VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ

ÚSTAV MECHANIKY TĚLES, MECHATRONIKY A BIOMECHANIKY

FACULTY OF MECHANICAL ENGINEERING

INSTITUTE OF SOLID MECHANICS, MECHATRONICS AND BIOMECHANICS

EMBEDDED CONTROL SYSTEM FOR AN AUTONOMOUS MOBILE ROBOT

VESTAVĚNÝ ŘÍDICÍ SYSTÉM PRO AUTONOMNÍ MOBILNÍ ROBOT

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE — Bc. JAN HRBÁČEK

AUTHOR

VEDOUCÍ PRÁCE — Ing. JIŘÍ KREJSA, Ph.D.

SUPERVISOR

BRNO 2011

Vysoké učení technické v Brně, Fakulta strojního inženýrství

Ústav mechaniky těles, mechatroniky a biomechaniky
Akademický rok: 2010/2011

ZADÁNÍ DIPLOMOVÉ PRÁCE

student(ka): Bc. Jan Hrbáček

který/která studuje v **magisterském navazujícím studijním programu**

obor: **Mechatronika (3906T001)**

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

Vestavěný řídicí systém pro autonomní mobilní robot

v anglickém jazyce:

Embedded control system for an autonomous mobile robot

Stručná charakteristika problematiky úkolu:

Navrhněte embedded řídicí systém pro mobilní robot s těmito parametry:

* podpora běžných průmyslových sběrnic (CAN-bus, EIA-485,..)
* minimálně soft real-time chování
* rozšířitelnost a modularita

Cíle diplomové práce:

1. Stručná rešerše tématu.
2. Výběr vhodného hardware.
3. Volba vhodného operačního systému.
4. Volba mechanismu meziprocesní komunikace jednotlivých modulů
5. Implementace základních ovladačů sběrnic.

Seznam odborné literatury:

Hristu-Varsakelis, Dimitrios; Levine, William S. (Eds.): Handbook of Networked and Embedded Control Systems

Vedoucí diplomové práce: Ing. Jiří Krejsa, Ph.D.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2010/2011.

V Brně, dne 26.10.2010

L.S.

prof. Ing. Jindřich Petruška, CSc. Ředitel ústavu	prof. RNDr. Miroslav Doupovec, CSc. Děkan fakulty

ABSTRACT

The master's thesis deals with the design and realization of an embedded control system for the autonomous mobile robot Advee. The control system forms an abstraction layer between the hardware means of the robot and higher control layers that handle robot localization and autonomous navigation. Modular system structure has been designed and inter-process communication mechanism has been chosen. The designed control system has been then implemented with the support for EIA-485 and CAN bus communication standards.

The architecture of the system has been verified during more than 500 hours of commercial operation of the robot prototype equipped with the control system.

KEYWORDS

Robotics, control, real-time

ABSTRAKT

Diplomová práce se zabývá návrhem a realizací vestavěného řídicího systému určeného pro autonomní mobilní robot Advee. Řídicí systém tvoří vrstvu abstrakce mezi hardwarovými prostředky robotu a vyššími vrstvami řízení, které provádějí lokalizaci robotu a plánování pohybu. V rámci návrhu byla vyvinuta modulární struktura systému a zvoleny prostředky mezimodulové komunikace. Navržený systém byl pak implementován včetně podpory komunikačních standardů EIA-485 a CAN bus.

Zvolená architektura systému se v praxi osvědčila — prototyp robotu Advee řízený popsaným systémem má za sebou více než 500 hodin komerčního provozu s minimem poruch.

KLÍČOVÁ SLOVA

Robotika, řízení, real-time

HRBÁČEK, Jan. *Embedded control system for an autonomous mobile robot*: master's thesis. Brno: Brno University of Technology, Fakulta strojního inženýrství, Ústav mechaniky těles, mechatroniky a biomechaniky, 2011. 69 p. Supervised by Ing. Jiří Krejsa, PhD.

DECLARATION

I declare that I have elaborated my master's thesis on the theme of "Embedded control system for an autonomous mobile robot" independently, under the supervision of the master's thesis supervisor and with the use of technical literature and other sources of information which are all quoted in the thesis and detailed in the list of literature at the end of the thesis.

As the author of the master's thesis I furthermore declare that, concerning the creation of this master's thesis, I have not infringed any copyright. In particular, I have not unlawfully encroached on anyone's personal copyright and I am fully aware of the consequences in the case of breaking Regulation § 11 and the following of the Copyright Act No 121/2000 Vol., including the possible consequences of criminal law resulted from Regulation § 152 of Criminal Act No 140/1961 Vol.

Brno ....

(author's signature)

ACKNOWLEDGEMENT

I would like to thank my thesis supervisor Ing. Jiří Krejsa, PhD. for his support, supervision and expert advice during the whole work on the thesis subject. My gratitude belongs also to the whole Bender Robotics team for being such great colleagues.

Contents

Introduction **11**
Mobile robot Advee . . . 11
Required functionality of the control system . . . 12
Real-Time constraints . . . 13

1 Recherche of existing solutions **14**
1.1 Microsoft Robotics Developer Studio . . . 14
1.2 Player/Stage . . . 15
1.3 IPC: Inter-Process Communication . . . 15
1.4 LCM: Lightweight Communications and Marshalling . . . 15

2 Computing platform selection **16**
2.1 Hardware means . . . 16
2.2 Operating system . . . 18
2.2.1 VxWorks, QNX, Windows CE etc. . . . 18
2.2.2 GNU/Linux . . . 18

3 Modular software conception **21**
3.1 Development tools used in lower levels . . . 22
3.2 Folder structure of the control system . . . 23
3.3 Standard module architecture . . . 24
3.3.1 Error reporting library `liberror` . . . 26
3.4 Inter-module communication . . . 27
3.4.1 LCM: Lightweight Communications and Marshalling . . . 28
3.4.2 Naming conventions in communication . . . 31
3.4.3 Communication interface of the modules . . . 31
3.4.4 User-space library `libmessaging` . . . 32
3.5 List of implemented modules . . . 32
3.6 The watchdog process . . . 33
3.7 Remote diagnostics and simulation support . . . 34
3.7.1 Simulation . . . 36

4 Detailed hardware description **37**
4.1 Power subsystem . . . 38
4.2 Motion subsystem . . . 39
4.3 Sensory equipment . . . 40
4.4 Computer box . . . 41
4.4.1 TS-7800 peripherals . . . 42

5 EIA-485 subsystem **43**
5.1 Bus description . . . 43
5.1.1 Communication protocol . . . 44
5.2 Kernel driver TS-UART . . . 45
5.3 User-space library `librs485` . . . 46
5.3.1 Device drivers . . . 49
5.4 Driver module `driver_rs485a` . . . 49
5.4.1 Interruptible waiting . . . 50
5.4.2 Data-polling functionality . . . 51
5.5 Driver module `driver_rs485b` . . . 52

6 CAN-bus subsystem **53**
6.1 Bus description . . . 53
6.2 CANopen higher-layer protocol . . . 54
6.3 Kernel driver `lincan` . . . 54
6.4 User-space library `libcanbus` . . . 55
6.4.1 DS-301 standard layer . . . 55
6.4.2 DSP-402 standard layer . . . 55
6.5 Driver module `driver_canbus` . . . 56

