{1}$$

Also note

$$\text{Frequency}[\text{Hz}] = \frac{1}{\text{Period}} \tag{2}$$

3.3 Servo Motor

Servo motors are usually driven by a PWM signal, this signal is used to control the servo motor's position. The desired position can be reached by altering the duty cycle accordingly, which allows for precise angular positioning. The motor has 3 pins, two of them are used for power supply, the PWM signal should be relayed to the third pin.

This application uses one motor by Modelcraft (WG-90MG). They require a PWM frequency of about $50\,\mathrm{Hz}$. This servo motor is used to control the cage in front of the robot which can be used for catching different objects.

3.4 H-Bridge

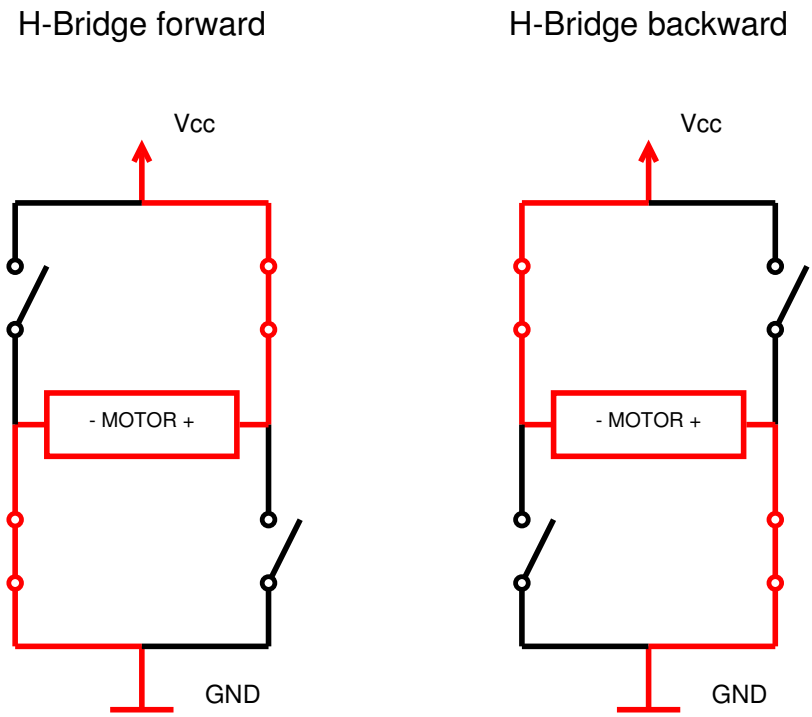


Figure 7: H-Bridge at work

A H-Bridge is a special kind of a motor driver used in combination with a DC motor. These drivers make it easy to control the motors from digital devices like a microcontroller. In this application two VNH3SP30-E chips are used, one for each motor. They provide a digital interface to state the motors direction and a PWM interface to control velocity.

The motor is wired between 4 switches. By closing 2 switches like shown in fig. 7 current flows through the motor in one direction. If the 2 other switches are closed instead, current flows in the opposite direction. Velocity is altered by limiting the current flowing through the motor. This limit is regulated by the PWM signal.

More information on the VNH3SP30-E can be retrieved from [13].

3.5 Inter-Integrated Circuit (I2C)

The Two-wire Serial Interface (TWI) is a bus system developed by Philips in the early 80's and also known as I2C.

> [It] is ideally suited for typical microcontroller applications. The TWI protocol allows the systems designer to interconnect up to 128 different devices using only two bi-directional bus lines, one for clock (SCL) and one for data (SDA). The only external hard- ware needed to implement the bus is a single pull-up resistor for each of the TWI bus lines. All devices connected to the bus have individual addresses, and mechanisms for resolving bus contention are inherent in the TWI protocol. [3, p. 169]

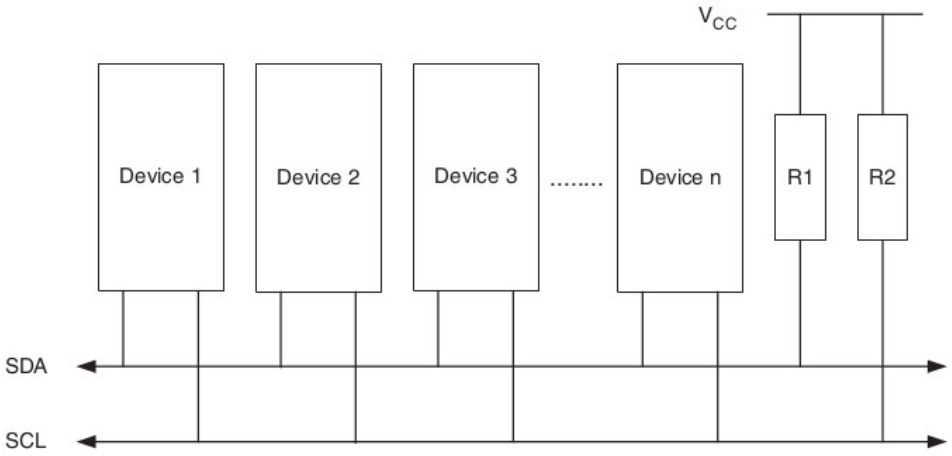


Figure 8: Outline of an I2C Bus with multiple devices [3, p. 169]

Figure 8 sketches such a bus setup with multiple devices. For this application only 2 devices are connected via this bus, one operates as master the other as slave.

Transmitting information via the bus happens by either pulling the SDA wire to ground to indicate a logic 0 or switching the output pin on the SDA wire to high-impedance - the pull-up resistor will drag the wire up to V_{CC} potential indicating a logic 1.

Some of the advantages of the I2C bus system are that communication is possible using only two wires, components maintain a simple master/slave relationship and there are no strict transfer rate limitations like in RS232[2].

3.6 IR Sensor

The IR sensors are used to detect obstacles. This sensor measures the distance using a positive sensitive detector (PSD) and an infrared emitting diode (RED). According to the datasheet [12] its measuring distance ranges from 100 mm to 800 mm.

Using this infrared sensor it is easy to measure distances fast and accurate. It is supplied with 5 V and provides analog output voltage related to the measured distance. Figure 9 shows this characteristic. Note that the relation between distance and sensor output is *not* linear.

The output of the sensors are read by ADCs on the control board.

3.7 Microcontroller

A microcontroller is an integrated circuit (IC) which can be split into 3 different parts. Memory, peripherals and the microprocessor(s). Among variables and constants memory can hold a sequence of instructions (a program) which can be executed by a microprocessor. Each processor consists of an Arithmetic Logic Unit (ALU) and some logic for stepping through the program and decoding each instruction.

[2] see http://en.wikipedia.org/wiki/Rs232

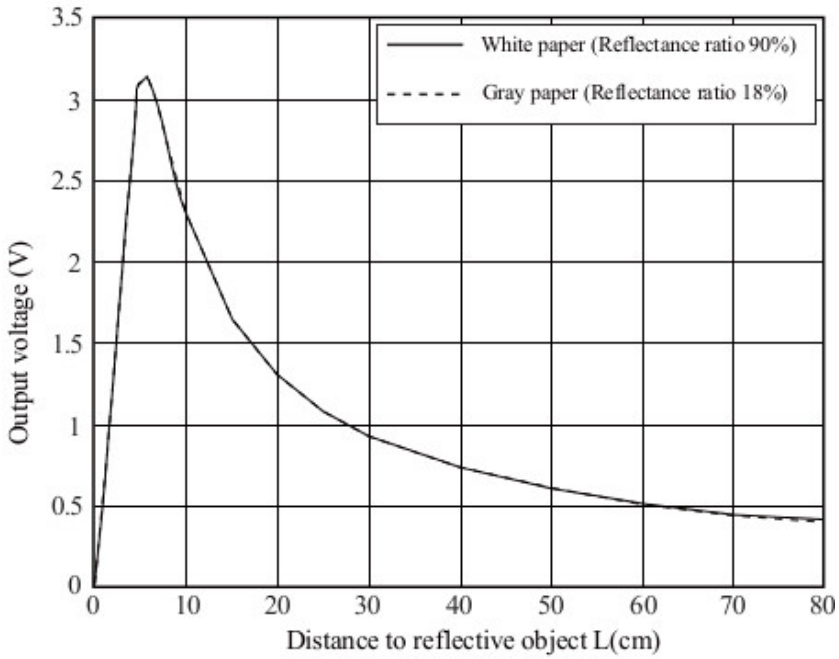


Figure 9: Relation between measured distance and output voltage [12]

Peripherals are made up by a set of mostly digital components like timers, counters, analog to digital converters, PWM generators, low level communication interfaces, general purpose input/output, comparators, etc. These can be configured by the microprocessor.

Since most of the components inside the chip are digital circuits a clock signal has to be generated. This clock signal is distributed among the chip. Because peripherals are supplied by this clock signal, they can operate in parallel to the micro processor. For instance, a timer will still run after it has been started while the processor is doing some calculations. Some microcontrollers have multiple processors which can be used independently, others only feature one single processor. In case of the control board a ATmega644 AVR microcontroller is used. This is a high-performance, low-power 8-bit Microcontroller by Atmel featuring an advanced RISC Architecture. More information can be obtained from [4].

3.8 Microcontroller Software

In order to use a microcontroller one has to program it. This is usually done by utilizing the C programming language. For nearly all microcontrollers a compiler bundle including libraries is available.

A typical microcontroller program consists of two phases, the setup (also known as initial) phase and the running phase. During the setup peripherals are configured, in-memory datastructures are setup, etc. The running phase then serves the purpose of the microcontroller.

To keep microcontroller programs organized the code is split into multiple modules. Each module consists of a header file stating exports and a C file containing the implementation. Usually each module has a dedicated purpose and is related to either a software or hardware component.

Table 1 lists the modules of the control board firmware.

Module	Description
adc	setup analog digital converter and get analog values
comTwi	setup I2C hardware and manage communication
flexport	contains various macros for basic bit operations
iMotorCtl	main entry point
led	setup and change PWM signal for LEDs (uses Timer2)
mBridge	low-level motor control using H-Bridges (uses Timer1)
mCtl	high-level motor control (uses mBridge module)
pwrManagement	manages switching to sleep state and wakeup
sensor	setup sensors and get sensor data (uses adc module)
timer	internal use for motor control (uses timer0)

Table 1: Control board modules

3.9 In-System Programming (ISP)

Programmable logic devices, microcontrollers and other embedded systems can be programmed via ISP.

> In-System Programming allows programming and reprogramming of any AVR microcontroller positioned inside the end system. Using a simple Three-wire SPI interface, the In-System Programmer communicates serially with the AVR microcontroller, reprogramming all non-volatile memories on the chip. [2]

In order to use this feature a ISP programmer is required, these programmers are relatively cheap but do not provide debug features like JTAG. One end of the programmer is connected via USB to a computer, the other via a header to the device holding the AVR chip. See [2] for more information.

3.10 Universal Asynchronous serial Receiver and Transmitter

Communication over UART is similar to RS232 but using either 5 V or 3.3 V as logic 1 (and 0 V for logic 0). Compared to USB this interface does not require any special handshake mechanism. Both devices use the same parameters in order to talk to each other.

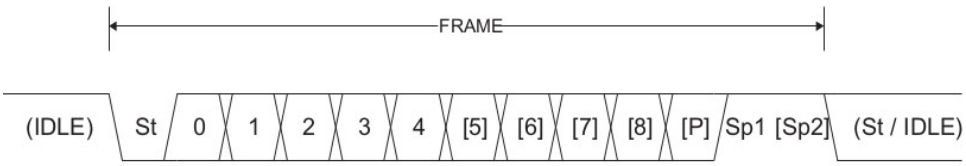


Figure 10: An outline of a single frame send over UART devices [3, p. 144]

Apart from Baud rate (speed) the protocol defines a format how data is transmitted. This format can be viewed in Figure 10, it consists of following items.

St Start bit

0-8 Data bits

P Parity bit

Sp Stop bit

The number of data bits as well as the number of stop bits are variable to some degree, but both device have to use the same settings. Also parity can be set to either odd or even.

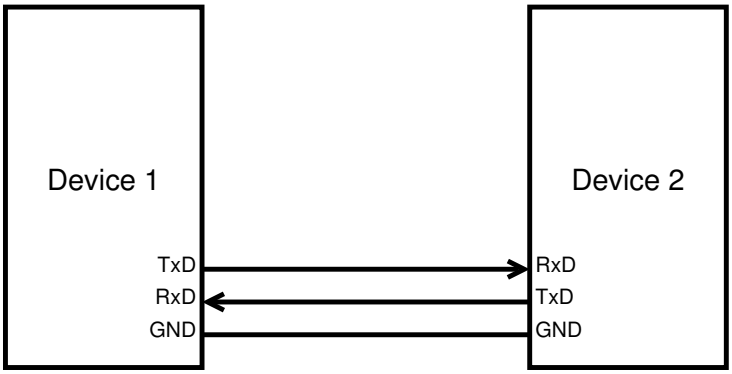


Figure 11: Connection between 2 devices using UART

For a working communication only RxD and TxD wires are required. Flow control is available through additional wires although it is not used in this application. A common ground reference has to be used.

