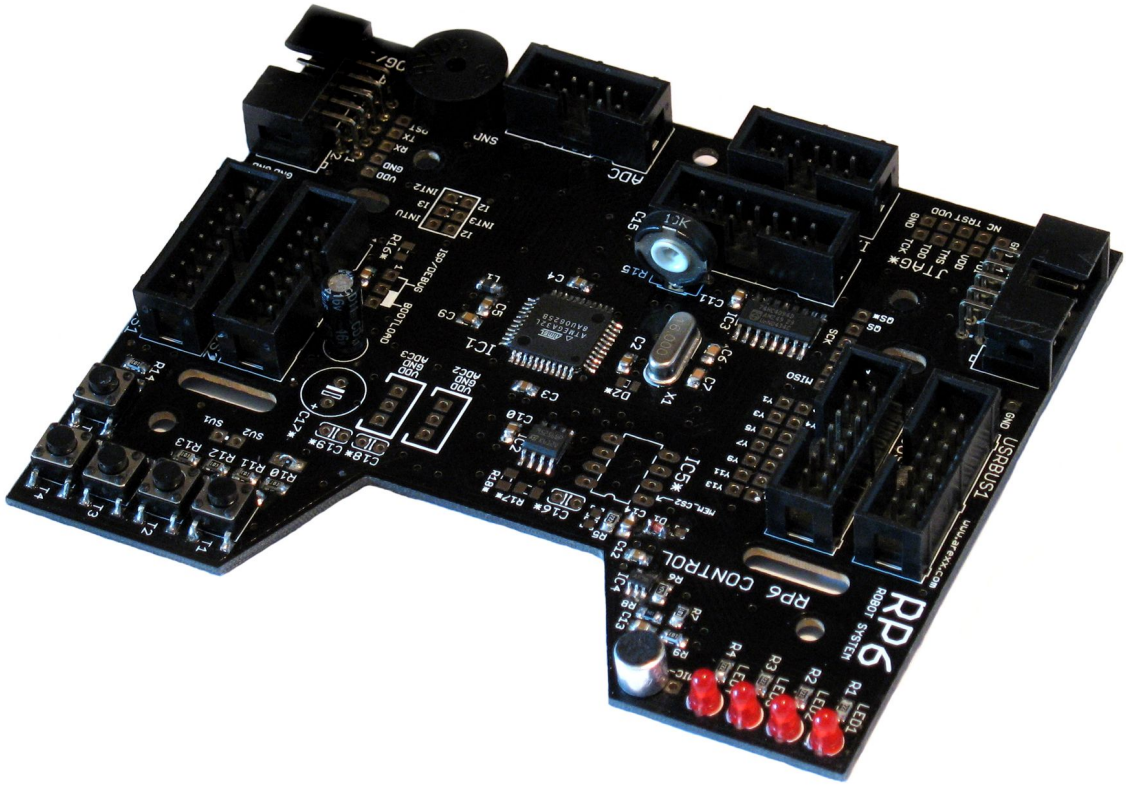
RP6 ROBOT SYSTEM

RP6 CONTROL M32
Expansion module



AREXX Engineering

RP6-M32

www.arexx.com

RP6 CONTROL M32

Manual

- English -

Version RP6-M32-EN-20071031

PRELIMINARY VERSION

IMPORTANT INFORMATION!
Please read carefully!

Before you start operating this RP6 expansion module, you must read this manual AND the RP6 ROBOT SYSTEM manual completely! The documentation contains information about 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. The RP6 CONTROL M32 manual is only supplementary documentation!

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" in the RP6 ROBOT SYSTEM manual!

Nervistraat 16
8013 RS Zwolle
The Netherlands

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

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. The RP6 CONTROL M32 Expansion module

The RP6 CONTROL M32 (or shortly "RP6-M32") expansion module enables you to upgrade your robot with an additional Atmel ATMEGA32 Microcontroller. Compared to the controller on the mainboard, this one features twice as high clock frequency. And as a bonus, on the RP6-M32 you have a lot more free processing time, because the Motor Control, ACS, IRCOMM, etc. can be handled by the controller on the mainboard.

The external 32KB SPI EEPROM provides the module with a reliable (1 million cycles) read- and writeable memory, which may be used for data-logging or as program storage for Bytecode Interpreters like the NanoVM for Java. Optionally the board allows you to solder an extra DIP 8 socket to the PCB for a second EEPROM.

The buttons, LEDs, piezoelectric beeper and the optionally available LC-Display give you lots of additional possibilities. They enabe you to control the robot, e.g. by programming a small command menu with a few push buttons for selecting special functions – and of course it can be used as a display for measurement values and status messages. The beeper may generate a range of sounds and for instance play a greeting melody at program's start or an alarm sound at low battery level.

For controlling your own circuits on the Experiment modules, the 14 free I/O-ports are available on two standard 10 pin connectors. Six out of these 14 I/O-lines may be used as Analog/Digital Converter channels.

Last but not least, the module has a microphone sensor, which was available on the predecessor CCRP5 as well. The microphone will allow you to start the RP6 by clapping your hands or similar things. We designed the control circuit as a "Peak Detector" for responding to maximum acoustic levels in the robot's environment. Of course, the detector will only perform well if the motors are stopped or run at slow speeds only, because the microphone is much more sensitive to the noise generated by the robot.

Before starting any operations with the RP6-M32 you should make yourself familiar with the RP6 Robot itself by testing all example programs WITHOUT the RP6-M32 expansion module mounted on the robot!

This manual has to be considered as a supplementary manual only. Please read the complete RP6 manual before starting with the RP6-M32!

Important note for beginners: Programs written for the RP6-M32 clearly CAN NOT be run on the Robot base unit and vice versa, as both systems have different pin assignments and clock frequencies!

ATTENTION: Loading a program to an inappropriate controller may probably damage the controller or interface devices! If an I/O-pin is usually connected to a circuit in input mode and the wrong program sets this pin to output mode, the I/O-pin may overload and/or cause damage to the circuit!

Usually nothing terrible will happen if you make this mistake, but we can not guarantee this! The RP6Loader is unable to discern between hexfiles for different controllers, as these files all look the same. To avoid mistakes, you may use the RP6Loader's functionality to create several categories and define dedicated categories for every expansion module...

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 CONTROL M32 box:

- **RP6 CONTROL M32 module**
- 4 pcs 25mm M3 distance bolts
- 4 pcs M3 screws
- 4 pcs M3 nuts
- 4 pcs 14pin connectors
- 2 pcs 14pin flat cable

Please find the software and the PDF manual on the main RP6 CD-ROM. Updated versions of the the software and this manual will be published on our website!

