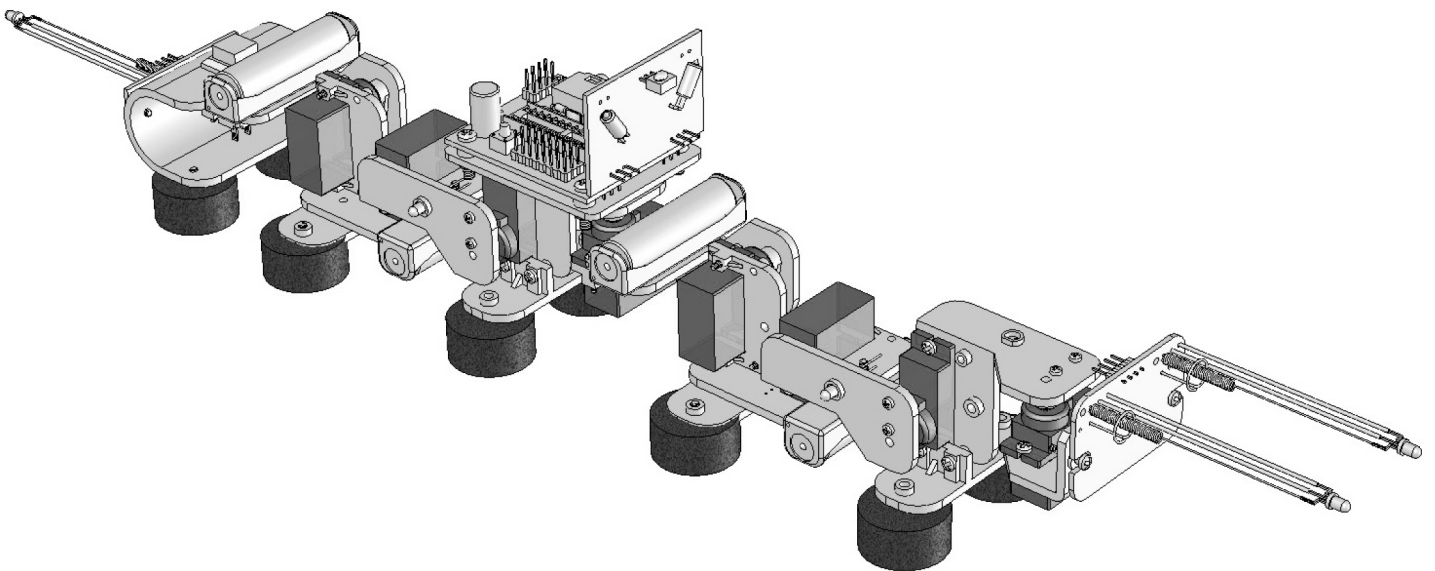




---

# Model: R500 Robotic Caterpillar



[globalspecialties.com](http://globalspecialties.com)

800-572-1028

# 1. Table of Contents

1. Product description CATERPILLAR	5
2. Tools and parts	6
3. Assembly instructions	8
4. Software	23
4.1 Java	29
4.2 Connection USB programmer - Windows	30
4.3 Connection USB programmer - LINUX	30
4.4 First hardware test	31
5. Programmer and loader	33
5.1. Testing the USB programmer and Robotloader	35
5.2. Selftest	36
6. Programming the CATERPILLAR	37
6.1. Open and compile a program	38
6.2. WHY C and what's GCC	40
7. Crash Course for beginners	41
7.1 First program	42
7.2 Basics	44
7.3 Functions	52
7.4 Caterpillar Library	58
<b>APPENDIX</b>	
A. PCB'S	72
B. Circuits	74

**CATERPILLAR Robot is a registered trademark.**

© English translation (March 2013): Global Specialties (Yorba Linda, CA).

This manual is protected by laws of Copyright. Any full or partial reproduction of the contents are forbidden without prior written authorization by :

**Global Specialties.**

©2013 Global Specialties

22820 Savi Ranch Parkway  
Yorba Linda, CA  
92887

Ph: (800) 572-1028  
Fax.: (215) 830-7370

E-Mail: info@globalspecialties.com

This manual is protected by the laws of Copyright. It is forbidden to copy all or part of the contents without prior written authorization.

Product specifications and delivery contents are subject to changes. The manual is subject to changes without prior notice.

You can find free updates of this manual on **globalspecialties.com**

"Caterpillar Robot" is a registered trademark.

All other trademark are the property of their owners. We are not responsible for the contents of external web pages that are mentioned in this manual.

### **Information about limited warranty and responsibility**

The warranty granted by Global Specialties is limited to the replacement or repair of the Caterpillar and its accessories within the legal warranty period if the default has arisen from production errors such as mechanical damage or missing or wrong assembly of electronic components except for all components that are connected via plugs/sockets.

The warranty does not apply directly or indirectly to damages due to the use of the robot. This excludes claims that fall under the legal prescription of product responsibility.

The warranty does not apply in case of irreversible changes (such as soldering of other components, drilling of holes, etc.) of the Caterpillar or its accessories or if the Caterpillar is damaged due to the neglect to the instructions of this manual.

The warranty is not applicable in case of neglect to follow this manual. Please adhere above all to the "Safety recommendations" in the Caterpillar manual.

Please note the relevant license agreements on the CD-ROM.

### **IMPORTANT**

Prior to using this robot arm for the first time, please read this manual thoroughly up to the end. They explain the correct use and inform you about potential dangers. Moreover they contain important information that might not be obvious for all users.

### **Important safety recommendation**

This module is equipped with highly sensitive components. Electronic components are very sensitive to static electricity discharge. Only touch the module by the edges and avoid direct contact with the components on the circuit board. Please never overload the servos and read all warnings carefully.

## Symbols

This manual provides the following symbols:



*The "Attention." Symbol is used to mark important details. Neglecting these precautions may damage or destroy the robot and/or additional components and additionally you may risk your own health or the health of other persons.*



*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".*

## Safety recommendations

- Check the polarity of the batteries or power supply.
- Keep all products dry, when the product gets wet remove the batteries or power directly.
- Remove the batteries or power when you are not using the product for a longer period.
- Before taking the module into operation, always check it and its cables for damage.
- If you have reason to believe that the device can no longer be operated safely, disconnect it immediately and make sure it is not unintentionally operated.
- Consult an expert if you are unsure as to the function, safety or connection of the module.
- Do not operate the module outdoors or under unfavourable conditions.
- Do not overload the servos.
- Do not assemble the robot in presence of small children.
- Read all the warnings in this manual.
- This module is equipped with highly sensitive components. Electronic components are very sensitive to static electricity discharge. Only touch the module by the edges and avoid direct contact with the components on the circuit board.

## Normal use

This product was developed as an experimental platform for all persons which are interested in robotics. The main goal is to learn how you can program the device in C-language. This product is not a toy, it is not suitable for children under 14 years of age.

The Caterpillar is also not an industrial robot, with industrial specifications and performance. It may only be used indoors. The product must not get damp or wet. Also be careful with condensation. When you take it from a cold to a warm room give it time to adapt to the new conditions before you use it.

Any use other than that described above can lead to damage to the product and may involve additional risks such as short circuits, fire, electrical shock etc. Please read all the safety instructions and warnings in this manual.

# 1. Product description

Thank you very much for choosing our Caterpillar with its eight servomotors, various sensors, electronic components, hardware and metallic parts. It is an excellent training tool in learning the fundamentals for programming and electronics.

## Product Specification

1. Eight degrees of freedom (DOF)
2. User-programmable
3. Additional input ports for a great variety of sensors

Before you start the assembly phase we advise you to carefully read this manual. Strictly follow the assembly instructions to avoid any problems. Errors in the assembly phase may result in the robot's operations.

## Specifications:

Power	: 5.2V to 6V (4 AAA batteries) (batteries are not included)
Processor	: ATMEGA16
Servo	: 8 pcs mini servo
Power consumption	: ca. 100 mA in rest and 3A with all servos working
Height	: 90 mm
Length	: 500 mm
Width	: 85 mm

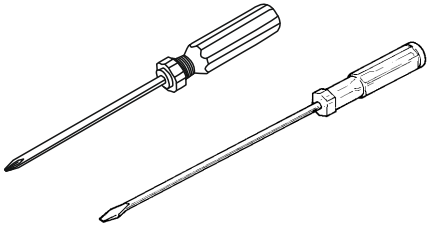


## Warning:

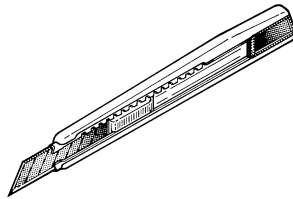
- \* The right of return does not apply after opening the plastic bags containing parts and components.
- \* Read the manual thoroughly prior to assembling the unit.
- \* Be careful when handling tools.
- \* Do not assemble the robot in presence of small children. They can get hurt with the tools or swallow small components and parts.
- \* Check the correct polarity of the batteries.
- \* Make sure that batteries and holder remain always dry. If the Caterpillar gets wet, remove the batteries and dry all parts as thoroughly as possible.
- \* Remove the batteries if the Caterpillar will not be used for more than one week.

## 2. Required tools and parts

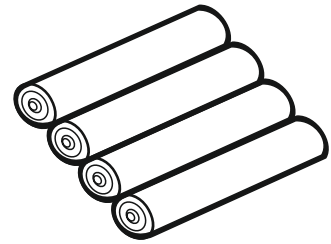
TIP: First read the complete manual before you start building the robot . Failures and mistakes during building can damage the robot and create malfunction during operation. Take special care when you connect all wires and the power.



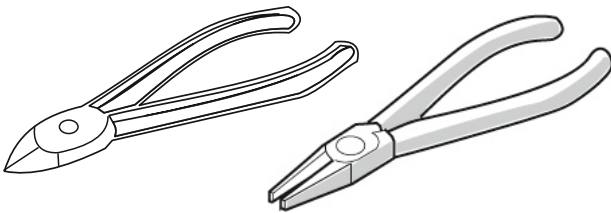
Screwdriver set



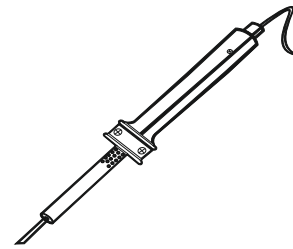
Hobby knife



4 Pcs. AAA batteries



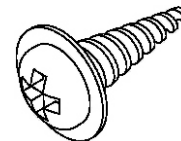
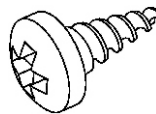
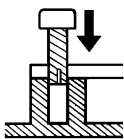
Pliers



Soldering iron and solder

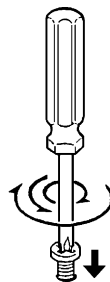
## Selftapping Screws

Selftapping screws behave like wood screws i.e.they cut a thread into the material in a rotating motion that functions like a nut. To this end, this type of screw has a larger thread and a sharper tip as a normal screw.



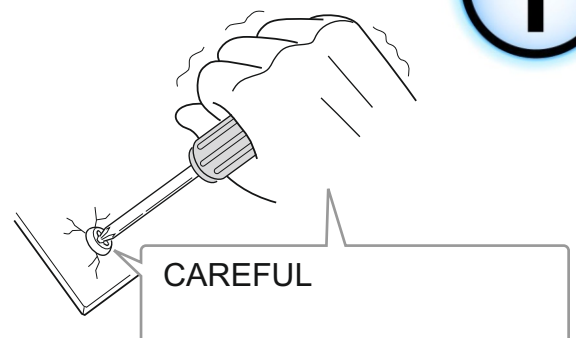
Selftapping screws have a cutout at the top that makes it easier to drill into the material. The best way to fasten such a screw is:

- 1 Drive the screw into the material
- 2 Slightly loosen the screw
- 3 Tighten the screw again

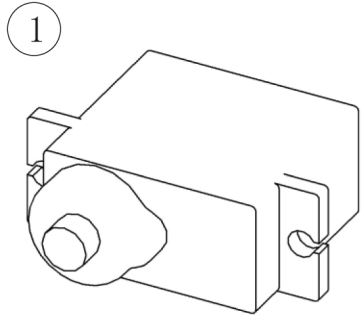


**Important:**

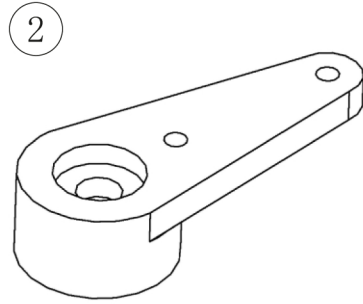
***If the screws are loosened and tightened too often, the hole enlargens gradually and the screw will no longer fit properly.***



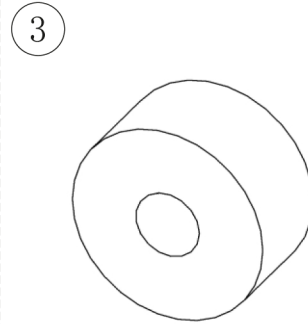
## 2.1. Partlist



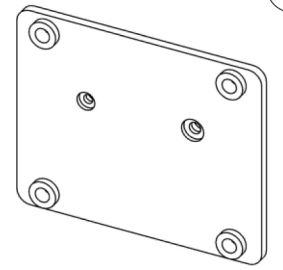
Miniature Servomotor  
O 8 pcs.



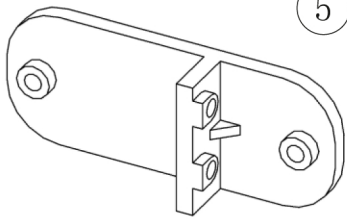
Lever arm for the Servomotor  
O 8 pcs.



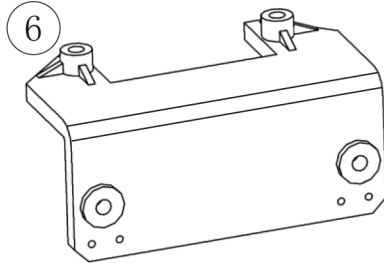
EVA pedestal feet  
O 10 St.



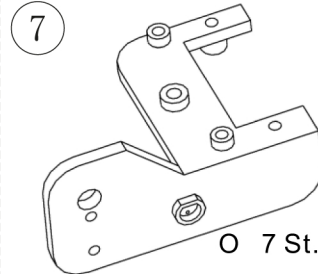
Base for the PCB mounting  
(PCB = Printed Circuit Board)  
O 1 St.



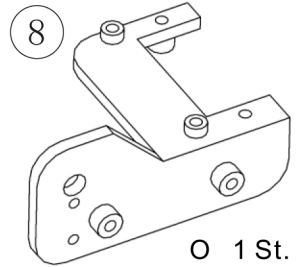
Foot-panel  
O 4 St.



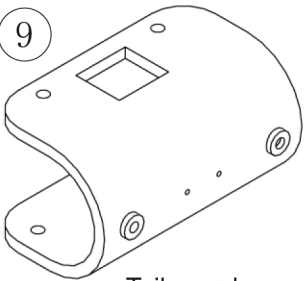
Head panel  
O 1 St.



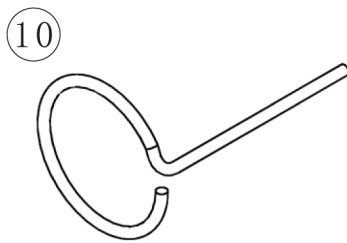
Body panel with a  
LED-holder  
O 7 St.



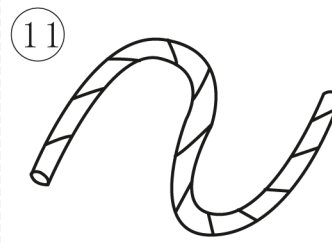
Body panel with a PCB  
mounting  
O 1 St.



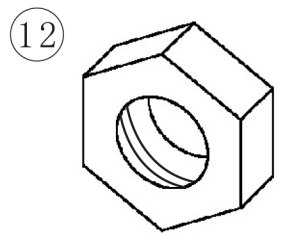
Tail panel  
O 1 St.



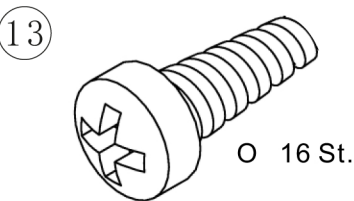
Sensor-ring  
O 3 St.



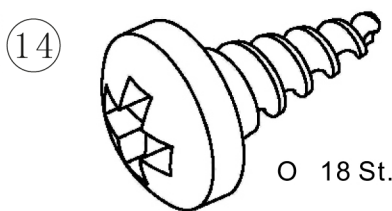
Spiral winding  
O 1 St.



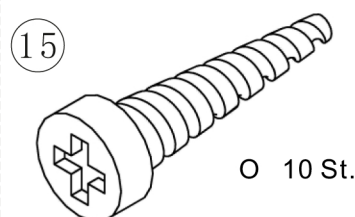
Nut m3  
O 4 St.



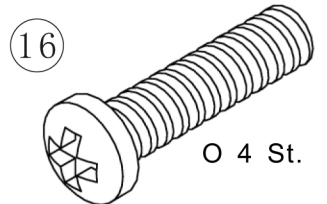
Screw  
Roundhead M2 x 6  
O 16 St.



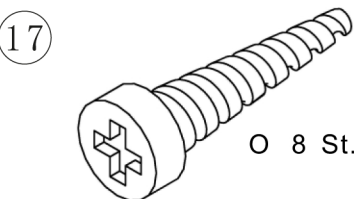
Selftapping screw  
Roundhead M2.6 X 6  
O 18 St.



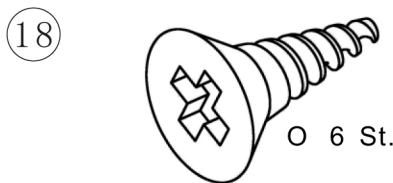
Selftapping screw  
Flathead M3 X 10  
O 10 St.



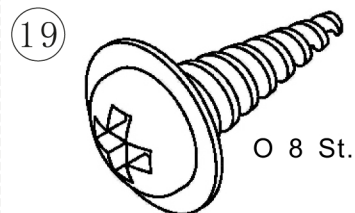
Screw  
Roundhead M3 x 12  
O 4 St.



Selftapping screw  
Roundhead M2.6 x 10  
O 8 St.



Selftapping screw  
Countersunk head M2.6 x 6  
O 6 St.



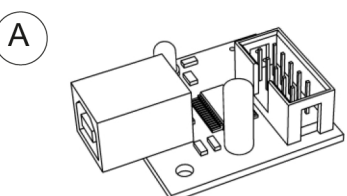
Selftapping screw  
Roundhead with ring M2 x 8  
O 8 St.



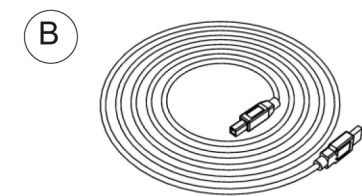
Tie Rap



Shrinking tube



USB PROGRAMMER



USB Cable

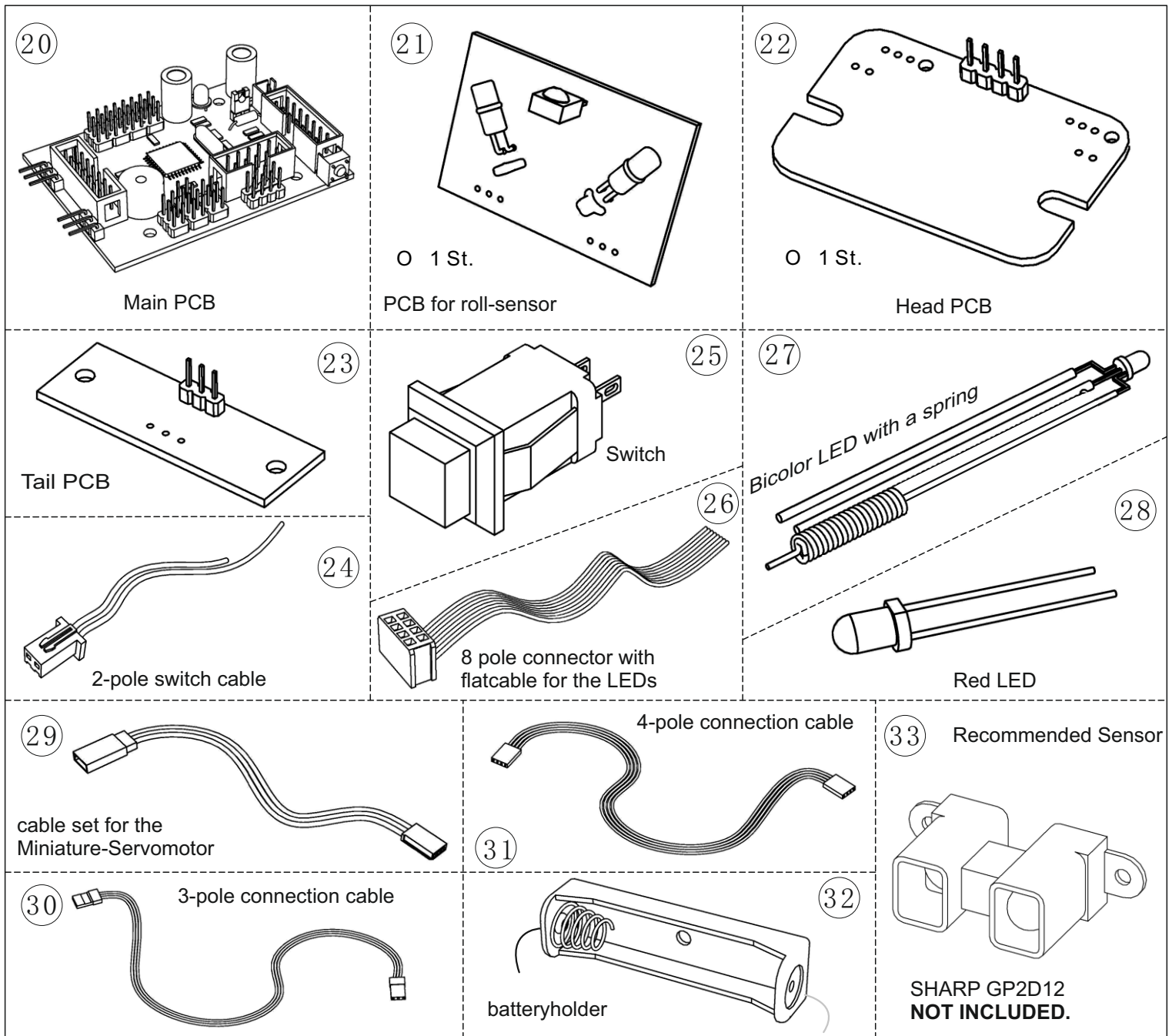


Programmer cable



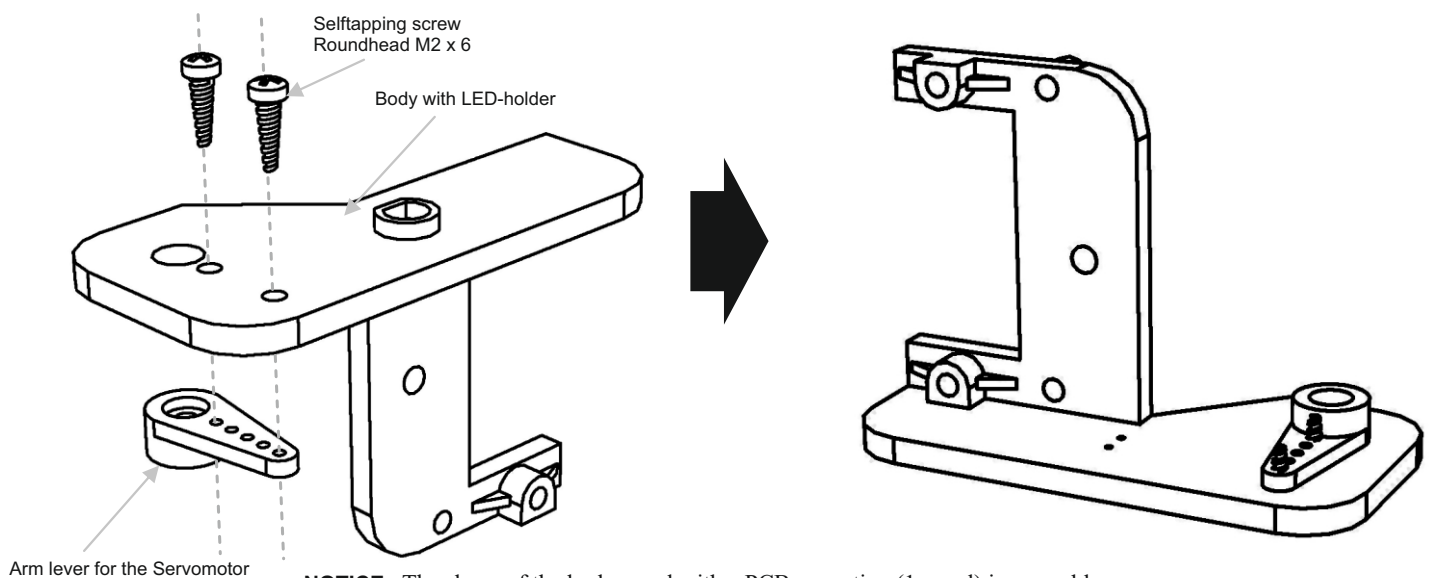
CD

## 2.1. Part List (continued)



## 3. Assembly instructions:

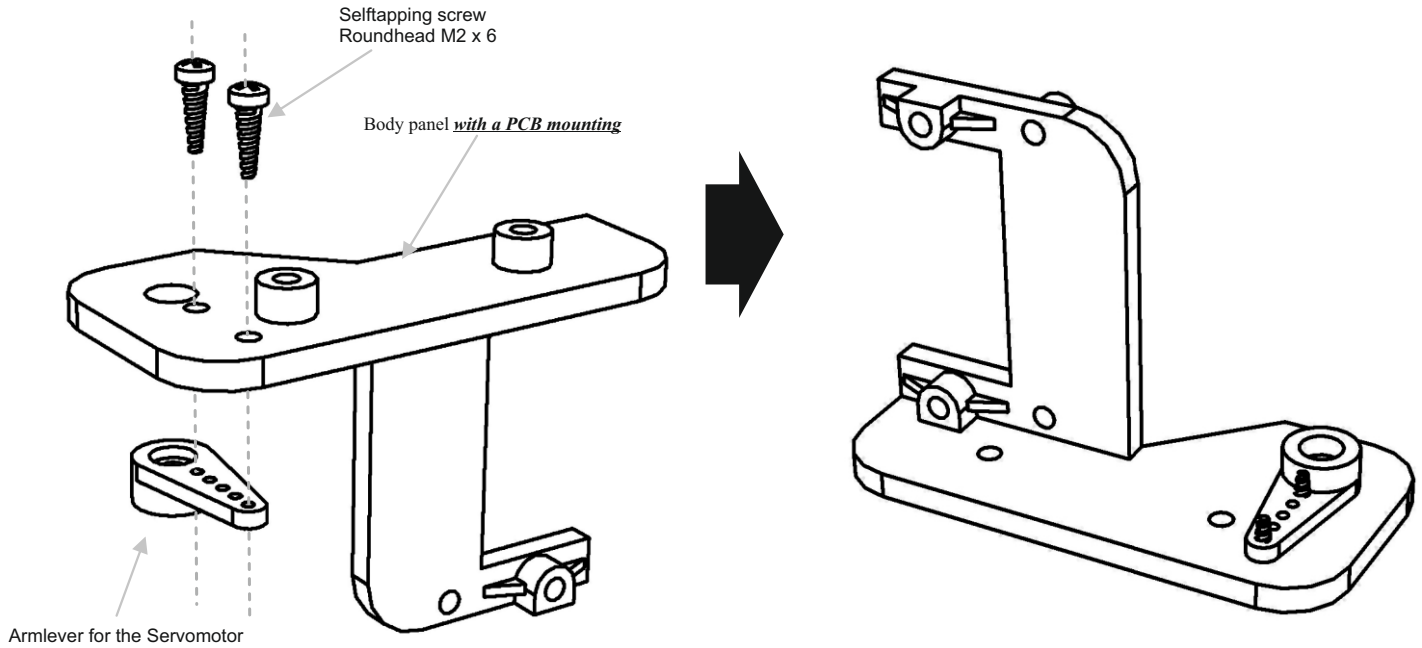
**Step 1:** Attach the lever arm for the servomotors to the body elements with a LED-mounting.