3.11 USB Host

A USB connection between two devices is similar to a master slave relationship. In this case the two devices are distinguished as host and accessory. By definition the USB host has to power the accessory and initiate the handshake. Most of the bus management has to be done by the host, for more details see the USB specification[3].

3.12 USB On The Go (OTG)

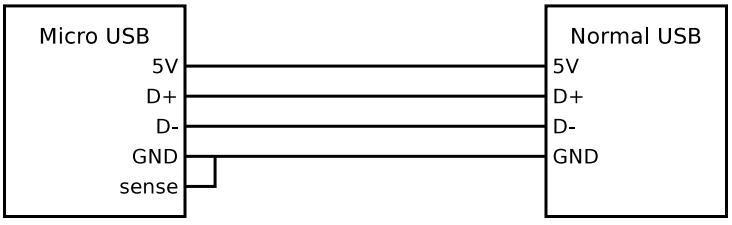


Figure 12: Internal USB OTG cable wiring

USB On the Go (OTG) is a specification which allows certain USB devices which usually operate as USB accessory to become a USB host. This can be very useful for a smart phone because this way it can use its USB port to access a thumb drive or utilize a USB to serial converter.

[3] see http://www.usb.org/developers/docs/usb20_docs/

A special USB cable must be used for USB OTG and for this setup to work. This cable pulls the sense pin present on mini USB and micro USB ports down to ground as can be seen in fig. 12. This notifies the internal USB chip that it should switch into USB host mode.

Chapter 4

Evaluation Process

Introduction

This chapter will present some possible solutions (candidates) and rank them against each other. A prototype with the most promising solution will then be created. For the evaluation process certain categories will be explained and used, each candidate will receive a score determined by me. The candidate with the highest score will then be considered *the best solution*.

4.1 Categories

Following categories will be used for the evaluation process.

Cost needed components and manufacturing should be cheap since multiple robots have to be equipped with the new hardware.

Simplicity easy to build as well as easy to maintain are huge benefits not only for documentation purposes but also for continued work.

Availability all required components must be available for purchase and should be delivered in an acceptable time frame.

Modularity the robot's components are already designed in a modular fashion, therefore a solution which does not break current modularity is preferred.

Future Oriented even though a candidate might work for now, things will probably change in future. New Android versions will ship, new demands will be made against the robot. A good solution has to be robust in order to support upcoming changes.

4.2 FT311D

According to the datasheet [8] this chip provides six interfaces the user can chose from, among them UART and I2C. Since the control board communicates with the IOIO board via I2C this chip could easily replace it.

> The FT311D is a Full Speed USB host specifically targeted at providing access To peripheral hardware from an Android platform with a USB device port. [8]

Of course to work for this, a new board has to be created which holds the chip including all required components. But there is a PWM output needed in order to control the servo motor, this output has to be provided by the new communication board or made available on the control board.

4.3 Arduino Uno

This famous board is commonly used for simple electronic tasks and has an enormous amount of hardware *and* software extensions. Apart from I2C and USB communication it can also control the servo motor and provides huge potential for future work. It uses an ATmega328 microcontroller. [1, /en/Main/ArduinoBoardUno]

The downside here is that the boards footprint is much bigger than the IOIO board and hence it is a complete board extending it beyond capabilities of its extension headers will be hard since a new board has to be designed from scratch.

Arduino provides more products similar to the Arduino Uno, some with smaller packages and different extension capabilities. Despite a complete board is easy to use and requires less work, a single chip is preferred since a new PCB can be designed to suite the given requirements.

4.4 AVR ATmega32

Similar to the chip on the Arduino Uno this microcontroller can handle the task nicely. There are also AVRs available which contain USB communication peripherals, but since the programming has to be done by the developer I prefer the use of a USB to serial (UART) converter.

> The Atmel®AVR®ATmega32 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the ATmega32 achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed. [3, p. 3]

4.5 Raspberry Pi Model B

The Raspberry Pi is a credit card sized board containing an ARM processor and including various extension capabilities. This candidate is not just intended to replace the IOIO board but also the necessity of having an Android phone. Although the processor is not that powerful compared to a modern Android phone, the board is able to do basic image processing using a web cam or the original Raspberry Pi camera. Only the CPU can be used because the graphics processing unit (GPU) cannot be programmed directly. But the original camera can take advantage of the GPU, hence a HD image can be read from the camera without interfering with the CPU's computations.

> Model B is the higher-spec variant of the Raspberry Pi, with 512 MB of RAM, two USB ports and a 100mb Ethernet port. [11, /product/model-b/]

4.6 Beagle Bone Black

The Beagle Bone Black is quite similar to the Raspberry Pi, it costs slightly more but has a more powerful CPU. Apart from the CPU it holds 2 additional less powerful CPUs which can be programmed individually. Like the raspberry pi this would be an alternative to the Android phones, although it is not as powerful as modern day smart phones are.

> BeagleBone Black is a $45 MSRP community-supported development platform for developers and hobbyists. Boot Linux in under 10 seconds and get started on development in less than 5 minutes with just a single USB cable. [5, /Products/BeagleBone+Black]

4.7 Evaluation

Candidate	Cost	Simplicity	Availability	Modularity	Future Oriented	Total
FT311D	5	5	5	4	3	**22**
Arduino Uno	3	3	4	4	3	**17**
AVR ATmega 32	4	3	5	5	4	**21**
Raspberry Pi B	2	2	4	3	4	**15**
Beagle Bone Black	1	1	2	3	4	**11**

Table 2: Evaluation score, bigger is better

Table 2 shows the final evaluation of the candidates. A candidate can score points between 1 and 5 in each category, the more points the better. While the Raspberry Pi and Beagle Bone Black would be good solutions to replace the whole Android related setup, using them goes beyond the scope of this work since only the communication board should be replaced. Using an Arduino Uno would be cheap and simple but building on top of it will introduce new problems. In the end a self created board will turn out to be more beneficial. Using an ATmega32 opens up a whole lot of possibilities but for now keeping it cheap and simple with a focus on extensibility and modularity should get the job done.

Chapter 5

FT311D Prototype

Introduction

The FT311D chip will be introduced in this chapter. General operations will be explained including its setup. Next the prototypes will be described concluding with an explanation why this chip is not used in the final solution.

5.1 The Chip

According to the evaluation process the FT311D chip provides the best solution for the stated problem, this chapter will document the creation of a prototype utilizing this chip.

The FT311D is a USB Host chip made by FTDI[1] and is dedicated to Android devices. The chip itself contains multiple interfaces the user can select from. [8]

Following interfaces are available.

- 7 GPIO lines
- UART
- 4 PWM channels
- I2C master
- SPI[2] master
- SPI Slave

Most notably this chip does all USB protocol handling required for you. Example applications (Android apps) for each interface are provided by FTDI. The chip itself runs at 12 MHz and requires an external crystal. Inputs are 5 V tolerant despite the chip's supply voltage is 3.3 V.

A block diagram of the chip can be viewed in fig. 13. The Pins CNFG2 - CNFG0 are used to select the desired interface. Table 3 shows the pin configuration for each interface configuration (mode).

[1] Future Technology Devices International Ltd.
[2] Serial Peripheral Interface

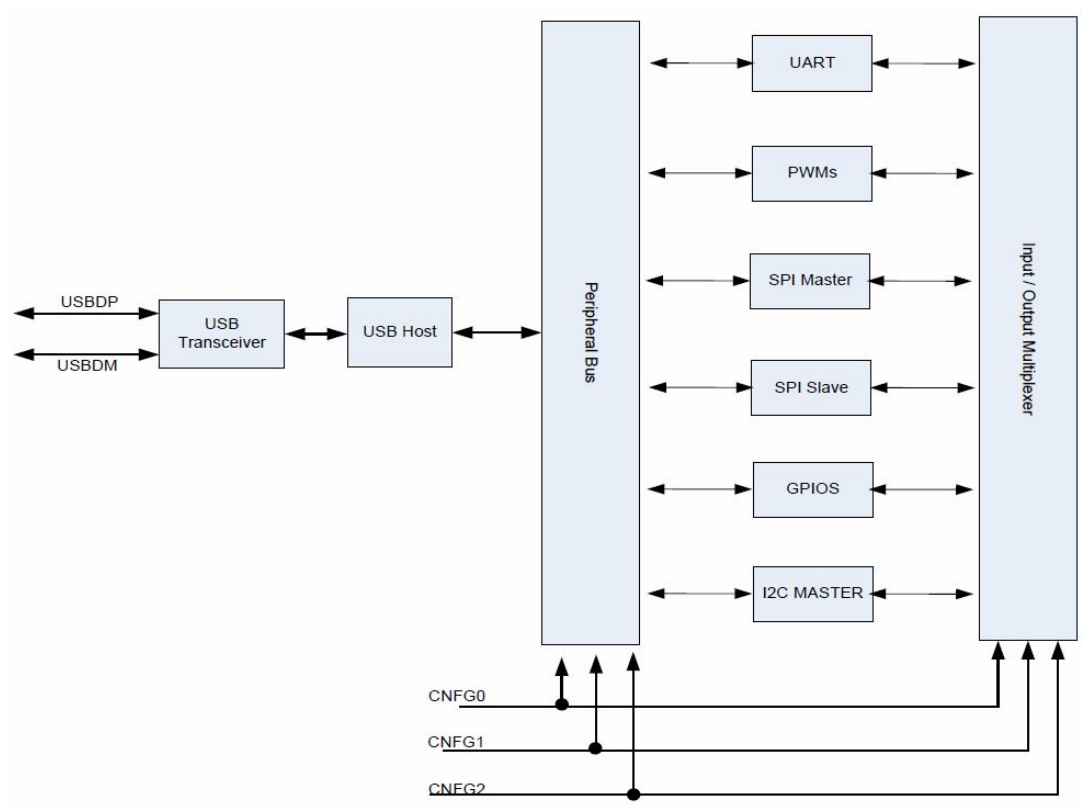


Figure 13: Block diagram of the FT311D internals

GND means the corresponding pin should be connected to ground indicating a logic 0, open will result in a logic 1 since the chip has internal pull-ups.

2 prototypes have been created, one using the UART configuration another using the I2C master configuration. Connecting the chip directly to the control board using the I2C bus should work since the control board is configured as I2C slave.

CNFG2	**CNFG1**	**CNFG0**	**Mode**
GND	GND	GND	GPIO
GND	GND	open	UARO
GND	open	GND	PWO
GND	open	open	I2C masteO
open	GND	GND	SPI slavO
open	GND	open	SPI master
open	open	GND	factory testing (do not use)
open	open	open	defaults to GPIO

Table 3: CNFG2 - CNFG0 configuration [8]

5.2 Prototypes

The following layout is based on the example configurations provided by the data sheet [8]. The circuit itself has to be powered by the robots battery, it can *not* be powered by the phone since the FT311D will be USB host. By definition the USB host has to provide power for the accessory. For easy connection a pin header is provided which connects wires from the control board to the prototype. 5 V have to be passed over this header, otherwise we cannot provide 5 V for the accessory. Although the chip itself is powered by only 3.3 V a voltage regulator is needed. Capacitors have been added to compensate unforeseen voltage drops as well as a reset circuit for the chip because it is considered good practice.

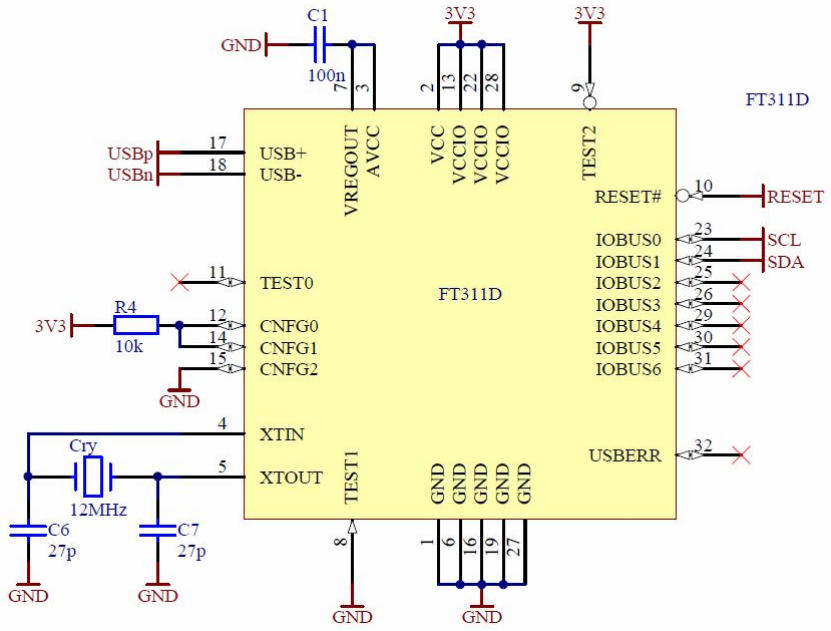


Figure 14: FT311D circuit in I2C configuration

Figure 14 displays the main circuit. It has been derived from the data sheet.

Also another layout has been created utilizing the UART configuration. The circuit looks exactly the same except CNFG2-CNFG2 are wired differently. Also net labels have been altered from SDA and SCL to RxD and TxD to use the convention.

