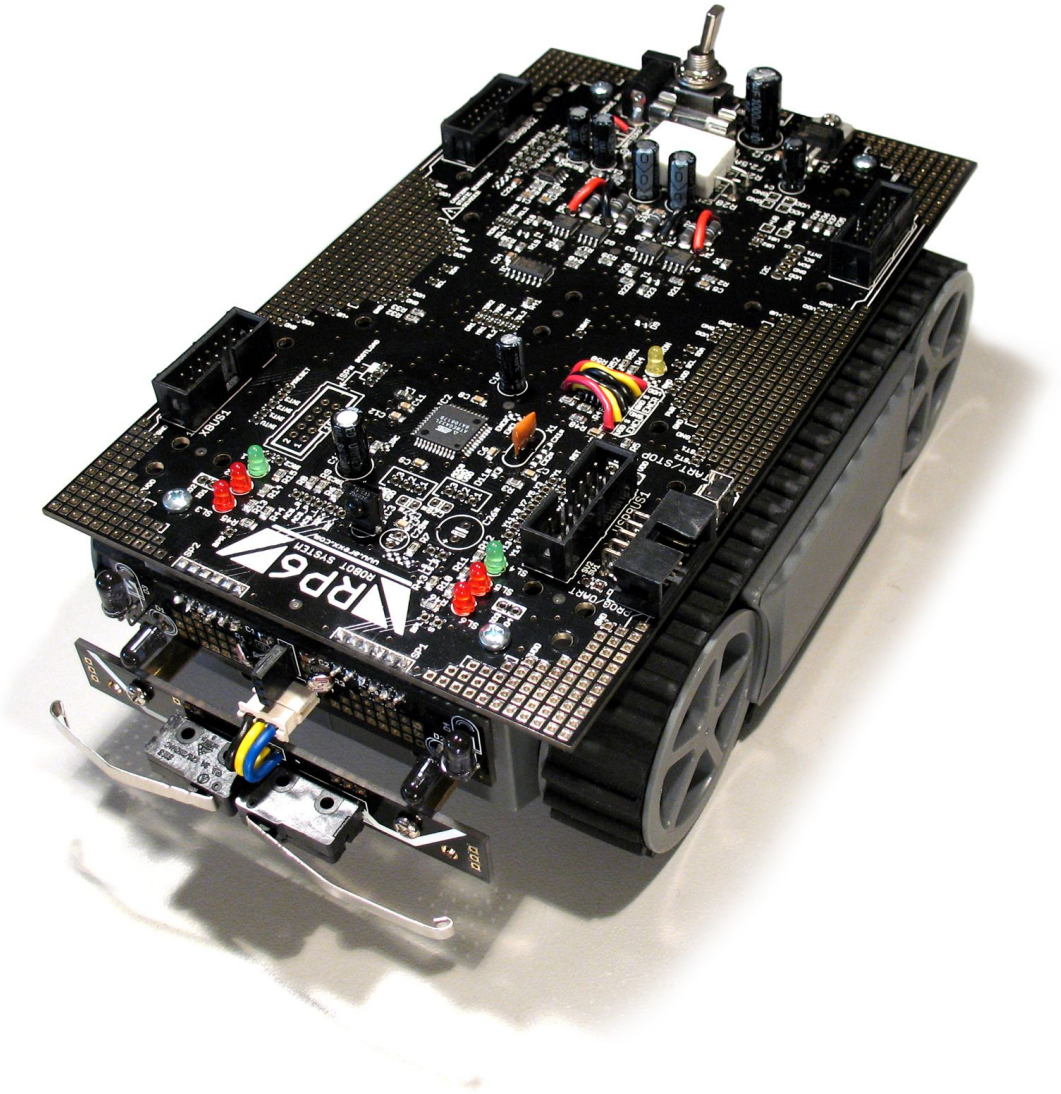
RP6 ROBOT SYSTEM

RP6 ROBOT BASE



AREXX Engineering

RP6-BASE

www.arexx.com

RP6
Robot System

Manual

- English -

Version RP6-BASE-EN-20071029

PRELIMINARY VERSION

IMPORTANT INFORMATION!
Please read carefully!

Before you start operating the RP6 or any of the additional equipment, you must read this manual and the manuals for add-on modules completely! The documentation contains information how to operate the systems properly and how to avoid dangerous situations! Furthermore the manuals provide important details, which may be unknown to average users.
Paying no attention to this manual will cause a loss of warranty! Additionally, AREXX Engineering cannot be made responsible for any damages caused by neglecting the manual's instructions!

Please pay special attention to the chapter "Safety instructions"!

Do not connect the USB Interface to your PC before you have read chapter 3 – "Hardware and Software Setup" and completed the software installation!

Nervistraat 16
8013 RS Zwolle
The Netherlands

Tel.: +31 (0) 38 454 2028
Fax.: +31 (0) 38 452 4482

Limitations in Warranty and Liability

The warranty of AREXX Engineering is limited exclusively to the exchange of devices within legal warranty periods in case of hardware defects, such as mechanical damage, missing or wrong assembly of electronic components, excluding socketed circuits. To the extent permitted by applicable law, AREXX Engineering assumes no liability for any damage resulting directly or indirectly from the use of the device.

Irreversible modifications (e.g. soldering additional components, drilling holes, etc.) or damaging the devices by neglecting the instructions in this manual will void warranty.

No warranties can be given with respect to individual requirements to the included software, nor to the error-free and uninterrupted operation of the software.
Additionally, the software may be modified and loaded onto the system by the user. Therefore the user is responsible for software quality and the overall system performance of the robot.

AREXX Engineering guarantees the functionality of supplied example software as long as the specified operating conditions are respected. If the devices are operated beyond these conditions and the device or the PC-software is malfunctioning or gets defective, the customer will be charged for all service costs, repairs and corrections. Please also pay attention to the corresponding license agreements on the CD-ROM!

Symbols

The following Symbols are used in this manual:

The "Attention!" Symbol is used to mark important details. Neglecting these instructions may damage or destroy the robot and/or additional components and you may risk your own or others health!

The "Information" Symbol is used to mark useful tips and tricks or background information. In this case the information is to be considered as "useful, but not necessary".

Contents

1. Introduction

The RP6 is a low cost autonomous mobile robot system, designed for beginners as well as experienced electronics and software developers as an introduction to the fascinating world of robotics.

The robot is delivered completely assembled. Thus it is very well suited for all users who are unexperienced with soldering and tinkering and want to concentrate on software development. However, implementing your own circuits and adding additional things to the robot is possible, too! In fact, the RP6 offers a lot of expansion possibilities and may be used as a platform for a variety of interesting electronic experiments!

It is the successor of the very successful "C-Control Robby RP5", which had been released in 2003 by Conrad Electronic SE. The shortcut "RP5" is to be interpreted as "Robot Project 5". The new robot and the predecessor system do not have too much in common except for the mechanics. The C-Control 1 Microcontroller of Conrad Electronic has been replaced and thus the new robot cannot be programmed in Basic anymore. Instead, the far more powerful ATMEGA32 from Atmel, which is programmable in C is used. Additionally we plan to provide an expansion module for adapting the latest C-Control modules (e.g. CC-PRO MEGA128) to the robot. This module will allow the system to be programmed in the more simple Basic language and provides a great number of additional interfaces and lots of additional memory.

The new design includes an USB interface, a new expansion system with improved assembly options, high resolution odometry sensors (resolution is 150x higher compared to the predecessor system), a precise voltage regulator (this was only provided as an expansion module for the old system), a bumper composed of two microswitches with long levers and many other things. The already included experiment expansion module for your own circuits is also a very useful addition. Compared to the predecessor system, the price performance ratio has been improved considerably.

Basically the mechanical design has been adopted from the RP5 system. However, we did optimize the design for lower noise operation and now it provides some additional drilling holes for mechanical expansions.

The RP6 robot has been designed to be compatible with our other robots, ASURO and YETI, both using the smaller ATMEGA8 and identical development tools (WinAVR, avr-gcc). In contrast, ASURO and YETI are delivered as do-it-yourself construction kits and have to be assembled by the user. The RP6 has been designed for the more demanding users, looking for good expansion options, bigger microcontrollers and more sensors.

Several expansion modules are planned or already available and can be used for expanding the robot's capabilities. For example, there will be the previously mentioned C-Control expansion, an expansion module providing an extra MEGA32 (already available) and of course the experiment expansion board for individual electronic circuitry, which is also available seperately. You can stack several of these modules onto the robot.

Other interesting modules are planned shortly and of course you may develop your own expansion circuitry!

We wish you a lot of fun and success with your RP6 Robot System!

1.1. Technical support

You may contact our support team via internet as follows (**please read this manual completely before contacting the support!** Reading the manual carefully will answer most of your possible questions already! Please also read appendix A – Troubleshooting):

- through our forum: `http://www.arexx.com/forum/`

- by E-Mail: `info@arexx.nl`

You will find our postal address in the legal notice at the beginning of this manual. All ***software updates, new versions of this manual*** and further informations will be published on our homepage:

`http://www.arexx.com/`

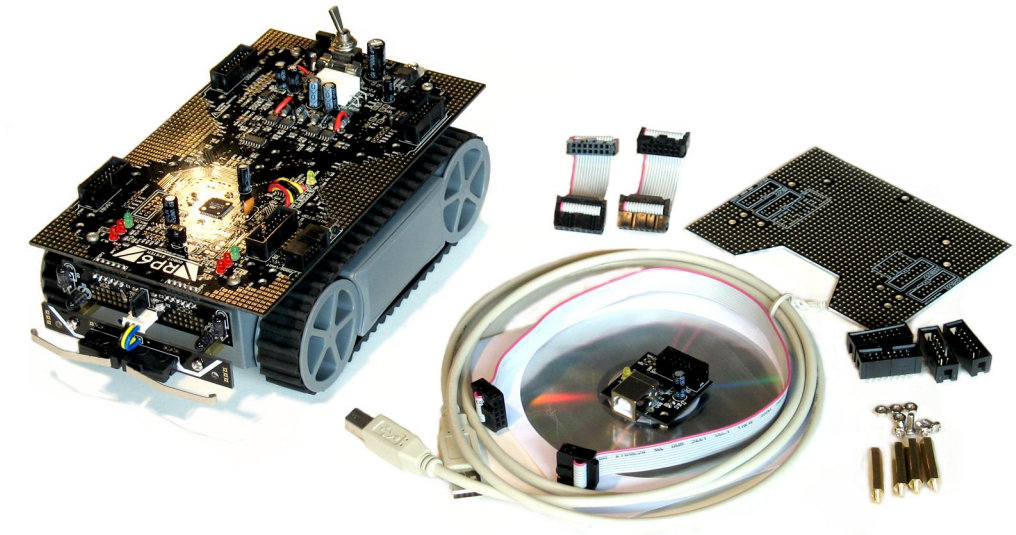
and on the robot's homepage:

`http://www.arexx.com/rp6`

1.2. Scope of delivery

You should find the following items in your RP6 box:

- **Completely assembled Robot**
- RP6 USB Interface
- USB A->B cable
- 10pin flat cable
- RP6 CD-ROM
- Quickstart manual
- **RP6 Experiment board**
- 4 pcs 25mm M3 distance bolts
- 4 pcs M3 screws
- 4 pcs M3 nuts
- 4 pcs 14pin connectors
- 2 pcs 14pin flat cable



1.3. Features and technical Data

This section provides an overview of the robot's features and an introduction of some basic keywords, to make you familiar with the terminology used in this manual. Most of these keywords will be explained in later chapters.

Features, components and technical data of the RP6 ROBOT SYSTEM:

- **Powerful Atmel ATMEGA32 8-Bit Microcontroller**
  - ◊ Speed 8 MIPS (=8 Million Instructions per Second) at 8MHz clock frequency
  - ◊ Memory: 32KB Flash ROM, 2KB SRAM, 1KB EEPROM
  - ◊ Freely programmable in C (using WinAVR / avr-gcc)!
  - ◊ ... and many more features! Further details will be given in chapter 2.
- **Flexible expansion system, based on the I²C-Bus**
  - ◊ Only two signals required (TWI -> "Two Wire Interface")
  - ◊ Transfer speed up to 400kBit/s
  - ◊ Master->Slave architecture
  - ◊ Up to 127 Slaves may be connected to the bus simultaneously
  - ◊ Very popular bus-system. The market provides a lot of standard ICs, sensors and other components, which may often be connected directly.
- **Symmetrical mounting possibilities for expansion modules at front and rear**
  - ◊ Theoretically you may stack any number of expansion modules, but the supply capability and the overall weight allows a maximum of about 6 to 8 modules (3 to 4 modules each at front and rear).
  - ◊ The mainboard provides 22 free 3.2mm mounting holes and the chassis provides another 16, summing up to 38 mounting holes in total – additionally the chassis provides ample space for individual drills.
- **Experiment PCB already included in delivery** (see scope of delivery photo)
- **USB PC Interface** for program uploads from PC to microcontroller
  - ◊ Wired connection for maximum transfer speed. Program upload will usually run at 500kBaud, filling the total free memory space (30KB, 2KB are reserved for the Bootloader) within seconds.
  - ◊ The interface may be used for programming all available expansion modules for the RP6 with AVR Microcontrollers.
  - ◊ It may be used for communication between the robot and expansion modules. For example you can use this for debugging purposes by transferring measurement data, text messages and other data to the PC.
  - ◊ The interface driver provides a virtual comport (VCP) for all popular operating systems including Windows 2K/XP/Vista and Linux. The VCP can be used in standard terminal programs and customized software.

◊ The **RP6Loader** Software for Windows and Linux allows comfortable program uploads. It also contains a small terminal for communicating with the robot through text messages.

- **Powerful caterpillar drive unit** in combination with a new gearing system for minimising noise *(compared to the predecessor CCRP5...)*
  - ◊ Two powerful 7.2V DC-Motors
  - ◊ Maximum speed ca. 25 cm/s – depending on charge state and quality of batteries, total weight and other conditions!
  - ◊ Self-lubing, sintered bearings at all four 4mm wheel-axles
  - ◊ Two rubber tracks
  - ◊ Capable of traversing small obstacles (up to ca. 2 cm height) like carpet edges, thresholds or ramps of up to 30% steepness (with mounted bumper switches). Removing the bumpers and restricting the number of modules to a maximum of 2 modules allows the robot to drive over ramps with up to 40% steepness.
- **Two powerful MOSFET Motor-drivers** (H-Bridges)
  - ◊ Rotational velocity and direction can be controlled by the Microcontroller system.
  - ◊ **Two current sensors** providing a measurement range up to ca. 1.8A for each motor. This allows to quickly sense blocked or heavily loaded motors.
- **Two high resolution encoders** for speed- and motion-control
  - ◊ Resolution **625 CPR** ("Counts Per Revolution") which implies the system counts 625 segments of the codewheel per revolution of a wheel! (150x higher resolution compared to the predecessor system CCRP5 with only ca. 4 CPR).
  - ◊ Exact and fast speed measurement and control!
  - ◊ High resolution of ca. 0.25mm per counted segment!
- **Anti-collision-system (ACS)** which can detect obstacles with an integrated IR receiver and two IR diodes aligned to left and right
  - ◊ Detects obstacles in the middle, left or right of the robot's front.
  - ◊ Sensitivity and transmitter power are adjustable, allowing reliable detection of badly reflecting objects.
- **Infrared Communication-system (IRCOMM)**
  - ◊ Receives signals of standard universal infrared Remote Controls of TVs or Video recoders. You may control your robot with a standard (RC5-) remote control! The protocol may be changed in software, but we provide only an implementation of the standard RC5-protocoll by default.
  - ◊ May be used for communication with several robots (using direct line of sight or reflections from the ceiling and walls) or for transmitting telemetry data.
- **Two light sensors** – e.g. for light intensity measurement and light source tracking
- **Two bumper sensors** for collision detection
- **6 Status LED's** – for sensor and program status displays
  - ◊ Four LED Ports can also be used for other functions if necessary!

- **Two free Analogue/Digital Converter (ADC)** Channels for external sensor systems (Alternatively available as standard I/O Pins).
- **Accurate 5V voltage regulation**
  - ◊ Maximum current supply: 1.5A
  - ◊ Large copper-area for heat dissipation to the PCB
  - ◊ Constant current should not get higher than 1A. More than this requires extra cooling! We recommend a maximum constant current value below 800mA.
- Replaceable **2.5A fuse**
- **Low standby current** of less than 5mA (4mA typ. and ca. 17 up to 40mA in use, of course this depends on system load and activity (LEDs, Sensors etc.). These values only include electronic circuits and do not take motors and expansion modules into account!).
- **Power supply with 6 NiMH Mignon accumulator batteries** (not included!)
  - ◊ E.g. Panasonic or Sanyo (NiMH, 1.2V, 2500mAh, HR-3U , Size AA HR6) or Energizer (NiMH, 1.2V, 2500mAh, NH15-AA)
  - ◊ Operating time ca. 3 up to 6 hours depending on usage and quality/capacity of the batteries (if the engines are not being activated too much, the robot may be operated a lot longer. These operating time specifications are only for the robot system itself, without expansion modules).
- **Connector for external battery chargers** – the robot's main power switch toggles between "Charge/Off" and "Operate/On".
  - ◊ This may be adapted by using a few solder pads on the PCB, allowing you to connect the robot with external power supplies or additional batteries.
  - ◊ Any external chargers that are suitable of charging 6 NiMH Cells in series may be used. External chargers drastically vary in performance and additional options, providing charging times between 3 and 14h. You need a charger with round 5.5mm plug.
- The Mainboard provides **6 small expansion areas** (and additionally 2 very tiny areas on the small sensor PCB on the front) for your own sensor circuits, e.g. for implementing additional IR sensors to improve obstacle detection. Expansion areas may also be used for mounting purposes, e.g. for fixing mechanical objects.
- **Lots of expansion possibilities!**

Furthermore we supply quite a few C example programs and an extensive function library, for comfortable software development.

The robot's Website will soon provide additional programs and software updates for the robot system and its expansion modules. Of course we invite you to share your own programs with other RP6 users via internet. The RP6Library and the example program files are released under the Open Source Licence GPL!

1.4. Was can the RP6 do?

Well, not much - directly taken out of the box!

The Software enables the RP6 to actually do something – what this is exactly, is up to you and your creativity to teach the robot how to perform well. The attraction of robotics bases on the fascinating process of implementing new ideas or optimizing and improving excisting things! Of course you may start by simply executing and modifying the prepared sample programs to have a look at the standard features, but it is not limited to that!

The following list mentiones only a few examples and it is up to you to expand the RP6. There are hundreds of possibilities (s. next page for example).

Basically the RP6 can …:

- ... cruise around autonomously (this means independently, without remote control)
- ... avoid obstacles
- ... follow light sources and measure light intensity
- ... detect collisions, blocked engines, low battery level and react properly on that
- ... measure and control the rotational speed of the motors – virtually independently of the power level of batteries, weight, etc. (this is performed with the high resolution encoders)
- ... move for a given distance, rotate for specific angles and measure the driven distance (see chapter 2 for deviations)
- ... move geometric figures, e.g. circles, polygons and others
- ... exchange data with other robots or devices. Commands may be received from standard TV/Video/HiFi remote controls and you will be able to control your robot just like a remote controlled car.
- ... transfer sensor data and other data to a PC with the USB Interface
- ... be expanded easily by using the flexible bus-system!
- ... modified according to your ideas. Just have a look at the schematics on the CD and the PCB! But please restrict modifications to those you fully understand! It is usually a better idea to start off by using an expansion board – particularly if you are unexperienced in soldering circuits and electronics in general.

1.5. Application proposals and ideas

The RP6 has been designed with good expansion possibilities. If you equip your RP6 with some additional sensor circuits, you can "teach" your Robot some of the following things (some of the following tasks will turn out to be quite complex and the list is roughly sorted in order of complexity):

- Expand the robot with additional controllers providing more CPU power, add additional memory or simply some I/O-ports and ADCs as it will be discussed in the example programs with simple I²C port expanders and ADCs.
- Output sensor data and text on a LC-Display
- React on noise and generate acoustic signals
- Measuring the distance to obstacles with additional ultrasonic sensors, infrared-sensors or similar in order to achieve better collision avoidance
- Track black lines on the floor
- Track and trace other robots or objects
- Control the robot from your PC by using infrared signals (this needs extra hardware. Unfortunately it does not work with standard IRDA interfaces). Alternatively you might start straight away by using wireless RF modules.
- Control the RP6 by using a PDA or Smartphone (in this case we suggest to mount these devices to the robot instead of using them as remote control. But that is possible, too!)
- Collect objects (e.g. tea lights, marbles, tiny metal objects …)
- Attach a tiny robot arm to grasp objects
- Navigate with the help of an electronic compass and/or infrared beacons (made up of small towers equipped with a number of IR-LEDs and positioned at a well known location), in order to determine the robot's position and head for a given location.
- Providing a number of robots equipped with a ball kick and handling mechanism and some extra sensors you might be able to raise a team of robots playing soccer!
- … anything else, which might come to your thoughts!

However, first of all you have to read this manual and become familiar with robotics and programming. The previous list of ideas is just meant as a little motivation.

And if programming does not succeed at first glance, please do not give up immeadiately and throw everything out of the window: all beginnings are difficult!

2. The RP6 in detail

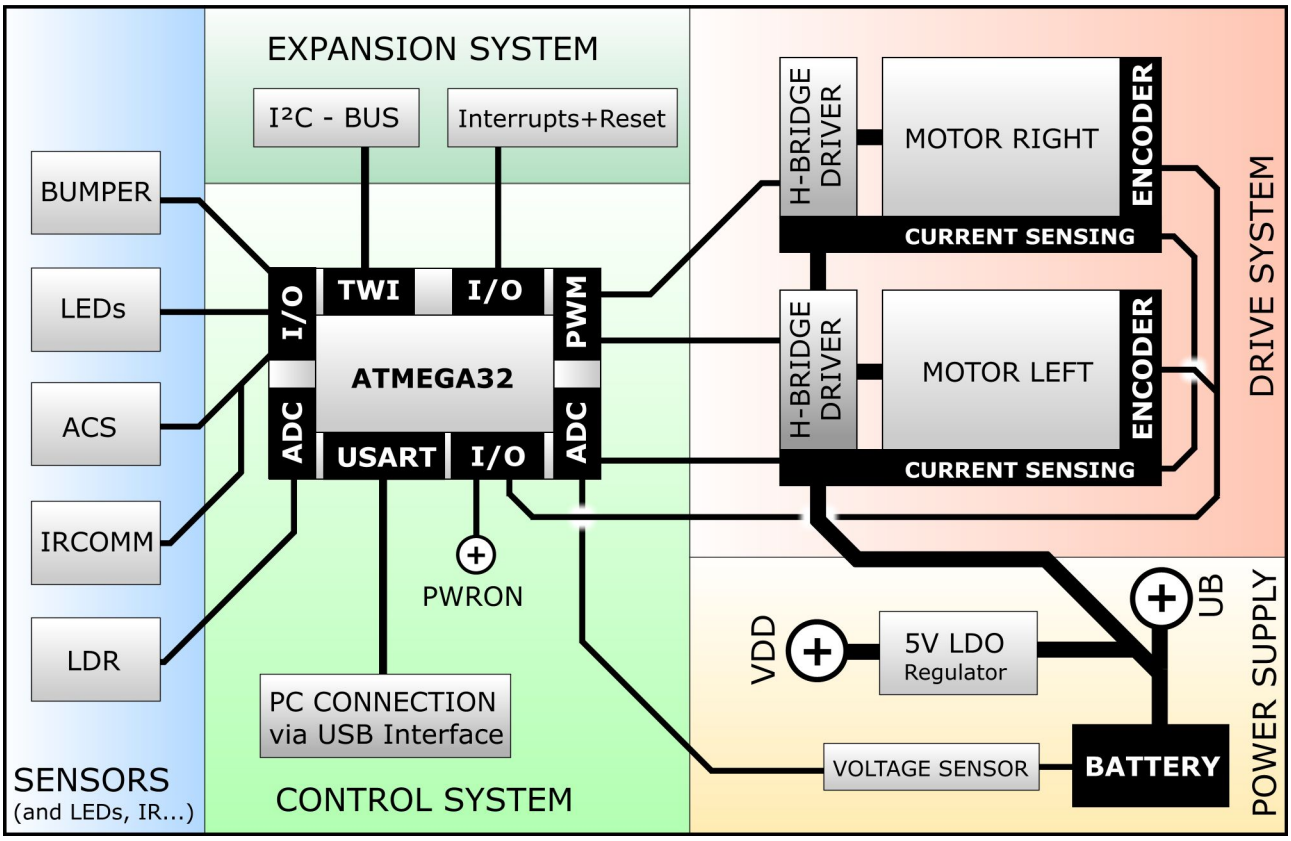
This chapter describes the most important hardware components of the RP6 ROBOT SYSTEM. We will discuss the electronics, the microcontroller and the interaction of software and hardware. If you are already familiar with microcontroller technology and electronics, you will probably just glance through this chapter. Beginners in robotics however should study this chapter to gain insight to the RP6 basics.

If you do not want to wait and rather like to test the robot, then please proceed to chapter 3, but return to this chapter later on as it certainly contains a number of useful explanations of the robot's programming details. And you do want to know what is controlled with the software and how this works, don't you?

We will not go deeply into details, but still a few topics in this chapter might be hard to understand – the author tried to explain things as simple as possible.

If you wish to study special topics in detail you may also look for additional information at `http://www.wikipedia.org/`, which definitely is a good starting point for most topics.

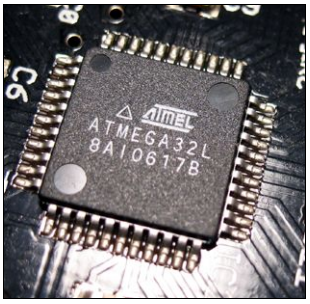
Images often tell more than words and that is why we start with an overview diagram of the RP6. The diagram shows a drastically simplified schematic of the robot's electronic components and how they are connected together:



We can divide the Robot in five main functional units:

- Control System
- Power Supply
- Sensors, IR Communication and Displays (Sensors) – everything communicating with the outside world and measureing physical values.
- Drive System
- Expansion System

2.1. Control System



As you can see in the diagram, the robot's central unit is an ATMEL ATMEGA32 8-Bit Microcontroller (see fig.).

A microcontroller is a complete computer in a single chip. This microcontroller differs from bigger computers (e.g. your PC) by providing less peripherals than the big ones. Of course, the tiny processor does not have a normal hard disk drive and Gigabytes of RAM. A microcontroller does not need that much memory. The MEGA32 provides "only" 32KB (32768 Bytes) Flash ROM – which might be compared to a normal "hard disk drive" - or nowadays a flash-drive. This Flash ROM is used to store all program data. The Random Access Memory (RAM) size is limited to 2KB (2048 Bytes) and is already more than sufficient for our needs. Imagine for comparison the controller of the old CCRP5 Robot with only 240 Bytes RAM, which was almost completely reserved for the Basic Interpreter.

But what on earth enables a microcontroller to work with this tiny memory capacity? That's simple: The processor neither handles huge amounts of data nor does it need to provide an operating system such as Linux or even Windows and it does not need to show a complex graphical interface or serve similar tasks. There will be just one program running and that's our own one!

These limitations are by no way a disadvantage, but one of the main advantages of microcontroller systems compared to large computers (additionally we may include power consumption, size and cost)! The processor is designed to handle jobs in known time slots (also called "realtime"). Usually we do not have to share the processor's power with a great number of processes as in a standard PC and programmers are able to carefully determine the time slot for any special function module.

The RP6 controller runs at 8MHz, which enables a processing speed of 8 Million instructions per second. The processor would even allow up to 16MHz clock, but we use the slower clocking option which allows some power savings. The machine is still fast enough for all of our standard jobs! Again we are comparing to the controller of the old predecessor CCRP5 with a 4MHz clock, allowing only approximately 1000 (interpreted) Basic instructions per second. For this reason the ACS control had to be managed by another slave controller on that old Robot – we do not need this slave processor anymore! And whoever needs more processor power may add one or more controllers to the expansion interface. The additionally available RP6 Control M32 Expansion Module provides an additional MEGA32, which is clocked with the maximum 16MHz clock frequency.

The controller is communicating to the world outside via 32 I/O Pins ("Input/Output Pins"), organized in "Ports", each composed of 8 I/O Pins. This way the MEGA32 provides 4 "Ports": PORTA to PORTD. The controller is able to read the logical status of these ports and process the information in software. Of course, the processor will equally use the ports to output logical signals in order to control small loads up to 20 mA currents (e.g. LEDs).

Additionally the controller provides a number of integrated hardware modules for special tasks. Implementing these tasks in software would normally be either very difficult or even impossible. One of these special tasks is timing. Three Timers for counting clock periods are available. The timer modules are completely independent from program flow. In fact, the microcontroller may even process other jobs while waiting for a programmed counter level.

RP6 is using one of the timers to generate PWM signals (PWM="Pulse Width Modulation") for speed-control of the motors and as soon as the timer has received appropriate input parameters it will manage this task in background. We will discuss more details of the PWM signal generation in the chapter “Drive System”.

For example some other modules of the MEGA32 are:

- A serial interface (UART) for PC-communication with the RP6 USB Interface. Using this interface you might also connect another microcontroller with an UART, as long as the USB Interface is not connected.
- The "TWI"-module (="Two Wire Interface") providing the I²C Bus for expansion modules.
- An Analog-to-Digital Converter (ADC) providing 8 input-channels for measuring voltages with 10bit resolution. RP6 is using the ADC for monitoring the battery voltage level, motor current-sensors and light intensity with two light-dependant resistors.
- Three external interrupt inputs for generating interrupt signals, which will interrupt the program flow in the controller and force the program to jump to a special "Interrupt Service Routine". The microcontroller will then process this routine immeadiately and return back to the normal program. We will be using this programming feature for the odometry sensors. We will discuss this sensor in detail later on.

The integrated hardware modules do not have their own individual pins, but may be used alternatively instead of standard I/O Pins. Normally you may freely choose these special function mapping for the I/O Pins, but the RP6 almost completely provides a standard configuration (as it is hard-wired to all components) for pins and modifying the standard configuration will hardly be useful.

The MEGA32 provides a lot of other things, which cannot be described in detail in this manual. You get more information on this in the datasheet of the manufacturer (which can be found on the RP6 CD-ROM).

2.1.1. Bootloader

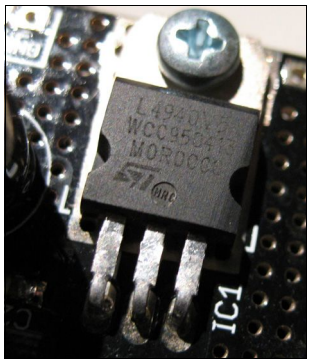
There is a so called Bootloader located in a special memory area of the microcontroller. This tiny program is responsible for loading new user programs into the microcontroller's memory via the serial interface. The Bootloader communicates with the the RP6Loader software on the host PC. Like this, no additional programming hardware is required. The USB Interface can be used for communication with the controller through text messages and additionally to program the controller. However there is one drawback in using a Bootloader: it needs 2KB of the flash memory, which will leave 30KB free memory for your own programs. This does not bother us too much as there is plenty of room even for very complex programs (compared to the 7KB free memory of the tiny ASURO robot)!

2.2. Power Supply

Of course the robot needs energy. The RP6 is carrying this energy in form of 6 accumulator batteries. Operating time will heavily depend on battery capacity and although the electronic systems will consume relatively small amounts of energy the bulk load of energy will end up in the motors, depending on their load.

In order to provide long operating times you might favour batteries with ample capacities of up to 2500mAh. Capacities of 2000mAh however will be useable as well. High quality batteries will provide between 3 to 6 operating hours, depending on motor load and battery quality. You will need 6 batteries, summing up to a voltage of 6x 1.2V = 7.2Volts. The block schematic diagram labels this battery voltage "UB" (= "U-Battery", U is the standard letter for voltage in electrical engineering formulas). "UB" is defined as a nominal voltage only, as the voltage may vary over time. Completely charged NiMH batteries can deliver up to 8.5V! The voltage drops while the Battery is discharged and may change drastically, depending on load and quality as well. The critical value for this is the internal resistance.

Of course, an altering supply voltage is not useable for sensor measurements. More important however is the limited operating voltage range of semiconductor components. The microcontroller for instance might be destroyed by applying voltages too high over 5V. Therefore we have to reduce and stabilize the voltage level to a well defined level.



This is performed by an integrated voltage regulator capable of supplying a current up to 1.5A (see figure). At 1.5A this device would dissipate a lot of heat and therefore it is attached to a large copper plane on the PCB. Even with this heat sink we suggest to limit currents over 1A to a few seconds only. Otherwise you will have to attach an additional heat sink.

Continuous current load should be limited to about 800mA. Such a heavy load would quickly discharge batteries anyway.

Under normal load conditions and without expansion modules the robot will not draw more than 40mA, unless the IRCOMM transmitter is active. This current level will not cause any problems for the regulator and it can supply enough power for lots of expansion board. Usually the expansions will need something in the range of 50mA, if no motor loads, power LEDs, etc. are used on them.

2.3. Sensors

Most sensors have been mentioned in preceding chapters, but now we will have a closer look at them.

In the overview diagram you will find sensors outside of the blue-coloured area "Sensors". Actually these sensors belong to other modules. However, the odometry encoders, the motor current sensors and the battery voltage sensor are sensors and will be discussed in this chapter, too!

2.3.1. Battery Voltage Sensor

Basically this sensor is a simple voltage divider consisting of two resistors in series. We can assume to have a set of batteries with a maximum voltage of 10V. 6 NiMH batteries will certainly never exceed this level. The ADC reference voltage, which is compared to the monitored voltage, is set to 5V. The maximum 5V operating voltage of the Microcontroller must not be exceeded at any time. For this reason the monitored voltage will have to be divided by 2. To achieve this we use the voltage divider with two resistors, matching the monitored voltage to the ADC's voltage range.

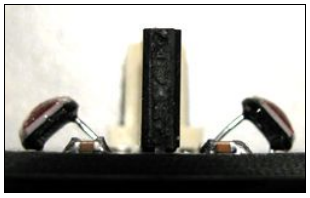
The ADC measures the voltage at a resolution of 10 Bits (which implies a value range between 0 and 1023 units), resulting in a voltage resolution of $\frac{10\text{V}}{1024} = 9.765625\text{mV}$.

A measurement value of 512 units corresponds to 5V and 1023 to approximately 10V! These limits are usually not reached with 6 normal NiMH batteries!

The measurement is not too accurate, as we are not using precision resistors. A few percent tolerance has to be taken into account. The reference voltage is not accurate as well and may be fluctuating in heavy load conditions. We do not care about these tolerances, as we only need an indicator for checking the discharging limit of the batteries. If you need to determine the exact voltages, you will need to use a Multimeter to check the exact voltage values and then add correction values in software.

If you can accept tolerances you may even directly estimate the voltage from the ADC value: 720 units are corresponding roughly to 7.2V, 700 to 7.0V and 650 to 6.5V. A value of constantly 560 can be considered as empty batteries.

2.3.2. Light Sensors (LDRs)



At the front side of a small sensor-PCB you may spot two so-called LDRs (="Light Dependant Resistors"), which are aligned to the left and to the right respectively. There is a black partition wall between the two sensors in order to prevent light entering the "wrong" side of the light sensor system. Just like the Voltage sensor, both light sensors form a voltage divider together with a resistor, but here to determine the light intensity. In this case, the 5V rail is divided as well, but now we have a variable resistor. The division relation will change according to the surrounding light intensity and provide a light dependant voltage level to one of the ADC channels!

The voltage difference between both sensors may be used to determine at which side of the robot the brightest source of light is located: left, right or in the middle. A suitable program can trace a bright torch in a *darkened* room or guide the robot to the

brightest spot in the room! For example if you illuminate the floor with a very bright halogen-torch, the robot will follow the bright light spot on the floor.

Of course you may try the opposite: the robot could be programmed to hide from bright light sources.

You can refine this by mounting one or two additional LDRs at the backside of the robot. By using only the two sensors which robot has by default, it can not distinguish between bright light in the front and in the back too well. Two of the A/D-converter channels are still free...

2.3.3. Anti Collision System (ACS)

From the software's point of view, the most complex Sensor is the ACS – the "Anti Collision System"! The ACS consists of an integrated infrared (IR)-receiver circuit (see fig.) and two IR LEDs, located at the left and right front-side of the sensor PCB. The Microcontroller is controlling the IR-LEDs directly. The controlling functions can be changed and improved by yourself if necessary! The predecessor model had a special controller for this purpose and the user could not modify the software of this device.

The IR LEDs are transmitting short infrared pulses modulated with 36kHz, which can be detected by the IR-receiver. Whenever IR-pulses are reflected by an object and received back by the IR-receiver, the microcontroller may react to the situation and start an escape manoeuvre. In order to avoid too much sensitivity, the ACS routines will delay detection events until the system has received a predefined number of pulses within a small period of time. Additionally, the ACS synchronizes the detection with the RC5-receiver routines and will not react on RC5-signals from TV/Hifi remote controls. Other codes however may interfere with the ACS and the robot may try to avoid non-existent obstacles!

Given that the ACS has one IR LED aligned to the left and one to the right, it can roughly determine whether the obstacle is in the middle, left or right.

The system allows you to change the pulsed intensity of both IR LEDs at three levels. But even at the highest current level, the ACS may not detect all obstacles reliably. This is greatly dependant on surface reflectivity of the obstacles!

Of course a black object will reflect less IR-light compared to a white obstacle and a reflecting square-edged object may lead the IR-light mainly into a few special directions. Therefore the ACS range is drastically depend on obstacle-surface! This dependency must be considered as a basic drawback of all IR-sensor systems (at least in this price-class).

However the robot can detect and avoid most obstacles flawlessly. If ACS-detection fails, there are still the bumpers with touch sensor elements. And if the touch sensors fails the robot may detect motor blocking by its current sensors or encoders!