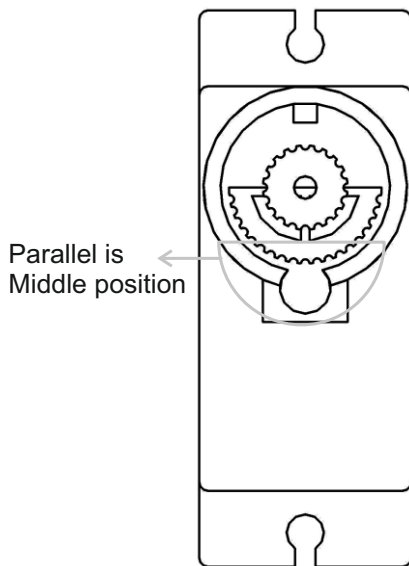
**NOTICE.** The shape of the body panel with a PCB-mounting (1 panel) in assembly step 2 is different from the body panels with a LED-mounting (7 panel elements)



**Step 2:** Attach the lever arm for the servomotor to the *body panel with a PCB-mounting*.



**Step 3:** Attach one of the servomotors (2) to one of the lever arms for the servomotors with LED-mounting.

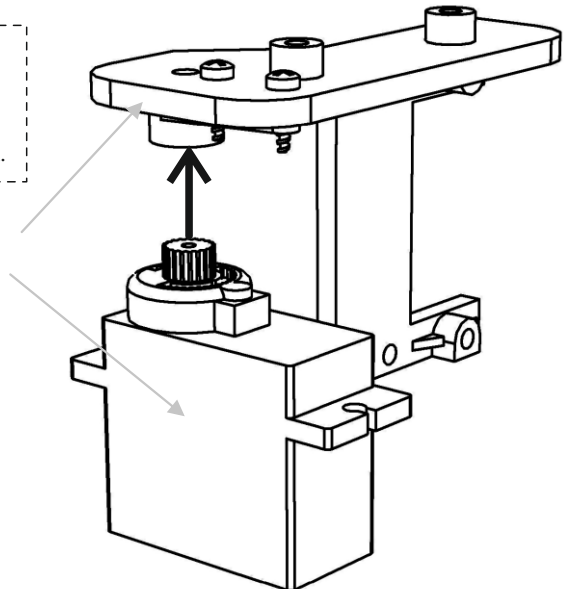


**IMPORTANT.**

Before mounting please position all servomotors into their central position.

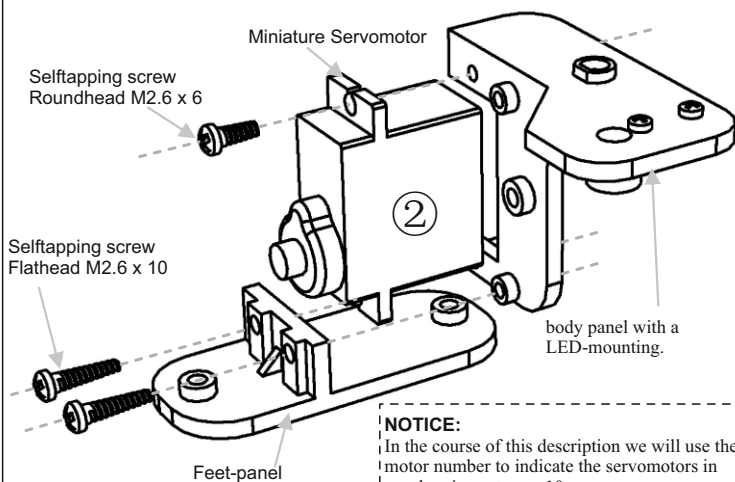
**IMPORTANT.**

Both sides must be positioned parallel.



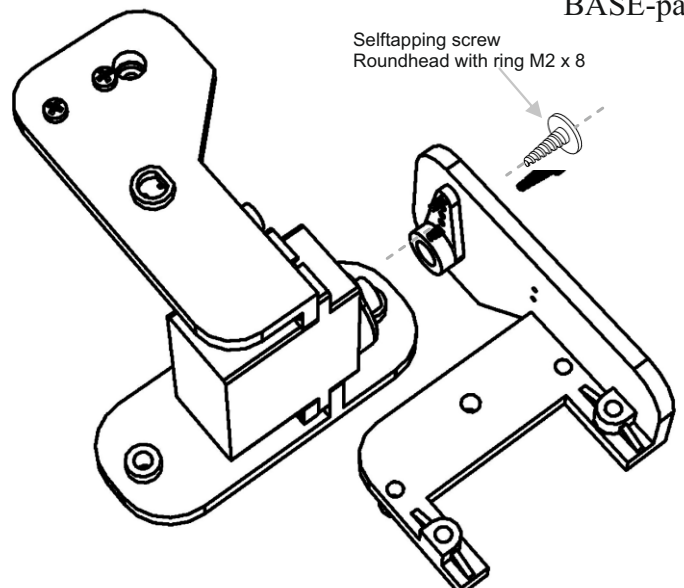
**NOTICE:** Be careful to rotate the servomotor in the central position before you attach this component to the lever arm for the servomotor. Attach the servomotor exactly according to the drawing.

**Step 4:** Attach the pedestal foot-panel to the body panel with a LED-mounting.

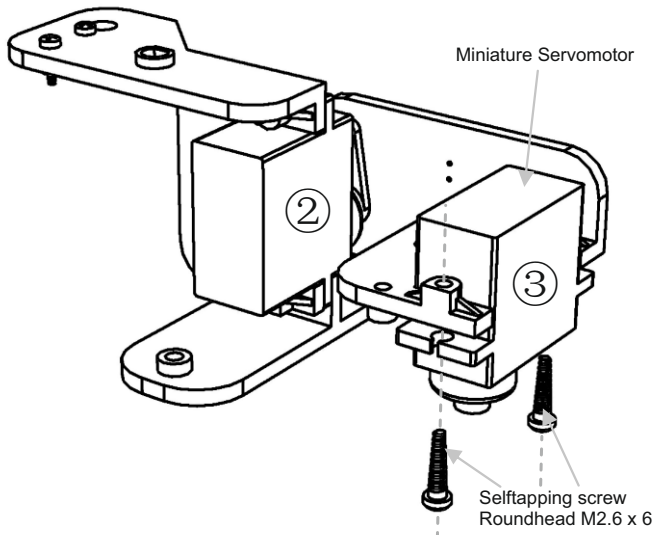


From now we will name each completely prepared panel as a BASE-panel.

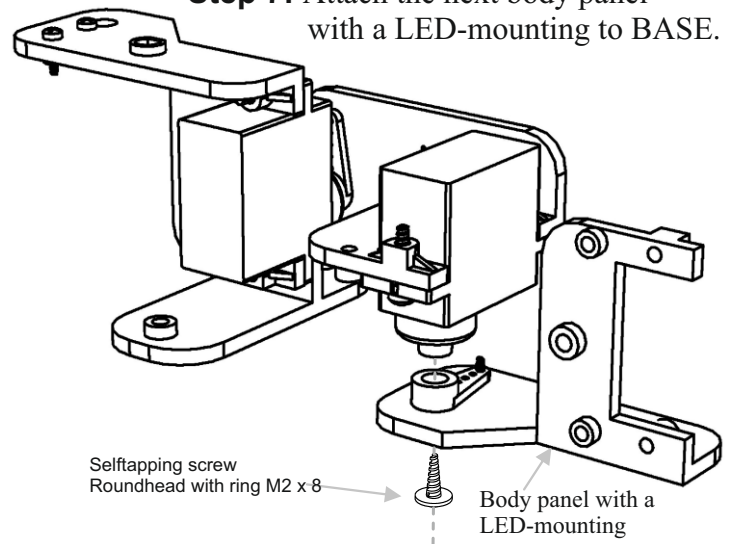
**Step 5:** Attach (as illustrated in the drawing) the next body panel with a LED-mounting to the BASE-panel.



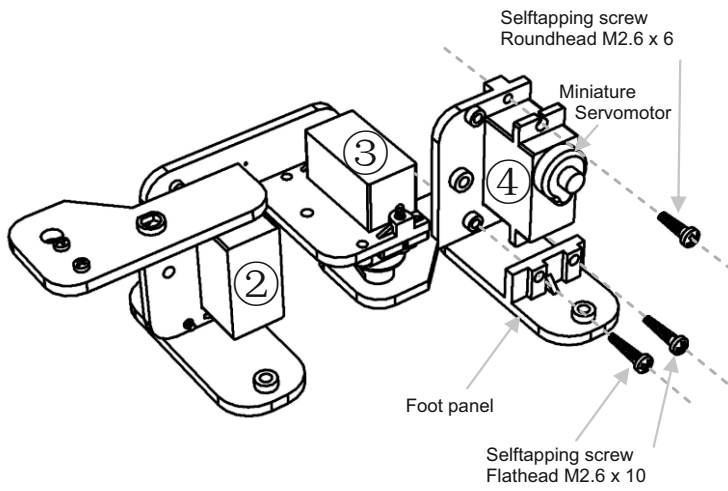
**Step 6:** Attach the next servomotor (3) to BASE.



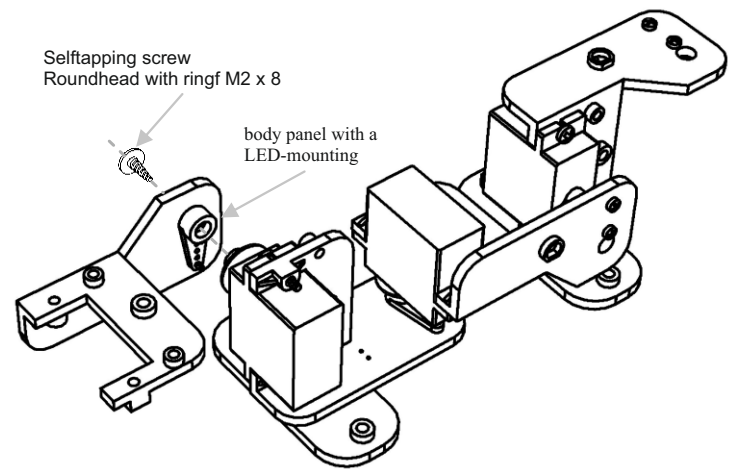
**Step 7:** Attach the next body panel with a LED-mounting to BASE.



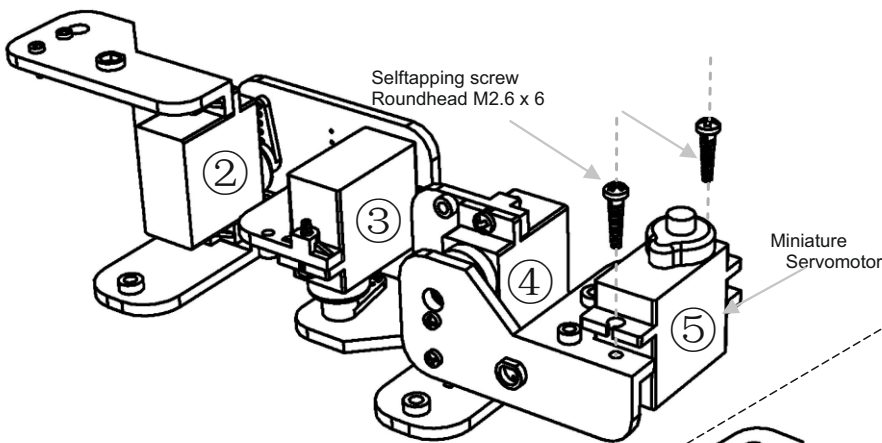
**Step 8:** Attach the next servomotor (4) and the Foot-pedestal to BASE.



**Step 9:** Attach the next body panel with a LED-mounting to BASE.

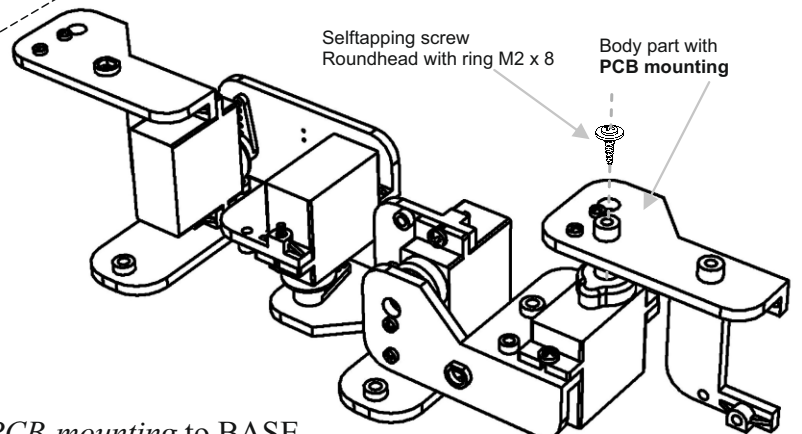


**Step 10:** Attach the next servomotor (5) to BASE.



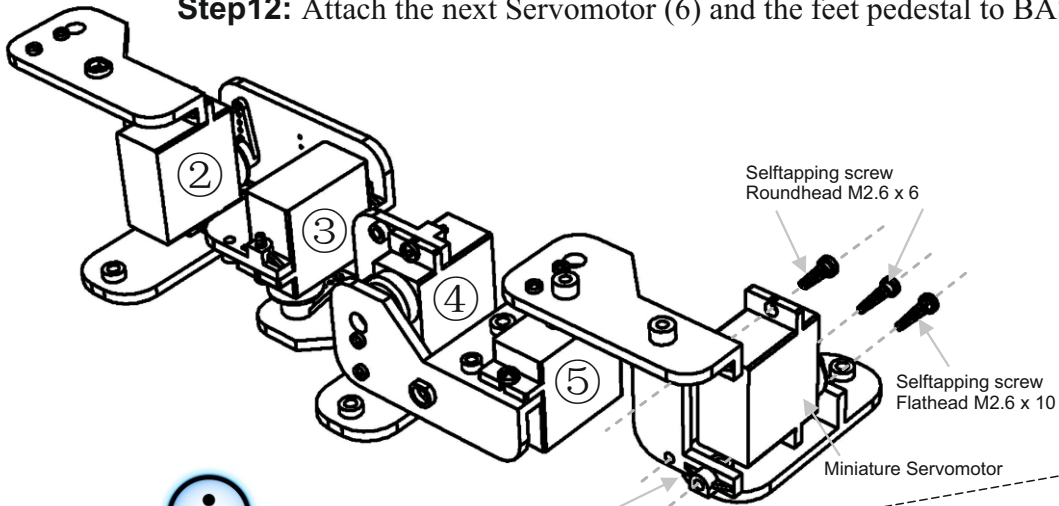
**NOTICE.**

Be sure that you are using the body panel with PCB mounting.



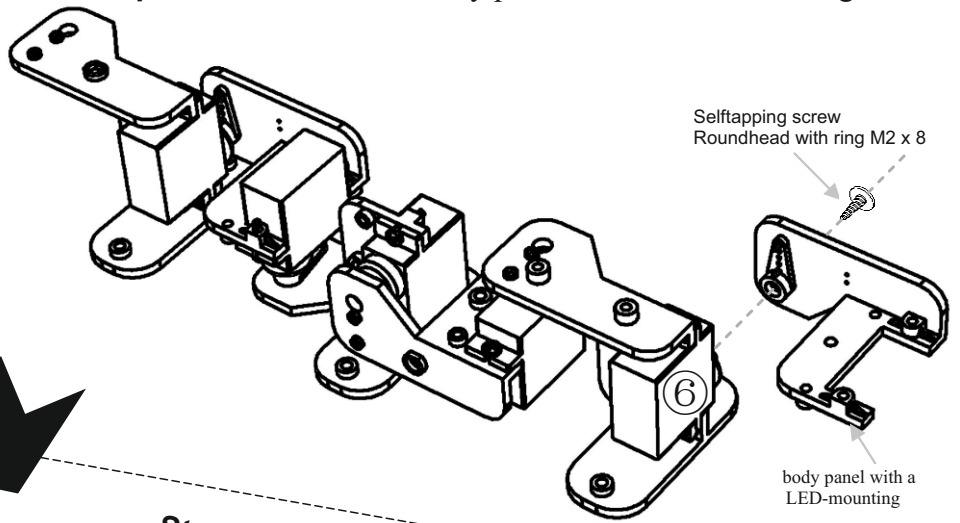
**Step 11:** Attach the next *body panel with a PCB-mounting* to BASE.

**Step12:** Attach the next Servomotor (6) and the feet pedestal to BASE.

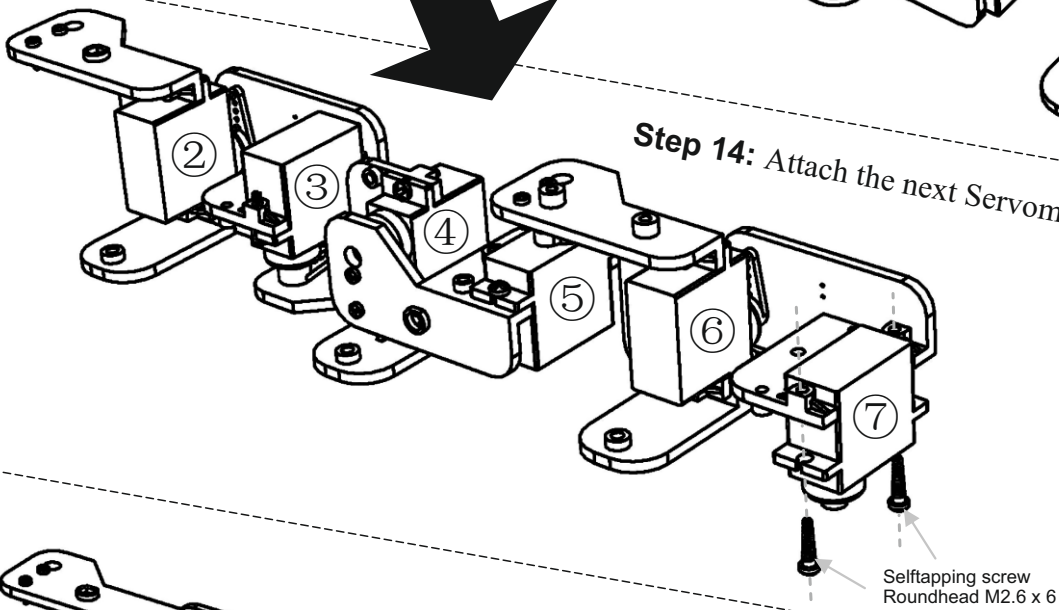


Wire should be under the Foot panel

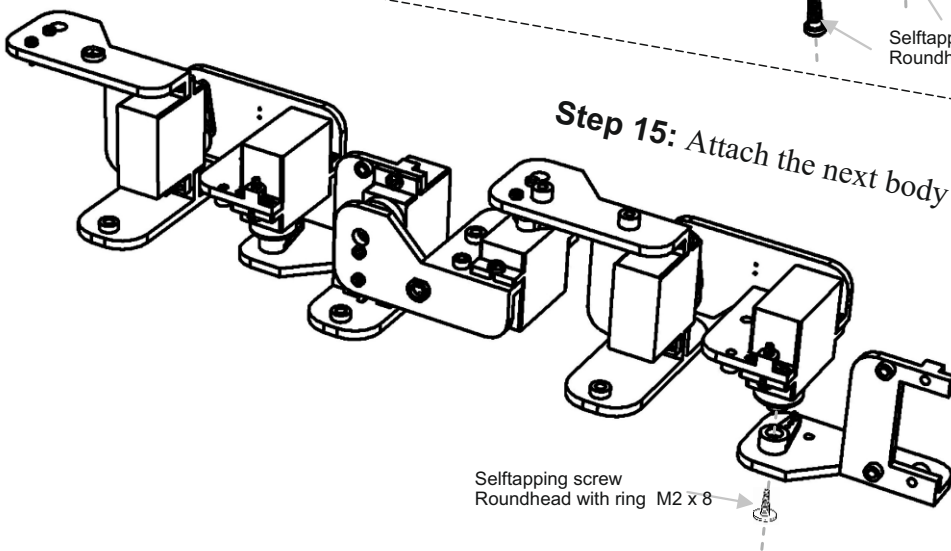
**Step 13:** Attach the next body panel with a LED-mounting to BASE.



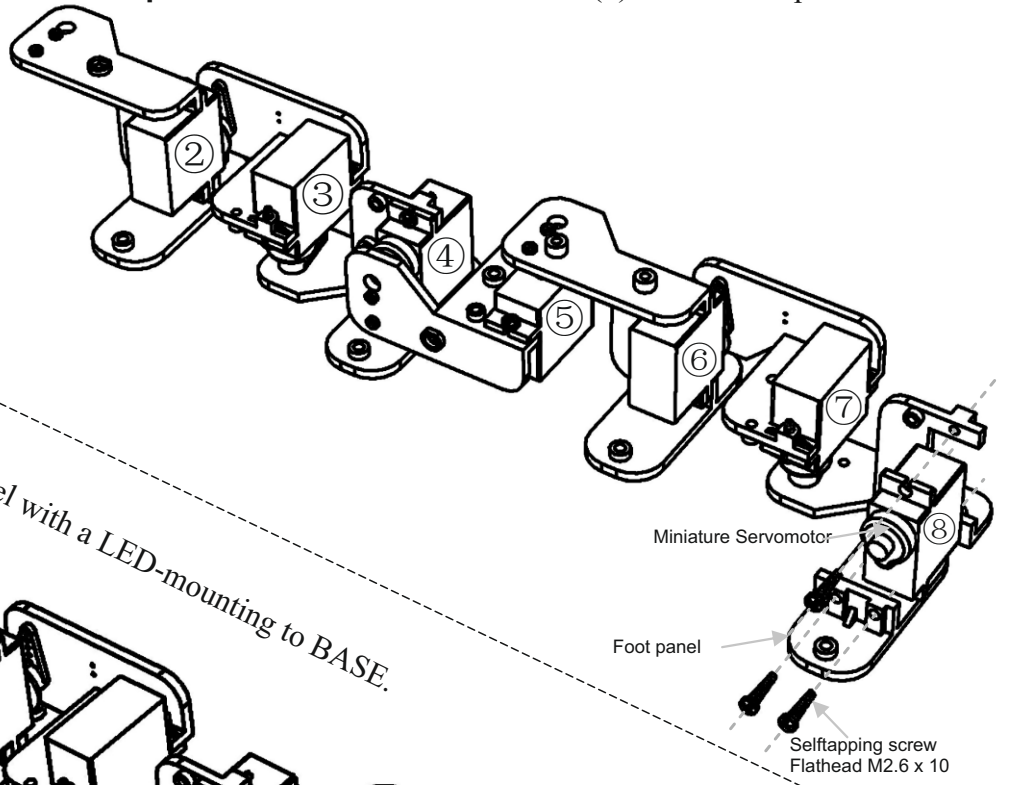
**Step 14:** Attach the next Servomotor (7) to BASE.



