Halobject manual

Solberg, R.H.M.; de Jong, R.J.M.

Published: 01/01/2005

**Document Version**

Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the author’s version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

**Citation for published version (APA):**


**General rights**

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

**Take down policy**

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 15. Dec. 2018
Halobject manual
Part 1

DCT 2005 – 30

R.H.M. Solberg, R.J.M. de Jong

Tutors:

Dr. Ir. M.J.G. van de Molengraft
Dr. Ir. G.W.M. Peters

Eindhoven, March 2005

Eindhoven University of Technology
Department of Mechanical Engineering
Division Dynamical Systems Design
Control Systems Technology Group
# Table of contents

**INTRODUCTION** 4

**CHAPTER 1: HARDWARE** 5

- §1.1 FRAME 5
- §1.2 CONSTRUCTION 5
- §1.3 WALKING 9
- §1.4 ELECTRICAL SWITCH BOX AND OPERATOR PANEL 11
  - §1.4.1 INSTALLATION AND CONNECTIONS 11
  - §1.4.2 OPERATOR PANEL 12
  - §1.4.3 OPERATION OF THE HALOBJECT 13
- §1.5 PROTECTION 13
  - §1.5.1 FAILURE RECOVERY 15
- §1.6 FEELER MECHANISMS 16
  - §1.6.1 CONSTRUCTION 16

**CHAPTER 2: HARDWARE CONFIGURATION** 18

- §2.1 DIAMOND-MM-16-AT 18
- §2.2 RUBY-MM 18
- §2.3 MESA-4I30 18

**CHAPTER 3: INSTALLATION DEBIAN 3.0 ON COMPACT FLASH-MEMORY CARD** 20

- §3.1 STEP-BY-STEP INSTALLATION 20
- §3.2 INSTALLATION OF 2.6.7-ADEOS KERNEL 22
- §3.3 RTAI INSTALLATION 23

**CHAPTER 4: DRIVER SOFTWARE** 24

- §4.1 IO.c FOR THE RUBY-MM 24
- §4.2 ANALOGIN.c FOR THE DIAMOND-MM-16-AT 25
- §4.3 ENCODER.c FOR THE MESA-4I30 27
- §4.4 USE OF THE UNIVERSAL DRIVER SOFTWARE 28
- §4.5 WINTARGET REAL-TIME TARGET FOR MATLAB/SIMULINK/RTW 28
  - §4.5.1 SOFT REAL-TIME BASED ON RTC 28
  - §4.5.2 HARD REAL-TIME BASED ON RTAI-LXRT 29
Introduction

This report describes the development of a moving artwork for the central hall of the NET3-building at the media park in Hilversum. Many people pass this hall and find it a dull place to be in. In 2002 an artist wanted to do something about this by creating a moving sculpture to be placed in this hall. A full year of study followed about walking, sensors, feelers, skin and character to be used for this object. In September 2003 students began with the realization of the eventual design.

This is a survey of several subjects that have passed in review during the last year as well as the main subject of the traineeship, namely to build the heart of the so-called ‘halobject’. This heart consists of a real-time Linux-based PC/104 system, in which boards can be stacked to handle the digital and analog in- and outputs. The drivers for these boards had to be written in order to control the different motions of the walking and feeler mechanisms as well as the sensors that are attached to the object.

The safety aspect of the halobject is also very important, so the electrical design will also be pointed out. The basic configuration (kernel, RTAI and driver software) will also be described together with the steps that have to be taken for a safe operation of the halobject to make sure that our results until now can be re-produced. This report will serve as a manual for future students that will work with the halobject.
Chapter 1: Hardware

This chapter describes the development of the framework, including the total electrical system and operator panel.

§1.1 Frame

The halobject needs a strong skeleton to carry all its equipment and to execute the imitated movement of a large insect. Because of this, a frame with driving wheels wouldn’t suffice, so a real walking mechanism had to be created. Also, the demanded dimensions of the artist had to be achieved. An important starting constraint in the design was the final weight of the object. The frame had to be strong enough to withstand the forces both statically and dynamically. For all the expected parts, an estimate for their weight was made. The only uncertainty was the weight that the designed frame itself would finally have. Therefore an estimate for the frame was made as well. The total weight of the object finally was assumed to be 175 kg at most. Within this estimation, a safety margin of 50 kg was taken. Out of the demands of the artist and the examples of former OGO-projects, a walking mechanism with 6 legs (hexapod) has been developed.

§1.2 Construction

The most commonly used materials that fulfill the demands are steel tube profiles and aluminium tube profiles. The whole frame construction is therefore made of these materials. For the construction, a square base housing was made (see figure 1). On this housing, the rest of the frame is mounted. By using truss-like constructions, the frame will be relatively light and very strong (more than strong enough for its purpose).

Figure 1: square base housing
First of all the walking mechanism had to be fixed to the framework. The legs on the front and backside are connected to the housing by large aluminium beams that are mounted on a turning axis, which is fixed to the square housing. For the suspension of the beams to the axis one angular contact ball bearing and one radial ball bearing is used to support the weight parallel to the axis and perpendicular to the axis, respectively. For the suspension of the legs to the beams the same type of bearings are used (see figure 2).

*Figure 2: the ‘legs’ of the halobject*

Until now only four legs have been discussed. These legs are mostly used for translating and turning the frame by statically determining the object and pushing it forward. The weight of the object has to be displaced by two other legs. Therefore a cantilever leg has been made which is fixed below the square housing (see figure 3).
Figure 3: cantilever leg attached to the frame

To drive this leg a large electric motor and a large driving belt is used. The belt is mounted between the horizontal stand-beams of the leg and guided over the housing of the motor. Turning this motor will result in lifting one side of the leg and this will tilt the whole frame to one side.

One of the most important parts of the object are the batteries. The batteries will approximately weigh 40 kg each, so their place on the object is very important. They are placed low on each side as possible to balance the object and to keep its centre of mass as low as possible (see figure 4).

Finally a kind of roof is created on the housing. All other components of the object are mounted on top of this roof. Rubber dampers connect the roof and housing to each other. This is done to protect the roof from vibrations caused by the walking mechanism and vice versa. The feeler mechanisms, the switch box, the distributor case, the mantle and the PC/104 (see figure 5) are attached to this roof.
Figure 4: space for batteries

Figure 5: ‘roof’ of the frame
§1.3 walking

The movement of the object by this frame is based on the spinewalker mechanism which originally was created during the OGO-projects (student projects).

The spinewalker also was equipped with 6 legs. The two legs in the middle function as the cantilever leg which tilts the object on its side. The object then stands only on three legs, on one side only in the middle and the other on the front and back leg. As mentioned before the frame is also equipped with the cantilever leg, so the whole object is also tilted from one side to the other. The front and back legs of the spinewalker were interconnected with each other on each side, and the legs were connected to a square housing by making use of cross hinges. Now by turning the legs via these hinges a rotation and translation was made. So by “rotating” the right and left legs of the spinewalker around the square housing a translational movement was obtained.

For the final object a frame with the use of cross hinges would have become too heavy because of the large amount of strong connections that were necessary (e.g. 16 hinge points would have been used). Now by using driving belts, which are cross-connected between the middle axis and the axis of the legs, the amount of hinges is reduced to six. On the side of the beams a cogwheel is connected to a driving motor. By turning this motor the whole beam will turn around the middle axis. The belts are led over a fastened cog on this axis, so by turning the beam the belts are shifted. On the axis of the front and back legs cogs are mounted also. The legs can rotate around the beam, so when the belts are shifted (by turning the beam around the middle axis) the front and back legs will rotate around the beam. This happens in exactly the same angle (see figure 6).

![Figure 6: cross-connection of the driving belts](image)
These belts are mounted inside each side of the beams. So by rotating the two beams, the four legs will rotate also. Finally, the translation is created by turning the square housing around the beams and the front and back legs will then turn in the opposite direction. Now by using the cantilever leg also the object walks forwards or backwards (see figure 7).

*Figure 7: walking straight forward*

Besides walking in a straight line, the object needs the ability to turn. For walking in a straight line the two beams are rotated by exactly the same angle. Now if the beams are not rotated by exactly the same angle, the movement will not be straightforward anymore. By repeatedly turning one leg a smaller angle than the other; the object will turn over one side during its movement.
§1.4 Electrical switch box and operator panel

For the power supply and the driving of the halobject a special switch box has been developed. The goal was to put as much electrical components as possible (which are needed on the object) into this box to save space and reduce the weight (because no long interconnecting cables are needed).

In cooperation with the GTD (Gemeenschappelijke Technische Dienst) an electrical drawing was made for the whole object. These drawings contain all electrical components (fuseblocks, amplifiers, PLC, etc.) and their connections to each other and to the components outside the box (connectors, motors, sensors, etc.). Each component is coded and all wires have their own specific colour code. All the drawings are collected in a separate book.

To save a lot of I/O work and for safety a second computer, a PLC, was added to the switch box. The PLC drives all the switching sensors that are built in for safety, guards the battery condition and enables the connection between the PC/104 computer and the operating panel. So in case of any failure in the system the PLC takes over and stops the object and also notifies the operator.

§1.4.1 Installation and connections

Figure 8: electrical switch box at the back of the frame

According to the drawings made with the GTD the box was manually built in the laboratory of the TU/e. The box is installed at the back of the frame, as shown in figure 8. Two cables connect the box to the batteries and the coded connectors at the back of the
box are connected to the distributor case. This case is also connected to the PC/104 computer and connects the motor encoders, the mantle with its sensors and the external systems. The distributor case acts as the spine of the object so to speak.

§1.4.2 Operator panel

To operate the halobject a separate operating console has been made in the laboratory. This console consists of a touch panel and a wireless transmitting system (figure 9). This system is also built on the front of the switch box. So the connection between the touch panel and the PLC is wireless. The operating console will finally be placed on the wall in the hall of the NPS and the object can be operated from there (for wireless settings see appendix D).

![Figure 9: touch panel with wireless transmitting system](image)

The panel has two main functions. First of all the panel is used for switching the halobject on and off, and to control some sub-functions like resetting and homing. Secondly, the panel acts as an error notification screen. If the system indicates any type of failure the PLC will take over (as discussed before). Simultaneously, the encountered error will be printed on the screen to notify the operator.

The programs for the operator panel and the PLC are still under development; the currently used programs are only test versions. Harrie van de Loo will make the final program (supporting staff member of the Control Systems Technology group at the TU/e).
§1.4.3 Operation of the halobject

Starting to operate the halobject is easy. The only thing to do is to power up the switch box (red switch on the front) and the operator console. The whole system will start-up automatically. During tests a separate computer can be connected to the network port on the object. Via a secure shell protocol new programs can be downloaded and operated by the separate machine and the console.

§1.5 Protection

The halobject contains protection devices to prevent damage to itself and to other objects. During operation several sensors measure conditions, for instance the battery power or the location of the turning legs. If these sensors measure conditions that are out of the safe operating range, the PLC will respond by shutting down or adapting the system.

To protect the frame from damage by overload of the motors the frame is equipped with 14 inductive sensors and 14 emergency stop switches. If one of the movements exceeds its operating area an inductive sensor will notice this at first. This sensor is connected to the PLC that will communicate with the software program (via a digital input). If this is not sufficient (because of the motor speed or a broken sensor) there is an emergency switch next to the inductive sensor (see figure 10) that will disconnect the motor lines. Relays take care of this, so without intervention of the PLC.

Besides the mechanical protections the control-program will also be equipped with a so-called error comparator in the future. The measured error is compared to a maximum value. If this value is exceeded the program will stop itself.

Figure 10: inductive sensor and emergency switch

To prevent damage to the system by low battery power the PLC is equipped with two voltage sensors that measure the condition of the batteries and the sub-voltage source for the PC/104. If the voltage drops below the rated value the PLC stops the program and waits for reloading. If the system is connected to the charging system a sensor will sense the plug-in (message will be displayed on the touch-screen) of the cable and the PLC will
once more prevent the program from starting. So if the system is charging, the halobject will not be able to walk away.