If you are not satisfied with these sensor systems, you might consider mounting some ultrasonic sensors for example.

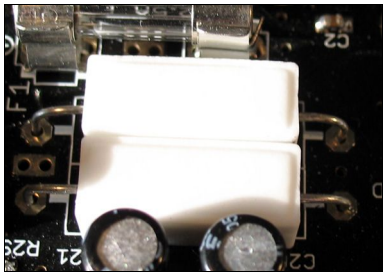
2.3.4. Bumpers

There is a small PCB with two micro switches with long levers mounted in front of the Robot. It protects the IR LEDs on the sensor-PCB from being damaged if the robot accidently hits an obstacle. With these switches the Microcontroller can detect collisions and reverse the robot's direction, turn around and then go on with moving forwards.

The switches are connected to ports already used by LEDs. Thus they do not occupy free ports of the Microcontroller. These dual usage causes the LEDs to light up as soon as one of the switches is pressed down! However, the switches will only be hit occasionally and activated LEDs will not disturb anything then.

The bumper PCB may also be removed and for example replaced by a kick/collecting device for balls or something else.

2.3.5. Motor Current Sensors



Each of the two motor current sensor circuits contains a power resistor. Ohm's Law $U = R \cdot I$ tells us that the voltage drop at a resistor is proportional to the current flow through it!

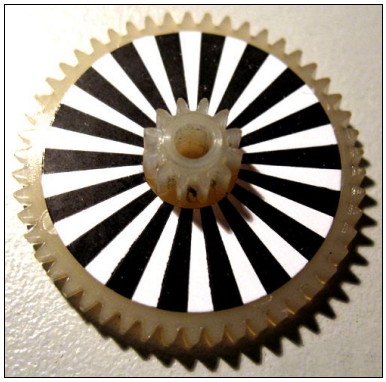
In order to prevent excessive voltage drops at these resistors, they need to have a very small resistance value. Here we used 0.1 Ohm.

With such a tiny value, the voltage drop is very small (0.1V at a 1A current) and has to be amplified before the A/D-conversion can take place. This is performed by a so-called operational amplifier (opamp). The RP6 uses an opamp for each individual current sensor. The measureable current range is about 1.8A. This current results in a voltage drop of 0.18V at the power resistor and an opamp output voltage of approximately 4V. This is the maximum output voltage for the opamp with 5V power supply.

The used power resistors are 10% tolerance types, the resistors at the opamp are 5% ones. All components are non-precision components and you may observe measurement deviations of up to 270mA if you do not calibrate this! However we only need a roughly estimated current level to detect critical motor load conditions. The robot will reliably detect blocked/heavily loaded motors and even defective motors or odometer wheel sensors! DC-Motors draw more current the higher the load is (Torque). With blocked Motors, the current gets very high for our motors. This is detected by the Software and an emergency shutdown is initiated. If this would not be done, the Motors would get very hot and and this (and the high torque) damages them over time.

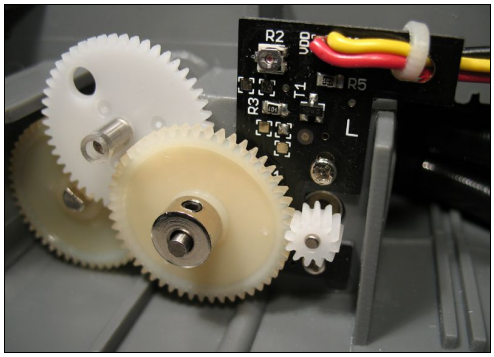
If an encoder fails – whatever may have caused this - the system can reliably detect this condition, too. Of course, the measured velocity would be zero. But if the motor drivers run at full power and the current sensors detect only low currents (which implies that the motors are not blocked!) you may conclude either motor or encoder failure or both. For example such a condition can arise if you forget to activate the sensors in software ...

2.3.6. Encoders



The encoders work completely different compared to the previously discussed sensors. They consist of reflective interrupters and code wheels which are attached to one of the gearwheels in each gearing system. This setup is used to determine the rotational velocity of the Motors. Both encoder wheels have 36 segments (18 black and 18 white fields, see figure). While the gears rotate, these segments move along in front of the reflective interrupter. The white segments reflect the IR-Light, whereas the black ones will only refelct a minor amount of light. Just like the other sensors the encoders produce an analog signal, but it will be interpreted digitally. First of all the signal has to be amplified and subsequently converted to a square wave signal by a so-called Schmitt Trigger. Both rising and falling edges of the signal (changes of 5V to 0V and 0V to 5V) trigger an interrupt event and these event are counted by software. This way the driven distance can be measured and together with a timer the rotational velocity can be calculated.

Determination of the speed is the main application of the encoders. Encoder feedback is the only reliable way to control the motor speed. In an uncontrolled system, the motor speed would be depending on battery voltage, load and motor parameters. The high resolution encoders even allow us to reliably control rather slow speeds.



Each of both cluster gears in the middle of the gearing system provide 50 teeth at the outer and 12 teeth at the smaller inner gearwheel (see figure). The code wheels are located at the gearwheel next to the motor pinion gear, thus we can calculate:

$$\frac{50}{12}\cdot\frac{50}{12}=17\frac{13}{36} \quad ; \quad 17\frac{13}{36}\cdot\mathbf{36}=625$$

This is where the 36 Segments come from, because this results in an integer number without fractional part for a complete wheel revolution. The encoders generate 625 egdes per revolution and whereas each represents one segment.

A wheel diameter of around 50mm including the rubber track theoretically results in a wheel circumference of approximately 157mm and thus 0.2512mm for each counting unit of the encoders. However the tracks may get deformed under pressure or they may get pushed into flexible surfaces. Therefore we can directly assume a maximum of 0.25mm for each counting unit. Often we will have to apply even less: 0.24 or 0.23mm. Calibration values may be determined by driving well defined test distances as described in the Appendix. This is not accurate because of slippery and similar effects. Moving straight forward will cause minor encoder accuracy errors, but rotating the robot will result in increased deviations. Especially rotating the robot on the point will cause deviations.

Deviations can only be determined and corrected by testing, trial and error. This is a drawback for all caterpillar drive systems – in our robot and in more expensive as well. Compared to robots with a standard differential drive unit with two wheels and an additional support wheel the caterpillar systems allows a far better behaviour in all-terrain surroundings. The caterpillar drive system will easily overcome small

obstacles, ramps and uneven floors. On such surfaces, the encoders are extremely helpful, as they allow optimal speed regulation under all load conditions, completely independent of surface condition, motor load and Battery voltage.

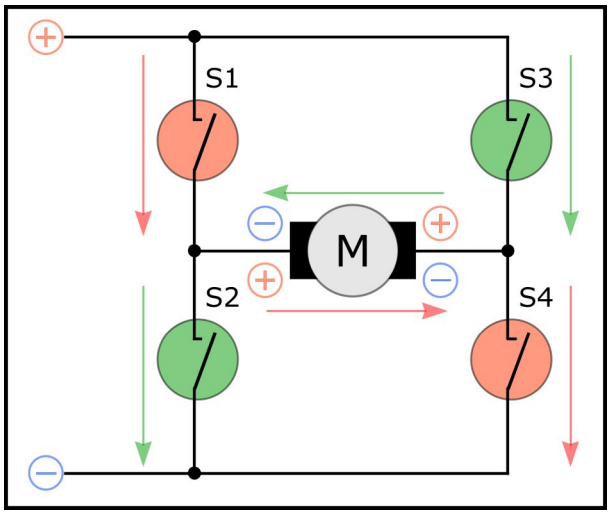
At a rate of 50 segments per second we have a speed of 1.25 cm/s assuming a value of 0.25mm per segment. This speed is the minimal speed, which can be controlled reliably (at least with the standard software implementation). The exact value may vary for individual robots. A rate of 1200 segments/second corresponds to the maximum possible 30 cm/s (at 0.25mm resolution, whereas 0.23 corresponds to 27.6 cm/s). Maximum speed depends on battery charging status and 30cm/s will are not possible for too long with usual Batteries. Because of this, the function library forces a limit of 1000 segments/second to be able to maintain a constant maximum speed for longer battery discharge periods. Additionally, the life time of gears and motors will be prolonged when using lower speeds most of the time!

Whenever the robot has counted 4000 segments, it will have covered a distance of approximately one meter. As already explained, this specification is valid for exactly 0.25mm resolution - without proper calibration you will notice more or less severe deviations. If you do not care for precise distance calculations, you just do not need to calibrate the encoders and simply assume a value of 0.25mm or better 0.24mm!

Good navigation systems usually do not rely completely on encoders for distance and angle controls, but use external fixed markers such as infrared beacons and a precision electronic compass. So it is usually a good idea to use external systems to correct odometry deviations as often as possible.

2.4. Drive System

The RP6 drive system consists of two DC motors with attached gearing systems for powering the caterpillar wheels (see preceding figure). The motors can consume a fairly high amount of power and a microcontroller can not directly serve such high currents.

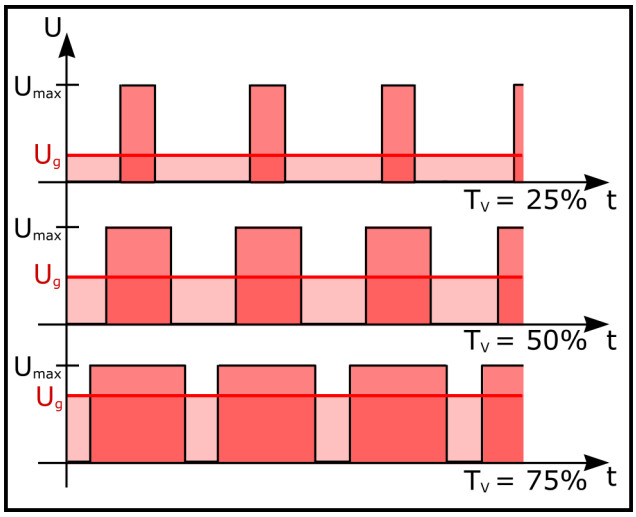


Thus we need powerful motor drivers. We use two so called H-Bridges for the RP6 Motors. The diagram on the left shows the basic principle. There you can also see why it is called like this: The Switches and the Motor form the letter "H" together.

Now let us assume all switches are open. If we close switches S1 and S4 (red) a voltage will be applied to the motor and it will start turning, say to the right. If we now open S1 and S4 again and subsequently close S2 and S3 (green), we reverse the applied voltage and the motor will start turning in the opposite direction (to the left). Of course we will have to take care not to close S1 and S2 or S3 and S4 simultaneously. Each of these combinations would result in a short circuit and might destroy the activated switches.

Of course the RP6-design will not be using mechanical switches but so-called MOSFETs, which are conductive if a suitable voltage is applied to the gate connection. MOSFETs can switch very fast at a rate of several Kilohertz is possible.

Now we have found a way to reverse the motor's rotational direction. And how are we going to accelerate or slow down the motor? A DC-Motor will rotate faster the higher the voltage gets and we may control the motor speed by increasing or decreasing the voltage. Let's have a closer look at the H-bridge again.



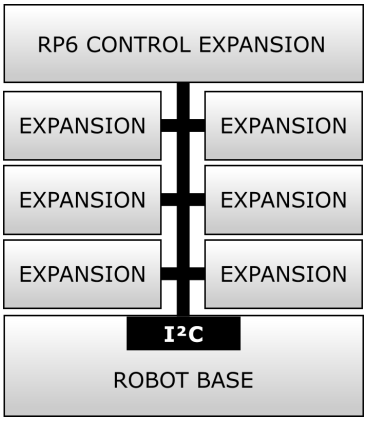
The figure shows what we can do. We generate a square wave at a *fixed* frequency and apply pulse width modulation, which changes the duty cycle. "Duty cycle" means the ratio between high and low signal periods.

The motor will now get a lower median DC-voltage, corresponding to the duty cycle.

The graph shows this behaviour with a red line (U_g) and the red areas under the lines. For example if a battery voltage of 7 Volts is applied to the motor controller circuit and the motor is being controlled at a PWM duty cycle of 50% the equivalent median DC-voltage would be roughly 3.5 Volts! This is not completely corresponding to the real circuit conditions, but it is good to visualize it like this.

The RP6 uses a rather high gear reduction ratio (~ 1:72) which results in a quite strong driving system, enabling the robot to carry heavier loads, for example compared to the small ASURO robot. However with increasing weight, we must consider a higher power supply load, resulting in an increased discharge rate…

Compared to remote controlled racing cars you might think the RP6 is a slow vehicle - which is true - but we designed the robot to be slow! The robot is build to be controlled by a microcontroller and if the programmer makes a mistake in the software it would be rather unfavourable if the robot crashes into a wall at a speed of 10mph! So by using a moderate speed, the RP6 will not run into trouble that easy and at a slower movement the sensors will have ample of time to react on obstacles. Additionally the robot is more powerful and the speed control is more accurate! Slower speed enables the RP6 robot to drive very very slowly at a constant speed!



2.5. Expansion System

One of the most useful features of the RP6 is the expansion system, allowing you to easily add additional things to the Robot. The basic RP6 platform includes rather few sensors. Still this number of sensors is well above the average of comparable robots in this price-class, but the robot will only become really attractive with several additional sensor modules. The ACS for example will only detect the *existence* of obstacles in front of the robot. Using ultrasonic sensors or improved additional IR-sensors you might be able to determ-

ine the distance and start sophisticated manoeuvres to avoid the obstacles!

Apart from sensor circuits, additional controllers could be useful to perform additional tasks, e.g. the RP6 CONTROL M32 providing an extra MEGA32 microcontroller.

Of course the expansion system has to be capable of connecting several expansion modules (see figure), while using a minimum number of signal lines and providing sufficient communication speed.

2.5.1. The I²C Bus

The I²C Bus will satisfy these requirements. The name stands for **Inter Integrated Circuit** Bus and is pronounced *I-squared-C.* Sometimes we may write "I2C" instead of "I²C", because in plain C language the "²" symbol is not allowed for variable names and other things. The bus requires only two signal lines and may connect 127 participants communicating at a rate of 400kBit/s.

The extremely popular I²C Bus, designed by Philips Semiconductors during the eighties and nineties, is used in a great number of electronic equipment, e.g. video recorders, televisions, but also in industrial systems. Most of the modern PCs and notebooks use a variant of this bus called SMBus to control air flow and temperature of the internal devices. A great number of robots also uses the I²C Bus system and for this reason a number of sensor modules like ultrasonic sensors, electronic compasses, temperature sensors and similar devices are available on the market.

The I²C Bus is a master/slave-oriented bus. One or more master devices are controlling communication with up to 127 slave devices. But even though the bus is able to handle multi-master communication, we will only describe a bus communication with a single master device. Multi-master topology would only complicate things.

The two required signal lines are named SDA and SCL. SDA is to be read "Serial Data" and SCL is named "Serial Clock" – which already explains we are using a data- and a clock signal line. SDA is used as a bidirectional signal and therefore both master and slave devices are allowed to output data. SCL is completely controlled by the master device.

Data bits are always transferred synchronous to the clock signal as delivered by the master. The SDA level is only allowed to change as long as SCL is low (except for Start- and Stop-conditions, see below). Transfer rates are allowed to change between 0 and 400kBit/s even while data is being transmitted.

The preceding figures show usual transmission protocols. The first one shows a transmission from a master to a slave device, in which white boxes refer to data transmissions from master to slave and the dark boxes represent the responses from the slave device.

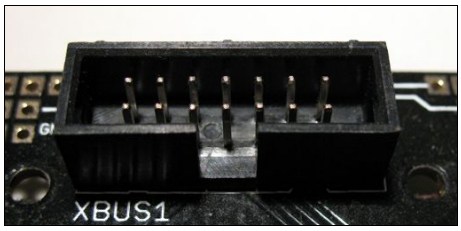
Each transmission has to start with an initial start condition and must be ended by a stop condition. The start condition is raised whenever at a high SCL-level the data line SDA is pulled from high to low level. A reversed level pattern applies to the stop-condition: whenever at a high SCL-level the data line SDA is pulled from low to high level we meet a stop-condition.

Immediately after the start-condition we have to send the 7 Bit long slave-address for the device to be addressed, followed by a bit defining whether we want to write or read data. The slave will respond by sending an ACK ("Acknowledge"). Any number of data bytes may follow and each individual received byte will have to be acknowledged by the slave (using the ACK-signal). Communication will be terminated with a stop-condition.

This description is only a very short explanation of the I²C Bus. Interested readers may find more information in the I²C Bus specification sheets by Philips. And also the specs for the MEGA32 do contain more information to this topic.

The example programs demonstrate how to use the bus hardware. The RP6 library already provides functions for controlling the I²C Bus. You will not have to go into details of the protocol, but it is useful to understand how bus the communication is basically working.

2.5.2. Expansion Connectors



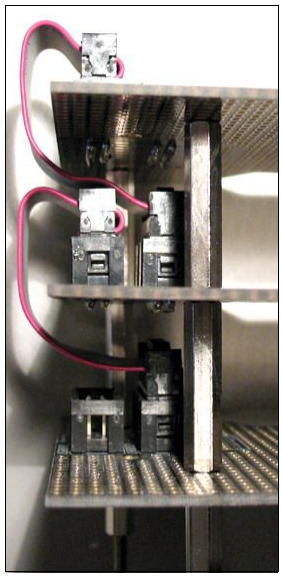
The mainboard provides four expansion connectors. Two of these devices are labelled "XBUS1" and "XBUS2" respectively. "XBUS" is a shortcut for "eXpansion BUS". "XBUS1" and "XBUS2" are interconnected completely and have been arranged symmetrically on the main board. For this reason you will be allowed to mount expansion modules both at front and rear of the robot. Each expansion module provides two XBUS connectors at one side of the module. A 14-pin flat cable is used for interconnecting the modules to each other and to the mainboard. For interconnections each expansion module provides *two* identical interconnected plugs. The outer plug has to be used for downward interconnections, whereas the inner plug has to be used for upward interconnections. This way you are (theoretically) allowed to stack any number of modules (see figure, showing three RP6 breadboard expansion modules, which may be used for your individual circuits).

The XBUS plugs provide power supply, the previously described I²C-Bus, a master rest and interrupt signals.

The power supply provides two voltages at the connectors: first of all the stabilized 5V from the voltage regulator, but the battery voltage as well. The battery voltage will vary with time and load – usually between 5.5 (discharged batteries) up to approximately 8.5V (newly charged batteries – varying from manufacturer to manufacturer). Voltages may however exceed these limits depending on load, type and charging status of batteries.

The master reset signal is important for resetting all microcontroller devices when pressing the Start/Stop-button or for programming. The boot loader programs in the microcontrollers will start their user program at a low pulse (high-low-high) on SDA. This way all programs on the (AVR) controllers will simultaneously start after pressing and releasing the Start/Stop-button or by starting the program by boot loader software... (the boot loader does not only generate a low impulse to start, but also a complete I²C General Call with 0 as data byte.)

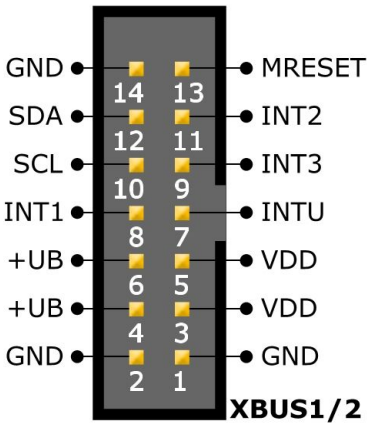
A few modules can use the interrupt lines for signalling the master microcontroller either the arrival of new data, or whether a job has been completed and new commands are being expected. Not providing these lines would force the master device to repeatingly query some specific expansion modules for new data. Of course this method would be possible, but the alternative design with additional interrupt lines will usually reduce bus traffic and CPU load. As the number of interrupt lines is restricted to 3 signals and one free line reserved for user signals, you may have to assign one line to several modules (e.g. all ultrasound sensors) and poll all modules subsequently when an interrupt is signalled.



The other two expansion connectors labelled "USRBUS1" and "USRBUS2" on the mainboard are not interconnected. All lines are routed to soldering pads on all expansion modules and you may apply your own signals to these pads.

"USRBUS" is an abbreviation of "User-Bus". You may use this 14-pin expansion connector for anything you want - your own bus system, additional power supply lines (but be careful, the traces are rather thin and for low currents of max. 500mA only) or anything else. Example given: you are able to interconnect two expansion modules without providing connections to other modules. This might be useful for more complex circuits or sensors, which cannot be placed on a single expansion module. This method will tidy up your wiring.

Of course you can not add any number of expansion modules – 6 stacked modules at the front- or backside will definitely overload your vehicle. Too many modules will also cause problems by overloading the battery supply. As a general rule you may mount a maximal number of 8 modules to the RP6: 4 at the front side and 4 at the backside.



The figure shows the connection diagram for both expansion connectors. On the mainboard, pin 1 is always located near the white label XBUS1 and XBUS2, respectively. Alternatively, the pin is labelled with "1" at the connector-position.

+UB is the battery-voltage, VDD is the +5V rail, GND labels "Minus" or "Ground" (GND = Ground), MRESET labels the Master Reset Signal, INTx are the Interrupt-lines, SCL is the clock- and SDA the data-line of the I²C Bus.

Anything else you need has to be soldered to the USRBUS connector pins.

Important notice: **Do not overload the supply lines VDD and +UB! These lines can provide a maximum current of 1A each (this applies to both Pins TOGETHER! This means joined Pins 4+6 (+UB) and 3+5 (VDD) of the connectors)!**

3. Hardware and Software Setup

Before you start with setting up the RP6 or accessories, you have to read the following safety instructions carefully. Especially if children are handling the RP6 later on!

Please read this chapter extra carefully!

3.1. Safety Instructions

Due to the open frame architecture of the RP6, there are several sharp edges. Thus the Robot may not be used as a toy for children aged less than 8 years! Please supervise children that are in the room while operating the RP6 and inform your children about the described dangers!

Do not operate the robot in locations with freely moving animals, for example hamsters, as they may get hurt. The other way round, bigger animals like dogs and cats might damage the robot...

The caterpillar drive system has some dangerous sectors **between tracks and wheels, where the caterpillar may draw in you fingers**. These sectors are largely covered by the wheel wells of the RP6 and therefore mostly secure. Still, take care not to get your fingers between wheels and tracks! The motors are quite powerful and may easily hurt you! Also keep your fingers out of the area between PCB and tracks!

ATTENTION: Even if you are using the standard software, the motors may automatically increase their power level! Depending on the programming style, motors may start operation at any time and unforeseen reactions and movements may occur! ***Never operate the robot without supervision!***

3.1.1. Electrostatic Discharges and Shorts

The surface of the main PCB, the USB Interface and all expansion modules is uncovered and reveals a great number of **unprotected components and PCB traces**. Please do not cause short circuits by deposition of metallic parts or tools on the surface of the Robot!

Supply voltages are at very low levels only and safe for human beings. **A great number of components however may get damaged by electrostatic discharges (ESD) and you should not touch these components** unless necessary!
Especially in combination with synthetic clothing, dry air may cause electrostatic charging of the human body. And the robot as well may be charged, mainly depending on the floor-covering. In touching metallic parts charged bodies will be discharged by tiny sparks. These discharges may damage or destroy electronic components while manipulating these parts. To prevent damages by ESD please touch a large grounded device (e.g. your PC's metal housing, a drainpipe or a heating pipe) before touching the electronic components. Touching a grounded device will discharge your body. Uncontrolled discharges of the robot touching grounded obstacles will not damage the robot, but it may cause program crashes or unforeseen reactions.

All electric lines from and to the system must be connected before connecting and applying the supply voltage.
Unexpected connecting or disconnecting plugs, cables or modules in an operating robot may damage or destroy parts of the system and additional components.

3.1.2. Environment of the Robot

Do not operate the robot on table tops or in areas with high precipies, where it may fall down to the ground. Please consider the climbing capability of caterpillar vehicles! The robot may easily drive over small obstacles and push light parts ayway! Please remove all objects containing liquids from the robot's operating area, e.g. cups, bottles and vases.

The robot's chassis will protect the mechanical parts against a number of environmental influences, but it is neither water- nor dustproof. The electronics are rather unprotected as well. You should operate the robot in clean and dry in-house areas only. Dirt, loose particles and humidity may damage or destroy mechanical and electronic components. Operating temperatures are to be restricted between 0°C and 40°C.

Especially inside operating DC-Motors, tiny sparks are generated. **Do not operate the robot at all** in an environment with combustibles or explosives (liquids, gases or dusts)!

If not operated for long periods of time, the robot should not be stored in locations with high humidity! Please also remove the batteries to prevent damage by leaking batteries!

3.1.3. Supply Voltage

The robot has been designed for a 7.2V supply voltage, provided by 6 rechargeable NiMH batteries. Maximal supply voltage is 10V and shall not be exceeded at any time. Only use charging devices with valid and legal safety certifications for charging batteries!

As a remedy you may also operate the robot with 6 heavy duty alkaline batteries. Normal batteries however will discharge rapidly and cause high costs and environmental damages, so please use rechargeable batteries if possible! Rechargeable batteries will also provide higher maximum currents and may easily be charged inside the robot!
Please pay attention to the safety and disposal remarks for batteries in the appendix!

Modifications of the robot should only be done by users, who are completely aware of what they are doing. You may irreversibly damage the robot or harm yourself and others by modifications (e.g. overheating components may cause fire in your house...)!

3.2. Software Setup

Software setup comes next. Correctly installed software is required for all following chapters.

You will need administrator rights to install, so please login as an administrator to your system.

We suggest that you first read the whole chapter and then subsequently follow the instructions step by step!

We need to assume, that you have basic knowledge in working with computers using the operating systems Windows or Linux and standard software packages such as a file manager, web browser, unpacker (WinZip, WinRAR, unzip, etc.) and if relevant e.g. the Linux-Shell! If you are not familiar with using computers, you should prepare yourself to acquire basic knowledge in this field before starting to operate the RP6! We cannot provide an introduction course in computer usage in this manual, as this topic is out of scope! This manual will describe the RP6, programming the RP6 and the dedicated system software.

3.2.1. The RP6 CD-ROM

You probably inserted the RP6 CD-ROM already into the CD-ROM-drive of your PC – if not, please insert the CD now! In Windows you should observe an auto start action and the CD menu should show up in a browser windows. If not, you can open the file "start.htm" in the CD's main directory in a web browser, e.g. Firefox. If your PC does not provide a modern browser you may find a Firefox installation package in the CD-directory:

```
<CD-ROM-Drive>:\Software\Firefox
```

You should use at least Firefox 1.x or Internet Explorer 6.

Having selected your language, the CD menu will offer you a lots of useful information and software. Apart from this manual (which may be downloaded from our homepage as well) you may have a look e.g. at data sheets of the Robot's components. The menu entry labelled "software" provides access to all software tools, the USB-driver, and example programs with source code for the RP6.

Depending on security settings in your web-browser you may start installation packages directly from the CD! If your browser's settings do not allow installation, please proceed by saving the files to a directory of your disk and start installation from there. You will find details to these procedures on the software page of the CD menu. Alternatively you might also browse to the CD root directory in your file manager and start installation directly from the CD. Directory names have been chosen to correspond to their respective software packages and operating system.

3.2.2. WinAVR - for Windows

First of all we will install WinAVR. *Win*AVR however is – as already indicated by its name – available for *Win*dows only!

Linux users may skip this section.

WinAVR (pronounced "whenever") is a package of useful and required tools for software development with AVR microcontrollers in the C-language. Apart from GCC for AVR target (which is called "AVR-GCC", more infos on this follow later on), WinAVR also provides a comfortable source code editor called "Programmers Notepad 2", which will also be used for software development for RP6. WinAVR is a privately organized project and the package is freely available on Internet for everyone. New releases and further information may be found on the official project website:

`http://winavr.sourceforge.net/`

Just recently, ATMEL started to officially support the project and AVRGCC may now be integrated into their integrated development environment AVRStudio. Programmers Notepad 2 is better suited for our project and we will not describe AVRStudio here. Nevertheless, you may also use AVRStudio for development with RP6 if you like.

The WinAVR installer can be found on the CD:

`<CD-ROM-Drive>:\Software\AVR-GCC\Windows\WinAVR\`

Installing WinAVR is very simple and self-explanatory – usually you do not need to change any settings - just click on continue all the time!

If you run into trouble with the most recent version of WinAVR, there are some older Versions available on the CD, too. There is also a Patch for Win x64 if you encounter problems with the standard version there.

3.2.3. AVR-GCC, avr-libc and avr-binutils - for Linux

Windows users may skip this section!

Installing avr-gcc in Linux environments may become a little bit more complicated. A few distributions already provide the required packages, but often the packages contain obsolete releases without some of the required patches.

Most likely you will have to compile and install new versions.

We cannot refer to details for each of the countless Linux distributions, such as SuSE, Ubuntu, RedHat/Fedora, Debian, Gentoo, Slackware, Mandriva etc. varying in versions and all their quirks. We will provide a general installation approach only.

This also applies to all following Linux topics in this chapter!

For your specific system setup, the following approach may not automatically be successful. Often you will find help by searching "<LinuxDistribution> avr gcc" and by varying the phrases in this search string. This is also a good idea for all other possible problems, which may occur on linux systems! If you are having trouble with avr-gcc installation, you might try to find a solution by visiting our forum or any of the numerous Linux forums out there.

First of all you will have to deinstall any preceding – and as already stated probably obsolete - versions of the avr-gcc, avr-binutils and avr-libc. To start deinstall you may use your distributions package manager tool, search for "avr" and then deinstall the packages from your system, if the tool finds corresponding objects starting with "avr-".

You may easily check whether avr-gcc has been installed or not. If it exists you may ask for the location of the program by executing the following on a command-line:

```
> which avr-gcc
```

If the system responds with a pathname, then most likely an avr-gcc version already exists on your system. In this case you can check its version:

```
> avr-gcc --version
```

If the version number is under 3.4.6, you definitely have to deinstall it. If the version number is between 3.4.6 and 4.1.0 you may test it by compiling programs (see next chapter). If compiling fails you can proceed by deinstalling old versions and installing the avr-gcc version from the CD. The following chapter will refer to a recent version 4.1.1 (released March 2007) including some important patches that are usually also included in WinAVR.

Attention: please check the availability of the standard Linux development packages, e.g. GCC, make, binutils, libc, etc. before you start compiling and installing! Use your distributions package manager. Each Linux distribution should provide the required packages on the installation CD or alternatively you should be able to obtain the latest packages via Internet.

Please make sure you have the program "texinfo" installed. If the program is missing, you have to install it before proceeding with the installation – otherwise the installation process will fail!

Having completed this, you may now start the actual installation.

You have two options to chose from: either you compile and install all packages manually or you may use a simple automatic install script.

We suggest to try the script first and use manual install only if problems occur!

Attention: Please check whether you have enough free disk space! You will need more than 400MB of free space. More than 300MB of this data is only required temporarily for compiling, but may be removed later.

A number of install jobs require root rights and we suggest you should be logged in as root by "su" OR alternatively start critical jobs with "sudo" (as usual for the Ubuntu-distribution) or corresponding commands. The install script, mkdir in /usr/local/ directories and make install require root-rights.

Please pay attention to the CORRECT spelling of the following commands! Each and every symbol is meaningful and even if some of these commands may look awkward – these lines are perfectly correct and do not contain typing errors! (Of course you will still have to replace the string `<CD-ROM-drive>` by the name of your CD-ROM-drive device!).

All relevant installation files for avr-gcc, avr-libc and binutils can be found in the following directory:

```
<CD-ROM-Drive>:\Software\avr-gcc\Linux
```

Start by copying all install files to a directory on your hard disk – this is valid for both installation methods! In this case we will be using the Home directory (a standard shortcut for the Home directory is the swung dash or tilde-character: "~").

```
> mkdir ~/RP6
> cd <CD-ROM-Laufwerk>/Software/avr-gcc/Linux
> cp * ~/RP6
```

These files may be removed after successful installation to save disk space!

3.2.3.1. Automatic install script

After having set the script to executable by using chmod, you may proceed as follows:

```
> cd ~/RP6
> chmod -x avrgcc_build_and_install.sh
> ./avrgcc_build_and_install.sh
```

You may respond with "y" to the question, whether you are willing to install this configuration or not.

ATTENTION: The compile and install process will take some time depending on your system's performance (e.g. approximately 15 minutes on a 2GHz CoreDuo Notebook – slower systems may need some more time).

The script will also apply a few patches – these are labelled .diff-files in the directory.

Having completed the process you should see the following message:

```
(./avrgcc_build_and_install.sh)
(./avrgcc_build_and_install.sh) installation of avr GNU tools complete
(./avrgcc_build_and_install.sh) add /usr/local/avr/bin to your path to use the avr GNU tools
(./avrgcc_build_and_install.sh) you might want to run the following to save disk space:
(./avrgcc_build_and_install.sh)
(./avrgcc_build_and_install.sh) rm -rf /usr/local/avr/source /usr/local/avr/build
```

Then you may proceed as suggested by executing:

```
rm -rf /usr/local/avr/source /usr/local/avr/build
```

This command will delete all temporary files, which are not needed anymore.

You may now proceed to the next step and set the path environment variable to the avr-tools.

If the script ends up with some error message, you will have to read the error-messages carefully (and scroll up in the console!) – sometimes programs are missing, which will have to be installed in a preceding step (e.g. the previously mentioned program texinfo).

Before proceeding after an error message, we advise you to delete the previously generated files in the standard installation directory "/usr/local/avr". We even advise you to delete the complete directory.

If you are unsure what went wrong, please save all command line outputs into a file and send the relevant error description and the text file to the support team. Please send all the available information! This will make it easier to help you.

3.2.3.2. Manual install procedure

If you prefer manual install or the automated install script fails, you may proceed with the following steps.

The description has been derived from the following article:

`http://www.nongnu.org/avr-libc/user-manual/install_tools.html`

It can also be found as a PDF document in the AVR Libc Documentation on the CD:

```
<CD-ROM-Drive>:\Software\Documentation\avr-libc-user-manual-1.4.5.pdf
```

Please start on PDF page 240 (respectively 232 according to the page numbering system in the document).

This description is only a summary of this document, but we also install a few important patches – if you do not install these patches a few things will not work properly (like the very useful binary constants).

First of all we will have to create a directory, in which we are going to install all tools. The directory should be named: `/usr/local/avr`.

In the terminal enter the following commands as **ROOT**:

```
> mkdir /usr/local/avr
> mkdir /usr/local/avr/bin
```

It does not need to be this directory. We simply define a variable called `$PREFIX` for this directory:

```
> PREFIX=/usr/local/avr
> export PREFIX
```

Now we definitely have to add the definition to the PATH variable:

```
> PATH=$PATH:$PREFIX/bin
> export PATH
```

Binutils for AVR

We proceed by unpacking the source code for Binutils and applying a few patches. Let us assume you have copied all files to the home directory ~/RP6:

```
> cd ~/RP6
> bunzip2 -c binutils-2.17.tar.bz2 | tar xf -
> cd binutils-2.17
> patch -p0 < ../binutils-patch-aa.diff
> patch -p0 < ../binutils-patch-atmega256x.diff
> patch -p0 < ../binutils-patch-coff-avr.diff
> patch -p0 < ../binutils-patch-newdevices.diff
> patch -p0 < ../binutils-patch-avr-size.diff
> mkdir obj-avr
> cd obj-avr
```

Now execute the configure script:

```
> ../configure --prefix=$PREFIX --target=avr --disable-nls
```

This script analyzes what is available on your system and generates the required makefiles. At the end of this script you can compile and install everything:

```
> make
> make install
```

Depending on your PC's performance this will take a few minutes – this is also true for the next two steps – especially for the GCC!

GCC for AVR

Using similar procedure as for Binutils, the GCC has to be patched, compiled and installed:

```
> cd ~/RP6
> bunzip2 -c gcc-4.1.1.tar.bz2 | tar xf -
> cd gcc-4.1.1
> patch -p0 < ../gcc-patch-0b-constants.diff
> patch -p0 < ../gcc-patch-attribute_alias.diff
> patch -p0 < ../gcc-patch-bug25672.diff
> patch -p0 < ../gcc-patch-dwarf.diff
> patch -p0 < ../gcc-patch-libiberty-Makefile.in.diff
> patch -p0 < ../gcc-patch-newdevices.diff
> patch -p0 < ../gcc-patch-zz-atmega256x.diff
> mkdir obj-avr
> cd obj-avr
> ../configure --prefix=$PREFIX --target=avr --enable-languages=c,c++ \
      --disable-nls --disable-libssp –with-dwarf2
> make
> make install
```

By using a "\" you may press Enter and continue typing on the commandline – this allows you to split a line up and write an extremely long command line in several lines for better overview. Of course you may also omit this character and write the command as a single very long line.

AVR Libc

Finally the AVR libc:

```
> cd ~/RP6
> bunzip2 -c avr-libc-1.4.5.tar.bz2 | tar xf -
> cd avr-libc-1.4.5
> ./configure --prefix=$PREFIX --build=`./config.guess` --host=avr
> make
> make install
```

Attention: At –build=`./config.guess` you must pay attention to the "Accent grave" (à <-- the tiny stroke on top of the letter a! Do not use a normal apostrophe, as this will not work.

3.2.3.3. Setting the path

Now you have to make sure that the directory `/usr/local/avr/bin` is in your Path variable! Otherwise you will not be able to start the avr-gcc from the terminal and from makefiles. You have to add the avr-gcc path to the file `/etc/profile` or `/etc/environment` or similar files (this varies from distribution to distribution). You have to add the new path to the excisting string, separated by a ":" character. The line in the file may *more or less* look like this:

```
PATH="/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/usr/local/avr/bin"
```

Now you can check if the installation works by entering "avr-gcc --version" in a terminal as discussed in previous sections. If this gives proper response, installation was successful!