1.3. Features and technical Data

This section provides an overview of the RP6-M32'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 CONTROL M32:

- **Powerful Atmel ATMEGA32 8-Bit Microcontroller**
  - ◊ Speed 16 MIPS (=16 Million Instructions per Second) at 16MHz clock, which is twice as fast compared to the controller on the RP6 mainboard!
  - ◊ Memory: 32KB Flash ROM, 2KB SRAM, 1KB EEPROM
  - ◊ freely programmable in C (by using WinAVR / avr-gcc)!
  - ◊ ... and many more features (please have a look at the datasheet)!
- **External 32KB SPI EEPROM**
  - ◊ Very fast SPI Interface (8MHz clock frequency)
  - ◊ Each memory cell is specified for at least 1.000.000 write/erase cycles.
  - ◊ ... see datasheet on the CD-ROM (AT25256A) for additional information.
  - ◊ Well suited for data-logging or program storage for Bytecode Interpreters (for instance a Java VM like the small NanoVM: http://www.harbaum.org/till/nanovm/ . This VM however will have to be adapted for using an external EEPROM ...)
- **I²C-Bus Expansion connectors**
  - ◊ can control any I²C Bus Slaves
  - ◊ The module's MEGA32 may be used as master or slave device. Usually we suggest to use the expansion board controller as master for complete control of the Robot and to use the mainboard's controller as slave for motor speed control, ACS, IR-COMM, battery monitoring, etc. in order to disburden the expansion board controller.
- **Microphone sensor**
  - ◊ to detect noise, e.g. clapping hands, etc.
- **Piezo beeper**
  - ◊ useful for generating simple melodies
  - ◊ Signal generator, e.g. to indicate errors or state changes
- **4 Status LEDs**
- **5 Buttons**
- **LC-Display Port**
  - ◊ suitable for connecting a 16x2 character LC-Display, but you may use other LC-Display formats as well. However, these will have to be attached by two spacer bolts and may not fit on one side ... Please check dimensions before you order a display and additionally obtain suitable installation material! In order to use formats differing from 16x2 you will also have to do some small changes in the

library functions (mainly for display initialisation and maybe for cursor positioning). The settings in all example programs match 16x2 character displays only, but it is easy to modify these settings for other displays!

◊ The display may be used to show text messages, menus, program status messages or sensor values.

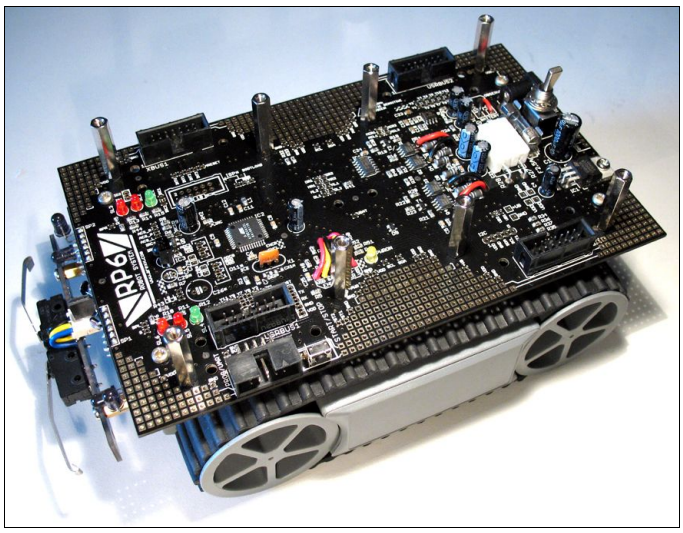
- **14 freely available I/O Ports for controlling your own circuits and sensors.**
  ◊ **6** of them may be used for **ADC-channels** (Analog/Digital Converter)
- **Up to 3 external interrupts are available on the XBUS-connector.**
- **USB PC Interface connector** available for program upload.
  ◊ Program upload works just as quick and easy as with the robot base unit through the USB Interface and the comfortable RP6Loader program.

We provide some C example programs and an extensive function library, which will definitely be a great help for beginners.

We are planning to publish more programs and updates for the RP6 and its expansion module on the robot's website. Of course, we also encourage you to exchange programs with other RP6 users! The RP6ControlLibrary and all sample programs are published under the Open Source Licence GPL.

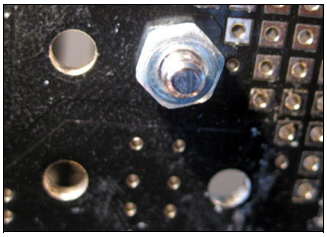
2. Installing the expansion module

The specific way you can mount the module on the robot depends on other extension modules, which you already may have installed on the robot.



In order to attach the module to the robot you will first have to loosen the four screws of the mainboard just like you did when inserting the batteries. You may also carefully detach the bumper PCB's tiny plug from the Sensor PCB to lift the main board completely. However, a complete lift of the mainboard is not necessary as long as you can use your fingers for fixing the distance bolts with the M3 nuts under the main board area.

Attention: To prevent mechanical damage to the Sensor PCB, support it by pressing a finger against the sensor PCB's backside during re-connection of the cable! Alternatively, you may also remove both screws of the bumper's PCB and leave the cable connected to the system...

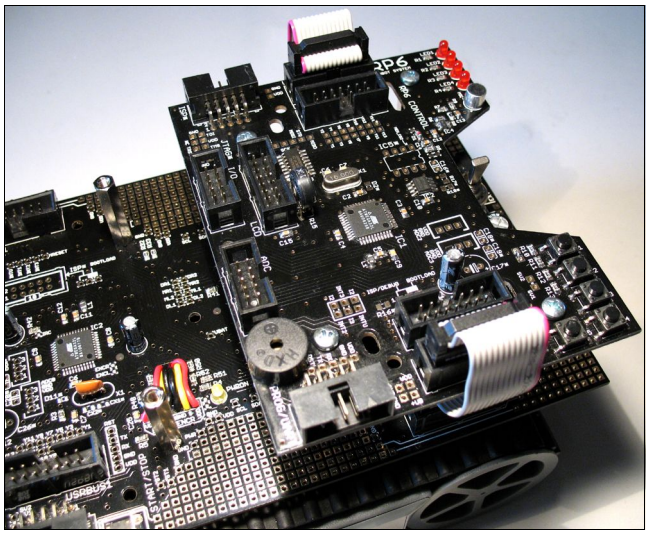


Subsequently you have to attach four 25mm M3 distance bolts with M3 nuts to the main board's mounting holes as shown in the photographs.

The upper photograph shows how all 8 distance bolts have been mounted, including the bolts for the breadboard expansion module!

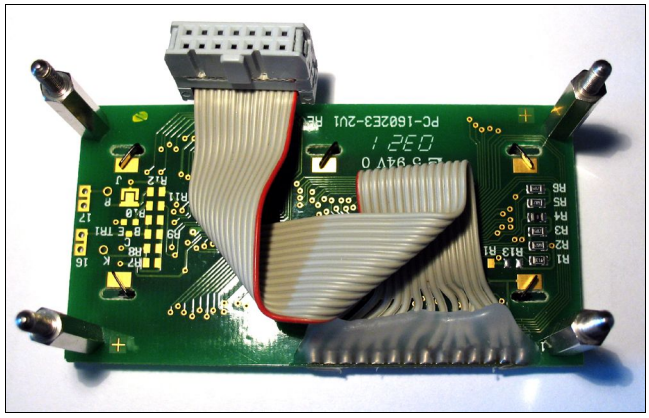
Having completed these preparations, you can insert the expansion module on top of the distance bolts and fix it with four M3 screws.