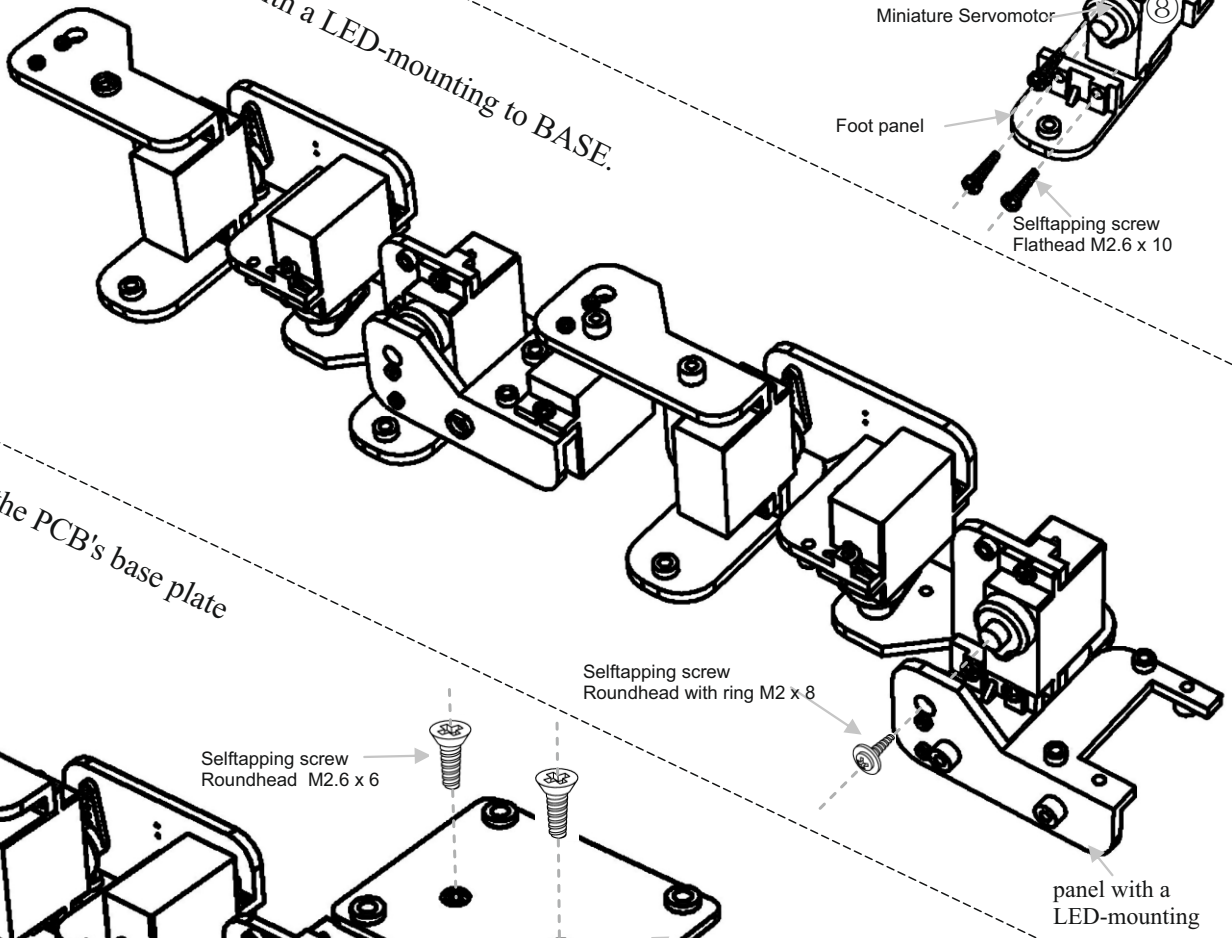
**Step 15:** Attach the next body panel with a LED-mounting to BASE.



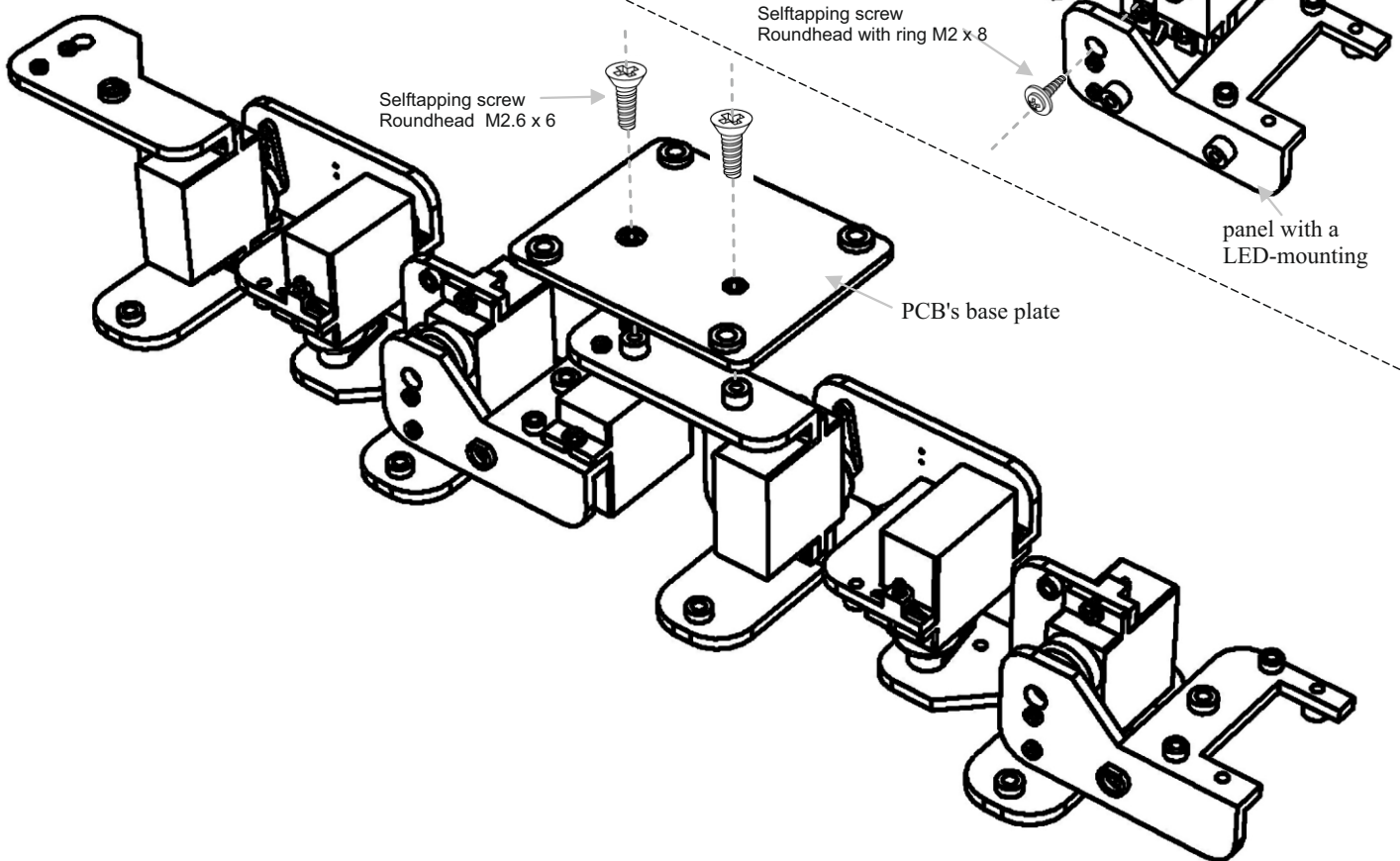
**Step 16:** Attach the next Servomotor (8) and the feet pedestal to BASE.



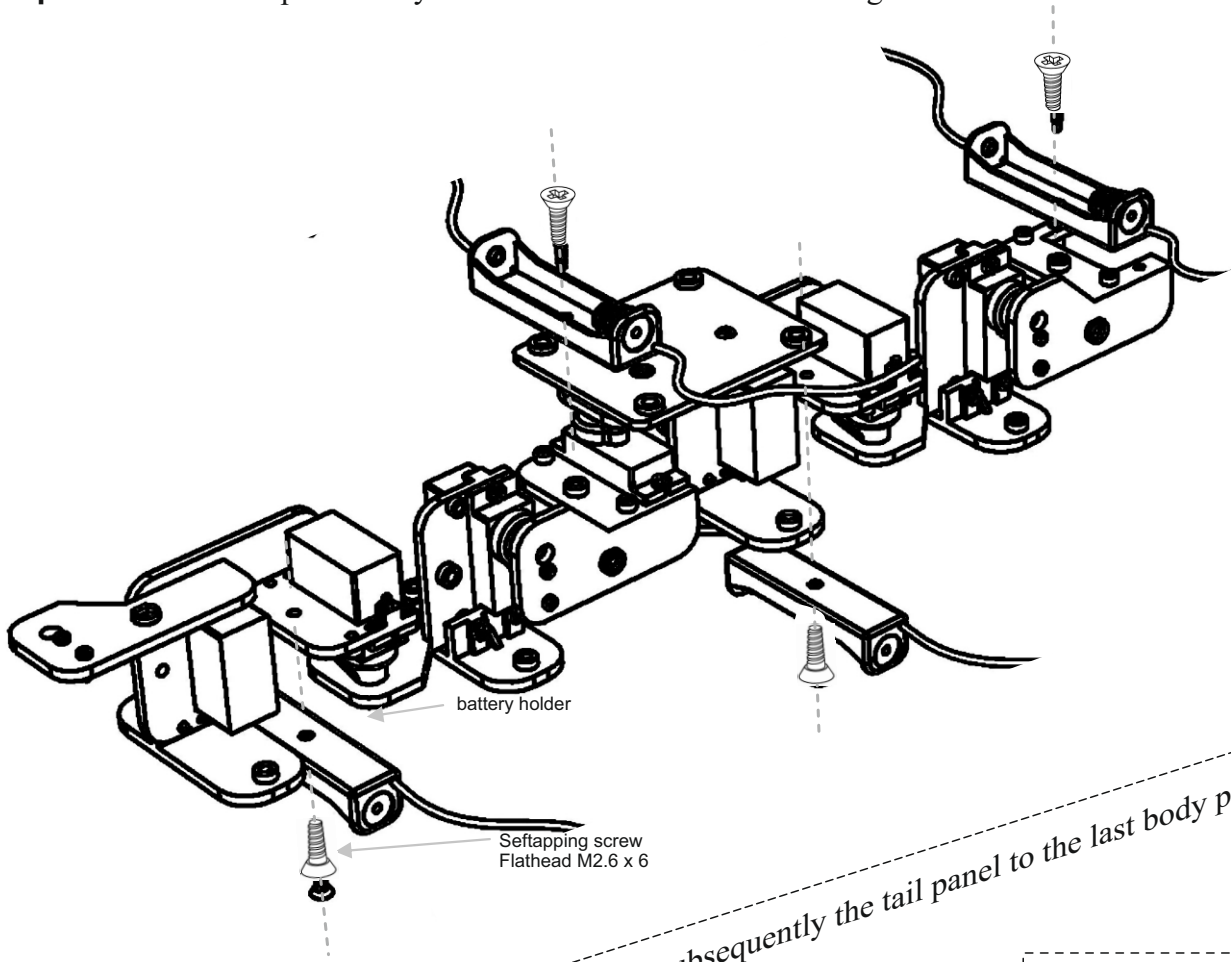
**Step 17:** Attach the next body panel with a LED-mounting to BASE.



**Step 18:** Attach the PCB's base plate



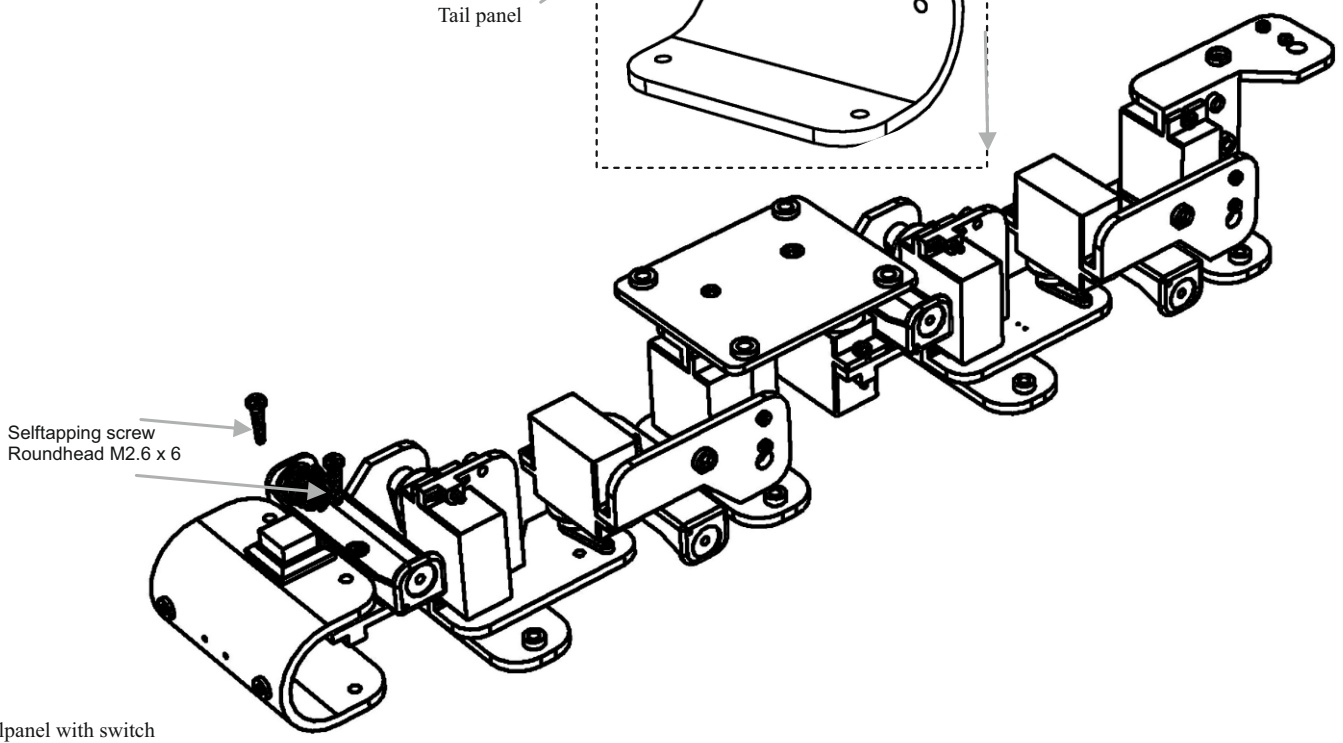
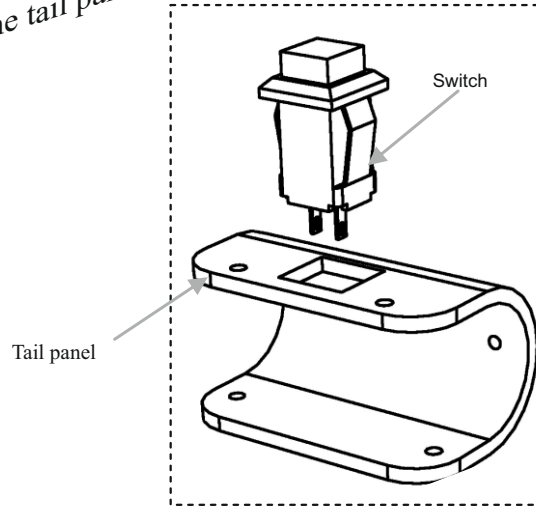
**Step 19:** Install the 4 pcs. battery holders as sketched in the drawing.



**Step 20a:** Attach the switch to the tail panel and subsequently the tail panel to the last body panel.

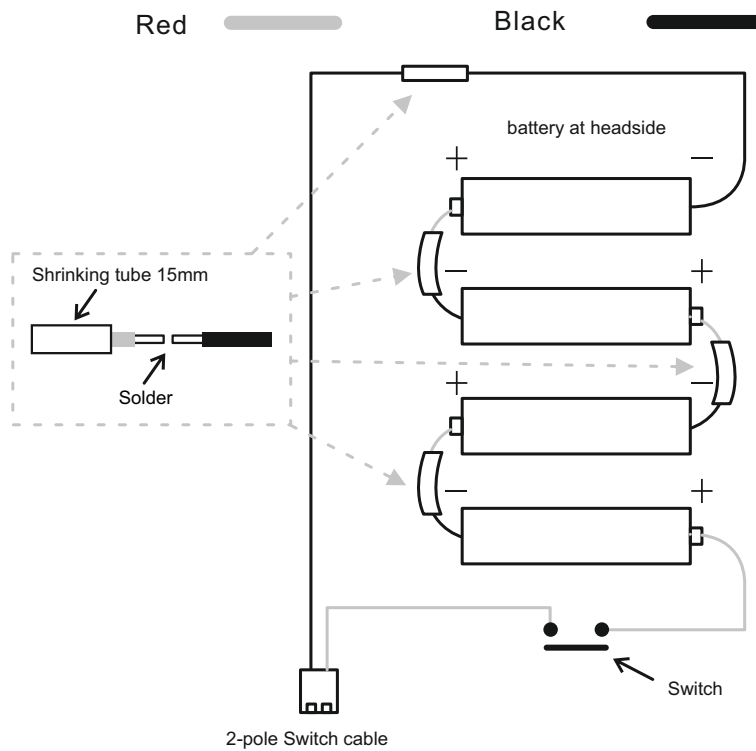


**IMPORTANT.**  
Solder first the red + wire from the battery and the 2- pole wires to the switch before you build in the switch, **see step 20b**



Tailpanel with switch

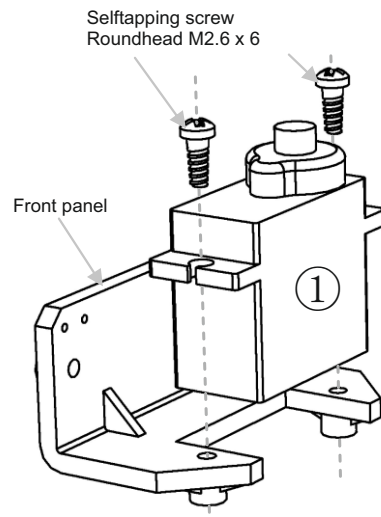
**Step 20b:** Solder the wiring cables to the battery compartments as illustrated.



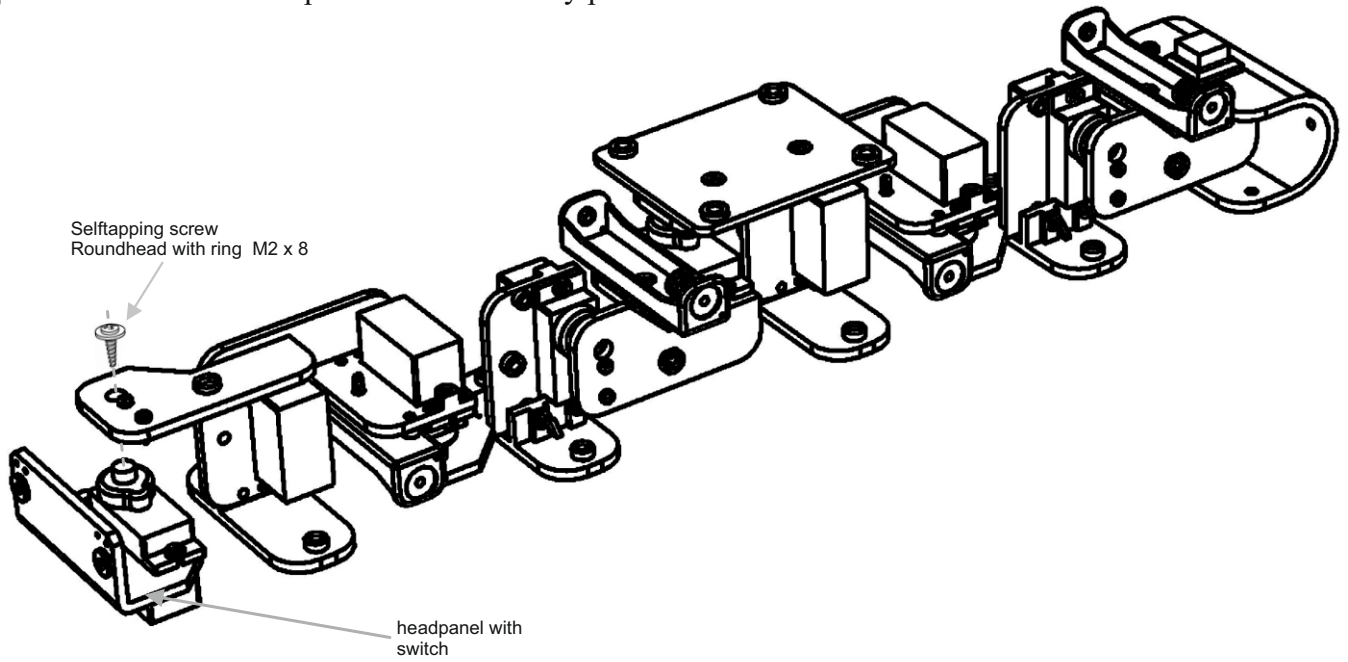
**IMPORTANT.**

Avoid short circuits and errors in this installation phase.

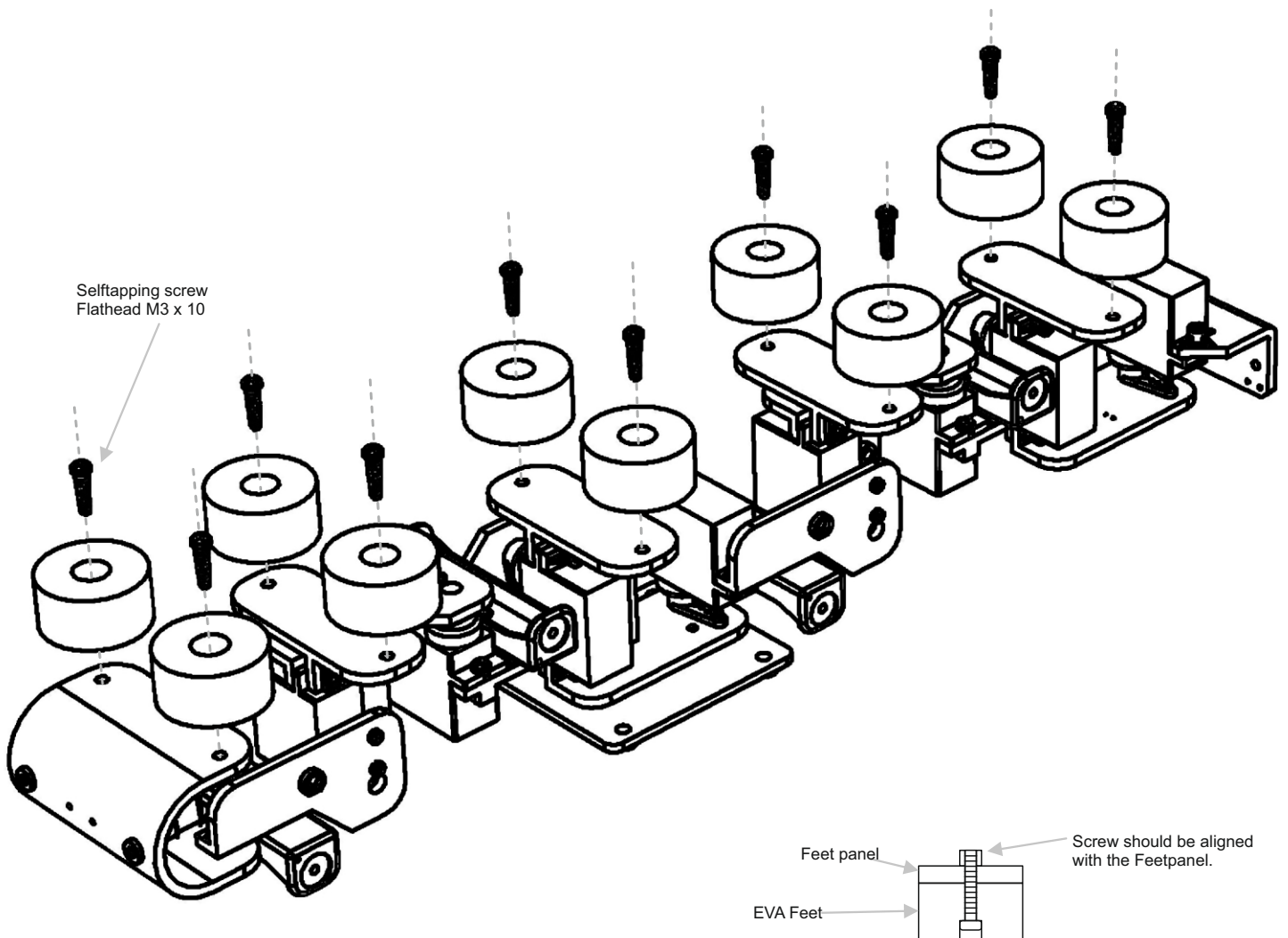
**Step 21a:** Attach Servomotor (1) to the head panel.