3.2.4. Java 6

The RP6Loader (see below for more information about it) has been designed for the Java environment and may be used in Windows and Linux (in theory, other operating systems such as OS X should work as well, but unfortunately AREXX Engineering does not support this so far). In order to run RP6Loader, you have to install a recent Java Runtime Environment (JRE). Maybe you already have it installed on your computer, but it should be at least Version 1.6 (= Java 6)! If you have not installed a recent JRE or JDK yet, please install SUN Microsystems' JRE 1.6 from the supplied CD or alternatively obtain a more recent version from the websites `http://www.java.com` or `http://java.sun.com`.

3.2.4.1. Windows

In a Windows environment, the JRE 1.6 is located in the directory:

```
<CD-ROM-Drive>:\Software\Java\JRE6\Windows\
```

Under Windows the Java installation is quite simple – just start the Setup and follow the instructions – done! You can skip the next section.

3.2.4.2. Linux

Most of the time installing Java in Linux environments is as easy as with Windows, but some distributions may require some manual work.

You can find the JRE6 as RPM (SuSE, RedHat etc.) and as a self-extracting archive ".bin" in this directory:

```
<CD-ROM-Drive>:\Software\Java\JRE6\
```

We advice you to search Java packages with the help of the specific distribution's package manager (search for "java", "sun", "jre" or "java6" ...) and to use these packages instead of the supplied ones on the CD! Please make sure that you install at least Java 6 (= JRE 1.6) or a newer version!

For Ubuntu or Debian the RPM Archiv will not be working anyway – here you have to use the package manager of your distribution. Other distributors like RedHat/Fedora, SuSE and others can use RPM if you can not use their package manager.

If the installation is not successful, you may still try to extract the JRE from the self extracting archive (.bin) in a directory on your hard disk (e.g. /usr/lib/Java6) and then set the JRE paths manually (PATH and JAVA_HOME etc.).

Please follow Sun's installation instructions, which can be found in the previously mentioned directory and on the Java Website!

To verify that everything works properly, please execute the command "`java -version`". The response should look like this:

```
java version "1.6.0"
Java(TM) SE Runtime Environment (build 1.6.0-b105)
Java HotSpot(TM) Client VM (build 1.6.0-b105, mixed mode, sharing)
```

If the response is different, your may have installed be obsolete or you may be running another Java VM at your system in parallel.

3.2.5. RP6Loader

We created the RP6Loader to simplify the uploading process for new programs to the RP6 and all expansion modules (as long as these modules provide a microcontroller with a compatible bootloader). Additionally we implemented a few useful functions, e.g. a simple serial terminal program.

You do not have to install the RP6Loader – instead you may simply copy the program somewhere to a new directory on your hard disk. The RP6Loader is located in a ZIP-archive on the RP6 CD-ROM:

```
<CD-ROM-Drive>:\Software\RP6Loader\RP6Loader.zip
```

Please unzip the file somewhere on your disk – e.g. in a new directory `C:\RP6\RP6Loader` (or similar). This directory contains the executable program RP6Loader.exe.

In fact, the real RP6Loader is located in the Java Archive (JAR) RP6Loader_lib.jar. Alternatively you would be able to start this RP6Loader from a command line window.

Windows:

```
java -Djava.library.path=".\lib" -jar RP6Loader_lib.jar
```

Linux:

```
java -Djava.library.path="./lib" -jar RP6Loader_lib.jar
```

The long –D option is required to enable the JVM to locate all necessary libraries. Usually you will not need this option and you just start the .exe-file to run the program. Linux uses a Shell Script "RP6Loader.sh", which needs to be set executable by issuing `chmod -x ./RP6Loader.sh`. This will allow you to start "./RP6Loader.sh" from a terminal or in Desktop Environments.

We recommend to create a link to RP6Loader on the desktop or the start menu. To do so, right click on RP6Loader.exe in Windows and select "Send to" --> "Desktop (Create a link)".

3.2.6. RP6 Library, RP6 CONTROL Library and Example programs

RP6Library and the corresponding example programs are located in a ZIP-archive on the supplied CD:

```
<CD-ROM-Drive>:\Software\RP6Examples\RP6Examples.zip
```

Extract this archive to a directory of your choice on your harddisk. We suggest to use a directory on a data partition. Alternatively you might use the "My Documents"-directory and create a subdirectory "RP6\Examples\" or use the Home directory in Linux.

We will discuss the example programs in detail later on in this manual!

The archive also provides examples for the RP6 CONTROL M32 expansion module including the corresponding library files!

3.3. Connecting the USB Interface – Windows

Linux users can skip this section!

There are several ways to install the USB Interface Drivers. The simplest way is installing the drivers BEFORE connecting the device for the first time. The CD provides different install programs for the driver.

For **32 and 64 Bit Windows XP, Vista, Server 2003 and 2000**:

```
<CD-ROM-Laufwerk>:\Software\USB_DRIVER\Win2k_XP_Vista\CDM_Setup.exe
```

Unfortunately there is no such comfortable setup program in **Win98SE/Me** – in this case you will have to install an older driver version manually after connecting the device to the PC (see below).

Just execute the CDM Installer program – the program will only show a short info dialog confirming the successful installation of the driver. That's all.

After the installation you may connect the USB Interface to your PC, *BUT PLEASE DO NOT CONNECT IT TO THE ROBOT, YET!* Just connect it to the PC with the USB cable! Please try to touch the PCB only at the sides or at the USB-plug, respectively at the plastic cover of the programming plug (see the safety instructions about static discharges)! Please avoid unnecessary touching of any of the components on the PCB, soldering pads or contacting elements of the covered plug to avoid static discharges! This is a general handling rule for all electronics equipment without covering.

The previously installed driver will be automatically assigned to the device and no further action is required. On Windows XP/2k system a few messages will pop up – the last message should say something like: "Hardware has been installed successfully and is now ready for use"!

If you connected the USB interface before installation of the driver (or if you are using Win98/Me) – don't worry. Windows will ask you for a driver, which can be found unpacked on the supplied CD as well. Windows will usually show a driver installation dialog. You are asked to specify the path to the driver. In Windows 2k/XP you have to select "manual install" before. Don't choose "search the web" or something like this, as the driver is located on the CD in the previously specified directories.

So simply select the driver directory for your Windows version and maybe a few additional files, which are not directly found by the system (all files are located in the same directories, which will be described in the following section)...

Usually Windows XP or newer will now proceed with a note, in which Microsoft warns you the driver has not been signed or verified – this is an irrelevant warning and you can confirm it without any risk. In this case the FTDI driver is signed and the system should not show a warning.

For **32 and 64 Bit Windows XP, Vista, Server 2003 and 2000** Systems:

```
<CD-ROM-Laufwerk>:\Software\USB_DRIVER\Win2k_XP_Vista\FTDI_CDM2.02.04\
```

For old **Windows 98SE/Me**:

```
<CD-ROM-Laufwerk>:\Software\USB_DRIVER\Win98SE_ME\FTDI_D2XX\
```

A few old Windows versions, e.g. Win98SE require a restart after the driver installation! **ATTENTION**: For **Win98/Me** you can only install one of the two driver versions, either D2XX or VCP (Virtual Comport)! There is no driver with both functions for these old systems. Usually no Virtual Comport will be available, because the standard Windows version of the RP6Loader is using the D2XX driver (This can be changed if required – you may contact our Support Team for help!).

3.3.1. Check if the device is properly connected

In order to check the correct connection of the device in Windows XP, Vista, 2003 and 2000 you may use either the RP6Loader or the Windows device manager:
Right click on My Computer --> Properties --> Hardware --> Device Manager
OR alternatively: Start --> Settings --> Control Panel --> Performance and Maintenance --> Hardware --> Device manager
Check the tree view for "Connections (COM and LPT)" for a "USB-Serial Port (COMX)" – in which X represents the port-number or check "USB-Controller" for a "USB Serial Converter" and make sure it is not any standard USB serial port adapter that may be connected to your computer.

3.3.2. Driver uninstall

If you should ever need to uninstall the driver *(No, please do not uninstall anything now – this is just for your information)*: If you have been using the CDM-installation software you may uninstall tools by selecting Start --> Settings --> Control Panel --> Software. The list should contain an entry "FTDI USB Serial Converter Drivers". Just select it and click on remove/uninstall!

If you installed the driver manually, you may execute the program "FTUNIN.exe" in the directory of the USB-driver on the CD!

Attention: Any USB-->RS232 Adapter using the FTDI chipsets might be using this driver as well!

3.4. Connecting the USB Interface – Linux

Windows users can skip this section!

The Linux Kernel 2.4.20 or higher already includes the required driver for our USB Interface FT232R (at least for the compatible predecessor, the FT232BM). The device will be recognized automatically and you will not have to do anything else. Just in case you run into trouble, you may obtain a Linux driver (and support including new releases of the drivers and documentation) directly from FTDI:

`http://www.ftdichip.com/`

Having connected the device to a Linux machine, you can check if the USB-Serial Port has been recognized properly by entering the command:

```
cat /proc/tty/driver/usbserial
```

That's all.

Just for your informatin: the Windows version of the RP6Loader uses the D2XX drivers and will display the complete USB names in the port list (e.g. "USB0 | RP6 USB Interface | serialNumber"). In contrast, the Linux version of the program will display the virtual comport names /dev/ttyUSB0, /dev/ttyUSB1 or similar. Additionally standard comport labels ("dev/ttyS0", etc.) will be displayed as well. In this case you will have try out which is the correct port!
Unfortunately Linux does not provide an easily installable driver for both functions and for this reason we prefer the usage of a Virtual Comport driver here, which is already included in the standard Linux kernels. Installation of the D2XX driver would require some manual work.

3.5. Finalizing Software installation

That was all you have to do for software and USB Interface setup!

Finally you may copy the most important files from the CD to your harddisk (especially the complete directory "Documentation" and "Examples", if not doned yet). Like this, you don't need to search for the CD all the time you need a specific file! The directories on the CD are named after the contained Software packages, so you can find everything easily.

If you should ever loose the CD, you can download all relevant Software from our Homepage. There you will also find the most up to date version which may include important bug fixes or new features.

3.6. Inserting Batteries

It's about time to get to the Robot itself. First of all the robot needs 6 batteries!

We recommend to use high-quality NiMH Mignon batteries (manufacturers e.g. Sanyo, Panasonic, and others) specifying a real world capacity of at least 2000mAh (optimal capacity is 2500mAh)! Please do not use standard alkaline batteries, which would turn out to be extremely expensive over time and also cause unnecessary environmental pollution.

We suggest to use **precharged batteries**! Always make sure to use batteries charged at the same level (all batteries charged or all discharged) and to use relatively new batteries! Batteries may wear out by storage time, by charging cycles, charging style and temperature. It's best to use new batteries instead of aged ones, which were laying around in the shelf for the last few years. It is also very important to use *nearly* equal battery cells only! Same capacity, same age, same charge level...

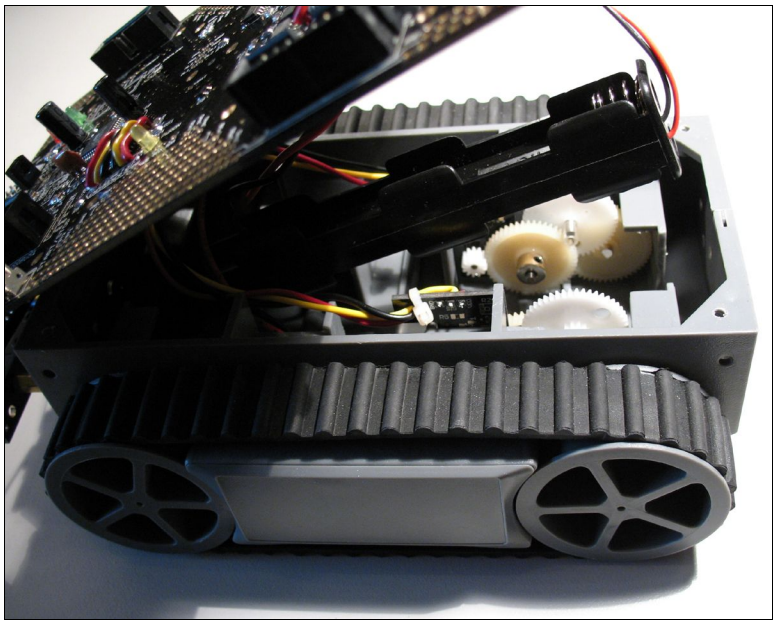
*If you prefer an external charger (highly recommended, but **not included** in the delivery!), you have to install batteries ONCE only! We highly recommend a microcontroller charging system, designed to optimally charge the batteries! For your own safety, please use certified and verified charging devices only!*

Not using an external charger equipped with a suitable adapter plug will require a rather time consuming procedure of removing discharged batteries from the system, re-charging and re-inserting them!

Inserting the batteries:

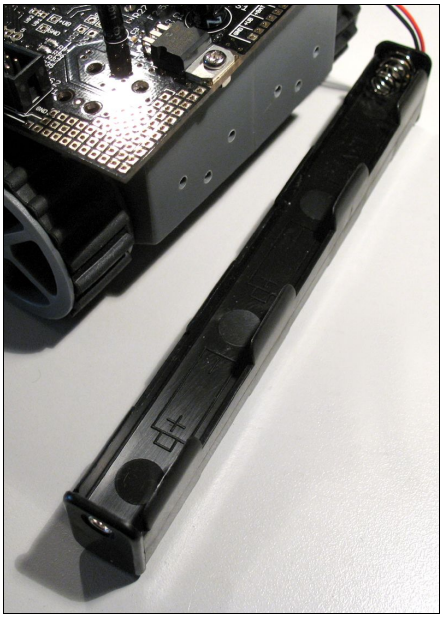
First of all you have to loosen the four screws fixing the mainboard (s. figure).

Now carefully lift the main board at the back side (see figure).



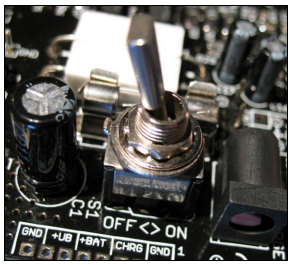
You do NOT need to unplug the tiny 3-pin connector of the bumper PCB (see fig.)! Be careful to touch the main board at the edges and at larger plastic parts only in order to avoid static discharges!

The main board is wired to the engines, the encoders and the battery holder by a bundle of soldered cables. Please move these cables – depending on their position – carefully out of the way.

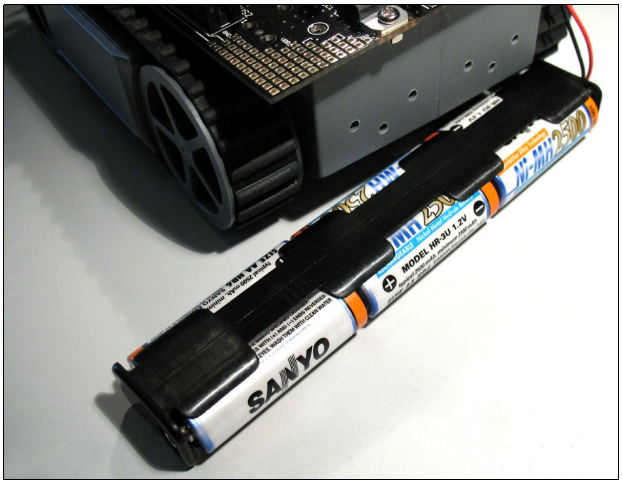


Please take out the black battery holder afterwards (see figure).

Make sure that the main power switch is in the position "OFF"! The switch lever must point to the direction of the text "OFF" and the large cylindrical capacitor on the main board (see figure)!



Before reactivating the robot, please check correct orientation of the batteries.

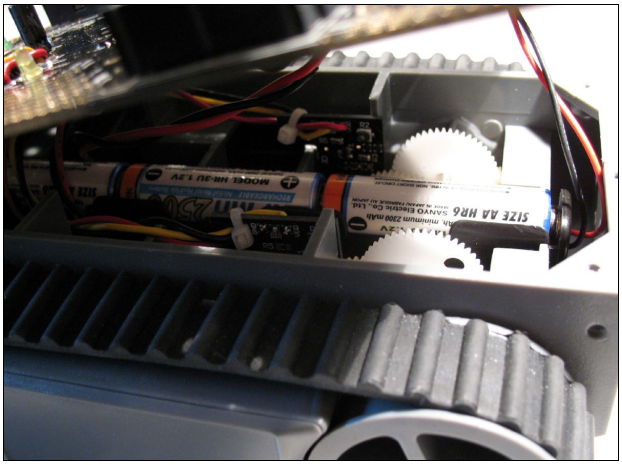


You may now insert 6 NiMH Batteries with **CORRECT ORIENTATION / POLARITY!**

CAUTION: The Fuse will blow if you insert the Batteries with wrong orientation!

In worst case, this may even damage parts of the electronics!

Thus you should better directly insert the Batteries in the correct way to avoid any problems! There are also markings in the Battery holder ((+) and (-), the negative terminal (the flat side) must point to the springs in the holder) to help you.



Check everything three times – just to be sure!

Now you can put the batteryholder back into the chassis. Take care of the cables! Avoid cables hanging around near the gears!

Having opened the robot anyway, you may now do a quick check of both gearing systems and the encoder wheels for transport damages or e.g. loosened bolts, screws and other components. Please **very carefully and slowly** turn the backside wheels ***for one revolution***!

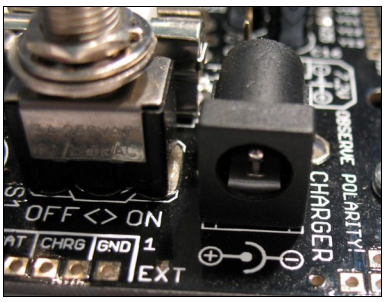
Turning half a revolution forward and backward would be enough already. You should be able to sense a remarkable resistance, but the wheel must be rotating freely. The gearwheels have to be moving freely! Please also have a look at appendix A!

The mainboard may now be put back onto the chassis. Move any cables between mainboard and plastic division bars or other chassis parts away by using your finger or a long screwdriver, in order to **locate the mainboard flat and even on the chassis!** Before fixing it again, please check for any cabling collisions between mainboard, chassis and gear-wheels! Now you may fix the main board by tightening the four screws – and we're done!

3.7. Charging the Batteries

If you did not install charged batteries already as we recommended it, you will now have to attach an external charger. Please turn the main power switch into position "OFF" for charging! Battery charging only works if the Robot is turned off. The main switch connects the batteries either to the RP6's electronics or to the charger connector.

Check the polarity of the charger compared to the charging supply plug (labelled "Charger") located next to the robot's main power switch!



You can see a polarity marking on the mainboard in front of the plug (see figure). **The negative terminal is located on the OUTER METAL PLATE and the positive terminal on the INNER pin!**

Charging time depends on the used charger and the batteries (Microprocessor controlled devices, e.g. Voltcraft 1A / 2A Delta Peak Fast chargers or Ansmann ACS110 / 410 will need between 3 and 4 hours, standard chargers e.g. AC48 require about 14 hours of charging time) – please read the details in the manual of the charging device!

Do not turn the robot's main power on while charging is in progress! Remove the charger before switching the robot on!

3.8. The first test

ATTENTION! Please read this and the following setion completely before perfoming the test!

If anything different from the following description happens, you should turn off the robot *immediately* and note exactly what went wrong! If the chapter "Troubleshooting" does not provide an answer, you may contact the support!

OK – ready to go! Turn the robot on! The two red status LEDs in the middle should light up. After a small delay they turn off, one of the other red LEDs (SL6) starts blinking and one of the green LEDs (SL1) is illuminated permanently. This indicates the absence of a user program in the controller's memory. If a user program is in the memory, only the green Status LED SL1 will be blinking.

The yellow PWRON LED should light up for about one second after turning the robot on – it saves energy to deactivate most sensors, e.g. the encoders.

After approx. 30 seconds, the red blinking LED SL6 and all other LEDs will turn off. The robot's microcontroller will automatically switch to standby mode as there is no user program to execute anyway. Standby mode may be terminated via the USB In-

terface, by pressing the Start/Stop Button or by shortly switching the robot off and on. Even in standby mode the robot uses a small amount of energy (up to 5mA) – and please remember to turn off the RP6 completely if you do not want to use the system for a longer time! With a program in memory the robot will not automatically switch to standby mode. Instead the system will continue waiting for user commands either from the serial interface (simply send an "s"), from the I²C Bus or the Start/Stop Button.

3.8.1. Connecting the USB Interface and start RP6Loader

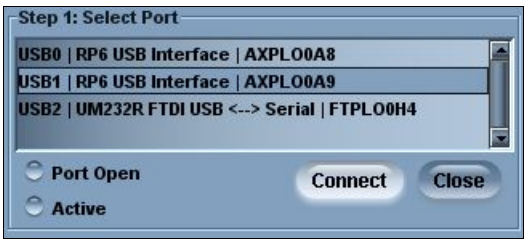
Next we will test a program upload with the USB interface. Please connect the USB interface to the PC **(always start by connecting it to the PC!)**. Then connect the USB interface to the "PROG/UART" connector of the Robot located directly beside the Start/Stop Button!

The connector has mechanical polarity protection and you cannot insert the 10-pin plug with reverse polarity unless you push it really hard.

Now start the RP6Loader.

Depending on selected languages the menus may be labelled different.

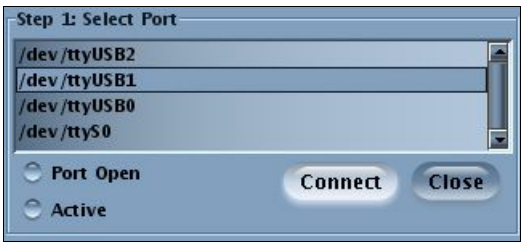
The screenshots show the English versions and you may alter the language if you want by selecting the menu "Options->Preferences", followed by "Language" (only English or German right now) and pressing OK. Having altered the language you have to restart the RP6Loader!



Open a port - <u>Windows</u>

You may now select the USB Port. As long as your PC does not provide another USB Serial Adapter with FTDI Controller, the port list will show only one single entry, which you can select. If there are several ports, you can identify the correct one by looking for "RP6 USB Interface" (or "FT232R USB UART"), followed by a pre-programmed serial number.

If no port is shown, please refresh the list by selecting the menu item "RP6Loader-->Refresh Port list"!



Open a port – <u>Linux</u>

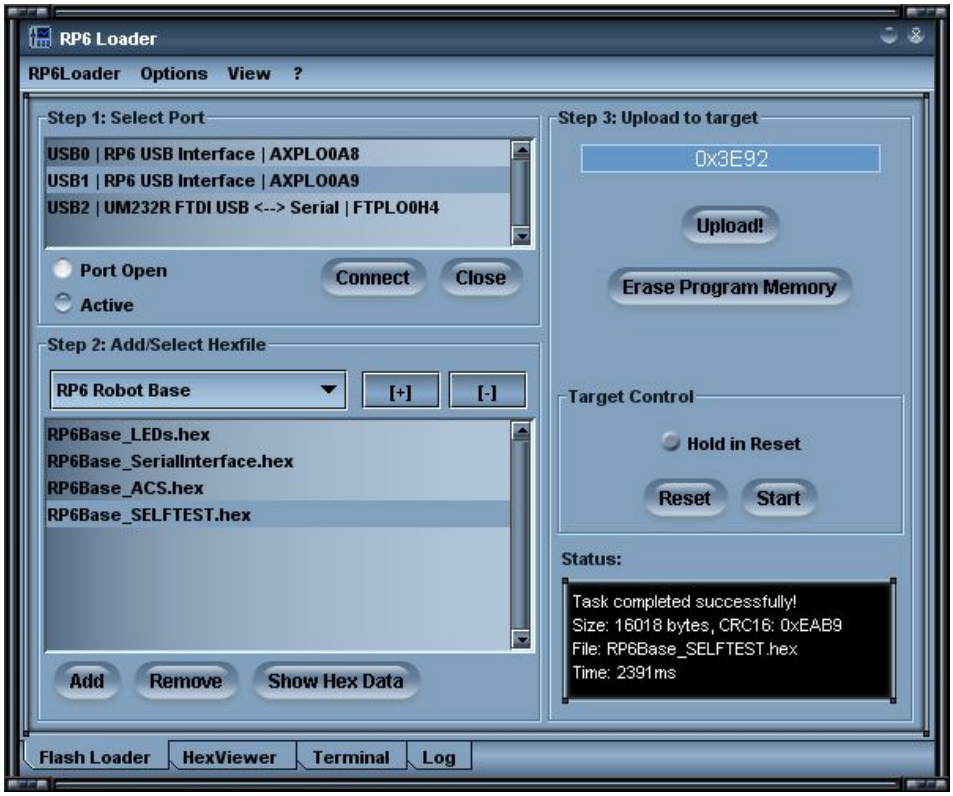
Linux handles the USB-Serial Adapter just like any other standard comport. Installing the D2XX drivers on a Linux-system is not that easy and modern Linux kernels already provide the standard Virtual Comport (VCP) drivers. In general, usage is similar to Windows, but you will have to try out which port actually is the RP6 USB interface and you should not remove the USB Port from the PC while the connection is open (otherwise you may have to restart the RP6Loader before you can open it again).

The Virtual Comports will be labelled "/dev/ttyUSBx", in which x represents a number, e.g. "/dev/ttyUSB0" or "/dev/ttyUSB1". Also the standard comports labelled "/dev/ttyS0", "/dev/ttyS1" will be shown as well.

The RP6Loader remembers the previously selected port and will automatically pre-select this port at program start (most of the presets and selections are remembered).

Now click on the "Connect"-button! The RP6Loader will try to open the port and check communication with the robot's Bootloader. If everything works OK, the black "Status" field will show "Connected to: RP6 Robot Base ...", accompanied the measured battery voltage. If this fails, please wait a second,and retry it! If the retry fails, a more serious error occurred! In this case immediately switch off the robot and proceed by reading the chapter "Troubleshooting" in the appendix!

At low battery voltage the program will show a warning message. Whenever you see this message, you have to recharge batteries. We advise recharging as soon as battery voltage drops below 5.9V!



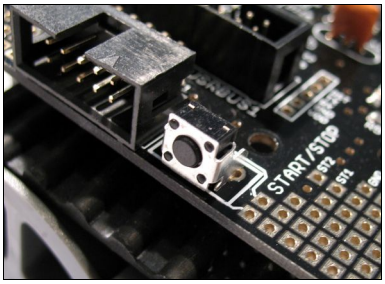
Having passed this initial check, you may start a simple self test program in order to verify that the robot's subsystems are working properly. First you have to add the Selftest program to the Hexfile list. This can be done by pressing the "Add" Button and selecting file "RP6Base_SELFTEST\RP6Base_SELFTEST.hex" in the example directory. The selected file contains the self test program in a hexadecimal format – that's why such files are called "Hexfiles".

The selected file will now appear in the list. This way you may select other Hexfiles from your own programs or from the other examples and add them to the list

(s. Screenshot, in which we already added a few hex files). The RP6Loader is able to manage several hexfiles in order to make the upload comfortable. With several expansion modules or different versions of programs you will appreciate this. At termination of the program the list will be saved automatically! Of course only path names for the hexfiles are stored. During program development, you need to add a hex file only once. After a recompilation of the program you can directly upload the new Version without adding it to the list again (you may use the shortcuts [STRG+D] for upload only or [STRG+Y] to start the program after a transfer). Path names vary for different operating systems, thus the RP6Loader uses separate file lists for Windows and Linux.

Now select the file "RP6Base_SELFTEST.hex" from the list and click on "Upload!" at the top right side below the progress bar. This starts the upload process to the MEGA32. The upload should be completed within a few seconds (up to 5 seconds for the self test program).

After the upload is complete, select the "Terminal"-tab at the bottom of the program window or alternatively select it from the menu "View".



Start the program by pressing the RP6's Start/Stop Button, located near the programming connector (see fig.)! Later you can use the buttons in RP6Loader Software or use the keyboard shortcut [STRG]+[S], but by using the hardware button now, you can directly verify that it works OK.

A warning message should appear in the Terminal. It tells you that the RP6 is going to start the motors during Test number 8!

ATTENTION! Please pick up and hold the RP6 in your hands while test number 8 is running ("Motors and Encoders Test") or alternatively place the RP6 on top of a suitable object – in order to prevent the caterpillar tracks from touching the ground surface! ***During test number 8 the caterpillars must NOT be touched or blocked!*** Otherwise the test will fail most likely! If the RP6 would touch the ground, the behaviour of the Motors would get influenced, resulting in a test failure. As a matter of fact the RP6 would also be driving some distance, forcing you to follow and carry the USB-cable as long as it lasts…

You have to hold the RP6 in your hands or alternatively place the RP6 on top of an object (e.g. a small box or remote control). Even if you place the RP6 on top of an object, please hold the RP6 with one hand during the test to prevent it from slipping away and accidentally falling of the table!

This warning message will be displayed directly before test number 8 and must be acknowledged before the test will start.

Please enter the lowercase letter 'x' in the terminal window and hit Enter (you will have to repeat this procedure whenever a similar message is displayed or a test has to be aborted...).

```
########################################################################
#########            RP6 Robot Base Selftest                #########
######### HOME VERSION                 v. 1.0 - 12.02.2007 #########
########################################################################
#####       Main Menu          #########      Advanced Menu      #####
#                                #                                    #
# 0 - Run ALL Selftests (0-8)    # s - Move at speed Test             #
# 1 - PowerOn Test               # d - Move distance Test             #
# 2 - LED Test                   # c - Encoder Duty-Cycle Test        #
# 3 - Voltage Sensor Test        #                                    #
# 4 - Bumper Test                #                                    #
# 5 - Light Sensor Test          #                                    #
# 6 - ACS (and RC5 receive) Test #                                    #
# 7 - IRCOMM/RC5 Test            #                                    #
# 8 - Motors and Encoders Test   #                                    #
#                                #                                    #
########################################################################
# Please enter your choice (1-8, s, d, c)!                           #
########################################################################
```

At this point the program will output the menu text shown on the left. The text may change a bit in future releases!

You can select and start the different test programs by entering the corresponding number or letter and hit Enter.

We want to run all standard tests – so type '0' and hit Enter!

The following output text will appear in the terminal window:

```
# 0

########################################################################
########################################################################
## Test #1 ##

### POWER ON TEST ###
Please watch the yellow PowerOn LED and verify that it lights up!
(it will flash a few times!)
```

Watch the yellow PowerON LED, which will flash a few times! If it does not flash, maybe the test was over before you looked at it or a real error occurred. The test program however will proceed, as there is no automatic method to detect the correct functionality of this – this is your job!

By the way the LED displays whether the encoders, the IR receiver and the current sensors are activated. Together with the LED, these devices consume a respectable amount of current - nearly 10mA --> to save power, we will only activate these devices if required.

The program now flashes all Status LEDs. A few times all LEDs together and then each of them alone. Here you can see if all LEDs are working correctly or if one of them is damaged.

The output looks like this:

```
## Test #2 ##

### LED Test ###
Please watch the LEDs and verify that they all work!
Done!
```

Battery sensor test is next. In fact, the sensor already has been tested as the RP6Loader has shown the battery voltage before. The battery check is now repeated to complete the list:

```
########################################################################
########################################################################
## Test #3 ##

### Voltage Sensor Test ###
Be sure that you are using good accumulators!

Enter "x" and hit return when you are ready!
```

Please acknowledge by entering 'x'!

```
# x
Performing 10 measurements:
Measurement #1: 07.20V --> OK!
Measurement #2: 07.20V --> OK!
Measurement #3: 07.20V --> OK!
Measurement #4: 07.20V --> OK!
Measurement #5: 07.20V --> OK!
Measurement #6: 07.20V --> OK!
Measurement #7: 07.20V --> OK!
Measurement #8: 07.20V --> OK!
Measurement #9: 07.20V --> OK!
Measurement #10: 07.20V --> OK!
Done!
```

This is like the output should look like in general – values may vary in the acceptable range of 5.5 up to 9.5V. An error is shown if values are out of these limits. If an error occurs, please check the batteries – they may not have been charged properly or are defect! If batteries are OK, then the sensor (two resistors...) might be damaged.

We will now check the bumpers. In order to test them you have to press the microswitches and observe the LEDs and the displayed messages in the terminal. Each "bump" has to be shown in the terminal and with the LEDs. The output message displays:

```
## Test #4 ##

Bumper Test
Please hit both bumpers and verify
that both Bumpers are working properly!
The Test is running now. Enter "x" and hit return to stop this test!
OBSTACLE: LEFT!
FREE: LEFT!
OBSTACLE: RIGHT!
FREE: RIGHT!
OBSTACLE: LEFT!
FREE: LEFT!
OBSTACLE: RIGHT!
FREE: RIGHT!
OBSTACLE: LEFT!
OBSTACLE: RIGHT!
FREE: LEFT!
FREE: RIGHT!
```

If this was successful, you may quit the test by typing 'x' + Enter

Now we will check the light sensor. In order to test these sensors, please cover each of them subsequently with one hand, move your hand close to these sensors and check for changes of the measured values and the LEDs – decreasing light intensity must result in decreasing measurement values! The LEDs will display which sensor is sensing brighter light. Usually, daylight produces values ranging from 200 to 900.

If you point a powerful torch at the sensors and illuminate them directly or if you hold the robot into bright sunlight, measurement values may rise to over 1000. In a rather dark room, values should be below 100.

Start the test by typing 'x' + Enter:

```
## Test #5 ##

### Light Sensor Test ###
Please get yourself a small flashlight!
While the test runs, move it in front of the Robot
and watch if the values change accordingly!
```

```
Enter "x" and hit return when you are ready!
# x
The Test is running now. Enter "x" and hit return to stop this test!
Performing measurements...:
Left: 0510,  Right: 0680
Left: 0511,  Right: 0679
Left: 0512,  Right: 0680
Left: 0560,  Right: 0710
Left: 0630,  Right: 0750
Left: 0640,  Right: 0760
Left: 0644,  Right: 0765

[...]
```

After testing the sensors please abort this test sequence by entering 'x'!

We now proceed with the ACS Test. There is nothing to confirm and the test will start immediately. Now waive a hand or an obstacle in front of the robot, but take care to clear a large area in front of the robot to prevent detection of other objects.

The test may show the following output:

```
## Test #6 ##

ACS Test
Please move your hand or other obstacles in front of the Robot
and verify that both ACS channels are working properly!

ACS is set to Medium power/range!

You can also send RC5 Codes with a TV Remote Control
to the RP6 - it will display the Toggle Bit, Device Adress
and Keycode of the RC5 Transmission!
Make sure your remote control transmits in RC5 and not
SIRCS or RECS80 etc.! There are several other formats that will NOT work!

The Test is running now. Enter "x" and hit return to stop this test!
OBSTACLE: LEFT!
FREE: LEFT!
OBSTACLE: LEFT!
FREE: LEFT!
OBSTACLE: LEFT!
OBSTACLE: RIGHT!
FREE: RIGHT!
FREE: LEFT!
OBSTACLE: LEFT!
OBSTACLE: RIGHT!
FREE: RIGHT!
FREE: LEFT!
```

The test also allows you to receive messages from RC5 compatible IR remote controls. In this case the received Toggle bit, Address and Key code will be shown.

To continue, please abort this test by entering 'x'!

Next one is the IRCOMM test procedure, which can be startet by entering 'x'. The procedure starts transmitting IR data-packets, displays received packets in the terminal and automatically checks if the received data is OK (using rather powerful IR-diodes, the IRCOMM usually will receive it's own signals back. Only in the absence of any reflecting objects or a ceiling the system may eventually fail – but this would be a very unusual condition).

The output should look like this:

```
#### TEST #7 ####

IRCOMM Test
[...]

TX RC5 Packet: 0
RX RC5 Packet --> Toggle Bit:0 | Device Address:0 | Key Code:0 --> OK!
TX RC5 Packet: 3
RX RC5 Packet --> Toggle Bit:0 | Device Address:3 | Key Code:3 --> OK!
TX RC5 Packet: 6
RX RC5 Packet --> Toggle Bit:0 | Device Address:6 | Key Code:6 --> OK!
TX RC5 Packet: 9
RX RC5 Packet --> Toggle Bit:0 | Device Address:9 | Key Code:9 --> OK!
TX RC5 Packet: 12
RX RC5 Packet --> Toggle Bit:0 | Device Address:12 | Key Code:12 --> OK!
[...]
TX RC5 Packet: 57
RX RC5 Packet --> Toggle Bit:1 | Device Address:25 | Key Code:57 --> OK!
TX RC5 Packet: 60
RX RC5 Packet --> Toggle Bit:1 | Device Address:28 | Key Code:60 --> OK!
TX RC5 Packet: 63
RX RC5 Packet --> Toggle Bit:1 | Device Address:31 | Key Code:63 --> OK!

Test finished!
Done!
```

The Test should take about 5 seconds.

Finally we will proceed with the Motor and Encoder Test! You must pick up the RP6 with your hands – the caterpillars must not touch the floor or any other objects!

Otherwise the test will most likely fail! If you put the RP6 on top of an object like described above, please make sure the RP6 cannot accidently fall off the table.

This test will not take too long – approximately 30 seconds. Carefully check for error messages in this test! It may happen that a single measurement fails, causing the test sequence to end with an error message. If the motors are starting as expected and the test is aborted somewhere in the middle, there is nothing to worry about. If this happens, please retry – after reading the "Troubleshooting"-chapter in the appendix!

The test procedure will ramp up both motor speeds up to 50% of the maximum speed and will alternate the turning direction of the motors a few times. The system will constantly be checking and supervising measurement values from encoders and current sensors. If something got damaged during the transport (e.g. a short circuit in one of the motors or a blocked gear – which should have been noticed in the previous testing phase after inserting the batteries) the monitored current values will rise to high levels and cause the test to be aborted immediately.