Finally connect both flat cables – that's it.



We recommend to install the RP6 CONTROL M32 at the rear robot's expansion-stack – as the top module. This location enables you operating the keys and easily reading the display. Additionally both USB Interface connectors are accesible on the same side now. The experiment module may now be installed at the robot's front stack (see an example configurationin the photograph on the next page).

Users who purchased a 16x2 character LC-Display must connect and install the display on the expansion module BEFORE installing the module on the robot.



The display's 14 pin flat-cable is very flexible and may easily be folded. In order to hide the cable underneath the display module, please fold the cable for the RP6-M32 expansion module as shown on the photo.

Then attach the display to the expansion module with e.g. 20mm or 25mm distance bolts and suitable nuts.

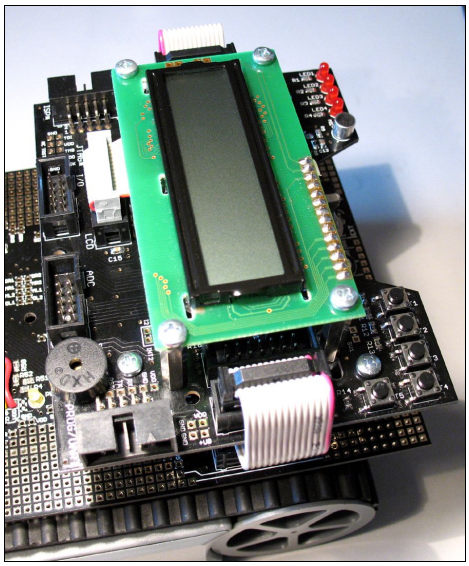
You may use any other Text LC-Display with a HD44780 compatible controller. Of course, you will have to connect your own cable to the display unit (by soldering). *Attention **must** be paid to the correct pin assignment!*

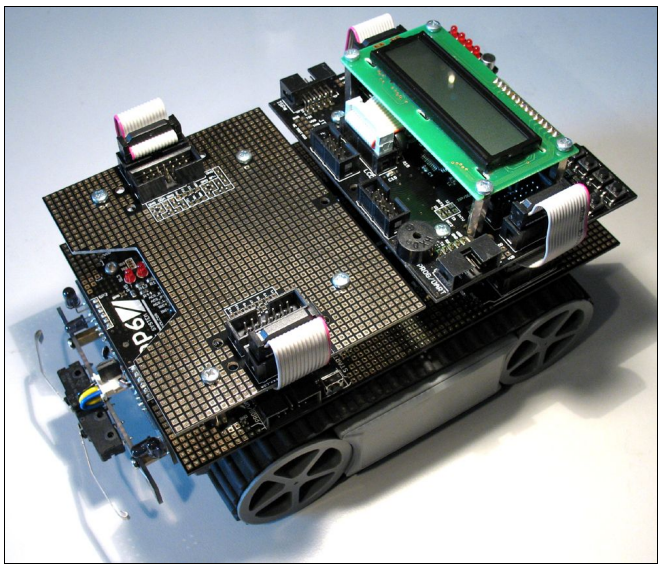
You do not necessarily need to fix the Display with 4 distance bolts as shown in the photograph! Two bolts (either both at the front side or alternatively at the rear side) will provide sufficient fixation of the module.

Tip: the RP6-M32 module does not contain additional distance bolts for the display, but each expansion module kit includes four distance bolts, including the nuts and screws. Of course these four parts are used for mounting the expansion module as shown in the photographs, but alternatively three bolts would be adequate as well – two at the front side and one at the rear side in the middle!

By fixing the experiment module and the RP6-M32 like this, you would get two unused spare distance bolts, nuts and screws for the display...

Completely assembled, the robot with display may look like this:





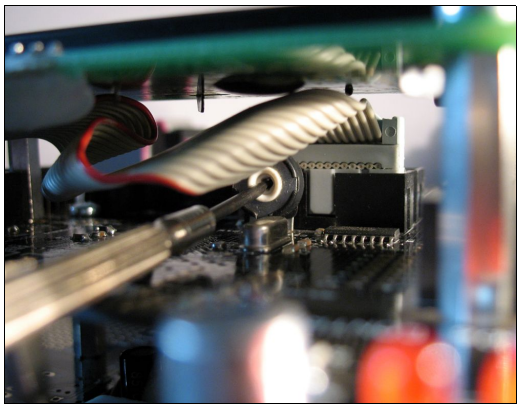
You may now easily connect both 10 pin connectors of the RP6-M32 module with the experiment expansion module by using tiny 10pin flat-cables. On these connectors, you have 14 free I/Os and 6 ADCs for sensor signal evaluation and measurements.

These flat cables can even be used for modules on different stack levels – the flat-cables easily fit in the gap between two stacked modules.

At this stage you will be able to start a simple **function check**:

Connect the USB Interface to the PROG/UART connector on the RP6-M32 with the flat-cable and start the RP6Loader. Now turn on the robot – a text message on the LC-Display should appear and one of the LEDs should start blinking. After a successful startup, please click on "Connect" in the RP6Loader – which should be confirmed by a text message "Connected to RP6 Control" in the status box. If this worked, finish reading this manual and go ahead with the example programs!

If the LC-display does not show any text (or only two rows of dark boxes) while the LEDs are blinking and the RP6Loader connection is initiated successfully you will have to adjust the contrast of the display (alternatively you may have used another display type – but connected using a wrong pin assignment...). Adjust the contrast with the trimmer potentiometer R16 on the PCB with a small flat-bladed screwdriver. A small screwdriver (for cross slots) is applicable as well, but these screwdrivers may often fail to match to the potentiometer...



In order to easily adjust the potentiometer you may choose to unscrew the display (or complete display adjustment before fixing it with the bolts). Please do not disconnect the display! This allows you to adjust the contrast while it's on.

Alternatively you might adjust the tightened display with a very small screwdriver as shown in the photograph. **Please avoid touching any components or PCB-areas with your screwdriver!** We suggest to turn off the robot, insert the screwdriver in the correct position, switch on the robot and adjust the contrast...

3. RP6 CONTROL Library

In analogy to the robot base unit we provided the RP6 CONTROL M32 with a extensive function library including a great number of helpful functions for beginners. We named the library RP6ControlLibrary or in short RP6ControlLib. A number of functions (stopwatch, delay, UART and I²C-Bus) is identical to the RP6Library. The UART- and I²C-Bus-functions even share identical files, which are located in subdirectory RP6common in the RP6Library. We will not describe these identical functions again – please lookup the corresponding chapter in the RP6 manual and it's example programs! This manual will concentrate on functions especially dedicated to the RP6Control or slightly deviating from the RP6Lib-equivalents.

