Robot Hardware Abstraction Layer

Special Course

Anders Beck, s021786

Danmarks Tekniske Universitet
Automation - DTU Electrical Engineering
Supervised by Nils A. Andersen
September 8, 2008

1 Abstract

The robot control framework, designed at the Institute of Automation, DTU Electrical Engineering is a project that has undergone constant evolution in around 10 years. Throughout the years the platform has evolved to a flexible and general framework, supporting many robots and configurations. One element has escaped this evolution, the robot hardware abstraction layer.

This project describes the design and implementation of a flexible robot hardware abstraction layer, RHD. RHD is designed to provide robust and flexible core functionalities such as a global variable database, TCP/IP server and Realtime scheduling. Hardware interaction is done through an XML configurable plug-in architecture to supply even more flexibility and future expansibility.

RHD is already thoroughly tested on the AU SMR platform and the KU Life HAKO autonomous tractor platform, but support for other robots including the AU MMR, iRobot ATRV-Jr and a plug-in supporting the *Stage* simulator is in full development.

2 Preface

This project was done as a special course at Automation, DTU Electrical Engineering and has a intellectual workload of 10 ECTS points. Robots and hardware for testing has kindly been made avalible by Automation, DTU Electrical Engineering and KU Life.

When reading this report, it is important to note that the main focus has been to create a full documentation of Robot Hardware Daemon (RHD). In this context, each section of the report is written to be read individually.

I will like to thank my supervisor Nils A. Andersen for his enthusiasm and very great sparring when developing, implementing and debugging RHD.

Anders Billesø Beck
s021786
Automation, DTU Electrical Engineering
September 8, 2008

Contents

3 Introduction

The Institute of Automation at DTU Electrical Engineering started the construction of the SMR in 1999. The platform proved to be a flexible and durable platform for robot development. Approaching the first decade of lifetime, a lot of software has been written for the robot platform. The SMR platform has founded the base for a robot control framework, being developed at AU.

In the course of time, many projects has involved controlling other robot platforms, such as the outdoor robot MMR and the automated HAKO tractor at KU Life. As any other development process, the expansion of the control system and perception servers has been continuous, and capabilities has moved from being a specific single-platform solution to be a general multi-platform framework.

The initial aim of this project, was to review the control structure of the evolutionized framework, and identify structural elements, that might be leftovers from the single-platform roots. As my M.Sc thesis project will attempt to add a planning layer on top of the existing control layer, it is important to have all preceding layers reviewed and optimized to fit the vision of the framework.

This analysis process turned out to be reasonably short. The robot control software MRC (previously SMRDEMO) and the perception robot servers had gone through many years of development and improvement, but one element of the control system had not evolved from it's original form: the robot Hardware Abstraction Layer (HAL).

Originally designed for the SMR, the HAL SMRD was designed to be a transparent interface, communicating with serial bus devices and forwarding raw data to clients through a TCP/IP interface. The design was clever for understanding and development of robot control systems, but as focus moved towards higher levels of control, it has showed too simple and most of all too inflexible.

All sensor elements was hard-coded into the SMRD-code, and all TCP/IP communication followed the serial protocol, used on the packed medium-bandwidth RS-485 bus. The result was several branches of this HAL existed, the MMRD and HAKOD, but as more and more new devices did not follow the RS-485 protocol, it was necessary to perform inflexible and cumbersome wrapping and unwrapping into this protocol.

The result of this, was a list of incompatible branches of SMRD that all needed time consuming handcoding to change, add or remove any hardware devices on the robots.

The need for a flexible HAL has been recognized for quite some time at AU and it was obvious for this project to solve this problem and design a configurable, flexible and HAL for the use in mobile and stationary robot configurations.

4 Robot Hardware Daemon, RHD

The need for a more flexible realtime hardware interface framework led to the design of Robot Hardware Daemon, RHD. Instead of being a simple synchronized data mirror as SMRD, RHD is intended to be a realtime synchronized variable oriented database. The variable database orientation provides great flexibility, but it also helps to enforce a clean cut interface between the various specific hardware formats and a user-manageable API.

As RHD is replacing the primary timebase in the AU robot control environment, a lot of effort was used to analyze the behavior of different scheduling mechanisms in Linux, and to create a robust, lightweight and flexible realtime-safe implementation. There is a balance of keeping compatibility to a traditional desktop Linux distribution and meeting the requirement of hard realtime performance. As described in the scheduler section, both has been achieved.

It is important to recognize, that RHD is a synchronized variable database, but it is running at a fixed samplerate, that defines how often the database snapshot is synchronized and how often the periodic call to the hardware drivers are executed.

RHD is build upon a set of Core components, and a range of specific Hardware Abstraction Layer plug-ins. The core components create the foundation of variable database, TCP/IP server and the realtime-sheduler. Plug-ins create the support of hardware devices, by managing hardware I/O, pre-processing and database interface.

All setup of RHD is based on a XML configuration file, that contains the needed setup parameters for all modules in RHD. To keep as much flexibility as possible, the intension is to place as few ”magic” variables compiled into the code as possible. The line between a over- and under-configurable system is thin, and the design-decision is only to place information in the configuration file, that can/will change on different robot types. Specific static hardware handling is coded directly into the plug-in, and as calibration does not belong in a HAL, it is left for the client. Following these simple guidelines, all a plug-in must do, is to initialize the hardware and transfer data between the readable variables in the database.

To allow easy development of client programs, RHD is supplied with a static C client library: `librhd`.

4.1 Core Components

The framework of RHD is based on it’s three core components: the database, the server and the realtime scheduler.

Each component is designed as an individual element, that can be used for other purposes, but it is required and compiled into RHD.

4.1.1 Variable Database

The variable database defines the functionality of RHD. Upon initialization, all plug-in modules create their needed I/O variables, and finally the database is locked when going into soft/hard realtime.

The database itself, are based upon a symbol table and a data area. The symbol table defines the static information regarding the variables and bookkeeping information. The data area, contains the dynamic information, the actual data and it's timestamp. The needed memory is dynamically allocated, when variables are created. Figure 1 below, show the internal structure of the variable database and the association between symbol table and data area.

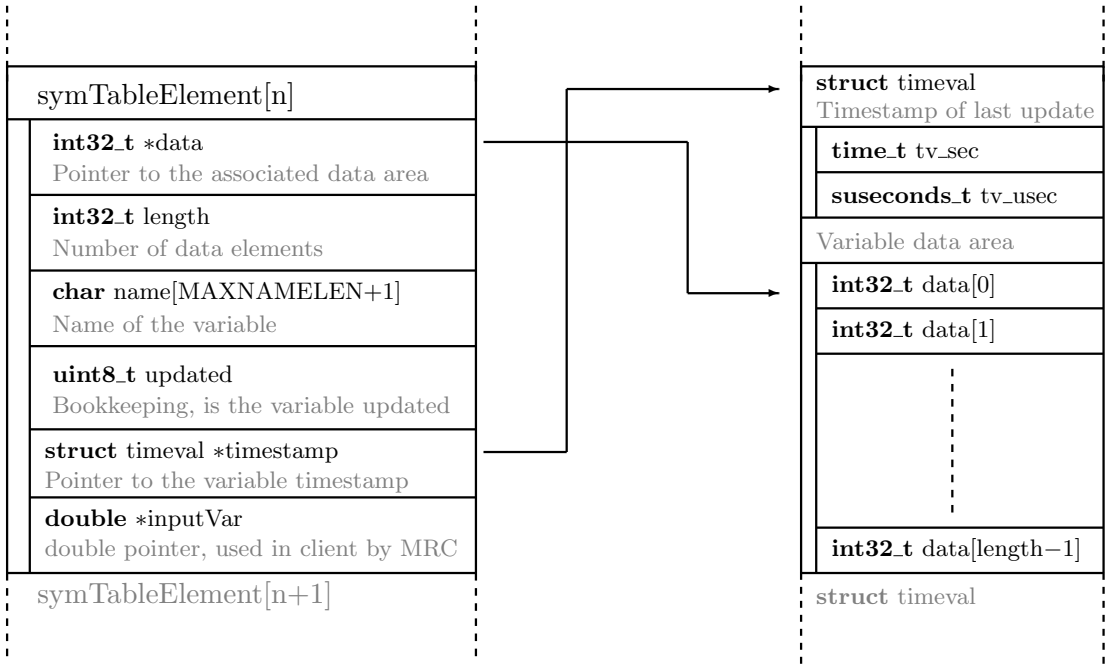

Figure 1: Structural overview of an element in the variable database

Note, that a standard variable size of a 32-bit signed integer is selected. This decision is made, to keep variables "large enough" for most future applications, and the fact that memory and bandwidth is not a dramatic issue on most modern platforms.

Some discussion has concerned to include a type to the variable database, as it might seem inefficient to represent text strings in 32-bit integers. A future enhancement of RHD might supply the possibility of using other integer sizes, but so far the design decision is to keep it simple with the highest flexibility.

Bidirectionality is obtained by using two separate symbol tables: a read and a write table. Data read from sensors are fed into the read-table, and data from the client is received in the write table.

Each variable is supplied with a ”updated” flag (see fig 1). The flag is used to implement reactions to updated variables, and to determine what data should be exchanged with the client.

There might always be a need to know the exact time a variable was updated, so all variables has a timestamp, that is set, every time a variable is updated. This makes it possible to control timing with a much higher resolution than the fundamental RHD scheduler period.

The API is defined in `database.h`. This API overview is not intended as a programming reference, but just to give a feel of, how the database is interfaced.

Variable creation is done by the plug-ins using the function below:

int createVariable (**char** dir, **char** len, **char** ∗name) *Add a variable array to the variable pool*

The direction, `dir`, is either the 'r' og 'w' character, that determines in which symbol table the variable is created. `len` assigns the number of elements in the variable array, and the name parameter is a pointer to a text string, containing the desired name of the variable. CreateVariable returns the `id` number of the variable, that is used to address the data, when performing I/O. When initialization is completed, the database is locked in soft-realtime-mode, and no further variables can be created.