7 Auxiliary subsystems **57**
7.1 Power supply control . . . 57
7.1.1 Support for **BR-PSC1** in the `librs485` . . . 57
7.1.2 Driver module `driver_power` . . . 58
7.2 Fan control . . . 58
7.2.1 User-space library `libi2c` . . . 59
7.2.2 Driver module `driver_aux` . . . 60

8 Conclusion **61**

Bibliography **63**

List of symbols, physical constants and abbreviations **66**

List of appendices **67**

A Instructions of current devices **68**

List of Figures

1 Autonomous mobile robot Advee 12

2.1 TS-7800 ARM Single Board Computer [23] 18

3.1 Advee hardware and software modules scheme 22
3.2 Module state transition diagram 24
3.3 C#.NET diagnostic center 35
3.4 Android OS diagnostic center 36

4.1 Simplified schematics of Advee's functional units 37
4.2 Power box with some of power modules fitted 38
4.3 Parts of the Maxon drive system [27] 39
4.4 Berger Lahr IclA N065 Intelligent Compact Drive drawing [3] 40
4.5 TS-CAN1 PC/104 CAN-bus interface board [24] 42

5.1 EIA-485 typical application [26] 43
5.2 Custom EIA-485 / EIA-422 communication protocol message structure 44
5.3 Reception finite state machine diagram 47

8.1 Advee in a real environment (19th International Trade Fair AMPER 2011) 62

List of Tables

A.1 Instruction set of BR-SD1 68
A.2 Instruction set of BR-SO1 68
A.3 Instruction set of BR-SBS1 68
A.4 Instruction set of BR-PSC1 69

Introduction

Robotics is an engineering branch where a control system has always played an irreplaceable role and has dramatically influenced overall performance. Autonomous robotics has posed even more challenging claims than the "traditional" stationary industrial robotics — the control systems should serve also as a real system brain that disposes of certain level of intelligence allowing to solve complex problems.

In order to quallify a system as embedded, it has to be dedicated to perform only a restricted set of functions and be a natural part of a more complex device. This demands are very often satisfied rightly in the field of (mobile) robotics — a robotic control system is usually expected to be robust, reliable and power efficient which are another common characteristics of an embedded system.

The design of an embedded control system is an extensive task that comprises a number of decisions. It should start from consideration of mechanic constraints (that influence e.g. needed sampling rates), continue with selection of suitable hardware means (that would flawlessly interface sensors and actuators) and finish with software architecture choice (that is mostly discussed hereinafter). One can denote the design as a complex mechatronic problem.

This thesis should provide a description of such design process — except the mechanical analysis that is given in [20] and which the thesis builds upon. To outline concrete parameters of the problem, a brief description of the advertising robot Advee will be given.

Mobile robot Advee

Advee is an autonomous mobile robot developed in cooperation between the Institute of Solid Mechanics, Mechatronics and Biomechanics at BUT FME and Bender Robotics s.r.o. It is designated specifically for presentational and advertising purposes and providing the user with a rich multimedial experience. Mechanical construction of the robot is described in [20] and shown on Figure 1. It disposes of the Ackerman chassis type that provides great stability and power efficiency at the cost of its nonholonomic constraints.

(a) Outer appearance (b) Mechanical construction

Fig. 1: Autonomous mobile robot Advee

The electric equipment of the robot is formed by an eight-cell $LiFeYPO_4$ accu pack powering the whole robot, a set of sensors (16 ultrasonic, 4 infrared, odometry and a beacon scanner device), a motion subsystem (drive and steering) and human-interface devices (a large touch monitor, a digital camera and a pair of loudspeakers).

Required functionality of the control system

The submission of this thesis formulates several general demands on the designed control system that should be met:

- support for standard *industrial busses* (CAN-bus, EIA-485, ...)
- minimally *soft real-time* behavior
- *extensibility and modularity* in design

To master the design process, the thesis should provide a brief recherche on the theme, describe the hardware and suitable operating system selection process and choose an applicable inter-process communication mechanism. On these fundaments basic bus drivers shall be implemented.

Real-Time constraints

A real-time system is one with explicit deterministic or probabilistic timing requirements [9]. In other words, the system is subject to a constraint on response time within it should process the request. Depending on the impact of not meeting the deadline to the system, the individual task or the whole system can be *hard* or *soft* [2]:

- as *hard real-time* is usually denoted a task or a system for which exceeding the constraint means a failure
- *soft real-time* is a task / system whose performace is negatively affected by missing deadlines — but not critically

This division is however not binary — every application has its specifics and the degree of tolerance to exceeding time limits strongly varies. This fact is sometimes expressed by introducing the third, middle level of real-timeness — the *firm real-time* category.

The designed control system falls rather to the area of soft real-time computing. All fast processes are distributedly controlled by specialized control units (drive motor with its control unit, steering compact drive), used sensors cannot be polled faster than a few times per second and the safety of the robot is guaranteed also for the case of control system failure. The role of the control system is thus relatively high-level and the time constraints are measured in milliseconds rather than in microseconds.

Chapter 1

Recherche of existing solutions

One can find a wide variety of existing robotic control system realizations. Some of these are built without using any framework or library, but such approach it usually not sustainable for larger projects. Building the control system on the top of a set of libraries brings both positives and negatives. The positive aspects (using a tested code as the base, reduction of needed work) commonly outweigh the negative ones (e.g. possible occurrence of errors in the third party code) so that the utilization of a framework / library is generally beneficial.

Because each control system is unique, designed for different purposes and with various motivations (concreteness × abstraction etc.), the requirements posed on the control framework / library differ as well. A number of systems that can potentially satisfy these requirements exists; let's focus the core functionality of inter-process communication (IPC). The libraries differ in the basic communication paradigm (client-server vs. publisher-subscriber), complexity (encapsulating more IPC methods, data marshalling etc.) or operating system/programming language support. A good comparison of several popular IPC libraries targeted for use in robotics is provided in [10], including popular frameworks such as *IPC* or *Player/Stage*.

1.1 Microsoft Robotics Developer Studio

Microsoft Robotics Developer Studio [17] is a representative of the most complex frameworks. Based on the Concurrency and Coordination Runtime for the .NET framework, it provides a service-oriented model with multiple transport layer option and automatic data marshalling and un-marshalling. The framework features also a rich 3D simulational environment and a number of supported standard hardware platforms. However, it is limited to be used on Microsoft Windows operating systems that disqualifies it from further considerations.

1.2 Player/Stage

The framework is comprised of three projects [5]: the Player providing network interface layer to a variety of common robot hardware, the Stage project implementing a multi-body 2D simulation environment and finally the Gazebo project that runs the simulation in a 3D world. While the framework is probably one of the most used in the area of mobile robotics, it has been found to not fit the demands of a robust and scalable embedded system as featuring only a "pull" publisher-subscriber model over the TCP transport layer and running the complete robot driver software in a single process.

1.3 IPC: Inter-Process Communication

Another popular library is IPC [22], a part of the CARMEN toolkit. On the contrary to the Player it features the better "push" publisher-subscriber model, still using TCP transport. However, it has been decided to not use the library because of its need for a central communication server.

1.4 LCM: Lightweight Communications and Marshalling

LCM [10], [11] is a set of libraries designed at the Massachusetts Institute of Technology. It counts to the most lightweight frameworks (as the name suggests) and does not provide any simulational environment. However, it features a rich variety of supported programming languages and operating systems.