From the top, both boards look exactly the same, fig. 15 shows one of the boards, a 1 Euro coin has been placed beside the board to get a better understanding of its size.

5.3 Android Version Problem

First attempts establishing a working connection between the chip and an Android phone failed. A Nexus 4 smart phone is used running Android version 4.3 at this very moment. The phone works with the IOIO board. After checking the circuit again and verifying the assembling of the board, other devices have been tested.

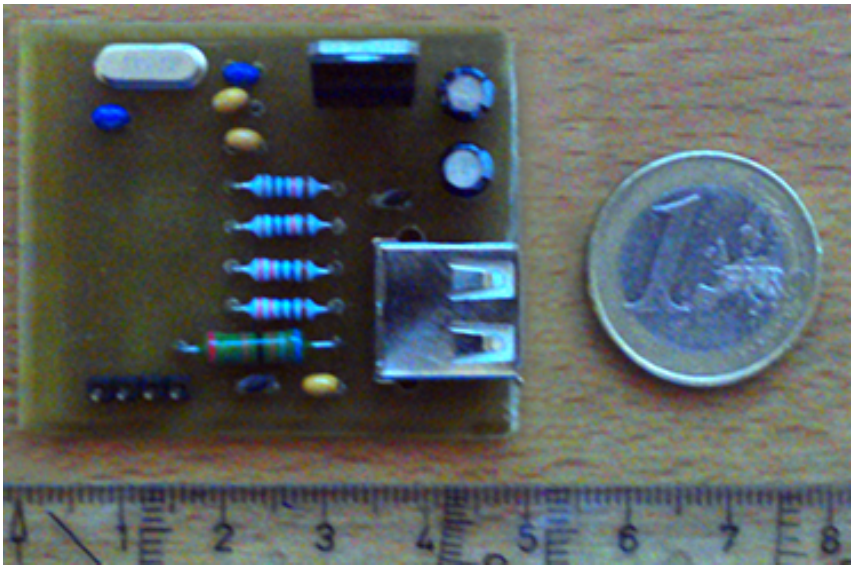


Figure 15: FT311D prototype printed circuit board

I chose to test a different device instead of debugging the current setup because of the complexity a USB setup introduces. Other Android devices turned out to work just fine with the prototype, though after some digging around and more testing it became clear. The Android version is the problem, devices running an older Android version (3.1 - 4.0) work without a problem. But when using a newer version (4.1 or higher) the phone does not recognize the chip.

Apparently similar issues have happened in the past already, FTDI released a new revision[3] to fix some errors between Android version 3.X and 4.01. Although the chip on the prototype is of revision C and is therefore the latest available, it is still not recognized.

After encountering this issue, other downsides of this setup became clearer. Even though this solution is intended to work out of the box, it is highly dependent on the support of FTDI and there is no guarantee to work with future versions. New versions of the Android OS will be released in future and might not work with the already shipped and built-in chips. And since there is no way to update the firmware, old chips must be replaced with new ones.

And even though the robots are used together with Android right now, a communication board which is not Android dependent might proof useful for the future.

Because of these downsides I chose to create a second prototype using an ATmega32 microcontroller as alternative.

[3] see http://www.ftdichip.com/Support/Documents/ProductChangeNotifications/PCN_FT_014.pdf

Chapter 6

ATmega32 Prototype

Introduction

The prototype described in this chapter takes a different approach compared to the FT311D chip prototype. Again the overall structure of the board will be explained followed by hardware and software.

6.1 ATmega32 microcontroller

In contrast to the FT311D prototype, which is running as USB host, this prototype runs as USB accessory. Hence the Android phone must be capable of switching its internal USB chip into host mode (section 3.11 for more information). This change opens up a whole new set of possibilities, if this prototype is designed well, the robot can be decoupled from Android and use a more platform independent communication layer. With such a communication layer one can easily swap the Android phone for an embedded system or simply connect the robot with a laptop for more sophisticated tasks.

But in order to achieve this, the communication board itself has to be made up of more than just a simple converter. Using a more powerful chip on the board also introduces some benefits. On the one hand, the communication board can take care of controlling the servo motor, provides a simple protocol suited for the robot and do various optimizations like caching sensor data. Being able to control the servo motor provides the benefit of not having to change the control board, the communication between the two boards can remain on using the I2C bus.

> The Atmel®AVR®ATmega32 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the ATmega32 achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed. [3, p. 3]

This microcontroller has various peripheral features which can be used for all sorts of things. Following list provides on overview of the most important ones for this application.

- Two 8-bit Timer/Counters
- One 16-bit Timer/Counter

- 4 PWM channels
- 8 10-bit ADC CHannels
- I2C interface (master and slave)
- UART
- SPI (master and slave)
- watchdog timer

There are other, similar chips available which provide different and sometimes even more features but I picked this one since I have used it in many previous application. It should suffice for this prototype and can be changed later on if necessary.

Like all microcontrollers this chip has to be programmed before it does something useful, this is done by using the C programming language together with controller specific libraries provided by Atmel. All required tools (compiler, linker, flash utility, etc.) are available for GNU/Linux. In general, written C code gets compiled and linked into a single binary file which then gets flashed on the chip using a flash utility program together with a (hardware) programmer. The chip is programmed via ISP (section 3.9).

Even though there are microcontrollers available by Atmel which contain fully programmable USB peripherals, I chose not to use one of them for the sake of simplicity. Implementing the required components for a fully working USB communication will require a lot of time and may introduce bugs. Also the produced code might not be easy to maintain. The UART interface provides a simple alternative because there is no handshake or special control sequence involved. And because the amount of data transmitted between the communication board and the Android phone is quite small, there is no need for high speed USB.

6.2 FT232

Despite the fact that nearly all ARM chips integrated into an Android phone feature at least one UART peripheral, the UART interface is usually not available for external use. The practical way of communication with an Android phone is via USB, but since I have decided to not use USB on the communication board side, a segment between is required. A simple USB to serial converter will do the job. These serial converter can be acquired from different vendors and with different specifications.

The chip used in this application is designated FT232BL and designed by FTDI.

There is no special configuration needed in order to use this chip, all commonly used platforms can interact with the chip out-of-the-box. [7].

6.3 Layout

Starting with the FT232BL USB to serial converter in fig. 16 [7]. Next the ATmega32 chip is featured in fig. 17. A 100 nF capacitor is provided to prevent unforeseen voltage drops and a reset circuit has been added. The 16 MHz crystal is decoupled by two 22 pF capacitors.

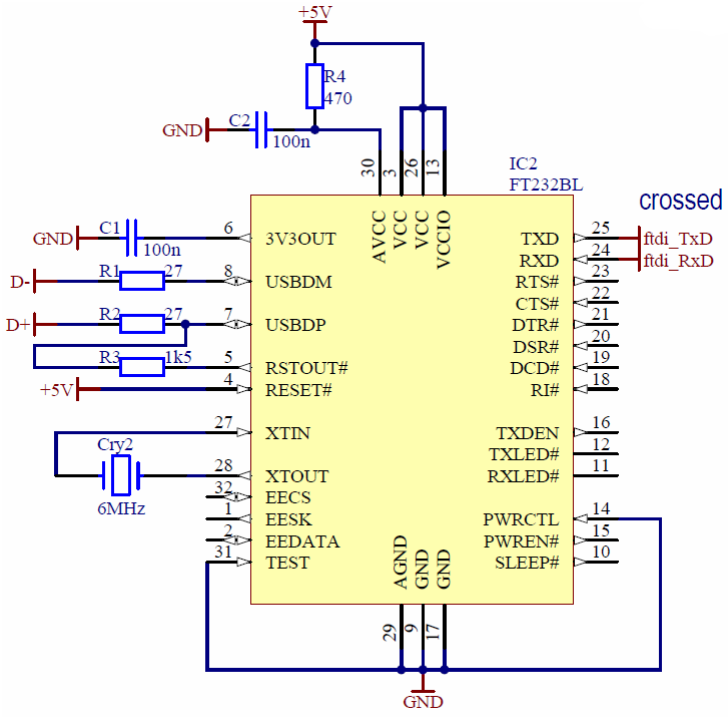


Figure 16: FT232BL circuit

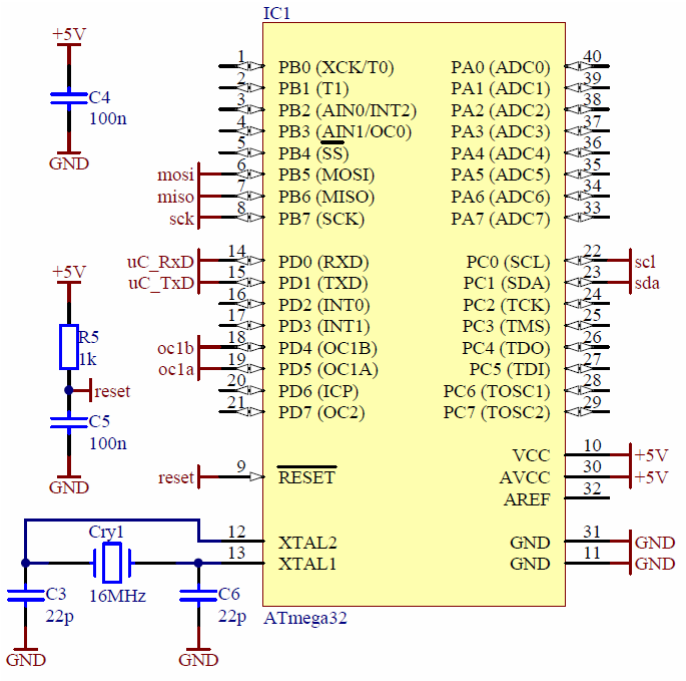


Figure 17: ATmega32 circuit

Figure 18 shows all pin headers placed on the prototype. They are used for communication with an Android phone or a similar device, to communicate with the control board, provide power and a common ground reference.

When connecting an Android phone to the prototype a USB OTG cable must be used, therefore the

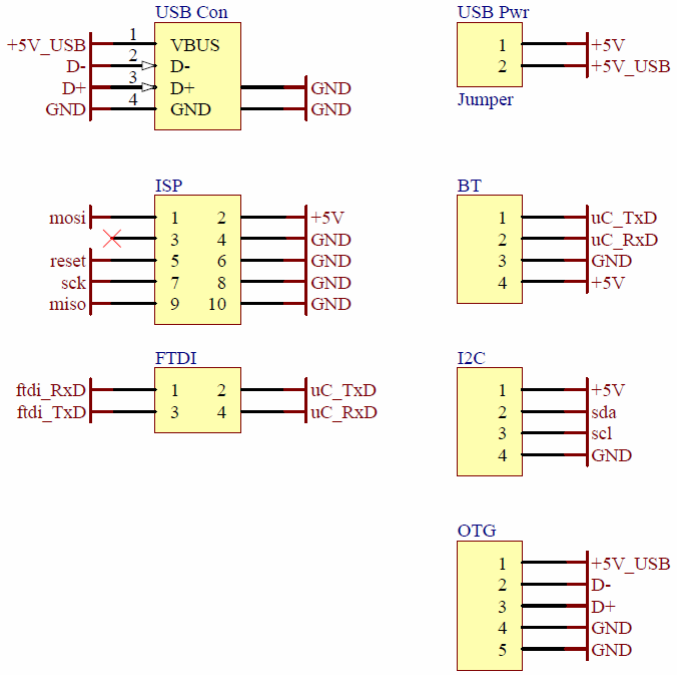


Figure 18: ATmega32 pin headers

pin header OTG can be used as interface. As an alternative interface the USB jack can be used, both of them are connected to the FT232BL chip. Apart from USB a Bluetooth module (HC-07) can be used as communication channel. Therefore the Bluetooth module has to be connected to the controller via UART using the BT pin header. But in order to use the Bluetooth module jumpers on pin header FTDI must be taken off to disengage the FTDI chip from the UART interface. I2C can be used to connect to the control board. ISP features a standard ISP header pinout to connect a programmer to the board.

As the Android phone is USB host in this setup, it provides the required power for the prototype, this way there is no need to connect a 5 V wire from the control board to this board although ground has to be connected to ensure a common reference. If a Bluetooth module is used instead the control board has to provide the required supply voltage for the board. This feature can also be used for connecting the Nexus 4 via OTG (appendix A.8).

Note pin header USB Pwr can connect the two 5 V circuits (5 V from USB, 5 V from the robot) which would otherwise be separated. This jumper should be set if the board is powered by USB from a USB host. If the robot's control board provides power this jumper should be cleared.

The assembled board including a mounted micro USB cable can be viewed in fig. 19. An Android phone can be connected via this USB OTG cable.