Sample test report (abbreviated):

```
######################################################################
######################################################################
#### TEST #8 ####

Automatic speed speed regulation test

######################################################################
### ATTENTION!!! DANGER!!! WARNING!!!
Make sure that the RP6 can __NOT__ move!
The caterpillar tracks should __NOT__ touch the ground!
(hold it in your hands for example...)
THE RP6 WILL START MOVING FAST! YOU CAN DAMAGE IT IF YOU DO NOT
MAKE SURE THAT IT CAN __NOT__ MOVE!
Make sure both crawler tracks are FREE RUNNING! DO NOT BLOCK THEM!
--> OTHERWISE THE TEST WILL FAIL!
######################################################################

Enter "x" and hit return when TO START THIS TEST!
Make sure the RP6 can not move!

# x
T: 000 |VL: 000 |VR: 000 |PL: 000 |PR: 000 |IL: 000 |IR: 003 |UB: 07.28V
T: 000 |VL: 000 |VR: 000 |PL: 000 |PR: 000 |IL: 002 |IR: 003 |UB: 07.28V
[...]
Speed Left: OK
Speed Right: OK
T: 020 |VL: 000 |VR: 000 |PL: 000 |PR: 000 |IL: 000 |IR: 003 |UB: 07.28V
T: 020 |VL: 000 |VR: 000 |PL: 000 |PR: 000 |IL: 000 |IR: 003 |UB: 07.28V
T: 020 |VL: 000 |VR: 000 |PL: 000 |PR: 000 |IL: 000 |IR: 003 |UB: 07.28V
T: 020 |VL: 000 |VR: 000 |PL: 020 |PR: 020 |IL: 006 |IR: 009 |UB: 07.26V
T: 020 |VL: 001 |VR: 014 |PL: 039 |PR: 030 |IL: 020 |IR: 020 |UB: 07.27V
[...]
Speed Left: OK
Speed Right: OK
T: 040 |VL: 021 |VR: 019 |PL: 037 |PR: 028 |IL: 025 |IR: 021 |UB: 07.25V
T: 040 |VL: 020 |VR: 020 |PL: 037 |PR: 029 |IL: 026 |IR: 022 |UB: 07.25V
T: 040 |VL: 018 |VR: 020 |PL: 044 |PR: 036 |IL: 028 |IR: 023 |UB: 07.23V
T: 040 |VL: 038 |VR: 038 |PL: 055 |PR: 044 |IL: 035 |IR: 029 |UB: 07.23V
T: 040 |VL: 037 |VR: 042 |PL: 055 |PR: 043 |IL: 033 |IR: 028 |UB: 07.24V
T: 040 |VL: 043 |VR: 041 |PL: 052 |PR: 042 |IL: 032 |IR: 026 |UB: 07.23V
T: 040 |VL: 043 |VR: 041 |PL: 052 |PR: 040 |IL: 030 |IR: 024 |UB: 07.24V
T: 040 |VL: 037 |VR: 041 |PL: 052 |PR: 040 |IL: 030 |IR: 023 |UB: 07.24V
T: 040 |VL: 043 |VR: 040 |PL: 050 |PR: 039 |IL: 029 |IR: 022 |UB: 07.24V
Speed Left: OK
Speed Right: OK
T: 060 |VL: 040 |VR: 039 |PL: 053 |PR: 040 |IL: 033 |IR: 024 |UB: 07.24V
T: 060 |VL: 036 |VR: 040 |PL: 053 |PR: 040 |IL: 034 |IR: 026 |UB: 07.24V
T: 060 |VL: 042 |VR: 039 |PL: 052 |PR: 041 |IL: 034 |IR: 027 |UB: 07.23V
T: 060 |VL: 042 |VR: 040 |PL: 063 |PR: 052 |IL: 038 |IR: 032 |UB: 07.22V
T: 060 |VL: 058 |VR: 060 |PL: 068 |PR: 056 |IL: 038 |IR: 032 |UB: 07.25V
T: 060 |VL: 062 |VR: 062 |PL: 067 |PR: 054 |IL: 037 |IR: 029 |UB: 07.22V
T: 060 |VL: 060 |VR: 062 |PL: 067 |PR: 053 |IL: 038 |IR: 028 |UB: 07.23V

[...]

Speed Left: OK
Speed Right: OK
T: 100 |VL: 082 |VR: 078 |PL: 080 |PR: 068 |IL: 043 |IR: 036 |UB: 07.23V
T: 100 |VL: 079 |VR: 079 |PL: 081 |PR: 069 |IL: 047 |IR: 038 |UB: 07.22V
T: 100 |VL: 078 |VR: 082 |PL: 092 |PR: 078 |IL: 049 |IR: 039 |UB: 07.23V
T: 100 |VL: 095 |VR: 099 |PL: 101 |PR: 082 |IL: 055 |IR: 039 |UB: 07.20V
T: 100 |VL: 098 |VR: 100 |PL: 109 |PR: 081 |IL: 056 |IR: 040 |UB: 07.19V
T: 100 |VL: 095 |VR: 099 |PL: 111 |PR: 082 |IL: 062 |IR: 042 |UB: 07.19V
T: 100 |VL: 102 |VR: 101 |PL: 111 |PR: 082 |IL: 058 |IR: 041 |UB: 07.21V
T: 100 |VL: 102 |VR: 101 |PL: 109 |PR: 081 |IL: 056 |IR: 039 |UB: 07.20V
T: 100 |VL: 093 |VR: 100 |PL: 113 |PR: 081 |IL: 063 |IR: 038 |UB: 07.20V
T: 100 |VL: 104 |VR: 099 |PL: 112 |PR: 082 |IL: 056 |IR: 042 |UB: 07.22V
Speed Left: OK
Speed Right: OK
T: 080 |VL: 086 |VR: 071 |PL: 022 |PR: 000 |IL: 020 |IR: 012 |UB: 07.28V
T: 080 |VL: 000 |VR: 000 |PL: 000 |PR: 000 |IL: 001 |IR: 003 |UB: 07.28V
T: 080 |VL: 004 |VR: 011 |PL: 088 |PR: 084 |IL: 051 |IR: 045 |UB: 07.21V
T: 080 |VL: 079 |VR: 101 |PL: 103 |PR: 077 |IL: 064 |IR: 039 |UB: 07.21V
```

```
T: 080 |VL: 082 |VR: 076 |PL: 098 |PR: 072 |IL: 061 |IR: 041 |UB: 07.19V
T: 080 |VL: 081 |VR: 081 |PL: 096 |PR: 071 |IL: 055 |IR: 040 |UB: 07.20V
T: 080 |VL: 080 |VR: 082 |PL: 095 |PR: 070 |IL: 057 |IR: 038 |UB: 07.21V
T: 080 |VL: 082 |VR: 080 |PL: 094 |PR: 069 |IL: 058 |IR: 036 |UB: 07.22V
T: 080 |VL: 077 |VR: 080 |PL: 095 |PR: 069 |IL: 056 |IR: 036 |UB: 07.23V
Speed Left: OK
Speed Right: OK
T: 060 |VL: 082 |VR: 079 |PL: 095 |PR: 069 |IL: 054 |IR: 038 |UB: 07.22V
T: 060 |VL: 079 |VR: 079 |PL: 095 |PR: 071 |IL: 058 |IR: 040 |UB: 07.21V
T: 060 |VL: 082 |VR: 081 |PL: 093 |PR: 070 |IL: 056 |IR: 039 |UB: 07.19V
T: 060 |VL: 069 |VR: 070 |PL: 080 |PR: 054 |IL: 048 |IR: 029 |UB: 07.23V
T: 060 |VL: 064 |VR: 059 |PL: 075 |PR: 054 |IL: 046 |IR: 029 |UB: 07.22V
T: 060 |VL: 058 |VR: 057 |PL: 075 |PR: 055 |IL: 043 |IR: 032 |UB: 07.24V
T: 060 |VL: 059 |VR: 059 |PL: 075 |PR: 056 |IL: 046 |IR: 034 |UB: 07.23V
T: 060 |VL: 060 |VR: 059 |PL: 075 |PR: 056 |IL: 046 |IR: 035 |UB: 07.23V
T: 060 |VL: 057 |VR: 060 |PL: 076 |PR: 056 |IL: 047 |IR: 033 |UB: 07.22V
T: 060 |VL: 058 |VR: 061 |PL: 077 |PR: 055 |IL: 045 |IR: 030 |UB: 07.23V
Speed Left: OK
Speed Right: OK
T: 040 |VL: 045 |VR: 035 |PL: 043 |PR: 023 |IL: 027 |IR: 018 |UB: 07.24V
T: 040 |VL: 000 |VR: 000 |PL: 011 |PR: 000 |IL: 013 |IR: 007 |UB: 07.28V
T: 040 |VL: 002 |VR: 000 |PL: 038 |PR: 038 |IL: 015 |IR: 014 |UB: 07.24V
T: 040 |VL: 038 |VR: 061 |PL: 059 |PR: 052 |IL: 035 |IR: 035 |UB: 07.24V
T: 040 |VL: 044 |VR: 043 |PL: 057 |PR: 044 |IL: 035 |IR: 028 |UB: 07.23V
T: 040 |VL: 038 |VR: 039 |PL: 057 |PR: 044 |IL: 035 |IR: 027 |UB: 07.24V
T: 040 |VL: 039 |VR: 042 |PL: 055 |PR: 043 |IL: 033 |IR: 025 |UB: 07.23V
T: 040 |VL: 043 |VR: 041 |PL: 053 |PR: 041 |IL: 032 |IR: 023 |UB: 07.24V
T: 040 |VL: 040 |VR: 041 |PL: 054 |PR: 041 |IL: 032 |IR: 023 |UB: 07.25V
Speed Left: OK
Speed Right: OK
T: 020 |VL: 037 |VR: 040 |PL: 054 |PR: 041 |IL: 031 |IR: 024 |UB: 07.24V
T: 020 |VL: 022 |VR: 019 |PL: 022 |PR: 012 |IL: 017 |IR: 016 |UB: 07.28V
T: 020 |VL: 000 |VR: 000 |PL: 000 |PR: 000 |IL: 004 |IR: 007 |UB: 07.28V
T: 020 |VL: 000 |VR: 006 |PL: 030 |PR: 027 |IL: 020 |IR: 020 |UB: 07.24V
T: 020 |VL: 013 |VR: 019 |PL: 043 |PR: 030 |IL: 029 |IR: 022 |UB: 07.24V
T: 020 |VL: 026 |VR: 020 |PL: 038 |PR: 029 |IL: 027 |IR: 022 |UB: 07.24V
T: 020 |VL: 020 |VR: 021 |PL: 038 |PR: 029 |IL: 028 |IR: 023 |UB: 07.25V
T: 020 |VL: 021 |VR: 020 |PL: 038 |PR: 029 |IL: 028 |IR: 023 |UB: 07.24V
T: 020 |VL: 018 |VR: 019 |PL: 038 |PR: 030 |IL: 027 |IR: 024 |UB: 07.24V
T: 020 |VL: 022 |VR: 020 |PL: 037 |PR: 029 |IL: 027 |IR: 023 |UB: 07.23V
Speed Left: OK
Speed Right: OK

***** MOTOR AND ENCODER TEST OK! *****
```

The measurement values reported in this test are (from left to right): T - desired speed, VL/VR – measured speed left/right, PL/PR – PWM value left/right, IL/IR – motor current left/right, UB – battery voltage.

If the output values look similar to the above report – everything is OK.

If things do not work properly and error messages appear, please read the "Troubleshooting"-chapter in the appendix!

That's it. If all systems passed the test, you may now continue with the next chapter.

4. Programming the RP6

At last, we have reached the programming section.

4.1. Configuring the source code Editor

We will start by setting up a small development environment. The so-called "source code" of our C programs needs to be entered and edited somehow!

Of course we will not use text processing systems like OpenOffice or Word! This may not be obvious for everyone and therefore we explicitly emphasize this. Office Software may perfectly enable you to write a manual like this one, but they are completely inadequate for software development. Source code is just plain text – without any formatting. Text sizes, fonts and colours are meaningless to the compiler...

However, for human beings, automatic coloured highlighting of special keywords or text passages (e.g. comments) can be very helpful. This and serveral other features are included in Programmers Notepad 2 (we will simply call it "PN2" in the following chapters), which will be used as our source code editor (*ATTENTION: Linux users will need to use another, similar editor. Usually Linux provides several pre-installed editors, e.g. kate, gedit, exmacs and others).*

Apart from highlighting keywords and equivalent structures (this is so-called "syntax highlighting") the editor offers rudimentary project mangement. You may organize a bundle of source files in a project. Additionally PN2 allows you to comfortably call programs like AVR-GCC to compile programs by clicking on a single menu entry. AVR-GCC is a plain command line program, lacking any kind of graphical interface...

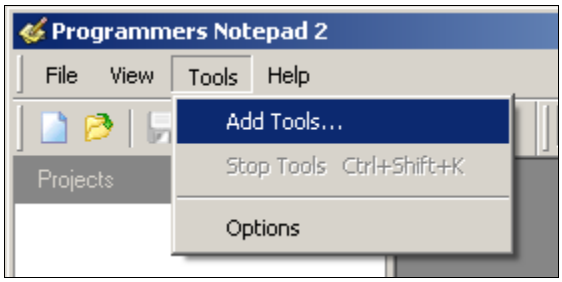
The most recent version of Programmers Notepad can be found at the project's homepage:

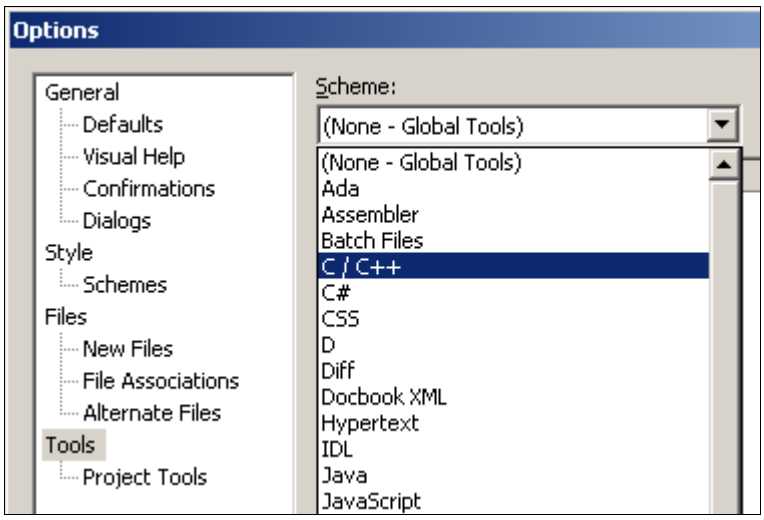
`http://www.pnotepad.org/`

4.1.1. Creating menu entries

ATTENTION: You may skip this chapter if PN2 already provides menu-entries. (These menu-entries labelled "[WinAVR] Make All", etc.. can be found in the menu. Please check the menu for these entries). This is not included in all versions of PN2. And you might be interested to add other programs to the menu.

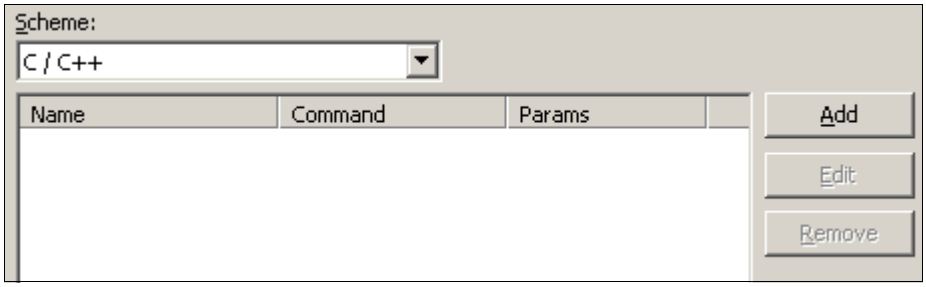
Start PN2 and select "Add Tools..." in the menu "Tools" (see Screenshot).



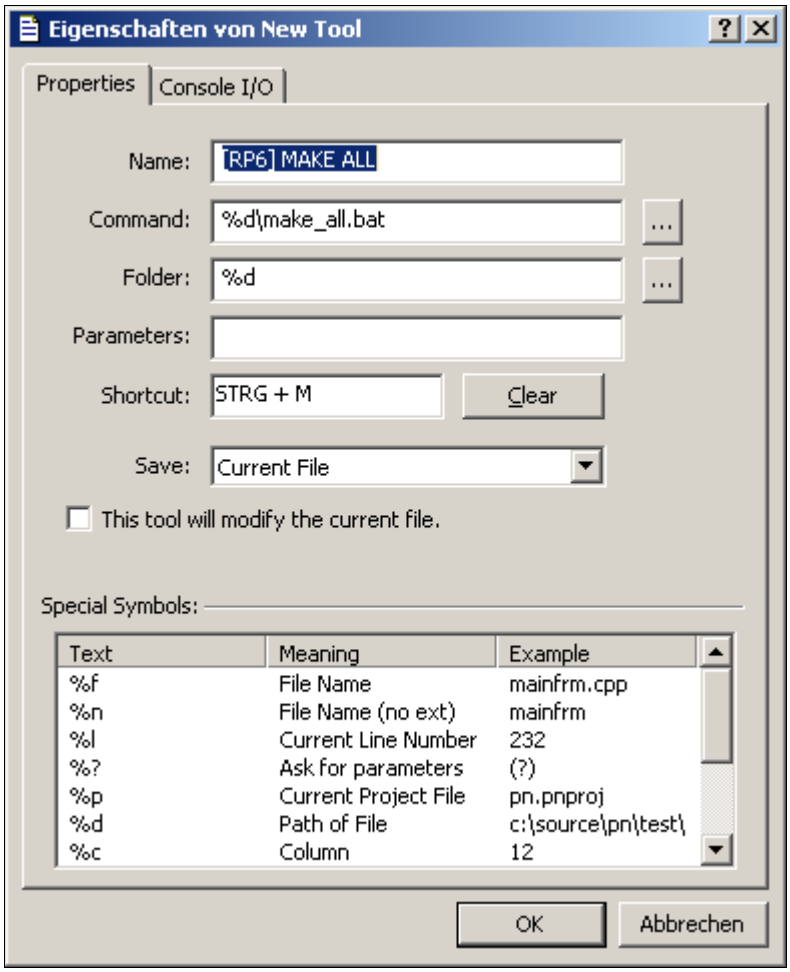


You are now entering the options dialog, which allows you to change several settings. However we will only add new entries to the tools menu.

To proceed, select "C/C++" in the dropdown list to the "Scheme:"-menu!



Click on "Add"!



The dialog on the left should appear.

Please exactly enter the things you see on the screenshot.

The phrase "%d" refers to the directory of the selected file and "%d\make_all.bat" refers to the batch file, which can be found in any of the sample RP6 projects.

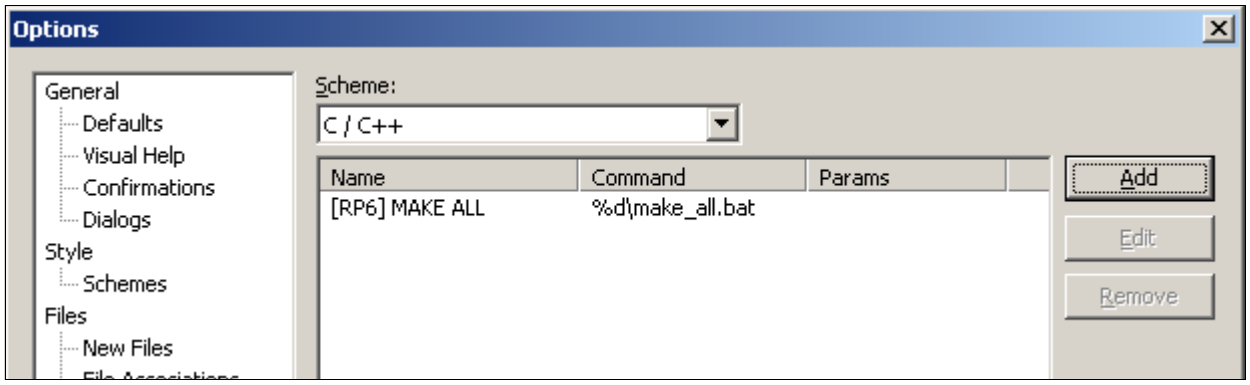
As an example of a Shortcut entry you may enter [STRG] + [M] with the keyboard!

This entry will start the "make" tool by calling the "make_all.bat" batchfile, which initiates the compilation job of files in the corresponding directory of the selected file. We will discuss this method in the following chapters.

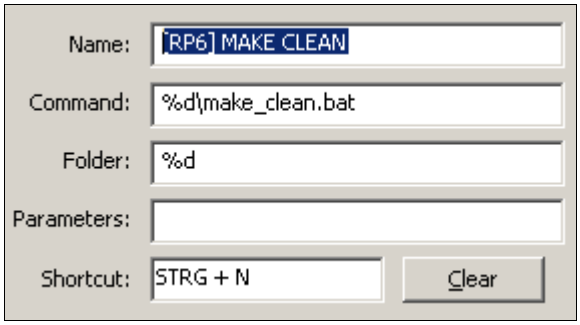
As an alternative method to "%d/make_all.bat" you may also simply enter "make" into the field "Command" and "all" into the "Parameters" field.

In fact, the batch file simply executes exactly these commands, but the batch file simplifies starting the compiler from Windows Explorer.

Now click OK – and a new entry will be displayed in the list:



...click on "Add" once again!



Just like for make all, you now have to enter all you see on the screenshot and click on OK.

This will create a new entry in the List: "[RP6] MAKE CLEAN"

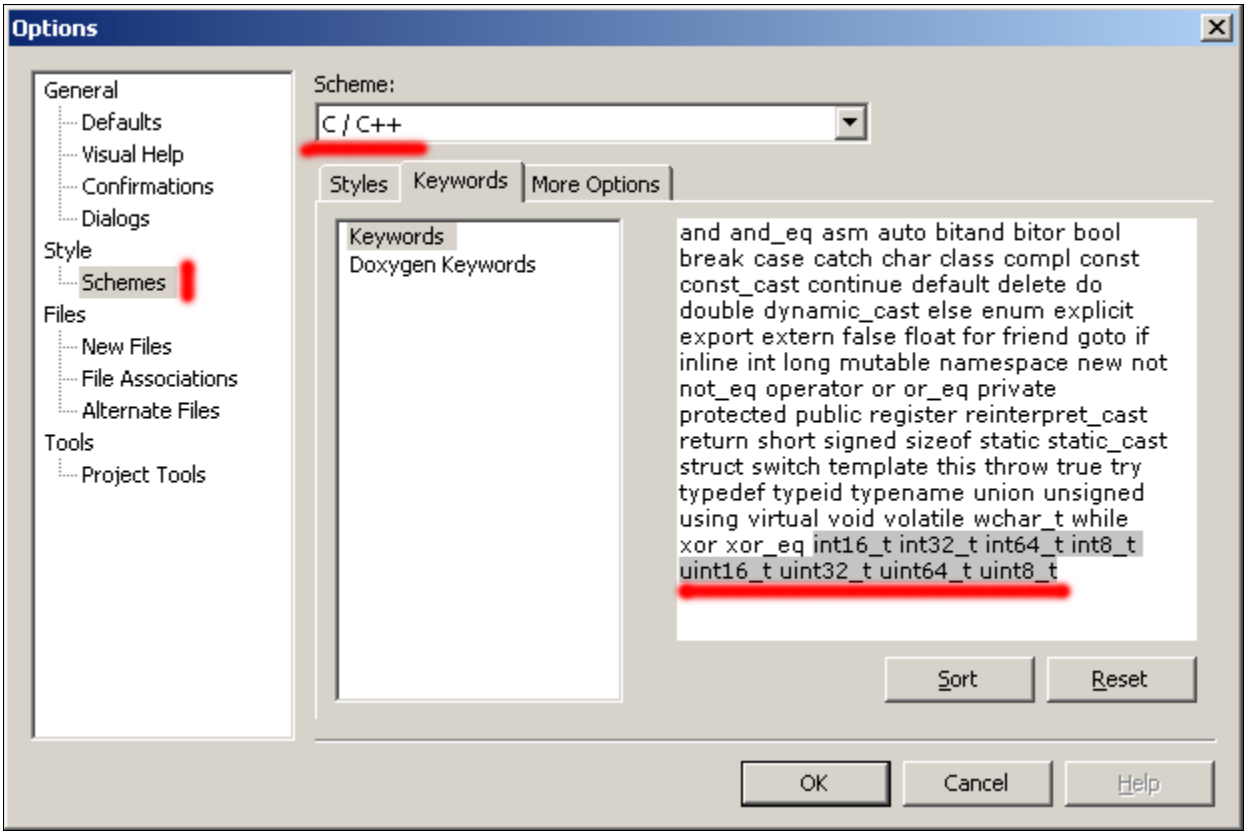
This entry allows you to comfortably delete all temporary files, which are generated during the compilation process. Usually we will not need those temporary files after successful compilation. By the way: the generated hexfile will not be deleted and may still be transferred to the robot.
As mentioned before (alternatively to "%d/make_clean.bat") you may also enter "make" into the "Command" field and "clean" into the "Parameters" field.

Leave the options menu by clicking "OK".

4.1.2. Configure Syntax Highlighting

Another setting you may change is the Syntax Highlighting. You can add a few "Keywords" to the standard C/C++ Scheme. You may directly Copy & Paste ([STRG]+[C] = copy, [STRG]+[V] = paste/insert) them into the dialog-field:

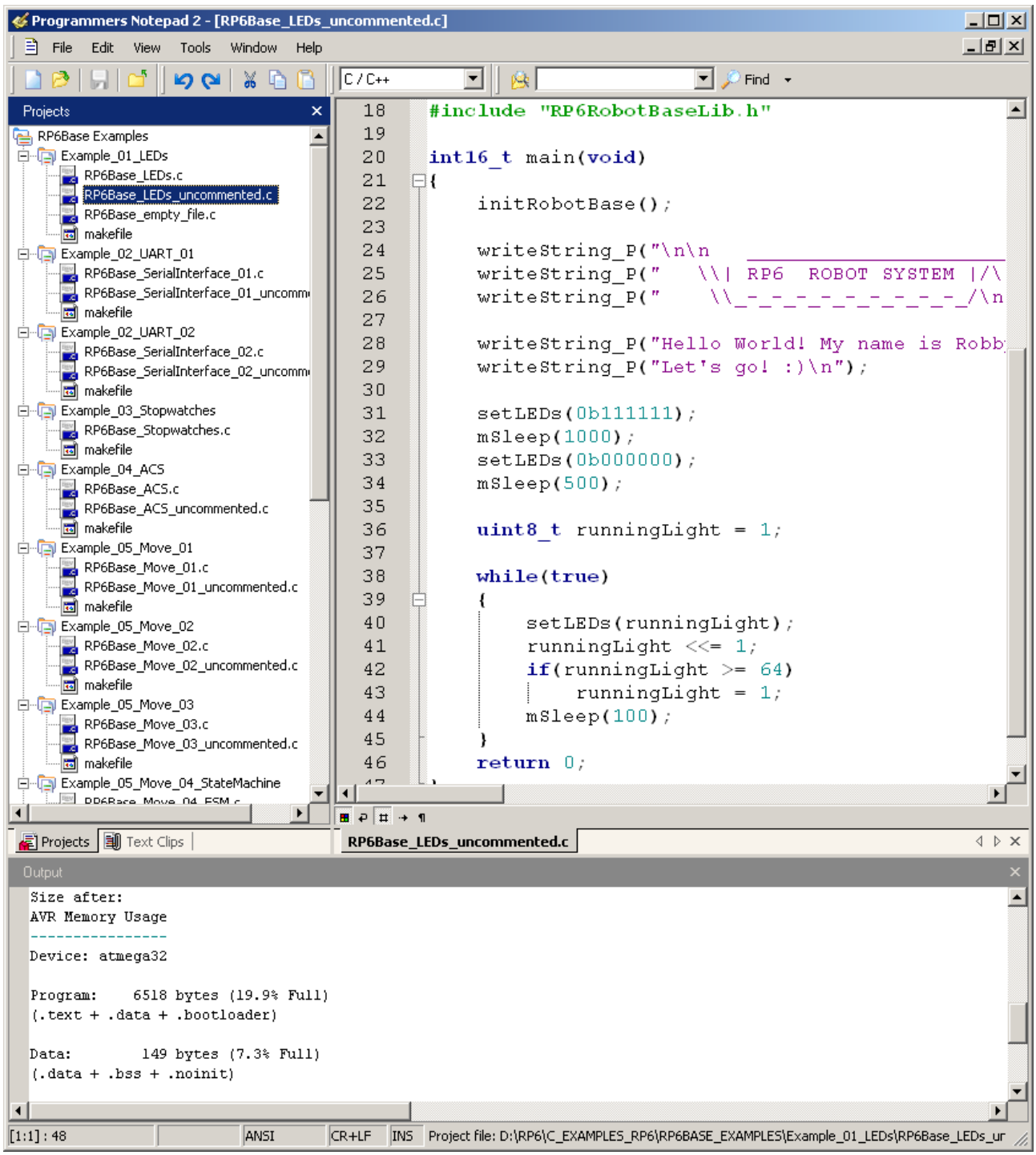
int8_t int16_t int32_t int64_t uint8_t uint16_t uint32_t uint64_t



Then click "Sort" and OK!

Attention: *Later versions of WinAVR and Programmers Notepad (WinAVR-20070525) already include these Keywords in Programmers Notepad! If you see these keywords already in the list, you do not have to modify these settings! These latest versions of Programmers Notepad will also look slightly different from the screenshots in this manual.*

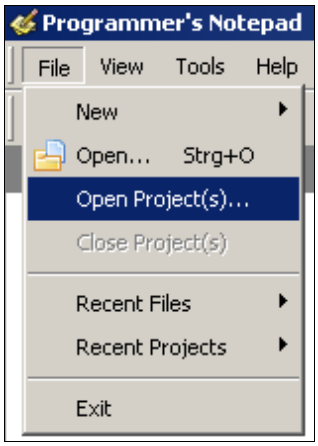
After customizing and opening a sample project *according to the next section* PN2 should look like in the following screenshot:



On the left side you see all sample projects in a treeview, the the source editor (featuring the previously discussed syntax highlighting) is on the right and and the tool output (in this case the compiler output) is on the bottom.

You can customize PN2 in various other ways and it provides a great number of useful functions.

4.1.3. Opening and compiling sample projects



Let's try if everything works fine and open all the sample projects:

Select "Open Project(s)" in the "File" menu.

In the standard file select dialog, you have to search for the subdirectory "RP6Base_Examples" in the directory of the example programs.

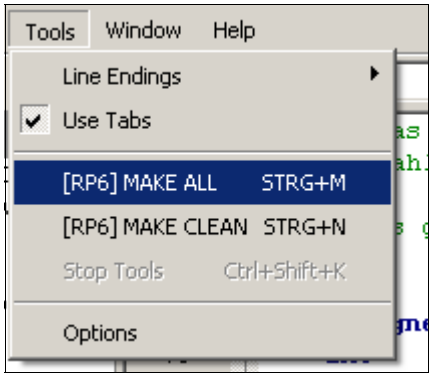
Open the file "RP6BaseExamples.ppg", which is a PN2 project group. It will load all example programs and the RP6Library as well into the Projects list. This way, you can comfortably browse through the example programs and look up functions in the RP6Libary easily.

Now open the first example program at the top of the projects treeview ("Example_01_LEDs" and select the file "RP6Base_LEDs.c")! Simply double click on "RP6Base_LEDs.c" and a source file editor will be opened!

You should see the output window at the bottom of PN2. If not, you may activate the window by selecting View->Output in the menu or (it the window is too small) by dragging the edges with the mouse (the mouse cursor will change its shape into a double arrow at the upper edge of the narrow grey area labelled "Output" belonging to the lower program window...).

Just take a quick look at the source code in the editor if you like. Of course you are not expected to understand all the text right now, but we will soon learn how to handle these codes. Just for a start: the green colored textlines are comments, which are not part of the program itself and only document what the program does. We will explain this (another copy of the program *without* any comments exists to show you how short the source really is. Comments will enlarge it quite a lot, but are very useful for documenting the program flow. The uncommented version may be used to copy parts of the code into your own programs!).

Now we can check the compiling functionality.



The Tools-menu should display both previously added menu entries (see figure) or alternatively the standard [WinAVR]-entries of PN2. You may select any of these, usually both will work without any problem.

Now please click on "MAKE ALL"!

PN2 will now call the previously described batch file "make_all.bat", which will run "make". We will explain what "make" does later.

The example program gets translated (="compiled") and a hexfile is generated, which contains the special code for the microcontroller and may be loaded and executed!

The compiler will generate a great number of temporary files (using file extensions like ".o, .lss, .map, .sym, .elf, .dep"). You do not have to look at any of these files and you may use the newly created tool "make clean" to easily delete these files! Only the hexfile will is an important result for you! And "make clean" will not delete the hexfile.

Starting the MAKE ALL command in the menu will result in the following output (however the listing is abbreviated and may deviate from this example a bit!):

```
> "make" all
-------- begin --------

[...]

Compiling: RP6Base_LEDs.c

avr-gcc -c -mmcu=atmega32 -I. -gdwarf-2   -Os -funsigned-char -funsigned-bitfields -fpack-struct
-fshort-enums -Wall -Wstrict-prototypes -Wa,-adhlns=RP6Base_LEDs.lst -I../../RP6lib
-I../../RP6lib/RP6base -I../../RP6lib/RP6common -std=gnu99 -MD -MP -MF .dep/RP6Base_LEDs.o.d RP6-
Base_LEDs.c -o RP6Base_LEDs.o

Compiling: ../../RP6lib/RP6base/RP6RobotBaseLib.c

[...]

Creating load file for Flash: RP6Base_LEDs.hex
avr-objcopy -O ihex -R .eeprom RP6Base_LEDs.elf RP6Base_LEDs.hex

[...]


Size after:
AVR Memory Usage

----------------

Device: atmega32


Program:    6858 bytes (20.9% Full)
(.text + .data + .bootloader)

Data:        148 bytes (7.2% Full)
(.data + .bss + .noinit)


-------- end --------
> Process Exit Code: 0
> Time Taken: 00:01
```

An important line is "`Process Exit Code: 0`" at the very bottom. It tells us that the compiling and linking process has been completed without any errors. Any other codes indicate errors in the source code, which need to be corrected prior to successful compiling. If there are mistakes in the source code, the compiler issues several error messages in the output, which contain more info about what causes the error.

However you must understand that the "Process Exit Code: 0"-message does not imply an error-free program! Of course the compiler will not be able to detect logical errors in your program and it will not prevent the robot from hitting the wall ;-)

IMPORTANT: The output can also contain warnings, which might be helpful to identify important problems! Please look carefully for warning messages and try to solve such problems directly after you see them! PN2 uses colors to highlight warnings and errors. This allows you to easily see important messages. The program will also list the line number referring to the error message. You can click on the colored message lines and PN2 will automatically jump to the referenced line in the source code.

Also very helpful is an overview labelled "AVR Memory Usage" at the end:

```
Size after:
AVR Memory Usage
----------------

Device: atmega32

Program:    6858 bytes (20.9% Full)
(.text + .data + .bootloader)

Data:        148 bytes (7.2% Full)
(.data + .bss + .noinit)
```

This indicates that our program occupies 6858 Bytes in memory and reserves 148 Bytes RAM for static variables (of course this does not include dynamic values for Heap and Stack, but explaining this would go beyond the scope of this manual. You should always keep at least a few hundred bytes of free RAM). We have a total amount of 32KB (32768 Bytes) Flash ROM and 2KB (2048 Bytes) RAM. The Boot loader requires 2K out of 32KB Flash ROM – leaving 30KB for free use. Always keep an eye on the program size, to make sure that it fits into the available memory! (The RP6Loader will not transfer to large programs!).

The above example program will leave 23682 free bytes in ROM. In fact, this rather short example program RP6Base_LEDs.c is occupying so much of the available memory because it contains the whole RP6Library! So don't worry, there is plenty of space for your programs and small programs do not require too much memory. The function library alone occupies more than 6.5KB flash memory, but it handles a lot of important work for you already. Usually your programs will be smaller than the RP6Libary.

4.2. Program upload to the RP6

The RP6Loader may now be used to upload the recently compiled program to the robot. Please add the generated hexfile to the RP6Loader's file list by clicking "Add", make sure the entry is selected and then click "Upload!", just like you did with the selftest-program. Now switch to the terminal tab and check the output. Of course, you will have to start the program before you see any output from it. In the terminal this is easily done by pressing [STRG]+[S] on the keyboard, by using the menu item "Start" or by sending a "s" – of course after a reset you have to wait a second for the "[READY]" message from the Bootloader! The key combination [STRG]+[Y] can be used as well. After the upload has finished, the program is started straight away with this shortcut!

The first sample program is very simple and will only start a running LED flashlight and output some text through the serial interface.

Before you start writing your own programs, we will introduce the language C in a mini Crash-Course...

4.3. Why C? And what's "GCC"?

The programming language C is widely being in use – in fact, C is the standard language, which anyone interested in software development should have used at least once. C compliers are available for nearly every microcontroller currently on the market and for this reason, all recent robots by AREXX Engineering (ASURO, YETI and RP6) can be programmed in C.

The popularity of C leads to a vast amount of documentation on the internet and in literature, allowing beginners to easily study the programming language. But remember: C is a rather complex language, which cannot be learned within a few days without prior programming experience (so please don't throw the robot out of the window if things aren't working straight away ;-)).

Luckily, the basics are easily understood and programmers may continuously develop knowledge and experience. It requires some initial effort! You can not learn C automatically – this could be compared to learning a foreign language.

But it's worth the effort, as basic C knowledge will simplify learning other programming languages as the concepts are often very similar.

Just like for our other robots, the RP6 requires a special version of the C compiler from the GNU Compiler Collection (abbreviation: GCC). The GCC is a universal compiling system, supporting a great variety of languages such as C, C++, Java, Ada and FORTRAN.

GCC's target support is not restricted to AVR. It may be used for much bigger systems and knows a few dozen different targets.