The switch box has some built-in protection as well. All the fuses are guarded by sitops (figure 11), so in case a fuse should break, the sitops will send this to the PLC. The sitops have their own built-in logic that controls the output current of each channel. If the output current should become slightly larger than the tuned value, the sitop will disconnect the line first. This can be undone by resetting the sitop module. If the output current shows a short-circuit then the fuse will break, and the switch box needs to be repaired.

![Figure 11: sitops for built-in protection](image)

The amplifiers for the motors also have their own built-in protection logic. An “open collector” transistor on which a relay is connected monitors the condition of the amplifier. In case of an amplifier-error the transistor is opened and the relay is switched on, which is connected to the PLC. The amplifier can be reset on the front of the switch box. After resetting the relay is switched off and the program can be restarted. If the error is not recovered, the amplifier has to be checked for damage.

Human intervention is also possible by enabling one of the two emergency switches on the console and the object. On the right side of the console a large red button is present (figure 12) and under the mantle a red rope is mounted along the contour. Pulling this rope will set an emergency switch. As soon as one of these switches is pressed, the object will immediately shut down irrespective of what it is doing. This is meant only for emergencies and should not be used for stopping! When the encountered problem is solved the pressed (or pulled) switch first needs to be reset before the program can be restarted. This is done by pulling out the switch on the console or the blue reset button underneath the mantle.
Figure 12: emergency-switch on the console and emergency rope-switch

All of these safety issues are again printed on the console to notify the operator.

§1.5.1 Failure recovery

Most of the encountered errors by the PLC and the PC/104 can be reset on the console. This is not totally functional yet, because the console program is still under construction. The error notification is the only function that is on-line at the moment. In the near future this program will reach its completion and most errors can be reset in the recovery window of the touch panel.

If one of the legs or feelers should exceed its operation range, the program is stopped by the PLC because one of the inductive sensors or switches is set. Before the program can be restarted one of the legs or feelers must be pulled out of the sensors. Once the mantle of the object and feelers are mounted this is not possible to do by hand anymore. Therefore relays in the switch box are present that can be operated via the console. These relays apply a low voltage (only for a few seconds) to the motors in the opposite direction of the sensor in order to drive the leg or feeler out of the sensor-field. After this operation the program can be restarted. Only the positions of the driving motors are now unknown. Therefore the object has to “home” first. By starting the homing program via the console, all the motors are set in their starting positions. This is achieved by driving the motors one at a time into the first inductive sensor. As soon as this happens the position of the motor is stored as an absolute reference position and the motor is directed to its starting position. Now the positions are known and the program can be restarted.
§1.6 Feeler mechanisms

The halobject is equipped with two feelers for which a mechanism was designed in order to move these feelers. One of the demands on this motion is that the stroke of 40 degrees can be made within 2 seconds. The whole mechanism must fit into a space of 25 cm in diameter located at the bottom of the feeler-mantle that will be fixed to the mechanism. All the steps that have been taken to realize the motion of the feelers will be discussed here.

§1.6.1 Construction

The basis of the mechanism consists of tube profiles made of aluminium (10 x 10 cm). To make sure the feeler can perform a scan mode, which is a rotation to ‘look’ at the surroundings, the decision is made to use two motors of which the directions of motion are at right angles to each other. To save space the drive of one rotation is constructed within the tube profile. The drive of the other motion has been fixed on a U-shaped profile in which the aluminum tube can rotate (see figure 13 for a drawing made in Inventor). The drive consists of an electric motor that is connected to a worm via a cross-joint (takes care of the misalignment). This worm moves a worm wheel (square transmission). The decision to use a worm-worm wheel transmission is based on the self-braking property of this mechanism and because it can handle large forces. Of course, bearings are used to suspend the shafts in order to reduce friction and to be able to use a small electric motor. The motor calculation can be found in appendix A and figure 14 shows the mechanism that has been realized.

Figure 13: feeler mechanism
Figure 14: eventual realization of the feeler mechanism
Chapter 2: Hardware configuration

This section describes the configuration of the stackable PC/104 cards, namely the DIAMOND-MM-16-AT [7] (for measuring the analog inputs), the RUBY-MM [6] (for analog outputs and digital I/O) and two MESA 4I30 Quadrature counter cards [9] (for reading out the encoders on the different motors). The configuration consists of setting the base-address, IRQ-level and several jumpers to set the card-properties. All the options can be found in the manuals.

§2.1 DIAMOND-MM-16-AT

This PC/104 module is used to measure the analog inputs that come from the sensors that are applied to the halobject: 3 microphones, 4 ultrasonic sensors, pet-safe system and 1 Light Dependent Resistor. The input-voltages that will be measured lie in the +/- 10V range; writing to a hardware-address in the software takes care of this configuration. This card can be used with either 16 single-ended or 8 differential inputs. Because at least 10 inputs are used, a jumper in the S location on header J4 is installed to configure the inputs as single-ended. The base-address for this card will be 300 Hex (0x300, this is the default) and can be set by installing jumpers on location 4,5,6 and 7 on header J6. The IRQ-level is set to 9 by installing a jumper on location 9 in the interrupt area of jumper block J6 (although this IRQ level will not be used in the software).

§2.2 RUBY-MM

The Ruby-MM is an analog output-module also made by Diamond Systems that will be used to drive the motors on the halobject and to control the digital in- and outputs. The full-scale and bipolar mode of this card is configured by removing the jumper from location 5 on header J4 (gain 2 means 10V instead of 5 V) and by installing a jumper in position B on J4. The base-address must be set different from the other PC/104 cards and with enough space between them to be able to use all registers of each board. The base-address is set to 380 Hex (0x380) and this can be done by installing jumpers at location A on header J5. The Interrupt-level for this card is set to 5 by installing a jumper at location 5 on header J5 (although IRQ is not used in present software).

§2.3 MESA-4I30

To read out the encoders we make use of the MESA-4I30 quadrature counter card. This module comes from a different supplier, namely Mesa Electronics. Two of these cards are used, because each card can handle four different encoder signals and we use seven encoders. The first counter card will be used to read out the encoders for the two turning motors attached to the frame and the encoder for the swing motor (RS-422 operation). Problems occurred with TTL-based encoders (turning legs), so the decision has been made to use RS-422 level encoders.
TTL or RS-422 operation is jumper selectable in groups of two channels. Setting jumpers W1 and W3 in the left hand position sets channels 0-3 to RS-422. The base-address of this first counter card is 200 Hex (0x200), which can be configured by setting the jumpers W5 and W6 in the down position.

The second counter card is used to read the encoders of the motors that are used to control the feelers. All four channels on this card are set to RS-422 (so W1 and W3 both in left position). The base-address for this card is 210 Hex (0x210, which is the default), this can be configured by setting jumper W5 in the up position and jumper W6 in the down position. No IRQ-level can be selected for this card.

In table 1 a summary of the configuration is displayed.

### Table 1: Settings for stackable PC/104 cards

<table>
<thead>
<tr>
<th></th>
<th>Base-address</th>
<th>IRQ</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counter card 1</td>
<td>0x200</td>
<td>-</td>
<td>Channels 0-3: RS-422</td>
</tr>
<tr>
<td>Counter card 2</td>
<td>0x210</td>
<td>-</td>
<td>Channels 0-3: RS-422</td>
</tr>
<tr>
<td>DIAMOND-MM-16-AT</td>
<td>0x300</td>
<td>9</td>
<td>Programmable range (in software)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bipolar, full-scale (+/- 10V)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Single ended: 16 channels available</td>
</tr>
<tr>
<td>RUBY-MM</td>
<td>0x380</td>
<td>5</td>
<td>Bipolar, full-scale (+/- 10V)</td>
</tr>
</tbody>
</table>
Chapter 3: Installation Debian 3.0 on Compact flash-memory card

This section describes the installation of Debian (3.0-r1 i386) on a Compact flash card that is connected to the PC/104 system. The original DiskOnChip (48 MB) has been removed from the system and instead of this a Compact flash card (512 MB) was purchased that can be plugged into a Compact flash-to-IDE adapter. The advantage of this is that the memory card will be recognized as an IDE-drive. This (apart from the amount of space) was one of the problems with the DiskOnChip. Next, a summary will be given of the complete installation, pointing out the changes that have been made with respect to the manual of Ward T. Oud [5]. Also, have a look at [4] for the use of some helpful, typical Linux commands.

§3.1 Step-by-step installation

Before starting the installation, make sure a keyboard, monitor and cd-rom drive are connected to the PC/104 system.

? Set up the Compact flash-to-IDE-adapter to being master and the cd-rom-player as the slave (by setting the jumpers on the correct location). Start the system from CD2 of the installation CD-set for Debian 3.0 and check the boot order in the BIOS-settings. The cd-rom comes first and the Compact flash card in second place, for which the settings have to be autoconfig, LBA.

? Press F3 to see the options for booting from this CD that includes the Linux-installation and a rescue option that might come in handy.

? Press enter to start the installation (default Linux installation).

? Choose English as language.

? Configure the keyboard: choose qwerty/us.

? Now the Compact flash-card has to be partitioned:
  - Delete all partitions on the Compact flash card
  - Choose Partition a hard disk
  - Choose /dev/hda, which corresponds to the master (Compact flash card)
  - Make a new (primary) partition
  - This partition will be the total disk space
  - Make the partition bootable
  - Write the partition table to the disk
  - Close this menu (quit)

? Choose ‘Do without a swap partition’.

? A bad block scan is not necessary.

? Initialize the Linux partition.

? This partition needs to be mounted as root.

? Install the kernel and driver modules.

? A network device has to be added when configuring the device driver modules. Choose eepro100, this is the driver that belongs to the Ethernet card that is present on the system.
The auto-installation will probably fail so this has to be done manually:
- IP 192.168.1.1
- Netmask 255.255.255.0
- Gateway 192.168.1.2 (to another PC)
- Don’t use a domain name

Choose a hostname, for example: debianpc104.

Install the base system; this may take a little while….

After completion of the base-installation we have to make the system bootable.

Choose to install LILO in MBR.

Now the system can be restarted after you’ve unloaded the cd-rom.

After rebooting an installation menu appears to ask which time zone you’re in.

Choose a password for root: rorapi.
Remove PCMCIA packages.
No PPP connection is needed.
Choose CDROM as Debian archive.
The correct path is /dev/cdrom, scan CD1 and CD2.
Don’t add new apt-sources.
No security updates can be downloaded (system is not connected to the internet).
No new tasks have to be selected after running tasksel.
Run dselect? Choose No.
The installation continues, follow the instructions on the screen.
“Do you want man and mandb to be installed setuid man?” Choose No.
A warning appears about linking the kernel that can be neglected.
“Add a mime handler for application?” Choose No.
“Select locales to be generated,” select en_US ISO-8859-1 and en_US.UTF-8 UTF8.
“Which locale should be the default in the system environment?” Select Leave Alone.
“Allow SSH Protocol 2 only?” Choose Yes. This is needed to make the communication possible between both systems (PC with Matlab installed and the PC/104 system) with Secure Shell via the crosslink UTP-cable.
“Do you want /usr.lib/ssh-keysign to be installed SUID root?” Choose Yes.
“Do you want to run the sshd server?” Choose Yes.
A question about the mail-system configuration appears. No mail system will be used, so choose option 5: No configuration.

If all went well the installation is completed and after this the logon-prompt appears and you can log on to the system as root and use the password you chose (rorapi).

During installation the option of installing a graphical interface was ignored, because of the amount of space this takes. Some extra software is needed, for instance to use the ‘make menuconfig’-command to configure the kernel. The CD, labeled with ‘Linux Software’, contains 2 Debian-packages that have to be installed: libncurses5-dev_5.2.20020112a-7_i386.deb and mesag-dev_5.0.0_5.1_i386.deb.

First, mount the cdrom:
Create a mount point, for example: `mkdir /mnt/cd`

Mount the cdrom with: `mount /dev/cdrom /mnt/cd`

Copy both packages to the Compact flash card:
- `cp /mnt/cd/libncurses5-dev_5.2.20020112a-7_i386.deb /usr/local`
- `cp /mnt/cd/ mesag-dev_5.0.0_5.1_i386.deb /usr/local`

Go to the `/usr/local`-directory to install these packages by typing:
- `Apt-get install libncurses5-dev`
- `Apt-get install mesag-dev`

§3.2 Installation of 2.6.7-adeos kernel

The goal is to build real-time applications so a patched kernel (2.6.7-adeos) with RTAI is needed, the steps to follow will be pointed out below.