Interfacing the variables are done through the following functions:

int setVariable (**int** id, **int** index, **int** value) *Update a array in the data pool*
int setArray (**int** id, **int** length, **int** ∗array) *Update a array in the data pool*
int setVariable (**int** id, **int** index, **int** value) *Get a read variable from the data pool*
int getWriteArray (**int** id, **int** index, **int** ∗array) *Get a read variable from the data pool*
int getReadVariable (**int** id, **int** index) *Get a read variable from the data pool*
int getReadArray (**int** id, **int** index, **int** ∗array) *Get a read array from the data pool*
int isUpdated (**char** dir, **int** id) *Check if a variable is updated*

Using this simple API, data can be transferred into the read database and out from the read and write database. The `id` parameter is the variable id, that is returned when the variable is created, and index is the array-index in the database. Check if variables are updated is performed through the `isUpdated()` function.

Other functions, that can be found in the RHD documentation, give the possibility of interfacing the symboltables directly, but their use is highly discouraged. The API above provide semaphore and boundary projection, and result in more safe and readable code.

4.1.2 TCP/IP Server

After some consideration around shared memory and direct library interface, it was decided to use TCP/IP sockets as interface for RHD. Socket based communication provides the best

independence of architecture and is a closer coupling to the previous server-infrastructure. The main argument for the ”low level” interfaces is speed and efficiency. However, as the data-throughput of RHD is modest, compared to the capabilities of most modern platforms, the effort of changing infrastructure and the loss of flexibility, was not worth it.

When designing the database, the primary focus was to create a layout, that is easily synchronized through a data-interface. That led to the distinction of static data in symbol table, and dynamic data in a data-table. When clients connect to RHD, an initial handshake is performed where the symboltables are transmitted. After the handshake, only data-table data is exchanged.

Situations can occour, when multiple clients attempt to connect to RHD and control the robot, without realizing the conflict. To avoid this issue, RHD only allow one client to perform write operations, but multiple clients as ”readers”. This is determined by the handshake.

When designing well-researched elements as communication, it is rarely a good idea to spend too much time inventing new and genius protocols, instead of using well tested and defined ones. In this case however, it was a strong desire to make the communication as simple as possible, as close to the database architecture as possible and using as little overhead as possible. The resulting protocol is basically a binary transfer of the database contents, wrapped in a minimum of control variables.

4.1.2.1 Client / Server handshake

After a client has connected to the RHD server, the client initiates the handshake by requesting either read or write permission. This check-in is done by sending the 8-bit character ’`r`’ or ’`w`’, as shown on figure 2.

Client check-in package: **char** ’w’ or **char** ’r’

Figure 2: Client initial check-in in RHD protocol

When the server receive a check-in from a new client, it will respond by transferring the symboltables from the database. As it is defined in the client structure of RHD, that only one client can have write permission, the response from the server. When write access is requested and write access is available, RHD transmits both the write-symboltable and the read-symbol table. If only read-access is requested or write access is occupied, RHD respond by only sending the read-symboltable. Figure below 3 show the data-structure of the transmission.

Server Response:

If write access is requested and allowed write-table is transmitted

Read-table are always transmitted, unless too many clients are connected

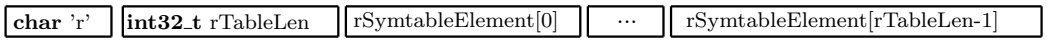

Figure 3: Server symboltable response to client checkin in RHD protocol

The protocol structure is simple. Initially a character identifies the following package as the read- or write-symboltable. Then the number of symboltable elements is transmitted in a 32-bit integer. Finally the symboltable elements is transferred in binary format (multi-byte values are transmitted in network-byteorder). When both symboltables are transmitted, they are simply transmitted sequentially. If more than the defined number of clients tries to connect, RHD simply closes the connection.

As the symbol table contain a lot of system specific pointers, the client completes the handshake by allocating the needed memory for the data-section and re-create the data-area pointers. After this procedure, each new RHD client has a complete copy of the relevant variable database structure.

4.1.2.2 Client / Server dynamic data exchange

When a client handshake is completed, RHD only need to transmit the data-area from the variable database, to keep the client and server synchronized. RHD is also working as the primary sheduler for realtime robot control software, such as MRC. This is an extra concern, when designing the data-exchange.

It is important to notice, that despite my best effort, no network protocol is better than the interface it is running on. If soft/(hard) realtime behavior is expected, the only safe way is a local loop-back connection. A wired network interface might also give somewhat predictable and fast response, but do not expect to maintain reliable or stable operation through a wireless network.

The basic consideration of the data exchange protocol is, that latency in the TCP/IP protocol, is just as much caused by the overhead of small packages than the actual data transfer. This lead to a protocol, where only **one** package is sent in each direction in one server period.

Figure 4 below, show a typical period, for a RHD server with two connected clients.

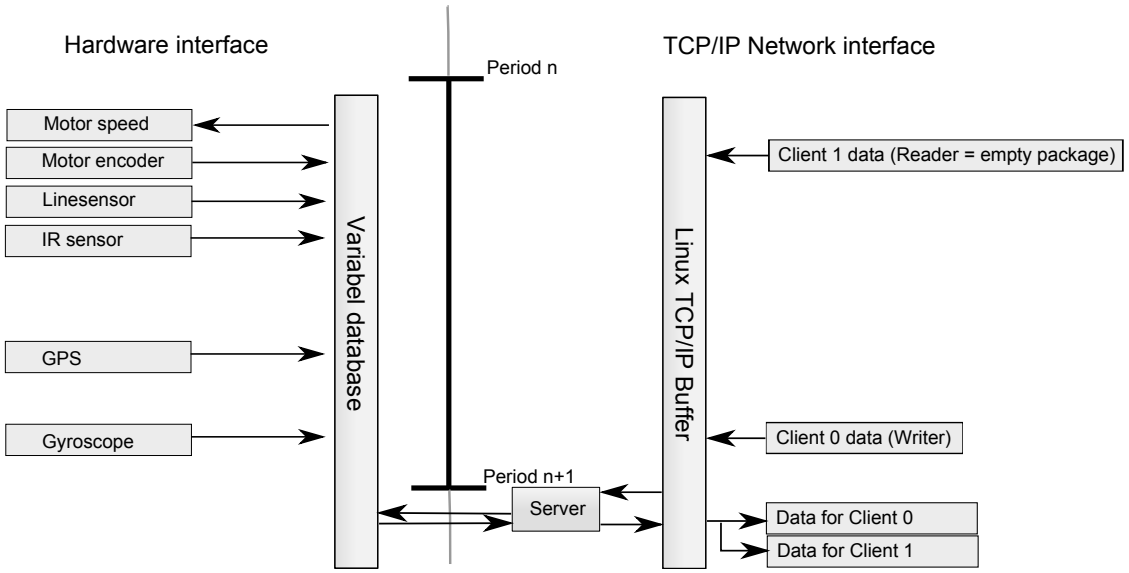

Figure 4: Hardware and Client - Server data traffic in one typical RHD period

When a client are done with it's periodic processing, it transmits a data package, containing all the variables that should be updated to the RHD server. This also counts as a 'ready' sign for the client to RHD. If a client is connected as a reader, this package should be empty, otherwise it is ignored. When the server receive the periodic tick from the scheduler, it start processing the client data-packages from its TCP/IP buffer and update the incoming variables in the database. When that is finished, the server transmits a datapackage, only to the 'ready' clients, containing the variables that should be updated in the client.

The package format is identical for both directions and is shown on figure 5

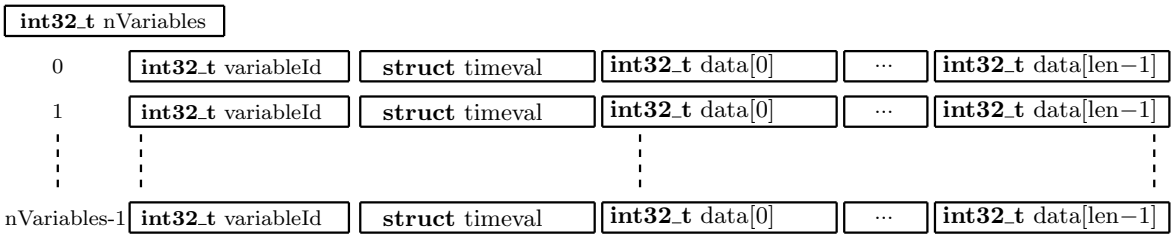

Figure 5: RHD Synchronization data package

The package header, is a 32-bit integer, determining how many elements there are in the payload. If a read-client transmits the datapackage, this integer must be zero (0). Then the payload follow, starting with the database ID of the variable beeing transmitted, followed by timestamp and data for the variable. It is easy to recognize, that the last part of each payload element, is a direct copy of the database data-area for each variable. The only processing performed on the data, is that all variables are converted between host and network byte-order at each end of the TCP/IP socket.

Using this, relatively simple protocol, an informative variable database is easily distributed to clients, and variable data is synchronized with a minimum of overhead.

4.1.2.3 XML Configuration of the server

The server XML configuration tag can be seen below:

```
<server>
  <port value="24902"/>
  <clients number="10"/>
</server>
```

The server can be configured to run on any TCP/IP port, but default is 24902. This is simply the incremental port from port 24901, used by SMRD. Also the number of accepted clients can be configured. The limit is theoretically system memory, but keep your systems capability in mind, when setting this value. The overhead of having more read-clients connected is, however, quite small. It is basically set by the resources used by the TCP/IP stack, to transmit variables to the extra receiver.

4.1.3 Realtime Scheduler

Besides being a networked variable database, RHD is also the main timebase for the low level robot control applications, such as MRC. The AU robot control platform is in strong growth, and is constantly expanding the range of supported robots. Some of these platforms is clearly in the heavy sector, that makes fault tolerance and especially realtime performance critical.