The most prominent project using the GCC is the famous Linux project, of course. Most of the programs for Linux have been compiled by GCC. Thus it can be considered as a very professional and stable tool, which is being used by several big companies.

By the way: If this manual is referring to "GCC" we do not necessarily mean the complete Compiler Collection, but the C compiler only. Originally "GCC" had been in use as an abbreviation for "GNU C Compiler" – the new meaning became necessary after adding some other languages.

If you would like to learn more about GCC we invite you to visit the official GCC website: `http://gcc.gnu.org/`

GCC does not directly support the AVR target and must be adapted. The adapted version of GCC is named AVR-GCC. The WinAVR distribution contains a ready to use version for Windows users. Unix users will usually have to compile a version by themselves and we expect that you have completed this already.

4.4. C – Crash Course for beginners

*This chapter ony provides **a short introduction to C-programming**, discussing only the absolutely required minimum amount of things used for RP6. This section has to be seen as an overview of general possibilities and methods of C. We will present a few examples and basics, but further investigation on these topics is up to the reader!*

So this chapter is not more than a tiny crash course. A **complete** introduction is far beyond the scope of this *manual* and would require rather thick textbooks. Luckily the market provides a great number of good books on this topic.

A few[1] may be viewed online free of charge.

4.4.1. Literature

The following books and tutorials describe C-programming mainly for PC and other large computers. A lot of details in these tutorials do not apply to AVR *microcontrollers – the language is the same, but most libraries for typical PC-usage are a bit too large for small 8 bit microcontrollers. The best example may be the "printf" function, a must have on a PC! The "printf" function is available for microcontrollers as well, but it requires a lot of memory space and execution time, so we do not prefer to use this function. Instead we will show some more effective alternatives for our applications.*

Some *C Tutorials / Online-books (just a very small selection):*

http://www.its.strath.ac.uk/courses/c/

http://www.eskimo.com/~scs/cclass/notes/top.html

http://www.iu.hio.no/~mark/CTutorial/CTutorial.html

http://en.wikibooks.org/wiki/C

http://www.le.ac.uk/cc/tutorials/c/

http://stuff.mit.edu/iap/c/CrashCourseC.html

There are also lot of good textbooks – in order to get an overview you can start by visiting a library or a bookshop!

However you do not need to buy a book if you just want to do a few experiments with the robot! The major part of programming experience has to be acquired "Learning by doing" anyway!

All relevant information can be found on the mentioned websites. The sample programs available on the RP6 CD are also quite extensive and show a lot of things. The tutorial in this manual is also good enough for the first experiments.

An AVR specific tutorial for beginners can be found here:

http://www.avrtutor.com/

for example. This website also mentiones some tools (programming equipment, etc.) and other things, which are not required for the RP6. Nevertheless, it's worth to have a look at it.

1 *A web-search on "c tutorial" results in millions of hits! Of course there are not really that many, but there should be quite some good ones out there...*

Additional information can also be found on the WinAVR-Homepage and in the WinAVR PDF-documentation, respectively:

http://winavr.sourceforge.net/

http://winavr.sourceforge.net/install_config_WinAVR.pdf

And especially the AVR-LibC documentation:

http://www.nongnu.org/avr-libc/user-manual/index.html

which can also be found as a PDF on the RP6 CD!

Of course you do not have to read all these tutorials and books! This list is only a guide for gathering more information. Tutorials vary in size and details, but it certainly helps to read more than one.

A general AVR community and info page is

http://www.avrfreaks.net/

Here you can find a very nice forum dedicated to AVR Microcontrollers, lots of general infos, projects, tutorials and code snippets!

4.4.2. First program

As already said - learning by doing is the most efficient way of learning the C language. Having read and understood something in this crash course, you should try it out by yourself!

Of course we will have to discuss a few basics before, but in order to give you an idea of what we are talking about, let's just start with a simple C program for the RP6:

```
1  /*
2   * A small and simple "Hello World" C Program for the RP6!
3   */
4
5  #include "RP6RobotBaseLib.h"
6
7  int main(void)
8  {
9        initRobotBase();
10       writeString("Hello World!\n");
11       return 0;
12 }
```

If you have never programmed in C before, this "source code" may look like a foreign language, but the basic concepts are easy to understand.

The tiny program above is already a complete functional program, but it only initializes the microcontroller and writes the text:

"Hello World!" + Carriage Return / Line Feed

to the serial interface. This is a typical programming example, which may be found in most books (of course not with the initRobotBase call at the beginning ;)).

To get familiar with the new language, you may copy this small program into a text editor by yourself and try to compile it.

Anyone feeling bored by the tiny sample program may find a more attractive "Hello World" program in the RP6 example directory, including a running light with the LEDs and some more text outputs!

Now let's discuss the program in Listing 1 and explain it line by line!

Line 1 - 3: `/* A small and simple "Hello World" C Program for the RP6! */`
These are comment lines and will not be interpreted by the compiler. Comments are used for documenting the source code and they start with /* and end with */.

Documentation will help understanding programs written by other people, but it will also be helpful in understanding your own programs as well, especially the source codes you have written years ago!

You may write comments with any length, or "comment out" parts of your source code in order to test another program variant without having to delete the original code. Apart from these standard multi-line comments GCC also supports single-line comments initiated by "//". AFTER "//" any text will be interpreted as a comment until the end of the line.

Line 5: `#include "RP6RobotBaseLib.h"`

This includes the RP6 function library, providing a great number of useful functions and predefined things for low level hardware control. To include such a library we use so-called header files (with extension "*.h") to inform the compiler where to look for these functions. Headers are used for all things in external C-files that should be accessible in other C-files. Please take a look at the contents of RP6RobotBaseLib.h and RP6RobotBaseLib.c – this should clarify the basic principle. We will discuss more details of the "#include"-feature in the pre-processor chapter.

Line 7: `int main(void)`

This line defines the most important function in the sample program: the main function. We still have to learn about what functions are in detail, but right now we may accept the idea that the program starts at this line.

Line 8 and 12: `{ }`

In C, so-called "blocks" can be defined with accolades '{' and '}'.

A block combines several commands.

Line 9: `initRobotBase();`

A function from the RP6Library gets called here. It will initialize the AVR microcontroller and configure the AVR's hardware modules. Most of the microcontroller's functions would not work as expected, if we do not initialize them with `initRobotBase()`, so please do not forget to always call this function at the beginning of a program!

Line 10: `writeString("Hello World!\n");`

This calls the function "writeString" from the RP6 Library with the parameter String "Hello World!\n". The function will output the text to the serial interface.

Zeile 11: `return 0;`

Our program ends here. We leave the main-function and return zero. A return code is usually used in larger systems (with operating system) as an error code or for similar

functions, but is not needed in a microcontroller system. We only need to add this return value to meet the standard C-conventions (and as we will see later, programs for microcontrollers will usually never terminate).

This tiny program gave you a first impression of C-programming. Now we have to discuss some other basics before we can go on with example programs.

4.4.3. C basics

As already mentioned before, a C program is written in pure ASCII (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) text. It is strictly case sensitive and if a function is named "MyFavouriteFunction" you will have to call the function by this exact name! A function call for "myfavouritefunction" would not be recognized!

You can insert any number of spaces, tabs and line breaks between all commands and symbols without interfering with the programming syntax. As you may have seen in the sample program the commands have been indented by tabulators to improve the program's readability. But that's not necessary! You could write the program text from line 7 in listing 1 e.g.:

```
1  int main(void){initRobotBase();writeString("Hallo Welt!\n");return 0;}
```

This is an identical program, but the text is rather confusing. However we only deleted tabs, spaces and line breaks! The compiler does not care for formatting styles at all! (Of course we will need a space as a separator between keywords and variables like "int" and "main" – and we are not allowed to use a line break between two quotation marks (at least not without an escape sequence)!)

The accolades { } allow us to combine several assignments and commands to blocks, which will be needed for functions, conditional statements and loops.

Each assignment is to be terminated by a semicolon ';' to allow the compiler to identify individual commands.

Before you start typewriting and copying the program snippets from this tutorial we would like to give you an important advice: most beginners do easily forget to terminate commands by a semicolon – or use the semicolon at wrong locations and wonder about the strange program behaviour! Forgetting to place one single semicolon at certain programming sections may result in a great number of error messages – even if the real error is only one single error. In fact, the first error message will most likely identify the real error location.

Forgetting to close one of several accolade pairs or bad syntax in spelling commands belong to the common error patterns for beginners. Compilers do not accept any syntax errors! It takes time getting used to all this rules, but you will quickly learn by trial and error.

Each and every C-program starts in the main function. Basically any following commands will be executed step by step, sequentially from the beginning to the end.

The AVR Microcontroller is unable to execute several commands simultaneously! This restriction is not causing any problems as we will have ample options to control the program flow and jump to other sections of the program (this will be discussed in a later chapter).

4.4.4. Variables

First we'll have a look at storing and reading data to and from RAM. Data access is done through variables. C knows several data types for variables. Basically we will use 8, 16 or 32 Bits integer data types, which may be used either signed or unsigned. The deserved value range determines the required number of bits for defining a storage location for a variable.
For the RP6 we will use the following data types:

Type	Alternative	Value range	Remarks
signed char	int8_t	8 Bit: -128 ... +127	1 Byte
char	uint8_t	8 Bit: 0 ... 255	'' unsigned
int	int16_t	16 Bit: -32768 ... +32767	2 Bytes
unsigned int	uint16_t	16 Bit: 0 ... 65535	'' unsigned
long	int32_t	32 Bit: -2147483648 ... +2147483647	4 Bytes
unsinged long	uint32_t	32 Bit: 0 ... 4294967295	'' unsigned

By a lack of standardisation, there are several varying sizes defined on different platforms especially for the data type "int" : for our microcontroller the size is 16 bits, but its 32 bits for (modern) PC's. For this reason we preferred the modern standard definition: int16_t

These data types are always made up like: [u] int N _t

```
u : unsigned
int : Integer
N : Number of bits, e.g. 8, 16, 32 or 64
_t : t for "type" to prevent collisions to other symbols
```

On a small microcontroller, every single byte counts and clearly defined data types will help to keep track of memory consumption. You can immediately identify a 16bit data type by the number 16 in the name. The letter "u" at the beginning marks an "unsigned" data type, whereas this letter is omitted for a "signed" data type.

For the normal (classic) datatypes we only used the "signed" for "signed char" in the table above, as int and long are defined as signed types anyway and char is unsigned, even if you do not explicitly write this. The reason for these definitions is an AVR-GCC compiler option, which is activated in most cases.

The data type "char" will be used for strings, because an "uint8_t"-definition would lead to a few incompatibilities with standard C libraries and "char" is a clear and logical name for a character/string anyway. We will explain details on this topic in the RP6Library chapter for text outputs via the serial interface.

By now we simply note: *we always use "char" for characters and strings, respectively uintN_t or intN_t for integers!*

In order to use a variable in a program we have to declare it first by defining the data type, a name and eventually an initial value for this variable. The name must start with an alphabetic character (including the underscore "_"), and may contain numbers. However the variable's naming convention *excludes* a great number of special characters, e.g. "äöüß#'[]²³|*+-.,<>%&/(){}$§=´°?!^".

Variable names are case sensitive, which implies aBc and abC are different variables! Traditionally, programmers use lower case characters at least for the leading character of variable names.

The following keywords are already reserved and are NOT useable as variable names, function names or any other symbols:

auto	**default**	**float**	**long**	**sizeof**	**union**
break	**do**	**for**	**register**	**static**	**unsigned**
case	**double**	**goto**	**return**	**struct**	**void**
char	**else**	**if**	**short**	**switch**	**volatile**
const	**enum**	**int**	**signed**	**typedef**	**while**
continue	**extern**				

Furthermore the types float and double are used for floating point numbers, but we prefer to avoid usage of these data types on small AVR microcontroller. Floating point numbers are very computation time and memory intensive and usually we are able to work perfectly well with integers. Most RP6 programs will not require floating numbers.

Declaring variables is extremely simple, which may be demonstrated by declaring a variable named x:

```
char x;
```

After its declaration the variable x is valid in the following program lines and may be used e.g. by assigning a value of 10 to it:

```
x = 10;
```

Alternatively we may assign a value to another variable y directly at declaration:

```
char y = 53;
```

Basic arithmetic operations may be used as usual:

```
signed char z; // please note the "signed" in front of char!
z = x + y;   // z gets the value z = x + y = 10 + 53 = 63
z = x - y;   // z gets the value z = 10 - 53 = -43
z = 10 + 1 + 2 - 5;   // z = 8
z = 2 * x;   // z = 2 * 10 = 20
z = x / 2;   // z = 10 / 2 = 5
```

```
The programming language also provides some useful abbreviations:
```

```
z += 10;  // corresponds to: z = z + 10; this means z = 15 in this case
z *= 2;   // z = z * 2 = 30
z -= 6;   // z = z - 6 = 24
z /= 4;   // z = z / 4 = 8
```

```
z++;  // abbreviation for z = z + 1; which implies z is now 9
z++;  // z = 10  // z++ is called "incrementing z"
z++;  // z = 11 ...
z--;  // z = 10  // z-- is called "decrementing z"
z--;  // z = 9
z--;  // z = 8 ...
```

We previously used the data type "char". However in most cases we prefer standard data types in all RP6 programs.

As an example, this: `int8_t x;`

is identical to: `signed char x;`

And this: `uint8_t x;`

is identical to: `unsigned char x;  // respectively for us this is also true for "char" only as our char is by default a signed one because of an compiler option.`

4.4.5. Conditional statements

Conditional statements using "if-else"-constructs play an important role in program flow. They allow us to check whether a condition is true or false and decide if a specific program part is executed or not.

A small example:

```
1  uint8_t x = 10;
2  if(x == 10)
3  {
4        writeString("x is equal to 10!\n");
5  }
```

The declaration in line 1 defines an 8-Bit variable x and assigns the value 10 to it. The succeeding if-condition in line 2 checks, whether the value of x is equal to 10. Obviously, this condition will always be true and the program will execute the succeeding block. It will output "x is equal to 10!". If we would initialize x with a value of 231 instead, the program would not output anything!

Generally, an if-condition will always have the following syntax:

```
if ( <condition X> )
      <command block Y>
else
      <command block Z>
```

Using plain English language we may also read: "If X then do Y else do Z".

One more example:

```
1  uint16_t myFavoriteVariable = 16447;
2
3  if( myFavoriteVariable < 16000) // If myFavoriteVariable < 16000
4  {                               // then:
5      writeString("myFavoriteVariable is less than 16000!\n");
6  }
7  else                            // else:
8  {
9      writeString("myFavoriteVariable is greater than or equal to 16000!\n");
10 }
```

"myFavoriteVariable" is set to 16447, which will result in an output "myFavoriteVariable is greater than or equal to 16000!". In this example, the conditional statement is false and will cause the else-branch to be executed.

As you can see on the name "myFavoriteVariable", you can use all names for your variables you can think of, as long as they meet the naming conventions.

We may also use If-then-else-constructs to create complex conditional branches:

```
1  if(x == 1) { writeString("x is 1!\n"); }
2  else if(x == 5) { writeString("x is 5!\n"); }
3  else if(x ==  244) { writeString("x is 244!\n"); }
4  else { writeString("x has a different value!\n");}
```

Conditional statements may be using the following comparison operators:

`x == y`	Logical comparison for equality
`x != y`	Logical comparison for inequality
`x <  y`	Logical comparison for "less than"
`x <= y`	Logical comparison for "less than or equal to"
`x >  y`	Logical comparison for "greater than"
`x >= y`	Logical comparison for "greater than or equal to"

Additionally the language provides logical conjunctions:

`x && y`	true, if x is true and y is true
`x || y`	true, if x is true and/or y is true
`!x`	true, if x is false

We are allowed to link, to combine and nest these structures by using conjunctions and any number of accolade-pairs:

```
1  if( ((x != 0) && (!(x > 10))) || (y >= 200)) {
1        writeString("OK!\n");
2  }
```

The previously listed conditional statement is true, if x is not equal to zero `(x != 0)` AND `x` is not greater than 10 `(!(x > 10))` OR if y is greater than or equal to 200 `(y >= 200)`. If necessary we could add any number of other conditions, as required in our program.

4.4.6. Switch-Case

Often we will have to compare a variable to a great number of different values and decide to execute further program code according to the result of these comparisons. Of course, we could use a great number of if-then-else conditional statements, but the language provides a more elegant method by using a switch-case-construct.

A small example:

```
1  uint8_t x = 3;
2
3  switch(x)
4  {
5        case 1: writeString("x=1\n"); break;
6        case 2: writeString("x=2\n"); break;
7        case 3: writeString("x=3\n");    // At this point, "break" is missing,
8        case 4: writeString("Hello\n");  // causing the program to proceed
9        case 5: writeString("over\n");   // with the next two lines
10       case 6: writeString("there!\n"); break; // and stop here!
11       case 44: writeString("x=44\n"); break;
12       // The program will jump to this line if none of the previous
13       // conditions is met:
14       default : writeString("x is something else!\n"); break;
15 }
```

This code snippet works quite similar compared to the previous example with an "if-else-if-else-if-else..."-conditional structure, but now we use case-branches instead. There is one main difference – if one condition is true, all the following case-branches will be executed. If you do not want that – just add a "break" instruction and it will quit the switch-case construct there.

The output of the example above would be (for the default value x = 3):

```
x=3
Hello
over
there!
```

Setting x = 1 would result in an output of "`x=1\n`" and x = 5 would result in an output of:

```
over
there!
```

You may now understand the "break"-instruction will terminate the case-branches. If you omit the "break"-instruction, the program will be wading through any following instructions until either the end of the switch-construct or another "break" is reached .

If we preset the value x = 7, none of the branches will be true. The program now executes the "default"-branch, resulting in an output of : `"The value of x is something else!\n"`.

Of course the text output is only an example, but real programs may be using these constructs to generate various different movements with the robot. Several example programs use switch-case constructs for finite state machines to implement a simple behaviour based robot.

4.4.7. Loops

We need loops if operations need to be repeated a number of times.

Let's demonstrate the basic principle in an example:

```
1  uint8_t i = 0;
2  while(i < 10)                  // as long as i is less than 10...
3  {                              // ... repeat the following code:
4        writeString("i=");       // output "i=",
5        writeInteger(i, DEC);    // output the "DECimal" value of i and ...
6        writeChar('\n');         // ... a line-break.
7        i++;                     // increment i.
8  }
```

Obviously the code snippet contains a "while"-conditional loop, generating the sequence: "i=0\n", "i=1\n", "i=2\n", ... "i=9\n". Following the while-conditional header "while(i < 10)" the block surrounded by the accolades will be repeated as long as the condition is true. In plain English this may be read as: "Repeat the following block as long as i is less than 10". As we have an initial value of i = 0 and increment i at every loop-cycle, the program will be executing the loop-body 10 times and output the numbers from 0 to 9. In the loop-header, you can use the same conditions as in if-conditions.

Beneath the while-loop we can use the "for"-loop which provides similar functionality, but offers extended features for the loop-header definition.

A sample code snippet may illustrate the for-loop:

```
1  uint8_t i;  // we will not initialize i here, but in the loop-header!
2  for(i = 0; i < 10; i++)
3  {
4        writeString("i=");
5        writeInteger(i, DEC);
6        writeChar('\n');
7  }
```

This for-loop will generate output identical to the previous while-loop. However, we could implement several things within the loop-header.
Basically the for-loop is structured as follows:

```
for ( <initialize control variable> ; <terminating condition> ; <modify the control variable> )
{
      <command block>
}
```

Working with microcontrollers, you will often need infinite loops, which virtually may be repeated eternally. In fact, most microcontroller programs contain at least one infinite loop – either to put the program into a well know state for terminating the regular program flow, or by endlessly performing operations until the device is switched off.

You may simply build endless loops with while- or for-loops:

```
while(true) { }
```
or
```
for(;;) { }
```

In both cases the command block will be executed "for ever" (respectively until the microcontroller receives an external reset signal or the program terminates the loop by executing the "break"-instruction).

For the sake of completeness we finish this overview by describing the "do-while"-loop, which may be considered as an alternative to the standard "while"-loop. In contrast to "while-loops the "do-while"-loop will at least execute the command block once, even if the condition is false at the beginning.

The loop-structure is as follows:

```
do
{
      <command block>
}
while(<condition>);
```

Please remember to place a terminating semicolon! (Of course, standard while loops will not be terminated with a semicolon at the end!).

4.4.8. Functions

Functions are a key element in programming languages. In the previous chapters we already met and even used functions, e.g. "writeString", "writeInteger" and of course the main-function.

Functions are extremely useful for using identical program sequences at several locations of a program – the text output functions used in previous chapters are good examples for this. Copying identical program code to all locations where it is used would be very unhandy. Additionally, we would unnecessarily waste a lot of program memory in doing something like this. Using one single function allows us to modify program modules at a single central location instead of modifying a great number of copies. Using functions will simplify the program flow and help us to keep the overview.

Therefore C allows us to combine program sequences to functions, which are always structured as follows:

```
<Return type> <Function name> (<Parameter 1>, <Parameter 2>, ... <Parameter n>)
{
      <Program sequence>
}
```

Let's explain the idea in a small example with two simple functions and the alerady known main-function:

```
8  void someLittleFunction(void)
9  {
10       writeString("[Function 1]\n");
11 }
12
13 void someOtherFunction(void)
14 {
15       writeString("[Function 2 - something different]\n");
16 }
17
18 int main(void)
19 {
20       // Always start an RP6-program by calling this function!
21       initRobotBase();
22
23       // A few function calls:
24       someLittleFunction();
25       someOtherFunction();
26       someLittleFunction();
27       someOtherFunction();
28       someOtherFunction();
29       return 0;
   }
```

The program would display the following text at the output device:

```
[Function 1]
[Function 2 - something different]
[Function 1]
[Function 2 - something different]
[Function 2 - something different]
```

The main-function serves as the entry point and any C program will start by calling this function. Therefore each C program MUST provide a main-function.

In the previous example, the main-function starts by calling the initRobotBase-function from the RP6Library, which will initializes the microcontrollers hardware. Basically the initRobotBase-function is structured similar to the two functions in this example. In the main function, the two previously defined functions get called several times and the program code of these functions is executed.

Apart from defining functions as described in the previous example, we may also use parameters and return values. The above example is using “void” as parameter and return value, which means we do not use any parameters or return values here. The parameter “void” always indicates functions without a return values, respectively without parameters.

You may define a great number of parameters for a function and parameters are separated by commas.

An example will demonstrate the basic idea:

```
1  void outputSomething(uint8_t something)
2  {
3        writeString("[The following value was passed to this function: ");
4        writeInteger(something, DEC);
5        writeString("]\n");
6  }
7
8  uint8_t calculate(uint8_t param1, uint8_t param2)
9  {
10       writeString("[CALC]\n");
11       return (param1 + param2);
12 }
13
14 int main(void)
15 {
16       initRobotBase();
17
18       // Now execute a few function calls with parameters:
19       outputSomething(199);
20       outputSomething(10);
21       outputSomething(255);
22
23       uint8_t result = calculate(10, 30); // return value...
24       outputSomething(result);
25       return 0;
26 }
```

Output:

```
[The following value was passed to this function: 199]
[The following value was passed to this function: 10]
[The following value was passed to this function: 255]
[CALC]
[The following value was passed to this function: 40]
```

The RP6 Library provides a great number of functions. A quick look at the code of a few modules and example programs will clarify the basic principles of developing programs with functions.

4.4.9. Arrays, Strings, Pointers...

A great number of further interesting C-features are waiting to be discussed, but for details we will have to refer to available literature!

Most of the program examples can be understood without further study. In the remaining sections of this crash course we describe only a few examples and concepts in a short overview, which of course is not very detailed.

First of all we will discuss arrays. An array allows you to store a predefined number of elements of a the same data type. The following sample array may be used to store 10 bytes:

```
uint8_t myFavouriteArray[10];
```

In one single line we declared 10 variables of identical data type, which now may be addressed by an index:

```
myFavouriteArray[0] = 8;
myFavouriteArray[2] = 234;
myFavouriteArray[9] = 45;
```

Each of these elements may be treated like a standard variable.

Attention: the index always starts at 0 and declaring an array containing n elements will result in an index ranging from 0 up to n-1 ! The sample array provides 10 elements indexed from 0 up to 9.

Arrays are extremely helpful for storing a great number of variables with identical data type and may easily be manipulated in a loop:

```
uint8_t i;
for(i = 0; i < 10; i++)
      writeInteger(myFavouriteArray[i],DEC);
```

The previous code snippet will output all array elements (in this case without any separators of line breaks). A quite similar loop may be used to fill an array with values.

In C, strings are handled with by a very similar concept. Standard strings will be coded by ASCII characters, requiring one byte for each character. Now C simply defines strings as arrays, which may be considered as arrays of characters. This concept allows us to define and store a predefined string `"abcdefghijklmno"` in memory:

```
uint8_t aSetOfCharacters[16] = "abcdefghijklmno";
```

The previously discussed programming samples already contained a few UART-functions for outputting strings with the serial interface. Basically these strings are arrays. However, instead of handling a complete array, these functions will only refer to the first element's address in the array. The variable containing this first element's address is named “Pointer”. We may generate a pointer to a given array element by writing `&MyFavouriteArray[x]`, in which x refers to the indexed element. We may find a few of these statements in sample programs, e.g.:

```
uint8_t * PointerToAnElement =  &aCharacterString[4];
```

However at this stage you will not need these concepts to understand most of our programming samples or to write your own programs.

4.4.10. Program flow and interrupts

As discussed before, a program will be executed basically instruction after instruction from the top to the bottom. Apart from this standard behaviour, there is flow control with conditional jumps, loops and functions.

Beneath these usualy things, there are so-called "interrupts". They may be generated by several hardware modules (Timer, TWI, UART, external Interrupts etc.) and require the microcontrollers's immediate attention. In order to respond as soon as possible, the microcontroller will leave normal program flow and jump into a so-called Interrupt Service Routine (ISR). This interrupt reaction is virtually independent from the program flow. Don't worry! All required ISRs have been prepared in the RP6Library and take care of all required events. You will not have to implement your own ISRs. All basic things you need to know about these special interrupt-functions will be discussed and explained briefly in this section.

Basically the ISR is structured as follows:

```
ISR ( <InterruptVector> )
{
      <command block>
}
```

e.g. for the left encoder connected to the external interrupt 0:

```
ISR (INT0_vect)
{
      // Here we increment two counters at each signal edge:
      mleft_dist++;     // driven distance
      mleft_counter++;  // velocity measurement
}
```

You can not call these ISRs directly! Calling an ISR is done automatically and may happen at any time! Any time and in any part of the program an interrupt call may stop normal program flow (except inside an interrupt service routine or in case interrupts have been disabled). At an interrupt event, the apropriate ISR-function will be executed and after termination of the ISR, the program will continue execution at the after the last position in the normal program. This behaviour requires inclusion of all time critical program parts into the ISR-functions (or disabling interrupts for a short time). Otherwise delay periods calculated by processor instruction cycles may get too long, if these delays are interrupted by interrupt events.

The RP6Library uses interrupts for generating the 36kHz modulation signals for infrared sensors and IR communication. Additionally they are used for RC5 decoding, timing and delay functions, encoder measurement, the TWI Module (I^2C-Bus) and a few other applications.

4.4.11. The C-Preprocessor

In this chapter we will briefly discuss the C-preprocessor, which has been used in the preceding programming samples already in the line: `#include "RP6RobotBaseLib.h"`!

The preprocessor evaluates this command before starting the GCC-compiling process. The command line `#include "file"` inserts the contents of the specified file at the include's position. Our example program includes the file RP6BaseLibrary.h, providing definitions of all user accessible functions of the RP6Library to allow the compiler to find these functions and to control the compiling process.

However, the preprocessor features a few other options and allows you to define constants (which may be considered as fixed values to the system):

```
#define THIS_IS_A_CONSTANT 123
```

This statement defines the text constant "THIS_IS_A_CONSTANT" with a value of "123". The preprocessor simply replaces all references to it by the defined value. Constants may be considered as text replacements! In the following statement:

```
writeInteger(THIS_IS_A_CONSTANT,DEC);
```

"THIS_IS_A_CONSTANT" will be replaced with "123" and is identical to:

```
writeInteger(123,DEC);
```

(by the way: the parameter "DEC" in writeInteger is just another constant – in this case defining the constant base value 10 – for the decimal numbering system.)

The preprocessor also knows simple if-conditions:

```
1  #define DEBUG
2
3  void someFunction(void)
4  {
5        // Now execute something...
6        #ifdef DEBUG
7           writeString_P("someFunction has been executed!");
8        #endif
9  }
```

This text output will only be performed if "DEBUG" has been defined (you do not have to assign a value to it – simply defining DEBUG is enough). This is useful to activate several text outputs for debugging phases during program development, whereas for normal compiling you can remove these outputs by outcommenting a single line.

Not defining DEBUG in the preceding sample program would prevent the preprocessor to pass the contents of program line 7 to the compiler.

The RP6Library also provides macros, which are defined by using a #define statement. Macros allow to process parameters similar to functions. The following example shows a typical a macro definition:

```
#define setStopwatch1(VALUE) stopwatches.watch1 = (VALUE)
```

This definition allows you to call the macro just like a normal function (e.g. `setStopwatch1(100);`).

An important detail: You usually do not add semicolons after preprocessor definitions!

4.5. Makefiles

The "Make"-tool simplifies the compiling process by automatically executing a great number of jobs required to compile a C program. The automated process is defined in a so-called "Makefile", including all command sequences and informations for the compile process of a project. We provide these makefiles for all RP6 example projects, but of course you may create makefiles for your own projects as well. We will not discuss a makefile's structure in all details, as this would go far beyond the scope of this manual. For all RP6-projects, you can concentrate on the four following entries. Other entries are not required for beginners and may be ignored.

```
TARGET = programName

RP6_LIB_PATH=../../RP6lib

RP6_LIB_PATH_OTHERS=$(RP6_LIB_PATH)/RP6base $(RP6_LIB_PATH)/RP6common

SRC += $(RP6_LIB_PATH)/RP6base/RP6RobotBaseLib.c
SRC += $(RP6_LIB_PATH)/RP6common/RP6uart.c
SRC += $(RP6_LIB_PATH)/RP6common/RP6I2CslaveTWI.c
```

Our makefiles contain some comment lines in between. Makefile's comments always start with "#" and will be ignored in the make-process.

RP6's sample projects provide customized makefiles ready for use and you will not need any modifications unless you are planning to include new C files into the project's structure or if you start renaming files.

Start creating a makefile by specifying the program's file-name containing the Main-Function in the "TARGET"-entry. You must specify the name without extension, so please never add the ".c"-extension here! Unfortunately many other extensions will have to be specified and it might be a good idea to study existing examples of makefiles and details in the comments!

The second entry "RP6_LIB_PATH" allows you to specify the pathname of the RP6Library files. Please specify a relative path name, e.g. "../RP6lib" or "../../RP6lib" (in which "../" is means "one directory level up").

A third entry RP6_LIB_PATH_OTHERS is used to specify all other directories. We splitted the RP6Library in a number of subdirectories and you must name all of the required subdirectories for your project.

Finally you have to define all C files in the "SRC" entry (do not include any header files with ".h"-extensions, which will be automatically searched for in all specified directories!), that are used beneath the file containing the main-function. Additionally you will have to specify all RP6Library's files you are using.

Now, what does $(RP6_LIB_PATH) mean? Well, that's the way to use variables in makefiles! We already defined a "variable" named RP6_LIB_PATH. Once a variable has been declared, the variable's content may be used by writing $(<Variable>) in the succeeding text of the makefile. This useful feature will prevent a considerable amount of typing effort in makefiles...

Usually you will not have to modify anything else in the RP6 makefiles. If you are looking for additional information on this topic you may look at the detailed manual: http://www.gnu.org/software/make/manual/

4.6. The RP6 function library (RP6Library)

The RP6 function library (abbr. RP6Library or RP6Lib) provides a great number of low-level functions to control the RP6 hardware. With this library, you usually don't have to care about all the hardware specific details of the Robot and the Microcontroller. Of course, you do not need to read the 300 pages long datasheet of the ATMEGA32 Microcontroller in order to be able to write programs for the RP6. However, by reading some important parts of the data sheet you may gain insight of how the RP6Library works in detail.

In fact, we intentionally avoided perfect tuning for all RP6Library functions, in order to leave some work for you! You are invited to add more functions and to optimize existing ones! Please consider the RP6Library as a good starting point, but not as an optimal solution.

This chapter discusses the most important functions and shows short examples. If you are interested in further details, you can read the comments in the library files and study the functions and the provided examples.

4.6.1. Initializing the microcontroller

```
void initRobotBase(void)
```

ALWAYS start the main function block by calling this function! It initialize the microcontroller's hardware modules. The microcontroller may not be working properly if your program does not start with this! Partially, the hardware modules are already initialized by the Bootloader, but not all.

Example:

```
1  #include "RP6RobotBaseLib.h"
2
3  int main(void)
4  {
5        initRobotBase(); // Initialization - ALWAYS CALL THIS FIRST!
6
7  // [...] Program code...
8
9        while(true);      // Infinite loop
10       return 0;
11 }
```

Basically any RP6 Program should at least look like this. The infinite loop in line 9 serves as a predefined end of the program. Skipping the infinite loop may result in unexpected program behaviour!

Just to point out the idea of the infinite loop again: usually the infinite loop will be used to execute your own program code. So you will delete the semicolon at line 9 and replace it with your own program block (surrounded by accolades). You can define your own functions in the lines preceding the main function (at line 2 in this case) and you may call your functions anywhere from the main loop.

4.6.2. UART Functions (serial interface)

A few of the RP6Library's functions have been used in the previous C crash course already, such as the UART functions. These functions allow us to transfer text messages through the robot's serial interface to and from the PC (or to another microcontroller).

4.6.2.1. Transmitting data

```
void writeChar(char ch)
```

This function transmits a single 8-Bit ASCII character via the serial interface.

Usage is simple:

```
writeChar('A');
writeChar('B');
writeChar('C');
```

This would output "ABC". The function can also transfer ASCII codes directly, e.g.:

```
writeChar(65);
writeChar(66);
writeChar(67);
```

This would also result in an output of "ABC", because any ASCII character may be represented by a number. The number 65 refers to the character 'A'. A special communication software can also directly interpret the binary values if necessary.

You will frequently need something like:

```
writeChar('\n');
```

to start a new line in the terminal software.

```
void writeString(char *string)
and  writeString_P(STRING)
```

These functions are important for debugging programs, as they allow transmitting any text messages to the PC. Of course they may be useful for data transfers as well.

We will now have to explain the difference between writeString and writeString**_P**. Working with writeString_P will cause the text strings to be stored in Flash-ROM (**P**rogram Memory) only and of course we will have to read these strings back from Flash-ROM for output. In contrast, for writeString the strings will get stored into RAM *and* the Flash-ROM, which requires a double amount of memory. Please remember the relatively small 2KB RAM! So, if you have to output fixed text strings you should prefer using writeString_P. Of course for transferring dynamic data, which has to be available in RAM anyway, writeString **must** be used.

Using the corresponding function is just as easy as using writeChar (please note the double quotes instead of the apostrophe used for writeChar...):

```
writeString("ABCDEFG");
```

which will output "ABCDEFG", but as mentioned above, this string will get stored in ROM and will be loaded into RAM at startup.

```
writeString_P("ABCDEFG");
```

will equally output "ABCDEFG", but it does not occupy RAM for the text!

```
void writeStringLength(char *data,  uint8_t length,  uint8_t offset);
```

Whenever you need to output text with a predefined length and/or offset, you can use this function.

An example:

```
writeStringLength("ABCDEFG", 3, 1);
```

Output: "BCD"

```
writeStringLength("ABCDEFG", 2, 4);
```

Output: "EF"

This function however will occupy RAM for these strings as well and has been designed for handling dynamic texts. This function is for example used by writeIntegerLength.

```
void writeInteger(int16_t number, uint8_t base);
```

This very useful function will output integer values as ASCII Text. From previous examples we remember, that writeChar(65) outputs 'A' instead of the number 65...

Thus we need a function to output numbers as text strings.

Example:

```
writeInteger(139, DEC);
```

Output: "139"

```
writeInteger(25532, DEC);
```

Output: "25532"

The function allows you to output the complete range of 16bit signed integers between -32768 up to 32767. Anyone planning to use numbers beyond these limits will have to modify the function or alternatively write a special function from scratch!

Now you may wonder why we are using a second parameter "DEC"! The answer is quite simple: this parameter is controlling the output format for this number. Of course instead of DECimal (base 10) we may use several alternative output formats, such as binary (BIN, base 2), octal (OCT, base 8) or hexadecimal (HEX, base 16).

Some examples:

```
writeInteger(255, DEC);
```

Output: "255"

```
writeInteger(255, HEX);
```

Output: "FF"

```
writeInteger(255, OCT);
```

Output: "377"

```
writeInteger(255, BIN);
```

Output: "11111111"

These functions are extremely useful for lots of applicatins. Especially to output integers in HEX or BIN format, as these formats allow you to directly see how the bits are set in this integer.

```
void writeIntegerLength(uint16_t number, uint8_t base, uint8_t length);
```

This function is a variant for writeInteger, enabling you to specify the number of digits (length) to be displayed. If the number's length is below the specified limit, the function will add leading zeros. If the number's length exceeds the specified limit, the function will only display the trailing digits.

As usual we will demonstrate the function's behaviour by a few examples:

```
writeIntegerLength(2340, DEC, 5);
```

Output: "02340"

```
writeIntegerLength(2340, DEC, 8);
```

Output: "00002340"

```
writeIntegerLength(2340, DEC, 2);
```

Output: "40"

```
writeIntegerLength(254, BIN, 12);
```

Output: "000011111110"

4.6.2.2. Receiving data

The reception of Data through the serial interface is completely interrupt based. The received data is written to a so called circular buffer automatically in the background.