In spite of a the great number of already available functions, the RP6ControlLib must be considered as a starting point only! A lot of functions may be added or improved. Here your own programming capabilities are demanded!

3.1.1. Initializing the Microcontroller

```
void initRP6Control(void)
```

As already stated in the RP6Lib, your programs must ALWAYS start by calling this function! For RP6ControlLib only the name changed...

The **initRP6Control**-function initializes the hardware modules of the RP6-M32's Microcontroller. The Microcontroller will only perform correctly if you start the program by calling this function! The bootloader may be able to initialise some, but not all components at startup.

Example:

```
1   #include "RP6ControlLib.h"
2
3   int main(void)
4   {
5    initRP6Control(); // Initialization - ALWAYS call this function FIRST!
6
7   // [...] Program code...
8
9         while(true);      // Infinite loop
10        return 0;
11  }
```

Each and every RP6 CONTROL M32 program will at least have to look like this! The endless loop in line 9 is required to guarantee a defined end of the program! In analogy to the mainboard's controller a program without this infinite loop may not perform as expected!

3.1.2. Status LEDs

LED control works similar to the robot mainboard's. However the number of LEDs is restricted to four and we will have to use different names as these LEDs are controlled by an external shift register, which is also used for the LC-Display. The 8-Bit shift register is called “External Port”.

The RP6-M32 also uses a function “setLEDs”:

```
void setLEDs(uint8_t leds)
```

Example:

```
setLEDs(0b0000); // This command will turn off all LEDs.
setLEDs(0b0001); // turn on LED1 only
setLEDs(0b0010); // LED2
setLEDs(0b0100); // LED3
setLEDs(0b1010); // both LED4 and LED2
```

We also provided the alternative control method:

```
externalPort.LED1 = true;  // activate LED1 in “External Port” Register
externalPort.LED2 = false;  // deactivate LED2 in “External Port” Register
outputExt();               // Commit the settings!

// outputExt() will send the contents of the externalPort variable
// to the shift register - in analogy to updateLEDs() in the RP6Lib.
// However this function will also update the LCD data lines.
```

3.1.3. Keys

In contrast to the RP6 we connected the 5 keys for the RP6-M32 to an ADC-channel, which enables us to use a single pin for all keys. On the other hand the simplified circuit using 5 identical resistors (see circuit diagram on the RP6-CD) does not allow you to simultaneously press more than one key. But usually this will be satisfactory for typical user inputs!

```
uint8_t getPressedKeyNumber(void)
```

This function returns which key is pressed down by evaluating the ADC and comparing the value with some predefined threshold values. Measurement values however may vary with the resistor's manufacturing tolerances and in order to make your own RP6-M32 work properly you may have adjust these thresholds in the library file! You will find the predefined threshold values in the function getPressedKeyNumber.

```
uint8_t checkPressedKeyEvent(void)
```

This function checks whether a key has been pressed and returns the key's code number only once – in contrast to function getPressedKeyNumber, which constantly returns the key code number as long as the key is pressed down. The function **`checkPressedKeyEvent`** is quite useful for evaluating a key's value in the main loop without interrupting the program flow.

A similar method is used in the function:

```
uint8_t checkReleasedKeyEvent(void)
```

In this case the key's value will be returned only once as soon as a key has been pressed and then released. Function **`checkReleasedKeyEvent`** does not block the normal program flow as well – you will not have to wait in a loop for the key release event.

3.1.4. Beeper

The RP6-M32's beeper may be controlled with the function:

```
void beep(uint8_t pitch, uint16_t time)
```

This is a non-blocking function – the function sets the pitch and the time period for the generated sound and then returns directly. After the predefined period of time the beeper will be deactivated automatically. However you must take into account the following side effect: each following call of this function will overwrite the previous settings. In order to play a melody or to generate single tones you should use the macro:

```
sound(pitch, time, delay)
```

which will accept parameters for pitch, duration and pause between consecutive tones. The macro uses mSleep for generation delays – and the program will be halted for *time+delay* milliseconds and then proceed the program's flow.

If you do not intend to define a pause or a duration at all, you may define the pitch by calling function:

```
void setBeeperPitch(uint8_t pitch)
```

which is very useful for constantly generating varying sound waves (e.g. a siren and similar sound effects). Attention: all beeper functions will only accept "pitch"-values ranging between 0 and 255, in which 255 represents the maximal frequency.

3.1.5. Microphone sensor

Apart from generating sounds, the RP6 CONTROL may also react on sounds. However the system can not evaluate the pitch, but responds to the amplitude only. This way you may for example start the robot on any loud noise level.

We designed the circuit as a "peak-detector", which samples the microphone's signal during a variable period of time and stores the peak-value. The detector allows the Microcontroller to use the ADC for evaluation of the peak value and subsequently re-set the stored peak value again. Basically the detector stores the peak voltage in a small capacitor and "resets" the maximum value to zero by discharging the capacitor.

Initially a special function:

```
void dischargePeakDetector(void)
```

must be called to discharge the capacitor. After initialisation another function:

```
uint16_t getMicrophonePeak(void)
```

allows you to monitor the maximal values in predefined intervals. This function will evaluate the value and automatically call dischargePeakDetector(). One of the example program will demonstrate how to use these functions.

3.1.6. LC-Display

The LC-Display is very useful for showing sensor values and status messages if the robot is disconnected from the PC. Writing messages on the LC-Display is comparable to writing to the serial interface – but of course we will have to consider a few things. Please have a look at the sample programs, which will quickly reveal how to use the LCD-module.

You will always have to initialize the LCD at the very beginning of a program by calling this function:

```
void initLCD(void)
```

Usually you will not need the following function:

```
void setLCDD(uint8_t lcdd)
```

and the other function (write4BitLCDData) – we will only describe these functions to demonstrate the display control.

The LCD is operated in 4-Bit Mode, which implies the use of four data lines and two control lines (Enable (EN) and Register Select (RS). Read/Write (R/W) has been permanently connected to ground to force the LCD in write mode (we will not be able and also do not need to read from the LCD). Just like the LED control lines the four LCD's data lines are connected to a shift register, which allows us to save some ports. In analogy to the setLEDs-function, setLCDD will set the LCD's data lines. Additionally setLCDD has to set the enable flag to trigger the data transfer to the LCD.

In order to transfer standard 8 Bit commands and data to the LCD the data bytes have to be split by calling:

```
void write4BitLCDData(uint8_t data)
```

This function takes care of splitting 8 bit data into two 4 bit "Nibbles" and transferring these halved bytes.