**Step 21b:** Attach the head plate to the first body panel.

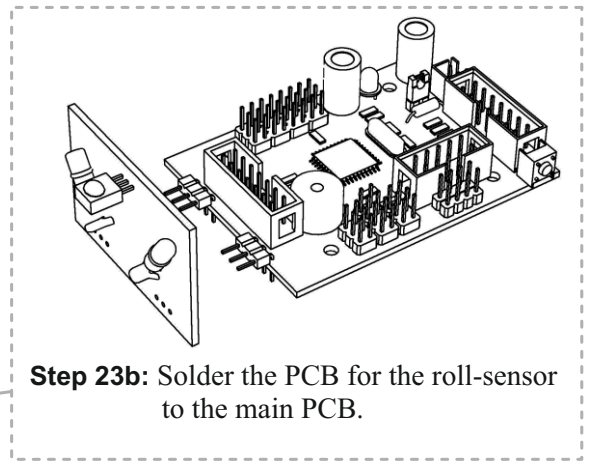
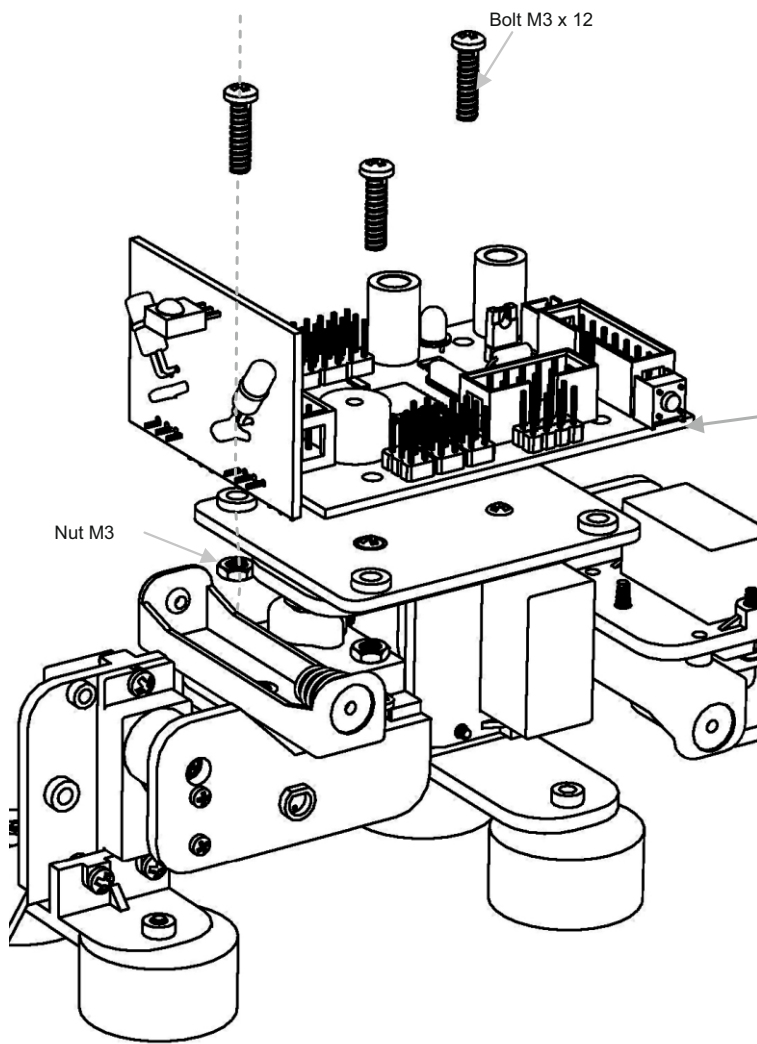


**Step 22:** Attach the 10 (Pieces) EVA pedestal feet to the feet panels.

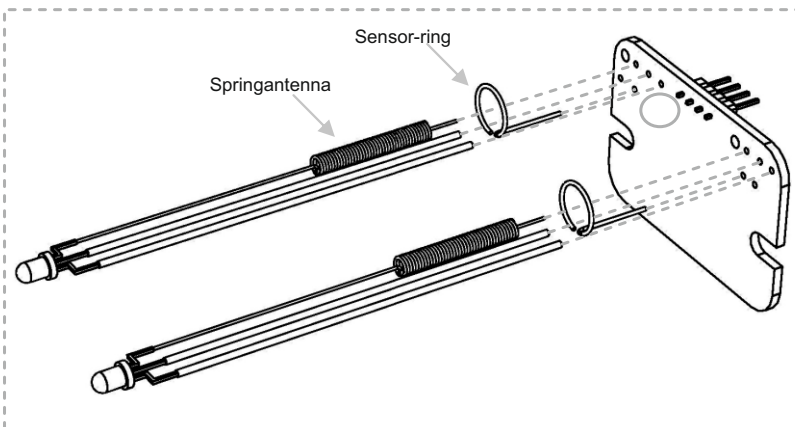


Connect the EVA Feet with M3 x 10 screws as drawn in the picture

**Step 23a:** Attach the main PCB to the PCB-mounting.



**Step 24:** Attach the antenna.



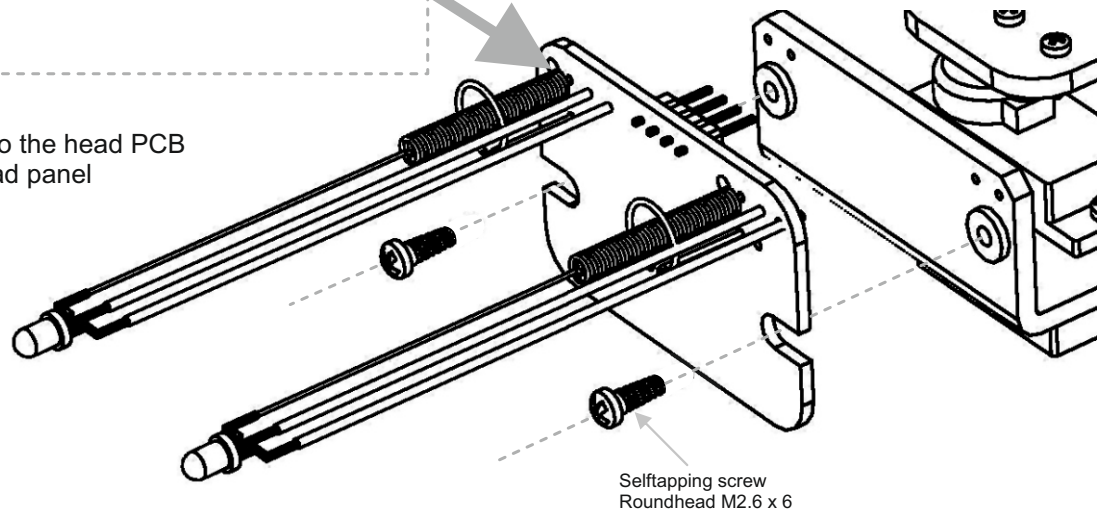
**IMPORTANT.**

Be careful: the spring may be very sharp.



Assembly in the following order; first solder the sensor-rings to the PCB and complete the antenna as illustrated. Adjust the the spring antenna as good as possible to the center of the sensor-rings without contacting the ring.

After soldering of the antenna to the head PCB attach the head PCB to the head panel

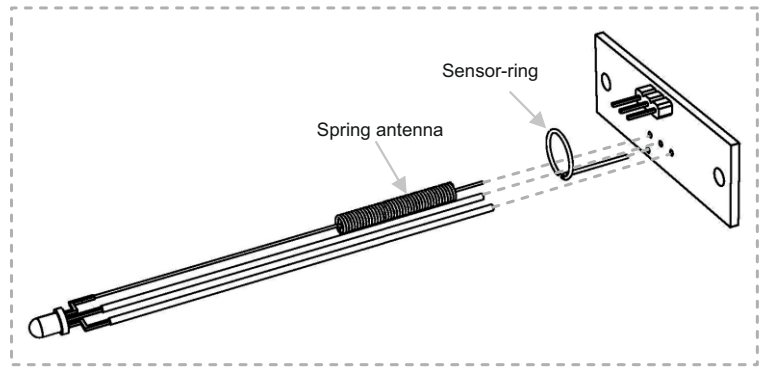
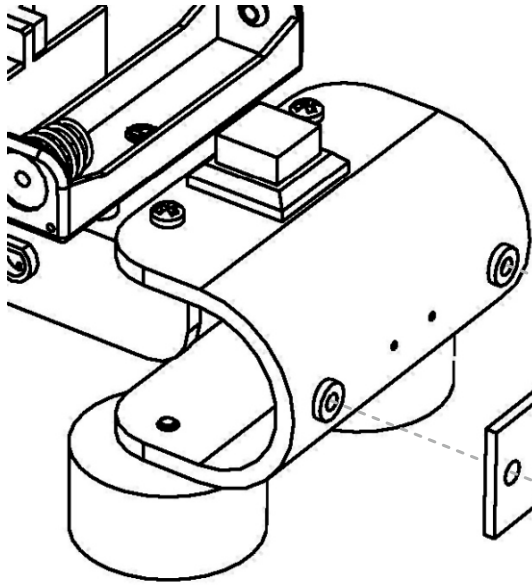




**Step 25:** Now attach the tail antenna.



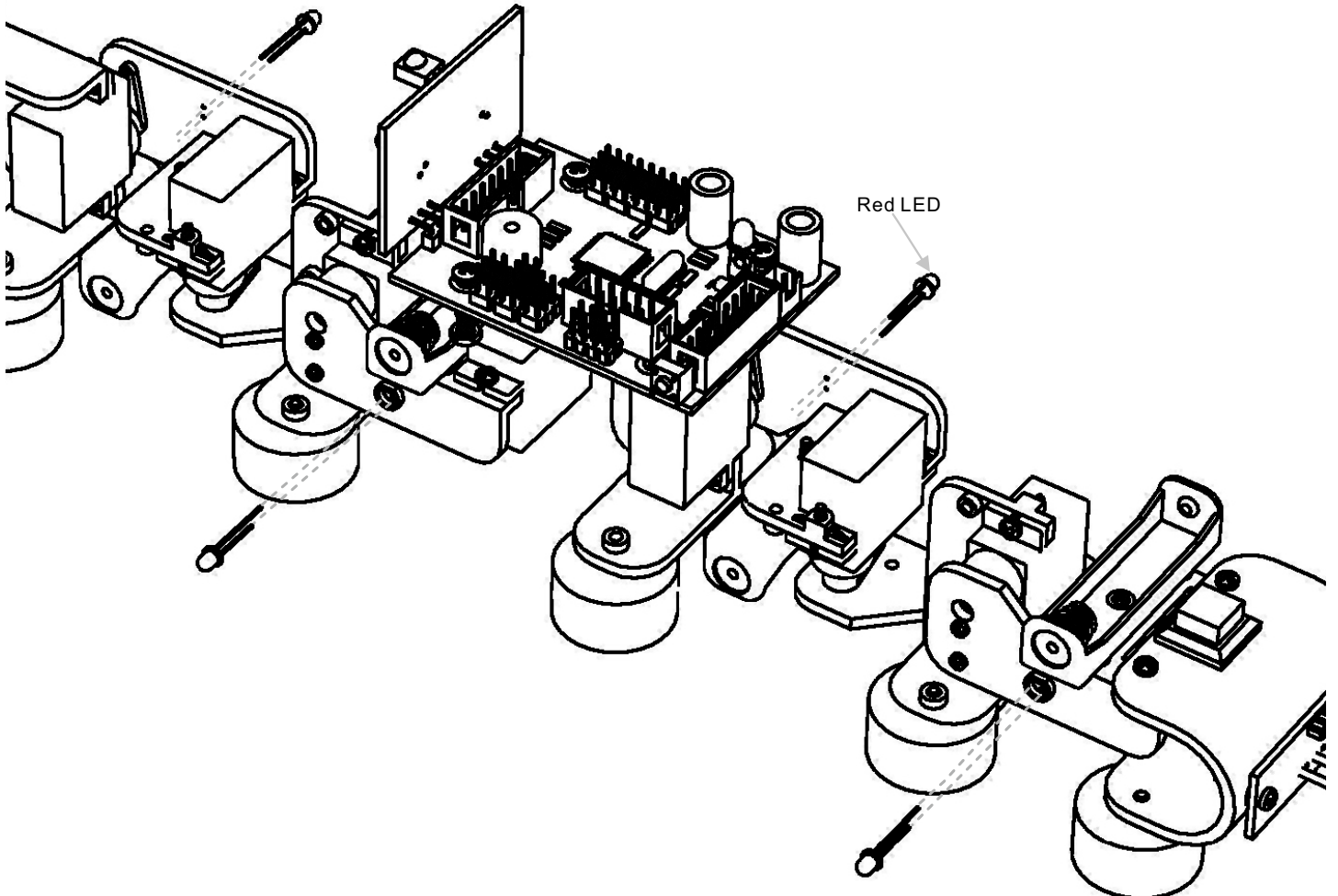
**IMPORTANT.**  
the spring may be very sharp.



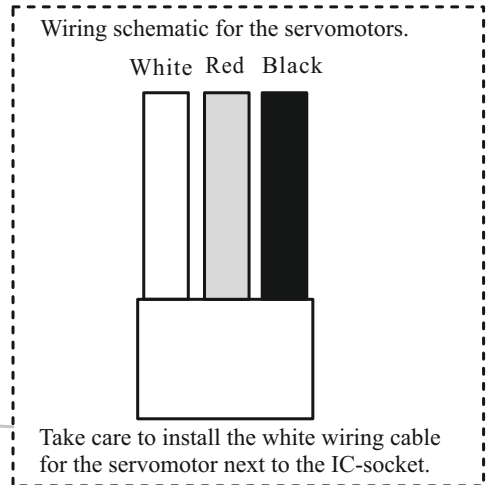
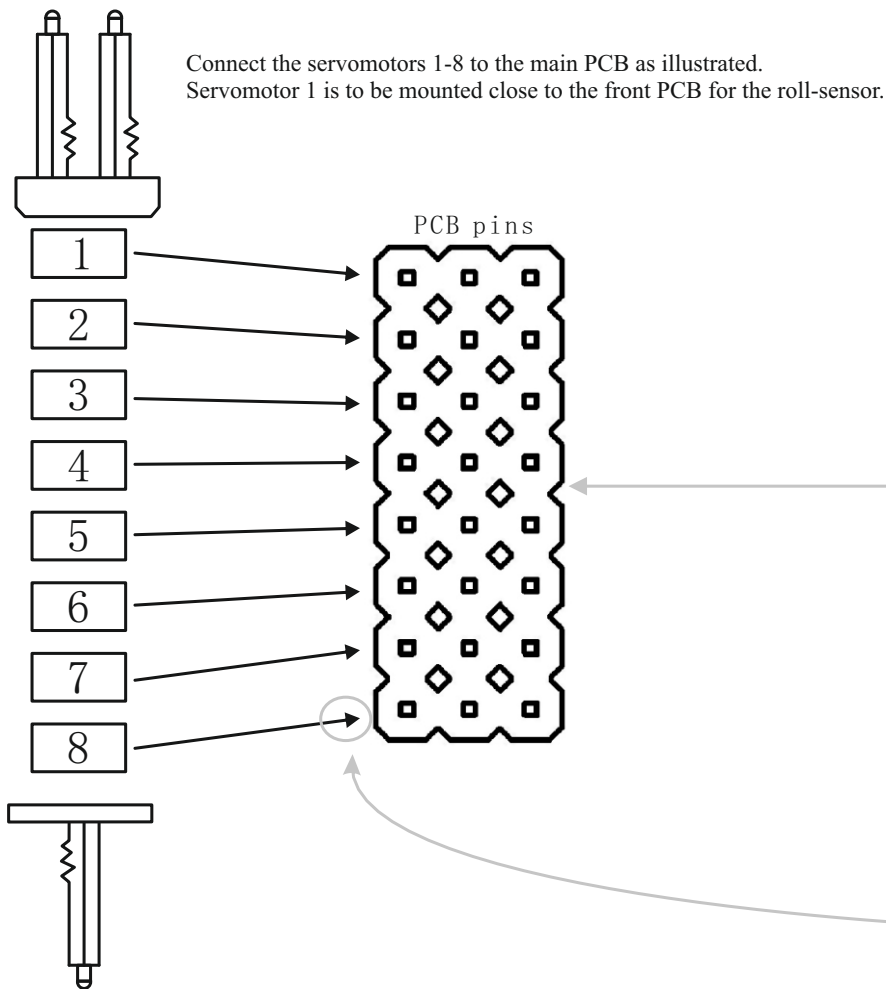
Assembly in the following order; first solder the sensor-rings to the PCB and complete the antenna as illustrated. Adjust the the spring antenna as good as possible to the center of the sensor-rings without contacting the ring.

Selftapping screw  
Roundhead M2.6 x 6

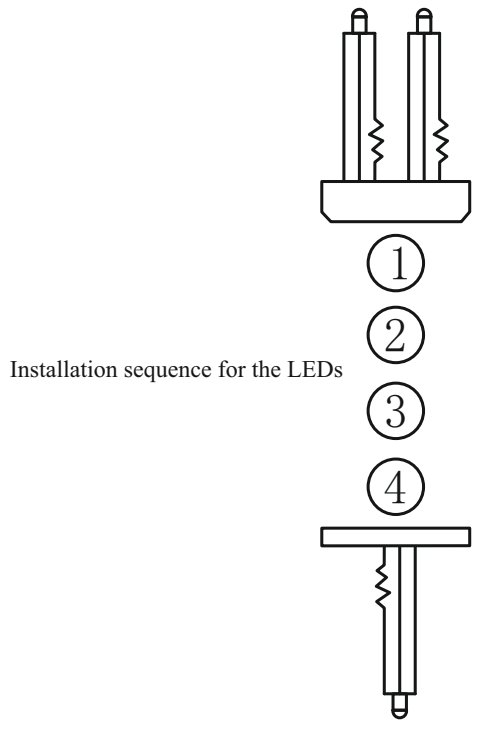
**Bauphase 26:** Attach the 4 pcs. LED



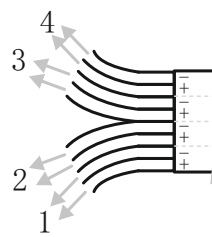
## Step 27: Connect the servomotors to the main PCB



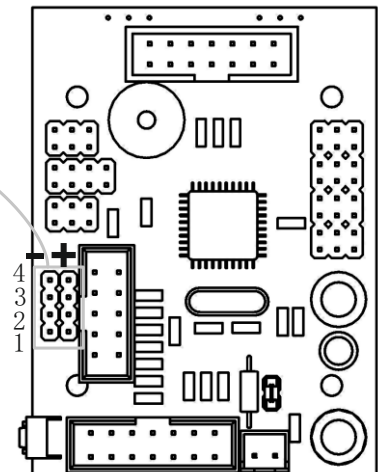
## Step 28: Connect the LEDs by 8-poled flat cable to the main PCB.



Installation sequence for the LEDs



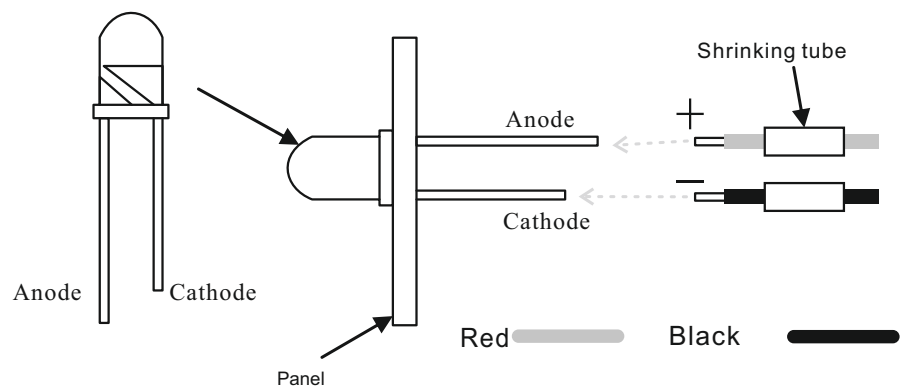
**Warning:**  
make sure the polarity is correct otherwise it will not function.



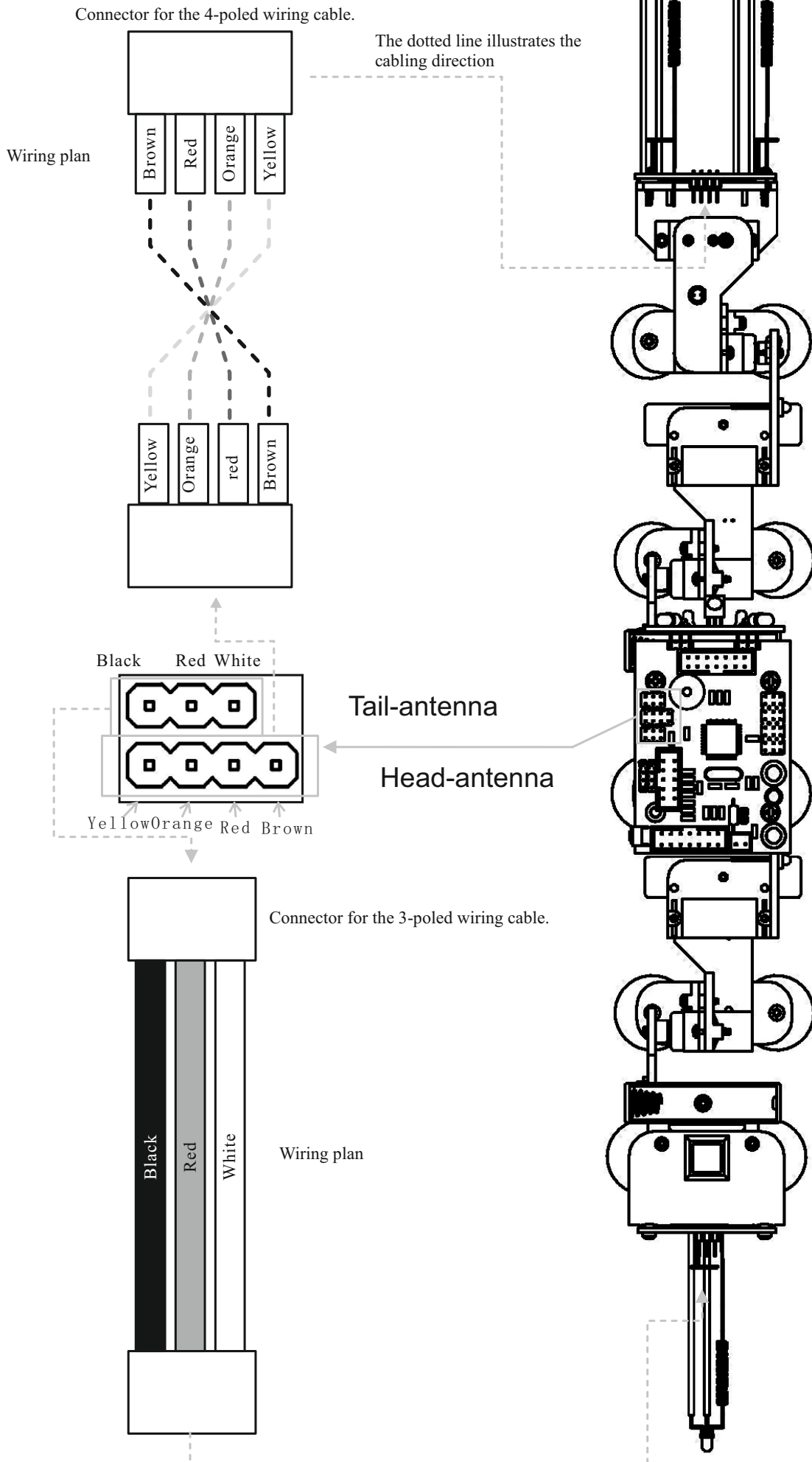
- Use a scissor to cut the shrink tube into pieces of 15 mm length.
- Remove the isolation at the wiring cables' ends.
- Before soldering move the shrink tube over the wiring cable.
- Solder the wiring cables' ends to the LEDs.  
Take care to solder the positive poled wire to the anode.
- Slightly heat the shrink tube with a cigarette lighter.



**Warning.**  
Do not heat the LED for more than 1 second.  
Otherwise the LED may be damaged.



**Step 29a:** Connect the head-plate and tail-plate to the main PCB.



## Step 29b: Test the battery voltage



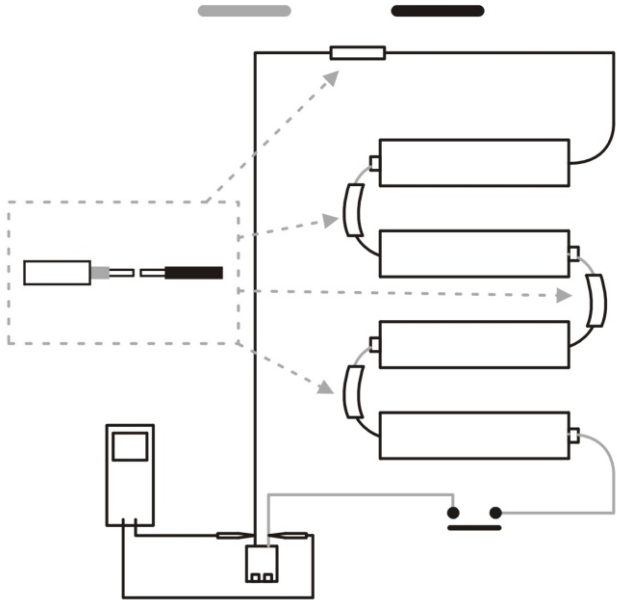
### NOTICE.