Chapter 2

Computing platform selection

The heart of the lower-level control system is usualy formed by a reasonably powerfull computer that coordinates the whole functionality. A suitable operating system that provides abstraction over the hardware means is similarly important as a functional hardware — it enables efficient use of the hardware on one hand and allows the software to be written in a hardware-independent and modular way on the other hand.

2.1 Hardware means

Appropriate hardware platform is esential to fulfill demands given by the submission. It should dispose of sufficient computing power to run both low-level (hardware drivers) and middle-level (state estimation, motion planning) software with a power margin reserved for future needs[1]. In order to minimize further development costs, the platform should have an easily exploitable support for needed standardized communication busses (at least Ethernet, CAN-bus and EIA-485/422).

The most important feature of the designed system is industrial-grade ruggedness and reliability that cannot be achieved without use of reliable hadware instruments. So called Single Board Computers[2] (SBCs) embody one of the best achievable reliability — industrially proven components are soldered directly to the board, minimizing the number of used connectors and sockets that are generally prone to a mechanical failure. These computers do not excell in computational power, RAM size or other parameters — often used is for example ISA bus that is something like a dinosaur among current PC technologies. But when long-term stable operation is a priority, it is a good choice.

[1] Reserve in operational load has positive influence also on the real-time behavior — the operating system can more easily schedule running tasks.

[2] Such computers are often found in embedded systems.

The class of the computer is thus clear; the next relevant consideration is the processor architecture. The architecture selection influences important parameters of the computer as computational power or consumption; there are differences in floating point calculations implementation or peripherals support. Exceptionally advantageous is so-called System on Chip (SoC) approach — something like a mature microcontroller that integrates almost all components of the computer into a single package; usualy only RAM and non-volatile storage memory are needed externally.

Single board computers are produced mainly using processors of following architectures:

- *x86* — a set of processor families ranging from the famous 32-bit Intel 80386 to recent Intel Core i7; the most widespread PC-class processor architecture supported by almost every operating system
- *ARM* (Advanced RISC Machine) — dominant architecture on the mobile and embedded market for its simplicity and low power demands
- *PowerPC* — known mainly as the architecture used for a long time by Apple (before Apple surprisingly switched to x86)

The computer can take virtually any form and size with the smallest ones occupying space not bigger than a credit card. However, there are several standard form factors used in embedded computing that are preferable to custom ones — industrial standardization allows to enrich basic functionality of the computer by using generic expansion modules. Probably the most used embedded form factor is PC/104; the standard specifies not only the size and shape but also used computer bus. In the oldest original version (that is still widely used) this bus is the already mentioned ISA. Newer mutations feature also PCI and PCI Express busses. Before choosing the form factor it is practical to verify availablity of peripheral modules that will be likely exploited (e.g. a CAN-bus interface module).

With all mentioned aspects in mind, the TS-7800 Single Board Computer [23] manufactured by Technologic Systems in PC/104 form factor has been chosen to host the control system (Figure 2.1). The board is built upon a 500MHz Marvell ARM9 SoC that disposes of a Gigabit Ethernet controller, two USB 2.0 host ports, two SATA ports and two native serial ports. It features 128 MB of DDR-RAM and 512 MB of high-speed NAND flash memory extendable using full-size and micro SD card sockets. Basic peripherals supported directly by the SoC are augmented by eight serial ports implemented in an on-board FPGA (two of them can be turned into EIA-485-compliant bus drivers) and five ADC inputs. The board can be optionally equipped with a real-time clock and a temperature sensor (both used in our case).

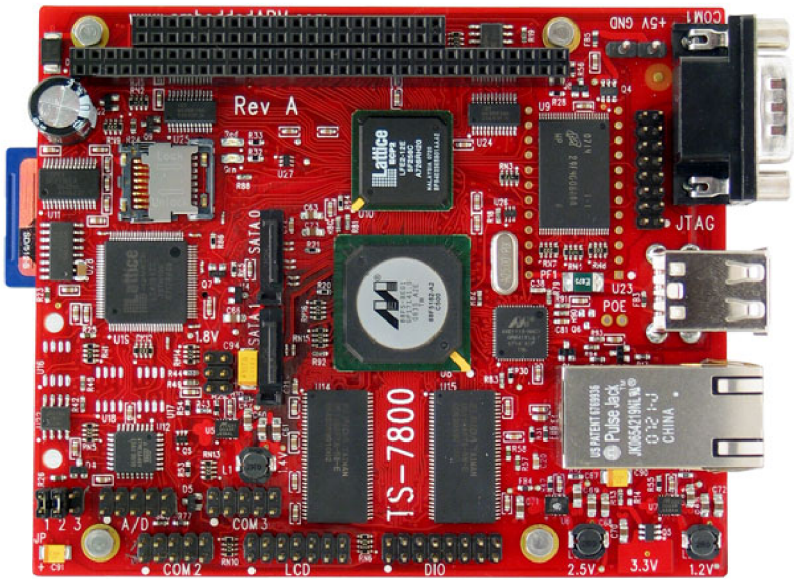


Fig. 2.1: TS-7800 ARM Single Board Computer [23]

2.2 Operating system

Efficient use of the hardware is conditioned by the choice of a suitable operating system that provides an interface to user-space programs. The OS should be capable of at least soft real-time (RT) scheduling to allow the control system to work as supposed.

2.2.1 VxWorks, QNX, Windows CE etc.

The biggest share on the real-time operating system (RTOS) market is held by commercial RT operating systems. Systems like the VxWorks can be found in devices ranging from cell phones to spacecrafts. Their scheduling and other system qualities are often supported by sophisticated development environments and custom hardware means ensuring the best compatibility between hardware and software.

However, their price is generally simply to high for anybody else than big companies which effectively disables it from the selection.

2.2.2 GNU/Linux

In the GNU/Linux notation, Linux (called after the founder Linus Torvalds) designates the kernel and GNU (recursive acronym of GNU's Not Unix) refers to user-space utilities and libraries provided by the GNU Project. The complete name is frequently shortened just to Linux — for clarity, "Linux kernel" will be used here to denote the kernel alone.

Linux kernel is monolithic with loadable modules support [15] and provides multitasking capabilities, shared user-space libraries or multistack networking including IPv4 and IPv6. Itself does not have real-time capabilities — there are generally two ways to add needed features: the dual kernel approach and making the kernel natively preemptive.

Dual kernel approach

This method introduces a new kernel (usually a micro- or nanokernel) which runs the whole non-RT Linux system as a thread with the lowest priority. The communication between RT and non-RT tasks is commonly possible using shared memory or FIFOs (First In First Out).

RTLinux [31] Hard real-time microkernel, Linux runs as a fully preemptive process.

Adeos Adaptive Domain Environment for Operating Systems [30] — a kernel patchset implementing a nanokernel hardware abstraction layer, thus enabling real-time behavior and more (virtualization, SMP clustering etc.).

RTAI Real Time Application Interface [21] — comprised of Adeos-based kernel patchset and supporting services. Unfortunately, ARM9-based MV88F5182 SoC is not supported.

Xenomai [29] A framework providing industrial real-time operating system APIs under Linux environment to allow simple migration of existing applications to GNU/Linux. Originally designed to wrap various dual-kernel extensions, version 3 aims to support also native kernel preemption. It supports the MV88F5182 SoC but is not known to be used with the selected TS-7800 SBC.

Natively preemptive kernel

The other approach modifies the standard ("vanilla") kernel to enable running both non-RT and RT tasks. The most important is the RT-Preempt patch:

PREEMPT_RT [16] — kernel patchset that makes almost all previously uninterruptible system calls preemptible.