Hard realtime performance, is not an issue that is generically supported by the Linux kernel, as there is no support for kernel preemption. In different projects, as RTAI and RT-Linux, this is solved by patching the kernel with a second scheduler, that allow tasks to work in kernel priority levels and full kernel preemption.

Timing is generally a platform specific issue, is solved in many different ways. X86 platforms use a range of timers, such as the old APIC-8259 (Advanced Programmable Interrupt Controller) timing IC to the new HPET (High Precision Event Timer), found on modern platforms. This makes an universal implementation of a realtime environment difficult. Our RT-implementation of choice is the RTAI (RealTime Application Interface) patches. Despite that RTAI does support a range of architechtures, installation can be cumbersome or impossible on some platforms.

Linux itself, provide some means of achieving soft realtime performance. RHD would be most flexible, if it could operate satisfactory on a standard Linux distribution and with enhanced stability using the RTAI sheduler. RHD is capable of exactly that.

4.1.3.1 Linux Scheduling

To obtain best possible realtime performance in Linux, all realtime threads must be raised in priority. The default priority of the realtime threads in RHD is one below the maximum allowable in userspace. Furthermore, when initialization is completed, threads must run with a locked heap, to prevent memory allocation.

The Linux kernel itself is not preemptive, so all scheduling happen in the period of the scheduler, the so-called *jiffy*. Previously, this frequency was normally set to 100Hz, but seemingly the default of the vanilla kernel is now 250 Hz. This create a period of 4 ms and make it impossible to obtain the 10 ms control cycle.

Figure 6 below, show the period times of a Linux Itimer interrupt, set at 10 ms and 40 ms target periods on the SMR X86 VIA platform and on a Atmel AVR32 Linux evaluation board.

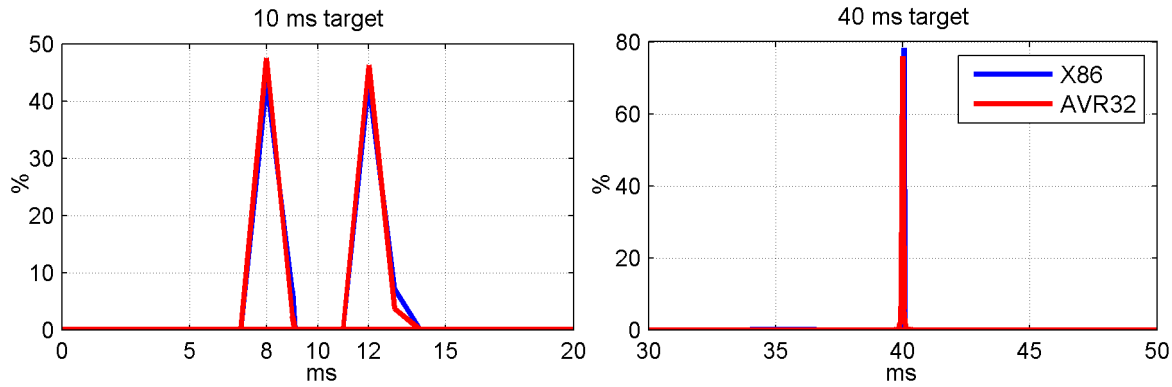

Figure 6: Scheduler periods with 10 and 40 ms targets on a 250 Hz kernel scheduling frequency

When trying to time a period, that is not divisible with the scheduler period, the timer will simply jump between the two closest scheduler ticks and thus create an accurate average timing period. This might be resonable, if the timer period is much larger than the scheduler period, but for realtime robot control it is highly undesired. If the period is dividable with the scheduler period, performance is very good. Figure 7 show a closeup of the second part of figure 6.

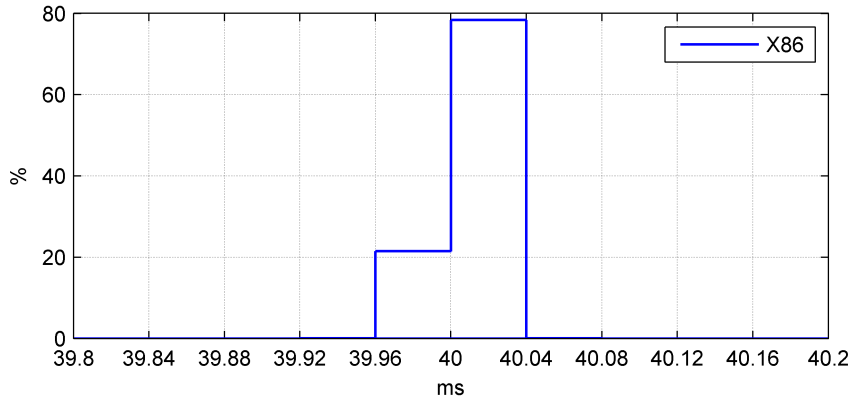

Figure 7: Closeup of samples from 39.8 to 40.2 ms at 250 Hz scheduler frequency and 40 ms timer target

Data to figure 7 is logged with a 40 μs interval. If samples from 39.96 - 40.04 ms is summed, it yields a total of 99.90 % of the samples, which is quite good performance.

When compiling the 2.6 Linux kernel, it is possible to set the scheduler frequency for 100, 250, 300 and 1000 Hz. To run RHD using the Linux scheduler, the timer loop period must be carefully selected as a multipla of the Linux sheduling period. It is strongly recommended to use a kernel compiled for 1000 Hz scheduling frequency, for greatest flexibility and resolution.

The RHD scheduler has two implementations of Linux scheduling. The recommended is based on the Interval timer interrupt, *Itimer*. The main thread is suspended by a call to the `pause()` function, and restarted when the itimer interrupt is caught. Alternatively, an algorithm using the `usleep()` function is available, but test has shown it to be less reliable.

SMRD solved the scheduling problem by linking scheduling to the serial port. In each period, a fixed number of bytes was transmitted, and processing is resumed when the serial port buffer was cleared. Waiting for the serial port, placed the thread in I/O-wait condition, instead of suspending it to the scheduler, making it independent of the scheduling period. The principle is good, but RHD should be a flexible framework, independent of hardware elements, so this method was not viable. This performance can be obtained through the RTAI framework.

4.1.3.2 RTAI Scheduling

RTAI supply two ways of implementing realtime applications. Traditional RTAI is only for writing kernel modules, but a new module LXRT support realtime threads in user space. Kernel space is a thing, that generally should be avoided if at all possible, and as LXRT supply the needed functionality, this is the implementation of choice.

When a realtime thread is created through LXRT, the LXRT kernel module creates a kernel realtime service-thread, to handle all realtime requests from userspace. This provides the

possibility of hard realtime from userspace, but it does not solve the flexibility issue of being able to compile and work with and without RTAI seamlessly.

In RHD this issue is solved by communication through realtime FIFO's. The scheduler spawn a separate hard realtime thread, who simply perform a periodic wait-loop. When each period expires, it transmit a timestamp through a realtime FIFO channel. The usual main thread performs a simple read on the FIFO, that make it go into I/O wait state (as done by SMRD). That never release the main thread to the Linux scheduler, but wait until data is written through the FIFO.

RHD can be compiled on a enviroment supporting RTAI, by supplying the command `make RTAI`. When the compile environment is properly setup at the automation servers, this will properly be changed to `make NORTAI`, to compile without RTAI, as RTAI scheduling should be used whenever possible.

4.1.3.3 Open loop sheduling

In some rare cases, it might be useful to be able to control the RHD timer period from one or more plug-ins. The development of RHD has made the previous robot simulators useless, as they emulate SMRD communication. The solution to this, is to write a plug-in for RHD, that uses the Stage simulator[1]. To be able to control timing from the Stage plugin, the RHD scheduler must be deactivated. To facilitate this request, the freerunning mode was designed.

Using freerunning mode in normal RHD operation could result in very high system load and invalid data from plug-ins so be careful with this option.

4.1.3.4 XML Configuration of the scheduler

The scheduler is configured by the XML tag below:

```
<sheduler>
  <period value="10000"/><!--in usec-->
  <type value="itimer"/><!--"usleep","itimer","LXRT", "freerunning" -->
</sheduler>
```

The timing period is defined in μs. Scheduling can be performed using four different algorithms, set by the keywords: `usleep`, `itimer`, `LXRT` or `RTAI` and `freerunning`. Using `usleep` is generally deprecated, but `itimer` show quite good performance, if the timer period is fixed to a multiple of the Linux scheduler period. RTAI can be enabled both the keywords `RTAI` or `LXRT`. If RHD is compiled without RTAI support, it will default to `itimer` scheduling, if it is set to use RTAI. When using the `freerunning`, RHD periods will only be limited by plug-in process time and server operation.

[1] Part of the Player/Stage project (http://playerstage.sourceforge.net)

4.2 Plug-in modules

The RHD core framework itself, does not supply possibilities for hardware interaction. Specific drivers are implemented through a plug-in structure.

As drivers are included in operating systems, the principle of drivers in RHD is to *"include em' all, use what you need"*. In this way, it is avoided that some branches of RHD start drifting away, as MMRD and HAKOD is specific, but incompatible, implementations of SMRD.

Plug-ins have the possibility to interface the variable database for client communication, but is responsible for interfacing and parsing data to and from hardware devices.

4.2.1 RHD Plug-in structure

For future developers, the most important issue, is to be able to write new hardware plug-ins for RHD. The procedure is documented in this section.

Up to version 1.x (current release version), RHD has not implemented hardware drivers as plug-ins, but as C-files that were compiled into RHD. This version also implement a unified driver architecture, but calls to the driver is done by explicit function calls in the RHD core thread. Work has now started on a version 2.0, that will introduce an actual plug-in structure, using Linux dynamically loaded libraries. Despite the plug-in architecture is in development, the interface is tested and final.