Single received bytes/chars can be read out of the buffer with the function:

```
char readChar(void)
```

It returns the next available character in the Buffer and deletes it from the Buffer.

If the circular buffer is empty, 0 is returned. You should check for the buffer size with this function:

```
uint8_t getBufferLength(void)
```

before calling readChar, otherwise you can't tell if a 0 is real data or not!

Several characters may be read with

```
uint8_t readChars(char *buf, uint8_t numberOfChars)
```

at once from the Buffer. You need to pass a pointer to an Array and the number of chars to receive as parameters to this function. It returns the actual number of chars that were written to the Array. This is useful if the buffer contains less chars than specified with numberOfChars paramter.

If the Buffer is completely full, any new received data will NOT overwrite data in the buffer. Instead, a status Variable (uart_status) will be set to signal a buffer overflow (UART_BUFFER_OVERFLOW). You should write your programs such that this can not happen. Usually a buffer overflow occurs if the datarate gets to high or the program is busy with something else for too long and does not read the data from the buffer. You should avoid using long mSleep delays. If required, you can increase the size of the circular buffer. Predefined size of the Buffer is 32 chars. In the file RP6uart.h, you can change the definition UART_RECEIVE_BUFFER_SIZE.

A bigger example program can be found on the CD-ROM (Example_02_UART_02).

4.6.3. Delay and timer functions

Microcontroller programs often have to be delayed completely for some time, or need to wait a period of time before a specific action is performed.

The RP6Library also provides functions for these purposes. It uses one of the MEGA32's timers to achieve relatively accurate delay control, which is independent from other program flow or interrupts which could disturb delay routines.

You will have to carefully decide where you can use these functions! Using these functions along with automatic speed control and ACS (will be explained later) may cause problems! If you need to use automatic speed control or ACS, please use very short delays of less than 10 milliseconds only! Instead of blocking delays, you may prefer the "stopwatch" functions instead, which will be discussed in the following section.

void sleep(uint8_t time)

This function will stop normal program execution for a predefined period of time. The delay is specified with a resolution of 100µs (100µs = 0.1ms = 0.0001s, which is extremely short for human perception...). The use of an 8 bit sized variable allows us to define delays up to 25500µs = 25.5ms. While the normal program is "sleeping", interrupts will still be processed immediately. This will only delay the normal program's execution. As mentioned before, it uses a hardware timer and is not influenced too bad by interrupt events.

Examples:

```
sleep(1); // 100µs delay
sleep(10); // 1ms delay
sleep(100); // 10ms delay
sleep(255); // 25.5ms  delay
```

void mSleep(uint16_t time)

Whenever you need long delays, you may prefer mSleep, which allows to specify delay period in milliseconds. The maximum delay period is 65535ms, or 65.5 seconds.

Examples:

```
mSleep(1);     // 1ms delay
mSleep(100);   // 100ms delay
mSleep(1000);  // 1000ms = 1s delay
mSleep(10000); // 10 seconds delay
mSleep(65535); // 65.5 seconds  delay
```

Stopwatches

The problem with these standard delay functions is, that they will stop the normal program flow completely. This may be unacceptable, if only a specific part of the program needs to wait for a period of time, whereas other parts are supposed to continue with their tasks...

One of the main advantages in using hardware-timers, is independence from the normal program flow. With these timers, the RP6Library implements universal so-called "Stopwatches". The author has chosen this unusual title for similarity with ordinary Stopwatches. These "Stopwatches" will simplify a great number of jobs. Usually customised timer functions for each individual program would be required, but the

RP6Library offers an universal module for general purpose usage.

Stopwatches allow you to handle a number of tasks "simultaneously" – at least this is what you will see from your point of view outside of the microcontroller.

The RP6 provides eight 16bit Stopwatches (Stopwatch1 to Stopwatch8), which may be started, stopped, set and read. As for the mSleep function we have chosen a resolution of one millisecond, which implies each of these timers will increment its counter in intervals of 1ms. This method is not useable for very critical timing, as checking the counter levels may not meet strict accuracy requirements.

The following example demonstrates the usage of the Stopwatches:

```
1  #include "RP6RobotBaseLib.h"
2
3  int main(void)
4  {
5        initRobotBase(); // Initialize the micro-controller
6        writeString_P("\nRP6 Demo Program for Stopwatches\n");
7        writeString_P("_____________________________________\n\n");
8
9        startStopwatch1(); // Start Stopwatch1
10       startStopwatch2(); // Start Stopwatch2
11
12       uint8_t counter = 0;
13       uint8_t runningLight = 1;
14
15       // Main loop:
16       while(true)
17       {
18             // A small LED running light:
19             if(getStopwatch1() > 100)  // Did 100ms (= 0.1s) pass by?
20             {
21                 setLEDs(runningLight); // Set the LEDs
22                 runningLight <<= 1;    // Next LED (shift operation)
23                 if(runningLight > 32)  // Last LED?
24                     runningLight = 1;  // Yes, restart with LED1!
25                 setStopwatch1(0);      // Reset Stopwatch1 to zero
26             }
27
28             // Output a counter level in the terminal:
29             if(getStopwatch2() > 1000) // Did 1000ms (= 1s) pass by?
30             {
31                 writeString_P("CNT:");
32                 writeInteger(counter, DEC); // Output counter level
33                 writeChar('\n');
34                 counter++;          // Increment the counter
35                 setStopwatch2(0);   // Reset Stopwatch2 to zero
36             }
37       }
38       return 0;
39 }
```

The program is quite simple. Every second, it outputs the counter level via the serial interface and increments the counter (lines 29 up to 36). At the same time we execute a simple running light with the LEDs (lines 19 up to 26), with a refresh interval of 100ms.

We are using Stopwatch1 and Stopwatch2 here, which get started at lines 9 and 10 respectively. Afterwards the Stopwatch counters are running. The infinite loop (at lines 16 up to 37) constantly checks, whether the Stopwatches exceed a predefined level. The if-condition in line 19 for example controls the running light and checks, whether the stopwatch has been running for at least 100ms since the last reset to zero. As soon as this gets true, the next LED will be activated and the counter will be reset to zero (line 25) in order to wait for another 100ms. The same procedure is used for the other counter, which in contrast checks for intervals of 1000ms, respectively 1 second.

You will find a slightly extended version of this program on the CD. It is just a small example, but you may build rather complex systems with the Stopwatches and start or stop them at certain events ...

The sample program on the CD also includes the running light and the counter (we have even 3 counters in this program...), but they are implemented in separate functions, which will be called from the infinite loop.

Seperating program code in functions will help you to keep an overview of complex programs and simplifies reusing program modules by Copy&Paste. E.g. the running light code can be used in other programs without big changes...

Several macros have been implemented for stopwatch control.

startStopwatchX()

starts stopwatch X. The command does not reset the Stopwatch and it will continue incrementing from the last counter level.

Examples:

```
startStopwatch1();
startStopwatch2();
```

stopStopwatchX()

stops Stopwatch X.

Examples:

```
stopStopwatch2();
stopStopwatch1();
```

uint8_t isStopwatchXRunning()

returns if stopwatch X is running.

Example:

```
if(!isStopwatch2Running) {
      // Stopwatch has been deactivated, so you may do sth. againt this...
}
```

setStopwatchX(uint16_t preset)

This macro sets the counter of stopwatch X to a given value.

Examples:

```
setStopwatch1(2324);
setStopwatch2(0);
setStopwatch3(2);
setStopwatch4(43456);
```

getStopwatchX()

This returns the counter level of stopwatch X.

Examples:

```
if(getStopwatch2() > 1000) { ... }
if(getStopwatch6() > 12324) { ... }
```

4.6.4. Status LEDs and Bumpers

void setLEDs(uint8_t leds)

This function allows you to control the 6 Status LEDs. Usage can be simplified with binary constants instead of usual decimal numbers. Binary constants are formatted like: 0bxxxxxx. The LEDs need 6 digits only.

Examples:

```
setLEDs(0b000000); // This deactivates all LEDs.
setLEDs(0b000001); // activates StatusLED1 and switches off all other LEDs.
setLEDs(0b000010); // StatusLED2
setLEDs(0b000100); // StatusLED3
setLEDs(0b001010); // StatusLED4 and StatusLED2
setLEDs(0b010111); // StatusLED5, StatusLED3, StatusLED2 and StatusLED1
setLEDs(0b100000); // StatusLED6
```

An alternative possibility the following:

```
statusLEDs.LED5 = true;    // activate LED5 in the LED-register
statusLEDs.LED2 = false;   // deactivate LED2 in the LED-register
updateStatusLEDs();        // commit the changes!
```

Here we activate StatusLED5 and deactivate StatusLED2, but we do not modify the state of any other LED! This simplifies LED control from different program parts.

Attention: `statusLEDs.LED5 = true;` will NOT directly activate LED5! This command will only set the corresponding bit in a variable! The LED5 will be illuminated after executing `updateStatusLEDs();`!

Two port-pins of the LEDs are additionally used to check the bumper status. In order to read the bumpers, the controller will quickly switch the pin direction to input mode and check if the connected microswitch is closed. We provide two functions for checking the bumpers. The first function:

uint8_t getBumperLeft(void)

will read the left bumper status, whereas:

uint8_t getBumperRight(void)

will read the right bumper switch.

The Microcontroller executes these functions very fast and you will not see that the LEDs turn off, although the Pin is set to input for a few instruction cycles. Of course you should not call these functions frequently without a delay of a few ms in between.

The LED Portpins should be controlled with the predefined functions only! There are resistors to protect the bumper ports, but if the pins are set to low level output AND a bumper switch is closed at the same time, the port terminal will conduct a bit more current. Such unecessary currents should be avoided of course (the AVRs have Tri-state outputs – to turn the LED off, they are set to floating)

Example:

```
if(getBumperLeft() && getBumperRight())  // Both Bumpers...
      escape(); // Define your own function here, e.g. drive back and rotate
else if(getBumperLeft()) // Left...
      escapeLeft(); // drive back again and turn to the right.
else if(getBumperRight()) // Right...
      escapeRight(); // drive back again and turn to the left.
mSleep(50); // Check bumpers at a rate of 20 times pro second (20Hz)...
```

Pressing the bumpers will illuminate the LEDs 6 and 3. This is intentionally and cannot be avoided. However, the bumper switches are usually not activated to often, so this does not bother us too much.

You can connect other sensors with digital output to the four remaining LEDs. Switching loads like additional LEDs or Motors on and off via Transistors is possible, too. The only thing missing are apropriate functions that control the four ports, but you can take a look at the existing functions for the LEDs and Bumpers...

Attention: *In order to protect the microcontroller's ports, always insert at least 470 Ohm resistors between sensors/actors and ports for current limiting!*

The RP6 allows you to deactivate the LEDs during the boot phase. This is useful to avoid port activity while booting is in progress and if there are other devices connected to the LED portpins.

The first byte in the internal EEPROM (addressed by 0) is reserved to control the LED modes, so please do not use this byte for your own programs (overriding this byte will not disturb anything else, but you may wonder why LEDs are no longer illuminated after turning on the RP6...) !

There are a lot of things on the RP6 that have to be evaluated constantly in order to make them work correctly. For instance, the ACS needs to transmit IR pulses in specific intervals and check for reception. We can not use automated interrupt functions for this, as the interrrupt service routines need to be as fast as possible. Thus you have to call several functions from the main program frequently. In a well designed program, these tasks will act like if they are just running in background.

We will discuss all these functions for ACS and similar later in this chapter and provide more details on this. However we had to anticipate a few details in order to make it easier for you to understand how the bumper functions work and why they are implemented like this!

Now, as we are performing background tasks anyway, we can run some other (smaller) tasks as well along with the bigger things – such as bumper evaluation. This is a simple and fast task which you would usually perform in the main loop anyway.

To automatically check the bumpers you have to call this function:

```
void task_Bumpers(void)
```

frequently from the main loop (s. a. chapter about driving functions, in which we will discuss this in detail). This function will automatically check the bumper sensors at intervals of 50ms (pressed or not) and writes their current state into the variables:

```
bumper_left and bumper_right
```

You may use these variables anywhere in the program e.g. in if-conditions, loops etc., or assign them to other variables.

Example:

```
#include "RP6RobotBaseLib.h"

int main(void)

      initRobotBase();  // Initialize the Microcontroller

      setLEDs(0b001001); // Turn LEDs 1 and 4 on (both green)

      while(true)
      {
            // Set the LEDs depending on which
            // bumpers are pressed down:
            statusLEDs.LED6 = bumper_left;  // Left bumper pressed
            statusLEDs.LED4 = !bumper_left; // Left bumper released
            statusLEDs.LED3 = bumper_right; // Right bumper pressed
            statusLEDs.LED1 = !bumper_right; // Right bumper released
            // Both bumpers pressed down:
            statusLEDs.LED2 = (bumper_left && bumper_right);
            statusLEDs.LED5 = statusLEDs.LED2;
            updateStatusLEDs(); // update LEDs...

            // Check bumper status:
            task_Bumpers(); // Frequently call this from the main loop!
      }
      return 0;
}
```

The sample program is using the Status LEDs to show the bumper status. Pressing down the left bumper will turn LED6 on and turn LED4 off. In contrast releasing the left bumper will turn LED6 off and turn LED4 on. Pressing down the left bumper turns LED6 on anyway, but here we want to demonstrate the LED usage in general and you could use anything else to control the LEDs like shown above!
The example works similar for the right bumper with LED3 and LED1. Pressing down both bumper sensors will turn LED2 and LED5 on.

Equipped with such an automatic check for the Bumpers, it was self-evident to create something that calls a self defined function automatically everytime the state of the bumpers changes. Usually the Bumpers will be hit very rarely only and it makes sense to check this in the main program only if necessary.

C allows us to define pointers to functions and call these functions without pre-defining the function in the library. Usually a function needs to be defined in our Library at the time of compilation in order to be callable.

This method allows us to use self-defined functions as so-called "Event Handlers". Pressing down a bumper will automatically result in calling a predefined dedicated function (within 50ms). This special function must be registered as an Event Handler and will have to provide a specific signature: the function must not return a value and has no parameter (both return value and parameter must be "void"). Therefore the function's signature will have to look like: `void bumpersStateChanged(void)`. For example you may register the Event Handler at the very beginning of the main function. Registering the Event Handler can be done with the following function:

```
void BUMPERS_setStateChangedHandler(void (*bumperHandler)(void))
```

You do not have to exactly understand this command – to make a long story short this function expects a pointer to a function as parameter...

We will explain this in a simple example:

```
1  #include "RP6RobotBaseLib.h"
2
3  // Our "Event Handler" function for the bumpers.
4  // This function will be called automatically by the RP6Library:
5  void bumpersStateChanged(void)
6  {
7        writeString_P("\nBumper status changed:\n");
8
9        if(bumper_left)
10             writeString_P(" - Left bumper pressed down!\n");
11       else
12             writeString_P(" - Left bumper released!\n");
13       if(bumper_right)
14             writeString_P(" - Right bumper pressed down!\n");
15       else
16             writeString_P(" - Right bumper released\n");
17 }
18
19 int main(void)
20 {
21       initRobotBase();
22
23       // Register the Event Handler:
24       BUMPERS_setStateChangedHandler(bumpersStateChanged);
25
26       while(true)
27       {
28            task_Bumpers(); // Automatically check bumpers at 50ms intervals
29       }
30       return 0;
31 }
```

The program will react on alterations of the bumper status once-only by outputting the current status of both bumpers. For example, if you press down the right bumper, the output would be:

```
Bumper Status has changed:
 - Left bumper has not been activated.
 - Right bumper has been activated!
```

Pressing down both bumper sensors will result in:

```
Bumper Status has changed:
 - Left bumper has been activated!
 - Right bumper has been activated!
```

You will hardly ever manage to activate both bumpers simultaneously and you might see an additional message in which only one of the bumpers is pressed down. If you press them down fast enough, it should show only one message. This is because of the 50ms interval...

You may notice that the example program never directly calls the `bumpersStateChanged` function! The RP6Library manages this automatically at each bumper status alteration from the `task_Bumpers` function. In fact, `task_Bumpers` first does not know our `bumpersStateChanged` function and must be calling this function by using a pointer, which will be set up properly in line 24.

Of course the Event Handler may be extended beyond text outputs – e.g. think of stopping the robot and driving back / rotating. However, such things should not be peformed in the Event Handler itself, but elsewhere in the program. You might set a command variable(s) in the Event Handler, which is then checked in the main program to identify which movement should be performed! Always keep Event Handlers as short as possible!

You can use all RP6Library functions in Event Handlers, but you must be careful with the "rotate" and "move" functions, which are to be discussed in later chapters! Do NEVER use the blocking mode of these functions in event handlers (repeatedly activating the bumpers will not quite work as expected ;-))!

The basic idea of Event Handlers is used by a number of other functions, too. For example the ACS – which is very similar to use by calling an Event Handler for each status alteration of the object sensors.

We also use an Event Handler for receiving RC5 Codes from remote controls. Any reception of RC5 Coded signals initiates a call to a corresponding Event Handler function.

There is no need to use Event Handlers for these jobs – of course you may simply use if-conditions to check for changes, but the Event Handlers simplify program design. Consider it a matter of of taste.

By the way: the CD provides you with a number of detailed sample programs on this topic!

4.6.5. Read ADC values (Battery, Motorcurrent and Light sensors)

There are a lot of sensors connected to the ADC (Analog to Digital Converter), as described in chapter 2. Of course, the RP6Library provides a function to read the measured ADC values:

```
uint16_t readADC(uint8_t channel)
```

This function returns a 10 Bit value (0...1023) and requires a 16 Bit variable for sensor values.

The following channels can be read:

```
ADC_BAT           --> Battery voltage sensor
ADC_MCURRENT_R    --> Motorcurrent sensor for the right motor
ADC_MCURRENT_L    --> Motorcurrent sensor for the left motor
ADC_LS_L          --> Left light sensor
ADC_LS_R          --> Right light sensor
ADC_ADC0          --> Free ADC channel for your own sensor devices
ADC_ADC1          --> Free ADC channel for your own sensor devices
```

Hint: the two connectors for the free ADC channels are not populated. You may solder connectors with standard 2.54mm grid and maybe additionally insert two 100nF capacitors and a large 470µF Elco, just in case your sensor circuitry required high peak current, like Sharp IR-distance-sensor do...

This requires some soldering-experience! If you are unexperienced, it may be a better idea to go for an extension module!

Examples:

```
uint16_t ubat = readADC(ADC_BAT);
uint16_t iMotorR = readADC(ADC_MCURRENT_R);
uint16_t iMotorL = readADC(ADC_MCURRENT_L);
uint16_t lsL = readADC(ADC_LS_L);
uint16_t lsR = readADC(ADC_LS_R);
uint16_t free_adc0 = readADC(ADC_ADC0);
uint16_t free_adc1 = readADC(ADC_ADC1);

if(ubat < 580) writeString_P("Warning! Low battery level!");
```

Basically the 5V supply is used as reference voltage, but the function could be modified such that the internal ATMEGA32's 2.56V reference voltage is used instead (see the MEGA32 data sheet). The standard RP6 sensors do not require this usually.

It makes sense to perform several ADC measurements subsequently, to store the results in an array and to calculate the average and/or Minimum/Maximum value before processing the ADC output any further.

Processing several values can reduce measurement errors. As an example where "averaging" methods are required, we may consider the battery voltage measurement. The Battery voltage will vary a lot under heavy load, especially with alternating load conditions like caused by the motors.

In analogy to the bumper sensors, we may automatically perform ADC measurements and simplify the main program by using a comfortable function:

```
void task_ADC(void)
```

which will shorten the time required to evaluate all ADC channels in a program. Calling this function will subsequentially read all ADC channels in "background mode" (whenever there is some spare time, the measurements are started / read out...) and store the results in predefined variables.

The ADC requires some time for each measurement and the readADC function would block the program flow during that time. The measurement itself does not require any program action, so we can do something else during this time (the ADC is a hardware module)

Individual channel measurements are stored in the following 16Bit Variables, which can be used any time and anywhere in your programs:

```
ADC_BAT:          adcBat
ADC_MCURRENT_L:   adcMotorCurrentLeft
ADC_MCURRENT_R:   adcMotorCurrentRight
ADC_LS_L:         adcLSL
ADC_LS_R:         adcLSR
ADC_ADC0:         adc0
ADC_ADC1:         adc1
```

As soon as you have started using the task_ADC() function, you must use these variables instead of the readADC-function!

Example:

```
1  #include "RP6RobotBaseLib.h"
2
3  int main(void)
4  {
5        initRobotBase();
6        startStopwatch1();
7        writeString_P("\n\nJust a sample ADC evaluation program...\n\n");
8        while(true)
9        {
10             if(getStopwatch1() > 300) // Every 300ms...
11             {
12                   writeString_P("\nADC Left-sided light-sensor: ");
13                   writeInteger(adcLSL, DEC);
14                   writeString_P("\nADC Right-sided light-sensor: ");
15                   writeInteger(adcLSL, DEC);
16                   writeString_P("\nADC Battery: ");
17                   writeInteger(adcBat, DEC);
18                   writeChar('\n');
19                   if(adcBat < 600)
20                         writeString_P("Warning! Low battery level!\n");
21                   setStopwatch1(0); // Reset Stopwatch1 to zero
22             }
23             task_ADC(); // ADC evaluation - this has to be called
24       }                 // permanently from the main loop!
25       return 0;         // But then you can NOT use readADC anymore!
26 }
```

This program will output measurement values of both light sensors and the battery

voltage at intervals of 300ms. The program will issue a warning as soon as the battery voltage drops below a level of *approximately* 6V.

4.6.6. ACS – Anti Collision System

In contrast to the CCRP5, which used a small co-processor, the Anti Collision System of the RP6 has been directly implemented on the MEGA32. This architecture needs some more programming effort, but also allows custom modifications and adaptations. The RP5 design did not allow any modifications of the co-processor's software at all...

The ACS detection range, respectively transmitting power of both IR-LEDs may be controlled by the following functions:

```
void setACSPwrOff(void)  --> Deactivate the ACS IR-LEDs
void setACSPwrLow(void)  --> Short range
void setACSPwrMed(void)  --> Medium range
void setACSPwrHigh(void) --> Long range
```

As the ACS is nearly completely implemented in software, it is required to frequently call the following function within the main loop:

```
void task_ACS(void)
```

This function completely controls the ACS. Further processing can be done in a similar procedure as it has been demonstrated for the bumpers.

The RP6Lib provides two variables:

```
obstacle_left and obstacle_right
```

each of which will be set to true as soon as the ACS detects an obstacle. If both variables have been set to true, the obstacle will be found located directly in front of the robot.

You may optionally use an Event Handler for the ACS.

```
void ACS_setStateChangedHandler(void (*acsHandler)(void))
```

This function registers the Event Handler, which must have the following signature: `void acsStateChanged(void)`

However, you may name the function whatever you like.

The next sample program will demonstrate how to use this. We start by registering the Event Handler (line 44), then activate all sensors including the IR Receiver (line 46 – of course it does not work without this!) and setup the transmitting power for the ACS IR LEDs (line 47). The main loop frequently calls the function task_ACS().

Further evaluation will be performed automatically. The acsStateChanged function gets called as soon as the ACS changes its state, which happens if an obstacle is detected or if it disappears again. The program will display the current ACS state with text messages in the terminal and with the LEDs.

```
#include "RP6RobotBaseLib.h"

void acsStateChanged(void)
{
	writeString_P("The ACS-status has changed!   L: ");

	if(obstacle_left)  // Obstacle on the left
		writeChar('o');
	else
		writeChar(' ');

	writeString_P(" | R: ");

	if(obstacle_right) // Obstacle on the right
		writeChar('o');
	else
		writeChar(' ');

	if(obstacle_left && obstacle_right) // Obstacle in the middle?
		writeString_P("   Amidships!");
	writeChar('\n');

	statusLEDs.LED6 = obstacle_left && obstacle_right; // In the middle?
	statusLEDs.LED3 = statusLEDs.LED6;
	statusLEDs.LED5 = obstacle_left;     // Obstacle on the left
	statusLEDs.LED4 = (!obstacle_left);  // LED5 inverted!
	statusLEDs.LED2 = obstacle_right;    // Hindernis on the right
	statusLEDs.LED1 = (!obstacle_right); // LED2 inverted!
	updateStatusLEDs();
}

int main(void)
{
	initRobotBase();

	writeString_P("\nRP6 ACS - Testprogram\n");
	writeString_P("_____________________\n\n");

	setLEDs(0b111111);
	mSleep(1000);
	setLEDs(0b001001);

	// Register the ACS Event Handler:
	ACS_setStateChangedHandler(acsStateChanged);

	powerON(); // Activate the IR receiver (incl. encoders etc.)
	setACSPwrMed(); // set the ACS medium transmit power.

	while(true)
	{
		task_ACS(); // Frequently call the task_ACS function!
	}
	return 0;
}
```

This sample program also demonstrates once again how to activate and deactivate individual LEDs.

You should connect the Robot to the PC and look at the output in the terminal and

also watch the LEDs. And then just move your hand or an object directly in front of the robot!

Several sources of interference are known to affect the ACS! Some types of fluorescent lamps and similar light sources may virtually blind the robot or at least decrease sensitivity. If you encounter such problems you may start by deactivating all possible interfering sources of light. (Hint: eventually you may have put the robot directly in front of a Flatscreen, which also must be considered as a potential source for problems as most of the Flatscreens use a fluorescent lamp as backlight...).
Of course the detection range heavily depends on the obstacle's surface. Obviously, black surfaces will not reflect the same amount of light as bright white surfaces. The ACS may even ignore some of the dark colored objects!
In critical situations we might prefer to support the ACS by ultrasonic sensors or by improved IR sensors.

Before allowing the robot to cruise around in a room you should at least perform a few simple tests with the ACS by testing detection capability of several different objects. Especially you could try to find out which objects FAIL to be detected properly. This test will allow you to remove such obstacles before operating the robot... but compared to the predecessor CCRP5, a failing ACS-system will not cause problems, as the bumper will prevent damages to the IR LEDs!

4.6.7. IRCOMM and RC5 Functions

The IR receiver allows the RP6 to receive IR-signals from standard TV/Hifi remote controls, but this is restricted to remote controls using the RC5 code! Most of the universal remote controls (see fig.) may be programmed to this code – please read the manual of your remote control to set up the RC5 code. If RC5-code is missing in the code table, you may simply test several different manufacturers.

The ACS will ignore remote controls transmitting RC5 signals and these will usually not interfere with the ACS obstacle detection. The system will still be able to detect obstacles, but may react slower as operation is restricted to pauses between RC5-transmits. If a remote control does not use RC5, the ACS could get malfunctional.

A suitable software would allow the RP6 to be controlled by an IR remote control.

And the IRCOMM may be used to transmit IR Signals as well. Both transmitting diodes at the robot's front panel are pointing upwards to the ceiling. Reflections from the ceiling and other objects or direct line-of-sight, allow communication with other robots or a base station.

Communication is relatively slow (transmitting a data packet takes approximately 20ms plus a short pause), but allows you to transmit simple commands and single measurement values. Transmitting range is restricted to distances of about 2 up to 4 meters inside one room (depending on lighting conditions, obstacles, ceiling surfaces

and robot's expansion boards mounted on the top). You will be able to extend the communication range by adding some more IR LEDs (for example controlled by another MOSFET with a large capacitor and a small series resistor).

Synchronisation to the ACS operation is controlled by the task_ACS() function, which must be called frequently from the main-loop in order to handle reception of IR-signals – and additionally for managing the IRCOMM transfers!

RC5-data packets consist of a device address, a key code and a toggle bit. The 5 Bit device address tells which device is controlled by the remote control. Such as a television, a video recorder, a Hi-Fi system, etc. For our application, the device address may also be used to address several individual robots. The 6 Bit Key code corresponds to the pressed key on the remote control, but may allow us to transfer any other data as well. This provides only 6 bits per transfer, but you can transmit 8 Bit Data in two seperate transfers or divert 2 bits of the device address and/or the toggle bit from their intended use.

Standard remote controls use the toggle bit to identify a continously hold down or repeatedly pressed key. However we may use the toggle bit for any other functionality for communication between robots.

RC5 data packets can be transmitted with the following function:

```
void IRCOMM_sendRC5(uint8_t adr, uint8_t data)
```

in which `adr` corresponds to the device address and `data` to the Key code respectively the data value. The parameter `adr` allows you to set the Toggle Bit at the most significant bit (MSB) by applying the constant `TOGGLEBIT` in the following way:

```
IRCOMM_sendRC5(12 | TOGGLEBIT, 40);
```

This command will transmit an RC5 data packet to the device with address 12, activated Toggle Bit and 40 as data value.

```
IRCOMM_sendRC5(12, 40);
```

This is what it looks like without activated Toggle Bit.

In analogy to the bumpers and ACS, reception of RC5 data can be managed by an Event Handler. As soon as a new RC5 packet has been received, an Event Handler will be called automatically by the task_ACS() function. For examle this allows you write a program that lets the robot turn to the left if it receives the key code 4 and turn to the right at a key code of 6...

One of the example programs provides this functionality: full motion control by using an IR remote control.

The prescribed signature for the Event Handler must correspond to:

```
void receiveRC5Data(RC5data_t rc5data)
```

but of course you may freely name the function!

```
void IRCOMM_setRC5DataReadyHandler(void (*rc5Handler)(RC5data_t))
```

This function allows you to register a predefined Event Handler, e.g. by:

```
IRCOMM_setRC5DataReadyHandler(receiveRC5Data);
```

After this, the specified function will be called on every valid RC5 code reception.

By the way: `RC5data_t` is a special pre-defined datatype, containing the RC5 Device Address, the Toggle Bit the Key code (respectively a data value). You may use these data just like ordinary variables with the following identifiers:

`rc5data.device, rc5data.toggle_bit, rc5data.key_code`

The CD provides a sample program that shows how to use this.

Attention: Never activate the IRCOMM output pin permanently! The IR LEDs and MOSFET driver circuit has been designed for pulsed operation and is allowed to be operated at pulse periods of about one millisecond only! Otherwise current consumption gets too high with fully charged accumulators. Do not modify any of the IRCOMM functions if you are unexperienced with such things. Especially the Interrupt Routine for controlling these IR devices must not be modified!

4.6.8. Power saving functions

In previous chapters we have been using powerON(), but we did not describe its functionality. The RP6 has can save a bit power by deactivating the ACS, the encoder system, the motor current sensors and the PowerON LED. It saves roughly 10mA to deactivate these sensors.

To turn the sensors ON you may call the macro:

```
powerON()
```

and to save some power and turn the sensors off you can call:

```
powerOFF()
```

Both macros will only set an I/O Pin.

Before using the ACS, IRCOMM or motor control, the powerON() macro has to be executed! Otherwise the corresponding sensor circuits will not be supplied with power. In order to operate correctly, the motor control routines require the encoder signals and current sensors feedback.
Whenever you forget to call powerON(), the motors will be shutdown immeadiately after a short start attempt. To indicate this error condition, the four red status LEDs will start flashing.

4.6.9. Drive system functions

The RP6Library provides comfortable functions for controlling the robot's drive system. Some functions will automatically control the motor speed by encoder feedback, check the motor current, automatically move certain distances and perform many other tasks. These features are very comfortable, but – just like with the other systems of the Robot - we need to consider some special things in order to use them.

The actual development status cannot be considered as optimal. There is lots of room for improvements!

```
void task_motionControl(void)
```

We will have to call the `task_motionControl` function frequently from the main pro-

gram's loop – otherwise the automatic control will not work! Frequently calling from the main program simply implies to call this function at each and every main loop cycle. Calling the function at intervals of 10 up to 50 milliseconds will be sufficient, but its better to call the function at considerably shorter intervals. Calling the function more frequently will not cause any problems, as a hardware timer is controlling the timing. For the same reason, we may call the function at changing interval periods, e.g. ranging from 1ms up to 10ms. Calling the function very frequently will not cost to much processing time, as the function will only be executed completely in predefined minimum intervals.

If the function is used correctly, it will automatically regulate the motor's rotational velocity to the desired value.

Speed control is achieved by determining deviations in every measurement cycle and summing them up (so-called integrating regulator). This error value is used to adjust the motor voltages via the Microcontroller's PWM-Modules. If the speed is too low, the error values will be positive and motor voltage has to be increased at an appropriate rate. If the speed is too high, the voltage must be reduced. This method will quickly adjust the RP6's speed to a relatively constant PWM value (in which minor deviations are quite normal). Speed control allows stabilizing speed independently of battery voltage, load (weight, surface conditions, slope, etc.) and manufacturing tolerances. If we would try to drive a robot at a fixed PWM value, the robot's speed would be extremely dependent on effective motor load and battery voltage. Additionally, manufacturing tolerances would result in different speeds for the left and right motor.

The speed control routine is also responsible for reversing the motor turning direction, as any reversing operation at 15cm/s might considerably accelerate wear out of the motors and gears. If a motor direction change has to be performed, the robot's speed will automatically be reduced to zero, followed by the direction change and subsequent acceleration up to the former setpoint speed.

In addition to speed and direction control the system also monitors the current consumption of the motors. It will automatically stop the motors in over current conditions. These safety precaution prevents motor overload and overheating, which may damage the motors over time.

If three overcurrent events occur within 20 seconds, the protection system will perform an emergency shutdown and start flashing the four Status LEDs. Then the Robot has to be resetted in order to continue operation.

Additionally the system monitors failing encoders or motors (which may happen if you've tinkered too much with it...). Whenever this happens, the motionControl function would ramp the PWM value up to maximum and the robot could get out of control due to this... which of course must be considered as a quite undesirable experience! Anyway, the robot will be halted completely in this case.

Just to keep this concise, we also included the functions for driving specific distances and rotating specific angles in the motionControl function.

As you can see, the function is very important for the automatic motor control. As a matter of fact, the motionControl function itself does not have any parameters like e.g. the desired speed. Operating parameters need to be set through other functions, which will be described in detail now.

```
void moveAtSpeed(uint8_t desired_speed_left, uint8_t desired_speed_right)
```

This function adjusts the setpoint speed. Both parameters will define the desired speed for the left and right motor. Frequently calling the motionControl function (as described in the previous chapter) results in regulation of the speed to the setpoint values. Setting these values to zero, initiates a slowdown, followed by complete deactivation of the PWM modules.

```
getDesSpeedLeft() and getDesSpeedRight()
```

These macros allow you to read the actual setpoint speed values.

Usage is quite simple as you can see in the following example program:

```
1  #include "RP6RobotBaseLib.h"
2
3  int main(void)
4  {
5        initRobotBase(); // Initialize the Microcontroller
6
7        powerON();  // Activate Encoders & Motor current sensors (IMPORTANT!)
8
9        moveAtSpeed(70,70); // set desired speed
10
11       while(true)
12       {
13             // Frequently call the motionControl function from the
14             // main loop - it will adjust both motor speeds:
15             task_motionControl();
16             task_ADC(); // has to be called for Motor current sensors!
17       }
18       return 0;
   }
```

... and now RP6 will start moving directly! Of course the robot will not react on any obstacles and is moving forwards only! The system will only try to regulate the speed level and automatically adjust motor power – e.g. in ascending or descending a ramp.

BE CAREFUL: This behaviour may be very dangerous for your own fingers – Take care to keep your fingers away from the caterpillar tracks and wheels, and keep clear of the area between printed circuit board and the caterpillar tracks! There is a considerable **risk of injury**! As already explained, the motor power will automatically be increased and the motors are quite powerful!

Speed parameters for the moveAtSpeed function is not specified in cm/s or equivalent units, but in a rotational Velocity unit.

After all, the robot's speed depends on the real circumference of caterpillar tracks and wheels or in other words the encoder's resolution. There are considerable tolerances from 0.23 up to 0.25mm for each encoder segment. Thus the Encoder resolution has to be measured!

The system will measure the rotational speed at intervals of 200ms which is equivalent to a rate of 5x pro second. So the unit is "Encoder Segments per 200ms". A value of 70 as it has been used in the example on the previous page has to be interpreted

as $70 \cdot 5 = 350$ counted encoder-segments per second (corresponding to ca. 8 up to 8.7 cm/s – depending on the real encoder resolution). The minimal controllable rotational speed is around $10 \cdot 5 = 50$ and the maximal rotational speed is approximately $200 \cdot 5 = 1000$. Chapter 2 already mentioned the reasons for this speed limit, but we even recommend to limit it to a value of 160 for continous movements and use 200 only for short times!

getLeftSpeed() and **getRightSpeed()**

These macros allow you to read the measured rotational speed. They will return values with the same unit as described above.

void changeDirection(uint8_t dir)

This function will set the motors rotational direction. As already discussed, first the robobt will decelerate, then change the direction and finally accelerate to the previous setpoint speed.

The following parameters are supported:

FWD – ForWarDs
BWD – BackWarDs
LEFT – rotate left
RIGHT – rotate right

The macro:

getDirection()

allows you to read the current direction.

Example:

```
1  #include "RP6RobotBaseLib.h"
2
3  int main(void)
4  {
5      initRobotBase(); // Initialize the Microcontroller
6      powerON();         // Activate Encoders and Motor current sensors!
7
8      moveAtSpeed(60,60); // Set desired speed
9      startStopwatch1();  // Start Stopwatch1
10
11     while(true)
12     {
13         if(getStopwatch1() > 4000) // Have 4000ms (= 4s) passed by?
14         {
15             // Change moving direction:
16             if(getDirection() == FWD) // If we are driving forwards,
17                 changeDirection(BWD); // then set direction to backwards!
18             else if(getDirection() == BWD) // If we are driving backwards,
19                 changeDirection(FWD); // then set direction to forwards!
20             setStopwatch1(0); // Reset Stopwatch1
21         }
22         task_motionControl(); // Automatic motion control
23         task_ADC(); // has to be called for the current sensors.
24     }
25     return 0;
26 }
```

In this example program, the RP6 will first drive forwards – which is the default setting for movements after a reset. We are using one of the stopwatches to wait 4 seconds and then the direction is reversed. In line 16 and 18 the current rotational direction is determined and changed accordingly. This repeats at intervals of 4 seconds, which will cause the robot to drive forwards and backwards the whole time.