The following function:

```
void writeLCDCommand(uint8_t cmd)
```

will call write4BitLCDData, but will set the RS signal to low in order to send a command to the LCD.

```
void clearLCD(void)
```

will cause the LCD to clear its contents.

```
void clearPosLCD(uint8_t line, uint8_t pos, uint8_t length)
```

will delete characters in a defined range on the display. Allowed parameters are: line, starting position in the line and the number of characters to be overwritten.

Example:

```
clearPosLCD(0,10,5);   // deletes the trailing 5 chars in the first LCD line
```

```
void setCursorPosLCD(uint8_t line, uint8_t pos)
```

sets the text cursor to a defined char position on the display. Parameter "line" has to be set to 0 for the upper line and to 1 for the lower line. The "pos" Parameter may range between 0 and 15 for 2x16 LCDs.

```
void writeCharLCD(uint8_t ch)
```

will send a single character to the LCD – in analogy to writeChar for the serial interface. However you will initially have to make sure the display cursor has been positioned at the correct location. Otherwise the module will not display the message!

Example:

```
setCursorPosLCD(1,5);  // Set the cursor to the second line, character #5.
writeCharLCD('R');     // now display "RP6" starting
writeCharLCD('P');     // at the cursor's location!
writeCharLCD('6');
```

```
void writeStringLCD(char *string)
```

In analogy to the corresponding function for the serial interface the writeStringLCD-function will transfer a null-terminated character string from the SRAM to the LCD. Of course this function call expects a valid text message in a RAM location and not a predefined "constant". If you just want to display a predefined "constant"-string, we advise you to use:

```
writeStringLCD_P(STRING)
```

as this function will read the text directly from flash-memory, without wasting RAM.

The following function:

```
void writeStringLengthLCD(char *string, uint8_t length, uint8_t offset)
```

displays a string with a defined number of characters on the LCD. The function's parameters are identical to the corresponding function for the serial interface.

```
showScreenLCD(LINE1,LINE2)
```

This function has been designed to easily output a message to both LCD lines in a single call. The function will automatically set the cursor at the correct position and previously delete the complete LCD contents.

Example:

```
showScreenLCD("LCD line 1", "LCD line 2");
```

In analogy to the output functions for converting and writing numbers to the serial interface the function

```
void writeIntegerLCD(int16_t number, uint8_t base)
```

will write numbers in BIN, OCT, DEC or HEX format to the LCD.

```
void writeIntegerLengthLCD(int16_t number, uint8_t base, uint8_t length)
```

Apart from the name, writeIntegerLengthLCD works similar as the equivalent function for the serial interface.

3.1.7. SPI Bus and SPI EEPROM

The SPI (=Serial Peripheral Interface) Bus connects the EEPROM and the 8-Bit shift register to the controller. A socket for another AT25256 compatible EEPROM (e.g. ST M95256) in an 8-pin DIP package may be soldered to the PCB optionally.

You may also use other SPI ICs and cascade another shift register after the one on the PCB – but we advise you to prefer the I²C-Bus and to restrict these alternative methods to special purposes, in which the I²C-Bus cannot be used!

We provided the system with special functions for accessing the EEPROM and the shift register, so you may avoid directly using the SPI Bus from your program. But in case you have to access the SPI Bus, the following functions may be useful:

```
void writeSPI(uint8_t data)
```

transfers one data byte to the SPI Bus.

```
writeWordSPI(uint16_t data)
```

transfers two data bytes to the SPI Bus by using a 16 Bit variable, in which the high byte will be transmitted prior to the low byte.

```
void writeBufferSPI(uint8_t *buffer, uint8_t length)
```

will transfer up to 255 bytes from a predefined array to the SPI Bus. The number of bytes to be transmitted from the "buffer"-array has to be specified in the parameter "length".

```
uint8_t readSPI(void)
```

will read a data byte from the SPI Bus.

```
uint16_t readWordSPI(void)
```

reads two data bytes from the SPI Bus and returns the contents in a 16 Bit Variable. The high byte will be read prior to the low byte.

```
void readBufferSPI(uint8_t *buffer, uint8_t length)
```

reads up to 255 bytes from the SPI Bus into a predefined array.

As already stated, we will usually not need these SPI-functions, but the following functions use them for accessing the EEPROM, which is connected to the SPI Bus.

```
uint8_t SPI_EEPROM_readByte(uint16_t memAddr)
```

reads a single byte from address "memAddr" in the EEPROM. The address value may range from from 0 up to 32767 for our 32 KB sized EEPROM.

Example:

```
// The next program line will read a byte from EEPROM address 13860:
uint8_t data = SPI_EEPROM_readByte(13860);
```

```
void SPI_EEPROM_readBytes(uint16_t startAddr, uint8_t *buffer, uint8_t length)
```

will read up to 255 Bytes (length) beginning at the address "startAdr" into a predefined array (buffer).

```
void SPI_EEPROM_writeByte(uint16_t memAddr, uint8_t data)
```

stores one data byte to the EEPROM address "memAddr".

```
void SPI_EEPROM_writeBytes(uint16_t startAddr, uint8_t *buffer, uint8_t length)
```

stores up to 64 bytes (length) in the array "buffer" beginning at the EEPROM address "startAddr".

Please respect the limit of 64 Bytes as maximum transfer size in a single write operation. 64 Bytes is the so-called pagesize of the EEPROM and this parameter represents the maximum memory size available as data cache. Basically data that is transferred sequentially will have to be located inside a page, e.g. between addresses 0 and 63, 64 and 127, 128 and 191 ...! If you exceed this page size limits the EEPROM will be overwriting the data at the page beginning. Of course it allows you to start writing e.g. at address 50, but if you transmit more than 14 bytes, the address counter will restart at zero and overwrite any data stored in the cache.

Reading data from the EEPROM however will not be restricted by pagesizes and in fact you might even read the complete EEPROM contents at once.

The EEPROM requires 5ms to write data, in which you cannot access the EEPROM-device. In order to evaluate the present device's status you can call the function:

```
uint8_t SPI_EEPROM_getStatus(void);
```

The following code snippet demonstrates how to evaluate the EEPROM's status:

```
if(!(SPI_EEPROM_getStatus() & SPI_EEPROM_STAT_WIP)) {
      // ...
}
```

and check whether the EEPROM may be accessed. This functionality is included in the previously described functions, and usually you will not have to use this function! Anyway, it can be useful to accelerate writing if you have to do other things during this 5ms write delay.

3.1.8. ADCs

The ADCs may be read with the following function, similar to one in the RP6Lib:

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

We did not yet provide the RP6-M32 lib with an automated alternative functionality for sequentially reading all ADC-channels in background-mode

Of course the channel-names are different from the RP6Lib:

```
ADC_7       --> ADC Channel 7 – available on the 10-pin connector "ADC"!
ADC_6       --> ADC Channel 6 ...
ADC_5
ADC_4
ADC_3
ADC_2       --> ADC Channel 2
ADC_KEYPAD  --> This channel is connected to the keypad
ADC_MIC     --> This channel serves the microphone.
```

3.1.9. I/O Ports

The RP6 CONTROL provides the system with 14 freely available I/O Ports and we will shortly describe how to generally access the AVR's I/O Ports.