#### Voltage on Power connector

FIRST test the battery voltage at the connector before you connect it with the main PCB.

With 6 Volt (Battery voltage) **do not forget** to OPEN jumper J11 .

With 4.8 Volt (Battery voltage) **do not forget** to CLOSE jumper J11 .



## Step 29c: Test the wiring

### NOTICE.

First check all the wires on the main PCB.

**Do not forget to open jumper J11 when you are using normal 1.5 Volt batteries.**



### Important pages:

Page 23; Software Installation

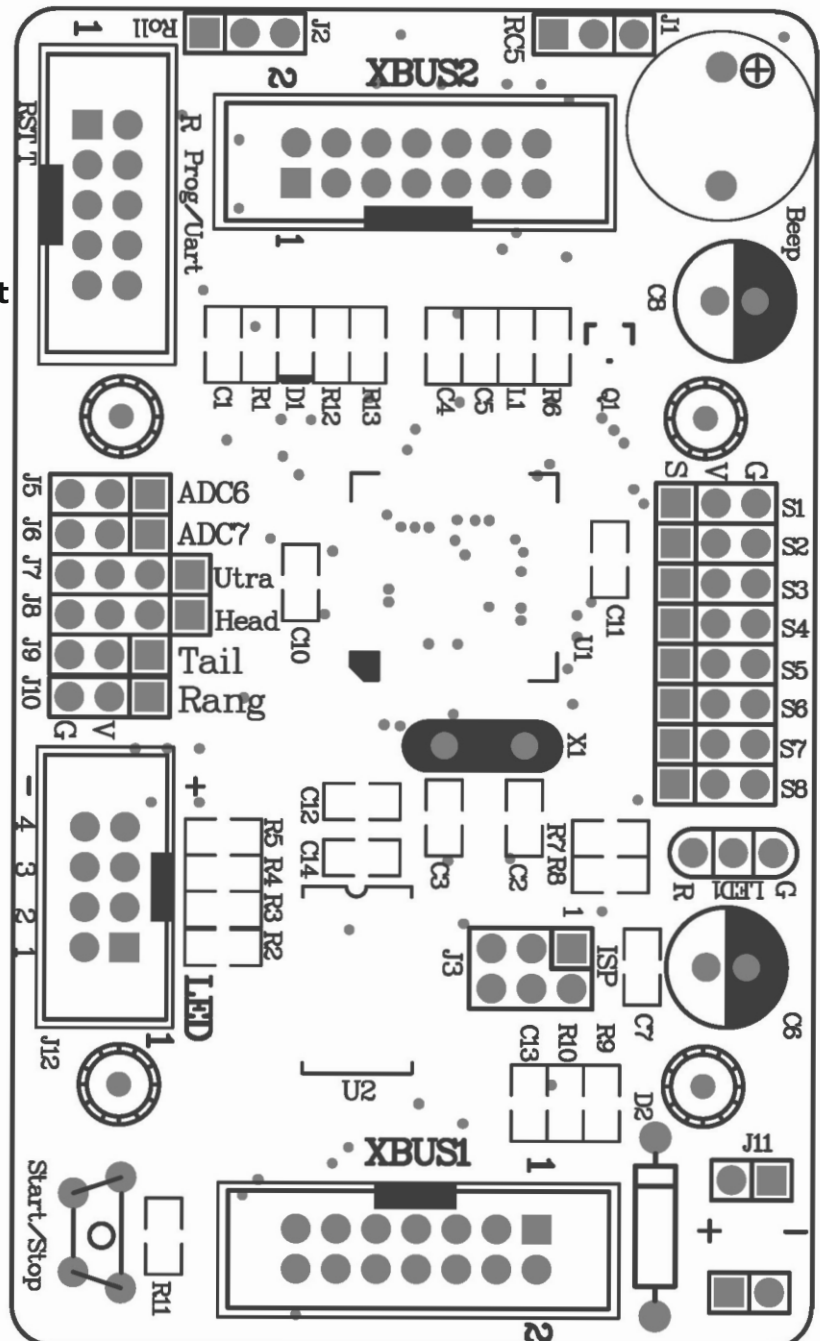
Seite 21; First Test

Page 35; Selftest

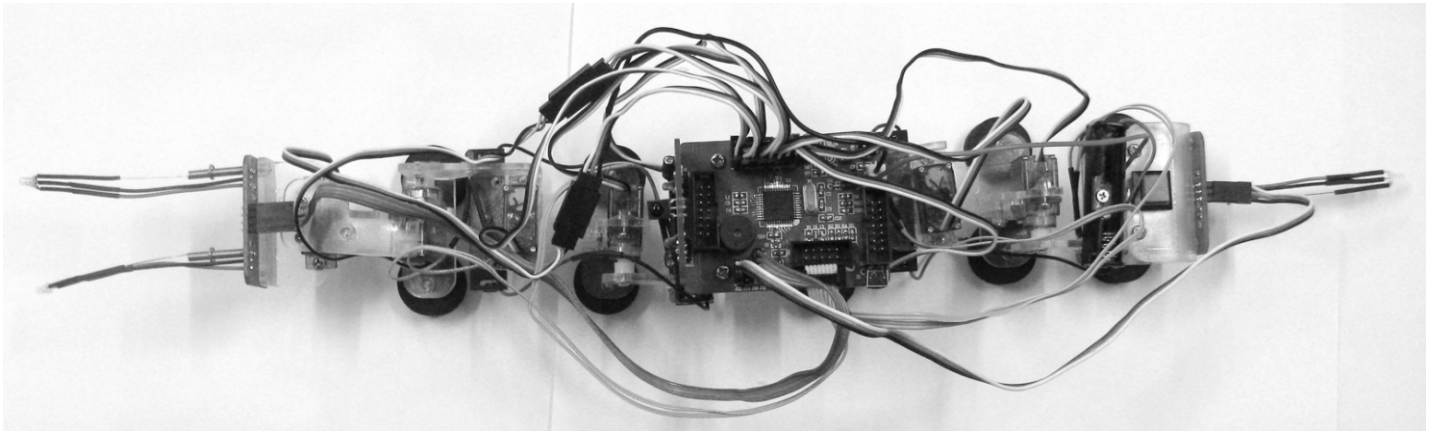
## IMPORTANT



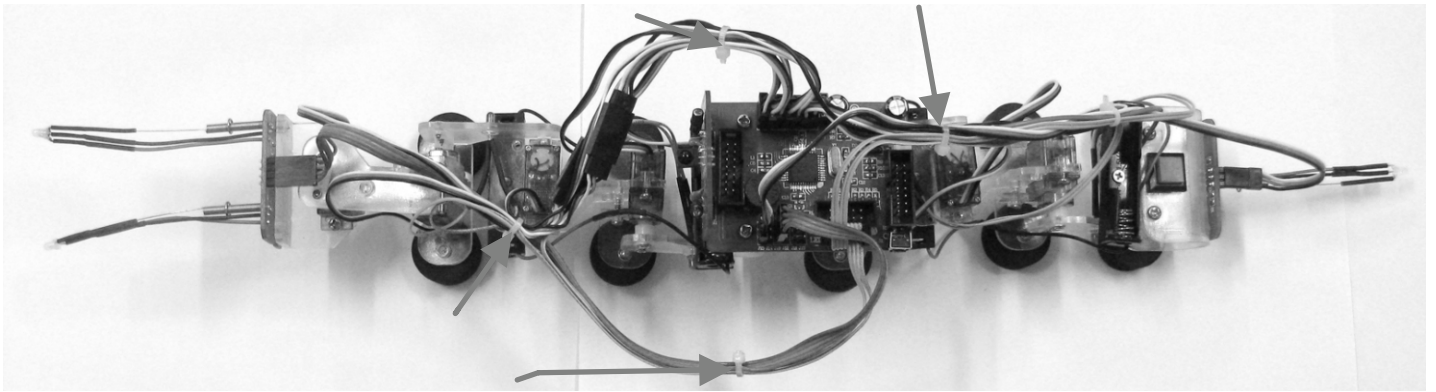
The old version of the RobotLoader can only measure 5.1 Volt maximum.



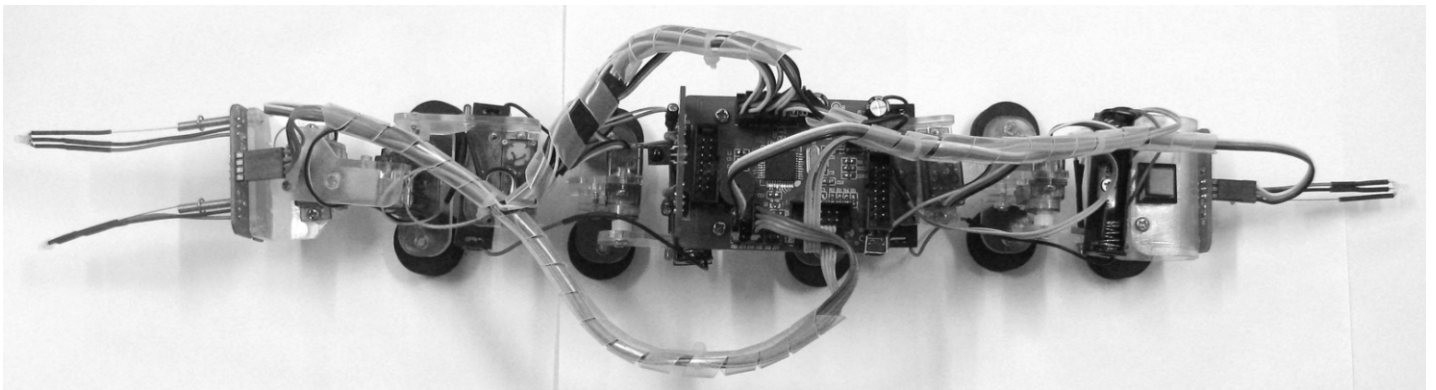
### Step 30: Final assembly



Having completed all wiring your Caterpillar will be looking as follows. We will have to rearrange some cabling, but take care to leave enough room for free movements of the robot's segments.



Join the wires as illustrated. Do not join the cabling to tight. The wires must have some room to move. Carefully move the Caterpillar's segments into all directions. Check to see that none of the wires will be stretched to tight. See that cables will be ordered evenly at both sides of the roll-sensor.



Use the spiral to neatly arrange the wiring cables. Be careful with LED-connectors and battery wiring. If the wiring is arranged too tight the cabling may be stressed by metal fatigue and cause wiring interrupts in a permanently moving section of the Caterpillar.

***Having completed the Caterpillar you may now program the robot. Please install the software from CD or download the actual version from: [www.globalspecialties.com](http://www.globalspecialties.com)***

# 4. Software Installation



Let's do the software installation now. A properly installed software is of paramount importance for all following chapters.

As you need administrator rights, you have to log into your system as an administrator. We recommend to read the whole chapter thoroughly first and then start with the installation step by step.

The user must have basic knowledge of Windows or Linux based computers and be familiar with current programs such as file managers, web browsers, text editors, file compression software (WinZip, WinRAR, unzip and others) and eventually Linux shell etc.. If your computer knowledge is very limited, you should learn more about systems before you start using the Caterpillar. This manual is not intended as an introduction to computers which would go beyond our scope. It is only aimed at the Caterpillar, its programming and the specific software required.

## Caterpillar CD-ROM

You have probably already inserted the CD-ROM into your computer drive - if not, please do it now. In Windows, the CD menu should appear shortly afterwards per autostart. If not, you can open the file "start.htm" with a web browser as e.g. Firefox or Chrome in the main directory of the CD through file manager.

If you have never installed an updated web browser, after the language selection you will find in the CD menu, in addition to this manual (that you can also download from our home page), information, data sheets and pictures, also the menu item "Software". It contains all software tools, USB drivers and example programs with source code for the Caterpillar.

Depending on the safety settings of your web browser, you can start the installation programs directly from the CD

## WinAVR - for Windows

We will start with the installation of WinAVR. WinAVR is - as the name says - **only available for Windows.**

**Linux users can skip to the next section.**

WinAVR (pronounce like the word "whenever") is a collection of many useful and necessary programs for the software development for AVR micro controllers in C language. In addition to the GCC for AVR (designated by the term "AVR-GCC", more details later) WinAVR includes the convenient source text editor "Programmers Notepad 2" that we will also use for the program development of the Caterpillar.

WinAVR is a private project that is not supported by a company. It is available for free in the internet. You will find updated versions and more information at:

<http://winavr.sourceforge.net/>

In the meantime the project gets the official support from ATMEL and the AVR-GCC is available for AVRStudio, the development environment for AVR's from ATMEL. However we will not describe it in this manual as Programmers Notepad is much better suited for our purpose. The WinAVR installation file is on the CD in the folder:

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

The installation of WinAVR is simple and self-explanatory. Normally you don't need to change any settings. So, just click on "Continue".

*If you use Windows Vista or Windows 7/8, you must install the latest version of WinAVR. It should also work perfectly with Windows 2K and XP. If not, you can try one of the older versions that are also on the CD (before you make a new installation of WinAVR, you have to uninstall the existing version first.). Officially Win x64 is not yet supported but the CD contains a patch for Win x64 systems if a problem arises. You will find more information on the software page of the CD menu.*

## **AVR-GCC, avr-libc und avr-binutils - for Linux**

*(Windows users can skip this section.)*

Linux might require more effort. Some distributions already contain the required packages but they are mostly obsolete versions. Therefore you need to compile and install newer versions. It is impossible to describe in detail the numerous Linux distributions as SuSE, Ubuntu, RedHat/Fedora, Debian, Gentoo, Slackware, Mandriva etc. that exist in many versions with their own particularities and we will keep here only to the general lines.

*The same applies to all other Linux sections in this chapter.* The procedure described here must not necessarily work for you. It is often helpful to search in the internet e.g. for "<LinuxDistribution> avr gcc" or similar. (Try different spellings). The same applies to all other Linux sections - of course with the suitable keywords. If you encounter problems with the installation of the AVR-GCC, you can also take a look in our robot network forum or in one of the numerous Linux forums. First of all, you have to uninstall already installed versions of the avr-gcc, the avr-binutils and the avr-libc because, as said, these are mostly obsolete. You can do that via the package manager of your distribution by searching for "avr" start up and uninstall the three above mentioned packages - as far as they exist in your computer. You can find out easily if the avr-gcc has already been installed or not via a console as e.g.

```
> which avr-gcc
```

If a path is displayed, a version is already installed. So just enter:

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

and look at the output. If the displayed version is smaller than 3.4.6, you have to uninstall in any case this obsolete version.

If the version number lies between 3.4.6 and 4.1.0, you can try to compile programs (see following chapter). If it fails, you have to install the new tools. We will install hereafter the currently most updated version 4.1.1 (status March 2007) together with some important patches.

If the packages above do not appear in the package manager although an avr-gcc has definitely been installed, you need to erase manually the relevant binary files- i.e.search in all /bin, /usr/bin etc. directories for files starting with "avr" and erase these (of course ONLY these files and nothing else.). Eventually existing directories as /usr/avr or /usr/local/ avr must also be erased.

**Important:** You have to make sure that the normal Linux development tools as GCC, make, binutils, libc, etc. are installed prior to compiling and installing. The best way to do so is via the package manager of your distribution. Every Linux distribution should be supplied with the required packages on the installation CD or updated packages are available in the internet.

Make sure that the "texinfo" program is installed. If not, please install the relevant package before you continue - otherwise it will not work.

Having done that, you can start with the installation itself.

**Now you have three options: either you do everything manually or you use a very simple to use installation script and as last you can use an already compiled .dep install package.**

**We recommend to try the installation script first. If this doesn't work, you can still install the compiler manually.**

Latest versions see; <http://www.wrightflyer.co.uk/avr-gcc/>

**Attention:**

You should have enough free disk space on your hard disk. Temporarily more than 400Mb are required. Over 300Mb can be erased after the installation but during the installation, you need all the space.

Many of the following installation steps require **ROOT RIGHTS**, so please log in with "su" as root or execute the critical commands with "sudo" or something similiar as you have to do it in Ubuntu e.g. (the installation script, mkdir in /usr/ local directories and make install require root rights).

**Please note in the following the EXACT spelling of all commands.**

Every sign is important and even if some commands look a bit strange, it is all correct and not a typing mistake. (<CD-ROM-drive> has of course to be replaced by the path of the CD-ROM drive.)

The folder on the CD:

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

contains all relevant installation files for the avr-gcc, avr-libc and binutils. First of all, you have to copy all installation files in a directory on your hard disk - **this applies for both installation methods**. We will use the Home directory (usual abbreviation for the current home directory is the tilde: "~"):

```
> mkdir ~/Robot Arm
> cd <CD-ROM drive>/Software/avr-gcc/Linux
> cp * ~/Robot Arm
```

After the successful installation you can erase the files to save space.



## Automatic Installation Script

Once you have made the script executable via `chmod`, you can start immediately:

```
> cd ~/Robot Arm
> chmod -x avrgcc_build_and_install.sh
> ./avrgcc_build_and_install.sh
```

Answer “y” to the question if you want to install with this configuration or not.

### PLEASE NOTE:

The compilation and installation will take some time depending on the computing power of your system (e.g about 15 min. on a 2GHz Core Duo Notebook. Slower systems will need longer).

The script will include also some patches. These are all the `.diff` files in the directory.

If the installation was successful, following message will be displayed:

```
(./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
```

### As suggested, you can execute

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

This erases all temporary files that you will not need anymore.

You can skip the next paragraph and set the path to the avr tools.

If the execution of the script failed, you have to look attentively to the error message (scroll the console up if necessary). In most cases it is just a matter of missing programs that should have been installed earlier (as e.g. the before mentioned `texinfo` file). Before you continue after an error, it is recommended to erase the already generated files in the standard installation directory “`/usr/local/avr`” – preferably the whole directory.

If you don't know exactly what has gone wrong, please save all command line outputs in a file and contact the technical support. Please join always as much information as possible. This makes it easier to help you.

## GCC for AVR

The GCC is patched, compiled and installed a bit like the binutils:

```
> cd ~/Robot Arm> 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
```

After the \ just press Enter and continue to write. This way the command can be spread over several lines, but you can also just drop it.

## AVR Libc

And last but not least the AVR libc

```
> cd ~/Robot Arm
> 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
```

Important: at `--build=`./config.guess`` make sure to put a backtick ` (à <-- the grave accent on the a. ) and not a normal apostrophy or quotation marks as this wouldn't work.

## Set the Path

You must make sure now that the directory `/usr/local/avr/bin` is registered in the path variable otherwise it will be impossible to retrieve the `avr-gcc` from the console or from the makefiles. To that end, you have to enter the path in the file `/etc/profile` or `/etc/environment` or similiar (varies from one distribution to another) – separated by a colon `“:”` from the other already existing entries. It could look in the file like:

*`PATH="/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/usr/local/avr/bin"` Now enter in a console `“avr-gcc --version”` as described above. If it works, the installation was successfull.*

## NEWER VERSIONS

*There are regular newer versions on internet, please always check if you can find a newer version as we put on the CD now.*

## Manual Installation

If you prefer to install the compiler manually, or the installation via the script failed, you can follow the instructions below.

The description is based on following article:

[http://www.nongnu.org/avr-libc/user-manual/install\\_tools.html](http://www.nongnu.org/avr-libc/user-manual/install_tools.html)

that is also included on the CD in PDF format in the AVR Libc documentation:

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

Our description here is much shorter but includes a few important patches. Without these, some things will not work properly.

First of all we have to create a directory in which we will install all tools. That should be `/usr/local/avr`. Also enter in a console AS A ROOT:

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

It must not necessarily be this directory. We just create the variable `$PREFIX` for this directory:

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

This must be added into the `PATH` variable:

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

## NEWER VERSIONS

*There are regular newer versions on internet, please always check if you can find a newer version as we put on the CD now.*

## Binutils for AVR

Now you must unpack the sourcecode of the binutils and add a few patches. We suppose in our example that you have copied everything into the home directory `~/Caterpillar`:

```
> cd ~/Caterpillar
> 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 detects what is available in your system and generates suitable makefiles. Now the binutils can be compiled and installed:

```
> make
> make install
```

Depending on the computing power of your system, this can take a few minutes. That applies also to the next two sections, especially to the GCC.

## 4.1. Java 6

The RobotLoader (see Info below) has been developed for the Java platform and is suitable for Windows and Linux (theoretically also for operating systems like OS X but Global Specialties is unfortunately not yet in a position to give official support). To make it work, you need to install an updated Java Runtime Environment (JRE). It is often already installed on the computer but it must be at least version 1.6 (= Java 6). If you have no JRE or JDK installed, you must install the supplied JRE 1.6 from SUN Microsystems or alternatively download a newer version from <http://www.java.com> or <http://java.sun.com>.

### **Windows**

The JRE 1.6 for Windows is in following folder:

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

Under Windows the installation of Java is very simple. You just have to start the setup and follow the instructions on the screen - that's it. You can skip the next paragraph.

### **Linux**

Under Linux the installation doesn't present any major problems although some distributions require some manual work. In the folder:

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

you will find the JRE1.6 as an RPM (SuSE, RedHat etc.) and as a self-extracting archive ".bin". Under Linux it is advisable to look for Java packages in the package manager of your distribution (keywords e.g. „java“, „sun“, „jre“, „java6“ ...) and use the packages of your distribution rather than those on the CD-ROM. However make sure to install Java 6 (=1.6) or a newer version but definitely not an older one.

Under Ubuntu or Debian, the RPM archive doesn't work directly. You will have to use the package manager of your distribution to find a suitable installation package. The RPM should however work well with many other distributions like RedHat/Fedora and SuSE. If not, you always have the solution to unpack the JRE (e.g. to /usr/lib/Java6) from the self-extracting archive (.bin) and set manually the paths to the JRE (PATH and JAVA\_HOME etc.).

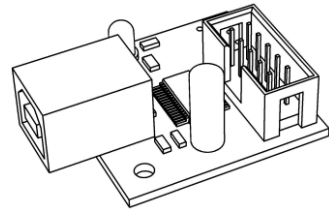
Please refer to the installation instructions from Sun that you will find also in the above mentioned directory and on the Java website (see above).

You can check if Java has been correctly installed by entering the command "java-version" in a console. The output should be approximately as follows:

```
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 output is totally different, you have either installed the wrong version or there is another Java VM installed in your system.

## 4.2. Connection of the USB interface – Windows



**Linux users can skip to the next section.**

There are several options to install the USB interface, the easiest being the installation of the driver BEFORE the first connection of the hardware.

The CD contains an installation program for the driver.

For 32 and 64 Bit Windows 7, 8, XP, Vista, Server 2003 and 2000 systems:

```
<CD-ROM drive>:\Software\USB_DRIVER\Win2k_XP\CDM_Setup.exe
```

For old Win98SE/Me systems, such a handy program does unfortunately not exist. You need to install an older driver manually after connecting the equipment (see below).

Just execute the installation program. There will just be a short note that the driver has been installed and that's all.

Now you can connect the USB interface to the PC. PLEASE DO NOT CONNECT TO THE ROBOT YET. Just connect to the PC via the USB lead. Please touch the PCB of the USB interface only at the edges or at the USB plug or at the plastic shell of the programming plug (see safety instructions on static discharges). Please avoid touching any of the components on the PCB, soldering points or contacts of the IDE connector unless absolutely necessary in order to prevent static discharges.

The previously installed driver will be used automatically for the device without any help from your side. Under Windows XP/2k small speech bubbles appear at the bottom above the task bar. The last message should be "The device has been successfully installed and is ready for use."

If you have connected the USB interface before the installation (or use Win98/Me) – it doesn't matter so much. Windows will ask you for a driver. This installation method is also possible. The driver is also in unpacked format on the CD.

If you are in this situation, a dialogue appears (under Windows) to install the new driver. You have to indicate the path to the system where it can find the driver. Under Windows 2k/XP you need to select first the manual installation and not to look for a web service. On our CD the driver is in the above mentioned directories. So, just indicate the directory for your Windows version and eventually a few other files that the system doesn't find automatically (they are all in the directories mentioned below.) ...

Under Windows XP and later versions there is often a message that the FTDI drivers are not signed/verified by Microsoft (normally not here as the FTDI drivers are signed). This is irrelevant and can be confirmed without any problem.

### Operation

For 32 and 64 Bit Windows 7, 8, XP, Vista, Server 2003 and 2000 systems:

```
<CD-ROM drive>:\Software\USB_DRIVER\Win2k_XP\FTDI_CDM2\
```

For older Windows 98SE/Me systems:

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

After the installation of the driver a re-start of the computer may be necessary with older versions like Win98SE. PLEASE NOTE: Under Win98/Me only one of both drivers is working: Either Virtual Comport or the D2XX driver from FTDI. Unfortunately there is no driver that offers both functions. Normally there is no virtual comport available as the RobotLoader under Windows uses as a standard the D2XX drivers (you can change this - please contact our support team.).

## Check the Connection of the Device

To check if the device has been correctly installed you can use the device manager as an alternative to the RobotLoader under Windows XP, 7, 8, or 2003 and 2000: Right click on My Computer --> Properties --> Hardware --> Device manager

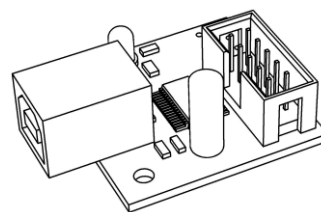
OR alternatively: Start --> Settings --> Control panel --> Performance and Maintenance --> System --> Hardware --> Device manager and check there in the tree view under "Connections (COM and LPT)" if you find a "USB-Serial Port (COMX)" - the X replacing the port number, or look under "USB serial bus controller" for a "USB Serial Converter" .