The use of Linux-based operating system was probable from the beginning — due to its high modularity it can be run in a variety of forms ranging from small embedded applications to multiprocessor supercomputers. The TS-7800 SBC is primarily designed to operate in conjunction with Linux[3] and the source code of

[3]Minimalistic BusyBox environment and a full Debian distribution both in OABI and EABI mutations is supplied by the manufacturer.

almost all custom hardware drivers is published by the manufacturer. This leads to simple adaptability to concrete needs of the application even when the demands grow over standard capabilities of the system. It brings also feasibility of fixing potential errors in implementation (our experience with the 9-bit serial port mode is described in Chapter 5 EIA-485 subsystem).

At present, no real-time augmentation of the Linux kernel has been utilized. The modified kernel supplied by the manufacturer is compiled with the PREEMPT option set — it forces the highest level of preemption that the standard kernel allows [15]. As [1] shows, the worst case latency is 22 ms (average 50 µs) which suffices the current control system demands.

Chapter 3

Modular software conception

Development of the software should not only reflect the actual needs but also prepare ways that will enable future changes and extensions. It is very important to find a balance between abstraction and specificity — too specific system structure will not allow easy updating, on the other side too abstract design will be difficult to understand and implement [7].

Modularity is one of the key parameters that strongly influence adaptability of the system to unexpected eventualities. In the case of described robot control system it is achieved in terms of both hardware and software means — hardware components are interconnected using shared busses that allow for adding and replacing individual components as long as defined communication interface is abided. The principle of software modularity is exactly the same — the modules can be added or reimplemented on condition of maintaining of specified interfaces.

As illustrated on Figure 3.1, the overall software equipment of the robot is divided into three layers:

Low level Provides interaction with hardware devices described in previous chapters — basically it translates device-specific data into standard inter-modular messages and vice versa

Middle level Implements robot state estimation and path planning

High level Handles image processing, printing and user interaction

This thesis deals with the two lower layers; both of them are hosted on the Linux SBC. However, thanks to used inter-process communication mechanism presented below the processes can be run on any configuration of computers and even under

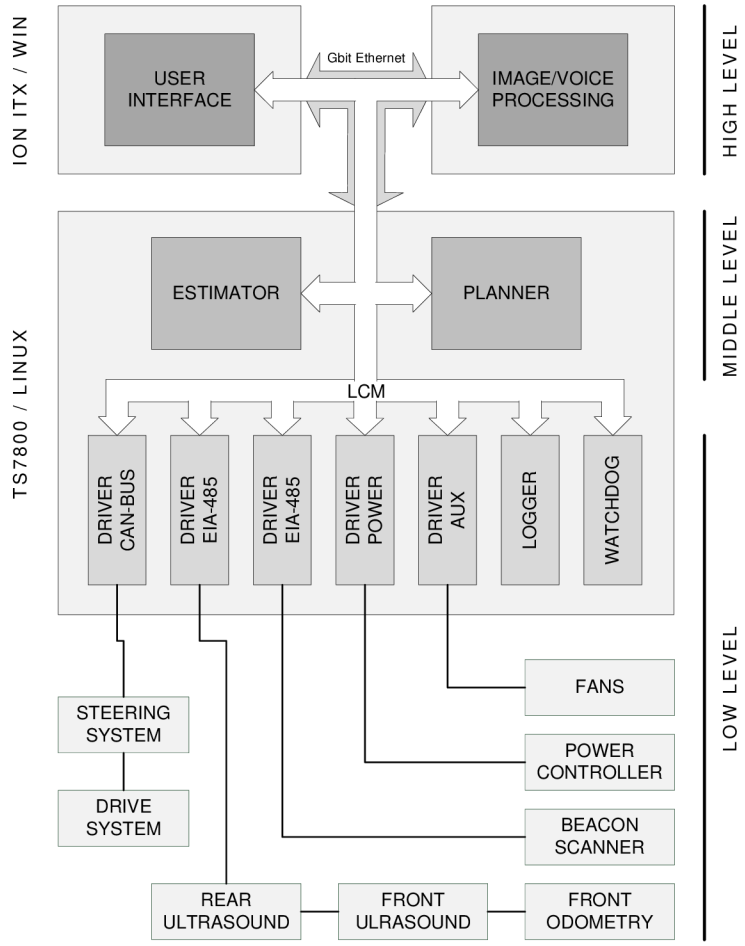


Fig. 3.1: Advee hardware and software modules scheme

different operating systems without any change of communication-related code. Almost every piece[1] of developed software functionality is a part of some *module* — an independent process communicating with other modules using hereinafter described mechanisms.

3.1 Development tools used in lower levels

There is a distinct difference in development philosophies between the high-level and the other software means. The high level runs on a dedicated computer equipped with a rather multimedia operating system (Windows 7), should provide the customer with a rich user experience and its software implements the whole "personality" of the robot. That is why the programming tools used for its development

[1] This includes all operational software equipment; tools used for debug, maintenance and diagnostic purposes naturally do not follow the modular pattern.

should enable in particular high level of abstraction from implementation details — chosen .NET framework with C# as its main language fits these demands very well.

On the contrary, the lower levels should act as a reliable fundament that runs efficiently to meet real-time deadlines and to not bring another transport delays into the data flow. C and C++ languages suit these demands well — providing rather low abstraction over the operating system services allow to use the hardware economically. Longer time needed to develop and debug a functional program is paid back by its predictable and reliable behavior.

The development for the TS-7800 SBC is by the manufacturer simplified by providing a complete bundle for Windows operating systems containing the Eclipse integrated development environment (IDE) supplemented by the GNU Compiler Collection (GCC) in a form of cross compilers producing ARM machine code under the Windows environment. The Eclipse IDE is extended by several plug-ins that make the development more comfortable (e.g. a remote system support that allows to connect to the target machine from within the IDE).

Highly important for a collaborative team work on the software is a version control system. There are dozens of versioning systems, among which several of them come on force due to their usability and support: Git (developed by Linus Torvalds, the original author of Linux), Mercurial (initially aimed to supersede BitKeeper used for Linux kernel versioning) and Subversion (often abbreviated as SVN). Whereas the architecture of the former two systems is distributed, allowing versioning without access to the central server too, SVN follows the older client-server paradigm. For purposes of Advee software development, Subversion has been chosen as being more user-friendly and supported by GUI clients.

There are three SVN repositories to host different parts of Advee's design — the HW (hardware; design of the circuitry), FW (firmware; software for microcontrollers) and SW (software run on computers and diagnostic smartphones). The SW repository is divided into four parts: android (diagnostic SW for Android OS), linux (low- and middle-level code), win (high-level software) and messages (platform-independent message definition files).