4.2.1.1 Plugin library interface

The RHD plug-in loader expect each plug-in to supply three functions:

Initialization The initialization function take a string, that point to the XML configuration file as parameter. Initialization is performed, before RHD is switched to soft realtime, so memory allocation is allowed. Most plug-ins use this function to parse the XML configuration, allocate memory, open I/O ports, and spawn asynchronous receive threads. If any of these operations fail, the initialization call **must** return a value < 0. The initialization function is mandatory, before RHD will accept the plug-in.

Periodic function The periodic function is called at each scheduler period. The function takes an integer as parameter, that is the number of scheduler ticks, since RHD start. This counter will overflow, at some point. The purpose of the periodic function is to perform write calls and other periodic maintenance. It is important, that execution of this function is as quick as possible, as the all periodic calls are performed from the main thread. No blocking reads or image processing must be done in this call. Write-calls are buffered by the Linux operating system and are allowed. This function is optional and can be omitted by i.e. read-only plug-ins.

Shutdown The shutdown function is executed, when RHD catches a kill signal. It can be used to close ports and end threads properly. The shutdown function is optional.

To enable RHD to load the plug-in properly, the header file must hold the following function declarations:

```
extern "C" int initXML(char *xmlfilename);
extern "C" int periodic(int tick);
extern "C" int shutdown(void);
```

Remember, that the initXML() function is mandatory, the others are optional and will be omitted, if they are not present in the header.

4.2.1.2 Compiling a plug-in

A plug-in should be complied as a Linux shared library. This is done by compiling each file, desired to be within the library into .o files, and finally constructing the library. It is important, that the library file is named lib[libname].so.[version].

The listing below, show an example of how to compile a shared library on Linux commandline:

```
gcc -fPIC -O2 -c -Wall plugin.c
gcc -fPIC -O2 -c -Wall extra.c
gcc -shared -Wl,-soname,libmyplugin.so.1 -o libmyplugin.so.1 plugin.o extra.o -lc
```

Note, that each .c file is compiled with the -fPIC parameter, that creates Placement Interdependent Code.

The RHD SVN package contains a `plugins` folder. Within this folder, all plug-ins have their own folder, that contain an individual makefile. The makefile makes the plug-in into a shared library and moves the resulting binary to the project `bin/rhdplugin` folder, that holds all rhd plug-ins. When adding a plug-in to RHD, just copy the makefile from another plug-in and adjust it to the new plug-in.

Many plug-ins use some shared functions, found in the `globalfunc.c` file. Plug-ins are loaded into RHD, to acquire access to all functions within RHD, including these functions. `globalfunc.c` contains functions to perform a secure read/write to a Linux filepointer and to set serial-port parameters.

4.2.1.3 Plug-in XML configuration

The RHD configuration file, normally rhdconfig.xml, has a <plugins> tag section. Before RHD is able to load a plug-in, it must have an entry, that follow the template:

```
<plugins basepath="rhdplugin/">
  <[plugin name] enable="true" lib="lib[plugin_name].so.1" critical="true">
      <plugin config tag 1.../>
      <plugin config tag 2.../>
      ...
```

```
    </[plugin name]>
  </plugins>
```

The tag identifier should be the plugin name, illustrated by [plugin name] on the listing. To be able to easily enable/disable the plugin, it must have an `enabled` parameter, that sets whether the plug-in will be loaded or not. A `lib` parameter sets the filename of the plug-in dynamically loadable library. This should normally be named lib[plugin name].so.1, if it is version 1, or .so.2 for version 2. It is not recommended to do too much version management with compiled plug-ins. The final template parameter is `critical`. It is an optional parameter, but if it is set to true, RHD will exit if the initialization function returns > 0. This is useful, if a sensor critical to the operation of a robot. It will ensure that the robot will not operate without this sensor properly initialized. If this tag is omitted or set to false, RHD will run if initialization fails, but not do any further interaction with the plug-in.

Configuration of the plug-in is done by adding child-tags, between the [plugin name] tags. It is also possible to add further parameters to the [plugin name] tag, but is discouraged.

4.2.2 AuSerial plug-in

One of the main motivations for developing RHD, was that the RS485 serial bus interface in SMRD was very inflexible. Adding new devices to the bus, or a simple reconfiguration, required a quite large change in both server and client code. A task, tedious and difficult for the maintainer and almost impossible for students. The AuSerial plug-in is providing a fully XML-configurable interface to the serial protocol, used on the SMR RS-485 bus.

AuSerial is the most complex plug-in for RHD, but also the most flexible. Unlike the other plug-ins, most of the AuSerial code is used on initialization. The periodic "running" code, is optimized to be as simple and efficient as possible, thus providing greatest functionality.

4.2.2.1 SMR serial protocol

The serial protocol, used on the SMR sensors, are developed by Per Koch Jensen in his thesis work in 2000. Some insight of the protocol structure is necessary to understand and use the configuration of AuSerial. The protocol is an 8-bit package oriented protocol, based on device ID's and commands. The package format is shown on figure 8 below:

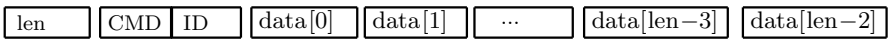

Figure 8: Datapackage in the SMR serial format (8-bit data)

Each package is initiated by a length byte. It determines the number of following bytes, to a maximum of 32 bytes. The following byte is the command and id byte. The least significant hex value is the device id, allowing a adress space of 16 devices. Id 0 is reserved for broadcast (but currently not used). The most significant hex value is the command,

also allowing a command space of 16 commands for each device. Command 0xF (16) is reserved for the extended command space, where the actual command is included in the following data byte. This adds 255 commands to the command space. After the id and command byte follow 0-31 payload bytes. It is allowed to send a command without any payload and there are no formal way of organizing data in the payload.

The communication on the SMR bus is full duplex but is working as master-slave. The computer is running the master bus and each sensor is driving the slave bus, when responding to a master request. To avoid collisions, no slave must transmit without being requested by the master. Another source of collisions is, if the master requests data from a new slave, before the previous finished its response. This is normally avoided by having the master emitting 0'es on the bus, that matches the number of bytes returned from a request.

The protocol itself, does not provide any means of fail safety. The most common problem is, that it might loose synchronization, and see command or data bytes as length bytes. This could leave a device non operational. This problem is solved by emitting 32 0'es in the end of each timer period. This would fill the receive buffer of any device that is out of sync. When sync is recovered, the 0'es would just be a length of zero bytes, and keep the device ready for a real package.

4.2.2.2 Configuring busses, devices and commands in XML

AuSerial supplies a fully configurable interface, that allow to several serial ports, each with 16 devices and each device with 16 commands. Configuration is placed within the <auserial> XML tag, and is quite lengthy for a full SMR configuration, with all functions enabled.

The first level of the configuration, is assign a serial port, represented by the <bus> tag. Configuration of a standard serial port is shown below:

```
<auserial enable="true" lib="libauserial.so.1" critical="true">
  <bus name="RS485" dev="/dev/ttyS0" baudrate="115200" holdoff="6">
    [Bus devices configuration here]
  </bus>
  <bus ...
</auserial>
```

Configuration of the plug-in is explained in section 4.2.1.3, and will not be discussed. Each <bus></bus> set define a serial bus. A bus configured by a name parameter, mostly for description. The dev sets the Linux device for the serial port. Communication speed is set by the baudrate parameter. The transmission capacity of the bus is calculated from the scheduler period and transmission speed. For 115.2 kBps, it is 115 byte pr 10 ms.

The holdoff parameter sets how many bytes should be reserved of the bus capacity, for "idle" time. This parameter is created, to ensure that all data traffic is cleared on the bus, in each period. So far RHD only issues a warning, if holdoff is violated, but it is discussed if data should be discarded. There is no safe solution to the problem so far, as both overrunning the bus or discarding possibly critical data is highly unsafe.

Below the <bus> tag, it is possible to define 16 devices. A device resemble a physical sensor on the bus, and must have an unique id. The XML listing below show the configuration of a SMR motor controller device:

```
<auserial enable="true" lib="libauserial.so.1" critical="true">
  <bus name="RS485" dev="/dev/ttyS0" baudrate="115200" holdoff="6">
    <!-- Left motor module -->
    <device name="motorl" id="1">
      [Command configuration here]
    </device>
    <device ...
  </bus>
</auserial>
```

A device is only defined by two parameters, a name, used for description, and id, defining it's the bus id. The id value **must** be assigned in HEX values 0-F. The last protocol specific tag, is the commands, assigned to each device. The standard commands used for a motor controller is shown on the XML listing below:

```
<auserial enable="true" lib="libauserial.so.1" critical="true">
  <bus name="RS485" dev="/dev/ttyS0" baudrate="115200" holdoff="6">
    <!-- Left motor module -->
    <device name="motorl" id="1">
      <cmd name="reset" type="request" cmd="0"/>
      <cmd name="speed" type="request" cmd="1">
        [Data configuration here]
      </cmd>
      <cmd name="enclTx" type="poll" cmd="2" pad="5" period="1" offset="0"/>
      <cmd name="enclRx" type="request" cmd="A">
        [Data configuration here]
      </cmd>
    </device>
  </bus>
</auserial>
```

The first defined command, is the reset command. There is no data payload with the reset command, and the tag is terminated with />. The <cmd> tag takes a range of parameters. The name is used for description. The type parameter accept two values: *poll* and *request*. Poll commands, is periodically transmitted by RHD but request commands are only transmitted when associated variables are updated from a RHD client (Configuring data variables are described in section 4.2.2.3). The device command number is set by the cmd parameter. So far, the extended command space is not supported by the protocol, but must be done manually, by setting cmd="F" and assigning the data payload to represent the extended command value. The command parameter must be assigned in HEX values 0-F.

Commands, that initiate returning of data from the devices, must have an appropriate pad parameter. This value assigns, how many 0's that should be written to the bus, while waiting for the device to respond. Normally this value is set to the length of the response, but some sensors might have some set-up time, before they answer and need a few bytes of extra padding. Test this carefully, before deciding a value.