Copy the file `linux-2.6.7.tar.gz` and `rtai-3.1.tar.gz` from the ‘Linux software’-CD to `/usr/src` after mounting the cdrom:
- `mount /dev/cdrom /mnt/cd`
- `cp /mnt/cd/linux-2.6.7.tar.gz /usr/src`
- `cp /mnt/cd/rtai-3.1.tar.gz /usr/src`

Now, unpack both tarballs:
- `cd /usr/src`
- `tar –xvzf linux-2.6.7.tar.gz`
- `tar –xvzf rtai-3.1.tar.gz`

To save space, delete both `linux-2.6.7.tar.gz` and `rtai-3.1.tar.gz`.

Go to the new `/usr/src/linux-2.6.7`-directory: `cd /usr/src/linux-2.6.7`

Patch the kernel by typing: `patch -p1 < /usr/src/rtai-3.1/rtai-core/arch/i386/patches/hal7-2.6.7.patch`.

Now, the kernel has to be configured by using the graphical menu: type ‘make menuconfig’.

Use the configuration file on the Linux software CD by choosing ‘Load configuration from file’. The right path is `/dev/cdrom/kernel-2.6.7-adeos.config`

Close menuconfig (save and exit).

Type ‘make’ at the command line.

After this type ‘make modules_install install’ (maybe a ‘make clean’-command is needed before doing this) and answer ‘yes’ to run LILO.

Now, the new kernel image will be located in the `/boot`-directory. You have to delete the symbolic link to the old kernel image and make a new symbolic link to the kernel that has just been built.

Go to the root-directory where a file called ‘vmlinuz’ is located by typing ‘cd /’ and then you have to delete this file: `rm vmlinuz`.

Now, create a new symbolic link: `ln –s /boot/vmlinuz-2.6.7-adeos vmlinuz`.

Re-run LILO by typing ‘lilo’ at the command prompt.

Restart the system and the new kernel is in operation!
You can check the kernel version that is currently running by typing `uname –a`, a line starting with ‘Linux debianpc104 2.6.7-adeos’ should be printed on the screen.

### §3.3 RTAI installation

- Go to the `/usr/src/rtai-3.1` directory
- RTAI can be configured now, by typing `make menuconfig`.
- Set the correct path to the Linux kernel source: `/usr/src/linux-2.6.7`
- No settings have to be changed, so save and quit from this menu.
- Now type ‘make’ at the command line.
- After the make program has finished type ‘make install’.
- Several warnings will appear on the screen, just ignore these!
- If no errors will be reported, the installation is complete.
- To test whether everything works, there is a test-file in `/usr/realtime/testsuite/kern/latency`. Go to this directory and type `./run` and to stop this test, type: ‘Ctrl C’. If no errors are printed on the screen the installation is successful.
Chapter 4: Driver software

All the PC/104 cards can be controlled via software, either by use of the Universal Driver Software (provided for the Diamond cards) or by use of direct I/O permission (custom written software). The direct I/O c-files are explained here for the 3 different modules, because then it becomes clear which steps have to be taken in order to read from or write to a PC/104 board. The use of direct I/O is advantageous because of real-time operation. Software that uses the Universal Driver makes calls to a driver called dscud that cannot be used within the RTAI-framework (no direct I/O access in the software). The C-files written in both user mode (direct I/O) and with use of the Universal Driver Software can be found in appendices B and C, respectively.

§4.1 IO.c for the Ruby-MM

The software for the Ruby-MM card will take care of the digital inputs as well as the digital and analog outputs. All these operations are programmed into a single file (every card has its own c-file). The c-file will be implemented in a standard s-function template for use in Simulink. In order to use direct I/O one has to make use of the ‘ioperm’-function (together with the sys/io.h-header file). After some declarations are made and the dimensions of the in- and outputs of the s-function are provided the initialization part of the c-file can be programmed. A suitable place to do this is in ‘mdlStart’ in the s-function structure, this part will be executed only once and this is what you want with initialization like board settings, etc.

The first part of the initialization is to set the I/O properties of the Ruby-MM with the ‘ioperm’-function. ‘ioperm’ needs three arguments: the base address to access, the locations the board occupies with all its I/O registers and to give access (Boolean, true: 1) or to remove access (Boolean, false: 0). In this case the ‘ioperm’-function will be used as: ‘ioperm (0x380, 16, 1 )’ specifying the base-address at 380 Hex, 16 I/O registers and give access to these addresses. ‘ioperm’ returns 0 if the I/O permission is succeeded so one can add a print-statement for the case that the I/O permission fails.

After the I/O properties have been set the inb( ) and outb( ) statements to read from or write to a hardware-address can be used, respectively.

The use of digital in- and outputs is quite difficult for the Ruby-MM, but it works quite similar to the DIAMOND-MM-32-AT, which has a better manual. This board contains a 82C55 chip for digital I/O operation that provides three 8-bit ports, called A,B and C for a total of 24 I/O lines. To specify which ports will be used for digital inputs and which for digital outputs, one has to write a configuration-byte to the base-address+15. In our case we chose to use port A as input and both port B and C as output ports. These settings can be written to Base+15 with ‘outb( 0x90, Base+15 )’, 0x90 is the configuration-byte which can be found in the table of the Ruby-MM User Manual on page 20. After the ‘mdlStart’-function is completed a print-statement will be displayed. Some parts of the c-file are placed between #ifndef MATLAB_MEX_FILE and #endif statements. This is to prevent error-messages when mex-ing your c-file. These statements will be ignored when the executable is created in the build process.
Now, the most interesting part of the function, namely the ‘mdlOutputs’ part will be discussed. This is where the outputs will be read from Simulink and sent to the hardware-addresses and the inputs will be read from the hardware-addresses and sent to Simulink.

First, consider the analog-outputs.
The voltage that must be written to the Ruby-MM comes as an input to the s-function block, so this is an input in Simulink but an output to the PC/104 card. This voltage must be converted to a D/A code with a formula that corresponds with the board configuration (see page 14 of the manual). In our case we set the board to bipolar and full-scale (+/- 10V). For the Ruby-MM one register is provided to hold the Least Significant Byte (or LSB) data for all analog outputs. This register is located at address Base+8. To calculate the LSB, use ‘LSB=DATA & 255’ which strips off the highest 4 bits and keeps only the lowest 8 bits for output to the LSB register. After this the Most Significant Byte (or MSB) must be written to the corresponding address (each channel has its own MSB register). To calculate the MSB, use ‘MSB=(DATA & 3840)/256’ which strips off the lowest 8 bits and keeps the 4 highest bits and shifts them into the low 4 bits for output to the board. Now the LSB and MSB can be sent to Base+8 (LSB) and Base+i (MSB, placed in a for-loop for 8 output channels). All the channels can be updated at once by reading from Base+8, so this statement is placed outside the for-loop.

The procedure for the digital outputs is the following. In the for-loop inputs 9-16 of the s-function block are scanned and for every step of the loop the input (0 or 1) is controlled by the bitout-variable and this value is stored in the u-variable. The output bit will be stored in the correct location of the u-variable, corresponding to the count of the loop-variable i. After the 8 output-bits are collected, the variable u will be sent to Base+13 with the statement ‘outb( u, Base+13 )’. After this, the u-variable is set back to 0 and the loop starts again.

The digital input byte can be read out from the hardware with the command ‘inb( Base+12 )’ and this value is attributed to the variable bytein. For each step of the for-loop the value of bytein is compared to a predefined mask to sort out which of the eight bits in this byte is equal to one and by doing so, these bit-numbers are known: Y=( bytein & mask[i] ) >> i. The value of each bit will be written to the corresponding output of the s-function block with the command ‘y[i]=Y’.

Once the file is stopped by the user, all the analog outputs must be set back to zero. A correct place to do this is in the ‘mdlTerminate’ part of the s-function structure. The commands in this function are performed at termination of a simulation. The A/D code for a voltage of 0V (2048) will be written to the addresses corresponding to all the analog output channels in the same manner as the ‘mdlOutputs’ part of this s-function. For this function a print-statement will also be shown on the screen indicating that the board will be cleared.

§4.2 Analogin.c for the Diamond-MM-16-AT

The c-file for the DIAMOND-MM-16-AT 16-bit analog I/O PC/104 Module consists of reading out the analog signals for the ultrasonic sensors, microphones, pet-safe system and the LDR. After all the necessary include- and define-statements (10 outputs will be used for the s-function block), the standard s-function methods can be used by setting the correct in- and output properties and by setting the sample-time to inherited.
The ‘mdlStart’ part of the c-file is used for initialization. Again, the ‘ioperm’-function will be used to gain direct access to the hardware. The base-address for this board is 300 Hex and it has 12 registers, so up to and including Base+11 will be used (see page 9 of the manual for the I/O map). The ‘ioperm’-function thus must look like this: ioperm( 0x300, 12, 1 ). There are a couple of steps that have to be followed in order to read an analog input. First of all a FIFO-reset has to be done by writing 0x80 to Base+10, so: outb( 0x80, Base+10 ). This value 0x80 corresponds to the decimal value of 128, which is the same as $2^7$, so the eighth bit (starting from 0!) of Base+10 will be set.

Next, one has to select the 10 channels (0-9) that will be used and configure this by writing to the register located at Base+2. Thus, the lowest channel to be used is channel 0 and the highest channel is channel 9. The register is split in two parts, the lowest 4 bits (bits 0-3) are used to set the lowest channel and the highest four bits (bits 4-7) are used to set the highest channel. The lowest four bits will be all zeroes, corresponding to channel 0 and bits 4-7 have to be equal to the decimal value of 9. This can be done by setting the bit-pattern to $1001_2=2^3+2^0=8+1=9$. So the byte that will be written to Base+2 will look like 10010000. This is the same as the decimal value $2^7+2^0=144$. This decimal value has to be converted to a hexadecimal value to use it in the outb-statement. A value of 144 is the same as $9\times 16^1+0\times 16^0$, so 90 Hex. To set the input-channels the following statement has to be used: outb( 0x90, Base+2 ).

The next step is to set the input range to bipolar and to full-scale (+/- 10V). The register located at Base+11 takes care of this and only the third bit of this register has to be set (see manual on page 18). A decimal value of $2^3=8$ is the same as 0x08 in the hexadecimal number system ($8\times 16^0$). So, to set the input range, we use: outb( 0x08, Base+11 ). After this, wait for the analog input circuit to settle before continuing. This can be done with the following statement: while( inb( Base+10 ) & 0x80), this tells that an A/D conversion can only be performed when the wait-bit (bit 7) =0. This completes the ‘mdlStart’-function.

The ‘mdlOutputs’-function calculates the analog outputs and writes these to the outputs of the s-function block. To scan all 10 channels the A/D scan mode has to be enabled with the command outb( 0x10, Base+10 ) which sets the fourth bit of this register (Base+10). Then the A/D conversion can be started by writing a value (value does not matter) to Base+0. So, ‘outb( 0x00, Base+0 )’ is a software trigger. In front of the for-loop you have to place the same while-loop as in the ‘mdlStart’-function to wait for the A/D busy bit before continuing. There is only one LSB and one MSB register for this board, which are controlled by the board itself to generate the correct output for each channel. Now, the data can be read from the board separated into an LSB and an MSB: ‘LSB= inb( Base+0 )’, ‘MSB= inb( Base+1 )’. The LSB and MSB have to be combined to a 16-bit value. This is done by the following pseudocode: ‘DATA= (MSB*256+LSB)’. One can also choose to shift the MSB eight places to the left, which is the same as multiplying by 256. After computing this A/D code, we have to convert it to a meaningful value. Here something strange happens because the A/D code that comes from the different channels lies between 0 and 65536, which is strange because it should lie between $-32768$ and $32767$ (because of the bipolar setting of the board and the sign-bit). So we have to make an adjustment to correct this. If the A/D code is less than or equal to 32767 the voltage that will be sent to the output of the s-function is: $y[i] = \frac{DATA}{32768} \times 10.0$. On the other hand, if the A/D code is higher than 32767
(corresponding to a negative voltage), the output voltage will be:

\[ y[i] = \frac{(65536 \cdot DATA)}{32768} \cdot 10.0 \]

With these adjustments the correct voltages will be computed.