If you wish to uninstall the driver some day

If ever you wish to uninstall the driver (no, not now - this is just a hint if you need this some day): If you have used the CD ROM installation program, you can uninstall it directly via Start --> Settings --> Control panel --> Software. In the displayed list you will find an item "FTDI USB Serial Converter Drivers" – select it and click on "uninstall".

If you have installed the driver manually, you can execute the program "FTUNIN.exe" in the directory dedicated to the USB driver for your system. Warning: USB-->RS232 adaptors with FTDI chip set often also use this driver.

## 4.3. Connection of the USB programmer – Linux



**Windows users can skip this section.**

Linux systems with kernel 2.4.20 or higher already include the required driver (at least for the compatible previous model FT232BM of the chip on our USB interface, the FT232R). The hardware is automatically recognized and you have nothing else to do. In case of a problem, you can get Linux drivers (and support and maybe also newer drivers) directly from FTDI:

<http://www.ftdichip.com/>

Once the hardware has been connected, you can check under Linux via:

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

if the USB serial port has been correctly installed. This is normally all you have to do.

It is worth to mention that the RobotLoader uses under Windows D2XX drivers and the full USB designations appear in the port list (e.g. "USB0 Robot USB Interface | serialNumber"). Whereas under Linux the virtual com port designations appear such as /dev/ttyUSB0, /dev/ttyUSB1 etc.. The normal com ports are equally displayed as "dev/ttyS0" etc.. In this case you have to try which port is the correct one.

Unfortunately Linux doesn't have such a convenient driver that does both. Therefore it made more sense to use the Virtual Comport drivers that are included in the kernel anyway. The installation of a D2XX driver would require quite a lot of manual work....

## Finalization of Software Installation

Now the installation of the software and the USB interfaces is completed. You just need to copy the most important files from the CD on a hard disk (especially the complete "Documentation" folder and, if it hasn't been done yet, the example programs). This avoids to look constantly for the CD if you need these files. The folders on the CD are all named in such a way that they can be easily allocated to the relevant software packages or documentation. If you "lose" the CD one day, you can download the most important files as this manual, the RobotLoader and the example programs from the Global Specialties home page. You will find there also the links to the other software packages that you require.

## 4.4 FIRST Hardware Test.

After completion of the Caterpillar and an initial switch-on you may observe the robot is not aligned in a straight line.



### **NOTICE.**

**Check the voltage on Power connector**

**With 6 Volt (Battery voltage) do not forget to OPEN jumper J11 .**

**With 4.8 Volt (Battery voltage) do not forget to CLOSE jumper J11 .**

Now you can connect the power to the Mainboard and switch on the Caterpillar. The head LEDs and tail LED should be green now. Normally at start the servos often make some movements.

The power LED (Led 1) should blink for about 20 seconds and then go out. When the voltage is too low the power LED will blink red/orange and also the 4 red LEDs on the Caterpillar body. After completion of the Caterpillar and an initial switch-on you may observe the robot is not aligned in a straight line, please continue with the software chapters now and start selftest and calibration.

Alignment will be corrected at the program's start by adjusting the central position of the servomotors. A value 150 corresponds to the theoretically correct central position, but the number may vary between 130 and 170. The required offset value depends on the individual servomotor and the assembly's accuracy.

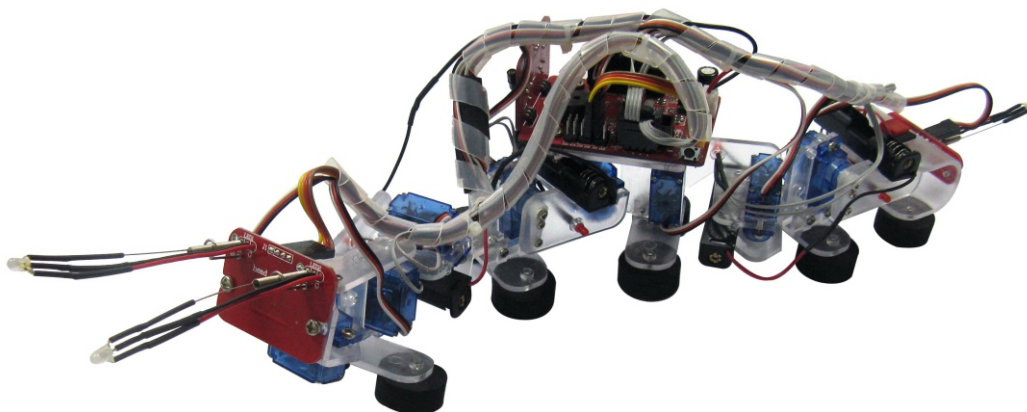
With the proper software the Caterpillar now is expected to move like a caterpillar on the floor.

As soon as an antenna contacts an obstacle the Caterpillar should retreat from the object. If the tail antenna contacts an object in a retraction phase the robot should rotate in order to inspect the situation.

If Caterpillar ends up in top-down position the robot will come to its feet again. If you use a universal remote control with a suitable "Sony TV"-Code you may also control Caterpillar from a moderate distance.

Now it is up to you how much experimenting you are going to invest into the control program. You may use the existing movement functions or create new functions and programs. For experiments you may add "SHARPGP2D12" which enables Caterpillar to detect objects before contacting obstacles. Installing a suitable transceiver (transmitter/receiver) at the programming socket enables a serial communication between PC and the moving robot.

Three analog inputs are available to connect additional sensors (including the distance sensor).

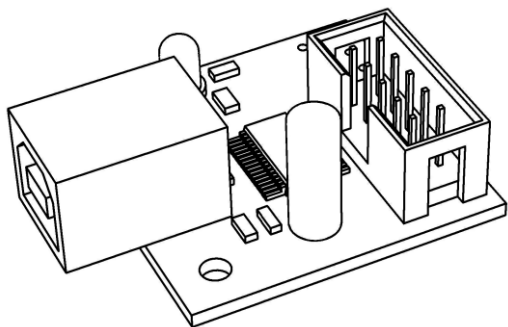


## 5. Programmer and RobotLoader

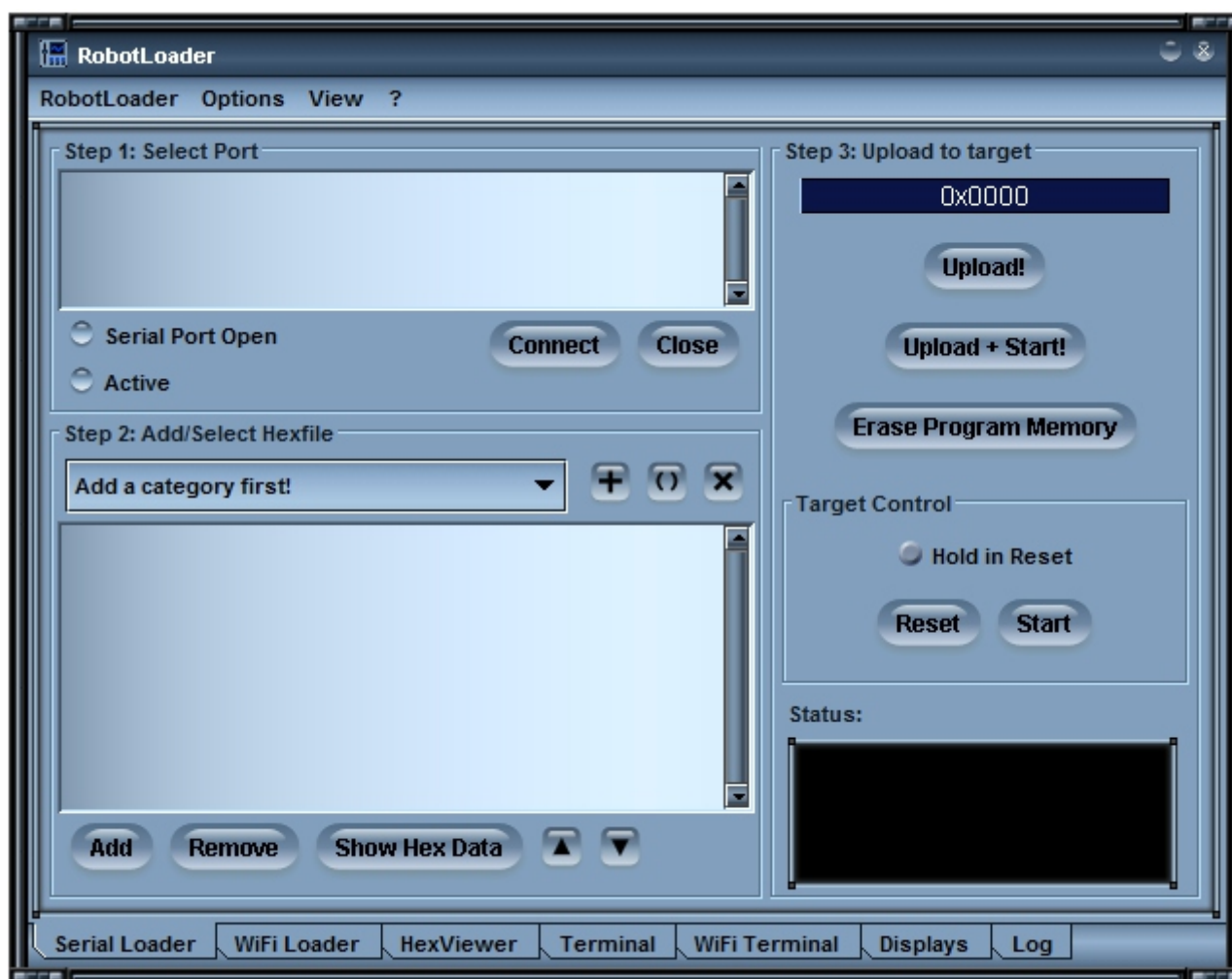
To load a HEX Caterpillar program from the PC into the Caterpillar Robot, we will use the USB programming adaptor and our RobotLoader software.

The external USB port programmer transmitter/receiver (transceiver), included in the package, must be connected on one side to a USB port of the computer and on the other side to the Prog/UART port of the Caterpillar PCB with a flat cable.

The program upload into the Caterpillar Robot erases automatically the previously existing program.



**USB Programmer**



**RobotLoader software**

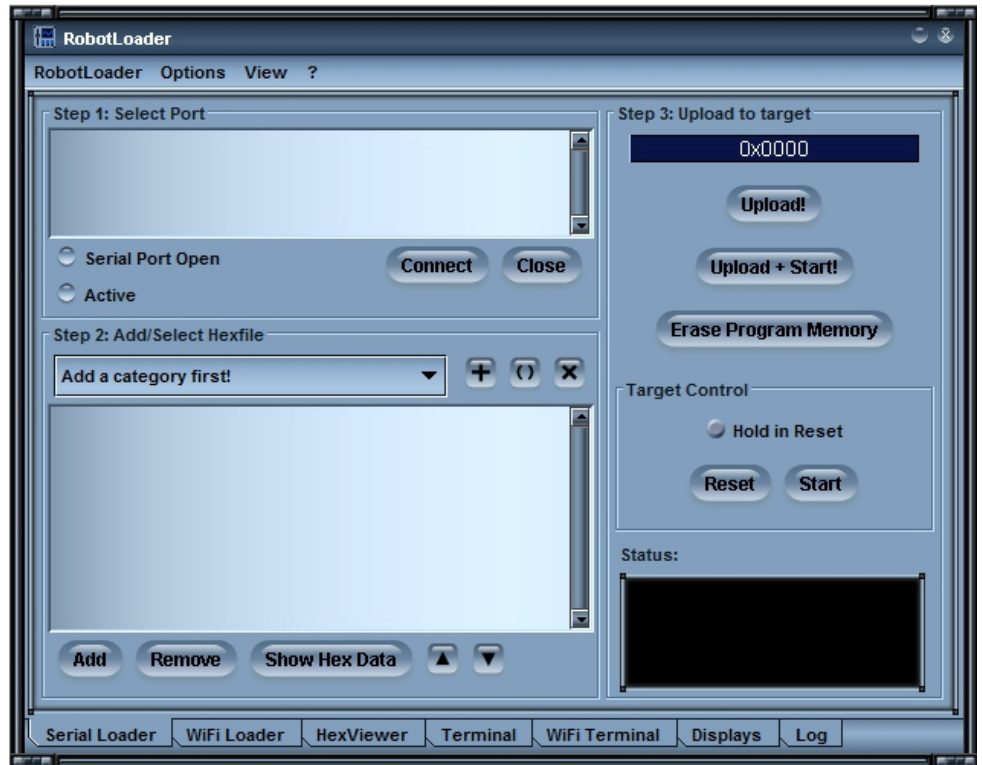
The RobotLoader has been developed to upload easily new programs into the Caterpillar and into several other Global Specialties robots (provided that they contain a compatible bootloader).

There are a few very useful additional functions integrated like a simple terminal program. The Robot Loader can generate a warning in the status display when the voltage is too low.

**The RobotLoader itself does not need any installation – just copy it somewhere in a file or put it on your hard drive.**



# RobotLoader



The RobotLoader has been developed to load easily new programs and all extension modules into the Caterpillar (as long as the modules are fitted with a compatible bootloader). Moreover it contains a few useful extra functions as e.g. a simple terminal program.

It is not necessary to install the RobotLoader. Just copy the program somewhere in a new folder on the hard disk.

<CD-ROM drive>:\Software\RobotLoader\RobotLoader.zip

Unpack the program somewhere on your hard disk e.g. in a new folder C:\Programme\RobotLoader (or similar). This folder contains the RobotLoader.exe file that you can start with a double-click. The RobotLoader program itself is in the Java archive (JAR) RobotLoader\_lib.jar. Alternatively you can start this via the command line:

## Under Windows:

```
java -Djava.library.path=".lib" -jar RobotLoader_lib.jar
```

## Linux:

```
java -Djava.library.path=".lib" -jar RobotLoader_lib.jar
```

The long -D option is necessary to enable the JVM to find all used libraries. Windows doesn't require this and you can just start with the .exe file. Linux requires the shell script "RobotLoader.sh". It might be necessary to make the script executable (chmod -x ./RobotLoader.sh). After that you can start it in a console with "./RobotLoader.sh".

It is advisable to create a shortcut on the desktop or in the start menu to make the start of the RobotLoader more convenient. Under Windows make a right click on the RobotLoader file.exe and then click on "Desktop (create shortcut)" in the "Send to" menu.

## Caterpillar Library, Caterpillar CONTROL Library and Example Programs

The Caterpillar Library and the related example programs are in a zip archive on the CD:

<CD-ROM drive>:\Software\Caterpillar Examples\CaterpillarExamples.zip

Just unpack them directly into a directory at your convenience on the hard disk. It is recommended to unpack the example programs into a folder on a data partition. Or in the "My files" folder in a sub-folder "Caterpillar\Examples" or else under Linux into the Home directory. It's entirely up to you.

The individual example programs will be discussed later in the software chapter.

## 5.1. Testing the USB Interface and starting the RobotLoader

The next step is a test of the program upload via the USB interface. Connect the USB interface to the PC (always connect the PC first.) and the other end of the 10-pin ribbon cable to the “PROG/UART” connector on the Caterpillar. (CATERPILLAR MUST BE SWITCHED OFF.) The 10-pin ribbon cable is mechanically protected against polarity inversion. As long as it is not forced, it can't be connected the wrong way round.



Then start the RobotLoader.

Depending on which language you have selected, the menus might have a bit different names. The screen shots show the English version. Via the menu item “Options- >Preferences” you can select under “Language /Sprache” the required language (English or German) and then click on OK.

Once you have selected your language, you have to re-start the Robot Loader to validate the changes.



### Open a port - Windows

Select the USB port. As long as no other USB->Serial Adaptor with FTDI controller is connected to the PC, you will see only one single entry that you have to select.

If more ports exist, you can identify the port via the name “Robot USB Interface” (or „FT232R USB UART“). Behind the port name the programmed serial number is displayed.

If no ports are displayed, you can refresh the port list via the menu item “RobotLoader-->Refresh Portlist” .

Now you can click on the button “Connect”. The RobotLoader will open the port and test if the communication with the bootloader on the robot is working. The black field “Status” on the bottom should show the message “Connected to: Caterpillar ...” or similar together with an information about the currently measured voltage. If not, just try again. If it still doesn't work, there is a mistake. Switch the robot off immediately and start searching for the error.

If the voltage is too low, a warning is displayed.

### Open a port – Linux

Linux handles the USB serial adaptor like a normal comport. The installation of the D2XX driver from FTDI would not be as simple as that under Linux and the normal virtual comport (VCP) drivers are included anyway in the current Linux kernels. It works almost the same as under Windows. You just need to find out the name of the Caterpillar USB interface and make sure that the USB port is not unplugged from the PC as long as the connection is open (otherwise you might have to re-start the RobotLoader to re-connect). Under Linux the names of the virtual comports are “/dev/ttyUSBx”, x being a number e.g. “/dev/ttyUSB0” or “/dev/ttyUSB1”. The names of the normal comports under Linux are “/dev/ttyS0”, „/dev/tty- S1” etc.. They also show up in the port list as far as they exist.

The RobotLoader remembers - if there are several ports - which port you have used last time and selects this port automatically when you start the program (in general, most of the settings and selections are maintained). Now you can click on the button “Connect”. The RobotLoader will open the port and test if the communication with the bootloader on the robot is working. The black field “Status” on the bottom should show the message “Connected to: CATERPILLAR ...” or similar together with an information about the currently measured voltage. If not, just try again. If it still doesn't work, there is a mistake. Switch the robot off immediately and start searching for the error.

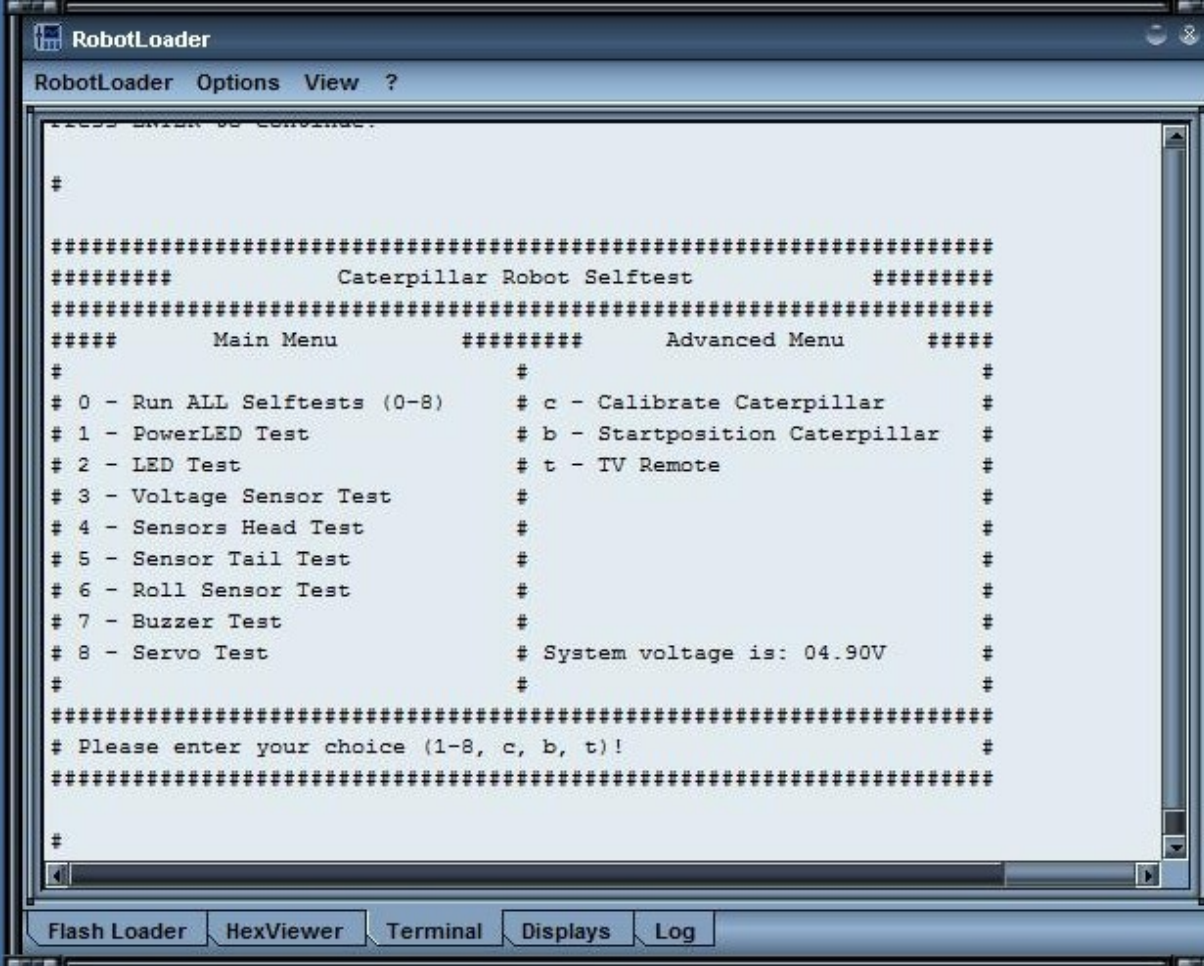
If the voltage is too low, a warning is displayed.

## 5.2. SELFTEST

The voltage LEDs lights up when the Caterpillar is switched on. The status LED Blinks when a HEX file is uploaded. If this worked, you can execute a small selftest program to test the functioning of all robot systems.

Please click on the button "Add" on the bottom of the Robot Loader window and select the file CaterpillarExamples, „Example\_11\_Selftest\Caterpillar\_Selftest.hex“ in the example directory. This file contains the selftest program in hexadecimal format - that's why this kind of program file is called "hex file". The file just selected appears afterwards in the list. This way you can add other hex files from your own programs and from the examples programs (see screen shot where some hex files have already been added).

Please select the "Caterpillar\_Selftest.hex" file in the list and click on the "Upload." button on the top right just below the progress bar. The program will now be transferred into the MEGA16 processor on the Caterpillar. This should not take more than a few seconds (max. 5 seconds for the selftest program). Switch to the tab (at the bottom of the window.) "Terminal". Alternatively you can also switch to terminal via the menu item "View".



```
RobotLoader
RobotLoader Options View ?
#####
#####          Caterpillar Robot Selftest          #####
#####
#####          Main Menu          #####          Advanced Menu          #####
#
# 0 - Run ALL Selftests (0-8)      # c - Calibrate Caterpillar      #
# 1 - PowerLED Test                # b - Startposition Caterpillar  #
# 2 - LED Test                      # t - TV Remote                  #
# 3 - Voltage Sensor Test          #                                #
# 4 - Sensors Head Test            #                                #
# 5 - Sensor Tail Test             #                                #
# 6 - Roll Sensor Test              #                                #
# 7 - Buzzer Test                  #                                #
# 8 - Servo Test                   # System voltage is: 04.90V     #
#                                  #                                #
#####
# Please enter your choice (1-8, c, b, t)!
#####
#
```

Now you can execute the selftest and the calibration of the Caterpillar. Press the switch Start/Stop Reset on the Caterpillar to start the program. Later you can do this alternatively via the RobotLoader menu --> Start or the key combination [CTRL]+[S]. However this time you can test if the switch works properly.

*It is recommended to start with the calibration.*

**If an error occurs in the selftest, switch the robot off immediately and start searching for the mistake.**

The Robot Loader is able to manage several categories of hex files.

This allows to sort the files in a clear way e.g. if several programmable extension modules are mounted on the robot or different program versions are used. The list is automatically saved at the end of the program. Of course only the paths to the hex files are saved, not the hex files themselves. If you work on a program, you just need to add and select the hex file once. Then you can load the new program into the microcontroller after every re-compiling of the program. (you can also use the key combination [CTRL+D] or [CTRL+Y], to start the program directly after the transfer). The path names are of course totally different under the various operating systems. Nevertheless the RobotLoader suits both, Windows and Linux, without any changes, as there is a separate list for Windows and Linux.

Either you continue now with the other example programs

# 6. Programming the CATERPILLAR

Now we are gradually coming to the programming of the robot.

## Setting up the source text editor

First of all, we need to set up a little development environment. The so-called “source text” (also called “sourcecode”) for our C program must be fed into our computer one way or the other.

To this end, we will definitely not use programs like OpenOffice or Word. As this might not be obvious for everybody, we stress it here explicitly. They are ideally suited to write manuals like this one, but they are totally inappropriate for programming purposes. Source text is pure text without any formatting. The compiler is not interested in font size and color...

For a human being, it is of course much clearer if some keywords or kinds of text are automatically highlighted by colors. These functions and some more are contained in Programmers Notepad 2 (abbreviated hereafter by “PN2”) that is the source text editor that we will use (ATTENTION: Under Linux you need to use another editor that offers about the same functions as PN2. Usually, several editors are pre-installed. (e.g. kate, gedit, exmacs or similar)). In addition to the highlighting of keywords and others (called “syntax highlighting”) it offers also a rudimentary project management. This allows to organise several source text files in projects and to display in a list all files related to a project. Moreover you can easily retrieve programs like the AVR-GCC in PN2 and get the programs conveniently compiled via a menu item. Normally the AVR-GCC is a pure command line program without graphic interface...