3.2 Folder structure of the control system

There is a folder tree that hosts the control system on single place — binaries, configuration files and logs are placed together which simplifies deployment. The root folder of the structure is (according to Linux conventions) placed to `/opt/advee/`. It contains following subfolders:

bin/ Executable system files — all modules, the **watchdog** and diagnostic tools

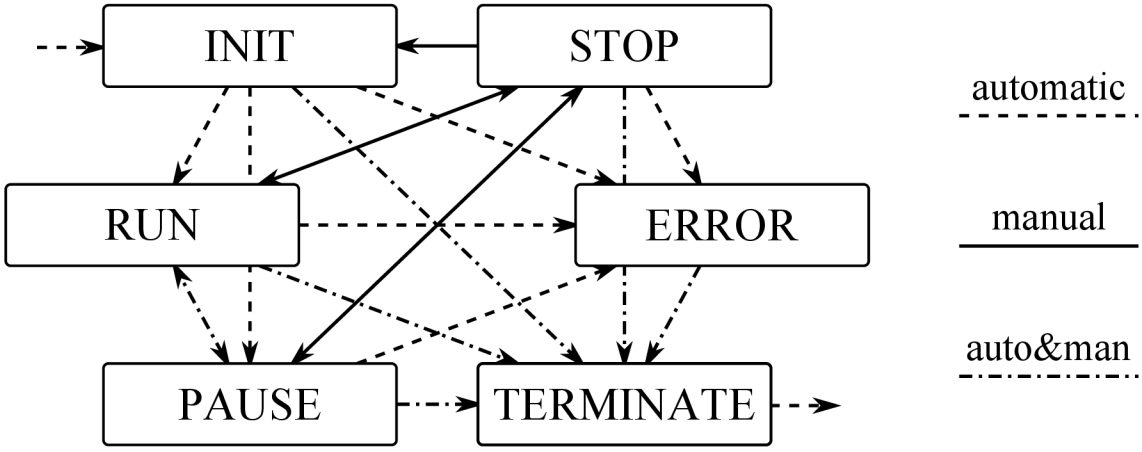


Fig. 3.2: Module state transition diagram

conf/ Configuration files influencing behavior of the system:

modules.txt A list of modules that should be run by the **watchdog**

srfs.txt Definition of ultrasonic sensors positions and communication IDs

log/ Root folder holding a number of log directories, for example:

2011_05_12_10_58_02/ Directory containing log files for robot operation started May 12, 2011 at 10:58:02

2011_05_12_14_16_34/ Directory containing log files for robot operation started May 12, 2011 at 14:16:34

...

The root directory holds also several shell scripts providing auxiliary functionality, e.g. a script executed periodically by `cron` that checks whether the **watchdog** runs and eventually starts is.

3.3 Standard module architecture

Each module supposed to run on the control system computer has to follow several conventions that enable uniform handling of the modules by the **watchdog** module that guards other modules from misbehavior (it can e.g. restart an unresponsive module — see Section 3.6 The watchdog process).

Definitions common to all modules can be found mostly in header file `advee.h`. Among others it contains definition of states the module can take (INIT, RUN, PAUSE, STOP, ERROR and TERMINATE); they form a state machine common

to all modules. Allowed state transitions are shown on Figure 3.2, meaning of the individual states is as follows:

INIT First state active after module startup; configuration loading and resources preparation is accomplished

RUN Operational state of the module; the module executes its functionality

PAUSE Module operation is suspended; other (superior) modules can however return the module to RUN state

STOP Module operation is manually suspended; the module is prevented to return to RUN state other than manually

ERROR The module tries to recover from fault situation

TERMINATE The module shuts down and cleans up used resources

After initializing, the module can enter either RUN or PAUSE mode. STOP state is accessible using manual commands only — it is currently used to temporarily disable the path planner when manual driving is needed. The ERROR state is not used for now — no complex error recovering is employed. For use in code, the states are defined as an enumeration type `module_state_t`:

```
1  typedef enum module_state_t
2  {
3      MODULE_STATE_UNKNOWN = 0,    //> unknown state
4      MODULE_STATE_INIT = 1,       //> initialize
5      MODULE_STATE_STOP = 2,       //> stopped
6      MODULE_STATE_RUN = 3,        //> running
7      MODULE_STATE_ERROR = 4,      //> error
8      MODULE_STATE_TERMINATE = 5, //> clean up & terminate
9      MODULE_STATE_PAUSE = 6       //> paused
10 } module_state_t;
```

Exit codes returned by the modules are subject to standardization as well. Defined codes are an extension of the Unix standard determining 0 (zero) as success and non-zero result as error. Furthermore, values from 1 to 127 are reserved for custom use; when the process is terminated by a Unix signal, the exit code is the signal number plus 128. In `advee.h` the custom range is divided into subranges of general error types:

```
1  typedef enum module_exit_t
2  {
3      MODULE_EXIT_SUCCESS = 0x00,     //> error-free end
4      MODULE_EXIT_RESOURCE = 0x10,    //> resource allocation error
5      MODULE_EXIT_DEVICE = 0x20,      //> device failure
6      MODULE_EXIT_OTHER = 0x40        //> other fatal error
7  } module_exit_t;
```

Each general type can then represent a module-specific error by adding a number from range <1; 16>.

3.3.1 Error reporting library `liberror`

Operational error reporting is further provided by the `liberror` support library loosely inspired by a similar standard function set provided by GLib [6]. It defines a custom structure type `error_t` that encapsulates a particular error instance. Each error can be as usual described by its number and textual form (message). Moreover, flags representing fatality, type and severity of the error and file path and line number were the error occurred is provided.

```
1  typedef struct error_t
2  {
3      int number;
4      char *msg;
5      char fatal;
6      error_type_t type;
7      error_severity_t severity;
8      char *file;
9      unsigned line;
10 } error_t;
```

Error severity levels are defined through the `error_severity_t` enumeration:

Notice A debug information that does not mean an error

Warning Unpredicted state of the algorithm that deserves attention

Error A real error situation, the algorithm cannot continue in operation

Similarly, error types can take one of these values:

Unknown Type not specified or not conforming any of the following types

Timeout Communication failure caused by a non-responding device

Resource Unsuccessful use of a system resource (e.g. memory allocation failure)

Device External device related problem

To report an error, two mutations of `error_set()` function are available: the original `error_set()` and `error_set_const()`. Their behavior differs only in handling of their `msg` parameter of `char` pointer type. The former one frees the memory allocated to hold the message (which is handy if we pass a dynamically generated message string) and the other does not (that is used when the message string is statically defined or should not be freed).

There is also a function for allocating a new `error_t` container (`error_new()`), for freeing and clearing the error (`error_free()` & `error_clear()`), for error propagation / copying (`error_propagate()`) and for printing the `error_t` type to a character string (dynamically allocated by `error_to_string()`). When the use of the LCM library is not prohibited by defining a preprocessor macro `NO_LCM`, there is also a function that send the error using standard message type `common_error_t` (`error_send()`).

The library supports two types of automatic error outputs — when the error is recorded by calling `error_set()` or `error_set_const()`, it can be automatically printed to the standard error output `stderr` and sent to the LCM message bus. Behavior of this automatic handling can be set using the function `error_set_automatic()` that accepts a parameter of type `error_automatic_t`:

```
1  typedef struct error_automatic_t
2  {
3      error_severity_t print_severity;
4  #ifndef NO_LCM
5      error_severity_t send_severity;
6      char *channel;
7      char *module;
8      lcm_t *lcm;
9  #endif
10 } error_automatic_t;
```

When compiled without the use of LCM, it contains only one field `print_severity` that determines level of severity from which the error will be printed out. Similarly, there is a field `send_severity` for use with LCM — in such case also the pointer to LCM instance, channel and module name have to be set.

3.4 Inter-module communication

Inter-module communication mechanism (or generally inter-process communication, IPC) belongs to the most important parts of the system. There are several roles that such mechanism should be playing:

- define *unambiguous interface* between modules with guaranteed inter-language and inter-operating-system compatibility
- provide *high-throughput* and *minimal-latency* communication channel to allow obeying real-time constraints
- support *multiple platforms* — at least Linux and Windows, C and C#.NET

As already stated, whole software equipment of the robot is spread over minimally two computers. This fact disqualifies many traditional IPC methods as for

example shared memory, files or pipes. While the computers are interconnected through Gigabit Ethernet, a solution based on the standard TCP/IP protocol suite seems to fit well.