In the ‘mdlTerminate part’, only a print-statement is needed indicating that the program has been stopped by the user after stopping the real-time code in the ‘External mode control panel’ that will be discussed later on.

§4.3 Encoder.c for the Mesa-4I30

On the halobject, 7 motors are fixed and to control these we have to know their encoder positions. Each MESA 4I30 quadrature counter PC/104 board can handle 4 encoders. This s-function does not use muxed signals (in contrast with the other c-files) but one separate line for each channel (see the ‘mdlInitializeSizes’-function). Again, we use direct hardware access via the ‘ioperm’-function. The base-addresses for the two counter cards are set with the jumpers and for the first card this address is 200 Hex. The I/O register occupies 8 contiguous bytes (BASE+0 till BASE+7), so the statement for direct I/O permission in the ‘mdlStart’-function looks like this: ioperm(0x200, 8, 1). Then, some more initialization steps have to be taken, like clearing the counters, turn the led on and wait while the circuit settles: ‘outb( 0x90 | 0x20, Base+6 )’ and ‘while( inb( Base+6 ) & 0x80 )’ as in the analogout c-file. Next, writing 0x0F or 15 as decimal value to Base+4 will enable the counters (15 is equivalent to setting bits 0-3 in this register). The initialization will be finished with the display of ‘Counters enabled’.

There is a slight difference in the ‘mdlOutputs’-function of this c-file compared to the other ones because of the use of separate lines, that’s why the ‘ssGetOuputPortRealSignal(S,i)’-statement is inside the for-loop. In the table on page 15 of the manual for this board one can see which commands have to be issued to read and clear counters. For the first counter (counter 0) the command byte is 84H, so we have to write this to the command register located at Base+6: outb( 0x84, Base+6 ). After this, wait for the busy bit to be clear (bit 7 of this register). The number that will be read from the board is a 32-bit value, consisting of 4 times 8 latched counter bits. These 4 bytes have to be collected and combined to a 32-bit value. Shifting the different bytes in the 32-bit value by the correct number of places does this. For example the MSB (read from Base+3) must be shifted 24 places in order to put it in the right place. The four bytes are then combined to a 32-bit value that is assigned to the variable b and this value will be sent to the corresponding s-function output-channel.

No termination, except from a handy print-statement (‘Function stopped’) is needed.

The second MESA-card must use a different hardware-address, in this case the address is set to 210 Hex (which leaves enough space between the two boards, 8 registers used for every card).

Then there is a remark to be made, namely the use of different variable names, especially for the Base-addresses. If you don’t give them different names, one card will not work so take care of this!
§4.4 Use of the Universal Driver Software

Before the direct I/O software was used, c-files were created with the help of the Universal Driver Software, which is provided for the Diamond product range. In these files one can make use of predefined functions like ‘dscInit’, ‘dscInitBoard’, ‘dscDASettings’, ‘dscDADeSettings’, ‘dscDIOSetBit’, ‘dscDIOClearBit’, ‘dscDIOInputBit’, ‘dscADSettings’ and ‘dscADScan’. These functions are very easy to use, but because of calls to files that define these functions (dscud driver) this code can’t be used for real-time implementation, as discussed before. However, if someone wants to run an executable in soft real-time this is also possible. To do so, one has to install the universal driver software:

? Copy the driver software to /usr/local (tarball named dscud-5.8.tar.gz).
? Then unwrap this file by typing: tar -xvzf dscud-5.8.tar.gz and the directory dscud-5.8 will be created.
? Go to the dscud-5.8 directory.
? Run the ‘install.sh’ script by typing: ./install.sh.
? After this run the ‘load.sh’ script by typing: ./load.sh then the module will be loaded, you can check this by typing ‘lsmod’

In order to use these files, one also has to install a different timer, designed by René v/d Molengraft. The settings in your Simulink model remain unchanged with respect to a model for real-time purposes.

§4.5 WinTarget real-time target for Matlab/Simulink/RTW

WinTarget is a real-time target for use with Matlab/Simulink/RTW. In combination with the files described in the next two paragraphs, one can create Simulink models that can be used for real-time purposes, both soft and hard real-time.

§4.5.1 Soft real-time based on RTC

In order to make use of the Universal Driver Software that is provided with the Diamond products (§4.4) in real-time programs one needs to install a timer as described below. After this is done, the programs will run in so-called soft real-time mode, which is based on the Real Time Clock. The driver that comes with the Diamond products, called dscud must be included in the c-code of the s-functions (see appendix C). Go to the /home/mmolengr/TUeDACS/Timer-directory on the PC (or use the Linux-software CD). Copy the following files to a directory on the PC/104 system:

- buildlib (executable)
- buildapp (executable)
- test_timer (executable)
- libtimer.c
- libtimer.o
- timer.h
- libtimer.h
- test_timer.c

The program ‘buildapp’ supposes the following files to be located in the /usr/lib-directory:
- libtimer.so
- libtimer.so.1
- libtimer.so.1.0

Now, run the buildlib-program (by typing ‘./buildlib’) and buildapp (‘./buildapp’). One can test whether the timer works by running the test file ‘test-timer’ (type: ./test-timer).

§4.5.2 Hard real-time based on RTAI-LXRT

LXRT is an extension to RTAI that makes it possible to run real-time programs in usermode. So by installing this software, c-files can be used that gain direct hardware-access. Then it’s possible to use the Linux-command ‘ioperm’ to set the Hex-addresses and the amount of registers for each PC/104-card. Also the ‘inb’ and ‘outb’-statements can be used to read from and write to the different Hex-addresses directly from the c-code as is described in paragraphs 4.1, 4.2 and 4.3.

To switch from soft real-time to the use of hard real-time applications, the relevant files are located in the /usr/src/rtai_rene-directory (or use the Linux-software CD). Just run the buildlib-program and everything should work again. There is an exception to this, if Matlab has been (or will be) re-installed, you need a new version of wt_main.c (also on the Linux-software CD) that is located in the /usr/local/matlab6p1/rtw/c/wintarget-directory (so just delete and replace the file).
Chapter 5: Running a Simulink model on the PC/104-system

Several steps have to be taken before a file can be run to drive the halobject. The most important thing is to make a connection between the PC/104 system and the PC (with Matlab installed) via an UTP-cable. When this works correctly (check by ping-ing) software needs to be configured to log on to the PC/104 system from the other PC. Here, software called ssh (secure shell) is used, which is a program to log into another computer over a network, to execute commands in a remote machine and to move files from one machine to another. This program provides strong authentication and secure communications over unsecure channels.

§5.1 SSH-configuration

First of all, turn on the red switch to start the PC/104-system and log on to this as root (Password is: rorapi). Now, turn on the other PC and also log on to this one as root (Password is: PydD67nB) After startup we need to configure the IP-address for the PC (the PC/104 had its IP-address configured as 192.168.1.1 during Debian installation):

```
halobject:/> ifconfig eth0 192.168.1.2
```

Now the connection can be checked by typing:
```
halobject:/> ping 192.168.1.2
```
The output then will look like this:
```
PONG 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl= 64 time= 0.5 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl= 64 time= 0.4 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl= 64 time= 0.4 ms
(time: CTRL C)
--- 192.168.1.1 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip  min/avg/max = 0.4/0.4/0.5
```
Below a sample session is displayed to configure ssh with the input in bold.

```
halobject:/> ssh-keygen –t rsa
```
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa.pub): [RETURN]
Enter passphrase (empty for no passphrase): rolframon
Enter same passphrase again: rolframon
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is: ....

One can change the passphrase at any time by using the –p option of ssh-keygen.
To allow access to a system for a given identity place the public key in your
`~/.ssh/authorized_keys` file on that system:

```
halobject:# cd /root/.ssh
halobject:# cp id_rsa.pub authorized_keys
```

Now you could copy the `~/.ssh/authorized_keys` file to the PC/104-system to allow
access from the PC:

```
halobject:# scp -p ~/.ssh/authorized_keys 192.168.1.1:.ssh/
192.168.1.1's password: rorapi
authorized_keys 100% 1839 1.2MB/s 00:00
```

After this, you can log on to the PC/104 system:
```
halobject:# ssh 192.168.1.1
root@192.168.1.1's password: rorapi
```

The log on message will be printed on the screen followed by the prompt:
```
debianpc104:~
```

So now it’s possible to work on one system and control two systems simultaneously.

### §5.2 Using Matlab/Simulink with WIntarget

The basic configuration is completed, so let’s concentrate on the use of Matlab/Simulink
now. Go to the `/usr/src/rtai_rene-directory`:
```
halobject:# cd /usr/src/rtai_rene
```

Next, start Matlab:
```
halobject:/usr/src/rtai_rene# ./ml2
```
This makes sure the correct modules will be loaded (simply ignore the errors that will
appear on your screen).

Matlab 6.1 will start and a Simulink model can be created.

If c-code needs to be implemented into a Simulink-model, s-functions come in
handy. The c-code has to be placed into an s-function template (this can be found in the
MATLAB6p1/Simulink/src-directory). After this is done, save the file as a .c-file. This
file must be mex-ed, do this by typing: `mex filename.c` at the Matlab-prompt. Don’t
forget to do this after every change in the s-function!

The Simulink model needs to have the right configuration to make sure it will be
built correctly and will be able to run on the PC/104-system. First, the simulation
parameters must be set (see figure 15). Enter a stop time that is long enough for the
application. Set the solver options to Fixed-step, ode1(Euler) and the fixed step time will
be the sample time under which the application will run (for example 0.001). Click the
Workspace I/O-tab and turn everything off, otherwise the Compact flash-card (where the
executable will run) will be flooded with data. Now, the settings for the Real-Time-
Workshop have to be configured, use the wintarget.tlc-file as system target file then click
the ‘Wintarget code generation options’ and make sure the ‘External Mode’ box is
checked on.
Figure 15: Simulink-model configuration

In the Simulink model go to the ‘tools’ menu and click ‘External mode control panel…’. A window will open as shown in figure 16.
Click ‘Target interface…’ Now type the IP-address of the PC/104-system on which the
executable will run as a MEX-file argument, so type: ‘192.168.1.1’.
Also make sure the file will run in external mode instead of normal mode.

Once you’ve completed your Simulink model it can be built by typing ‘Ctrl B’
that will start the build process. If all went well an executable will be created.
This executable must be copied to the PC/104 system (make sure to log on to it with ssh
as root). The following illustrates this and the example is based on an executable called
looptest3 that has been created in the /root/Sprieten-directory on the PC and on copying
the executable to the test-directory of the PC/104-system:

debianpc104:~ # cd /test
debianpc104:/test# scp 192.168.1.2:/root/Sprieten/looptest3 .
root@192.168.1.2’s password: PydD67nB
looptest3 100% |***************| #filesize# 00:00 (transfer time)

To start the executable in external mode:

debianpc104:/test# ./looptest3 –w
Warning: TUEDACS has not been defined
Wintarget v3.0 RTA started
Counters enabled (print-statements defined in our s-functions)
Ruby-mm initialized
Waiting for Ext-mode start.
Now, go to ‘Tools’ and click ‘External mode control panel’ (as shown in figure 16). Connect the model, a message (Parameter changes pending) will appear in blue. After this we can start the Real-time code by clicking this button. In the console, the following will be printed:

Rtai-timer running at 1000.000000 Hz

To stop the executable, click ‘Stop real-time code’ and the output on the screen will look like this:

Rtai-timer reports:
Missed interrupts: 0.000% (0 out of …)
Waiting for Ext-mode to end…
Ext-mode shutdown.
Function stopped
Board clearing

The final two print-statements confirm that the model has been terminated properly (these statements were written in the ‘mdlTerminate’-part of the s-functions, which is the last step of running an executable of this kind).