The ATMEGA32 contains 4 I/O-ports. Each of these ports is 8 bits wide and is controlled by 3 registers: One register controls the "direction" of the I/O Pins (DDRx) and specifies whether a pin is used in input- or output-mode. A second register is used to write data (PORTx) and a third register is used for reading (PINx).

Whenever you intend to use an I/O Pin for data output, e.g for LED-control, you must start by setting the corresponding bit in the DDRx Register to 1.

```
Example:
DDRC |= IO_PC7; // PC7 is set to output-mode
DDRC = IO_PC7 | IO_PC6 | IO_PC5; // PC5, PC6, PC7 are set to output-mode,
                                 // all other pins are set to input-mode!
```

You may now proceed to set the output to high respectively low level in the corresponding PORTx-register.

```
Example:
PORTC |= IO_PC7;  // High
PORTC &= ~IO_PC7; // Low
```

If a bit in the DDRx-register has been set to zero, the corresponding pin is operated in input-mode.

```
Example:
DDRC &= ~IO_PC6;  // PC6 is set to input-mode
```

You may now evaluate the PINx-register and read the pin's status (check for low or high level).

```
if(PINC & IO_PC6)
      writeString_P("PC6 is HIGH!\n");
else
      writeString_P("PC6 is LOW!\n");
```

By setting the appropriate bits in the PORTx-Register you may activate the pullup-resistors, which are integrated in the Microcontroller. This option is useful for keys, bumpers and other types of sensors.

The following I/O Pins are freely available on the RP6 CONTROL M32 (exact definitions can be found in the header file RP6Control.h):

```
IO_PC7
IO_PC6
IO_PC5
IO_PC4
IO_PC3
IO_PC2

IO_PD6
IO_PD5
```

You may use the ADC-channels as I/O Pins as well, but you will have to pay attention to the different names used compared to ADC channels (ADC_7 versus ADC7)!

```
ADC7
ADC6
ADC5
ADC4
ADC3
ADC2
```

Important note: Individual I/O Pins have been designed for a maximum current of 20mA. In total an 8 Bit Port may deliver a maximum of 100mA! So, if you are planning to control heavier loads you must use external transistors!

Relevant information on this topic can also be found in the MEGA32's datasheet, which you can find on the CD-ROM!

4. Example programs

On the RP6-CD we provided a number of sample programs, which demonstrate the RP6 CONTROL M32's basic functionality. In analogy to the robot base unit, these samples have to be considered as a starting point for your own programs and by no way as optimal solutions. We deliberately chose this concept to leave some interesting work for you – and it would be boring to just test a few prefabricated programs, wouldn't it?

Of course you may share your own programs with other RP6 users through an Internet forum for example. The RP6ControlLib and all other sample programs have been released under the Open Source Licence GPL (General Public License), which allows you to modify programs and to make them available to others under GPL conditions.

In the Internet a great number of example programs for the MEGA32 are already available, as the AVR Controller family is very popular amongst hobbyists. However you will have to adapt these example to the RP6ControlLib and take the hardware specific details of the RP6 into account – otherwise the programs may be malfunctioning (the most obvious problems are different pinouts, multiple use of hardware modules (e.g. timers), different clock frequencies, etc.)!

Example 1: "Hello World"-Program with LCD text output and LED running light
Directory: <RP6Examples>\RP6ControlExamples\Example_01_LCD
Source file: RP6Control_LCD.c

The program will output messages through the serial interface and on the LC-Display, you should connect the robot to your PC and observe the messages appearing in the terminal of the RP6Loader Software! Optionally you may also connect the LC-Display.

The Robot will not move in this example program – as long as you only load the *I²C-Bus Slave Program into the controller on the mainboard!* Therefore you are allowed to place the robot on top of a table near your PC.

The sample program outputs a short "Hello World"-message to the serial interface and subsequently executes a running light. Additionally the program begins with writing some static text messages to the LCD and proceeds by displaying a moving "Hello World"-text – in which the words "HELLO" and "WORLD" are slowly shifted back and forth. After a delay of around 16 seconds the program briefly pauses and generates two short beeps. Having waited for 8 seconds, the program proceeds – equally generating two short beeps.

Example 2: Buttons and Sound
Directory: <RP6Examples>\RP6ControlExamples\Example_02_Buttons
Source file: RP6Control_Buttons.c

The program will output messages to the serial interface and the LC-Display!

The Robot will *not* move in this example program!

This sample program demonstrates how to use the five keys of the RP6-M32. Each keystroke will result in a text-message, showing the key-number on the display and generating a melody with the beeper.

(Attention: Pressing T4 may soon be getting on your nerves ;-)).

Example 3: Microphone sensor
Directory: <RP6Examples>\RP6ControlExamples\Example_03_Microphone
Source file: RP6Control_Microphone.c

The program will output messages to the serial interface and the LC-Display!

The Robot will *not* move in this example program!

The Microphone sensor may be used for detecting loud noises. The sample program displays the measured noise-level as a bar graph on the LCD and with the LEDs. Additionally the measurement values are shown. Tip your finger on the microphone or make some other loud noise to check the functionality. Clap your hands or knock on wood while observing the display, the LEDs and the messages.

Example 4: External EEPROM
Directory: <RP6Examples>\RP6ControlExamples\Example_04_EEPROM
Source file: RP6Control_04_EEPROM.c

The program will output messages to the serial interface and the LC-Display!

The Robot will *not* move in this example program!

This example program demonstrates how to read and write the external EEPROM. Initially the program reads and displays the first two "Pages" (each of which are 64 Byte long). In order to check the EEPROM's ability to store contents inside a powered off robot, you may turn off the RP6 after the program has written data to the EEPROM and switch it on again! The written data is preserved and shown!

Now the program overwrites the first page with 64 bytes and rereads the first 128 bytes to check, whether the procedure really altered the first page's contents and did not accidentally disturb the rest of the EEPROM's contents (which would change these storage cell's contents from the default value 255).

The program now proceeds by writing and reading individual bytes. Initially a value 128 will be written to the memory cell with address 4 and subsequently the program reads the first two pages – now using a byte by byte reading mode, which of course is processing slower than directly transferring a complete memory page.

If everything is done, the program will rest in an infinite running light loop.

Example 5: Analog/Digital Converter (ADC) and I/O Ports
Directory: <RP6Examples>\RP6ControlExamples\Example_05_IO_ADC
Source file: RP6Control_05_IO_ADC.c

The program will output messages to the serial interface and the LC-Display!

The Robot will *not* move in this example program!

Even if it works just the same as on the robot, we will demonstrate how to use the free ADCs and I/Os in this example. They are used frequently on this module, so an example dedicated to this topic may be very helpful for beginners.