After filtering out unsuitable mechanisms, the next decision is conceptual — choosing the paradigm. Considering that most of the data flowing through the robot control system is of periodic nature, the publisher-subscriber model suits the system better. Compared to the client-server paradigm, the publisher-subscriber model realizes the inter-process communication as a bus shared among all modules where each module can publish its messages and listen to any other messages. It can be implemented as "push" or "pull" — the "push" mechanism delivers data instantly while the "pull" method uses clients asking periodically for any new data that effectively ruins the concept of a bus. To ensure lowest possible latency, "push" method has to be used.

Because individual parts of the robot software are written using diverse languages, it is essential to maintain unambiguity of the communication between different interface implementations. This is best achievable using dedicated message definition files and programming language specific generators of message-related code (marshalling, unmarshalling, publishing etc.).

Having all these demands in mind, minimalistic yet powerful *LCM* [11] has been chosen as an optimal mechanism for message passing in Advee.

3.4.1 LCM: Lightweight Communications and Marshalling

LCM is a set of libraries developed originally by the MIT DARPA Urban Challenge Team for their autonomous vehicle Talos that finished the DARPA Urban Challenge in fourth place. It was used as a backbone communication bus for the whole platform; its qualities can be illustrated on the fact that it managed to interconnect 70 separate modules resulting in average operational traffic of 16.6 MB/s built by sending 6,775 messages per second.

The library is composed of three main functional blocks:

- *message definition* using a C-like definition language that can be compiled into language-native message support files using the lcm-gen tool
- *message marshalling/un-marshalling* — conversion between native and byte representation with defined endianness and runtime type safety checking
- *communication itself* based on UDP multicast

Originally, C/C++, Java, Python and MATLAB on POSIX systems (Linux, OS X, Cygwin, Solaris, etc.) were supported [14]. To fit the needs of the control system

completely, .NET framework support had to be added. Communication with the LCM authors led to incorporating the .NET port to the library and participation on further development of the project.

Message definition

All messages have to be defined in a custom definition language that closely resembles the C language. The following code snippet provides definition of the most frequent message in the system:

```
1  package common;
2
3  struct data_state_t
4  {
5      const    int8_t     UNKNOWN = 0, INIT = 1, STOP = 2, RUN = 3, ERROR = 4, ↩
           TERMINATE = 5, PAUSE = 6;
6
7      int64_t  timestamp;      // UNIX timestamp
8
9      string   module;         // software module name (e.g. driver_canbus)
10     int8_t   state;          // UNKNOWN, INIT, STOP, RUN, ERROR, TERMINATE
11 }
```

The definition begins with optional package specification — this allows to categorize the messages into packages / namespaces known from higher-level programming languages. The message itself is a complex type, a structure, that is composed of primitive or other complex data types. All types can be also used to define multi-dimensional, fixed- and variable-dimension arrays.

This thesis however does not intend to be a reference manual of the message definition language — the details can be found in [10] or [11]. The most recent information on the library can be found on its Google Code page [14].

Message type specifications are stored in files named same as the type with extension ".lcm". These files are then fed to the utility *lcm-gen* that converts the LCM type specification into a language-dependent implementation.

Data marshalling / un-marshalling

The message instance has to be converted into a byte array to be sent through the network. This is done by generating language- and message-specific support files by the *lcm-gen* tool.

Used binary data format is extremely compact and efficient — in fact there is no formating information present in the marshalled message. This is also the reason why LCM has to have built-in type safety — message definition mismatch between the source and the recipient would have catastrophic consequences. On the contrary, LCM is able to detect any discrepancy and report a type error.

UDP multicast communication

The primary transport-layer protocol of the LCM communication subsystem is the User Datagram Protocol (UDP), a standard member of the Internet Protocol Suite. On the contrary to the Transmission Control Protocol (TCP), this minimalistic connectionless protocol does not provide any reception acknowledging or congestion control which makes it extremely powerful. On the other hand, the communication is on principle unreliable and the order of received datagrams is not guaranteed. The probability of an error in a properly-cabled Gigabit Ethernet network is however very low — the authors of the LCM state an estimation of the bit-error rate of 10^{-12} [10].

Unlike TCP, UDP is also capable of different routing schemes than the obbligato unicast. The LCM library employs IP multicast because of its great scalability — the amount of transmitted data does not grow with growing number of recipients.

TCP-based communication

The TCP-based message transport is not intended for operation; instead, it allows for processing of log files as quickly as possible while avoiding packet loss.

LCM C#.NET port

The LCM port for the .NET framework could have been implemented generally in two ways. The former one included utilization of the C port for Windows completed just at that time[2] — a wrapper would have to be used to form a layer between the unmanaged[3] LCM binary and the managed .NET world. The latter method is way more complicated but brings the advantage that the whole code is managed.

While choosing the suitable solution, a combination of two factors was crucial: the LCM features a native Java implementation and the Microsoft Visual Studio for .NET 2005 disposes of a Java $\rightarrow$ C# conversion wizard. It allows to process the Java code and allow its structure without the need of a hideous manual rewriting. Of course Java has slightly different naming conventions and the wizard does not produce extraordinarily code so that some post-processing is necessary, but the amount of needed work is greatly reduced.

The architecture mostly matches the Java implementation; the root namespace is called `LCM` and contains three subspaces:

LCM.LCM Contains LCM implementation and directly related support classes

LCM.Server A runnable LCM TCP provider host server

[2]Usage of LCM under windows was possible even before — through the *cygwin* environment.
[3]Managed is the CIL byte code running on the .NET virtual machine (CLR)

LCM.Util At the moment containing only class `BitConverter` similar to the built-in `System.BitConverter` class but with the support for data endianness

The main class `LCM` does not provide any communication nor marshalling functionality — it just encapsulates the provider list and manages channel subscriptions. Each provider has to conform to the `Provider` interface that guarantees uniform access to the functionality provided by all provides. The UDP multicast provider is implemented by the class `UDPMulticastProvider`. The TCP functionality it provided by the provider class `TCPProvider` and the server class `TCPServer`.

Two more interfaces reside in the main namespace — `LCMEncodable` defines functions that every message object has to implement. Similarly, `LCMSubscriber` defines the interface of objects that can be subscribed to receive messages. While this interface is primarily destined for the user code, one standard class implements it too — it is the `MessageAggregator` that allows message buffering and synchronous access.

3.4.2 Naming conventions in communication

It is largely advantageous to define a set of naming conventions also for the communication means — not for operational but debugging and diagnostic purposes. There are two subjects of conventions — channel names and message type names.

Channel names are built by three uppercase parts concatenated by underscores: the *software-level* prefix (`LL` for low-level, `ML` for middle-level, `HL` for high-level, `CM` for common and `SIM` for simulational functionality), *channel type* identificator (`CMD` for command, `DATA` for synchronous data and `EVENT` for asynchronous data) and the *name* itself. For example, one of the standard messages is the module state announcement channel `CM_DATA_STATE`.