§5.3 Installation of mingetty

No keyboard or screen can be connected to the PC/104-system so it would be very handy to log into this system automatically. A way of doing this is with the use of ‘mingetty’. This program has been installed on the Compact flash card:

? Copy mingetty-1.06.tar.gz from the CD labeled ‘Linux software’ to the /home-directory on the PC/104 system.

Since there is no cdrom-player connected to the PC/104-system, the crdrom-player of the other PC is needed to transfer the tar-file with the ‘scp’-command (after logging on with ‘ssh’ as described earlier).

? Go to the /home-directory on the PC/104-system and unpack this file.
? A directory called ‘mingetty-1.06’ will be created, go to this directory and type ‘make’ and ‘make install’. This installs the ‘mingetty’-program.
? Now the file named ‘inittab’ needs to be edited which is located in the /etc-directory. The easiest way of doing this is to use ‘Midnight Commander’ (by typing ‘mc’) or a text editor on the PC. So, copy the ‘inittab’-file to the PC and edit the file:

Originally the part of the file that needs to be edited looks like this:
1:2345:respawn:/sbin/getty 38400 tty1
2:23:respawn:/sbin/getty 38400 tty2
3:23:respawn:/sbin/getty 38400 tty3
4:23:respawn:/sbin/getty 38400 tty4
5:23:respawn:/sbin/getty 38400 tty5
6:23:respawn:/sbin/getty 38400 tty6

Use the autologin-option (as root) of ‘mingetty’ in the first line:
1:2345:respawn:/sbin/mingetty -autologin root tty1
2:23:respawn:/sbin/getty 38400 tty2
3:23:respawn:/sbin/getty 38400 tty3
4:23:respawn:/sbin/getty 38400 tty4
5:23:respawn:/sbin/getty 38400 tty5
6:23:respawn:/sbin/getty 38400 tty6

? Then copy this edited file back to the /etc-directory on the PC/104-system (simply overwrite it).

Now, the system will automatically log on as root after rebooting.
Chapter 6: identification&control of motions

§6.1 Transfers
For the transfers of the tilt mechanism, the turning legs and the feeler mechanisms, a Sensitivity measurement has been performed to identify the behaviour of the different motions.

§6.1.1 Walking mechanism
In figure 17 the transfer from input voltage to position of the tilt mechanism is displayed.

![Graph](image)

*Figure 17: transfer of the tilt mechanism*

The figure shows a fine -2 slope, which indicates pure mass displacement. This is correct, because of the used cross-joints with rubber connection rings. Through the joint only low-frequent signals are left, because the rubber rings act as low-pass filters, so no resonances are measured. A sensitivity-measurement has been performed which means that the transfers are measured in closed-loop (for safety reasons). So, a feedback loop is used and therefore a controller should be used (in this case chosen to be unity). This
controller needs some calculation time to convert the input (error) into an output (voltage). This voltage will then be converted from a digital to an analogue signal. This D/A conversion introduces a zero order hold effect. Both the calculation time and zero-order hold introduce a delay that can be seen in the phase-plot (see figure 17, for example). The delay time is:

\[ T_d = T_c \times \frac{T_s}{2} \]

with \( T_c \) the calculation time and \( T_s \) the sample-time. The second part of the delay-time is caused by the zero-order hold effect. The total delay-time introduces an extra transfer function: \( H(j\omega) = e^{j\omega T_d} \). This causes a phase loss (in degrees):

\[ \text{Phase}(H(j\omega)) = 360 \times T_d \times f \], so the higher the frequency the larger the phase loss.

For the turning legs the transfers are measured in the same way. These transfers show great resemblance with the tilt-transfer. They also have a -2 slope and no resonances, because here the same type of cross-joints is used (see figure 18). Also the effect of delay can be seen.

![Figure 18: transfers of the turning legs](image)

With the help of these transfers and diet (a software tool to create controllers quickly), controllers have been designed. The controllers are lead-filters with a bandwidth chosen to be 10 Hz. A higher bandwidth is possible, but no high-performance will be asked of the controllers so this is not necessary. These controllers have to be converted to discrete domain with a sample frequency of 1000 Hz (sample frequency of the program) to use them in the Simulink model:
For the turning legs the same controller is used because of the resemblance in transfer. Additional feedforward has been tuned over these controllers to decrease the tracking-error. For the tilt-mechanism an additional gravitation-feedforward is used, because of pure statically lifting the weight. The finally obtained tracking error was in the same order of magnitude as the resolution of the encoders, after a test with the object hanging in the air. The response will be different when the object walks, but further optimisation is not really necessary.

§6.1.2  Feeler mechanisms

The transfers (from input voltage to position) of each motion of the feeler mechanisms have also been measured and the result of this can be seen in figure 19.

\[ C_{\text{tilt}}(z) = \frac{644z + 632}{z + 0.8282} \]

\[ C_{\text{turn}}(z) = \frac{6900z + 6771}{z + 0.8282} \]

Figure 19: transfers of the two motions

Again the delay can be seen in figure 19, caused by the same effects as described in paragraph 6.1.1.

Motion controllers, based on the same assumptions as the turn- and tilt mechanisms, have been created:
\[ C_1(z) = \frac{124.3 - 121.8}{z - 0.8282} \]
\[ C_2(z) = \frac{66.31 - 64.94}{z - 0.8282} \]
§6.2 Control software

To create all the motions in order to move the halobject and its feelers, software has been developed. Here, the software that drives the walking and feeler mechanisms is discussed, pointing out all of its important topics. This has to be studied carefully for a safe operation of the halobject.

§6.2.1 Walking

With the help of the designed controllers a reference trajectory can be followed. For the ability to walk, three accurate trajectories have to be generated. This is done with the “Ref3 reference generator” (by René vd Molengraft). Finally for the walking mechanism a complete Simulink model has been designed (see figure 20).

![Simulink model for the walking mechanism](image)

The light-blue blocks are the links to the data-acquisition boards of the pc/104 and the dark-green block contains the controllers. The light-green block contains the reference trajectories of which the first path drives the tilt-mechanism and the second and third path drive the turning legs. The second path is used for walking in a straight line and the third for turning the object (see figure 21). Before the walking mechanism can be started, the turning legs have to be initialised. This initialising-program is located in the white subsystem. The legs are driven from the centre to the most right position by paths, which
are generated by look-up tables. After initialising, the current position is stored and can be used by other sections of the program. Now, the legs have to be manually centred before initialising, in the future the “homing” program will do this all automatically. From the initialised position the object can start to walk. As the figure shows, the tilt mechanism tilts the object from one side to the other whereas the turning legs turn from most left to most right and back. By generating the trajectories it is very important to synchronise the ground (zero-line in figure 21) positions of the different paths, otherwise e.g. the legs can already be turning before the object is lifted from the ground.

Another important aspect is the maximum stroke the mechanisms can make from their centred position. The allowable stroke for the tilt is 2.5 rad in each direction and the allowable stroke for the turning legs is 0.35 rad in each direction. If the stroke exceeds the allowed value, the legs will turn in an inductive sensor or emergency switch.

Figure 21: reference trajectories

The grey blocks are used to manipulate the walk of the object. The “kantelalgoritme”-block is used to mirror the movement of the tilt-mechanism. When the movement is mirrored the object will walk backwards. The other two blocks are used to switch from the second trajectory to the third. By this alteration the object is forced to turn, because one turning-leg will not make the same stroke as the other anymore. To guarantee a smooth transition an intelligent switch is used in the grey blocks to change from one movement to the other.
Digital I/O can operate the whole program. This is the final goal of the program, because the console controls the pc/104 by digital I/O lines. In the program only one movement has been implemented. In the future the movement can be extended by several step-sizes and step-speeds. The “character” (a neural network) then is able to choose between several types of walking depending on sensor-inputs.

§6.2.2 Feelers

The feelers have to react on sounds produced in the surroundings. Three microphones that are fixed to the mantle of the halobject will receive the sound. Out of these microphones an angle will be computed to indicate which direction the sound comes from. The feelers then have to point to that direction. When the sound level stays below a predefined level the feelers will start their scan motion to ‘scan’ the surroundings searching for sound sources. To create these motions and crossing between the two modes, software has been developed in Matlab Simulink (figure 22).

![Figure 22: control-software of the feelers](image)

In this software the Ref3-generator by René v/d Molengraft is used to create smooth reference signals to prevent shaky motion of the feelers. Also two so-called s-functions are used that contain c-files taking care of the signal handling. This is required because of the fact that nothing should happen at the moment the file would be run unless some digital input will be provided. After this there will be a check for incoming angles out of the microphones. When this is the case, the feelers will go and point to the corresponding direction. When no sound has been received, the scan motion will be started. To create this, the jog mode of the Ref3-generator has been used which is a signal with constant velocity (‘ramp’ in Simulink). The slope of this signal establishes the speed of one
rotation. In this case the slope has been fixed at $\frac{2}{3} \text{[rad/s]}$, which means that one rotation lasts 6 seconds. The reference-signal for the x-motor is calculated by taking the sine of this signal and by multiplying this value by $\frac{2}{9} \times 25 \left( \frac{2}{9} \right)$ is the angle of the feeler with respect to its vertical which has to be 20 degrees or $\frac{2}{9} \text{[rad]}$ and 1:25 is the transmission of the worm-worm wheel set). There is a slight problem with the reference signal for the y-motor considering the cosine of the jog mode, which means that when the jog mode will be started the cosine of this signal will be 1. This indicates that the y-motor would get a step-function to $\frac{2}{9} \times 25 \text{[rad]}$ at t=0. To solve this problem, one axis of the Ref3-generator will be used to send the feeler controlled by the y-motor to $\frac{2}{9} \times 25 \text{[rad]}$ with a 3rd order reference signal, before using the jog mode. To make sure there will be a smooth transition from scan-mode to a position where the sound comes from and vice versa memory-block are used (colored purple in figure 22) to remember the last positions sent to the x- and y-motors. In the c-file called ‘test1a.c’ this information will be used to correct the new inputs. An example of a reference signal that can be generated is shown in figure 23.
The following is an explanation of this example.
The angle that has been computed out of the sound-levels of the microphones is equal to 0 at the start of simulation, so the scan-mode will be started. With a smooth reference-signal the y-motor is sent to \( \frac{25}{9} \) [rad] whereas the x-motor remains at 0 [rad]. Once this position is reached the jog mode will be started and the x-motor will start to move as well. At time 6 [s] an angle of 290 degrees comes out of the microphones, which will be converted to a correct reference for each motor:

x-motor: \( \sin(\frac{290}{360})? \frac{25}{9}? 8.2[rad] \)

y-motor: \( \cos(\frac{290}{360})? \frac{25}{9}? 2.98[rad] \)

The feelers move from their final remembered positions from scan-mode to the new positions in a smooth way. After a while the input angle will become zero again, so the y-motor will move from its current position to the starting position of the scan mode after which this scan mode can be started. This will repeat, provided that an angle of zero degrees will be the input when the microphones have detected no sound. More attention has to be paid to the calculation of an angle out of the three microphones, because of this and because of the limited speed of the feelers, so the input angles shouldn’t vary too fast. So, for example, every 5 seconds an input angle can be sent to this file which means that
the length of the buffer in the c-file belonging to the calculation of the direction has to be chosen carefully. A typical length of a reference signal out of the Ref3-generator is 2.6 seconds and the feelers also have to stop for a little while to make it look like the halobject stares at the person who made the sound.
Recommendations and things left to do

To make the halobject a self-operating robot, so it can be placed in the hall of the NPS-building, a lot of things still need to be done. Most of the hardware is working properly at this moment. Only the sensors in the mantle need to be connected. Now, a demo-program is used to make the halobject walk. In the future the walking software is dependent of the input from the ultrasonic sensors to make sure that the robot doesn’t collide with pillars, walls and doors that are present in the hall. One student (Matthijs Dolsma) is currently working on the ultrasonic sensors.