The program uses the following names for the ADCs and I/Os:

```
ADC7    (1 << PINA7) // ADC Channel #7  - may also be used as a standard I/O Pin
ADC6    (1 << PINA6) // Channel # 6 ...
ADC5    (1 << PINA5)
ADC4    (1 << PINA4)
ADC3    (1 << PINA3)
ADC2    (1 << PINA2) // Channel # 2. Channels #0 and #1 have been reserved for keys and microphone.
```

```
IO_PC7  (1 << PINC7) // I/O-Pin 7 at PORTC
IO_PC6  (1 << PINC6) // Pin 6 ...
IO_PC5  (1 << PINC5)
IO_PC4  (1 << PINC4)
IO_PC3  (1 << PINC3)
IO_PC2  (1 << PINC2) // I/O-Pin #2 at PORTC
IO_PD6  (1 << PIND6) // I/O Pin #6 at PORTD
IO_PD5  (1 << PIND5) // I/O Pin #5 at PORTD
```

(see definitions in header file RP6Control.h)

The program will not perform anything useful – and you do really need to run this program. There's nothing connected to the I/O Pins by default so this does not make any sense. You have connected your own hardware to the I/Os and write your own control program for it.

Example 6: I²C Bus Interface – Master Modus
Directory: <RP6Examples>\RP6ControlExamples\Example_06_I2CMaster
Source file: RP6Control_06_I2CMaster.c

This program demonstrates how to use the I²C Bus in master mode. The I2C-Slave example program must be loaded to controller on the mainboard before you can run this example!

The example demonstrates how to control the MEGA32 on the mainboard in slave-mode. For proper operation you have to load the I2C-Slave example program (available in the RP6Base example folder) into the controller on the RP6 mainboard before!

The I²C Bus access is done in a similar way as for the controller on the mainboard. We use the exactly the same functions here.

The sample program allows you to transfer I²C Bus-commands to the mainboard controller (using the previously installed slave program) by hitting one of the five keys. Key T1 will increment a counter by one and transmit a setLEDs command with this counter-value to the slave. This simple procedure displays a binary counter with the mainboard's 6 Status LEDs.

At hitting key T2 the program will read all registers and output the contents on the serial interface. In contrast, T3 only reads the light sensor's values and additionally displays the results on the LC-Display.

T4 and T5 execute a "Rotate"-command and force the robot to perform a small rotation clockwise and counter-clockwise, respectively (of course you are allowed to hit the key a few times, which will cause the robot to continue to rotate...).

Basically this program – and all other programs as well – may be modified and is an extremely useful start point for testing new I²C-devices or any additional functionality you want to include in the slave example program.

Example 7: I²C Bus Interface – Master Modus – react on external Interrupts
Directory: <RP6Examples>\RP6ControlExamples\Example_07_I2CMaster
Source file: RP6Control_07_I2CMaster.c

This program demonstrates how to use the I²C Bus in master mode. The I2C-Slave example program must be loaded to controller on the mainboard before you can run this example!

Maybe you already noticed the interrupt-signals on the RP6's XBUS connectors? These interrupt signals allow you to react on sensor state changes without constantly polling the slave for new data. This will improve the system's performance as any bus access will cost some time.

A good example for using interrupts is the robot's ACS. In this case, the sensor state will rarely change (e.g. compared to an ADC which could alter its state all the time) and permanently evaluating the status would be quite inefficient.

As soon as the ACS-status changes the slave-program sets the INT1-signal to high level. INT1 is connected to the MEGA32's Interrupt input 0 on the RP6 CONTROL M32. The interrupt allows the controller to react immediately to this event and to evaluate the controller's status on the mainboard.

However, in the sample programs we will NOT use interrupt routines to react on this event, but we will poll the pin status in the main-loop instead. The I2C-Bus transfers are interrupt based and we can not use interrupt routines to initiate new transfers. In order to handle I2C-transfers, we have to frequently call the task_I2CTWI() function from the main-loop. For this reason an interrupt-routine would not be helpful as we would have to set a status flag and perform the rest in the main loop anyway. Of course this is possible, too – but we get the same result by monitoring the pin state. In fact, it would even cause problems to abort pending I2C Bus-transfers. Thus we decided to implement a task_checkINT0()-function for continuously checking the interrupt signal and eventually read the slave's status. As soon as we read the slave's status register with address 0, the interrupt-signal gets cleared. We may evaluate the first three registers to find out, which source triggered the interrupt.

The sample program has been designed to perform exactly like this and it will output the current ACS-status on different channels: on the 4 LEDs, on the LCD and on the serial interface as well. Additionally, an acoustic signal will be generated with the beeper.

Initially the program presets the ACS transmitter power-level with a specific I²C Bus command. The program does not only check the ACS, but also reacts on bumper-events and to any incoming RC5-transfers from remote controls or other robots.

Basically we may use this kind of control-flow for all other robot's sensors. Also for other expansion modules that may be released in the future.

Another interesting detail is the “Heartbeat”-display of the program. The task_LCD-Heartbeat() function constantly triggers the system to display a blinking asterisk-character '*' on the LCD. The blink-frequency has been set to 1Hz. The “Heartbeat”-signal is helpful for debugging any program that could hang-up/crash as you will immeadiately see if this happened. It will help you to isolate and analyse the program's functions causing the hang-up. These hang-ups may occur quite often during program development.

Therefore the I²C Slave sample program provides a “software-watchdog” for the mainboard's controller. If the master is not responding to an interrupt-event (by reading register 0) within a given time limit, the watchdog will stop all robot movements. Most important is to stop the motors! Imagine the slave receives a command to drive forwards at a speed of 10cm/s and then the master hangs-up! This would cause the robot to crash into the next obstacle at high speed without a chance to react on a sensor detecting the obstacle...

Initially the software watchdog-timer is deactivated. In order to activate the watchdog, you have to send a special command through the I²C Bus. You may also configure the watchdog to trigger an interrupt at 500ms-intervals to check the master-controller's activity. We will use this concept in the next example.

Example 8: I²C Bus Interface – reduced RP6 Library
Directory: <RP6Examples>\RP6ControlExamples\Example_08_I2CMaster
Source file: RP6Control_08_I2CMaster.c

This program demonstrates how to use the I²C Bus in master mode. The I2C-Slave example program must be loaded to controller on the mainboard before you can run this example!

By adding lots of functions and other things into a single source, you may soon loose the overview. For this reason we decided to split the previous example program 7 into two C-files and to add some additional functionality. It is kind of a reduced RP6 library for the I²C Bus robot control and allows you to virtually use this library just like the normal RP6Lib for the mainboard's controller. In order to simplify reusing of RP6Lib's program code for the RP6ControlLib, we defined a number of functions and variables identically to the RP6Lib equivalents. We also provided the library with well-known event handler functions for ACS, IRCOMM and the Bumpers. The library also contains new event handlers for low battery level and the watchdog-requests. Additionally the following example demonstrates how to control the robot's movements by using the available functions.