6.4 Firmware

The program running on the ATmega32 is split into multiple modules which interact with each other to fulfill the required tasks. Since the ATmega32 microcontroller does only feature a single processing core, there is no parallel execution of tasks, but the peripheral components inside the chip run independently from the processor. This means a counter for example keeps counting even though the

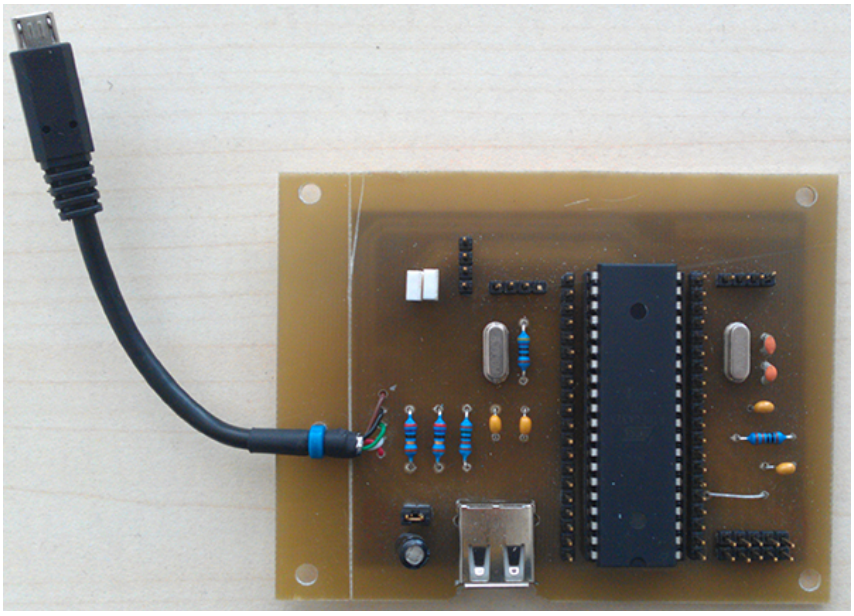


Figure 19: ATmega32 prototype

processor is doing some calculations at the same time. Therefore the counter has to be configured and started beforehand.

The hardware inside the chip is capable of triggering interrupts. These interrupts are handled by their corresponding interrupt service routine. This is somehow similar to a callback or occurring event. Therefore other tasks executed by the processor are suspended and continued after the interrupt has been handled.

Following sections will describe the modules used in this program.

6.4.1 UART Module

```
1  #define uart_printf(fmt, args...)\
2      fprintf_P(&uart_stream, PSTR(fmt LINE_ENDING), ##args)
3
4  extern FILE uart_stream;
5
6  void uart_init(void);
7
8  void uart_put_char(char c);
9
10 int uart_put_char_stream(char c, FILE* stream);
```

Listing 1: UART module exports

Listing 1 shows exported functions and macros. `uart_init()` configures the UART peripheral with hardcoded settings. `uart_put_char()` will be used to send a single character over the UART interface. This function can be combined with `uart_put_char_stream` and the FDEV_SETUP_STREAM macro to create a stream for writing. Now using this file stream a typical `printf()` can be derived. `uart_printf()` works like `printf()`, but writing the resulting string out via the UART interface.

Using the default settings of the chip (which can be observed in [3, p. 162]) together with the instructions in listing 2 the UART peripheral gets initialized to following protocol parameters.

```
1  void uart_init(void) {
2      // 9600 baud
3      UBRRH = 0x00;
4      UBRRL = 0x67;
5
6      UCSRB = (1<<RXEN)|(1<<RXCIE)|(1<<TXEN);
7
8      // setup stream
9      stdout = &uart_stream;
10     stdin = &uart_stream;
11     stderr = &uart_stream;
12 }
```

Listing 2: UART init function

- Baud = 9600 bit/s
- 8 data bits
- 1 stop bit
- no parity
- no flow control

Apart from these parameters the value in register UCSRB state that the receiving part as well as the transmitting part should be enabled and that an interrupt should be triggered when a value has been received. This interrupt will be handled in the corresponding interrupt service routine on vector `USART_RXC_vect`.

```
1  ISR(USART_RXC_vect) {
2      char c = UDR;
3
4      command_t *cmd = command_default();
5
6      switch (c) {
7          case '\r':
8              break;
9
10         case '\n':
11             command_set_execution_flag(cmd, true);
12             log("command execution marked");
13             break;
14
15         default:
16             if (!command_add(cmd, c)) {
17                 log("command to long");
18             }
19     }
20 }
```

Listing 3: UART interrupt service routine

The interrupt service routine, which can be viewed in listing 3, uses functions provided by the `command` module which will be described later on. Since the routine is invoked when a value is received by the hardware the value can be read from the register UDR. If this value corresponds to the ASCII value for carriage return ('\r') it will be ignored. If a line feed ('\n') is received, it will mark the last received command for execution. Otherwise the character will simply be appended to the current command buffer.

Carriage return and line feed are line ending characters commonly used to signal the execution of a typed in command. Sometimes only line feed is used, sometimes carriage return as well as line feed. Because of this we ignore the carriage return character to prevent double execution of the command.

6.4.2 Log Utility

```
#ifdef LOG
#define log(fmt, args...)\
    fprintf_P(&uart_stream, PSTR(fmt LINE_ENDING), ##args)
#else
#define log(fmt, args...)
#endif
```

Listing 4: Log macro

As listing 4 shows, the `log.h` file contains a macro which can be turned on or off and builds on top of the UART functions. This makes logging inside the program easy, also all log output can be turned off with a single line in `config.h`.

6.4.3 Command Module

```
typedef struct command {
    uint8_t buffer[COMMAND_BUFFER_LENGTH];
    uint8_t index;
    bool execute_flag;
} command_t;
```

Listing 5: UART interrupt service routine

The heart of the command module is a simple structure which can be viewed in listing 5. One static instance of this structure is available by default and can retrieved via the function `command_default()`. The members of this structure should not be accessed directly, instead use the exported functions (listing 6).

```
command_t *command_default(void);

bool command_is_full(command_t *cmd);

bool command_get_execution_flag(command_t *cmd);

void command_set_execution_flag(command_t *cmd, bool flag);

bool command_add(command_t *cmd, uint8_t b);

void command_reset(command_t *cmd);

bool command_exec(command_t *cmd);
```

Listing 6: Command module exports

`command_add()` will add a given character to the buffer if it is not full. `command_reset()` will clear buffer and execution flag. `command_exec()` executes the buffered command and clears the execution flag afterwards. Note that the buffer is not cleared after execution, this way a buffered command can be executed multiple times.

The execution flag is used as indicator to show whether a command in the buffer should be executed or not. Note that buffered commands can be executed even if this flag is not set.

6.4.4 I2C Module (TWI)

```
1  void twi_init(void);
2
3  void twi_start(void);
4
5  void twi_stop(void);
6
7  bool twi_send_slave_address(uint8_t addr);
8
9  bool twi_send_byte(uint8_t data);
10
11 uint8_t twi_receive_byte_ack(void);
12
13 uint8_t twi_receive_byte_nack(void);
```

Listing 7: I2C module exports

The functions defined in this module help interacting with the I2C (TWI) peripheral. `twi_init()` will configure the I2C hardware to operate at 100 kHz. The required components are activated on demand since the prototype will operate as master. `twi_start()` and `twi_stop()` will issue start and stop conditions according to the protocol respectively. To communicate with a device, the master has to send the corresponding slave address with the R/W bit either set or cleared over the bus. This is handled by the `twi_send_slave_address()` function. After issuing the slave address the master can either send bytes to the slave with `twi_send_byte()` or read bytes from it using one of the following two functions. `twi_receive_byte_nack()` should be used when receiving only a single byte or the last in a sequence. Otherwise `twi_receive_byte_ack()` should be used.

6.4.5 Timer1 Module

In this application the Timer1 hardware part of the microcontroller will be used for the generation of a PWM signal. This signal is used to control the servo motor. [3, p. 109] holds a table that shows which bit in the TCCR1 register has to be set to select the desired mode. Mode 15 (a Fast PWM mode) has been selected.

The selected mode will start the Timer1 at value 0, it will then be incremented until its value matches the one set in register OCR1A. It is then reset to 0, but keeps running. Now to generate a PWM signal a value between 0 and OCR1A has been set to the OCR1B register. Upon reaching the value stated in OCR1B the corresponding output pin flips and generates the demanded PWM signal.

To operate the Timer1 in this mode, a clock source has to be set. In this application we use the system clock with a pre scale divider of 256. [3, p. 110]

Now we can use OCR1B to alter pulse width and OCR1A to change the frequency. The servo motor requires a PWM frequency of about 50 Hz, the required value for OCR1A can be calculated with the formula from the datasheet. [3, p. 101]

$$f_{PWM} = \frac{f_{CLK}}{N \cdot (1 + OCR1A)} \tag{3}$$

Where N is the pre scale divider, in this case 256. This can be written as

$$\text{OCR1A} = \frac{f_{CLK}}{f_{PWM} \cdot N} - 1 \tag{4}$$

The clock frequency for the ATmega32 is 16 MHz and the desired PWM frequency is 50 Hz.

$$\text{OCR1A} = \frac{16\text{MHz}}{50\text{Hz} \cdot 256} - 1 = 1249 \tag{5}$$

Useful values for OCR1B will be discovered by experimenting, hence values in this register will be related to the servo motors position. `timer1_set_ocr()` will be used to change this value.

6.4.6 Robot Module

The robot module contains high level functions to control the robot. These functions use the already described lower level functions as foundation.

```
1  void robot_set_velocity(int8_t left, int8_t right);
2
3  void robot_set_led(uint8_t red, uint8_t blue);
4
5  void robot_drive_request(int8_t distance);
6
7  void robot_turn_request(int8_t angle);
8
9  void robot_get_odometry();
10
11 void robot_set_odometry(int8_t xlow, int8_t xheigh, int8_t ylow, int8_t
12         yheigh, int8_t alphalow, int8_t alphaheigh);
13
14 void robot_get_sensor(void);
15
16 void robot_set_bar_level(uint8_t value);
17
18 void robot_custom(uint8_t *senddata, uint8_t sendsize, uint8_t readsize);
```

Listing 8: Robot module exports

`robot_set_velocity()` takes two *signed* 8 bit integer values as argument, the first one is used for the left motor, the second one for the right motor. The absolute value of this integer corresponds to the velocity the motor will run, the sign on the other hand states the direction. `robot_get_sensor()` requests sensor data from the control board. `robot_set_bar_level()` will alter the servo motors position hence changing the bar's elevation.

`robot_drive_request()`, `robot_turn_request()`, `robot_get_odometry()` and `robot_set_odometry()` are related to an odometry software module inside the control board.

The `robot_custom()` function can be used to issue a custom command over the I2C interface. `sendsize` bytes of the array `senddata` will be sent to the robot. If `readsize` is greater than 0, the communication board will read `readsize` bytes and print them like `robot_get_sensor()`.

6.4.7 Main

The complete main function can be viewed in listing 9. After initializing the other modules, the internal watchdog timer is armed. This timer has to be reseted periodically. Otherwise it will trigger a system reset. This technique is used to counter a hangup which might occur for example when a blocking function does not receive the requested amount of data, hence blocks forever. `sei()` and `cli()` enable and disable interrupts respectively. The infinite loop in main will periodically reset the watchdog timer and check if a command has to be executed. If execution of a command is demanded interrupts will be disabled temporarily.

```
1  int main(void) {
2      uart_init();
3      twi_init();
4      timer1_init();
5
6      command_t *cmd = command_default();
7
8      if(gBit(MCUCSR, 3)) {
9          log("watchdog reset");
10     }
11
12     wdt_enable(WDTO_2S);
13
14     log("init done");
15
16     sei();
17
18     while (true) {
19         wdt_reset();
20
21         // execute command
22         if (command_get_execution_flag(cmd)) {
23             cli();
24             if (!command_exec(cmd)) {
25                 log("command not found");
26             }
27             command_reset(cmd);
28             sei();
29         }
30     }
31
32     return 0;
33 }
```

Listing 9: Main function

6.5 Available commands

Which commands are available is defined by the `command_exec()` function. Each command has to be transmitted via the UART interface following a certain, very simple format. The first byte in a transmission is used to determine which command should be executed. Following bytes are handled as parameters according to the selected command. The transmission is terminated by an optional carriage return and a required line feed. Available commands are listed in table 4. Each parameter except `payload` occupies one byte.

This protocol can be extended easily, simply add more commands, but remember that the command buffer has a fixed size. It is save against buffer overflow when using the related functions but limits the amount of parameters a command can take. If you need a bigger buffer edit `COMMAND_BUFFER_LENGTH` in `config.h` accordingly.

6.6 Connecting the devices

A laptop has been used to connect to the prototype before an Android device is used. A simple terminal application (in this case minicom) should work just fine to test the communication. But first the 2 devices have to be connected.

The control board features a pin header for connecting the IOIO board. This header can be used to connect the prototype, although as can be seen in fig. 20 the interface is designed for 3.3 V but the prototype operates at 5 V. Normally the 3.3 V are provided by the IOIO board and are passed to the control board over a wire using the same pin header. Hence the according pin can be connected to a 5 V source so both sides of the level converter are operating at 5 V. Apart from the I2C wires ground must be connected. The prototype itself will be powered by the laptop in this case so there is no need to pass 5 V from the control board.