Most of the work that needs to be done will be concerning the software. Especially the character of the halobject is a challenging aspect. Klaas v/d Molen (k.v.d.molen@student.tue.nl) has designed a toolbox to create a neural network that will function as the brain of the robot. This network should be able to handle inputs (from LDR, microphones) in order to generate a setpoint for the movement. In the neural network different weights can be attributed to the different inputs to specify to which input it should react best. Besides this, software has to be developed to compute the direction out of three microphones. A study has been performed on this before, but it needs fine-tuning to meet the demands that have been given in paragraph 6.2. The feelers should be able to follow the references that will be computed out of the microphone software. Also for the LDR software needs to be written, but this won’t be too difficult. A lot of calibration will be needed before every part will work properly.

Of course, when every subsystem has been tested one single program needs to be built that controls all motions, sensors, character and possible failures. One more important aspect is the homing of all the motions to make sure of correct initialization. Currently only a demo version of the homing-procedure is implemented. The final program should then start up automatically when the system is powered on. This is the final goal of the project offering a great challenge to all students that will work on it.
References

Literature
[2] C handboek (Dutch translation of [1])
[3] Maxon motor catalogue '00/'01 and '03/'04

Websites
? http://halobject.wtb.tue.nl
? http://www.debian.org
? http://www.rtaiorg
? http://www.diamondsystems.com
? http://www.mesanet.com
? http://kimmo.suominen.com/docs/ssh
? http://www.versalogic.com
? http://www.maxonmotor.nl
? http://www.turck.nl
? http://www.sensor.nl
? http://www.agradi.nl
? http://www.conrad.nl
Appendix A: Motor calculation

Starting from the requirement to make a stroke of 40 degrees, the worm wheel has to turn
\[ \frac{40 \text{°}}{360\text{°}} = 0.7 \text{ [rad]} \] in 2 seconds. One period (round trip) lasts 4 seconds:
\[ T = 4[s] \]
\[ f = 0.25[Hz] \]
\[ \omega = \frac{\theta}{\tau} = \frac{0.7}{2}[\text{rad/s}] \]

A function that describes the angular displacement is:
\[ \theta(t) = 0.35\sin\left(\frac{\pi}{2}t\right) \] (The amplitude is half the stroke of 0.7 [rad])

Then, the angular velocity is:
\[ \omega(t) = \frac{\theta}{\tau} = 0.35\cos\left(\frac{\pi}{2}t\right) \]
And the angular acceleration:
\[ \alpha(t) = \frac{\omega}{\tau} = 0.35\sin\left(\frac{\pi}{2}t\right) \]

So, the maximum acceleration will be:
\[ \alpha_{\text{max}} = \frac{\pi^2}{4} \text{rad/s}^2 \]

An estimate for the moment of inertia \( J_{\text{load}} \) with respect to the center of rotation is:
\[ J_{\text{load}} = \frac{1}{2}mr^2 \]
where \( r \) is the distance between the center of rotation and the (new) center of mass. This distance has been estimated to be 0.15 [m] (according to the drawings).

With \( m \)? \( m_{\text{feeler}} \)? \( m_{\text{threaded shaft}} \)? \( m_{\text{compensate}} \)? 4[kg] (estimate, the feeler weighs 1 [kg], the threaded shaft weighs 0.4 [kg] and a mass is needed to balance the mechanism) the moment of inertia will become:
\[ J_{\text{load}} = 5\times10^{-2}[\text{kgm}^2] \]

The required torque at the shaft of the worm wheel then becomes:
\[ T_{\text{as..wormwheel}} \geq J_{\text{load}} \frac{\alpha_{\text{max}}}{2} \]
\[ T_{\text{as..wormwheel}} = 5\times10^{-2} \times 0.86 \times 0.043[\text{Nm}] \]

The power to do this:
\[ P_{\text{as..wormwheel}} = \frac{T}{\tau} s \geq \frac{0.043}{2} \times 0.0216[\text{W}] \]

The worm- worm wheel set which is used has a 25:1 transmission and an efficiency of 25\%, so the required power at the shaft of the worm will be:
\[ P_{\text{as..worm}} \geq \frac{0.0216}{0.25} \times 0.09[\text{W}] \]
The worm wheel makes \( \frac{40}{360} \) rotation, which means \( \frac{1}{9} \)? rotations of the worm in 2 seconds, which is the same as 83.33 [rev/min]. The ‘peak value’ of the mechanical power: 

\[ P_{\text{max}} \]  

\[ T_{\text{max}} \]  

\[ \theta_{\text{max}} \]  

\[ \frac{22}{60} \]  

\[ 1.6[W] \] with 

\[ T_{\text{max}} \]  

\[ P_{\text{all\_worm}} \]  

\[ \theta \]  

0.09 \? 0.18[Nm]

The gearing has to handle a torque of 0.18[Nm]. A desired rotational velocity of 6000 [rev/min] of the motor (according to Maxon catalog ’03-'04) leads to a transmission of 

\[ \frac{6000}{83.33} \]  

\[ 72:1 \] 

The maximum torque to be delivered by the motor is:  

\[ \frac{0.18}{72 \times 0.7} \]  

\[ 0.00357[Nm] \]  

\[ 3.57[mNm] \] (Small RE-motors have an efficiency of approximately 70%)

An RE 13 (3W) motor would suffice, knowing that this motor is capable of a maximum continuous torque of 3.08[mNm] and a stall torque of 10.8[mNm].

However, on this motor an encoder can be fixed with a resolution of only 256 CPT. So the decision has been made to switch to a RE 16 (4.5 W) motor. A gearing of 84:1 (possible because of the over-dimension) and a Digital MR encoder with 512 CPT come with this motor. The digital encoder also has a line driver to prevent it from missing encoder pulses over long wires.

A summary:  

- Motor: RE 16, 4.5 Watt (Graphite brushes)  
- Gearhead: planetary GP 16A (0.1-0.3 Nm) 84:1, metal version  
- Encoder: digital MR encoder with line driver, 512 CPT
Appendix B: C-files direct I/O

Rubydirect.c

/* Rubydirect.c version 28-10-2004 for diamond ruby-mm pc/104 board
   (c) Halobject-team
   R. de Jong
   R. Solberg
   */

#define S_FUNCTION_LEVEL 2      /* Level 2 for models, Level 1 for
   Backwards compatibility for Simulink 2.1
   and previous $functions */
#define S_FUNCTION_NAME  Rubydirect /*You must specify the
   $FUNCTION_NAME as the name of your $function. */

#include <stdlib.h>      /* for exit procedure */
#include <simstruc.h>    /* Need to include simstruc.h for the
   definition of the SimStruct and
   * its associated macro definitions. Note,
   * MathWorks specific head$ file that your $-
   function includes.*/
#include <tmwtypes.h>

#include <sys/io.h>
#include <math.h>

#define NINPUTS 16
#define NOUTPUTS 8

int ioperm(unsigned long, unsigned long, int );
int_T Basem2;

const mask[8]={1,2,4,8,16,32,64,128};

/*********************************
 * S-function methods *
 */

/* Function: mdlInitializeSizes
  =========================================
   * Abstract:
   * The sizes information is used by Simulink to determine the $ function
   * block's characteristics (number of inputs, outputs, states, etc.).
   */

static void mdlInitializeSizes(SimStruct *S)
ssSetNumSFcnParams(S, 0); /* Number of expected parameters */
if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
    /* Return if number of expected != number of actual parameters */
    return;
}

if (!ssSetNumInputPorts(S, 1)) return;
ssSetInputPortWidth(S, 0, NINPUTS);
ssSetInputPortDirectFeedThrough(S, 0, NINPUTS);

if (!ssSetNumOutputPorts(S, 1)) return;
ssSetOutputPortWidth(S, 0, NOUTPUTS);

ssSetNumSampleTimes(S, 1);

/* Function: mdlInitializeSampleTimes */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

/*=====================================================================
===== Function: mdlStart */
static void mdlStart(SimStruct *S)
{
    if (ioperm(0x380, 24, 1) != 0)
    {
        printf("I/O permission not succeeded\n");
        return;
    }
    #ifdef MATLAB_MEX_FILE
    Basem2 = ((int_T)0x380); /* adress for specific board*/
    /*configuration voor i/o chip */
    outb(0x90, Basem2+15); /* A=in  B,C =out */
    printf("ruby-mm initialized\n");
    #endif
}
/=====================================================================
=====
Function: mdlOutputs
Abstract:
In this function, you compute the outputs of your S-function block. Generally outputs are placed in the output vector, ssGetY(S).
=======================================================================
===*/
static void mdlOutputs(SimStruct *S, int_T tid)
{
#ifndef MATLAB_MEX_FILE
InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
int_T i, Y, u, bytein, bitout, DATA, LSB, MSB;
real_T *y = ssGetOutputPortRealSignal(S,0);
u=0;

for (i=0;i<8;i++)
{
    DATA = (*uPtrs[i]/10.0f)*2048.0f+2048;/* output voltages */
    LSB = ((DATA & 3840)/256);
    MSB = (DATA & 255);
    outb(LSB, Basem2+8);
    outb(MSB, Basem2+i);
    bitout = *uPtrs[i+8]; /* output bits */
    u= u | (bitout << i);
    bytein=inb(Basem2+12); /* input bits */
    Y=( bytein & mask[i] )>>i;
    y[i]= Y;
}
inb(Basem2+8); /* update values on board (output voltages) */
outb(u,Basem2+13); /* update values on board (output bits) */
#endif
} /* end mdlOutputs */

/*=====================================================================
=====
Function: mdlTerminate
Abstract:
In this function, you should perform any actions that are necessary at the termination of a simulation. For example, if memory was allocated in mdlStart, this is the place to free it.
=======================================================================
===*/
static void mdlTerminate(SimStruct *S)
{
    printf("board clearing\n");
}
#ifndef MATLAB_MEX_FILE
outb(1,Basem2+9); /* this command clears the board, setting all
values to 0 */

int_T i,LSB, MSB, DATA;

for (i=0;i<8;i++)
{
    DATA = 2048;
    LSB = (DATA & 255);
    MSB = ((DATA & 3840)/256);
    outb( LSB, Basem2+8);
    outb( MSB, Basem2+i);
}

inb(Basem2+8); /*update values on board*/

#endif
}

/*=============*
* Required S-function trailer *
*=============*/

#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-
file? */
    #include "simulink.c"    /* MEX-file interface mechanism */
#else
    #include "cg_sfun.h"    /* Code generation registration function */
#endif
Analogin.c

/* analogin.c version 29-9-2004 for Diamond-MM-16-AT 16-bit Analog I/O
 * The original file written to user-mode (direct I/O)
 * Copyright Halobject-team
 * R. de Jong
 * R. Solberg
 * Read analog Input signal from this module:
 * Outputs in Simulink:
 * y[0]-y[3]: Ultrasonic sensors (1-4)
 * y[4]-y[6]: Microphones (1-3)
 * y[7]-y[8]: Pet-safe
 * y[9]: LDR
 * Output: Voltages [range +/-10 V]
 */

#define S_FUNCTION_LEVEL 2      /* Level 2 for models, Level 1 for
                                Backwards compatibility for Simulink 2.1
                                and previous $functions */
#define S_FUNCTION_NAME analogin /*You must specify the
                                S_FUNCTION_NAME as the name of your $function. */

#include <stdlib.h>      /* for exit procedure */
#include <simstruc.h>    /* Need to include simstruc.h for the
 definition of the SimStruct and
 * its associated macro definitions. Note,
 simstruc.h should be the only MathWorks specific header file that your
 S-function includes.*/
#include <tmwtypes.h>
#include <sys/io.h>
#define NINPUTS 1
#define NOUTPUTS 10
int ioperm(unsigned  long,  unsigned  long, int );
int_T Basem3;
unsigned int LSB1;
unsigned int MSB1;
int_T DATA1;

/*==========================================*
 * S-function methods *
 *==========================================*/
/* Function: mdlInitializeSizes
 ********************************************
 * Abstract:
 * The sizes information is used by Simulink to determine the S
 function * block's characteristics (number of inputs, outputs, states,
 etc.).
 * /
 * static void mdlInitializeSizes(SimStruct *S)
 { ssSetNumSFnParams(S, 0); /* Number of expected parameters */
   if (ssGetNumSFnParams(S) != ssGetSFnParamsCount(S)) {