Clearly the robot will still ignore any obstacles!

Its not that easy to drive specific distances with the functions discussed until now. There are two special functions for this purpose:

```
void move(uint8_t desired_speed, uint8_t dir, uint16_t distance,
                  uint8_t blocking)
```

The move-function allows the Robot to drive a specific distance. You need to pass desired speed, direction (FWD or BWD) and distance in encoder counts.

The macro:

```
DIST_MM(DISTANCE)
```

is helpful for converting a distance from millimetres to encoder counts. Of course you will need to calibrate the encoder resolution before (see appendix). The sample program further down below shows how to use this.

The robot will try to drive the desired distance as accurately as possible. The motionControl function starts by accelerating to the setpoint speed and slows down the robot shortly before the distance is reached to avoid overshooting. Accuracy is around 5mm, which usually may be considered to be all right.

The function does not support driving very short distances under 5cm, but this can be improved, of course!

The trailing parameter, named "blocking" is a special feature, which needs a detailed description.

Usually the function will only set a few variables and immediately returns to the program. The robot is then controlled by the motionControl-function "in background". This is useful for performing other jobs such as avoiding obstacles. However, if the robot just has to follow a predefined geometric figure, you can change this with the blocking parameter.

Setting the parameter "true" (this means 1), the function will call the motionControl function in a loop until the predefined distance has been reached. The program does not leave this function – instead it will "block" the normal program flow for the required time.

Setting the parameter "false" will cause the function to perform as described previously. It will immediately return after it set the command "start to drive a predefined distance". If you call other functions, which set the speed or give other movement commands, the program could behave incorrectly. You will have to either wait for the original movement command to finish, or alternatively you can abort the command.

The function:

```
uint8_t isMovementComplete(void)
```

can be used to check if movement-commands are completed. If there are unfinished movement-commands, the return value will be "false".

Whenever a move-command has to be aborted, e.g. at detection of an obstacle, you can terminate all movements by calling:

```
void stop(void)
```

which will stop all movements.

It would be already possible to use the move function for rotating by simply setting the direction parameter to LEFT or RIGHT instead of FWD or BWD and specifying a suitable distance value corresponding to an angle. This is a rather clumsy method and does not perform very well. For this reason we provided a dedicated function for rotation on the spot:

```
void rotate(uint8_t desired_speed, uint8_t dir, uint16_t angle,
                  uint8_t blocking)
```

This function behaves just like the "move"-command, the only difference is that you need to specify an angle instead of a distance. The blocking parameter can be used with this function, too.

The following example program shows how to use both functions:

```
1  #include "RP6RobotBaseLib.h"
2
3  int main(void)
4  {
5        initRobotBase();
6        setLEDs(0b111111);
7        mSleep(1500);
8
9        powerON(); // Activate encoders and motor current sensors!
10
11       while(true)
12       {
13             setLEDs(0b100100);
14             move(60, FWD, DIST_MM(300), true); // Move 30cm forward
15             setLEDs(0b100000);
16             rotate(50, LEFT, 180, true);  // Rotate 180° to the left
17             setLEDs(0b100100);
18             move(60, FWD, DIST_MM(300), true); // Move 30cm forward
19             setLEDs(0b000100);
20             rotate(50, RIGHT, 180, true); // Rotate 180° to the right
21       }
22       return 0;
23 }
```

The Robot will move 30cm forward, rotate 180° to the left, move 30cm backwards, rotate 180° to the right and start from the beginning. If you would set all of the blocking parameters to false, the program would not work at all. The main loop does not call the task_motionControl function and all movement function calls are in one sequence. Changing only one blocking parameter to false, the program will not work as intended anymore. One of the movement phases will be skipped completely then.

In order to perform such sequential processes, we always have to set the blocking-parameter to true!
You may even perform rudimentary reaction on obstacles with the blocking-parameters set to true – by using event handlera. These event handlers will be called anyway, no matter if the blocking parameter is set true! However this method may fail in more complex situations.

In general, for avoiding obstacles, interpreting master controller commands and for similar processes we recommend to use the non-blocking mode and set the blocking parameter to false.
As we mentioned earlier in this chapter, this mode allows the move/rotate-functions to control the robot's movement independently of other program flow.

You will find a few detailed examples to this topic on the CD.

4.6.10. task_RP6System()

In the last few chapters we have learned, that it is essential to frequently call four functions within the main loop for correct operation of ACS/IRCOMM, motion control, bumpers and ADC evaluation in the background. Just to simplify this and to help keeping a better overview of your program, the RP6Library provides the following function:

```
void task_RP6System(void)
```

which will sequentially call the functions:

```
task_ADC();
task_ACS();
task_bumpers();
task_motionControl();
```

Most of the sample programs on the CD will only use this function – we will hardly need one of the other functions directly.

4.6.11. I²C Bus Functions

At the end of this chapter we will focus on the I²C-Bus functions, which can be used for communication with other Microcontrollers and expansion modules.

There are two versions of I²C-Bus functions – one for slave and another one for master mode.

Attention: You can <u>not</u> use both versions simultaneously!

You may include one of both versions only and you need to make sure that they are listed in the makefile. Apropriate entries have been added to the Makefiles of the example programs already – *however they have been outcommented in most of the examples*. Once again: use only one of these entries! Otherwise the compiler will issue an error message (this is because the TWI Interrupt Vector would be defined twice if you would include both versions).

4.6.11.1. I²C Slave

On the robot base unit, the slave mode is most important, because it is a very common task to add another Microcontroller to the Robot in order to control it. There is an Example program, which allows you to access nearly all functions of the robot base unit through the I²C Bus (RP6Base_I2CSlave).

Basically both, master mode and slave mode use interrupts. It is not easy to implement I2C Slave mode in pure Software (at least not with reasonable effort). The master mode could be easily implemented in Software, but in order to keep the structure of both similar, we also used interrupts this mode. Another advantage is, that the master mode transmissions can be performed in background to save some time.

```
void I2CTWI_initSlave(uint8_t address)
```

This function will initialize the Microcontroller's TWI module as I²C Slave and allows you to define the address as parameter. In the address you can simultaneously define, whether the controller should react on so-called "General Calls" or not. When the bus is addressed with a 0, this is called "General Call". For example you may use this functionality to easily switch all controllers on the bus to a power saving mode simultaneously.

Examples:

```
I2CTWI_initSlave( adr |  TWI_GENERAL_CALL_ENABLE ); // Enable general call
I2CTWI_initSlave(adr); // Disable general call
```

I²C Registers

Usual I²C peripherals can be controlled through a few readable/writeable registers. Therefore the slave routines are designed to provide a number of "Registers" (in this application an array of 8 Bit variables), which may be written or read by the master device. In order to read data from a register or to write data into a register, the master device has to send the slave address and subsequently the register address.

There are two uint8_t arrays. One for readable and one for writeable Registers.

The arrays and the variable for general calls are called:

I2CTWI_readRegisters, I2CTWI_writeRegisters and I2CTWI_genCallCMD

The readable registers are named I2CTWI_readRegisters and the writeable registers I2CTWI_writeRegisters. The I2CTWI_genCallCMD variable stores the most recently received General Call command. Data exchange in slave mode works completely with these registers.

In order to make specific data available on the bus, you have to put it in the array I2CTWI_readRegisters. A master may now read this data by addressing the corresponding array position (equivalent to a register number). For instance, if a master needs to read sensor data from a slave, the slave first has to put the information into a predefined location in the I2CTWI_readRegisters array. Then the master can read the data by transferring the register number to the slave and subsequently read the information. The register number will be automatically incremented to allow the master to read several registers in a single run.

A similar procedure has to be followed to write data. The master initially transmits the register number and then starts transferring data. In analogy to the reading process, the register number will be automatically incremented to allow the master to write several registers in a single run. The slave performs this completely in background by using interrupts.

While writing to the slave, data structures may easily get inconsistent if the data is used simultaneously. If you read data from one register, another one belonging to this register may have been overwritten by the master in the meantime. A better way to handle these transfers is an intermediate storage location. Reading data may also lead to inconsistencies in handling related variables (e.g. low and high bytes for 16 Bit Variables).

I2CTWI_readBusy and I2CTWI_writeBusy

The interrupt routine sets the variable I2CTWI_writeBuisy to true – and this can be used to check for writing access to this data. If it is set to "false", we may transfer data from the registers into temporary variables and use these for further processing.

The example program on the following page and the "slave"-example program on the CD demonstrate this – there is a command register, used by the master device to transfer commands to the slave (e.g. "start driving forwards at a speed of 100"). The main loop of the slave device is constantly evaluating register 0 as long as I2CTWI_busy is set to false. The arrival of a master command in register 0 will be followed by transferring data from register 0 and registers 1 up to 6 into temporary variables, which may be evaluated afterwards. Parameter usage depends on the content of the command variable. E.g. parameter 1 may describe a speed value for a movement command and parameter 2 the direction. Any other parameters would be ignored with this command.

The variable I2CTWI_readBusy works similar – it is set whenever a register is being read and allows us to check write ability for registers and to prevent inconsistencies. Current implementations cannot guarantee consistency to 100% as this would imply deactivation of TWI-Interrupt during write processes to the registers, which might cause other problems...

This example shows how a very simple slave program could look like:

```
1  #include "RP6RobotBaseLib.h"
2  #include "RP6I2CslaveTWI.h"  // Include the I2C Library file (!!!)
3       // ATTENTION: do not forget to add this to the Makefile (!!!)
4
5  #define CMD_SET_LEDS 3 // LED command, which should be received
6                         // through the I2C Bus
7  int main(void)
8  {
9        initRobotBase();
10       I2CTWI_initSlave(10); // Initialise TWI set slave address to 10
11       powerON();
12
13       while(true)
14       {
15             // did we receive some command and is there NO write access?
16             if(I2CTWI_writeRegisters[0] && !I2CTWI_writeBusy)
17             {
18                   // save register contents:
19                   uint8_t cmd = I2CTWI_writeRegisters[0];
20                   I2CTWI_writeRegisters[0] = 0; // and reset cmd reg (!!!)
21                   uint8_t param = I2CTWI_writeRegisters[1]; // Parameter
22
23                   if(cmd == CMD_SET_LEDS) // LED command received?
24                         setLEDs(param); // set LEDs with the parameter
25             }
26             if(!I2CTWI_readBusy) // No read activities?
27                   // Proceed by writing current LED state to register 0:
28                   I2CTWI_readRegisters[0] = statusLEDs.byte;
29       }
30       return 0;
31 }
```

The program itself will not perform anything (visible), thus you need a master to control the slave device. In this case, the slave can be accessed with address 10 on the I²C Bus (see line 10). It provides two registers for writing and one register for reading data. The first register (= Register number 0) is used for receiving commands. This simplified example uses the command "3" to set the LEDs (can be any number). On reception of any command - and no write access (see line 16) - the program will store the contents of command register 0 to the variable "cmd" (line 19) and reset the command register 0 to prevent repeated command execution! The program proceeds by storing the parameter from register 1 to another temporary variable and check the reception of command 3 (line 23). If the comparison is true, the LEDs will be set by applying the received parameter value.
Line 26 checks if there is no read access in progress and stores the current LED register value to the readable register 0.
When the Controller on the Mainboard is programmed like this, a Master controller can set the LEDs on the Mainboard through the I²C Bus and read back their current state.

The program will not perform anything else – a more detailed example program, which allows control of virtually all available robot funtions can be found on the CD. The sample program above only demonstrates the basic principles.
Basically there are 16 writeable and 48 readable registers. Whoever needs more or less registers may adapt the corresponding definitions in the RP6Library file RP6I2C-SlaveTWI.h.

4.6.11.2. I²C Master

In Master mode, the ATMEGA32's TWI Module can be used to control other devices/Microcontrollers/sensors through the I²C Bus.

```
void I2CTWI_initMaster(FREQ)
```

This function initializes the TWI Module as master. Of course the Master mode does not require an address – but we have to specify the data transmission frequency for the TWI Module. You have to define the frequency in kHz by using the parameter FREQ. A usual value is 100kHz, which can be set by a parameter value of 100. You may speed up transmission by using values up to 400. Do not exceed the upper limit 400 for the TWI Module!

According to Atmel's specifications (see data sheet) the TWI Module of the MEGA32 may only be operated at rates of up to a maximal 220kBit/s in master mode! A transmission frequency of 400kHz would require a clock frequency over 14.4MHz, but for power saving reasons, we chose a 8MHz clock frequency. But this is only a small timing issue and it may still work properly for you. In slave mode, this does not cause any problems at all and 400kBit/s can be used. If you really need that fast communication, you can either try to set the 400kBit/s mode and see if it works properly with your specific slave devices or use the RP6 CONTROL M32 expansion module which is clocked at 16MHz. Usually it should also work with 8MHz as this is only a small timing issue. But we can not guarantee this!

Data transmission

There are a number of functions for transferring data with the I²C Bus. Basically these functions are all quite similar, but they allow different number of bytes for transmission.

```
void I2CTWI_transmitByte(uint8_t adr, uint8_t data)
```

transfers one byte to the specified address.

```
void I2CTWI_transmit2Bytes(uint8_t adr, uint8_t data1, uint8_t data2)
```

transfers two bytes to the specified address. You will need this function frequently as a number of I²C-devices requests data formats like:

Slave address – Register address – data

```
void I2CTWI_transmit3Bytes(uint8_t adr, uint8_t data1, uint8_t data2,
uint8_t data3)
```

is used quite frequently as well, especially the previously described slave program communicates with data formats like:

Slave Address – command register – command – parameter1

```
void I2CTWI_transmitBytes(uint8_t targetAdr, uint8_t *msg,
                          uint8_t numberOfBytes)
```

Basically this function will transfer up to 20 Bytes to the specified address. For transferring greater data blocks, you may increase the I2CTWI_BUFFER_SIZE constant in the Header file.

In order to specify a register for data transfer you may simply use the first byte of the buffer.

Using all these functions is rather simple, e.g.:

```
I2CTWI_transmit2Bytes(10, 2, 128);
I2CTWI_transmit2Bytes(10, 3, 14);
I2CTWI_transmit3Bytes(64, 12, 98, 120);
```

The preceding example transmits successively two times two bytes to a slave device with address 10 and additionally three bytes to a slave with address 64.

The other transmitXBytes-functions are used in a similar way.

Another example:

```
uint8_t messageBuf[4];
messageBuf[0] = 2; // Here you may optionally specify the addressed register.
messageBuf[1] = 244;  // Data...
messageBuf[2] = 231;
messageBuf[3] = 123;
messageBuf[4] = 40;
I2CTWI_transmitBytes(10,&messageBuf[0],5);
```

Like this, you can transmit several Bytes (5 in this case) via the I²C Bus.

The previously described functions will not block the program flow, unless the I²C Interface is busy. A busy I²C Interface will cause the functions to wait until all transfers are completed. Therefore checking for completion before calling the function will allow you to perform other jobs while data transfer is in progress. Data transfer with the I²C Bus is relatively time consuming compared to the micro-controller's speed and you can save some time by checking this.

The following macro indicates whether the TWI Module is busy or not:

```
I2CTWI_isBusy()
```

If the module is free, you may transfer new data.

Data reception

The RP6Library provides several options for data reception. First we present a few blocking functions, which have been designed in analogy to the writing functions. Additionally we will discuss functions for receiving data in the background.

First of all the simple blocking function for reading data:

```
uint8_t I2CTWI_readByte(uint8_t targetAdr);
```

This function reads one byte from a slave device. This functions can not be used alone, most likely you will have to transfer the register number with I2CTWI_transmitByte before.

For example if you would like to read register 22 from a slave device with address 10:

```
I2CTWI_transmitByte(10, 22);
uint8_t result = I2CTWI_readByte(10);
```

The following function allows you to read several bytes:

```
void I2CTWI_readBytes(uint8_t targetAdr, uint8_t * messageBuffer,
                      uint8_t numberOfBytes);
```

Example:

```
I2CTWI_transmitByte(10, 22);
uint8_t results[6];
I2CTWI_readBytes(10,results,5);
```

This code snippet reads 5 bytes from register 22 of a slave device with address 10. If this data is really read from Register 22 varies from slave to slave device. Some will increment the register number automatically (just like the slave code of the RP6Library does) and other work completely different. You have to check the documentation of your devices about this!

Reading data in background is a bit more complex task. First of all you will have to start a request for a number of bytes from a slave. A background process will be started to retrieve these bytes from the slave device. In the meantime the controller is allowed to perform other jobs, intermittently being disturbed by the interrupt routine. Of course, we will frequently have to call a function from the main loop to check for the arrival of requested data from a slave device or for a failure condition in the communication process. On data arrival, this function automatically calls a predefined Event Handler function for further data processing and it may immediately start retrieving the next set of data from other registers. Each request will be managed by its own ID.

```
void task_I2CTWI(void)
```

is the function, which has to be called frequently from the main loop. This task has been designed to check all transfers for error-free completion and to call the Event Handler as required.

```
void I2CTWI_requestDataFromDevice(uint8_t requestAdr, uint8_t requestID,
                                  uint8_t numberOfBytes)
```

This function allows you to request data from a slave device. After calling the function a background process will automatically retrieve the data as described above.

Subsequently we can fetch the data by calling the function:

```
void I2CTWI_getReceivedData(uint8_t *msg, uint8_t msgSize)
```

We can fetch the data as soon as the Event Handler is called. The Handler has to be registered by calling the function:

```
void I2CTWI_setRequestedDataReadyHandler(void (*requestedDataReadyHandler)
(uint8_t))
```

The Event Handler must have the following signature pattern:

```
void I2C_requestedDataReady(uint8_t dataRequestID)
```

This Handler will be called with the ID of the data request as parameter. We can use the ID to differentiate between different slave devices.

Apart from the requestedDataReady Handler there is an Event Handler for error-processing. Whenever an error occurs in data transfers, e.g. by a non-responding slave, a special Event Handler will be called. It has to be registered with this function:

```
void I2CTWI_setTransmissionErrorHandler(void (*transmissionErrorHandler)
(uint8_t))
```

The handler must have the signature pattern:

```
void I2C_transmissionError(uint8_t errorState)
```

defining an error state-code as parameter. You can find an overview of these error codes in the "I²C Master Mode"-header file.

In fact, you can use this Event Handler for detecting errors for all, background, foreground and blocking functions.

There are a few example programs for Master Mode on the CD – and we will discuss then in detail in the next chapter.

4.7. Example Programs

The CD contains quite a few short example programs, demonstrating the robot's basic functionality. Most of these examples are quite simple and far away from optimized solutions. You have to consider most of the progrmas as a starting point for your own programs. This is absolutely intentional, as this leaves some interesting tasks up to you – would be quite boring just to load pre-fabricated programs onto the robot, wouldn't it?

A few example programs focus more or less on experienced users. Especially the behaviour controlled robot programs, which allow the robot to simulate an insect's behaviour, belong to this category. With one of the example programs, the RP6 behaves like a moth and searches for bright light sources and avoids obstacles. However explaining details about this would lead us far beyond the scope of this manual and for all advanced applications we have to refer to the relevant literature.

Of course you may exchange your own programs with other users through the Internet. The RP6Library and all example programs have been released under the open source licencse "GPL" (General Public License). This allows modification and publication of derived programs according to the GPL, which of course implies you must release your derived programs under the GPL as well.

The AVR Microcontroller family is very popular and there are plenty of example programs for the MEGA32 available on the Internet already. However you will have to pay attention to adapt example programs to the RP6 hardware and the RP6Library. Otherwise programs will be malfunctioning most likely (common problems are different pin assignments, different use of hardware modules like timers, different clock frequencies, etc.)!

Except for the I²C Bus applications, all example programs have been designed to run on the robot base unit only – without any expansion modules. Although this will usually not interfere with anything, you should start using expansion modules only after you tried out all example programs and got familiar with the robot base unit.

Any programmable expansion kit is delivered with appropriate example programs. Alternative and additional software may also be available on our homepage (example programs for the RP6 CONTROL M32 are included on the RP6 CD).

By the way, programming of expansion modules like the RP6-M32 can be a bit easier in some cases, as you do not need to care about too critical timing issues like ACS, motor control and so on.

Additionally the RP6-M32 provides you with a lot of additional CPU power and memory for more demanding tasks.

Example 1: "Hello World"-Program featuring a running LED-light
Directory: <RP6Examples>\RP6BaseExamples\Example_01_LEDs
File: RP6Base_LEDs.c

The program will output messages on the serial interface, thus you should connect the robot to your PC and watch the output in the RP6Loader Terminal!

The robot does not move in this example program! You may simply put the robot on a table next to your PC.

This program outputs a small "Hello World" text through the serial interface and subsequently executes a running light with the LEDs on the Mainboard.

One minor detail in the running light is the "shift-left"-Operator, which has been used in previous samples without any explanation:

```
1  setLEDs(runningLight); // Set the LEDs
2  runningLight <<= 1;    // Next LED (shift-left Operation)
3  if(runningLight > 32)  // Last LED?
4      runningLight = 1;  // Yes, so, let's start again!
```

We will explain this operator right now. Basically a shift-left operation "<<" (see line 2), allows you to shift the Bits in a variable for a predefined number of digits to the left. Of course you may also use the equivalent shift-right operator ">>".

This means that `runningLight <<= 1;` shifts all Bits in the runningLight variable to the left by one digit. By the way: this is just a shortcut for:

```
runningLight = runningLight << 1;
```

which works similar for += and *=.

Initially the runningLight variable starts with a value of 1, which means only the Bit number 1 is set. Each shift operation will move this Bit stepwise to the left.

Using this variable for LED control will result in a "moving" light-dot --> a running light! In order to allow the human eye to follow the moving light dot, mSleep is used to generate 100ms delays in between the loop cycles.

If Bit number 7 in the runningLight variable is set (which implies reaching a value > 32), we have to return to the start by only setting Bit 1 in the variable (at lines 3 and 4).

Example 2: Some more applications for the serial interface
Directory: <RP6Examples>\RP6BaseExamples\Example_02_UART_01
File: RP6Base_SerialInterface_01.c

This program will output messages on the serial interface

The robot does not move in this example program!

This sample program demonstrates the usage of the functions writeInteger and writeIntegerLength, which are used to output a few integer values in different formats through the serial interface.

Furthermore the new "timer" variable introduced in Version 1.3 of the RP6Lib is demonstrated. It can be used for time measurements with a resolution of 100µs.

Example 3: Question-and-Answer Program
Directory: <RP6Examples>\RP6BaseExamples\Example_02_UART_02
File: RP6Base_SerialInterface_02.c

This program will output messages on the serial interface

The robot does not move in this example program!

This more complex example is a little question-and-answer dialog, in which the robot will ask four simple questions and you may reply by entering any answer in the terminal. The robot will react with a text message or by starting a short running light. The program demonstrates how to retrieve data from the serial interface and use this data for further processing. Additionally you will learn how to use the writeStringLength-function and there is another example for a switch-case construct.

The old functions for data reception through the serial interface have been replaced by more powerful one in the latest RP6Lib version. This example program demonstrates their usage directly. Now it is possible to process inputs of any length and to react better on wrong inputs.

Example 4: Stopwatches Demo Program
Directory: <RP6Examples>\RP6BaseExamples\Example_03_Stopwatches
File: RP6Base_Stopwatches.c

This program will output messages on the serial interface

The robot does not move in this example program!

This program is using four of the stopwatches. The first one is generating a running light with 100ms refresh Interval and the others are being used for three different counters incrementing in different intervals and outputting their values through the serial interface.

Example 5: ACS & Bumper Demo Program
Directory: <RP6Examples>\RP6BaseExamples\Example_04_ACS
File: RP6Base_ACS.c

This program will output messages on the serial interface

The robot does not move in this example program!

Although the filename only indicates an ACS application, this program demonstrates ACS a*nd* Bumper usage with the corresponding Event Handlers. The example shows the state of both ACS channels with the LEDs and outputs it with the serial interface. The bumper status is only transferred with the serial interface.

Just move your hands in front of the robot and hit the bumpers!

You can use the program for testing the ACS with several different objects – just to check which objects are detected well. You may also change the transmit power of the ACS! The default value has been set to medium level.

Example 6: The Robot is driving in a circle
Directory: <RP6Examples>\RP6BaseExamples\Example_05_Move_01
File: RP6Base_Move_01.c

ATTENTION: The robot will move in this example program! Please remove the cable between the PC and the robot after program upload and put the robot in a big free area! You must provide a free area of at least 1m x 1m or even 2m x 2m.

Finally the robot is ready to start its engines! The example program lets the robot dirve in circles by running both motors at different speeds. Please provide enough of free space for the movements and allow the robot to move freely in this and all of the following programs! Some of the programs require a free area of about 1 or 2 square meters.

If the robot hits an obstacle during the circle movements, it will stop and two LEDs will start blinking! The program now waits to be restarted.

Example 7: The Robot drives forwards and backwards – with a 180° rotation
Directory: <RP6Examples>\RP6BaseExamples\Example_05_Move_02
File: RP6Base_Move_02.c

ATTENTION: The robot will move in this example program!

We already demonstrated time controlled movement with a small example in the previous RP6Library chapter. In contrast, this example will let the robot drive forwards for a specific distance, perform a 180° turn, drive backwards for the same distance, do a second 180° turn and start from the beginning again.

The example uses the blocking mode of the move and rotate functions, which will automatically call the task_RP6System function. Therefore we do not have to call this function frequently.

Example 8: The Robot moves in a square path
Directory: <RP6Examples>\RP6BaseExamples\Example_05_Move_03
File: RP6Base_Move_03.c

ATTENTION: The robot will move in this example program!

Now the robot will try to move in a square path with 30cm edge length. It rotates after each complete tour and moves on in the opposite direction.

This works fine with exactly calibrated encoders only, otherwise the 90° angles may be inaccurate (e.g. 80° or 98°). Because of this, the example also demonstrates the difficulties in accurately controlling a caterpillar vehicle with encoder feedback signals only. In chapter 2 we have discussed these problems already. Depending on surface conditions of the floor, the caterpillars will be slipping and sliding, which will reduce the real distance compared to the measured values. Clever programming could enable us to compensate these errors, but most likely it is not possible to achieve 100% accuracy with the encoders only. Other external sensor systems have to be used for exact positioning control (see appendix).

Example 9: Excursion – Finite state machines
Directory: <RP6Examples>\RP6BaseExamples\Example_05_Move_04_FSM
File: RP6Base_Move_04_FSM.c

This program will output messages on the serial interface

The robot does not move in this example program!

For more complex programs, the simple methods we discussed so far may not be sufficient.

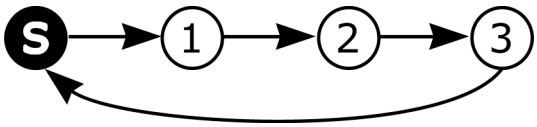
For instance, behaviour controlled robots require so-called Finite State Machines (shortcut FSMs) – this would not be that easy with simple methods. This example program demonstrates a simple FSM changing its state if a bumper is hit. To understand the program's operation, just try it out and press the bumpers twice while you carefully observe the terminal's display and the status-LEDs. Press and then release the bumpers slowly!

In C, most Finite State Machines will be designed by using switch-case constructs, or alternatively a bunch of conditional statements by using "if-else-if-else"-constructs... however switch/case will result in more clearly arranged source code.

Let's have a look at a simple example:

```
1  #include "RP6RobotBaseLib.h"
2
3  #define STATE_START  0
4  #define STATE_1      1
5  #define STATE_2      2
6  #define STATE_3      3
7
8  uint8_t state = STATE_START;
9
10 void simple_stateMachine(void)
11 {
12    switch(state)
13    {
14       case STATE_START: writeString("\nSTART\n");   state = STATE_1;
15         break;
16       case STATE_1:     writeString("State 1\n");   state = STATE_2;
17         break;
18       case STATE_2:     writeString("State 2\n");   state = STATE_3;
19         break;
20       case STATE_3:     writeString("State 3\n");   state = STATE_START;
21         break;
22    }
23 }
24
25 int main(void)
26 {
27    initRobotBase();
28    while(true)
29    {
30        simple_stateMachine();
31        mSleep(500);
32    }
33    return 0;
34 }
```

To show you the basic principle, this example has been reduced to its essentials. A state machine consists of different states and transitions between them. In our example we have four states: STATE_START and STATE_1 up to 3. We may also visualize the state machine above with the following graph:



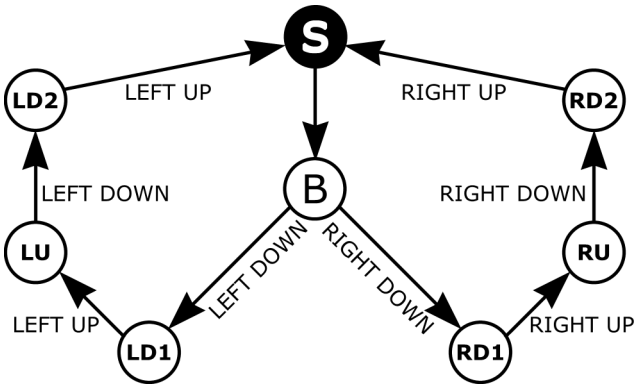
"S" is the start state. We do not use any conditional state transitions here and therefore the system will change states up to state 3 and restart the sequence from state S in each step. For better visualisation, there is a 500ms delay between the state changes.
The program will generate the following output:

```
START
State 1
State 2
State 3

START
State 1
State 2
...
... etc.
```

The program may prolong this list endlessly.

The example program in file **RP6Base_Move_04_FSM.c** contains a more complex state machine, featuring 8 states. The basic structure is shown in the following graph (in this overview we abbreviated the labels of the states):



The state machine starts at state S and immediately transits to state B (while displaying a short text message). Having arrived at state B the system will wait until you hit one of the bumpers.
If you press down the left bumper, the state machine will transit to state LD1 ("Left Down 1") and to state RD1 on pressing the right bumper. Now the next transition has the condition that the bumper is released again. If this happens, it will change state to either LU or RU. In both of these states, the machine will only react on one of the bumper switches (left or right) and ignore any activities on the other side. Only if you press the selected bumper once again, the state machine will transit to the states RD2 and LD2, respectively. Releasing the bumper again, the system returns to the S-state.
Of course this example program outputs an appropriate message for each state transition and will set the Status LEDs accordingly, but there was not enough space in the graph for this additonal information. It just shows the general FSM layout.

Example 10: Finite state machines, Part 2
Directory: <RP6Examples>\RP6BaseExamples\Example_05_Move_04_FSM2
File: RP6Base_Move_04_FSM2.c

ATTENTION: The robot will move in this example program!

Now let's try out an FSM with a moving robot! The program is quite simple. First Status LED 5 is blinking. With this, the Robot wants to tell us: "Would somebody push the left Bumper, please?". When you do this, RP6 moves backwards for about 15cm and Status LED 2 starts blinking. Obviously, you are now supposed to push the right Bumper, which lets the Robot move 15cm forwards again and the cycle starts from the beginning.

The structure of the FSM is nothing special. It is nearly the same as in the first example you saw above, where the program wrote "START State1 State2..." all the time, but extended by adding a few conditional statements. In order to make the robot wait until a movement is complete, two states use the "isMovementComplete()" function. Within these states, the program additionally executes a simple code fragment using a stopwatch to toggle an LED in 500ms intervals. Of course, this fragment may be replaced by any other code sequence – e.g. the running light from example 1.

We could write lots of pages about FSMs and similar topics, but as already stated – this is only a manual and no reference book for Finite State Machines. So let's go on with the example programs...

Example 11: Behaviour controlled robots
Directory: <RP6Examples>\RP6BaseExamples\Example_05_Move_04
File: RP6Base_Move_04.c

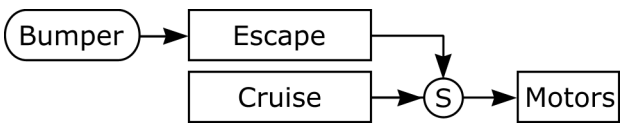
ATTENTION: The robot will move in this example program!

The previous automaton examples lead the way to the following program, which implements a simple behaviour controlled robot. To reduce the complexity we only use two small behaviours. We will extend this basic program step by step in the following program examples in order to create a simple insect-like behaviour. The first step only provides our "insect" with two tiny sensors, which report collisions.

Our first two behaviours are named "Cruise" and "Escape". The "Cruise" Behaviour will only tell the Robot to "drive forwards" and does not perform any other activities. Of course, we may add some other things, e.g. driving a curved track after some time or accelerating and slowing down according to the battery voltage level. However, for modularity in a well designed program we advise you to seperate this in additional behaviours.

In contrast to "Cruise", the "Escape" behaviour is quite complex. It will get active as soon as the bumpers detect collisions. According to which bumper was hit, the behaviour lets the Robot drive back for a few centimetres, roate a little bit and subsequently returns control to the "Cruise" behaviour.

We can visualize this structure in a simple digram:



By using the "Suppressor" S the "Escape" behaviour suppresses the "Cruise"-behavi-

our's outputs and forces the system to feed the motor-control with its own commands. Obviously "Escape" has higher priority than "Cruise".

With these two very basic behaviours, the robot will already start to cruise around and explore its' environment. The sensor system simply consists of two sensors for detecting collisions. Of course this is nothing for very complex behaviours.

Just imagine a situation in which you would have to move around inside your home restricting your senses to two fingertips – no eyes, no ears, none of your other senses to help you except these two fingertips, which have to be held straight in front of you all the time...

The more sensors the robot gets, the more the complexity of the robot's behaviours may increase! Typically an insect is equipped with a vast number of sensors, which can even provide analoge values indicating the sensed intensity. You may easily imagine how the system's "intelligence" could be increased by equipping the robot with insect's compound eyes...

Here we use a procedure invented by Rodney Brooks (`http://people.csail.mit.edu/brooks/`) around 1985, the so-called Subsumption-Architecture.

The original publication(s) can be found here:

`http://people.csail.mit.edu/brooks/papers/AIM-864.pdf`
`http://people.csail.mit.edu/brooks/papers/how-to-build.pdf`

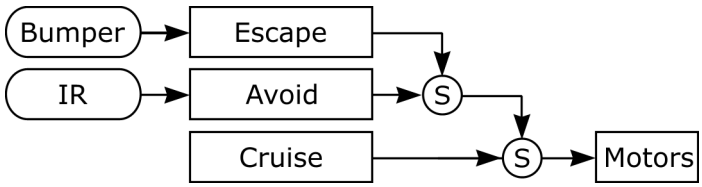
but you may find other relevant documents and summaries

(The Internet provides lots of additional information – simply start searching!)

Example 12: Behaviour controlled Robot 2
Directory: <RP6Examples>\RP6BaseExamples\Example_05_Move_05
File: RP6Base_Move_05.c

ATTENTION: The robot will move in this example program!

Next step is to add an "Avoid" behaviour, which uses the ACS to avoid obstacles instead of first colliding with them. The three behaviours are structured as shown in the following figure:



Of course collisions have to be considered as most important events and therefore the "Escape" behaviour still has the highest priority level. "Escape" will suppress the "Avoid" commands and similarly "Avoid" suppresses the commands of "Cruise".

As soon as the IR-Sensors have detected an obstacle the "Avoid" behaviour – just as its name tells us - will initiate an avoidance-manoeuvre. If the left ACS-channel has detected an obstacle the robot will drive a left turn and a right turn for the right ACS-channel. If both ACS-channels detect obstacles, the robot will rotate to the left or to the right according to the first reporting channel. The robot will even turn a bit longer after both channels report free space again. The program uses stopwatches to control these delay periods.

Example 13: LDRs – Light sensors
Directory: <RP6Examples>\RP6BaseExamples\Example_06_LightDetection
File: RP6Base_LightDetection.c

This program will output messages on the serial interface

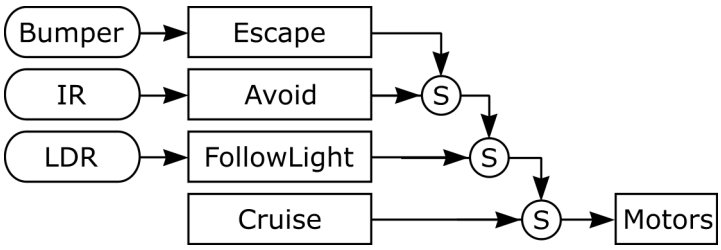
The robot does not move in this example program!

This sample program demonstrates how to use the two light sensors. In order to indicate which sensor is most intensely illuminated (or whether both sensors are illuminated equally) we use the StatusLEDs. Along with measurement values, the program will continuously report these informations to the serial interface, too.

Example 14: Behaviour controlled robots 3
Directory: <RP6Examples>\RP6BaseExamples\Example_07_LightFollowing
File: RP6Base_LightFollowing.c

ATTENTION: The robot will move in this example program!

Of course, the light sensors can be used to extend the robot's design by implementing a behaviour named "FollowLight":



The priority or "FollowLight" is below Escape and Avoid, but above Cruise, which implies the "Cruise" behaviour will never be activated in this example unless the room is very dark (as soon as both LDR-levels drop below 100, the "FollowLight" behaviour will be deactivated).

The "FollowLight"-behaviour tries to follow bright light-sources or the to find the brightest light spot in a room by using the LDRs. Of course the simple algorithm and simple sensor arrangement may fail in special conditions – e.g. if the robot is confronted with a great number of equivalent light sources. However the system can be quite successful in a darkened room, if it tries to find an intense torch light.

This program uses the LEDs in a dual mode – if ACS is not reporting an obstacle, the LEDs indicate which of both light sensors is detecting the highest intensity and as soon as the ACS is reporting an obstacle, the LEDs indicate that.

This program completes the overview of examples for behaviour controlled robots by simulating a simple insect behaviour. The robot behaves like a moth in searching and tracing light sources and simultaneously avoiding obstacles.