First Character	Parameters	Description
‘w’	*none*	Move forward
‘s’	*none*	Stop
‘a’	*none*	Turn left
‘d’	*none*	Turn right
‘x’	*none*	Move backward
‘-’	*none*	Lower bar a few degree
‘+’	*none*	Rise bar a few degree
‘r’	*none*	LEDs on
‘e’	*none*	LEDs off
‘q’	*none*	Read sensors
‘u’	red, blue	set LED brightness
‘i’	left, right	set motor velocity
‘o’	level	set bar elevation
‘h’	*none*	get odometry data
‘j’	x_{low}, x_{heigh}, y_{low}, y_{heigh}, α_{low}, α_{heigh}	set odometry
‘k’	distance	drive request (odometry)
‘l’	angel	turn request (odometry)
‘m’	sendsize, readsize, *payload*	custom I2C command

Table 4: Available commands

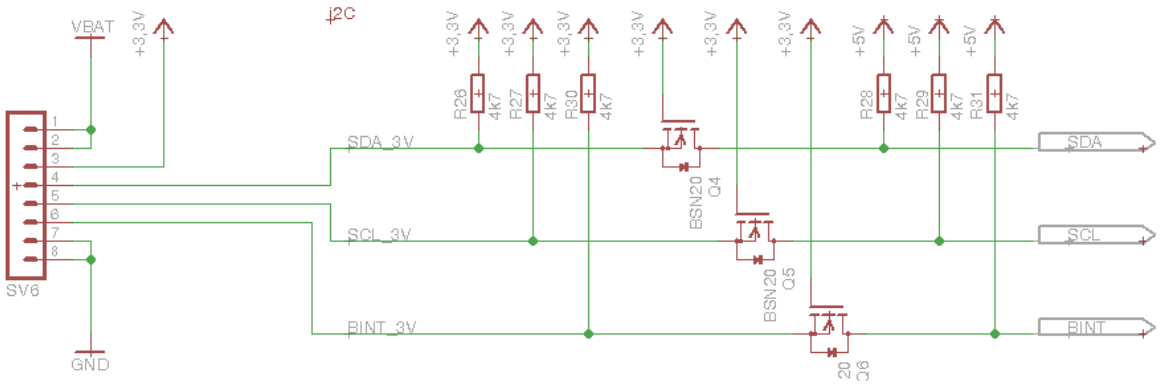


Figure 20: I2C bus schematic on the control board

As a side note, there was no need for a level converter in the first place. The I2C bus operates by using pull-up resistors to indicate a logical 1. Therefore all devices have to set their pins to a high resistive state (also called tristate) where the voltage level only raises via the pull-up resistors. Connecting the pull-ups to a 3.3 V source will allow device which operate at 3.3 V as well as device which operate at 5 V to access the bus. This is possible because 3.3 V will still be recognized as logical 1 by 5 V operating devices. And since these devices do not apply 5 V to the bus, they will not damage their neighbors.

After connecting everything the FT232 chip is recognized by the laptop and provides a serial (COM under windows) device which can be used to communicate with the prototype. Upon entering commands in a terminal the prototype receives the commands and forwards them accordingly. The robot starts moving when the forward command is issued, hence a successful communication has been achieved. Note that some terminals require a key combination like <CTRL>+j to send a line feed

character.

Next a Nexus 5 Android phone is connected to the prototype after disconnecting the laptop. The Nexus 5 is preferred to the Nexus 4 because the later one has some issues with USB OTG which is mandatory for this setup. For testing purposes a serial terminal[1] available from the Android Play store is used for communication. According to the description this application is compatible with the used FTDI chip. Similar to the setup using the laptop the phone is able to connect to the prototype, issue commands and hence control the robot.

This prototype may not seem as simple as the first one, but in comparison this one provides a much higher degree of extensibility. Now everything which supports UART can directly connect to the robot. Every phone which is capable of USB OTG can connect to the robot via a tiny converter. Phones which do not support OTG can still use Bluetooth. Because these communication channels do not rely on Android specific hardware / software they can be considered independent with respect to changes introduced by upcoming Android versions.

6.7 Final Board

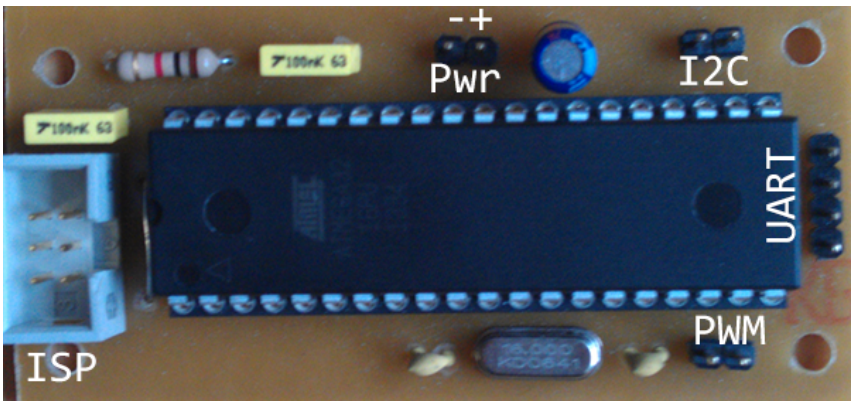


Figure 21: Final board - top view including annotations

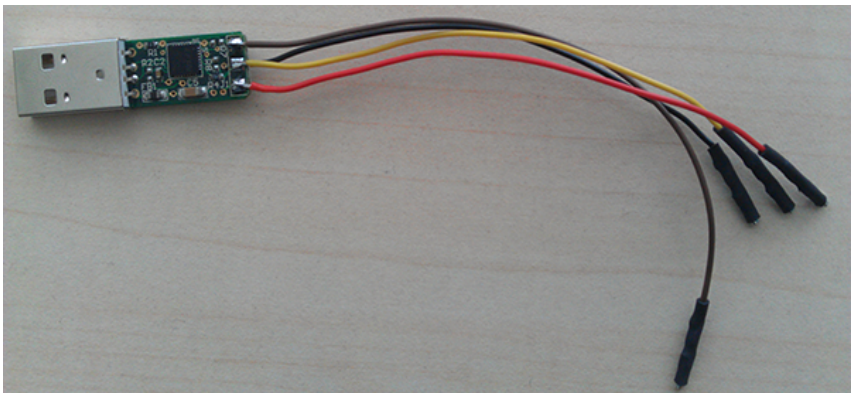


Figure 22: USB to serial 5 V converter - breakout board

[1] USB Serial Monitor Lite (https://play.google.com/store/apps/details?id=jp.ksksue.app.terminal)

The final board as can be seen in fig. 21 is very similar to the prototype, although has been simplified even further. Still an ATmega32 is used running the same firmware, but the FT232BL chip has been removed from the board to ensure a smaller footprint. Instead of the FT232BL chip, fully assembled breakout boards will be used, these breakout boards are much smaller and hold a FT232RQ which is very similar to the one used until now. These breakout boards are provided by FTDI and can be found under the designation `TTL-232R-PCB`. Figure 22 shows a picture of this breakout board which has already soldered some wires to it so it can be plugged into the communication board.

The layout of the board has been changes to match the IOIO boards exact footprint. Due to identical size and exact same drill hole diameter and positions the IOIO board can be replaced with the new communication board without touching the current mechanical setup of the robot. The pin headers, as can be viewed in fig. 23, show the amount of simplifications. The bigger ISP header has been replaced with the smaller ICSP header. There are now single connectors for the I2C bus, PWM output and power supply (last one is designated as `robot`). The UART header can be used to connect either the breakout board or a Bluetooth module.

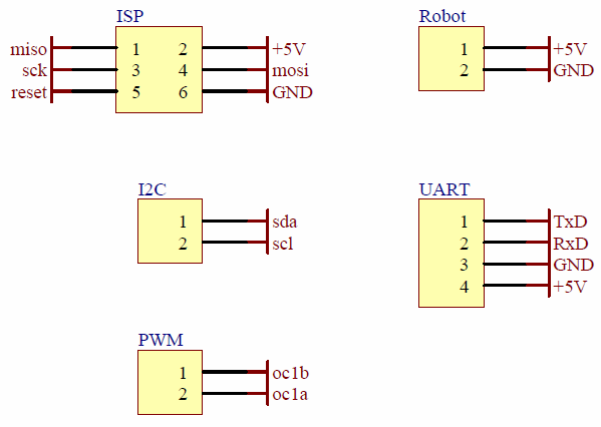


Figure 23: Final board headers

Chapter 7

RobotWASD App

Introduction

This chapter describes the Android app which has been created to provide easy access to the robot. After going through the code the Android manifest will be explained together with permission acquiring mechanism.

7.1 Requirements

This Android app has to serve as a simple example which can be used by students. The app shows them how to control the robot from an Android phone and provides basic functions which can be copied to their application. Error handling should be kept to a minimum since it only obfuscates the code.

Apart from providing clean example code, it should be easy to maintain and easy to extend.

7.2 FTDriver

The FTDriver[1] library makes it easy to connect the FT232 USB to serial convert to the Android phone. It provides a typical serial interface which can be used for byte oriented reading and writing. The project's README file contains some examples and should be considered if more information is needed. It is recommended to grab the latest version of the library using Git[2] for compatibility reasons.

After downloading the library has to be linked in Eclipse in the Java build path section of the actual app *as well as* inside the Android settings. The Android settings for the project are used to select which SDK version is used, and for linking other projects. The library can be opened by Eclipse as Android project and then be added as related project.

[1] see https://github.com/ksksue/FTDriver
[2] see http://git-scm.com/

Eclipse will have problems compiling the app if the library cannot be found in Java's build path. But if the library is not linked to the project the SDK tools will not add it to the `.apk` package. Hence the app will throw a `ClassNotFoundException` as soon as it is run.

7.3 App

For documentation purposes as well as to provide easy access to the new communication board, a small Android app has been created. This app can control the robot via the new hardware. But before details about the app can be explained, lets have a look at the following requirements. In the appendix details for setting up the required environment for usage and development are described.

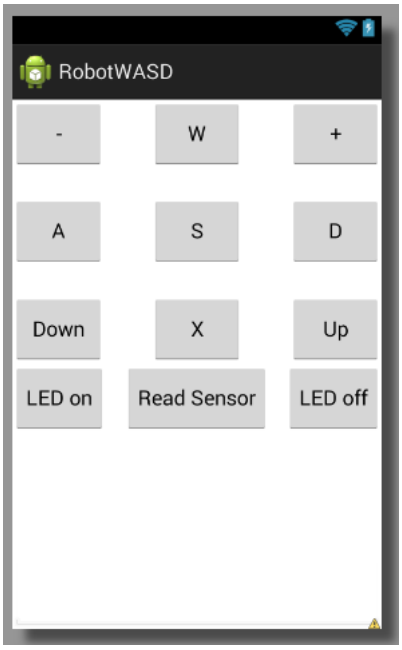


Figure 24: RobotWASD app screenshot

A screenshot can be observed in Figure 24. It might not be very stylish but it provides all things needed and does not obfuscate the code by applying a special design.

The view is divide into two segments. The upper segments holds multiple buttons which can be used to control the robot. Note that the buttons have been labeled similar to their related commands (table 4). The lower part contains a `textbox` which will show received data and can be used for logging purposes as well.

When the app is started an instance of the FTDriver class has to be created, this is done via Android's `UsbManager`. After creating this instance, a connection method is called which opens a connection with the desired baud rate as can be seen in listing 10.

A wrapper function has been created to prevent writing to a closed connection, see listing 11.

Due to multiple buffering the `read()` command has to be issued at least three times to ensure new data is read. Since this behavior might confuse some students, it is hidden inside a wrapper function, see listing 12.

```java
public class MainActivity extends Activity {
    private FTDriver com;
    private TextView textLog;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textLog = (TextView) findViewById(R.id.textLog);

        com = new FTDriver((UsbManager) getSystemService(USB_SERVICE));

        connect();
    }

    public void connect() {
        if (com.begin(9600)) {
            textLog.append("connected\n");
        } else {
            textLog.append("could not connect\n");
        }
    }
}
```

Listing 10: Instantiation of the FTDriver class

```java
    public void comWrite(byte[] data) {
        if (com.isConnected()) {
            com.write(data);
        } else {
            textLog.append("not connected\n");
        }
    }
```

Listing 11: wrapper for write method

Since it is quite common to issue a command and wait for the response a helper function has been created which writes data to the interfaces, waits a few milliseconds and then returns the received answer, see listing 13.

Building on top of these functions robot control becomes much easier. The three methods defined in listing 14 control the robot from a high level perspective.

Sensor data can be retrieved via the 'q' command. The `onClick` method for the `Sensor` button (listing 15) will issue this command. The returning string contains measurements of all sensors formated as space separated hex values, each prepended with `0x`.

7.4 Manifest and USB Permission

Each Android app has a `manifest` which contains meta data about itself. This manifest is written in XML[3] and contains important settings like the minimum supported SDK version or which permissions are required. [9, /guide/topics/manifest/manifest-intro.html].