You will find more recent versions of Programmers Notepad on the project homepage:  
<http://www.pnotepad.org/>

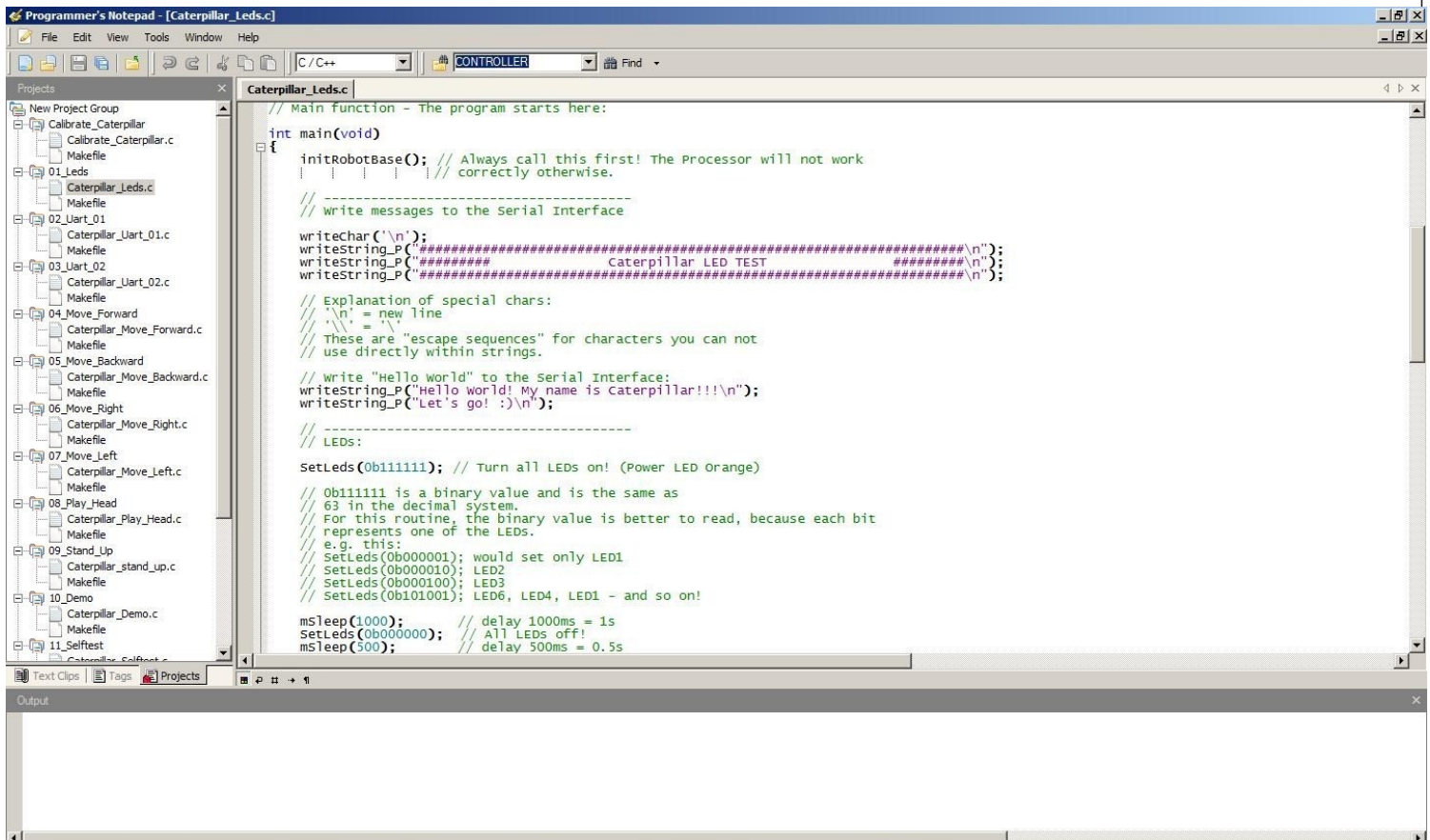
The newest versions of WINAVR don't require the setting up of menu items anymore.

### PLEASE NOTE:

In this section we don't describe anymore how you have to set up menu items in PN2 as the newest WINAVR versions have done this already for you.

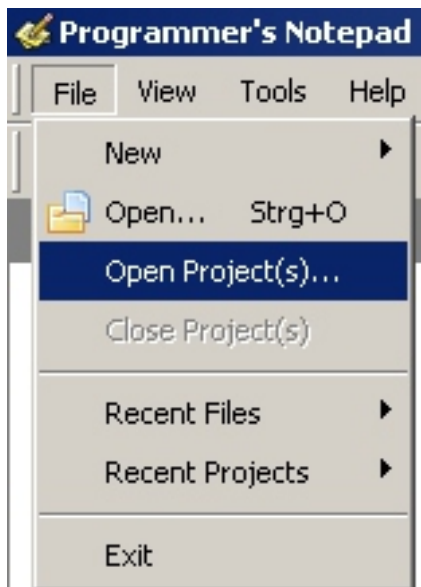
See on page 35 “Open and compile an example project” how you can open an example project.

If you have opened an example project, it should look a bit like this on the Pn2 screen;



On the left hand side are shown all example projects, on the right hand side the source text editor (with the mentioned syntax highlighting) and at the bottom the tools output (in this case the output of the compiler). You can convert many other things in PN2 and it offers many useful features.

## 6.1. Open and compile an example project



Let's test now if everything runs properly and open the example projects: Select in the "File" menu the item "Open Project(s)".

A normal file section dialogue appears. Search the folder "CaterpillarExamples" in the folder into which you have saved the example programs.

Open the "CaterpillarExamples.ppg" file. This is a project group for PN2 that uploads all example programs as well as the Caterpillar Library into the project list ("Projects").

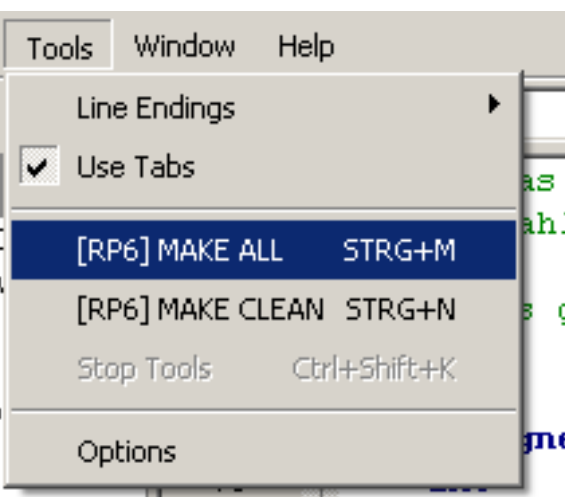
Now all example projects are conveniently at hand if you want to refer to them at the beginning or look for functions in the Caterpillar Library etc..

Open the first example program on top of the list ("01\_Leds" and select file "01\_Leds") that appears on the left edge of the program window. Just doubleclick on "01\_Leds.c". A source text editor is displayed in a window inside the program.

An output area should appear on the bottom of the program window of PN2. If not, you have to enable this area via the "View" menu --> "Enable output" OR if the area is too small, increase the size by pulling the edges with the mouse (the mouse cursor changes into a double arrow at the upper edge of the grey area marked "output" at the bottom of the program window...).

You can take a quick look at the program that you just opened with the source text editor but you don't need to understand right now what is happening exactly. However as a first info: The green text are comments that are not part of the actual program. They are only used for description/documentation purposes.

We will explain this in detail a bit further down (there is also a version of this program WITHOUT comments so that you can see how short the program is in fact. The comments inflate it a lot but are necessary for the understanding. The uncommented version is also useful to copy the code in your own programs.).



First of all we just want to test if the compilation of programs works properly.

In the Tools menu on top both freshly installed menu items (see fig.) should appear (or the [WinAVR] inputs existing as a standard in PN; whatever, it works normally with both).

Please click now on "MAKE ALL".

Pn2 retrieves now the above mentioned "make\_all.bat" batch file. This will on its turn retrieve the program "make". More info about "make" will follow later.

The example program will now be compiled. The generated hex file contains the program in the translated format for the microcontroller and can be uploaded and executed later. The compilation process generates a lot of temporary files (suffixes like ".o", ".lss", ".map", ".sym", ".elf", ".dep"). Just ignore them. The newly set up tool "make clean" will erase them all. Only the hex file is of interest for us. By the way, the function "make clean" will not erase this file.

After the activation of the menu item MAKE ALL, following output should display (below in a considerably shortened version. Some lines may look of course a bit different):

```
> "make" all
----- begin -----
avr-gcc (WinAVR 20100110) 4.3.3
Copyright (C) 2008 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
Size before:
AVR Memory Usage
-----
Device: atmega16
Program: 3074 bytes (4.7% Full) (.text + .data + .bootloader)
Data: 68 bytes (1.7% Full) (.data + .bss + .noinit)
EEPROM: 14 bytes (0.7% Full)
(.eeprom)
Compiling C: Caterpillar_Leds.c
avr-gcc -c -mmcu=atmega16 -I.
-gdwarf-2 -DF_CPU=16000000UL -Os -funsigned-char -funsigned-bitfields -fpackstruct -fshort-enums -Wall
-Wstrict-prototypes -Wa,-adhlns=./Caterpillar_Leds.lst -std=gnu99 -MMD -MP -MF .dep/Caterpillar_Leds.o.d Caterpillar_Leds.c -o
Caterpillar_Leds.o
Linking: Caterpillar_Leds.elf
avr-gcc -mmcu=atmega16 -I. -gdwarf-2 -DF_CPU=16000000UL -Os -funsigned-char -funsignedbitfields
Creating load file for Flash: Caterpillar_Leds.hex
Creating load file for EEPROM: Caterpillar_Leds.eep
avr-objcopy -j .eeprom --set-section-flags=.eeprom="alloc,load" \
--change-section-lma .eeprom=0 --no-change-warnings -O ihex Caterpillar_Leds.elf
Caterpillar_Leds.eep || exit 0
Size after:
AVR Memory Usage
-----
Device: atmega16
Program: 3074 bytes (4.7% Full) (.text + .data + .bootloader)
Data: 68 bytes (1.7% Full) (.data + .bss + .noinit)
EEPROM: 14 bytes (0.7% Full)
(.eeprom)
----- end ----->
Process Exit Code: 0
```

The "Process Exit Code: 0" at the end is most important. It means that no error occurred during compilation. If another code appears there, the sourcecode contains an error that must be corrected before it will work. In this case, the compiler will output various error messages that give some more information.

Please note however that the "Process Exit Code: 0" is not a guarantee of a fully error-free program. The compiler will not find flawed thinking in your program and it can't prevent the robot from running into a wall ;-)

**IMPORTANT:** You might find warnings and other messages further above.

These are often very helpful and always indicate important problems. That's why these always need to be solved. Pn2 highlights warnings and errors by colors to make the identification easier. Even the line number is indicated that the compiler is criticizing. If you click on the colored error message, Pn2 skips in the relevant editor directly to the faulty line

The indication at the end "AVR Memory Usage" is also very useful.

```
Size after:
AVR Memory Usage
-----
Device: atmega16
Program: 3074 bytes (4.7% Full)
(.text + .data + .bootloader)
Data: 68 bytes (1.7% Full)
(.data + .bss + .noinit)
EEPROM: 14 bytes (0.7% Full)
(.eeprom)
```

This means for the Atmega16 processor that our program has a size of 3074 bytes and that 68 bytes of RAM are reserved for static variables (you have to add to this the dynamic ranges for heap and stack but this would go too far... just keep always at least a few hundred bytes of memory free). We dispose in total of 64kb (65536 bytes) of Flash ROM and 2kb (2028 bytes) of RAM. On the 64kb, 2k are occupied by the bootloader - so we can only use 62kb. Make always sure that the program fits into the available memory space. (The RobotLoader doesn't transfer the program if it is too big.)

The just compiled program can now be uploaded via the RobotLoader into the robot. To do that, you have to add the newly generated hex file into the list in the RobotLoader via the button "Add", select it and click on the "Upload" button exactly as you did for the selftest program. After that you can switch back to the terminal and look at the output of the program. Of course you need to launch the execution of the program. The easiest way to do it in the terminal is to press the key combination [CTRL]+[S] on the keyboard or to use the menu (or just to send an "s" - after a reset you have to wait a little bit though until the message "[READY]" is displayed in the terminal.). The key combination [CTRL]+ [Y] is also very convenient as the currently selected program is uploaded into the Robot Arm and immediately started. This avoids to click on the "Flash Loader" tab in the terminal or to use the menu.

## 6.2. 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 compilers are available for nearly every microcontroller currently on the market and for this reason, all recent robots by Global Specialties (ASURO, ROBOT ARMS, and RP6v2) 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 Caterpillar 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. Linux users will usually have to compile a version by themselves and we expect that you have completed this already.

## 7. C - Crash Course for beginners

*This chapter only provides a **very short introduction to C-programming**, discussing only the absolutely required minimum amount of things used for caterpillar. 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<sup>1</sup> may be viewed online free of charge.

### 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 Caterpillar and RP6v2 CD (see [www.globalspecialties.com](http://www.globalspecialties.com)) 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 mentions some tools (programming equipment, etc.) and other things, which are not required for the Caterpillar. Nevertheless, it's worth to have a look at it.

### IMPORTANT NOTICE



\* *This crash course is made for the RP6v2 robot and described in the RP6v2 manual. This means all explanations are specific for the RP6v2 robot and not for the Caterpillar. We still think the explanations are also meaningful for the Caterpillar users so we decided to put the RP6v2 manual chapter also in this Caterpillar manual.*

<sup>1</sup> *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...*

**Source: From the RP6v2 manual "C Crash course for beginners"**



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](http://winavr.sourceforge.net/install_config_WinAVR.pdf)  
and the AVR-LibC Documentation:  
<http://www.nongnu.org/avr-libc/user-manual/index.html>

*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.*

## 7.1. 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:

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

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 `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 CATERPILLAR. */`

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 "CaterpillarRobotBaseLib.h"**

This includes the caterpillar 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 `CaterpillarRobotBaseLib.h` and `CaterpillarRobotBaseLib.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 curly brackets '{' and '}'. A block combines several commands.

### **Line 9: initRobotBase();**

A function from the CaterpillarLibrary 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 Caterpillar Library with the parameter String `"Hello World.\n"`. The function will output the text to the serial interface.

### **Line 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.

## 7.2. 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("Hello World!\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 brackets { } 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 bracket 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).

-

## 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
<b>signed char</b>	<b>int8_t</b>	8 Bit: <b>-128 ... +127</b>	1 Byte
<b>char</b>	<b>uint8_t</b>	8 Bit: <b>0 ... 255</b>	' ' unsigned
<b>int</b>	<b>int16_t</b>	16 Bit: <b>-32768 ... +32767</b>	2 Bytes
<b>unsigned</b>	<b>int uint16_t</b>	16 Bit: <b>0 ... 65535</b>	" unsigned
<b>long</b>	<b>int32_t</b> 32 Bit:	<b>-2147483648 ... +2147483647</b>	4 Bytes
<b>unsigned long</b>	<b>uint32_t</b> 32 Bit:	<b>0 ... 4294967295</b>	" 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 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 Caterpillar Library 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#[ ]<sup>23</sup>|\*+ .,<>%&/(){}\$§ = ' °?.^".

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:

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

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 RP6v2 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 RP6v2 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 a compiler option.

## 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  if(tolleVariable < 16000) // When myFavoriteVariable < 16000
3  { // Then:
4  writeString("myFavoriteVariable is less than 16000.\n");};
5
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 bracket-pairs:

```
1  if( ((x != 0) && !(x > 10)) || (y >= 200) ) {
2  writeString("OK.\n");
3  }
```

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.

## 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("Hallo\n"); // causing the program to proceed
9  case 5: writeString("du\n"); // with the next two lines
10 case 6: writeString("da.\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.

-



## 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="); // "i=" output
5  writeInteger(i, DEC); // output the "DECimal" value
6  // of i and...
7  writeChar('\n'); // ... a line-break.
8  i++; //increment i..
9  }
```

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 brackets 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; // 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.).

## 7.3. 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>)

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

Let's explain the idea in a small example with two simple functions and the already 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     initRobotBase(); // Always start an Caterpillar-program by calling this function..  
21  
22     // A few function calls:  
23     someLittleFunction();  
24     someOtherFunction();  
25     someLittleFunction();  
26     someOtherFunction();  
27     someOtherFunction();  
28     return 0;  
29 }
```

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 initialize 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  uint8_t calculate(uint8_t param1, uint8_t param2)
8  {
9      writeString("[CALC]\n");
10     return (param1 + param2);
11 }
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 Caterpillar 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.

## Arrays, Strings, Pointers...

EA 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.

## 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 usual things, there are so-called "interrupts". They may be generated by several hardware modules (Timer, TWI, UART, external Interrupts etc.) and require the microcontroller'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 appropriate 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<sup>2</sup>C-Bus) and a few other applications.

## 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 "CaterpillarBaseLib.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 `CaterpillarBaseLibrary.h`, providing definitions of all user accessible functions of the `CaterpillarLibrary` 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 `CaterpillarLibrary` 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.
```

like this f.e.:

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

It can be called like a normal function ( `setStopwatch1(100);` ).

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

## 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 Caterpillar 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 Caterpillar-projects, you can concentrate on the four following entries. Other entries are not required for beginners and may be ignored.

**TARGET** = programmName

**CATERPILLAR\_LIB\_PATH**=../CaterpillarLib

**Caterpillar\_LIB\_PATH\_OTHERS**=\$(Caterpillar\_LIB\_PATH)/Caterpillarbase (Caterpillar\_LIB\_PATH)/Caterpillarcommon)

**SRC** += \$(Caterpillar\_LIB\_PATH)/Caterpillarbase/CaterpillarRobotBaseLib.c

**SRC** += \$(Caterpillar\_LIB\_PATH)/Caterpillarcommon/Caterpillaruart.c

**SRC** += \$(Caterpillar\_LIB\_PATH)/Caterpillarcommon/CaterpillarI2CslaveTWI.c

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

Caterpillar'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 "Caterpillar\_LIB\_PATH" allows you to specify the pathname of the CaterpillarLibrary files. Please specify a relative path name, e.g. "../Caterpillarlib" or "../Caterpillarlib" (in which ".." is means "one directory level up").

A third entry Caterpillar\_LIB\_PATH\_OTHERS is used to specify all other directories. We splitted the CaterpillarLibrary 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 CaterpillarLibrary's files you are using.

Now, what does \$(Caterpillar\_LIB\_PATH) mean? Well, that's the way to use variables in makefiles. We already defined a "variable" named Caterpillar\_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 Caterpillar 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/>



## 7.4. CATERPILLAR CaterpillarLibrary

The Caterpillar function library (abbr. CaterpillarLibrary or CaterpillarLib) provides a great number of lowlevel functions to control the Caterpillarhardware. 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 ATMEGA16 Microcontroller in order to be able to write programs for the Caterpillar. However, by reading some important parts of the data sheet you may gain insight of how the CaterpillarLibrary works in detail. In fact, we intentionally avoided perfect tuning for all CaterpillarLibrary functions, in order to leave some work for you.

You are invited to add more functions and to optimize existing ones. Please consider the CaterpillarLibrary 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.

### 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
23
45
67
89
10
11
#include
1  #include „CaterpillarRobotBaseLib.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 }
12
```

**Basically any Caterpillar 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 brackets). 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.

## UART Functions (serial interfaces)

A few of the CaterpillarLibrary'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).

### 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)  
und 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("ABCDEFGH");
```

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

```
writeString_P("ABCDEFGH");
```

will equally output "ABCDEFGH", 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("ABCDEFGH", 3, 1);
```

Output: "BCD"

```
writeStringLength("ABCDEFGH", 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);
```

Ausgabe: "000011111110"

### 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 parameter.

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 Caterpillaruart.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).

## 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 CaterpillarLibrary also provides functions for these purposes. It uses one of the MEGA16'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 100 $\mu$ s (100 $\mu$ s = 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 25500 $\mu$ s = 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); // 100 $\mu$ s delay  
sleep(10); // 1ms 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(65535); // 65.5 Sekunden 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 CAterpillarLibrary 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

CaterpillarLibrary 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 Caterpillar 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 "CaterpillarRobotBaseLib.h"
2
3  int main(void)
4  {
5      initRobotBase(); // Initialize the micro-controller
6      writeString_P("\nRP6 Stopwatches Demo Programm\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, Start again from the beginning LED1.
25             setStopwatch1(0); // Reset Stopwatch1 to zero
26         }
27
28         // Output a counter level in the terminal:
29         if(getStopwatch2() > 1000) // Did 100ms (= 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. Separating 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 läuft nicht, also mache irgendwas...
}
```

`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()`

## Status LEDs and Bumpers (Antenna)

### void setLEDs(uint8\_t leds)

This function allows you to control the 4 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); // Switches all LEDs off.
setLEDs(0b000001); // switches StatusLED1 on all other off
setLEDs(0b000010); // StatusLED2
setLEDs(0b000100); // StatusLED3
setLEDs(0b001010); // StatusLED4 and StatusLED2
setLEDs(0b010111); // StatusLED5, StatusLED3, StatusLED2 and StatusLED1
```

### An alternative possibility the following:

```
statusLEDs.LED5 = true; // LED1 im LED Register activation
statusLEDs.LED2 = false; // LED2 im LED Register deactivation
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.LED1 = true; will NOT directly activate LED1. 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 antenna status. In order to read the antenna, the controller will quickly switch the pin direction to input mode and check if the connected springswitch is closed. We provide two functions for checking the antenna.

The first function:

```
uint8_t getHeadLeft(void)
```

will read the left antenna status, whereas:

```
uint8_t getHeadRight(void)
```

will read the right antenna switch.

```
uint8_t getTail(void)
```

will read the Tail antenna 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 antenna ports, but if the pins are set to low level output AND a antenna switch is closed at the same time, the port terminal will conduct a bit more current. Such unnecessary currents should be avoided of course (the AVR's have Tristate outputs – to turn the LED off, they are set to floating)

### Example:

```
if(getHeadLeft() && getHeadRight()) // Both antenna...
    escape(); // define your own function f.e. turn left or reset
else if(getHeadLeft()) // Left...
    escapeLeft(); // Hier also turn or reset.
else if(getHeadRight()) // Right...
    escapeRight(); // turn left or reset
mSleep(50); //check the antenne 20 times per second (20Hz) ...
```

The antenne LED will always change color when activated this is because the mechanical construction.



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 Rp6 robot example:

```
1  #include "RP6RobotBaseLib.h"
2
3  // Unsere „Event Handler“ Funktion für die Bumper.
4  // Die Funktion wird automatisch aus der RP6Library aufgerufen:
5  void bumpersStateChanged(void)
6  {
7      writeString_P("\nBumper Status hat sich geaendert:\n");
8
9      if(bumper_left)
10         writeString_P(" - Linker Bumper gedrueckt.\n");
11     else
12         writeString_P(" - Linker Bumper nicht gedrueckt.\n");
13     if(bumper_right)
14         writeString_P(" - Rechter Bumper gedrueckt.\n");
15     else
16         writeString_P(" - Rechter Bumper nicht gedrueckt.\n");
17 }
18
19 int main(void)
20 {
21     initRobotBase();
22
23     // Event Handler registrieren:
24     BUMPERS_setStateChangedHandler(bumpersStateChanged);
25
26     while(true)
27     {
28         task_Bumpers(); // Bumper automatisch alle 50ms auswerten
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 Caterpillar or 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 performed 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 Caterpillar- / 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 or antenna 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.

With the RP6 and Caterpillar robot 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 taste.

## Read ADC values (Battery, Motor current and Light sensors)



With the Caterpillar we read the ADC values in a different way.

Below is a sample of how we read the ADC value from the RP6v2 robot.

There are a lot of sensors connected to the ADC (Analog to Digital Converter), as described in chapter 2. Of course, the Rp6v2 and Caterpillar Library 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\_ADC0 --> Free ADC channel for your own sensor devices

ADC\_ADC1 --> Free ADC channel for your own sensor devices

Example:

```
uint16_t ubat = readADC(ADC_BAT);  
uint16_t adc0 = readADC(ADC_ADC0);  
uint16_t adc1 = readADC(ADC_ADC1);  
if(ubat < 580) writeString_P("Warnung. battery fast leer.");
```

Basically the 5V supply is used as reference voltage, but the function could be modified such that the internal ATMEGA16's 2.56V reference voltage is used instead (see the MEGA16 data sheet). The standard Caterpillar 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 servos.

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 subsequently 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\_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 from the RP6 robot:

```
1  #include "RP6RobotBaseLib.h"
2
3  int main(void)
4  {
5      initRobotBase();
6      startStopwatch1();
7      writeString_P("\n\Small ADC Measurement program...\n\n");
8      while(true)
9      {
10         if(getStopwatch1() > 300) // Alle 300ms...
11         {
12             writeString_P("\nADC Lichtsensor Links: ");
13             writeInteger(adcLSL, DEC);
14             writeString_P("\nADC Lichtsensor Rechts: ");
15             writeInteger(adcLSL, DEC);
16             writeString_P("\nADC Akku: ");
17             writeInteger(adcBat, DEC);
18             writeChar('\n');
19             if(adcBat < 600)
20                 writeString_P("Warning. Accu will be empty soon.\n");
21             setStopwatch1(0); // set Stopwatch1 back to 0
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

## ACS – Anti Collision System

There is NO standard anti collision system on the Caterpillar only the antenna's.  
There is a good option for an anti collision system f.e. Sharp GP2D12 .

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 antenna's.

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 Rp6 robot 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.