54
/* Return if number of expected != number of actual parameters */
return;
}
if (!ssSetNumInputPorts(S, 1)) return;
ssSetInputPortWidth(S, 0, NINPUTS);
if (!ssSetNumOutputPorts(S, 1)) return;
ssSetOutputPortWidth(S, 0, NOUTPUTS);
ssSetNumSampleTimes(S, 1);
}

/* Function: mdlInitializeSampleTimes
=========================================
* Abstract:
* This function is used to specify the sample time(s) for your
* S-function. You must register the same number of sample times as
* specified in ssSetNumSampleTimes.
*/
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

/*================================================================*****
***** Function: mdlStart
Abstract:
This function is called once at start of model execution. If
you
have states that should be initialized once, this is the place
to do it.
================================================================*****
===*/
#define MDL_START     /* Change to #undef to remove function */
#if defined(MDL_START)
static void mdlStart(SimStruct *S)
{
    if (ioperm(0x300, 12, 1)!=0) /* registers tot Base+11 */
        {printf("I/O permission not succeeded\n");
            return;
        }
#endif   MATLAB_MEX_FILE
Basem3 = ((int_T)0x300);    /*address for specific board*/
outb( 0x80, Basem3+10);    /*FIFO reset */
outb( 0x90, Basem3+2);    /* set input-channels (0-9)*/
outb( 0x08, Basem3+11);    /* set input-range to bipolar and full-
                        scale (+/- 10 V) */
while ( inb(Basem3+10)&0x80); /* perform an A/Bconversion only
                                   when
            WAITbit = 0 */
printf("Input-channels enabled (+/- 10V)\n");
#endif /* MEXFILE */
}
#endif /* MDL_START */

/*=====================================================================
=======
Function: mdlOutputs
Abstract:
In this function, you compute the outputs of your S-function
block. Generally outputs are placed in the output vector,
ssGetY(S).
=======================================================================
===*/
static void mdlOutputs(SimStruct *S, int_T tid)
{
    #ifndef MATLAB_MEX_FILE
    real_T          *y = ssGetOutputPortRealSignal(S,0);
    int_T           i;
    outb( 0x10, Basem3+10); /* enable scan (bit 4) */
    outb( 0x00, Basem3+0);/* Software A/D trigger, value does not
matter*/
    while (inb(Basem3+8)&0x80); /* wait for A/D busy bit (bit7) = 0 */
    for (i=0;i<NOUTPUTS;i++)
    {
        LSB1= inb(Basem3+0); /* Read LSB */
        MSB1= inb(Basem3+1); /* Read MSB */
        DATA1=(int_T) (MSB1*256+LSB1);     /* Combine LSB and MSB
to 16-bit */
        if (DATA1 <= 32767)
        { y[i] = (real_T) DATA1/32768 * 10.0;
        }
        else
        { y[i] = (real_T) (65536-DATA1)/32768 * -10.0;
        }
    }
    #endif
}
/* end mdlOutputs */

/*=====================================================================
=======
Function: mdlTerminate
Abstract:
In this function, you should perform any actions that are
necessary
at the termination of a simulation. For example, if memory was
allocated in mdlStart, this is the place to free it.

56
static void mdlTerminate(SimStruct *S)
{
    printf("Function stopped\n");
}

/*=============================*
 * Required S-function trailer *
 *=============================*/

#ifdef MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-
    file? */
    #include "simulink.c"    /* MEX-file interface mechanism */
#else
    #include "cg_sfun.h"     /* Code generation registration function */
#endif
#define S_FUNCTION_LEVEL 2      /* Level 2 for models, Level 1 for Backwards compatibility for Simulink 2.1 and previous Sfunctions */
#define S_FUNCTION_NAME encoder /*You must specify the S_FUNCTION_NAME as the name of your S-function. */

#include <stdlib.h>      /* for exit procedure */
#include <simstruc.h>    /* Need to include simstruc.h for the definition of the SimStruct and *its associated macro definitions. Note, simstruc.h should be the only *MathWorks specific header file that your S function includes.*/
#include <tmwtypes.h>
#include <sys/io.h>

#define PortsIn        0 /* # input ports */
#define MultiPortsIn   0
#define PortsOut       4 /* # output ports */
#define MultiPortsOut  1
#define NUM_PARAM      0 /*Number of parameters*/

int ioperm(unsigned  long,  unsigned  long, int );
int_T Basem;

/*===================================================================== 
Function: mdlInitializeSizes 
Abstract: 
The sizes information is used by Simulink to determine the S function block's characteristics (number of inputs, outputs, states, etc.).=====================================================================
static void mdlInitializeSizes(SimStruct *S) 
{
ssSetNumSFcnParams(S, NUM_PARAM);          /* Number of expected
parameters */
ssSetNumContStates(S, 0);                  /* Number of continuous
states */
ssSetNumDiscStates(S, 0);                  /* Number of discrete
states */

/* Set input-port properties */

/* Set output port properties */

if (!ssSetNumOutputPorts(S, PortsOut)) return;  /* If no ports,
return */
ssSetOutputPortWidth(S,0, MultiPortsOut);        /* Multiple ports
width port 1 */
ssSetOutputPortWidth(S,1, MultiPortsOut);        /* Multiple ports
width port 2 */
ssSetOutputPortWidth(S,2, MultiPortsOut);        /* Multiple ports
width port 3 */
ssSetOutputPortWidth(S,3, MultiPortsOut);        /* Multiple ports
width port 4 */

/* Set miscellaneous properties */
ssSetNumSampleTimes(S, 1);                   /* number of sample times */
ssSetNumRWork(S, 0);                       /* number of real work vector
elements */
ssSetNumIWork(S, 0);                       /* number of integer work vector
elements */
ssSetNumPWork(S, 0);                       /* number of pointer work vector
elements*/
ssSetNumModes(S, 0);                       /* number of mode work vector
elements */
ssSetNumNonsampledZCs(S, 0);               /* number of nonsampled zero crossings
*/
ssSetOptions(S, 0);                        /* general options (SS_OPTION_xx) */
}

/* end mdlInitializeSizes */

/*=====================================================================
===== Function: mdlInitializeSampleTimes
Abstract:
This function is used to specify the sample time(s) for your
S-function. You must register the same number of sample times as
specified in ssSetNumSampleTimes.
=======================================================================
====*/
static void mdlInitializeSampleTimes(SimStruct *S)
{
    /*Register one pair for each sample time */
    ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
}
ssSetOffsetTime(S, 0, 0.0);

/*end mdlInitializeSampleTimes*/

/*=====================================================================
=====
Function: mdlStart
Abstract:
This function is called once at start of model execution. If you have states that should be initialized once, this is the place to do it.
=======================================================================
===*/
#define RegLatch0 (Basem+0)
#define RegLatch1 (Basem+1)
#define RegLatch2 (Basem+2)
#define RegLatch3 (Basem+3)
#define RegEna (Basem+4)
#define RegSta (Basem+5)
#define RegCmd (Basem+6)
#define LedOn (0x20)
#define ClrAll (0x90)

#define MDL_START /* Change to #undef to remove function */
#if defined(MDL_START)
static void mdlStart(SimStruct *S)
{
  if (ioperm(0x200, 8 ,1)!=0)
    {printf("I/O permission not succeeded\n");
     return;
    }
  #ifndef  MATLAB_MEX_FILE
Basem = ((int_T)0x200);     /*adress for specific board*/
  outb(ClrAll | LedOn,RegCmd );   /* Clear All Counters, Led On */
  while (inb(RegCmd)&0x80) ; /* wait while busy */
  outb(0x0F,RegEna);     /* Enable All Counters inregister ENA0 */
  printf("Counters enabled\n");
  #endif /*  MEXFILE */
  
  #endif /* MDL_START */
}

/*=====================================================================
=====
Function: mdlOutputs
Abstract:
In this function, you compute the outputs of your S-function

60
block. Generally outputs are placed in the output vector, 
ssGetY(S).

static void mdlOutputs(SimStruct *S, int_T tid)
{
    #ifndef MATLAB_MEX_FILE
    real_T *y;
    int_T i,b;

    /*printf("Reading counters
");*/

    for (i=0;i<PortsOut;i++)
    {
        y = ssGetOutputPortRealSignal(S,i);

        outb(0x84+i,RegCmd);

        while (inb(RegCmd)&0x80) ; /* wait while busy */

        b= (inb(RegLatch3)<<24)
            | (inb(RegLatch2)<<16)
            | (inb(RegLatch1)<< 8)
            | (inb(RegLatch0)    );

        y[0]= (real_T) b;
    }
    #endif
}
/* end mdlOutputs */

/*====================================*
* Required S-function trailer *
*====================================*/

#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-
file? */
#include "simulink.c"    /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"     /* Code generation registration function */
#endif
Appendix C: C-files with Universal Driver Software

Aodio.c

/* aodio.c Analoge outputs en digital I/O op Ruby-MM
u[0]-u[7]: Analoge outputs (op kaartje = ingang van sfunction blok!)
u[8]-u[15]: Digital outputs
optioneel: u[16]-u[23]: Digital outputs
y[0]-y[7]: Digital inputs
*/
#define S_FUNCTION_NAME   aodio
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

#include "dscud.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <sys/timeb.h>
#include <time.h>

#define NINPUTS  16 /* verander eventueel in 24 indien port C
gebruikt wordt */
#define NOUTPUTS  8 /* Dit zijn dus in-en uitgangen van het s-function
blok! */
#define ERROR_PREFIX "RMM Driver ERROR:

#define BYTE unsigned char
#ifdef   MATLAB_MEX_FILE
/* var declarations */
static BYTE result;     /* returned error code */
static DSCB ds cb;      /* handle used to refer to the board */
static DS CCB dsc cb;   /* structure containing board settings */
static DSCDACS dscdacs; /* structure containing DA conversion settings */
static ERRPARAMS errorParams;  /* structure for returning error code
and error string */
static BYTE config_bytes; /* Configuration of input and output ports */
static BYTE digital_value; /* used by dscDIOInputBit function, see
mdlOutputs */
#endif

static int i;           /* miscellaneous counter */

/* Function: mdlInitializeSizes
=============================== */
* Abstract:
*   Setup sizes of the various vectors.
*/
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, 0);
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        return; /* Parameter mismatch will be reported by Simulink */
    }

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, NINPUTS);
    ssSetInputPortDirectFeedThrough(S, 0, NINPUTS);

    if (!ssSetNumOutputPorts(S, 1)) return;
    ssSetOutputPortWidth(S, 0, NOUTPUTS);

    ssSetNumSampleTimes(S, 1);
}

/* Function: mdlInitializeSampleTimes
========================================
* Abstract:
*    Specify that we inherit our sample time from the driving block.
*/
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}
#define MDL_START  /* Change to #undef to remove function */
#if defined(MDL_START)
/* Function: mdlStart
=======================================================================
==
* I. DRIVER INITIALIZATION
*
*    Initializes the DSCUD library.
*
*    STEPS TO FOLLOW:
*    
*    1. initialize the driver, using the driver version for validation
*/

static void mdlStart(SimStruct *S)
{

}
#ifndef MATLAB_MEX_FILE
if( ( result = dscInit( DSC_VERSION ) ) != DE_NONE )
{
    dscGetLastError(&errorParams);
    fprintf( stderr, "dscInit error: %s %s\n",
    dscGetErrorString(errorParams.ErrCode), errorParams.errstring );
    return;
}
printf("Driver Initialization completed\n");
/
/*=======================================================================
==
* II. BOARD INITIALIZATION
*
*    Initialize the RMM board. This function passes the
various
*    hardware parameters to the driver and resets the
hardware.
*
*    STEPS TO FOLLOW:
*
*    1. set the board type to DSC_RMM for RMM board
*    2. set the base address
*    3. set the interrupt level
*    4. initialize and register the board with the driver,
after which
*       the struct, dscb, now holds the handle for the
board
*======================================================================
==
/*

dscccb.boardtype = DSC_RMM;
dscccb.base_address = 0x380;
dscccb.int_level = 5;
config_bytes = (BYTE) 0x90; /* Configuration byte (defining
Digital Input and
       Output channels */
for (i=0; i<8; i++)
    { dscdacs.channel_enable[i] = TRUE;
    }

dscdacs.output_codes=(DSCDACODE*)malloc( sizeof(DSCDACODE)*8);
if( ( result = dscInitBoard( DSC_RMM, &dscccb, &dscb ) ) !=
DE_NONE )
{
    dscGetLastError(&errorParams);
    fprintf( stderr, "dscInitBoard error: %s %s\n",
    dscGetErrorString(errorParams.ErrCode), errorParams.errstring );
    return;
}
*/
printf("Board Initialization complete\n");
if( (result = dscDIOSetConfig(dscb, &config_bytes)) != DE_NONE) {
    dscGetLastError(&errorParams);
    fprintf(stderr, "%s %s\n", ERROR_PREFIX, errorParams.errstring);
    return;
}
printf("DIO Configuration complete\n");
#endif /* MDL_START */