Poll commands have the option of two extra parameters: period and offset. They are used to distribute bus traffic, if poll commands is not needed at every period. A good example is

the SMR IR sensor modules, where data is updated at 6 Hz. There is no need for sampling them at 100 Hz. period is used to set how many scheduler periods there is between each poll. To avoid all commands being polled at one given period, the parameter offset is used to move the poll a number of scheduler periods, within it's own period. offset should be less than period.

The motor module is a good example of, how problems can emerge, when there is no fixed standard for using a protocol. Poll for encoder values has the command number 0x2, but return values in a package with command number 0xA. This is fixed in the example by creating entries for a poll-command to request values, and a request-command to receive values. Messy, but it works.

A plea to all sensor designers, please let sensors return the same command id as the requesting command id.

Using the XML formatting and examples, described in this section, it is possible to create robots that has any configuration of busses and devices, as long as they follow the SMR serial protocol. Adding an extra linesensor or creating a robot with 10 motor controllers, is just a matter of configuring the sensors with the right id's and writing the AuSerial configuration. Linking the commands to RHD variable data is discussed in the following section

4.2.2.3 Associating database variables to AuSerial commands

When a sensor structure is defined in the AuSerial configuration, the real challenge is to map payload data to RHD database variables. The SMR serial protocol is 8-bit oriented, but sensors use a wide range of payload encondings and representing data in 1-bit, 8-bit, 10-bit and 16-bit formats.

To allow full support of existing sensor devices, and allow full flexibility of the SMR serial protocol, it was necessary to create a method to map each bit of variable to bus data and back from bus data to variable data.

The principle is, that every command can write data to the serial bus, and read data that is returned. This can be directly linked to a number of RHD database read and write variables.

As the RHD variable database is array oriented, it is possible to create arrays and variables. A variable is basically an array with one element. For the sake of simpliticy, the first example will show the configuration of a variable.

Variable mapping (single element array)

This XML listing show the variable configuration of the return data from a motor controller (without the preceeding tags):

```
<cmd type="request" name="enclRx" cmd="A">
  <variable name="encl" dir="r" byte0="1" byte1="0"/>
  <variable name="pwml" dir="r" byte0="3"/>
</cmd>
```

In the example *"enclRx"* create two variables, *"encl"* and *"pwml"*. Both variables have the parameter dir set to *"r"*, meaning that they are read-variables. The interesting part, is the byte-mapping. The parameter byte0..byte4, assign the 4 bytes in the 32-bit database variable to the 0-31 byte of serial bus payload data. Figure 9 show how the payload bytes are mapped to the variables in the *enclRx* example.

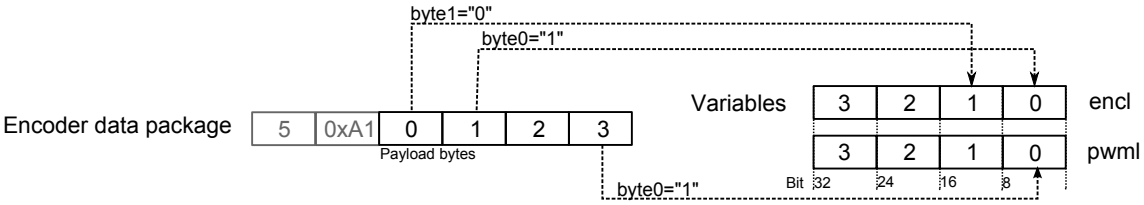

Figure 9: Mapping of serial bus payload bytes to RHD variable bytes for motor controller encoder values

Array mapping

Often multiple data values are associated, such as the 8 readings of the SMR linesensors. To allow easiest access to these values, the RHD variable database is array-oriented. In the same way as the encoder example before, serial payload data can also be mapped to arrays. Below is an example of how to map the 8 linesensor mesurements to an 8-element array

```
<cmd type="poll" name="values" cmd="1" pad="10">
  <array name="linesensor" dir="r">
    <element byte0="0"/>
    <element byte0="1"/>
    <element byte0="2"/>
    <element byte0="3"/>
    <element byte0="4"/>
    <element byte0="5"/>
    <element byte0="6"/>
    <element byte0="7"/>
  </array>
</cmd>
```

The plugin automatically create an database variable with an array size, that matches the number of element tags. Each array element has byte0 mapped to 8 consecutive bytes of the payload, as illustrated on figure 10.

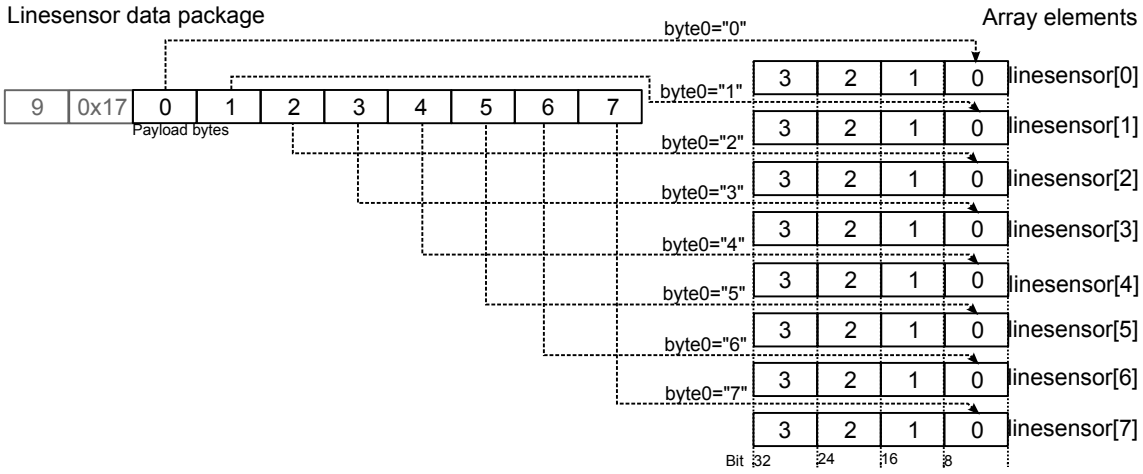

Figure 10: Mapping of several array elements to a linesensor serial bus package

Bit-mapping

The final mapping method is individual bit mapping. As the transfer capacity of the serial bus is limited, variable data is often packed bitwise into the payload. A good example of this, is the SMR power supply module, that has 5 10-bit ADC mesurements and 6 1-bit boolean values mapped into 6 payload bytes. Figure 11 illustrate, how the vaulues of different resolution are packed into the 8-bit serial format.

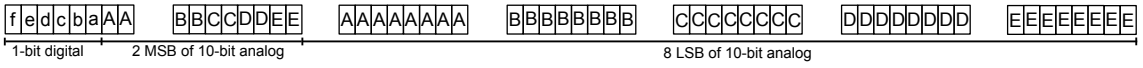

Figure 11: Payload package from power module, boolean digital values a-f and 10-bit analog measurements A-E are packaged into 8-bit blocks

To resolve payloads like this, into variables, it is nesseary to be able to map each individual bit of the payload into a specific bit of the database variable. Bit mapping is done by the parameter b0 – b31, that assigns bit 0 to 31 of the database variable to the bit assigned in the parameter value. The syntax for the bit-map parameter is bN="bit,byte", where bit is the addressed payload bit and byte is the addressed payload byte. An example of the full mapping of the power module payload is shown in the XML listing below:

```
<cmd type="poll" name="status" cmd="1" pad="10">
  <array name="digital" dir="r">
    <element b0="2,0"/>      <!-- a -->
    <element b0="3,0"/>      <!-- b -->
    <element b0="4,0"/>      <!-- c -->
    <element b0="5,0"/>      <!-- d -->
    <element b0="6,0"/>      <!-- e -->
    <element b0="7,0"/>      <!-- f -->
  </array>
  <array name="analog" dir="r">
    <element byte0="2" b8="0,0" b9="1,0"/>   <!-- A -->
    <element byte0="3" b8="6,1" b9="7,1"/>   <!-- B -->
    <element byte0="4" b8="4,1" b9="5,1"/>   <!-- C -->
```

```
    <element byte0="5" b8="2,1" b9="3,1"/>   <!-- D -->
    <element byte0="6" b8="0,1" b9="1,1"/>   <!-- E -->
  </array>
</cmd>
```

Note that the analog array in the example above uses both byte, and bit mapping. Byte-mapping is always performed first, then bit-mapping, so beware that a wrongly mapped bit can overwrite byte-mapped information.

Mapping write commands

The same methods is avaliable for mapping write commands. The byteN and bN parameters always adresses the database variables and their values in " " marks adresses the payload bit and bytes. The example listing below, show how to create the motor-speed and motor-reset variables and map the data into the payload bytes:

```
<cmd type="request" name="reset" cmd="0">
  <variable name="resetleftmotor" dir="w"/>
</cmd>
<cmd type="request" name="speed" cmd="1">
  <variable name="speedl" dir="w" byte0="0"/>
</cmd>
```

In the example, the variable "speedl" has it's lowest byte mapped into the first payload byte. Note that the reset command does not have any data mapped. This is because there are no payload bytes in the reset command. But the command will be transmitted to the bus, if the variable "resetleftmotor" is updated from the client, no matter what value is written into it.

Optional parameters

Besides the mapping of variable data, the variables has a few other, optional parameters.

Data can be transmitted in signed form. This creates a problem, when it is transferred into the 32-bit variable database, as the sign-bit is lost. The parameter **signed**="true", makes it possible to sign-extend the database variable, to maintain signedness. The listing below show an example of it's use:

```
<array name="analog" dir="r">
  <element byte0="2" b8="0,0" b9="1,0" signed="true"/>
  <element byte0="3" b8="6,1" b9="7,1"/>
  <element byte0="4" b8="4,1" b9="5,1"/>
  <element byte0="5" b8="2,1" b9="3,1"/>
  <element byte0="6" b8="0,1" b9="1,1"/>
</array>
```

This listing makes the first element of the analog variable sign-extended. This means, that the most significant bit (in this case b9) is copied to the bits b10-31 (two's compliment).