```

1  #include "RP6RobotBaseLib.h"
2
3  void acsStateChanged(void)
4  {
5      writeString_P("The ACS-status has changed. L: ");
6
7      if(obstacle_left) // Obstacle links
8          writeChar('o');
9      else
10         writeChar(' ');
11
12     writeString_P(" | R: ");
13
14     if(obstacle_right) // Obstacle right
15         writeChar('o');
16     else
17         writeChar(' ');
18
19     if(obstacle_left && obstacle_right) // obstacle in the middle?
20         writeString_P(" Middle.");
21     writeChar('\n');
22
23     statusLEDs.LED6 = obstacle_left && obstacle_right; // Obstacle in the Middle?
24     statusLEDs.LED3 = statusLEDs.LED6;
25     statusLEDs.LED5 = obstacle_left; // obstacle links
26     statusLEDs.LED4 = (.obstacle_left); // LED5 inverted.
27     statusLEDs.LED2 = obstacle_right; // obstacle right
28     statusLEDs.LED1 = (.obstacle_right); // LED2 inverted.
29     updateStatusLEDs();
30 }
31
32 int main(void)
33 {
34     initRobotBase();
35
36     writeString_P("\nRP6 ACS - Test program\n");
37     writeString_P("_____ \n\n");
38
39     setLEDs(0b111111);
40     mSleep(1000);
41     setLEDs(0b001001);
42
43     // Register the ACS Event Handler:
44     ACS_setStateChangedHandler(acsStateChanged);
45
46     powerON(); // Activate the IR receiver (incl. encoders etc.)
47     setACSPwrMed(); //set the ACS medium transmit power.
48
49     while(true)
50     {
51         task_ACS(); // Frequently call the task_ACS function.
52     }
53     return 0;
54 }

```

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... ).*

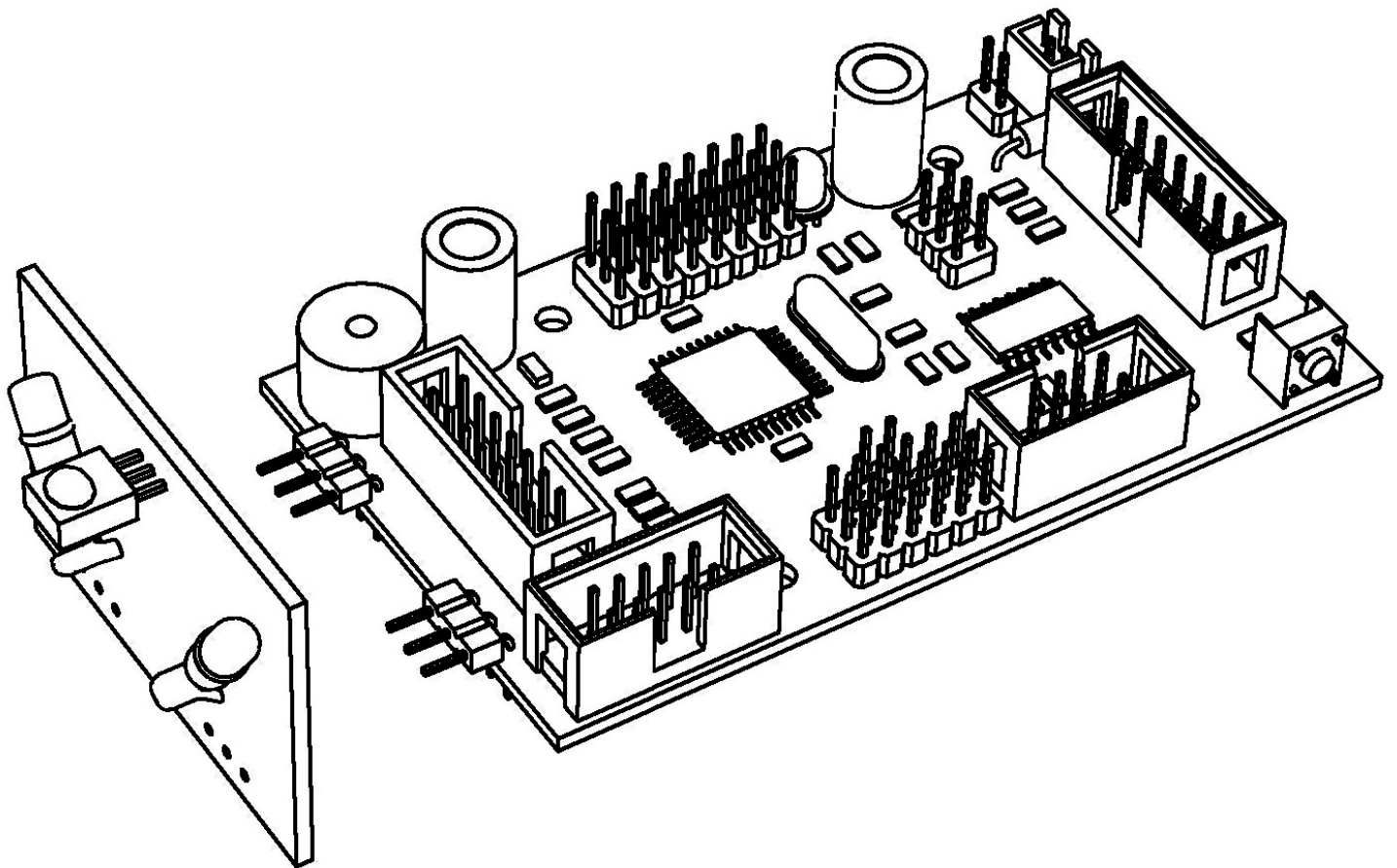
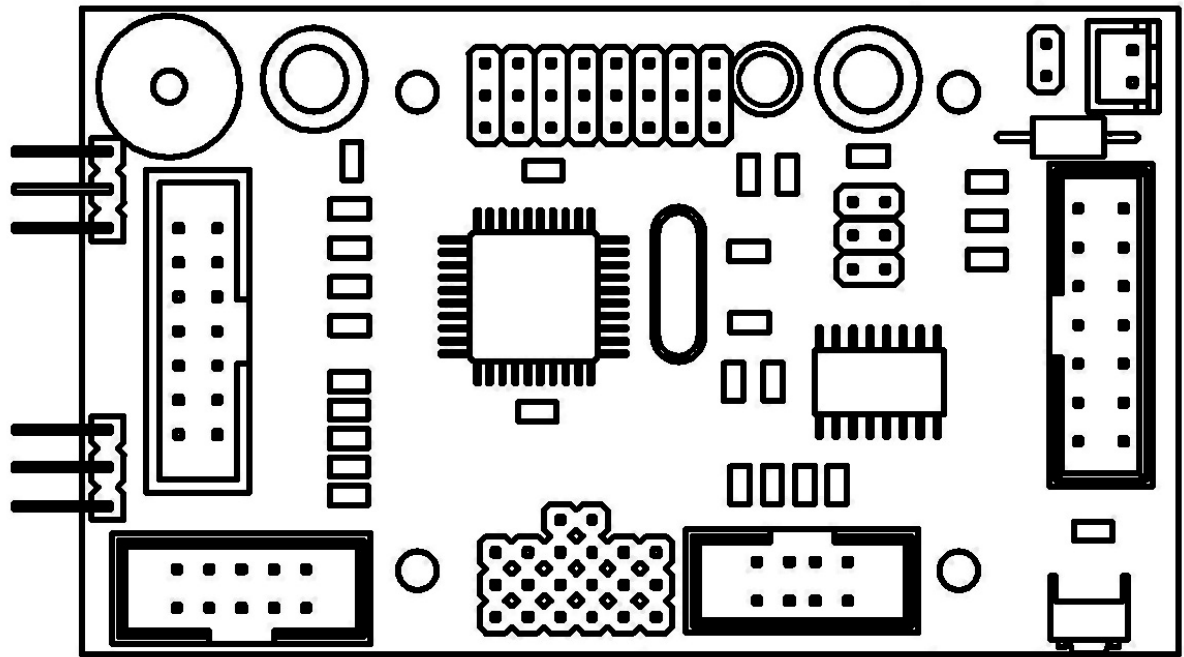
*Of course the detection range heavily depends on the obstacle's surface. Obviously, black surfaces will not reflect the same amount of light as bright white surfaces. The ACS may even ignore some of the dark colored objects.*

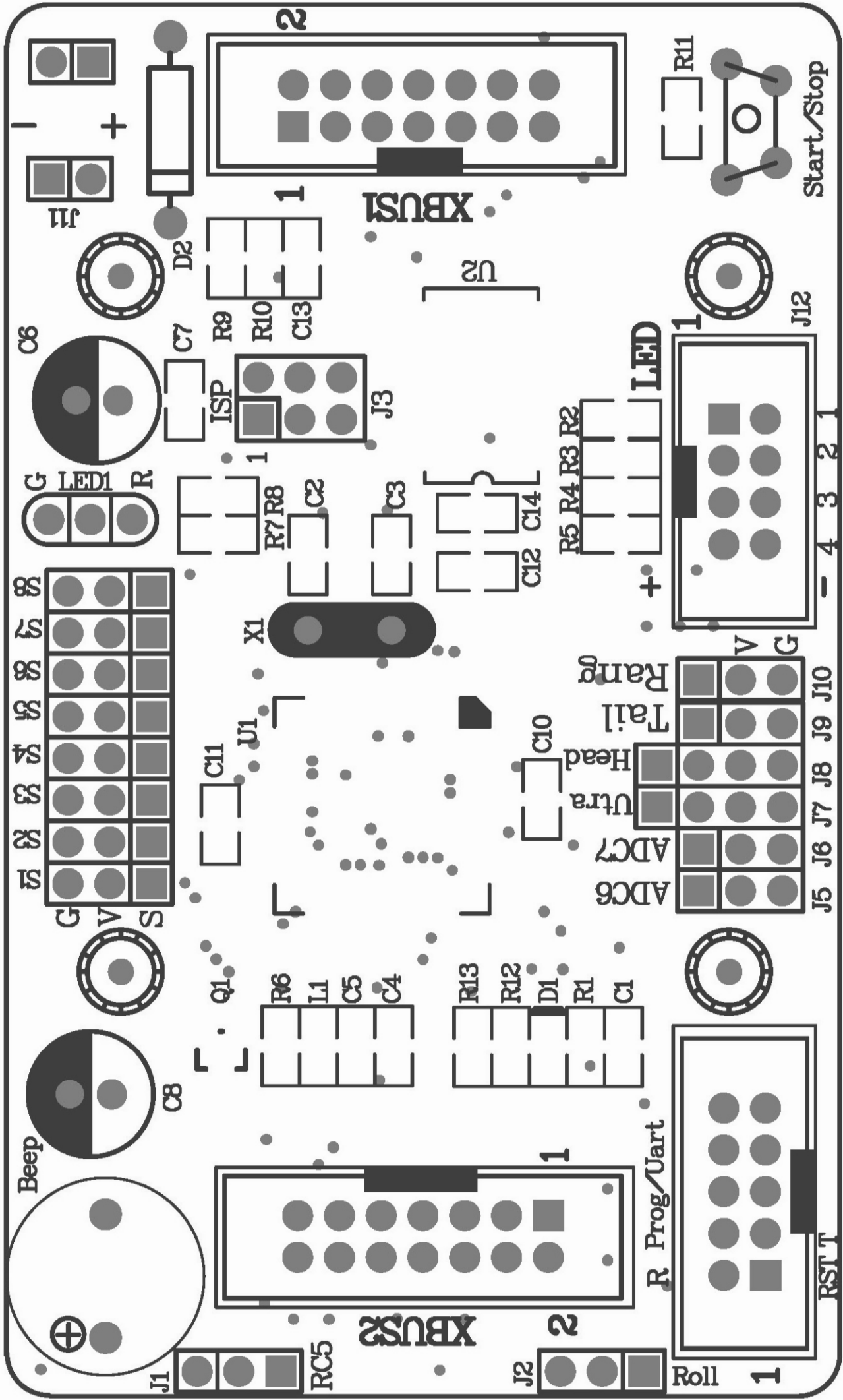
*In critical situations we might prefer to support the ACS by ultrasonic sensors or by improved IR sensors.*

# APPENDIX



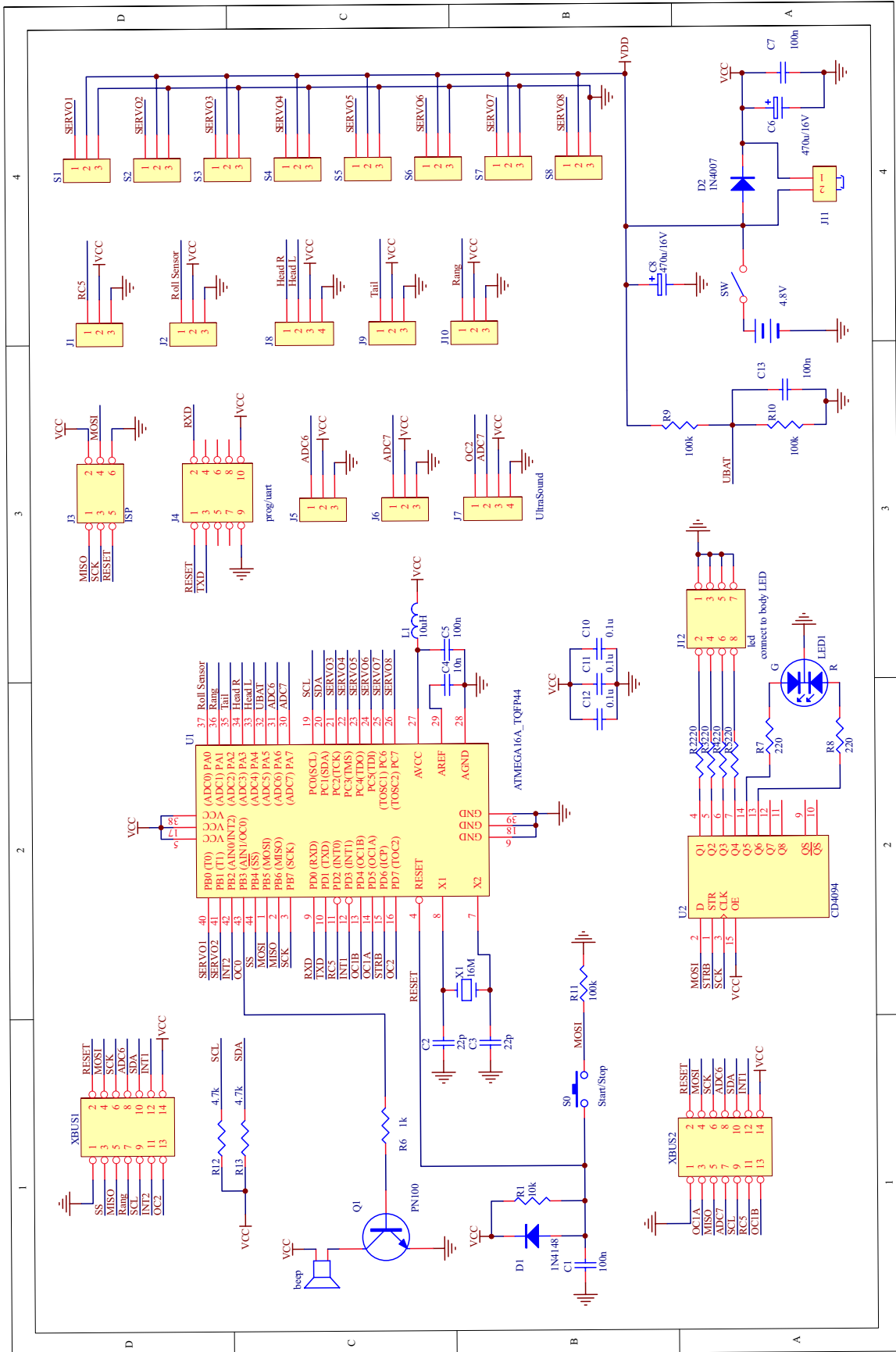
# A. PCB'S



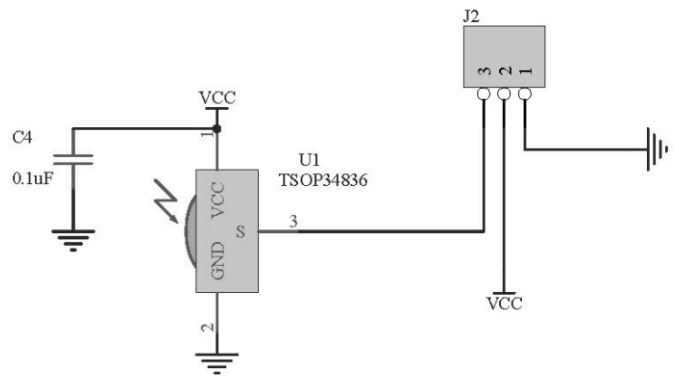
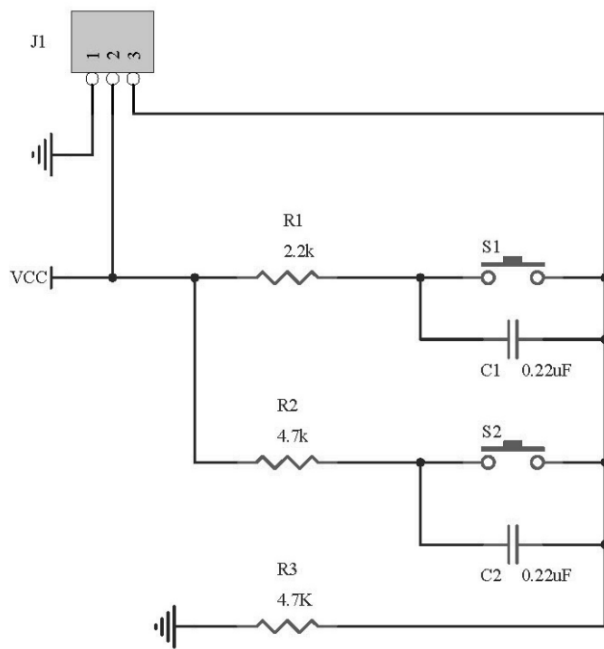


PCB layout diagram showing various components and their connections. Components include resistors (R1-R14), capacitors (C1-C14), LEDs (LED1, LED2), a buzzer (Beep), and a microcontroller (U1). Connectors J1 through J12 are shown, along with XBUS1, XBUS2, R Prog/Uart, Start/Stop, and Roll buttons.

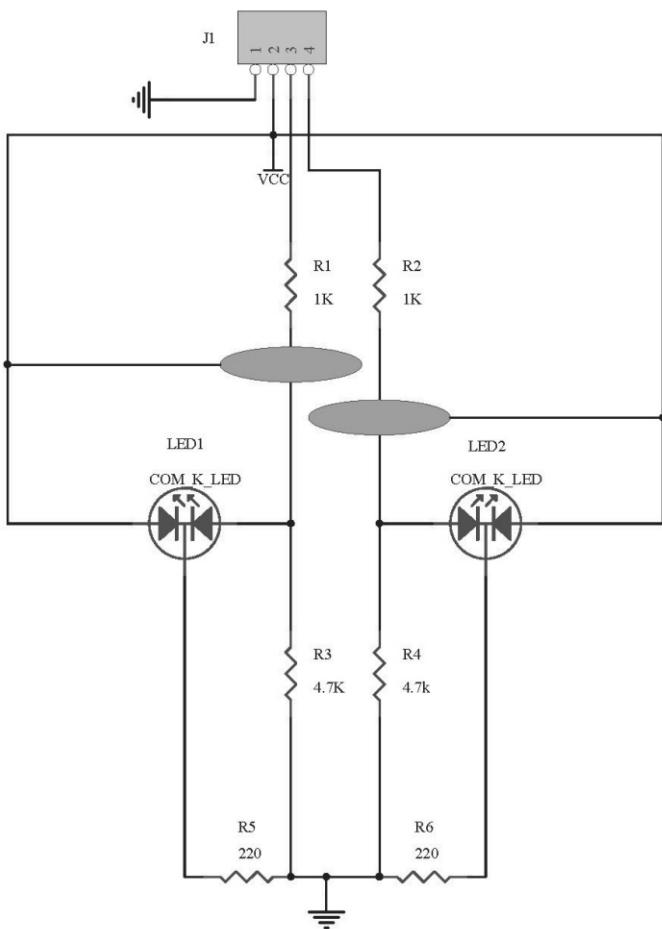
# B. Circuits



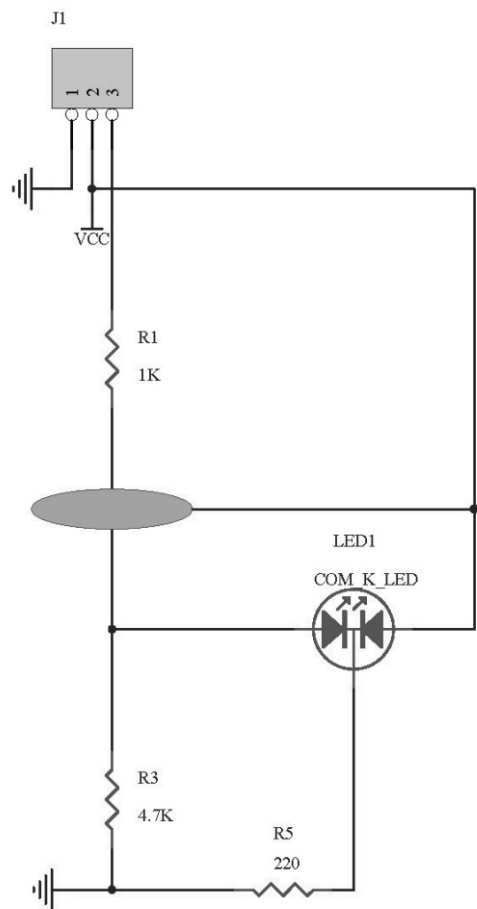
# Roll sensor

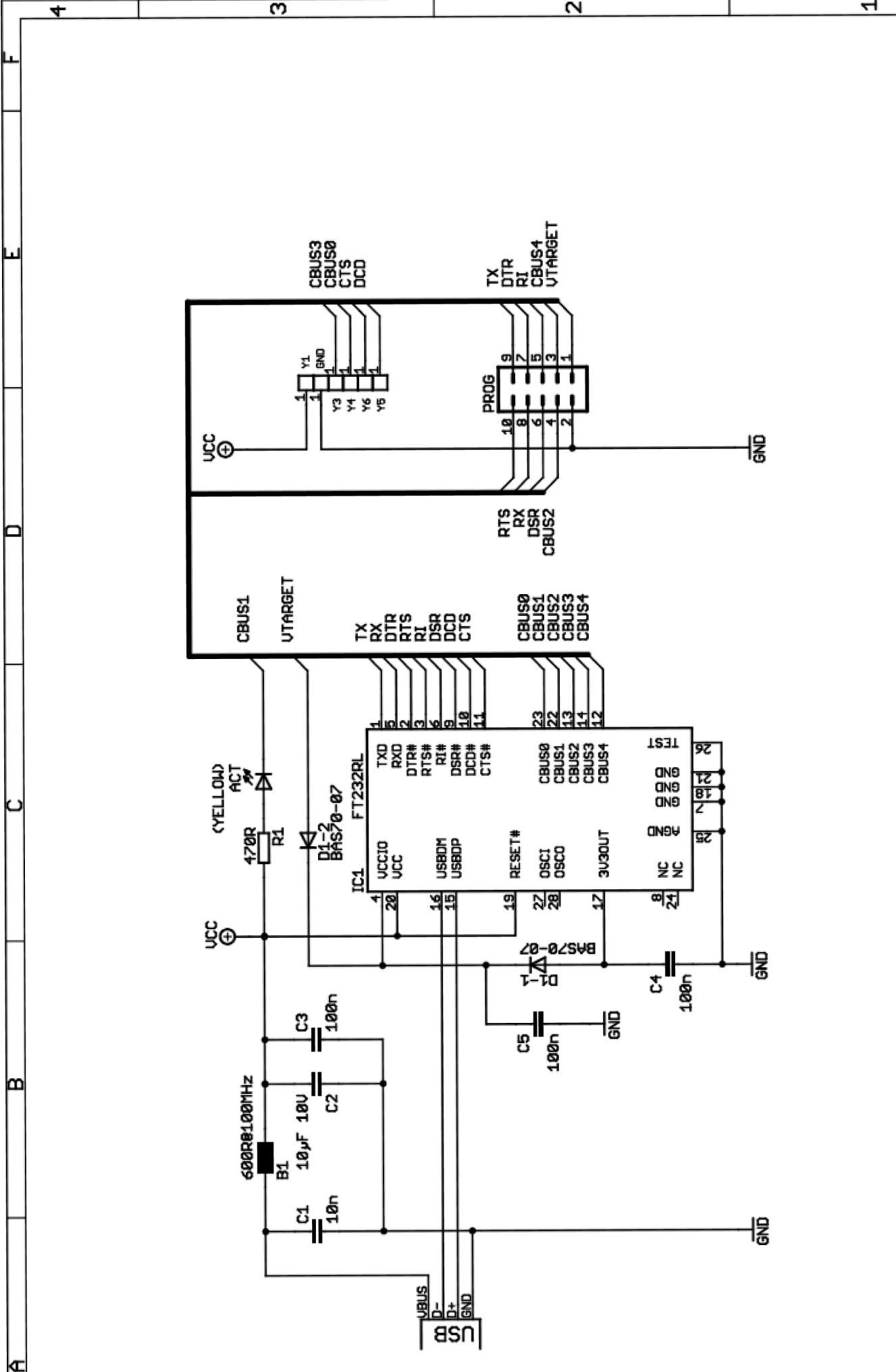


# Head Antenna



# Tail Antenna





RP6 ROBOT SYSTEM

USB INTERFACE

RP6v2\_USB\_REVG

01.09.2011 17:00:00