The LCM-specified convention for messages naming is very similar except of being lowercase and the trailing `_t`. The SW-level prefices are realized as message namespaces (`lowlevel`, `middlelevel`, `highlevel`, `common`, `simulation`). The C language does not support namespaces, the namespace is thus bound using an underscore too, e.g. the message sent in channel `CM_DATA_STATE` is denominated `common_data_state_t` in C. In languages that support namespaces, the namespace is utilized: the state message is named `common.data_state_t`.

3.4.3 Communication interface of the modules

On the top of LCM there are two standard messages defined in package `common` that each module has to implement:

data_state_t A module state announcement message sent every second

cmd_state_t A command message that tells the module to change its state

Because every module uses `liberror` for error reporting, it also usually transmits messages of another standard type `common_error_t`.

3.4.4 User-space library `libmessaging`

To support the functionality of the LCM library, another custom library has been introduced: the `libmessaging` library. Its main purpose is to wrap C-specific message files generated by the *lcm-gen* tool and compile them to one binary file statically linkable to executable projects. In addition, the library also implements function `messaging_get_timestamp()` to get `int64_t`-represented timestamp and defines several preprocessor macros to simplify subscribing, unsubscribing and handler functions declaration.

3.5 List of implemented modules

The current low- and middle-level functionality is divided into the following modules:

driver_aux Driver of auxiliary devices, currently comprising of the fan controller only. Future development should bring more helper devices, such as lighting of the robot of temperature measurement; it will be then decided whether to place the new functionality also to this module or whether to introduce new modules (the latter solution is more probable).

driver_canbus CAN-bus devices driver; at the time, only motion-related devices are interconnected by the CAN bus. As soon as more devices will be moved to the CAN bus, the driver will be probably renamed to **driver_motion**.

driver_joydev An obsolete driver formerly used for manual control of the robot using a joystick / gamepad. The standard `joydev` kernel driver was utilized.

driver_power Driver communicating with the **BR-PSC1** power source controller through a full-duplex EIA-485 (sometimes denoted as EIA-422) line. It can get and set the state of all driven power sources; new future will bring current measuring on almost all channels.

driver_rs485a The first of two half-duplex EIA-485 drivers — the channel A hosts devices with higher polling frequencies. Both **BR-SD1** distance readings concentrators and the **BR-SO1** odometry board are governed by this driver.

driver_rs485b The second half-duplex EIA-485 driver that handles slowly polled devices. Currently only the **BR-SBS1** beacon scanner is connected.

estimator Middle-level module that fuses sensory information using the Extended Kalman Filter and continually updates the estimate of robot position.

planner Another middle-level module that emits motion commands on the basis of position estimate, goal position, surrounding obstacles and operation mode.

watchdog Subsidiary module that is responsible for running other modules and guarding their responsiveness; is described in detail in the next chapter.

Modules **estimator** and **planner** have not been developed by the author of this thesis and therefore will not be closely described.

3.6 The watchdog process

As already mentioned above, the state of modules intended to be run is continually monitored by a process-control module, the **watchdog**. It is the only control system process launched on system startup and it handles executing of modules listed in the configuration file `modules.txt`. Running of the **watchdog** is also supervised: a shell script is being executed as a cron task every minute[4] to check that the **watchdog** process is running and eventually restart it.

The configuration file `modules.txt` is a simple text file that contains a list of module names and executable file names (that are mostly the same as the module name):

```
driver_aux      bin/driver_aux
driver_canbus   bin/driver_canbus
...
```

For each listed module it creates a thread that handles only one particular process; its workflow is denoted in Algorithm 3.1. The thread runs in an infinite[5] loop that begins with a check whether the process should run (line 2). If positive, it sets signals actions and tries to send `SIGTERM`[6] to all processes named identically as the module (line 7). Returned `pid` differing from `-1` means that the module was running; line 9 guarantees that the module will be killed even if it does not react to

Algorithm 3.1 Watchdog process thread algorithm

```
1: while !terminate do
2:   if command = TERMINATE then
3:     sleep();
4:     continue; {loop until the module should be executed}
5:   end if
6:   setSignalActions();
7:   pid ← sendKill(module, SIGTERM);
8:   if pid ≠ −1 then
9:     killTheProcess(module);
10:  end if
11:  pid ← executeTheProcess(module);
12:  exitCode ← waitForResult(pid);
13:  evaluateExitCode(exitCode);
14: end while
```

the termination signal. Then the module process can be executed (line 11); after it exits, returned exit code is evaluated.

The **watchdog** defines three phases of module lifecycle: `RUNNING`, `TERMINATING` and `TERMINATED`. Another thread periodically (every 200 ms) runs through the module list and for modules declared as `RUNNING` checks timestamp of their last status announcement message. In the case the last message arrived more than two seconds ago, the algorithm marks the module as `TERMINATING` and send `SIGTERM` to the process. In next iteration it verifies that the module terminated for real (sets its phase to `TERMINATED` in such case) and kills it if not.

Algorithm 3.1 mentions on line 2 pseudocode variable `command`; in fact it is a field of the structure `watchdog_process_t` and its value is set in the handler of LCM channel `CM_CMD_STATE` messages — it is the state that other parts of the system the particular module to be in. Another such field is the `state` that gets actualized in the `CM_DATA_STATE` handler.

3.7 Remote diagnostics and simulation support

Inherent feature of the LCM's transport layer (UDP multicast) is that virtually unlimited number of recipients can subscribe to messages from outside of the system

[4]The standard Linux `cron` daemon does not provide sub-minute intervals.

[5]Infinite only as long as the **watchdog** module should run; when its state is set to `TEMINATE`, it stops also process threads.

[6]A standard POSIX signal that tells the process to shut down.