Another optional property, inversion, is made mostly for backward compatibility. As the motor-setup on the SMR's are mirrored, each motor is running in opposite directions. To make programming of the SMR movement intuitive, the right motor speed-command

and the resulting encoder readings are inverted in the old SMRD enviroment. This gives the illusion that both motors are running in the same direction. To give this functionality, AuSerial has the invert="true" parameter. The example listing below, show the full configuration of the right motor controller:

```
<!-- Right motor module -->
<device name="motorr" id="2">
  <cmd type="request" name="reset" cmd="0">
    <variable name="resetrightmotor" dir="w"/>
  </cmd>
  <cmd type="request" name="speed" cmd="1">
    <variable name="speedr" dir="w" byte0="0" invert="true"/>
  </cmd>
  <cmd type="poll" name="encrTx" cmd="2" pad="5"/>
  <cmd type="request" name="encrRx" cmd="A">
    <variable name="encr" dir="r"  byte0="1" byte1="0" invert="true"/>
    <variable name="pwmr" dir="r"  byte0="3" invert="true"/>
  </cmd>
</device>
```

Note that all values are inverted on the motor.

4.2.2.4 Final notes on the AuSerial plug-in

AuSerial is by far the most advanced plug-in written for RHD so far. The clear vision was to create a fully configurable interface, that can provide full support for existing and new devices on the SMR serial bus. I believe that it has been successfull.

It has been highly focussed, that most advanced code is placed in the initialization phase, and simplifying the run-time code. Actual run-time code is actually only around 300 lines of the 1000 lines plug-in.

When the configuration of AuSerial is understood, it is not overwhelmingly difficult. It is, however intended that it should be using standard configuration files, and only changed by "SMR Experts" for new hardware or special projects as Eurobot.

One functional thing is missing. It is not possible to execute the shutdown function, from status of a variable. In SMRD, the Linux shutdown is executed when *bit a* (see figure 11) from the power-module is set. The vision of a solution, is a possibility to map any system command to the status of a variable. This will also provide a useful functionality to execute system functions, such as audio playback or screen output from external sensors.

4.2.3 GPS plug-in

The first task for the RHD system, was a full implementation on the automated HAKO tractor at KU Life. The original hardware daemon for this purpose, HAKOD, was written by Asbjørn Mejnertsen and Anders Reeske Nielsen in 2006. It provided interface for the specific hardware modules on the HAKO tractor, but wrapped in the SMRD protocol. That made it possible to re-use most interface code and easily create RHD plug-ins for the hardware.

The GPS plug-in was created in combination of using the code from HAKOD and the GPS server by Lars Mogensen and Christian Andersen, with a slight re-write of the serial interface. Using these code-bits, the GPS module provides support for standard NMEA GPS and the RTK GPS, used on the HAKO tractor.

When a standard NMEA GPS is used, koordinates are converted from Lat-Lon to UTM, using a default UTM Zone, assigned in the configuration file. It is possible to change the zone using a write-variable.

The configuration of the plug-in is done on in the following piece XML

```
<gps enable="true" lib="libgps.so.1" critical="true">
  <serial port="/dev/rfcomm0" baudrate="4800"/>
  <utmzone value="32"/><!-- Default UTM Zone -->
</gps>
```

The configuration give the possibility of setting the serial port, baudrate and default UTM zone. In this example, the port is configured a bluetooth serial NMEA GPS.

The GPS plug-in creates the following database variables:

Variable	Dir	Variable contents	Description
gpstime	r	[hour][min][sec][ms]	Time of day in millisecons resolution
gpsdate	r	[dd][mm][yyyy]	Date in day, month and year
gpstimeofday	r	[sec][μsec]	Linux timeofday in second and μseconds
gpsnorthing	r	[m][μm]	UTM Northing coordinate
gpseasting	r	[m][μm]	UTM Easting coordinate
gpslattitude	r	[deg][μdeg]	Lattitude coordinate (NMEA GPS only)
gpslongitude	r	[deg][μdeg]	Longitude coordinate (NMEA GPS only)
gpsquality	r	[quality]	Quality of GPS fix
gpsfixvalid	r	[0 / 1]	Binary value of valid fix
gpssatused	r	[sats]	Number of sattelites used in the fix
gpsdop	r	[dop][1/10 of dop]	Horizontal dillution of precision
gpsaltitude	r	[m]	GPS Altitude
gpsheigth	r	[m]	GPS Heigth
gpsheading	r	[deg][mdeg]	GPS Heading
gpsspeed	r	[m/s][mm/s]	Speed over ground
gpsegnos	r	[egnos]	Binary EGNOS fix (Unused)
gpsllfixes	r	[fixes]	No. of fixes with lat-long (NMEA GPS)
gpsutmfixes	r	[fixes]	No. of fixes with UTM (RTK GPS)
gpsutmzone	r	[zone]	UTM Zone for lat-long to UTM conversion
gpssetutmzone	w	[zone]	Write variable to set UTM zone

Table 1: Database variables created by the GPS plug-in

4.2.4 Crossbow gyro plug-in

Another device used on the HAKO tractor is the Crossbow IMU400. It is a full fledged 3-axis IMU, providing gyro rates and accelleration on all three axis. As it is based on

MEMS technology, it does have a small drift on gyro mesurements, but provide pretty decent performance. The price-tag is arround 15.000 dkk.

The RHD driver is mainly from Asbjørn Mejnertsen and Anders Reeske Nielsen's HAKOD, but the Crossbow initialization has been completely rewritten and a lot of busy-wait loops has been removed. The Crossbow unit available on the tractor does have some freeze issues with it's firmware, that can make the initialization fail. It is highly recomended to use the critical ="true" parameter, if the gyro operation is needed. This way, RHD will not operate, if the crossbow is not initialized properly.

XML configuration of the Crossbow plug-in is fairly simple:

```
<crossbow enable="false" lib="libcrossbow.so.1" critical="true">
  <serial port="/dev/ttyUSB1" />
</crossbow>
```

When initialized properly, the Crossbow plug-in creates the following variables:

Variable	Dir	Variable contents	Description
xbowroll	r	[roll]	IMU rate of roll
xbowpitch	r	[pitch]	IMU rate of pitch
xbowyaw	r	[yaw]	IMU rate of yaw
xbowx	r	[accl X]	IMU accelleration in X direction
xbowy	r	[accl Y]	IMU accelleration in Y direction
xbowz	r	[accl Z]	IMU accelleration in Z direction
xbowtemp	r	[temp]	IMU Temperature
xbowtime	r	[time]	IMU Time

Table 2: Database variables created by the Crossbow plug-in

Further description of the data, can be found in the IMU400 datasheet.

4.2.5 Fibre Optic Gyro plug-in

When working on their project of "Fault tolerant navigation for Mobile Robots", Peter Tjell and Søren Hansen received a Fibre Optic Gyro (FOG), that was just lying around at KVL (KU Life). It was promptly interfaced to RHD and intensively used on the HAKO tractor during their thesis project.

The FOG uses two RS-422 serial busses. One for setup and one for data. Using a clever cable, it is possible to run the FOG, using only one RS-422 port. This works if the TX pair is connected to the setup connector input and the RX pair is connected to the data connector output. This plug-in is written, so that both using two RS-422 ports or just one RS-422 port is possible. Setup is done at 9600 baud and data communication is done at 57600 baud.

The XML configuration is seen in the listing below:

```
<fogyro enable="true" lib="libfogyro.so.1" critical="true" >
  <dataserial port="/dev/ttyUSB0" baudrate="57600"/>
  <configserial port="/dev/ttyUSB0" baudrate="9600"/>
  <sampletime value="10"/><!-- time pr sample in ms -->
</fogyro>
```

In the configuration, it is possible to assign serial ports for both configuration and data. It is also possible to set the time pr. sample, performed by the gyro. See datasheet for further details.

The FOGyro plug-in creates the following database variables:

Variable	Dir	Variable contents	Description
fogphx	r	[ϕx]	ϕX angle
fogphy	r	[ϕy]	ϕY angle
fogphdz	r	[$\phi\Delta$z]	$\phi\Delta$Z rate. (aka. Yaw rate)
fogtempb	r	[tempb]	Temperature mesurement
fogtempe	r	[tempe]	Temperature mesurement
fogfailstatus	r	[status]	Failure status bits
foggyrostatus	r	[status]	Gyro status bits

Table 3: Database variables created by the FOGyro plug-in

See datasheet for further information regarding the variables

4.2.6 SMRDSerial plug-in

AuSerial was the last developed module for RHD. To have a stabile plug-in for testing, the main code of SMRD was ported into a RHD plug-in, named SMRDSerial. That made it possible to control all functions of the SMR, with a very short plug-in development time and through verified software.

This plug-in was essential to the development and debugging of RHD and is still in use for "long time verification" of the operation. When AuSerial is properly tested and configured, SMRDSerial will be declared depreciated.

Configuration of SMRDSerial is done through the following piece of XML:

```
<!-- SMRD Serial bus module -->
<smrd enable="true" lib="libsmrdserial.so.1" critical="true">
  <serial1 port="/dev/ttyS0" baudrate="115200"/>
  <serial2 port="/dev/ttyS1" baudrate="115200"/>
</smrd>
```

It is possible to configure both serial ports, for RS-485 (serial1) and optional RS-232 for the RS-232 linesensor.