The complete manifest can be observed in listing 16. Here the app is granted USB host permission via the `<uses-feature />` tag. But that is not enough. When using a USB device from an Android app the app needs special permissions to access the USB device. This goes beyond a simple `uses feature usb host` call. There are two ways how to obtain this special permissions. First the easy way like it is used in this app. By adding an `<intent-filter>` tag the manifest tells the Android OS when this app should be launched. Here the manifest states when a USB_DEVICE_ATTACHED event occurs and the connected device matches an entry inside `res/xml/device_filter.xml` the app should be launched. Listing 17 shows the content

[3] eXtensible Markup Language

```
1   public String comRead() {
2       String s = "";
3       int i = 0;
4       int n = 0;
5       while (i < 3 || n > 0) {
6           byte[] buffer = new byte[256];
7           n = com.read(buffer);
8           s += new String(buffer, 0, n);
9           i++;
10      }
11      return s;
12  }
```

Listing 12: wrapper for read method, hide multi buffering

```
1   public String comReadWrite(byte[] data) {
2       com.write(data);
3       try {
4           Thread.sleep(100);
5       } catch (InterruptedException e) {
6           // ignore
7       }
8       return comRead();
9   }
```

Listing 13: write then read helper

of this filter file. It contains an entry for the FT232 chip. Now the app will launch as soon as a FT232 chip is connected to the phone using an USB OTG cable. The Android OS grants the app permissions automatically. [9, /guide/topics/connectivity/usb/host.html#working-d]

The other way of obtaining the required permissions is by requesting them as they are needed. [9, /guide/topics/connectivity/usb/host.html#permission-d] shows an example of this use case including all information required.

```
    public void robotSetLeds(byte red, byte blue) {
        logText(comReadWrite(new byte[] { 'u', red, blue, '\r', '\n' }));
    }

    public void robotSetVelocity(byte left, byte right) {
        logText(comReadWrite(new byte[] { 'i', left, right, '\r', '\n' }));
    }

    public void robotSetBar(byte value) {
        logText(comReadWrite(new byte[] { 'o', value, '\r', '\n' }));
    }
```

Listing 14: methods for controlling the robot

```
    public void buttonSensor_onClick(View v) {
        logText(comReadWrite(new byte[] { 'q', '\r', '\n' }));
    }
```

Listing 15: methods for controlling the robot

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="at.ac.uibk.robotwasd"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="16"
        android:targetSdkVersion="19" />

    <uses-feature android:name="android.hardware.usb.host" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="at.ac.uibk.robotwasd.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />

                <action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED" />
            </intent-filter>

            <meta-data
                android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"
                android:resource="@xml/device_filter" />
        </activity>
    </application>
</manifest>
```

Listing 16: Android manifest of the RobotWASD app

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- 0x0403 / 0x6001: FTDI FT232R UART -->
    <usb-device vendor-id="1027" product-id="24577" />
</resources>
```

Listing 17: Android device filter for intent startup

Chapter 8

Conclusion

An open source microcontroller board named IOIO has been used for the communication process between the robot and an Android phone in the past. The multithreaded API and Android dependency has lead to various problems ranging from simple connection issues to race conditions between Android threads. A custom protocol has been used for communication over USB which made it difficult to simply replace the communication board.

The evaluation process shows two possible solutions which can be used to resolve problems in the described setup. The first prototype does not work fully as intended but was already an improvement to the IOIO board. Problems occurred when using the prototype with different Android versions. The second prototype performs much better and brings some new benefits along. The final solution is very close to this prototype. The software part is nearly identical only the hardware has been altered a little.

Apart from the firmware other software has been created. In order to provide easy access an Android application is provided which can control the robot using the new setup. Android application and firmware are designed in a simple and modular fashion so other people can build upon this setup.

Overall the project can be considered a success. All 16 robots have been fitted with a *final board* (section 6.7), replacing the IOIO board. The overall experience has been enhanced thanks to the much simpler communication platform.

But there are still issues left to solve, this project is just a single iteration in a longer sequence of improvements. Apart from mechanical tuning further steps could focus on combining the communication board and the control board. This would simplify the whole internal setup and would create enough room to place an embedded system on top of the robot.

Bibliography

[1] Arduino SA. Arduino webpage. http://arduino.cc, 2014. Retrieved April 15, 2014.

[2] Atmel. Avr910: In-system programming. http://www.atmel.com/Images/doc0943.pdf, 1997. Retrieved April 15, 2014.

[3] Atmel. Atmega32 datasheet, 2011. Retrieved April 15, 2014.

[4] Atmel. Atmega644 datasheet, 2011. Retrieved April 15, 2014.

[5] BeagleBoard.org Foundation. Beagleboard.org webpage. http://beagleboard.org, 2014. Retrieved April 15, 2014.

[6] Ytai Ben-Tsvi. Ioio wiki. https://github.com/ytai/ioio/wiki, 2013. Retrieved April 15, 2014.

[7] Future Technology Devices International Ltd. Ft232bl datasheet, 2011. Retrieved April 15, 2014.

[8] Future Technology Devices International Ltd. Ft311d datasheet, 2013. Retrieved April 15, 2014.

[9] Google. Android developer webpage. http://developer.android.com, 2014. Retrieved April 15, 2014.

[10] Joerg Wunsch. avrdude manpage, 2014. Retrieved April 15, 2014.

[11] Raspberry Pi Foundation. Raspberry pi webpage. http://www.raspberrypi.org, 2014. Retrieved April 15, 2014.

[12] SHARP. Gp2y0a21yk0f datasheet, 2007. Retrieved April 15, 2014.

[13] STMicroelectronics. Vnh3sp30-e datasheet, 2007. Retrieved April 15, 2014.

Appendix A

Additional Information

A.1 Assembly Files

Altium designer has been used to create the required schematics and PCB layouts in this project. All non-standard components have been created in the process and are stored in a library which is available in the `altium/lib` folder. All Altium projects used in this work use the non-standard components from this library.

A.2 AVR Flashing

After writing, compiling and linking a microcontroller program, the resulting binary file has to be flashed on the chip. A hardware programmer and a software utility are required for this task. One end of the hardware programmer can connect to your computer's USB or RS232 port the other end is connected to the microcontroller. The software utility is used to control the hardware programmer and feed it with compiled binary file.

A chip ISP programmer in combination with the `avrdude` utility has been used for programming the prototypes. When connecting the ISP programmer to the computer running Ubuntu via USB, a new device `/dev/ttyACM0` is available. `avrdude` accesses this device. It also needs to know which microcontroller will be programmed and which protocol is used for talking to the hardware programmer.

The most important parameter for calling `avrdude` is the `-U` flag. It tells `avrdude` which memory on the microcontroller should be read or written. A call for flashing the compiled binary file looks like this.

```
avrdude -p m32 -P /dev/ttyACM0 -c avrisp2 -B 10\
    -U flash:w:hex
```

`flash:w:hex` means **w**rite the content of the file named **hex** to the **flash** memory of the microcontroller.

More information about `avrdude` and its parameters can be retrieved from [10].

A.3 AVR Fuses

AVR microcontrollers feature 2 or 3 special registers which are used to configure general settings for the microcontroller. By setting these fuses one select the clock source for the microcontroller or disable programming entirely. A detailed description for the ATmega32 can be found in [3, p. 257], for the ATmega644 in [4, p. 285].

Following `avrdude` call was used for setting the fuses on the ATmega32.

```
avrdude -p m32 -P /dev/ttyACM0 -c avrisp2 -B 10\
    -U lfuse:w:0xbf:m -U hfuse:w:0x89:m
```

A.4 Setup Development Environment

Certain tools are required when developing an Android application, basically the Android SDK[1] covers everything needed, but an IDE[2] can help a lot and provide some comfortable shortcuts. For this application Eclipse has been chosen as IDE because Google recommended at the time development started. There is a plug-in available which will be used in combination with the SDK to move compiled application to the phone and debug them.

The starting point is a fresh installation of Ubuntu[3] 13.10 (64 bit). First we start by installing the Eclipse IDE using the system's package manager. Enter following command in a terminal.

```
sudo apt-get install eclipse
```

This will issue the package manager to download the eclipse package including all its dependencies and installing it on the system. After the command terminates eclipse should be installed and can be opened via the launcher.

As a side note, Google provides a complete package which includes the Eclipse IDE, Android SDK and the Android plug-in for Eclipse already setup. Though this might seem handy, the author does not recommend using it since the Eclipse version shipped with this package might not be fully compatible with your system. Where in contrast the package maintainers will provide you with an Eclipse version suitable for your system.

Next the Android plug-in can be added right away even though the SDK is not yet installed. Start Eclipse and select `Install New Software` from the menu bar's `Help` section. This will open a new window which enables you to install various Eclipse plug-ins from different sources. Now the source for the Android plug-in has to be provided. Click the `Add` button beside `Work with:` in the new window. Enter a sensible name and following link as location, then hit `Ok`.

`https://dl-ssl.google.com/android/eclipse/`

Now select the newly added entry and Eclipse will update the available package list. Check the `Developer Tools` group[4] and continue with the installation.

After the installation has been completed, the plug-in will complain about not finding the Android SDK. This will now be fixed. Go to [9, /sdk/index.html] and download the SDK `Tools` from the *SDK*

[1] Software Development Kit
[2] Integrated Development Environment
[3] `http://www.ubuntu.com/`
[4] check `Group items by category` if the named group does not show up

Tools Only section. Of course chose the package for Linux (Platform).

This package contains a tool which is used to grab the desired Android SDK version from a Google server. But since the binaries are only available for a 32 bit architecture we have to install some additional libraries (multilibs) to use them with the 64 bit kernel. Extract and run this tool afterwards. These commands with fulfil the described tasks.

```
sudo apt-get install libc6-i386 lib32stdc++6 \
    lib32gcc1 lib32ncurses5

tar xzvf android-sdk_*-linux.tgz
android-sdk-linux/tools/android
```

A new window opens which holds a listing of released Android SDK versions. Tick the checkbox beside the desired SDK version. If unsure check the version installed on your Android phone. Therefore enter the `Settings` menu on the phone and scroll down to the `System` section. Tap `About phone` and observe the Android version bullet. Continue by installing the ticked SDK. You can also installed multiple SDK versions side by side, this can be useful when handling multiple phones.

All new files will be installed inside the `android-sdk-linux` directory. Most tools which require access to the Android SDK will only need to know the path to this directory. This is also true for Eclipse. You can state the Android SDK path inside Eclipse's settings. Open `Preferences` available from the menu bar's `Window` section. On the left hand side an entry should be present named `Android`. Click this entry, now you can state the Android SDK path on the right hand side of the window, installed SDK versions will be listed below.

The most important tool contained in the Android SDK is the `adb`[5] binary. This tool starts a daemon which your phone can connect to via USB. Furthermore apps can pushed onto the phone and can be debugged via `adb`. This tool sits in `android-sdk-linux/platform-tools/`. Make sure to start the server with root privileges, this can be achieved with the following command. There is also a `kill-server` option to stop an already running server.

```
sudo android-sdk-linux/platform-tools/adb start-server
```

If you are not comfortable with running this tool under root privileges, you can run it as normal user, but you have to add special rules to `udev` otherwise the tool is not allowed to access the USB device by your system. Listing 18 shows an example.

```
# HTC Desire HD
SUBSYSTEM=="usb", ATTR{idVendor}=="0bb4", MODE="0666"
SUBSYSTEM=="usb", ATTR{idVendor}=="0bb4", ATTR{idProduct}=="0ca2", \
SYMLINK+="android_adb"
SUBSYSTEM=="usb", ATTR{idVendor}=="0bb4", ATTR{idProduct}=="0fff", \
SYMLINK+="android_fastboot"
```

Listing 18: udev example rules (`/etc/udev/rules.d/51-android.rules`) (lines have been wrapped for printing)

[5] Android Debug Bridge

A.5 Setup Android Phone

In order to use the Android phone for development the `Developer options` have to be activated. Go to the `About phone` inside the `Settings` menu. Scroll down until you see the entry named `Build number` and tap this entry multiple times (about 20 times or so) until a message appears notifying you that developer options have been enabled.

A new entry named `Developer options` is now available inside the `Settings` menu. Inside this new entry you can choose to enable `Android debugging`. This is necessary to connect the Android phone to your system, hence enable this option.

Now connect the phone to your system via USB and a message should appear notifying you that debugging has been enabled. You are also required to grant the connected computer access to the phone.

The phone should now be visible when using the `adb` tool. Following command will list connected devices. You do not have to run the tool with root privileges when the server is already running as root. If the server is not running it will be started as soon as `adb` is used for phone related tasks.

```
android-sdk-linux/platform-tools/adb devices
```

A.6 Python Example

```
1  #!/usr/bin/env python
2
3  import serial
4
5  s = serial.Serial('/dev/ttyUSB0', 9600)
6
7  while True:
8      s.write("q\r\n")
9      s.readline()
10     print s.readline(),
11
12 s.close()
```

For the sake of debugging a small python script has been created and stored inside the `utils` folder. This script opens a serial device present at `/dev/ttyUSB0` and issues the `sensor` command in an infinite loop. This script can be used for debugging the sensors and shows how simple commands can be issued from a python script.

A.7 OpenCV Example

The OpenCV library provides useful tools for machine learning and image processing. It is available for Android and comes with some examples. Using the robot control functions defined in the Robot-WASD app together with the image processing API from the OpenCV library it should be easy to setup a autonomous unit.

A.8 Nexus 4 enable OTG

Not all Android phones feature USB OTG. There are phones using a USB chip which is not capable of switching to USB host mode, others simply got this feature disabled by their firmware for not fully meeting the required specifications.