The robot may be extended by using additional sensors on expansion modules and by programming your own behaviours and improving existing behaviour functions. These options largely depend on your own creativity and programming capabilities!

Example 15: Remote control by using a universal RC5 IR RC
Directory: <RP6Examples>\RP6BaseExamples\Example_08_TV_REMOTE
File: RP6Base_TV_REMOTE.c

ATTENTION: The robot will move in this example program!

This example program enables you to control the robot just like a Remote Controlled Car by using a standard RC5 IR remote control. Compared to standard functionality for most RC-Cars, we added a few extra movements.

Of course, the vehicle can drive forwards and backwards, but is also able to rotate left and right on the spot. Additionally the robot may drive in a curve forwards/backwards to the left and right. It can even start a single motor in forwards or backwards direction.

In order to keep the program flexible, you can assign all these movement commands to custom key codes matching your specific remote control. Movements are started when a specific key code is received and the motor speed is accelerated slowly. When you release the key, the motor speed is decelerated even slower. This means that the robot accelerates and decelerates that slow intentionally (just in case you wonder about that). It is also possible to immediately stop the robot within one or two centimetres (depending on load and speed) by pressing a special button.

RC5 reception may be used for various other functions – e.g. to start different programs, to set parameters in behaviours, to modify control parameters in speed control, etc...

With some universal remote controls it would be possible to control several robots by using special function keys for selecting different devices – these function keys have to be programmed to transmit RC5 signals and to address different devices – this enables us to control several robots with one singule remote control.

Example 16: I²C Bus Interface – Master Mode
Directory: <RP6Examples>\RP6BaseExamples\Example_I2C_MASTER_01
File: RP6Base_I2C_MASTER_01.c

This program demonstrates how to use the Master Mode of the I²C Bus. Of course you will have to connect a suitable slave device to the I²C Bus before running this program.

By using 8 LEDs, this example program implements a simple "Knight Rider"-running light. The LEDs are connected to a standard 8-Bit I²C Bus Port expander PCF8574. You may insert and solder the PCF8574 on the experiment expansion module (or initially start a test by building the circuit on a breadboard). This already provides the system with 8 free I/O-Ports for evaluating digital sensors or alternatively control small loads, e.g. LEDs. Of course a big load requires external extra transistors or other driver devices.

This chip is a very useful device and you may use a several of them on the same bus. The only thing you have to do is to select an appropriate address for each chip with its three address pins. If you want to use more than 8, you have to use 8 normal PCF8574 and up to 8 more PCF8574A – which is using a different base address. This allows you to address 8 of each type for controlling a total of 16*8 = 128 I/O Port Pins through the I²C Bus!

Example 17: I²C Bus Interface – Master Mode
Directory: <RP6Examples>\RP6BaseExamples\Example_I2C_MASTER_02
File: RP6Base_I2C_MASTER_02.c

This program demonstrates how to use the Master Mode of the I²C Bus. Of course you will have to connect a suitable slave device to the I²C Bus before running this program.

We added PCF8591 routines to this example program. The PCF8591 chip contains an 8-Bit Analog/Digital Converter (ADC) with 4 channels and a Digital/Analog Converter (DAC) for generating analog voltages. Thus, PCF8574 and PCF8591 devices perfectly complement one other.

The PCF8591 allows you to monitor four different voltages – our example has been designed to evaluate four additional LDRs (arranged as voltage dividers). However the actual circuit is quite negligibly – we could use four Sharp GP2D120 IR distance sensors, some temperature sensors or any similar devices as well.

One main disadvantage in using these chips is that it is a bit time consuming to receive the measurements via the I²C Bus compared to an integrated ADC or I/O Port. This restricts both ICs to simple, non timing-critical applications. Whoever needs fast response and control should consider using another Microcontroller. Obviously a second, freely programmable controller will complicate the system, but will also make it more flexible!

We included detailed data sheets for the devices PCF8574 and PCF8591 on the CD!

Example 18: I²C Bus Interface – Master Mode
Directory: <RP6Examples>\RP6BaseExamples\Example_I2C_MASTER_03
File: RP6Base_I2C_MASTER_03.c

This program demonstrates how to use the Master Mode of the I²C Bus. Of course you will have to connect a suitable slave device to the I²C Bus before running this program.

This simple program demonstrates how to control a Devantech SRF08 or SRF10 Ultrasonic distance sensor via the I²C Bus. Of course, the program may be adapted for using similar types of sensors from other manufacturers.

Example 19: I²C Bus Interface – Master Mode
Directory: <RP6Examples>\RP6BaseExamples\Example_I2C_MASTER_04
File: RP6Base_I2C_MASTER_03.c

This program demonstrates how to use the Master Mode of the I²C Bus. Of course you will have to connect a suitable slave device to the I²C Bus before running this program.

In this example program, the Master is controlling four I²C Slaves: two SRF08/SRF10, one PCF8574 and one PCF8591. The program is reuses code from the three previous examples.

Other programming examples for peripheral I²C-devices will be supplied along with the corresponding expansion modules!

Example 20: I²C Bus Interface – Slave Mode
Directory: <RP6Examples>\RP6BaseExamples\Example_I2C_SLAVE
File: RP6Base_I2C_SLAVE.c

Initially this program will not perform anything visible from the outside. You will have to add an expansion module to the robot, which takes over control and acts as I²C master.

It makes sense to equip the robot with several additional Microcontrollers. In most cases it is a good idea to let one of the additional Microcontrollers control the complete Robot. Not only bigger and faster controllers can do this, but also another MEGA32. The Controller on the Mainboard performs several tasks already like ACS, Motion Control etc. and this consumes quite a bit of processing time (lots of Interrupt events). On an external Controller, you will have more spare time.

This is the idea behind this program: a master controls the high level overall functionality of the robot and sends commands to the slave through the I²C Bus, which handles the low level control. All known functions like the automatic motion control now show their real advantages – the master only has to transmit short commands like "drive forwards at <speed parameter>" and the rest is done automatically. Of course running tasks may be aborted for example after sensors detect obstacles.

The program's source code contains a list with all valid commands and the corresponding parameters. Additionally the program source code contains further details for controls.

If necessary, the slave controller automatically initiates an interrupt signal on the first interrupt signal line (INT1). This can be used by the Master to detect sensor or motion state changes fast and easily.

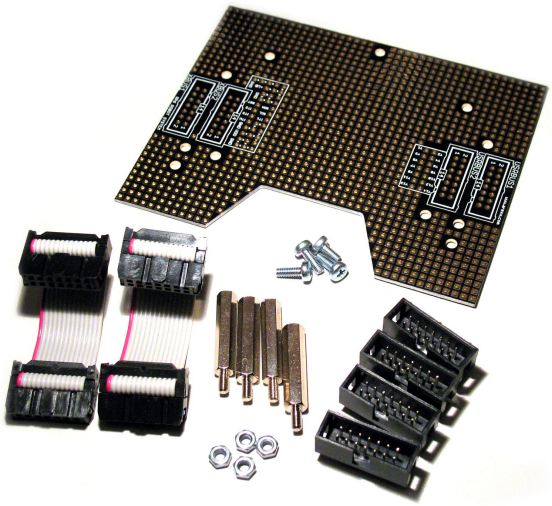
The Sensor values can be read from a number of registers. If an Interrupt event has been triggered, the master can start reading some status registers and check what caused the interrupt event. Afterwards the corresponding register can be read.

Alternatively you might simply read ALL sensor registers at once, which takes some time, of course.

Detailed informations to this topic and of course specific register details are to be found in the program's source code.

Further example programs for robot control will soon be published along with expansion modules. A few of these programs and the RP6 CONTROL M32 examples are available on the CD-ROM already.

5. Experiment Board



There is already an experiment expansion module for assembling your own circuits included in the delivery of the robot. The module is supplied as a kit. You will have to CORRECTLY insert and solder the connectors by yourself. Pay special attention to the correct polarity - check the PCB's white silkscreen printing.

Assembling your own circuits with wired components requires some experience in soldering and basic knowledge of electronics. Of course you must have a basic idea of what you are doing...

Now let's see what kind of circuits could be assembled on the experiment board.

In the previous chapter we already provided two examples! You could add additional I/O Ports or A/D Channels with the described Port-Expanders and A/D-Converters. These can be used to interface with light sensors, IR-sensors, touch-sensors, LEDs, and so on. Additionally the I²C Bus system enables you to interface with more complex sensor modules, e.g. ultrasonic distance sensors, electronic compass modules, gyro or acceleration sensors. Other interesting sensors are pressure-, temperature- and humidity-sensors in order to build a small mobile weather-station.

Basically the system allows you to attach any number of expansion boards to the robot. You do not have to restrict it to a single board. Whenever you are planning to design complex systems, you should consider to install another Microcontroller. RP6 Control M32 would be a suitable device, providing your system with an extra controller, 14 free I/O lines (including 6 A/D converters). The I/O-lines are available on 10-pin connectors. You may easily assemble some flat cables to connect the module to your expansion boards. Soon there will be a “C-Control”-expansion module, which will equally be suitable to serve complex designs.

Of course, the robot's main board also provides 6 extension areas, which may be attractive for sensors that need to be as low over the floor as possible (e.g. additional IR-sensors). However we advise you to start expansion-projects by working with the experiment module before soldering parts to the main board as removals of components will be easier if something goes wrong...

6. Closing words

Here, this small introduction to the world of robotics ends. We hope you enjoyed all activities with the RP6 so far!

All beginnings are difficult, especially this complex topic, but once you mastered the first hurdles, you will have a lot of fun with robotics and you will easily learn a lot.

As soon as you got familar with C-programming, you may start your own interesting projects. We already suggested some ideas for projects in the field of robotics, but of course your own ideas may be more interesing!

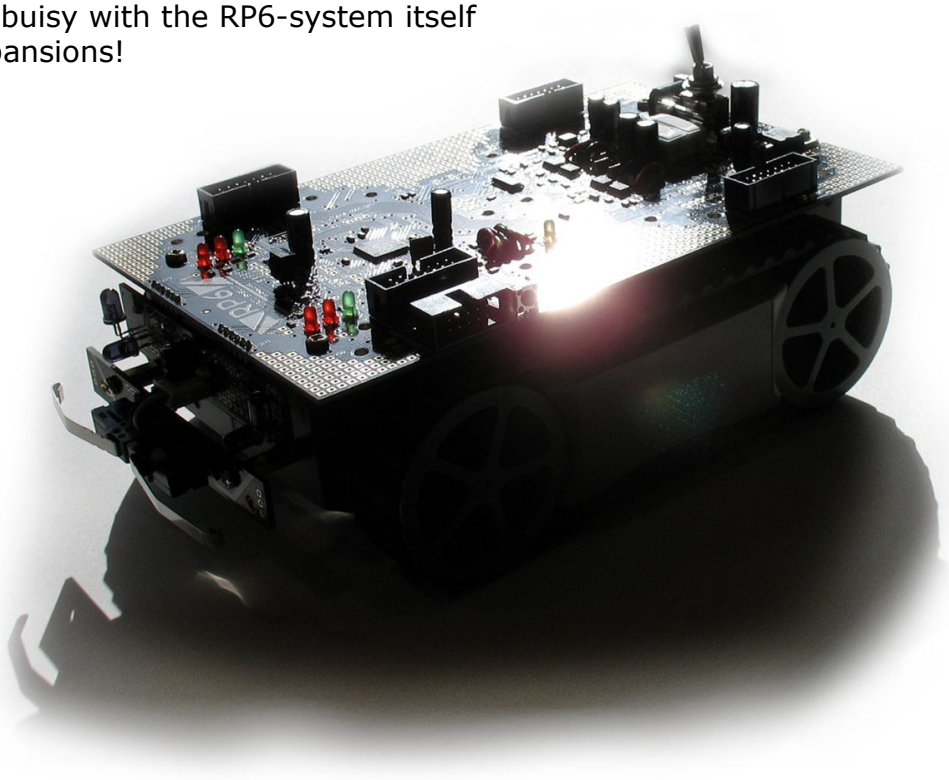
You are invited to discuss about robotics and electronics in several dedicated web forums. Usually this will be much more informative than any manual could be. Even this fairly long manual does not allow us to discuss all relevant topics and a great number of details have been left out.

Of course we plan to develop some more expansion modules for the RP6. We already released the breadboard experiment module and the RP6 CONTROL M32 providing another MEGA32 Controller (including a microphone sensor, piezo sounder, a hughe external 32KB EEPROM memory, keys, LEDs and a LCD Port). We also plan an expansion module for newer Conrad Electronic C-Control Modules (e.g. for the C-Control PRO MEGA128).

We will develop modules with additional sensors, but right now we can not tell you any specific details...

As soon as new modules are released, we will announce this on the AREXX Homepage and in our forum!

Until then, you will surely be buisy with the RP6-system itself and the already available expansions!



APPENDIX

A - Troubleshooting

This appendix lists a number of problems and possible causes, solutions and/or suggestions how to avoid problems. The list may be extended and updated from time to time!

1. **The robot can not be powered on – none of the LEDs lights and the robot does not react on the start/stop button!**
   - The USB-Interface is connected to the robot, but not to the PC! This connection will hold the MEGA32 in reset mode, which does not allow the controller to turn on any of the LEDs. Always start by connecting the USB Interface to the PC before connecting it to the RP6! The same problem may occur if the PC has been switched off.
   - One expansion module holds all Microcontrollers in reset mode – which may easily happen in your own designs (it may happen in pre-assembled designs as well, but usually this is restricted to defects or software bugs). You may test for this condition by removing all expansion modules!
   - You may have removed batteries or you are using empty or defective batteries.
   - The main fuse is open. Please check the fuse with an Ohm meter (or a Multimeter). Quite often a broken fuse can be easily checked by looking at the thin wire inside of the fuse's glass-housing. Fuses normally will blow at the middle of the thin wire, but sometimes the disconnection cannot be observed by visual inspection and must be checked with a Multimeter!

     ATTENTION: Whenever the fuse blows you must check for irregular conditions in the system! Somewhere a short-circuit may have been causing high currents (did you forget to remove some metallic tools or other objects from the robot's body?) or individual parts are drawing a heavy load current. Please inspect the PCBs and all parts! You may have modified the robot's circuits and/or you may have been using wrong components. Did you insert batteries with correct polarity? Did you avoid squeezing the cables between main board and frame? Did you recently attach expansion modules? If you did, you will have to dismount all expansion modules before restarting the tests! If all checks seem to be OK and all expansion modules have been removed, you may insert a suitable, fast 2.5A fuse and try again.
2. **The robot refuses to start programs!**
   - Did you press the Start/Stop button after the program upload?
   - Did you correctly upload a program to the RP6? Did you select the right program?

3. **While moving, the robot repeatedly generates resets and halts the program!**
   - The batteries do not provide the system with enough power!
   - The batteries are malfunctioning (poor quality or too old) and causing voltage drops during operations, which will automatically result in a reset. Please use fresh batteries for your robot in this case and try again!
   - Batteries are not inserted tightly in the battery compartment or the power wiring system has loose contacts somewhere.
   - An expansion module may have caused an overload!
4. **The RP6Loader fails to connect to the robot!**
   - The USB cable or the 10pin flat cable may have loose contacts (please carefully wobble the USB plug a bit!).
   - You selected an incorrect port in the RP6Loader Software.
   - The port is used by another application and therefore it is not shown in the port list. Please close all applications, which may be using USB Ports or USB-serial converters. Subsequently refresh the RP6Loader's port list and restart the RP6Loader!
   - You may have activated "Invert Reset Pin" in the RP6Loader's Preferences dialog – deactivate this option!
   - The robot has been switched off or the batteries are empty/nearly empty.
   - The cable or the plug have been damaged – which is rather uncommon, but may happen with high mechanical stress.
5. **The robot generates strange noises when moving!**
   - Strange noises may have several reasons. You will have to open the robot to check the gears and motors. The power cables may have moved into the gearing system? **Just a few rattling noises are quite normal and will not cause problems!** At high speeds the robot certainly will generate more noise (but definitely not reach the extreme sound levels of the old predecessor CCRP5-system).
   - Maybe one of the adjusting collars (fixing bolts) at the gearwheels, at the front wheels or anything else has loosened? All wheels must rotate freely (test this by carefully turning the wheels on the rear!), but the gearwheels are not allowed to be grinding at each other!
   - A few drops of oil applied to both ball bearings of each motor – at the front-side and at the backside where you may view the motor shaft – may reduce noise generation (**ATTENTION**: Please strictly use special oil for ball bearings only!! e.g. Conrad part number 223441-62 "Team Orion Free Revs Ball Bearing Oil #41701"). During and after this oiling-procedure for the ball bearings, the motor shafts have to be rotated in order to adequately spread the oil over the bearing's surfaces (hold the motor upright for a few seconds to allow the oil film to flow into the bearing)! After this, any oil residues should be wiped away! Bearings should have been oiled during manufacturing in the factory already, but the oiling procedure may need a refresh again at certain intervals!

Please do ***not*** consider to oil the gearwheels this way! This may destroy the encoder system! Even if the oil is not damaging the electronic components, the fluid may damage the encoder wheels/stickers, which can soak any fluids very well and this will drastically influence reflectivity! Additionally rotating gearwheels may spread these fluids and lubricants anywhere inside the robot's interior! **The application of lubrication grease (non-fluids!) is *restricted* to the shafts (!) of both cluster gears only!** Usually grease will not improve noise at all...

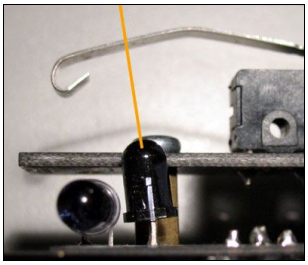
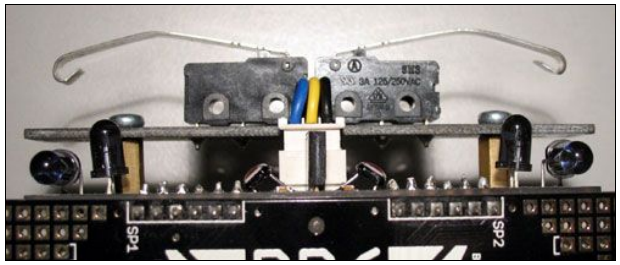
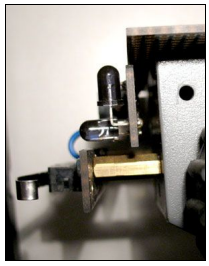
6. **With every program, the motors just accelerate very shortly up to a high speed, and then immediately stop. Subsequently four red LEDs start blinking!**

   - The simplest possibility: You loaded a self-written or modified example program to the robot and forgot to execute "`powerON();`" before starting the motors!
   - Did you modify the software (especially the library) in any way? Try some other programs and the self-test program.
   - If other programs and the self-test behave in a similar way: check the output on the serial interface! Additionally to the four red blinking LEDs, the system outputs an error message which you can look at in the terminal. Most likely, the encoder system may cause this malfunction. Sometimes the adjusting collars, fixing the encoder wheels, may have detached, causing the gearwheels to drift away from the sensors. In this case the encoder system will stop delivering feedback signals! By the way: we provided the encoders with tiny potentiometers (=variable resisters) to adjust the sensors (use a fine 1.8mm slot screwdriver or a suitable cross slot screwdriver! BUT BE VERY CAREFUL: Adjustment usually requires the use of an oscilloscope! Otherwise the calibration procedure may be difficult. The self-test program will show if the encoders work properly. We provided the program with a special mode (c - Duty Cycle Test) to *roughly* check the duty cycle for the square wave signal. 50% duty cycle are optimal. However, deviations and fluctuations between 25 and 75% can be considered as tolerable (incidentally the program may report errors, which is quite normal: the cause for these errors is located in the program and not in the encoder system). Most of the duty cycle values should be in the range of 30 up to 70% and the Duty Cycle Test should report an "OK"-status most of the time. During tests the test-program will deactivate the motor-control and run the engines at a fixed PWM-value. Speed measurement values will be shown in encoder counts per 200ms.
   - Maybe the encoder wheels have been damaged? For instance by oil or lubrication grease as described previously?
   - Check the cables to the encoder system for possible damage, e.g. by using a Multimeter (switch the robot off and check every single wires for connection between both ends and for short circuits to its neighbours!)!
   - Check the tiny IR reflection sensors of the encoders for polluted areas.
   - The LEDs will also start blinking as soon as one motor is (or both motors are) malfunctioning – so check the wiring and the printed circuit board, especially near the motor drivers.

7. **The robot is either not moving or moving slowly and/or starts blinking four red LEDs immediately or shortly after a program's start!**
   - The 4 LEDs will also start blinking on low battery voltage! Just connect the robot to the PC and see if a low voltage warning appears if you click "Connect" in the RP6Loader! If you press the robot's reset button and the Bootloader begins to wait for a start signal, low battery voltage will also cause the four LEDs to start blinking! In this situation, the program may still be started, but the robot will soon stop again and start flashing the LEDs...
   - If you are sure that batteries are fully charged and the status field of the RP6Loader reports a battery voltage well over 6V, check for other error-messages in the terminal. One of the gearing systems or both may be blocked, e.g. by over-tightened adjusting collars, by a gearwheel catching a cable, or gearwheels interfering with other parts. Remove obstacles and carefully loosen over-tightened adjusting collars should fix this.
   - Worst case, the motors or even the electronic control circuits may have been damaged, which requires an exchange of damaged components... Please start the selftest program #8 (Do NOT allow the robot to touch the floor while the test is in progress – both caterpillars must be moving freely! Do not block or brake the caterpillars manually! Blocking will immediately result in a test failure!). The selftest program will check the motors and the electronic control circuits including the motor current sensors. Does the selftest report any error messages? (We advise you to copy the complete terminal's contents to a text-file e.g. by Menu "Options-->Save Terminal").

8. **My battery charger does not charge the batteries inside the robot!**
   - Did you correctly connect the battery charger? Check polarities of cables, and plugs.
   - Did you correctly insert the batteries? Check for defective or loose contacts. Check polarities of cables, plugs and batteries.
   - Maybe the charger's plug fails to match to the inner pin of the charger connector resulting in an open power connection? (Different versions of these plugs exist, which usually may be interchanged without problems. Standard battery chargers are supplied with an appropriate set of adapter plugs)

9. **The ACS fails to reliably recognize obstacles!**
   - Did you already test various different ranges and power settings?
   - Are the IR LEDs or the IR receiver misaligned? Check the alignment and adjustment of IR LEDs and the IR receiver. IR LEDs must be adjusted to almost stick straight out of the sensor-board, but should be aligned to point outwards with an angle of approximately 5-10°. The IR receiver has to be aligned straight 90° upwards (see photographs on this page and chapter 2).

- The ACS will often fail to detect black or very dark obstacles. Black objects more or less tend to absorb IR light completely. This is normal physical behaviour! You may additionally have to use other sensor types (ultrasonic rangers for example...) in order to make object detection reliable!
- The robot is operated in an environment with bright illumination, e.g. with direct sun light, fluorescent lamps, or background illuminations of flat screens. These light sources may disturb IR-communication and ACS.

10. The IRCOMM fails to receive my remote control's signals! The robot does not react to these signals!

- Does your remote controller really transmit RC5 formatted signals? If you do not know for sure, it probably does not. The software will only recognize the RC5 format. You will have to switch the remote control to another coding system (most universal remote controllers allow you to set an RC5 format) or use another remote control.
- Key assignments may vary from type to type and from manufacturer to manufacturer. Always start by assigning the keys for your program. The comments in the RC5 example program source code provide you with guidelines how to assign keys. Check the output codes in the terminal, which will display the key-codes for the RC5 example program!

11. The robot fails to exactly turn a specific angle.

- This is normal behaviour and the manual already explained the reasons for these deviations! Caterpillar drive units will always cause deviations by slipping and sliding. Also the Encoders have to be calibrated in order to work as intended. See appendix B for more details.

12. Why does even an extremely short program occupy 7 KB of memory?

- Each program always includes the RP6Library, which will occupy over 6.5KB of program memory, so 7 KB must be considered as a standard program's size. You will soon notice the program's size will not increase that much with your source-code's length. Don't worry for your memory's size. The memory provides you with ample room for your programs. And if your program's size ever exceeds the memory's limits, you may consider the use of an expansion-module with an additional Microcontroller.

13. My programs cannot be compiled – the compiler ends up with some error-message!

- Check the error-message and try to understand what causes this. If you have to contact the support, first copy the complete compiler report to a text file. Then send the compiler messages and your source file to the support! The following overview lists common programming errors:
  - You forgot to include the RP6Library, or the pathnames in the Makefile are incorrect. If you start a new project do not forget to alter the pathnames in the Makefile! Otherwise the compiler will fail to find these files!
  - Did you forget a semicolon anywhere in the program?
  - Is an accolade missing or superfluous in the program?

- Did you respect the correct C syntax? Apart from comments and usual formatting rules for spaces and tabulators any writing variant of the symbols is relevant for the C compiler. If tHis manuAl contais erorS,, the text's contnt may sill b cleary uderstoooD bi tha readr.! Compilers do not allow any mistakes and may generate a hughe number of errors for the smallest mistakes. Unfortunately the compiler does not have an automatic error correction comparable to that of us humans...

14. My programs are still not working and the robot does not obey to my commands – what is going wrong?

- No idea! ;-) You will have to be more specific in describing your problems to the support. We often receive questions like "Why is this program malfunctioning?". Dot. Of course, we need a bit more detailled description of what the program is supposed to do at all and what is not working! Otherwise this quickly turns out to be a quiz game...
- Usual beginner mistakes are:
  - Adding a semicolon at the wrong location – e.g. closing a loop or an if-statement. Often you are allowed to insert a semicolon-symbol there, but it will not work as you may have intended!
  - Providing if/else-constructs with accolades at wrong positions – this may easily happen if your program's indentation structure is bad.
  - Using incorrect data types for variables – e.g. the uint8_t data type may accept values in the range from 0 up to 255, but cannot be used to count up to 1500! To count this far you will have to use an uint16_t! The uint8_t data type will not accept negative values as well. To work with negative values, you will need a signed data type like int8_t! Check the overview of all data types at the start of the small C crash-course!
  - Forgetting the infinite loop at the program's end – if this infinite loop is missing, your program may produce strange results.
  - You are using the non-blocking mode for "move" or "rotate" functions, but you do not frequently call "task_motionControl" or the "task_RP6System" functions! Or your program contains long pauses generated with mSleep. In using the non-blocking mode for "move" or "rotate" functions or using the ACS you must use stopwatches for all delays over approximately 10 Milliseconds! The mSleep-function and other blocking functions should not be used in combination with non-blocking functions! Please read the RP6Library's chapter again for details and study the example programs!
  - **Always remember to save altered program's sources before re-compiling! Otherwise the compiler will compile the previous unaltered version on your harddisk! In doubt, you may execute MAKE CLEAN and re-compile again!**

15. You are confronted with other problems?

- Try reading the important sections of the manual again! It may often help, but sometimes it does not...
- Did you already download the latest software versions and the recent manual version from http://www.arexx.com/ or from the RP6 homepage http://www.arexx.com/rp6 ?
- Did you use the search function of our forum?
  --> http://www.arexx.com/forum/
- For C language, AVR or Microcontroller related problems, did you visit the AVRFreaks website? --> http://www.avrfreaks.net/
- Did you generally visit both forums at all? (Please do not start by immediately posting a new topic – start by searching for your problem using different search queries!). You can also try using a general web search engine.

B – Encoder calibration

The effective encoder's resolution depends on the real wheel and rubber track diameter. The caterpillar's rubber tracks get pressed into the surface or get impressed themself. Additionally, manufacturing tolerances cause variations in diameter sizes. To compensate for these deviations we will have to perform some measurements and calibrate the encoder resolution.

The resolution is the distance covered with each encoder step. A theoretical value for the robot's resolution is 0.25mm / encoder step, but from practical experience we derived resolution values ranging from 0.23 up to 0.24mm.

To calibrate the encoder's resolution, we let the robot drive a predefined, long and straight distance (e.g. one meter or more), which subsequently has to be measured accurately with measuring tape. In order to show the number of encoder steps, the robot is connected to the PC during this movement. The USB cable and the flat ribbon cable will have to be guided loosely over the robot – do not pull or hold the cable! The bumper's PCB-front-side could be aligned exactly with the beginning of the measuring tape. Adjust the robot's path accurately to a straight line parallel to the the measuring tape.

As an exercise, we suggest to write a program for actuating the robot to drive exactly one meter. Alternatively you may also choose 2 meters or any other distance (most important is that the cable is long enough). Of course you may re-compile the program for various distances and reload the modified program to the robot. Each program version should output lists of the covered distances in encoder steps. (If you are too lazy – there is also a menu option in the selftest program that can do this ;-))

One meter corresponds to exactly 4000 encoder-steps at a 0.25mm-resolution. Now if in a test run the robot moves only 96.5cm = 965mm and the counter reports a total count of 4020 encoder steps, we may calculate a resolution of approximately 0.24mm by simply dividing 965mm by 4020. Please note these values in a table. Repeat the calibration procedure a few times and note the values in a table. Now calculate and enter the average value into the parameter-field `ENCODER_RESOLUTION` in the file RP6Lib/RP6Base/RP6Config.h (a relative path from the main example program directory – don't forget to save the file!), the re-compile the program and load it into the robot. Repeat this test thrice. Each test should improve the results in driving exactly one meter's distance. If no improvement is observed, repeat this and enter the new value in the configuration file. You will however fail to perform a 100% exact calibration – to do so you would need lots of additional sensors.

Rotating the robot on the spot will even deteriorate calibrations. These rotations will cause the caterpillars to slide on the floor, resulting in much shorter real distances with respect to measured values. Results will heavily depend on surface conditions. Slipping conditions on a parquet or a carpet may vary the caterpillar's calibration parameters slightly. Therefore you always have to consider calibration tolerances up to 10° (for rotation). Additionally certain surfaces may cause the robot to slip aside. To include these side effects in calibrations you will need to do some more trials.

You can try to lift up the robot while it is rotating at the front Bumper Panel, such that only the back wheels touch the ground. You will notive how much faster the robot turns now – this gives you an idea of how big the difference is.

Improved methods for distance- and positioning-measurements

Encoder measurements will never get 100% accurant. If you need better navigation, you will have to use additional sensors.

For example, the University of Michigan in the USA designed a tiny trailer carrying the encoder sensors for one of their big caterpillar-robots. The robot pulls the trailer, which has been connected by a pivotable metallic rod. Encoder wheels at the trailer's axles deliver accurate positioning data after performing a few calculations. Details may be studied on the following website:

`http://www-personal.engin.umich.edu/%7Ejohannb/Papers/umbmark.pdf`

starting at page 20 (or page 24, respectively in the PDF-version)
or in: `http://www.eecs.umich.edu/~johannb/paper53.pdf`

We might design a similar construction for the RP6 (though it will not be easy) – alternatively you may consider to start some experiments by fixing an optical computer mouse on the robot's rear or front. Additionally you could use an electronic compass or similar devices, which will be mounted at the very top of the robot (to avoid interferences with the motors and other electronics) and enable accurate rotational movements.

Of course a Gyro may be an alternative positioning sensor as well...

We did not test these ideas, which must be considered as thought provoking impulses for further improvements and studies... and of course some users may not be interested in accurate movements anyway - adding a mouse sensor also restrains the vehicle's offroad capabilities.

Basically perfect positioning requires the use of landmarks, which have to be positioned at accurately defined locations. Of course, these positions must be well known and easily recognizable by the robot, e.g. infrared beacons transmitting a guidance signal for the robot. The Robot can detect direction of the beacons.

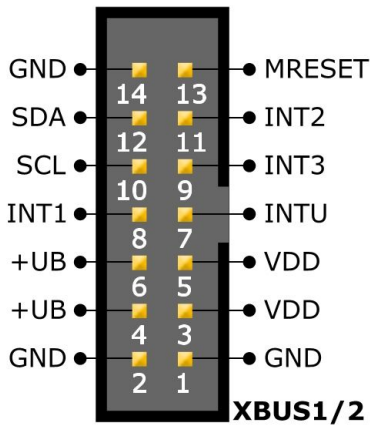
Or you implement line tracing sensors for following markings on the floor.

As a matter of fact we might think of much more complicated positioning methods ... but we will restrict this overview only to a few ideas requiring a minimal amount of CPU power. We might install a camera to the ceiling and remotely control the robot by special pattern-recognition software and transferring command data by wireless or infrared signals. Some users already managed to create a similar remote control system for the small ASURO robot...

You will notice that it is easy to navigate the Robot relatively accurate by using a remote control. Obviously accurate controlling is largely supported by visual feedback of the user, who directly observes directions, obstacles and targets! The robot is unable to see these details without cameras and powerful image processing. A camera at the ceiling and a color marking on top of the robot would provide us with an external sensor for more accurate visual feedback...

C – Connector pinouts

This overviews provides you with the most important connector pinoutson the main-board. Additionally we extended the list by a number of details for usage.



For the sake of completeness we start by defining the connector pin assignments for the expansion connector once again (s. chapter 2):

At the main-board pin 1 is located at the side of the white-coloured labels XBUS1 and XBUS2, respectively, or at the label "1" next to the plug.

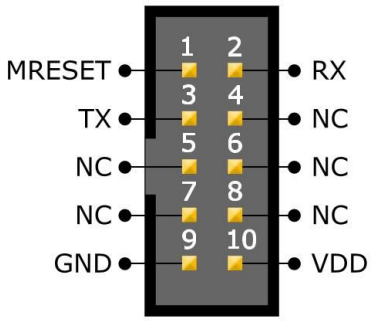
+UB is the battery voltage, VDD the +5V-supply voltage, GND the "negative" terminal (GND = Ground), MRESET the Master Reset Signal, INTx are Interrupt-lines, SCL the Clock- and SDA the Data-lines for the I²C Bus.

Important information: **Please restrict the maximal load for each expansion connector's power supply lines VDD and +UB to maximal 1A-currents each (this is valid for both pins in TOGETHER! That is: Pins 4+6 (+UB) and 3+5 (VDD))!**

Any other required signal lines can be soldered to the USRBUS connector pads, which have been connected 1:1 to the pads on the board, this means the connectors pin1 is connected to Y1, pin2 to Y2, etc...

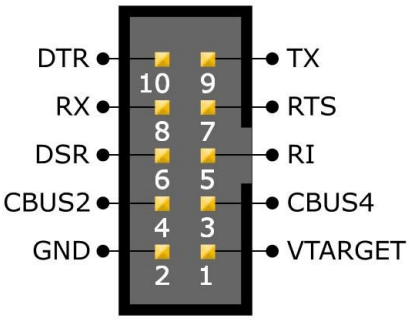
The connector pin assignments for the 10pin connectors for the USB-interface are different from each other:

Main board:



Of course RX and TX must be interchanged to allow communication. Additionally, the connector orientations had to be mirrored to allow a correct orientation for the protecting pull-relief of the flat ribbon-cable's plug.

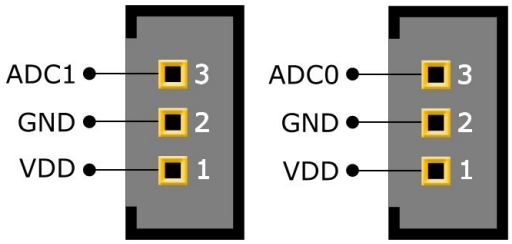
USB Interface:



If the mainboard's connector is not being used for the USB-interface, we may connect the device to other systems – e.g. for remote controlling the robot by UART from other Microcontrollers.

ADC Connectors:

The figure shows the pin assignments for both free ADC channel connectors.



We did not assemble these connectors and you may apply use 3-pin connectors with 2.54mm grid. Be careful in soldering and do not ruin the mainboard! If you are unexperienced with soldering, please refrain from soldering at the mainboard and prefer to start experiments by using an expansion board!

You may freely connect two analog or digital sensors to these free ADC channels. The sensor's output voltages are allowed to range from 0 up to 5V and the connectors provide the sensors with the 5V supply voltage. It might be wise to additionally solder a big electrolytic capacitor to the mainboard – values from 220 up to a 470µF (do not exceed this value! The capacitor's operating voltage should be bigger or equal to 16V) will be perfectly suitable for most applications.

However you will probably not really need a big electrolytic capacitor, unless you are working with sensors, which require vast peak currents – e.g. the popular Sharp IR distance sensors. Decoupling capacitors (100nF) on the mainboard are suitable for short connection wires only – for long wires, these have to be directly soldered to the sensor (we advise to directly mount these capacitors at the sensor's connection pads even for short wires as well!).

All other connections have been clearly labelled on the mainboard. It is a good idea to have a look at the relevant schematics on the CD!

D – Recycling and Safety instructions

Recycling

Disposal of the RP6 in domestic waste is not allowed! For disposal, the robot must be delivered to the local recycling centre or any other recycling centre for electronics!

Please ask you local sales contact for details.

Safety instructions for batteries

Batteries (accumulators and alkaline cells) must be kept out the reach of children! Do not let batteries laying around accessible for everyone, as the components may be swallowed by kids or animals. In case any of the robot's objects has been swallowed, you must immediately consult a doctor!

Contacts to leaking or damaged batteries may cause chemical burns at the skin. In order to handle these objects, you must be using appropriate protective gloves.

Do not short circuit batteries and do not dispose batteries in a fire. You are not allowed to charge alkaline battery cells! Recharging normal battery cells, may cause them to explode! Confine yourself to use only special rechargeable accumulator batteries (e.g. NiMH-accumulators) and suitable charging equipment compatible for these devices!

Recycling instructions for batteries

In analogy to the RP6, the disposal of the batteries (accumulators and alkaline cells) in domestic waste is not allowed! End users must recycle all defective and used batteries. Therefore, please return your defective, emptied and used batteries to your dealer or to your local recycling centre for batteries! You are allowed to return accumulator batteries and battery-cells at any store, which also sells these devices.

This way you are meet legal obligations and simultaneously contribute to environmental protection!