SMRDSerial creates the following database variables:

Variable	Dir	Variable contents	Description
encl	r	[enc]	Left encoder value
encr	r	[enc]	Right encoder value
linesensor	r	[ls1][ls2]...[ls8]	8 Linesensor mesurements
irsensor	r	[ir1][ir2]...[ir6]	6 Ir sensor mesurements
gyro	r	[pos1][pos2][pos3]	Uncompensated position of the 3 gyros
gyrotemp	r	[temp1][temp2][temp3]	Gyro temperatures
speedl	w	[speed]	Speed command for left motor
speedr	w	[speed]	Speed command for right motor
resetmotorl	w	[reset]	Reset command for left motor
resetmotorr	w	[reset]	Reset command for right motor
steeringangleref	w	[angle]	Steering angle for the Ackerman SMR

Table 4: Database variables created by the SMRDSerial plug-in

4.2.7 Hako CAN-bus plug-in

The HAKO tractor is operated through a CAN-bus interface to the ESX ECU (Engine Control Unit), that is running a custom control code. Asbjørn Mejnertsen and Anders Reeske Nielsen wrote a piece of interface code into HAKOD, that has been the foundation for this plug-in.

HAKOD was never compatible with the Slackware Linux distribution, that is used on the RSE platform, as it used a Debian CAN-bus driver. Through quite some effort from Nils A. Andersen and Ole Ravn, it was possible to port the HAKOD code to use the Slackware CAN-bus driver and a plug-in was developed, that is able to control the HAKO tractor ECU.

As this plug-in is developed by Nils A. Andersen and not yet re-integrated into the development branch of RHD, and will not be described further in this report.

4.3 librhd Client library

For easy client development, RHD is supplied with a client library. The library provides functions to connect and synchronize to RHD servers, maintaining of the local variable database and I/O functions to use database data.

Despite that RHD is a real-time server, a client does not need to fullfill any real-time requirements. RHD will operate fine, servicing a real-time client while also transmitting data to non real-time ”spectator” clients.

4.3.1 Communicating with the RHD server

The communication API is controlled by three functions:

```
char  rhdConnect(char rw, char *host, int port);
char  rhdSync(void);
char  rhdDisconnect(void);
```

`rhdConnect()` establishes connection to a RHD server. It takes three parameters:

rw Set the desired access level. 'w' requests write access and 'r' requests read access to rhd

host Char string holding host name for the RHD server

port Integer holding the TCP port, that RHD is running on

If connection fails, `rhdConnect()` returns -1, but if connection is successful, it returns either 'w' or 'r', depending on what access level was granted from the RHD server. If one write client already is connected, `rhdConnect()` returns 'r' and is only able to receive data from RHD.

`rhdSync()` is the periodic function, that transmits write variables to the RHD server (if connected as write-client), and waits for read variables from the server. When this function is called, the thread will block, until a RHD server period is expired and data is exchanged. This is used to keep real-time synchronization between RHD server and client.

`rhdDisconnect()` closes the connection to the RHD server and frees memory allocated for the database.

4.3.2 Communicating with the variable database

Getting variable data in and out of the synchronized variable database can be done in two ways. The most efficient way is to access the variable symbol table directly. This method is also the most unsafe, and should only be used if performance is critical, as in MRC.

The API for this access method is

```
symTableElement*  getSymbolTable(char rw);
int               getSymbolTableSize(char rw);
```

Using the function `getSymbolTable()` will return a pointer to the symbol table array. Iterating through this array, makes direct data access possible. `getSymbolTableSize()` sets the limits for iterating, when searching for variables. The symbol table structure is described in section 4.1.1 on page 3. Be very careful, when using this method, as there are no protection against writing outside the symbol table arrays or data-areas. Timestamp must also be updated manually.

The preferred method is to use the dedicated I/O API. The functions is described below

```
int     getReadValue(int id, int index);      //Get value from read-database
int *   getReadArray(int id);                 //Get array pointer from read-database
int     getWriteValue(int id, int index);     //Get value from write-database
int *   getWriteArray(int id);                //Get array pointer from write-database
int     setWriteValue(int id, int index, int value);      //Write value to write-database
int     setWriteArray(int id, int length, int* arrayptr); //Write array to write-database
char    isUpdated(char rw, int id);           //Is the r/w-database value updated
char *  getVarName(char rw, int id);          //Get pointer to variable name
int     getVarLength(char rw, int id);        //Get the length of variable data-array
```

This API will allow full access to the read and write databases, but also provide protection for writing outside desired memory areas and perform all required book-keeping. The API is still in development, and might be subject for change. Always check the rhd.h header and Doxygen documentation for the most recent API. Operations of the API is almost identical to the variable database API described in section 4.1.1 on page 3.

5 Testing on various platforms

Robot systems can run on a large range of embedded platforms. To ensure full versatility, RHD has been programmed using full endian-safe code.

To test the flexibility of the system, RHD and MRCs predecessor SMRDEMO was compiled and tested on two non-x86 platforms. T The requirement was that the platform supported Linux and the GCC compliler. The systems used for test were

- Atmel ATNGW100 development board with a 130 MHz AVR32 Application processor and running a custom Linux Buildroot on kernel 2.6.19
- Fujitzu-Siemens Loox 720 PDA with a 520 MHz Intel XScale processor and Debian Linux 2.6.21

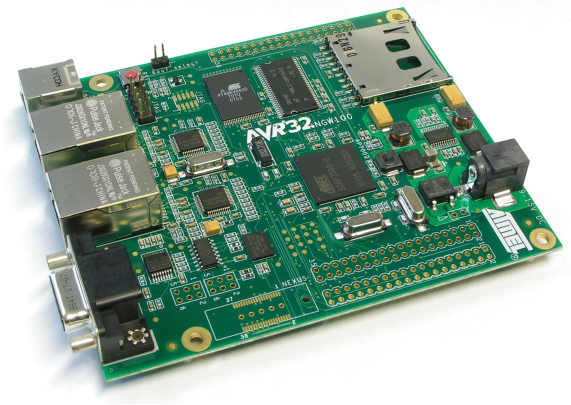

Figure 12: Atmel ATNGW100 development board

Figure 13: Fujitzu-Siemens Loox 720 PDA

The AVR32 platform was supported by a large toolchain from Atmel, that made it possible to cross-compile both RHD and MRC into AVR32-binary code.

The Intel XScale is supported by the Debian distribution, and the driver adjustment was done through an Loox720 open-source project. As the XScale is supported by Debian, all GNU tools are available for download through the Debian package-manager and compilation was done on the Loox720 itself.

Both software elements was tested individually. The functional test of SMRDEMO, were done using SMRD as hardware abstraction layer, instead of RHD, as MRC was not fully developed when the tests were performed. RHD was only tested using the demo client application.

The results of the is illustrated in table 5 below

	RHD		SMRDEMO+SMRD	
	Compile	Test	Compile	Test
AVR32	Works	Works	Works	Works
Intel XScale	Works	Works	Works	Works

Table 5: Test results of compiling and testing RHD and SMRDEMO on AVR32 and XScale architecture

As seen in table 5, both RHD, and (SMRDEMO + SMRD) worked on both platforms, without corrections in the code. Only when cross-compiling to the AVR32 platform, it was necessary to correct makefiles, to use the cross-compiling toolchain.

The conclusion of this test, is that the robot-control platform has now reached a level, where it allows a very high level of flexibility both in choice of robot hardware and computer platform.

6 Further development

RHD has now moved to the maturing development phase. The plug-in structure is not yet implemented, and some minor corrections are still in due process.

The program is now in care of the Robot Systems Engineering (RSE) group and the newest program version can be found in the RSE SVN repository. Note that SVN development should follow the guidelines for SVN[2], using trunk, branches and tags. The most recent stable development version is always found within the trunk section.

Updated documentation of RHD is always found in the RSE wiki[3], that will provide the information from this report and any new development information.

[2]See http://svnbook.red-bean.com/
[3]RHD can be found at: http://timmy.elektro.dtu.dk/rse/wiki/index.php/RHD

7 Conclusion

The aim of this project, was to review the AU robot control architecture, to identify and redesign any un-addressed leftovers from single-platform dawn of the architecture. The largest structural problem was quickly identified as the various hardware abstraction layers, used on the robots.

Through this project, a new, flexible hardware abstraction layer, the Robot Hardware Daemon (RHD), was implemented. RHD provides a fully XML configurable interface, and provides all the basic functionality of realtime scheduling and client-server communication. Hardware interaction is provided through a plug-in interface, to provide maximal flexibility and expansibility.

To provide proof-of-concept, plug-ins was programmed for the SMR platform and the HAKO platform. Both platforms runs exactly the same software, just using different configuration and plug-ins.

One drawback of changing to a different HAL, was that the robot simulators designed at AU, now become inoperational. This problem is already now in the process of being solved, by designing a plug-in for RHD that loads the simulator *Stage*, provided with the Player/Stage project. It had been a desire for a long time, to utilize *Stage* and possibly the 3D simulator *Gazebo* and that has been possible using RHD.

RHD must be considered an overall success. The core has proven itself flexible, stable and providing new functionality throughout the control system, such as dynamic variables. The plug-in architecture makes it simple to change robot architectures and add new hardware. In close future, another robot will join the family of RHD supported robots, the iRobot ATRV-Jr and hopefully many more will follow soon.