The program's structure is similar to example 7. As already announced, we added a new watchdog-timer function, which shows it's requests on the LCD. The program will also read all sensor registers and output the results to the serial interface. Of course the program will also output the ACS-, Bumper- and RC5-events.

Example 9: I²C Bus Interface – controlling the robot's movements
Directory: <RP6Examples>\RP6ControlExamples\Example_09_Move
Source file: RP6Control_09_Move.c

This program demonstrates how to use the I²C Bus in master mode. The I2C-Slave example program must be loaded to controller on the mainboard before you can run this example!

ATTENTION: The Robot will move in this example program!

Right now we will add a few movement control functions to the newly created library, which are well-known from RP6Lib: move, rotate, moveAtSpeed, changeDirection and stop. Usage of these functions is identical to the equivalent RP6Lib's functions. In order to simplify the code, we removed much of the program code from the last examples, except for the remote watchdog display. We want to try using the blocking mode of the movement controls and this would cause some things like the heartbeat display to stop working anyway. This example program will command the robot to be moving back and forth and rotate for around 180° - just like in example 7 for RP6Lib ("RP6Base_Move_02.c").

Example 10: I²C Bus Interface – Verhaltensbasierter Roboter
Directory: <RP6Examples>\RP6ControlExamples\Example_10_Move2
Source file: RP6Control_10_Move2.c

This program demonstrates how to use the I²C Bus in master mode. The I2C-Slave example program must be loaded to controller on the mainboard before you can run this example!

ATTENTION: The Robot will move in this example program!

The newly created library allows you to copy almost everything of the example programs for behaviour controlled robots you already know from the RP6Lib examples. This is exactly how we derived this example program from "RP6Base_05_Move_05". We only needed to do a few minor modifications – e.g. the mainboard's LEDs have to be controlled by a "setRP6LEDs"-function, as we use "setLEDs" for the RP6-M32's LEDs already...

In other respects the program will behave almost identical to it's counterpart – the robot will cruise around and tries to avoid obstacles. The program's functionality is identical, but the robot is controlled by the RP6-M32 module!

Some new features have been added, too. Now the LCD- and LEDs on the RP6-M32 show the currently activated behaviour. This allows you to check the behaviour status. We designed a tiny subroutine to make sure that identical strings will be transmitted only once to the display. Otherwise this would result in a flickering display. In case the "Cruise"-behaviour is active, the four red status LEDs will perform a running light.

The program also checks the battery's condition. At very low levels, the robot gets stopped. However, it will take some time until freshly charged batteries reach this low level...

Initially the program will wait for three loud noises (you will see a "WAIT" message on the second display line, followed by a counter for these noises) – e.g. clap your hands three times! As an alternative you may hit any key of the RP6-M32, which has been implemented as another seperate behaviour.

OK – we've reached the end of this short additional manual. You may now give your fancy full scope and start writing new programs or create new sensors for the RP6, controlled from the RP6-M32 - or do anything else you can imagine...

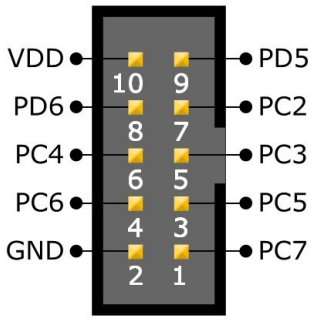
APPENDIX

A – Connector pinouts

This overviews provides you with the most important connector pinouts on the mainboard. Additionally we extended the list by a number of details for usage.

The serial interface connector pinout is identical to the mainboard's connector and of course the same pinouts are used for the XBUS- and USRBUS-connectors!

I/O-ports:

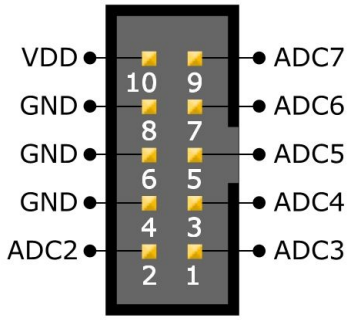


The I/O-connector provides all freely available I/O-ports and the +5V supply voltage.

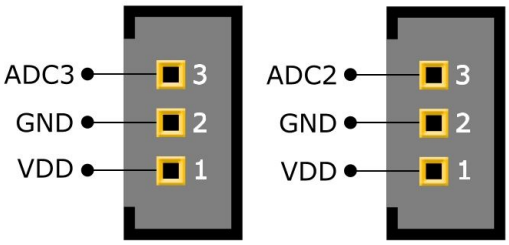
PC2, PC3, PC4, PC5, PC6, PC7, PD5 and PD6.

ATTENTION: Please do not interconnect this connector's supply voltage with another module's (e.g. an experiment expansion module) supply voltage to prevent large ground loops or equivalent problems.

ADC-channels:



6 freely available ADC-channels (which of course may also be used as I/O-pins) are available on the 10pin ADC connector, which also provides the supply voltage.



In analogy to the mainboard, two of the ADCs are available on unpopulated pads. This will allow you to solder your own connectors with a 2.54mm grid.

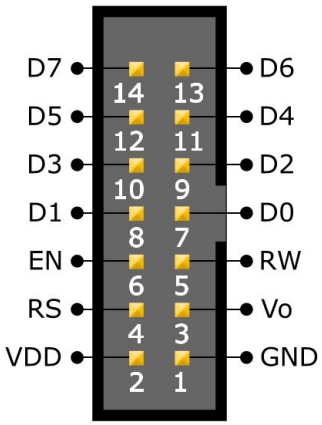
But be careful and avoid "excessive soldering"! You will need some experience in soldering to handle this.

You may use these pads for connecting any analog or digital sensors (the output voltage of these sensors may range between 0 and 5V). The connectors will also provide the sensors with a 5V supply voltage. Eventually a large electrolytic capacitor should be added – up to 220 or 470µF (do not exceed this value! The capacitor's operating voltage has to be minimal 16V) will be sufficient for most applications.

However you will probably not really need a big electrolytic capacitor, unless you are working with sensors, which require high peak currents – e.g. the popular Sharp IR distance sensors. Decoupling capacitors (100nF) soldered to the board are suitable for short supply lines only – for long supply lines, they would have to be directly soldered

on the sensor's connection pads (we advise to directly mount these capacitors on the sensor's pads even for short supply lines as well!).

LCD connector:



If you do not wish to use the pre assembled standard LCD, you may alternatively build your own cable by using the correct connector pinout for your specific LCD.

The signal lines D0, D1, D2 ,D3, RW have been grounded to signal GND, as we will only use the LCD in 4 Bit mode and do not read from the device.

Be careful to use the correct pinout and avoid mirrored connections of the plug!

The pin names may vary from manufacturer to manufacturer, but usually the pin names are identical to those shown in our documentation. In this case you may connect the LCD signals 1:1.