The Nexus 4's USB chip is capable of switching to USB host mode, but because the phone lacks the ability to provided power this feature has been disabled. But there is a patch available on the xda-devleopers forum[6]. Just follow this post, it will explain everything needed to get the Nexus 4 working with OTG.

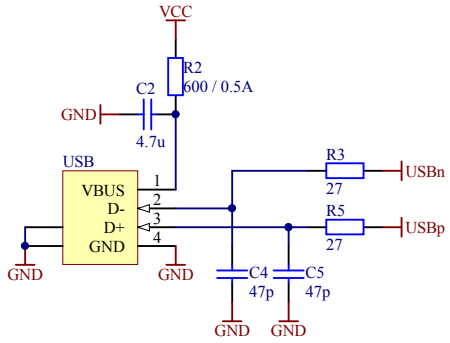
Because the USB sense pin is not properly wired up to the USB chip, applying 5 V to the USB V_{CC} pin will trigger the switching to USB host mode. Your phone will drain power like it does when it is getting charged. See the `Bugs / Notes` section in the mentioned post. It contains a system call which can be used to disable charging temporarily.

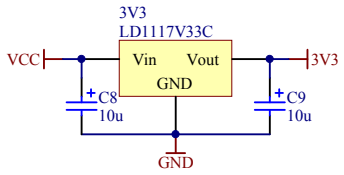
[6] http://forum.xda-developers.com/nexus-4/orig-development/usb-otg-externally-powered-usb-otg

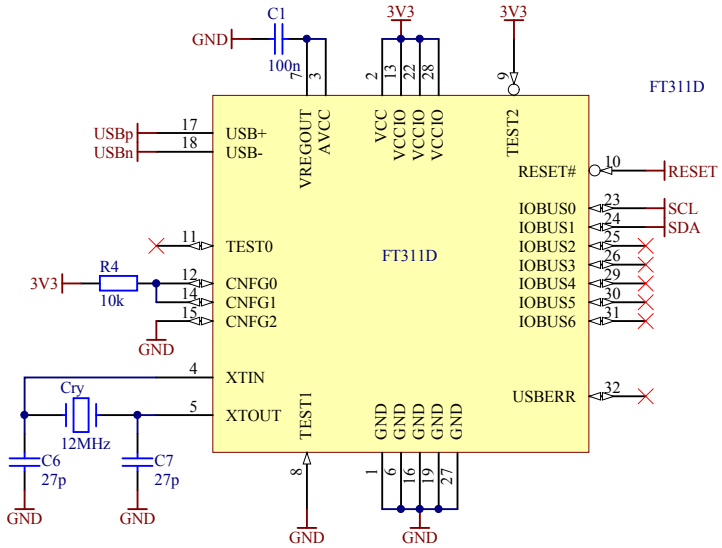
Appendix B

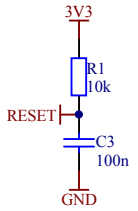
Assembly Drawings

B.1 FT311D in I2C Configuration

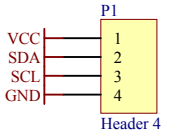




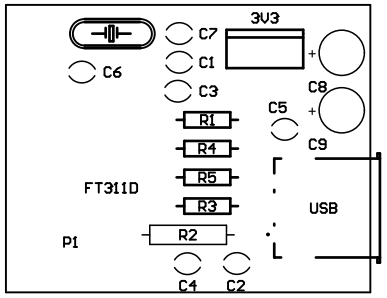




no pull-ups provided



Title: FT311D I2C configuration		
Size: A4	Number:	Revision:
Date: 2014-05-06		Sheet of
File: \\..\usb_i2c.SchDoc		Drawn By: Alexander Hirsch

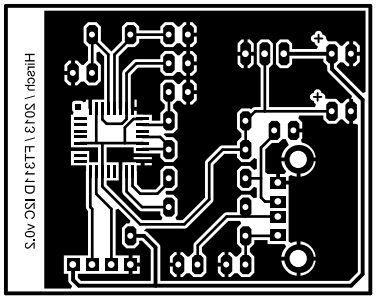


TITLE:

FT311D I2C configuration

AUTHOR:

Alexander Hirsch

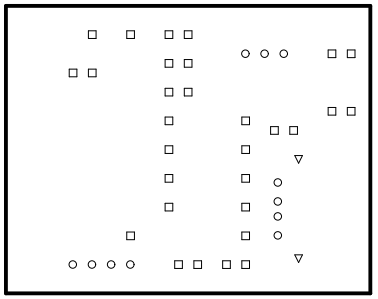


TITLE:

FT311D I2C configuration

AUTHOR:

Alexander Hirsch



Symbol	Hit Count	Tool Size	Plated	Hole Type
□	30	31.496mil (0.8mm)	PTH	Round
○	11	35.433mil (0.9mm)	PTH	Round
▽	2	90.551mil (2.3mm)	PTH	Round
	43 Total			

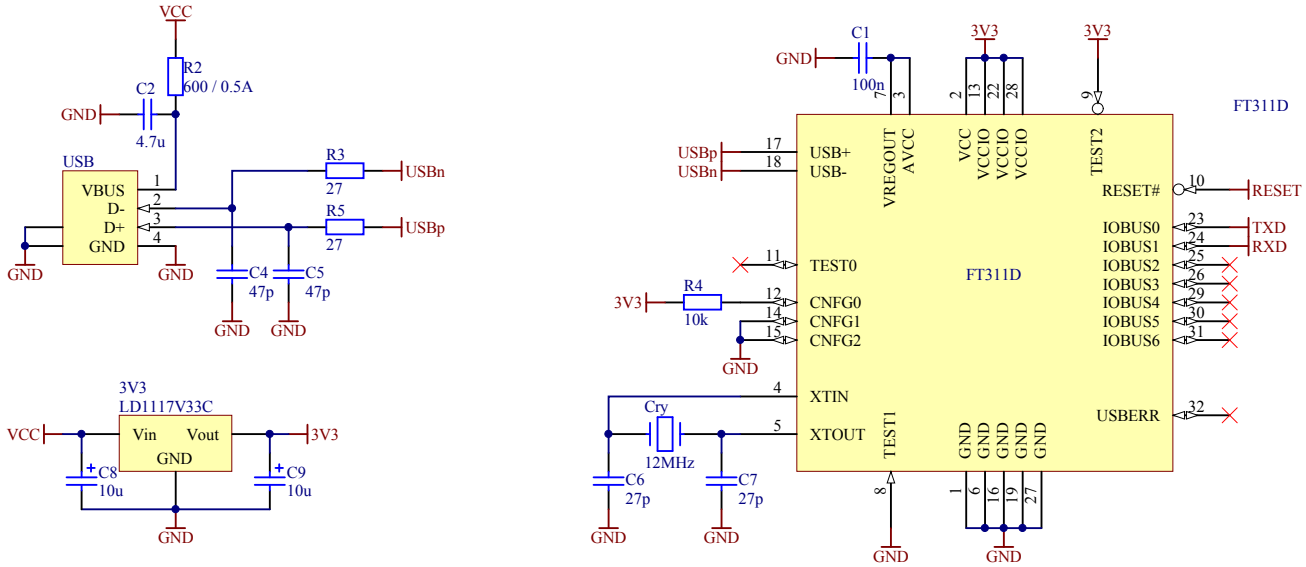
TITLE:

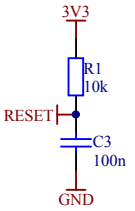
FT311D I2C configuration

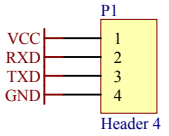
AUTHOR:

Alexander Hirsch

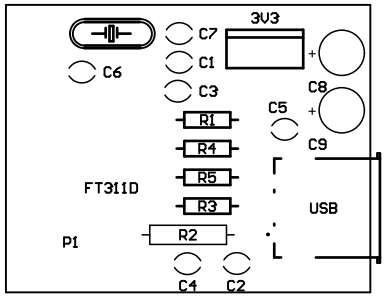
B.2 FT311D in UART Configuration







Title: FT311D + UART configuration		
Size: A4	Number:	Revision:
Date: 2014-05-06		Sheet of
File: \\..\usb_uart.SchDoc		Drawn By: Alexander Hirsch

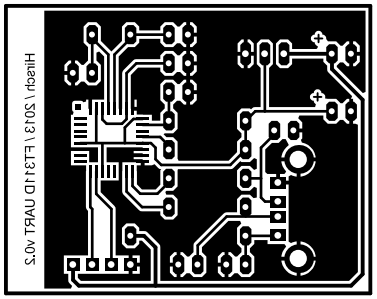


TITLE:

FT311D UART configuration

AUTHOR:

Alexander Hirsch

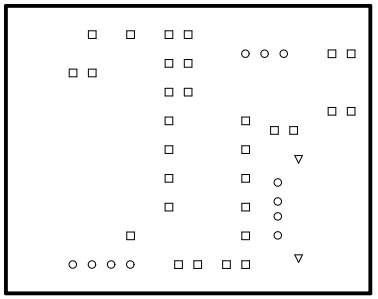


TITLE:

FT311D UART configuration

AUTHOR:

Alexander Hirsch



Symbol	Hit Count	Tool Size	Plated	Hole Type
□	30	31.496mil (0.8mm)	PTH	Round
○	11	35.433mil (0.9mm)	PTH	Round
▽	2	90.551mil (2.3mm)	PTH	Round
	43 Total			

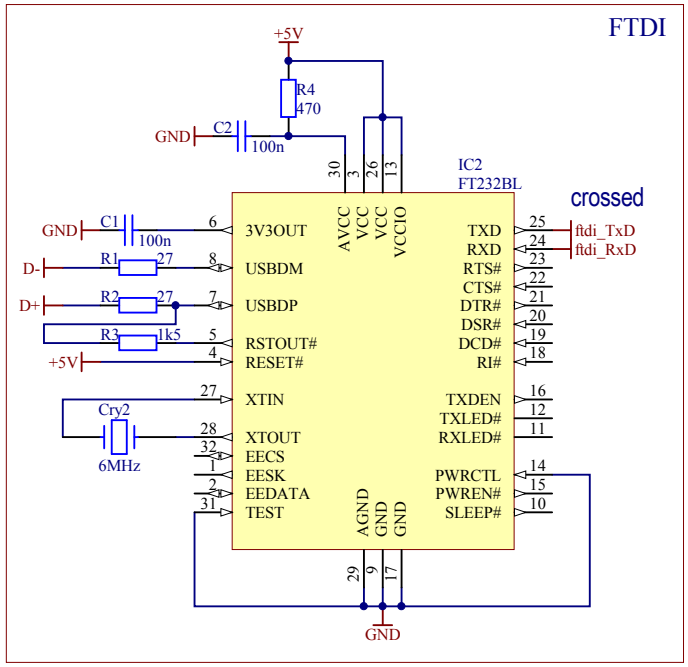
TITLE:

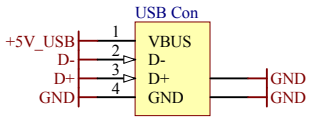
FT311D UART configuration

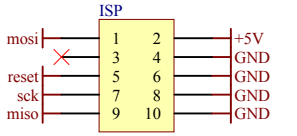
AUTHOR:

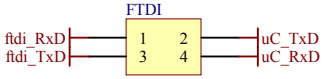
Alexander Hirsch

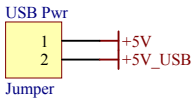
B.3 ATmega32 with FT232BL

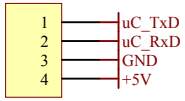


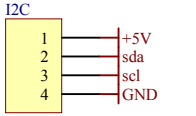


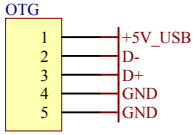


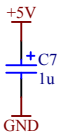


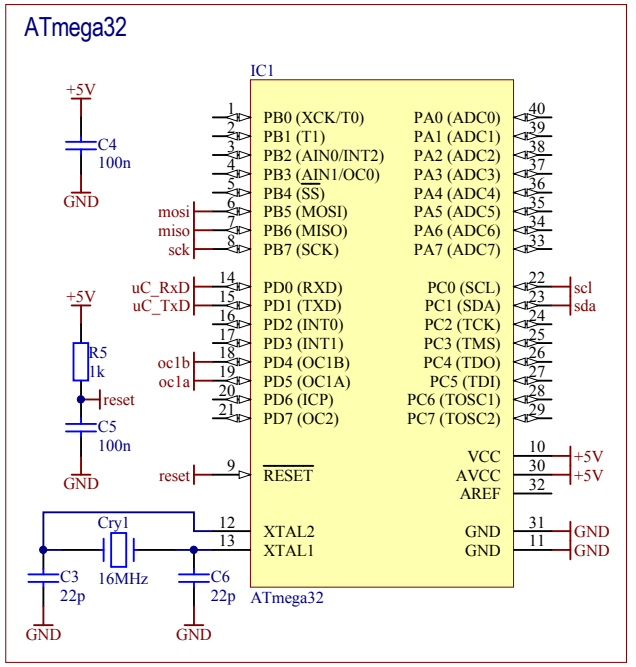




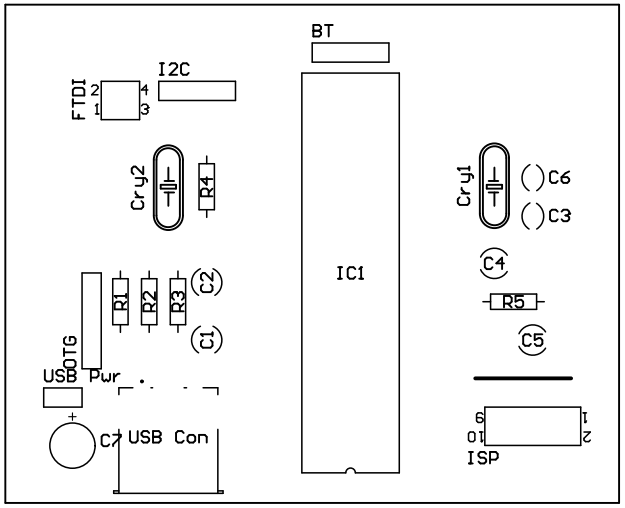








Title	ATmega32 + FT232BL	
Size: A4	Number:	Revision:
Date: 2014-05-06		Sheet of
File: \\..\main.SchDoc		Drawn By: Alexander Hirsch