/* Function: mdlOutputs
   ===================================
   * write voltage to 8 analog output channels
   *
   static void mdlOutputs(SimStruct *S, int_T tid)
   {
   real_T *y = ssGetOutputPortRealSignal(S,0);
   InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
   #ifndef MATLAB_MEX_FILE
   for( i = 0; i < 8; i++ )
   {
   dscdacs.output_codes[i] = (uint) ((*uPtrs[i]) / 10.0f * 2048.0f + 2048);
   } /* formula based on 10 V-full scale and bipolar, see documentation!*/
   if( ( result = dscDAConvertScan( dscb, &dscdacs ) ) != DE_NONE )
   {
   dscGetLastError(&errorParams);
   fprintf(stderr, "%s %s\n", ERROR_PREFIX, errorParams.errstring );
   free( dscdacs.output_codes ); /*remember to deallocate malloc() memory*/
   return;
   }
   if (*uPtrs[i+8] > 0)
   {if((result=dscDIOSetBit(dscb,1,i)) != DE_NONE)
   { dscGetLastError(&errorParams);
   fprintf(stderr, "%s %s\n", ERROR_PREFIX, errorParams.errstring );
   return;
   } /* if (*uPtrs[i+8] > 0) */
   */
} 
} 
else 
} if((result=dscDIOClearBit(dscb,1,i)) != DE_NONE) 
{ dscGetLastError(&errorParams); 
fprintf( stderr, "%s %s \n", ERROR_PREFIX, 
errorParams.errstring ); 
return; 
} 
} 
}

} if((result=dscDIOInputBit(dscb,0,i,&digital_value)) != DE_NONE) 
{ dscGetLastError(&errorParams); 
fprintf( stderr, "%s %s \n", ERROR_PREFIX, 
errorParams.errstring ); 
return; 
} 
else 
{ y[i]=digital_value; 
} 

} 
} 

*/

/* Function: mdlTerminate 
=====================================================================
* IV. CLEANUP
* Cleanup any remnants left by the program and free the
resources used
* by the driver.
* 
* STEPS TO FOLLOW:
* 
* 1. free the memory allocated for sample values
* 2. free the driver resources
=====================================================================

*/

#ifndef MATLAB_MEX_FILE
/* Set Vouts to 0 */
for( i = 0; i < 8; i++ )
{
    dscdacs.output_codes[i] = 2048;
    if( ( result = dscDAConvertScan( dscb, &dscdacs ) ) != DE_NONE )
    {
        dscGetLastError(&errorParams);
        fprintf( stderr, "%s %s\n", ERROR_PREFIX, errorParams.errstring );
        free( dscdacs.output_codes ); /*remember to deallocate malloc() memory*/
        return;
    }
    printf("\nSet Outputs back to 0\n");
    free( dscdacs.output_codes );
    printf("Memory for output-codes freed\n");
    dscFree();
    printf("DSCDAConvertScan completed.\n");
    printf("DSCDIOFunctions completed\n");
    return;
}
#endif

#ifndef MATLAB_MEX_FILE   /* Is this file being compiled as a MEX file? */
#include "simulink.c"    /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"    /* Code generation registration function */
#endif
Analogin.c

/* Analogin.c version 6-5-2004 for Diamond-MM-16-AT 16-bit Analog I/O
* PC/104 Module Universal Driver Software used:
http://www.diamondsystems.com/support/software
* Copyright Halobject-team
* R. de Jong
* R. Solberg
* Read analog Input signal from this module:
* Outputs in Simulink:
* y[0]-y[2]: Microphones (1-3)
* y[3]-y[6]: Ultrasonic sensors (1-4)
* Output: Voltages [range +/-10 V]
*/

#define S_FUNCTION_NAME Analogin
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"
#include "dscud.h"

#define NINPUTS 1
#define NOUTPUTS 7
#define BYTE unsigned char
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <sys/timeb.h>
#include <time.h>
/* diamond driver includes */

#define ERROR_PREFIX "DMM16AT Driver ERROR:"

/* var declarations */
#ifndef MATLAB_MEX_FILE
static BYTE result; /* returned error code */
static DSCB dsccb; /* handle used to refer to the board */
static DSCCB dsccb; /* structure containing board settings */
static DSCADSETTINGS dscadsettings; /* structure containing A/D conversion settings */
static DSCDASCAN dscdascan; /* structure containing A/D scan settings */
static WORD* samples; /* sample readings */
static FLOAT voltage; /* actual voltage */
static ERRPARAMS errorParams; /* structure for returning error code and error string */
#endif

static int i; /* miscellaneous counter */

/*=================================================================*
* S-function methods *
*===============================================*

/* Function: mdlInitializeSizes
===============================================
* Abstract:
*    The sizes information is used by Simulink to determine the S
*    function block's characteristics (number of inputs, outputs, states,
*    etc.).
*/

static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, 0); /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        /* Return if number of expected != number of actual parameters */
        return;
    }

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, NINPUTS);
    ssSetInputPortDirectFeedThrough(S, 0, NINPUTS);
    if (!ssSetNumOutputPorts(S, 1)) return;
    ssSetOutputPortWidth(S, 0, NOUTPUTS);
    ssSetNumSampleTimes(S, 1);
}

/* Function: mdlInitializeSampleTimes
=======================================
* Abstract:
*    This function is used to specify the sample time(s) for your
*    S-function. You must register the same number of sample times as
*    specified in ssSetNumSampleTimes.
*/

static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

#define MDL_START  /* Change to #undef to remove function */
#if defined(MDL_START)
/* Function: mdlStart
=======================================
* Abstract:
*    This function is called once at start of model execution. If you
*    have states that should be initialized once, this is the place
*    to do it.
*/

static void mdlStart(SimStruct *S)
{
}
I. DRIVER INITIALIZATION
* Initializes the DSCUD library.
*

II. BOARD INITIALIZATION
* Initialize the DMM-16-AT board. This function passes the various hardware parameters to the driver and resets the hardware.
*

III. AD SETTINGS INITIALIZATION
* Initialize the structure containing the AD conversion settings and then pass it to the driver.
*

---

```c
#ifndef MATLAB_MEX_FILE
if( dscInit( DSC_VERSION ) != DE_NONE )
{
    dscGetLastError(&errorParams);
    fprintf( stderr, "%s %s
", ERROR_PREFIX, errorParams.errstring );
    return;
}
printf("Driver Initialization complete\n");

dsccb.base_address = 0x300;
dsccb.int_level = 9;
if(dscInitBoard(DSC_DMM16AT, &dsccb, &dscb)!= DE_NONE)
{
    dscGetLastError(&errorParams);
    fprintf(stderr, "%s %sfn", ERROR_PREFIX,
    errorParams.errstring);
    return;
}
printf("Board Initialization complete\n");

dscadsettings.range = RANGE_10;
dscadsettings.polarity = BIPOLAR;
dscadsettings.gain = GAIN_1;
dscadsettings.load_cal = (BYTE)TRUE;
```
dscadsettings.current_channel = 0;

if( ( result = dscADSetSettings( dscb, &dscadsettings ) ) != DE_NONE )
{
    fprintf( stderr, "%s\n", ERROR_PREFIX, dscGetErrorString( result ) );
    return;
}
printf("AD Settings Initialization complete\n");

/*========================================================================
* IV. AD SCAN INITIALIZATION
*    Initialize the structure containing the AD scan setting and allocate memory for our buffer containing sample values.
*======================================================================

/* allocate memory for buffer */
samples = (DSCSAMPLE*)malloc( sizeof(DSCSAMPLE) * ( dscadscan.high_channel - dscadscan.low_channel + 1 ) );

printf("AD Scan Initialization complete\n");
#endif
#endif /* MDL_START */

/* Function: mdlOutputs */

static void mdlOutputs(SimStruct *S, int_T tid)
{
    #ifndef MATLAB_MEX_FILE
    real_T       *y = ssGetOutputPortSignal(S,0);
    #endif

    /========
    * V. SCANNING AND OUTPUT
    *    Perform the actual sampling and then output the results. To calculate the actual input voltages, we must convert the sample
must be cast to a short to get the correct code) and then plug it into one of the formulas located in the manual for your board (under "A/D Conversion Formulas").

if( ( result = dscADScan( dscb, &dscadscan, samples ) ) != DE_NONE )
{
    fprintf( stderr, "%s%s\n", ERROR_PREFIX, dscGetErrorString( result ) );
    free( samples ); /* remember to deallocate malloc() memory */
    return;
}

for( i = 0; i < (dscadscan.high_channel - dscadscan.low_channel)+ 1; i++)
{
    y[i] = (short)dscadscan.sample_values[i] / 32768.0f * 10.0f;
}

/* Function: mdlTerminate
*======================================================================
===

static void mdlTerminate(SimStruct *S)
{
    /*======================================================================
    VI. CLEANUP
    * Cleanup any remnants left by the program and free the resources used
    * by the driver.
    * STEPS TO FOLLOW:
    * 1. free the memory allocated for sample values
    * 2. free the driver resources
    ==*/

#ifndef MATLAB_MEX_FILE

    free( samples );
    dscFree();

#endif MATLAB_MEX_FILE
printf("\nMemory for input-channels freed\n");
printf("DSCADscan complete\n");
return;

#endif
}

 rencont*-----------------------------*
  * Required S-function trailer *
  *-----------------------------*/

#ifndef MATLAB_MEX_FILE    /* Is this file being compiled as a MEX
file? */
#define simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif

/* Required S-function trailer */
*-----------------------------*/
#endif
Appendix D: Dataeagle® wireless system

To operate the object a wireless data transferring system for the PLC was purchased. The systems from Schildknecht Industrieelektronik are specially developed for wireless data automation in PLC environments.

The product used in the halobject (the DE 5000 series) was developed for transferring MPI protocols. This protocol is next to the normal Profibus protocols, the common language between PLC and operator panels made by Siemens.

To make the DATAEAGLE® online just a few parameters have to be set on the consoles, named CPU-side (master) and PG-side (slave), for connection to the PLC and operator panel respectively.

Pressing the right-arrow button first can set the values. The default value for the password is ‘00’ so only confirmation with enter will do. Now scrolling trough the menu will display ‘change interface driver’. Set the console on the object on ‘CPU-side’ and behind the panel on ‘PG-side’. Next the addresses must be inserted. The settings inserted are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU address</td>
<td>1</td>
</tr>
<tr>
<td>Panel (PG) address</td>
<td>2</td>
</tr>
<tr>
<td>Transmit channel</td>
<td>21</td>
</tr>
</tbody>
</table>

For the ‘CPU-side’ the Partner address is 2 and Station address is 1 with F for transmitting channel 21. On the PG-side the first two settings are the other way around. Confirmation with enter will save the changes and by pressing escape one can leave the edit menu. The DATAEAGLE® is now fully functional and this can be seen on the screen that will display <- - > and <-leerlauf-> at random. If only <-leerlauf-> is displayed there is no connection. For more advanced settings look into the schildknecht dataeagle manual.
The settings are saved in the memory of the consoles and by restarting the systems the wireless connection will automatically be re-established.