Anders Billesø Beck
Automation, DTU Electrical Engineering
September 8, 2008

A Example XML Configuration files

A.1 RHD configuration XML file for version 1.x

```
1  <?xml version="1.0" ?>
2  <!--
3       Configuration file for
4       Robot Hardware Daemon
5
6       See something else for configuration description
7
8       $Id: rhdconfig.xml 144 2008-05-18 22:29:25Z andersbeck $
9  -->
10 <rhd>
11   <!-- *** Core Components Configuration *** -->
12   <!-- Sheduler configuration -->
13   <sheduler>
14     <period value="10000"/><!--in usec-->
15     <type value="itimer"/><!--"usleep","itimer","LXRT" -->
16     <rtaiinit cmd="./initrtai.sh"/><!-- Script to initialize RTAI -->
17   </sheduler>
18   <!-- Server configuration -->
19   <server>
20     <port value="24902"/>
21     <clients number="10"/>
22   </server>
23   <!-- *** Modules Configuration *** -->
24   <!-- SMRD Serial bus module -->
25   <smrd enable="true">
26     <serial1 port="/dev/ttyS0" baudrate="115200"/>
27     <serial2 port="/dev/ttyS1" baudrate="115200"/>
28   </smrd>
29   <!-- HAKO Tractor CAN-Bus module -->
30   <hakocan enable="false">
31     <controlcan port="/dev/can1"/>
32   </hakocan>
33   <!-- Crossbow IMU Configuration -->
34   <crossbow enable="false">
35     <serial port="/dev/ttyUSB1"/>
36   </crossbow>
37   <!-- Fibre Optic Gyro module -->
38   <fogyro enable="false">
39     <dataserial port="/dev/ttyUSB0" baudrate="57600"/>
40     <configserial port="/dev/ttyUSB0" baudrate="9600"/>
41     <sampletime value="10"/><!-- time pr sample in ms -->
42   </fogyro>
43   <!-- Serial GPS module -->
44   <gps enable="false">
45     <serial port="/dev/rfcomm0" baudrate="4800"/>
46     <utmzone value="32"/><!-- Default UTM Zone -->
47   </gps>
48   <!-- Automation serial bus driver -->
49   <auserial enable="false">
50     <bus name="RS485" dev="/dev/ttyS0" baudrate="115200" holdoff="6">
51       <!-- Linesensor module -->
52       <device name="linesensor" id="7">
53         <cmd type="poll" name="values" cmd="1" pad="10">
54         <array name="linesensor" dir="r">
55           <element byte0="0"/>
```

```
56            <element b0="0,1" b1="1,1" b2="2,1" b3="3,1" b4="4,1" b5="5,1" b6="6,1" b7="7,1"/>
57            <element byte0="2"/>
58            <element byte0="3"/>
59            <element byte0="4"/>
60            <element byte0="5"/>
61            <element byte0="6"/>
62            <element byte0="7"/>
63          </array>
64          </cmd>
65          <cmd type="request" name="idstring" cmd="9" pad="10">
66          <variable name="idstring" dir="r" byte0="0"/>
67        </cmd>
68        </device>
69      <!-- Left motor module -->
70      <device name="motorl" id="1">
71        <cmd type="request" name="reset" cmd="0"/>
72        <cmd type="request" name="speed" cmd="1">
73          <variable name="speedl" dir="w" byte0="0"/>
74        </cmd>
75        <cmd type="poll" name="enclTx" cmd="2" pad="5" period="2"/>
76        <cmd type="request" name="enclRx" cmd="A">
77          <variable name="encl" dir="r" byte0="1" byte1="0"/>
78          <variable name="pwml" dir="r" byte0="3"/>
79        </cmd>
80      </device>
81      <!-- Right motor module -->
82      <device name="motorr" id="2">
83        <cmd type="request" name="reset" cmd="0"/>
84        <cmd type="request" name="speed" cmd="1">
85          <variable name="speedr" dir="w" byte0="0"/>
86        </cmd>
87        <cmd type="poll" name="encrTx" cmd="2" pad="5"/>
88        <cmd type="request" name="encrRx" cmd="A">
89          <variable name="encr" dir="r"  byte0="1" byte1="0"/>>
90          <variable name="pwmr" dir="r"  byte0="3"/>
91        </cmd>
92      </device>
93      <!-- IR sensor module -->
94      <device name="irsensor" id="8">
95        <cmd type="poll" name="distances" cmd="8" period="10">
96          <array name="irdist" dir="r">
97            <element byte0="0"/>
98            <element byte0="1"/>
99            <element byte0="2"/>
100           <element byte0="3"/>
101           <element byte0="4"/>
102           <element byte0="5"/>
103         </array>
104       </cmd>
105     </device>
106     <!-- Power supply module -->
107     <device name="power" id="9">
108       <cmd type="poll" name="status" cmd="1" pad="10">
109         <array name="digital" dir="r">
110           <element b0="2,0"/>
111           <element b0="3,0"/>
112           <element b0="4,0"/>
113           <element b0="5,0"/>
114           <element b0="6,0"/>
115           <element b0="7,0"/>
116         </array>
```

```
117           <array name="analog" dir="r">
118             <element byte0="2" b8="0,0" b9="1,0"/>
119             <element byte0="3" b8="6,1" b9="7,1"/>
120             <element byte0="4" b8="4,1" b9="5,1"/>
121             <element byte0="5" b8="2,1" b9="3,1"/>
122             <element byte0="6" b8="0,1" b9="1,1"/>
123           </array>
124         </cmd>
125       </device>
126       </bus>
127     </auserial>
128 </rhd>
```

A.2 RHD configuration XML file for version 2.x

```
<?xml version="1.0" ?>
<!--
    Configuration file for
    Robot Hardware Daemon

    See something else for configuration description

    $Id: rhdconfig.xml 144 2008-05-18 22:29:25Z andersbeck $
-->
<rhd>
  <!-- *** Core Components Configuration *** -->
  <!-- Sheduler configuration -->
  <sheduler>
    <period value="10000"/><!--in usec-->
    <type value="itimer"/><!--"usleep","itimer","LXRT" -->
    <rtaiinit cmd="./initrtai.sh"/><!-- Script to initialize RTAI -->
  </sheduler>
  <!-- Server configuration -->
  <server>
    <port value="24902"/>
    <clients number="10"/>
  </server>
  <!-- *** Modules Configuration *** -->
  <plugins basepath="rhd/">
    <!-- SMRD Serial bus module -->
    <smrd enable="true" lib="libsmrdserial.so.1" critical="true">
      <serial1 port="/dev/ttyS0" baudrate="115200"/>
      <serial2 port="/dev/ttyS1" baudrate="115200"/>
    </smrd>
    <!-- HAKO Tractor CAN-Bus module -->
    <hakocan enable="false">
      <controlcan port="/dev/can1"/>
    </hakocan>
    <!-- Crossbow IMU Configuration -->
    <crossbow enable="false" lib="libcrossbow.so.1" critical="true">
      <serial port="/dev/ttyUSB1"/>
    </crossbow>
    <!-- Fibre Optic Gyro module -->
    <fogyro enable="true" lib="libfogyro.so.1" critical="true" >
      <dataserial port="/dev/ttyUSB0" baudrate="57600"/>
      <configserial port="/dev/ttyUSB0" baudrate="9600"/>
      <sampletime value="10"/><!-- time pr sample in ms -->
    </fogyro>
    <!-- Serial GPS module -->
    <gps enable="true" lib="libgps.so.1" critical="true">>
      <serial port="/dev/rfcomm0" baudrate="4800"/>
      <utmzone value="32"/><!-- Default UTM Zone -->
    </gps>
    <!-- Automation serial bus driver -->
    <auserial enable="true" lib="libauserial.so.1" critical="true">
      <bus name="RS485" dev="/dev/ttyS0" baudrate="115200" holdoff="6">
        <!-- Linesensor module -->
        <device name="linesensor" id="7">
          <cmd type="poll" name="values" cmd="1" pad="10">
      <array name="linesensor" dir="r">
        <element byte0="0"/>
              <element b0="0,1" b1="1,1" b2="2,1" b3="3,1" b4="4,1" b5="5,1" b6="6,1" b7="7,1
        <element byte0="2"/>
        <element byte0="3"/>
```

```
60          <element byte0="4"/>
61          <element byte0="5"/>
62          <element byte0="6"/>
63          <element byte0="7"/>
64        </array>
65            </cmd>
66            <cmd type="request" name="idstring" cmd="9" pad="10">
67          <variable name="idstring" dir="r" byte0="0"/>
68          </cmd>
69          </device>
70        <!-- Left motor module -->
71        <device name="motorl" id="1">
72            <cmd type="request" name="reset" cmd="0"/>
73            <cmd type="request" name="speed" cmd="1">
74          <variable name="speedl" dir="w" byte0="0"/>
75            </cmd>
76            <cmd type="poll" name="enclTx" cmd="2" pad="5" period="2"/>
77            <cmd type="request" name="enclRx" cmd="A">
78              <variable name="encl" dir="r" byte0="1" byte1="0"/>
79              <variable name="pwml" dir="r" byte0="3"/>
80        </cmd>
81        </device>
82        <!-- Right motor module -->
83          <device name="motorr" id="2">
84            <cmd type="request" name="reset" cmd="0"/>
85            <cmd type="request" name="speed" cmd="1">
86              <variable name="speedr" dir="w" byte0="0"/>
87            </cmd>
88            <cmd type="poll" name="encrTx" cmd="2" pad="5"/>
89            <cmd type="request" name="encrRx" cmd="A">
90              <variable name="encr" dir="r"  byte0="1" byte1="0"/>>
91              <variable name="pwmr" dir="r"  byte0="3"/>
92            </cmd>
93          </device>
94        <!-- IR sensor module -->
95          <device name="irsensor" id="8">
96            <cmd type="poll" name="distances" cmd="8" period="10">
97              <array name="irdist" dir="r">
98                <element byte0="0"/>
99                <element byte0="1"/>
100               <element byte0="2"/>
101               <element byte0="3"/>
102               <element byte0="4"/>
103               <element byte0="5"/>
104             </array>
105           </cmd>
106         </device>
107       <!-- Power supply module -->
108       <device name="power" id="9">
109           <cmd type="poll" name="status" cmd="1" pad="10">
110         <array name="digital" dir="r">
111         <element b0="2,0"/>
112         <element b0="3,0"/>
113         <element b0="4,0"/>
114         <element b0="5,0"/>
115         <element b0="6,0"/>
116         <element b0="7,0"/>
117         </array>
118         <array name="analog" dir="r">
119         <element byte0="2" b8="0,0" b9="1,0"/>
120         <element byte0="3" b8="6,1" b9="7,1"/>
```

```
121           <element byte0="4" b8="4,1" b9="5,1"/>
122             <element byte0="5" b8="2,1" b9="3,1"/>
123           <element byte0="6" b8="0,1" b9="1,1"/>
124           </array>
125           </cmd>
126           </device>
127         </bus>
128       </auserial>
129     </plugins>
130 </rhd>
```