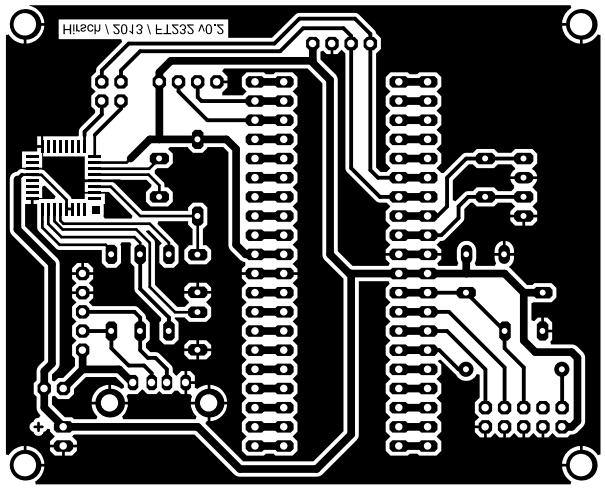


TITLE:

ATmega32 + FT232BL

AUTHOR:

Alexander Hirsch

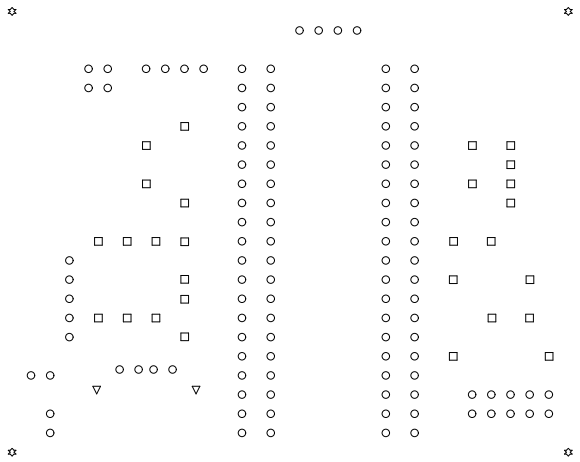


TITLE:

ATmega32 + FT232BL

AUTHOR:

Alexander Hirsch



Symbol	Hit Count	Tool Size	Plated	Hole Type
□	28	31.496mil (0.8mm)	PTH	Round
○	115	39.37mil (1mm)	PTH	Round
▽	2	90.551mil (2.3mm)	PTH	Round
☼	4	118.11mil (3mm)	PTH	Round
	149 Total			

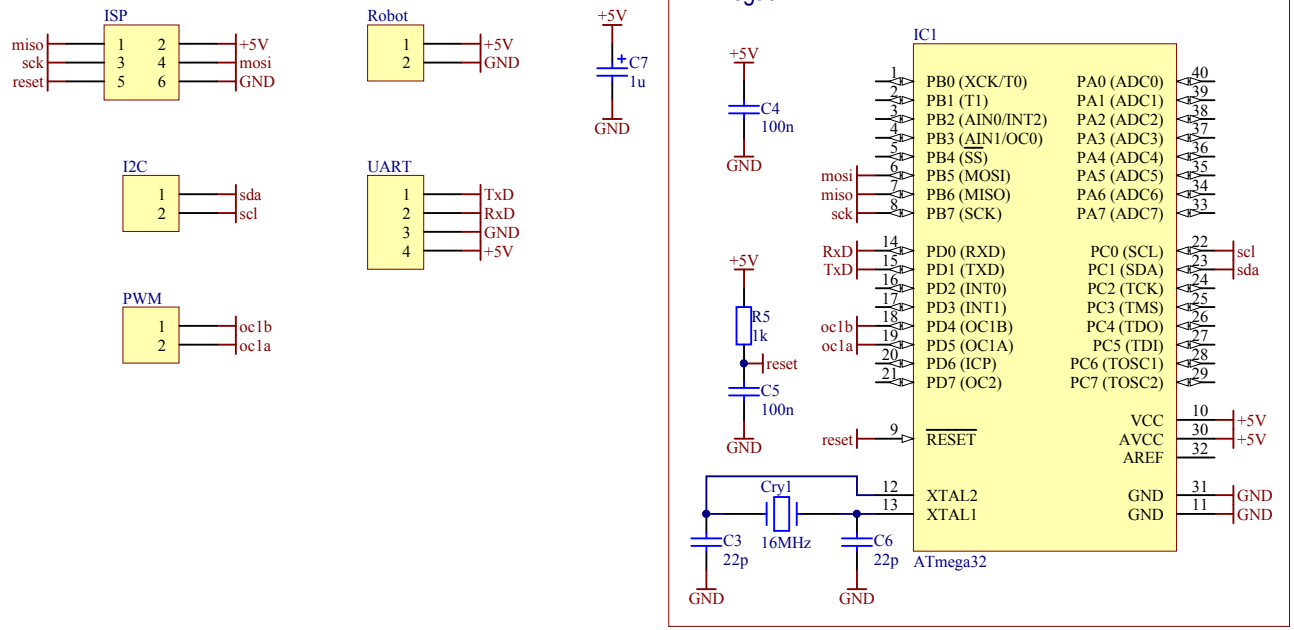
TITLE:

ATmega32 + FT232BL

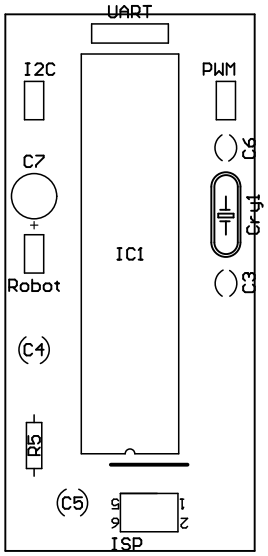
AUTHOR:

Alexander Hirsch

B.4 ATmega32 Final Board



Title	ATmega32 final	
Size: A4	Number:	Revision:
Date: 2014-05-06		Sheet of
File: \\..\main.SchDoc		Drawn By: Alexander Hirsch

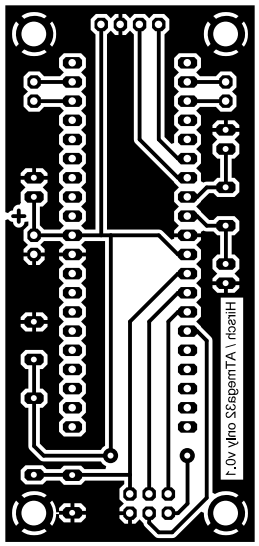


TITLE:

ATmega32 final

AUTHOR:

Alexander Hirsch

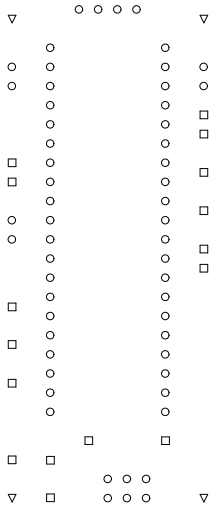


TITLE:

ATmega32 final

AUTHOR:

Alexander Hirsch



Symbol	Hit Count	Tool Size	Plated	Hole Type
□	16	31.496mil (0.8mm)	PTH	Round
○	56	39.37mil (1mm)	PTH	Round
▽	4	129.921mil (3.3mm)	PTH	Round
	76 Total			

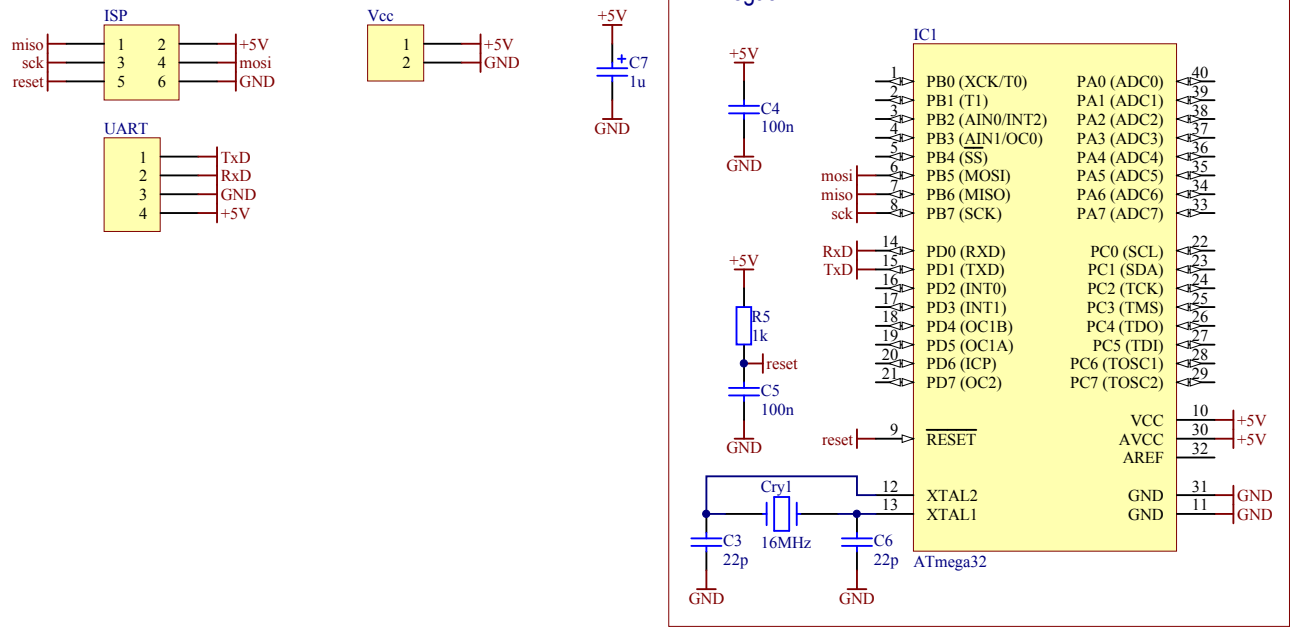
TITLE:

ATmega32 final

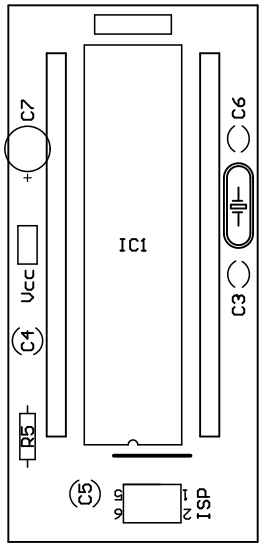
AUTHOR:

Alexander Hirsch

B.5 ATmega32 Final Board with Pin Headers



Title	ATmega32 final (full pin headers)	
Size: A4	Number:	Revision:
Date: 2014-05-06		Sheet of
File: \\..\main.SchDoc		Drawn By: Alexander Hirsch

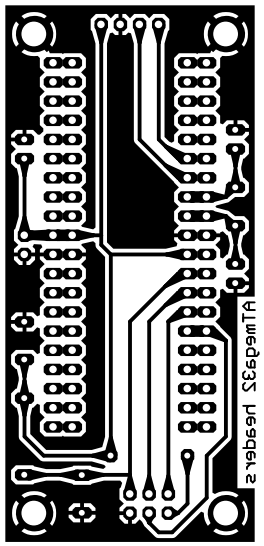


TITLE:

ATmega32 final headers

AUTHOR:

Alexander Hirsch

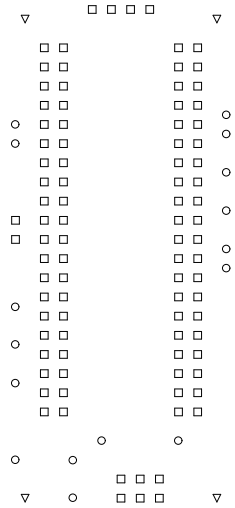


TITLE:

ATmega32 final headers

AUTHOR:

Alexander Hirsch



Symbol	Hit Count	Tool Size	Plated	Hole Type
○	16	31.496mil (0.8mm)	PTH	Round
□	92	39.37mil (1mm)	PTH	Round
▽	4	129.921mil (3.3mm)	PTH	Round
	112 Total			

TITLE:

ATmega32 final headers

AUTHOR:

Alexander Hirsch