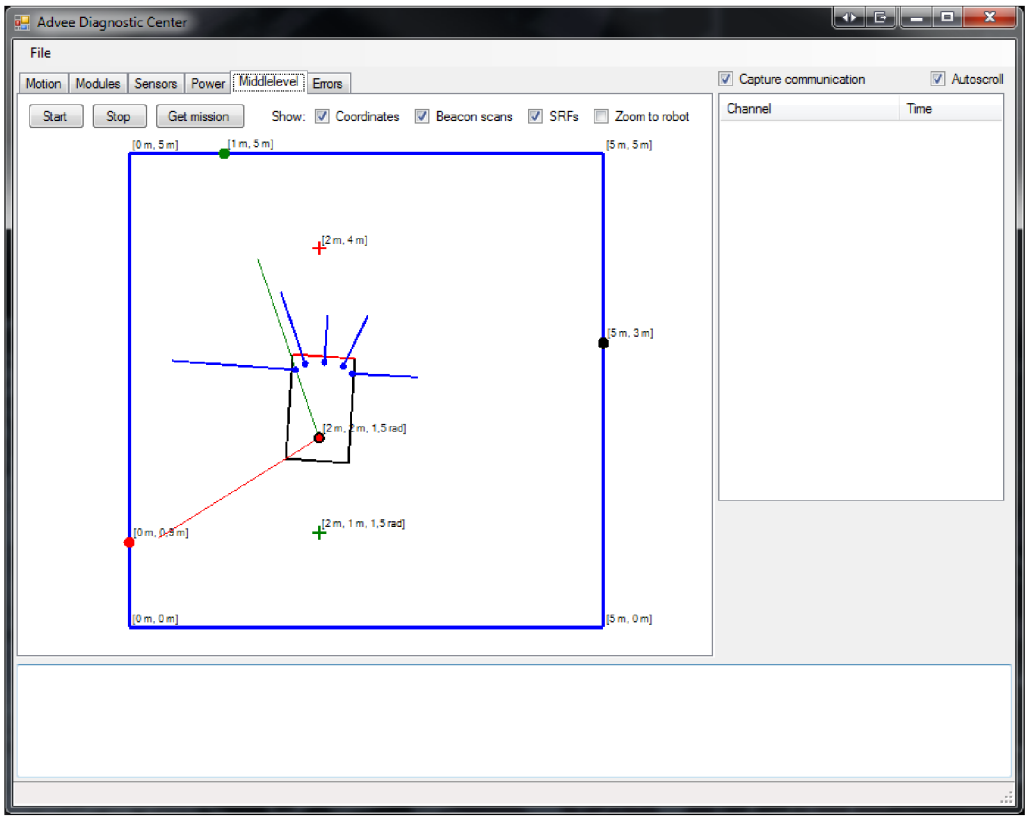
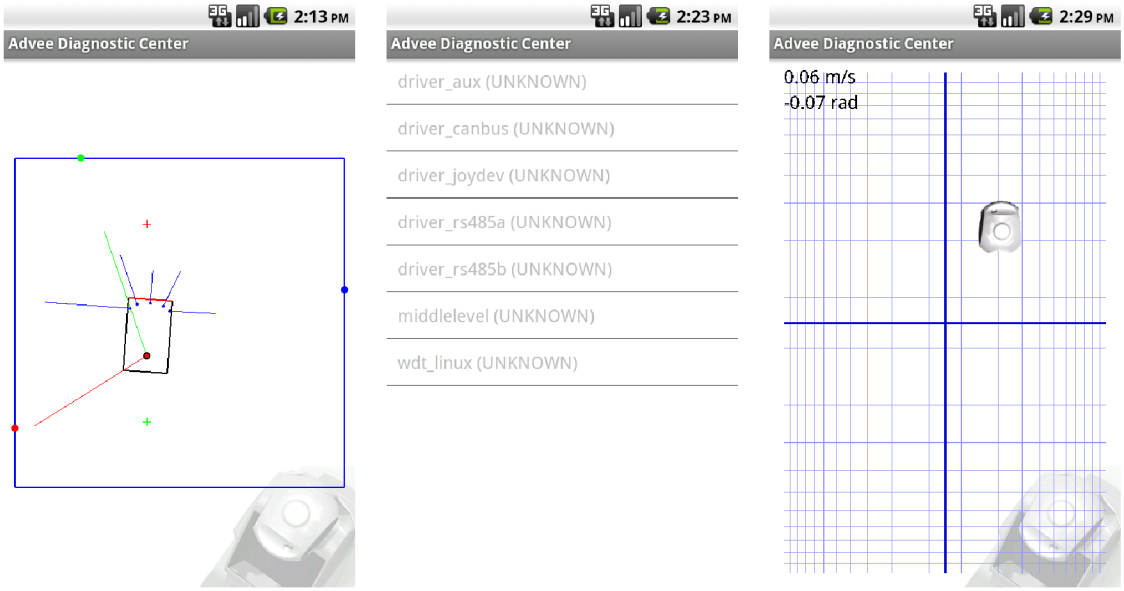


Fig. 3.3: C#.NET diagnostic center

without any negative impact upon the system itself. It is then easy to implement a non-obtrusive diagnostic system that can monitor system modules and visualize important quantities. Currently there are three separate diagnostic applications: for console, Android-based smartphones and the .NET environment.

The console utility named simply **diagnostic_center** is the least complex one and is designed for quick diagnostic operations through a SSH session. It exploits the well-known `ncurses` library to easily build a responsive user interface including a multi-dimensional menu etc. In the default view, a summary of the robot state is given — its position, actual motion command and proximity sensors readings. The robot can be also manually driven using keyboard arrows and homing of the steering servo can be performed. The other view gives an overview of recent error reports in the system.

Another diagnostic center is written using the C#.NET environment. It features the most complete overview and control over the robot from all diagnostic centers and serves as the main debugging tool. Its functionality is divided into six tabs: motion control, modules list (state viewing and setting), sensory data display, power subsystem control, middle-level software layer control (Figure 3.3) and error list. In addition, right side of the window provides a list of recent messages.



(a) Map view (b) Modules list (c) Manual motion

Fig. 3.4: Android OS diagnostic center

Most frequently used is the third diagnostic center destined for the Android OS and written in Java (Figure 3.4). It has been created to provide a practical tool for manual driving and diagnostics during commercial operation of the robot. It encapsulates all routinely needed functions ranging from manual motion control to graphical view of the robot state within a map and modules control.

A communication logger is now being implemented simply by listening to all messages and writing them to a file, allowing for later playback.

3.7.1 Simulation

As the interface of every module is unambiguously specified by consumed and produced messages, it is possible to replace any module by its simulation equivalent without letting know the rest of the system[7]. This is used to emulate outer environment of the robot by replacing the low layer (both software and hardware) by a SIMLIB-based [18] chassis simulator that runs a dynamic model of the robot in a given artificial world.

Several experiments have been done with described simulation environment that have tried to design a new motion planner. The recent approach utilizing a neural network was published in [13] and [8].

[7]When simulation speeds higher then real are needed, synchronously running parts of the system have to be adapted — LCM message package `simulation` is intended for this use.

Chapter 4

Detailed hardware description

While this thesis focuses mainly on the software part of the robot (specifically the lower two layers), it is necessary to describe the hardware that is interfaced and driven by the software.

The functionality of the robot can be divided into several subsystems; several division criterions can be employed, two basic ones are used in this thesis. The former one is rather logical and is visualized on Figure 4.1 — it partitions the system according to function, not implementation. This chapter follows the logical way to give a compact overview of robot sensors, actuators etc. On the contrary, the thesis is structured "technologically" because the implementation can be better described in that way.

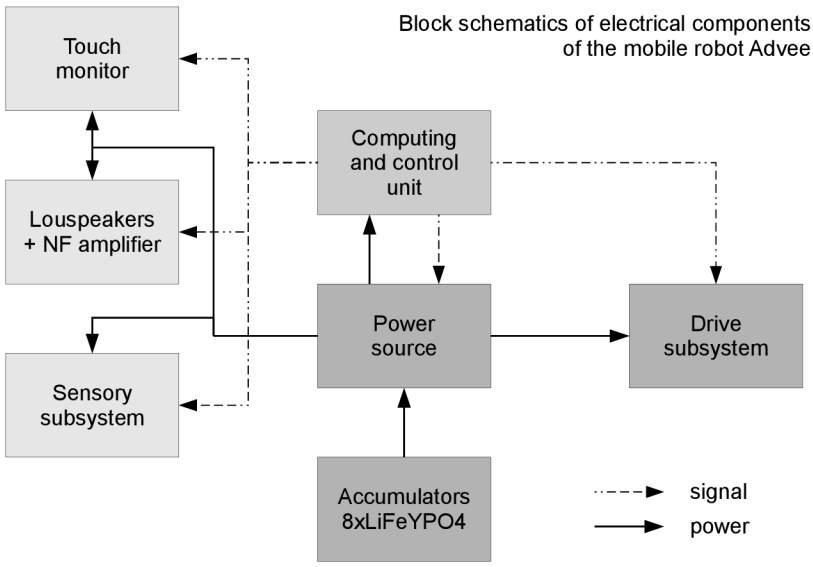


Fig. 4.1: Simplified schematics of Advee's functional units

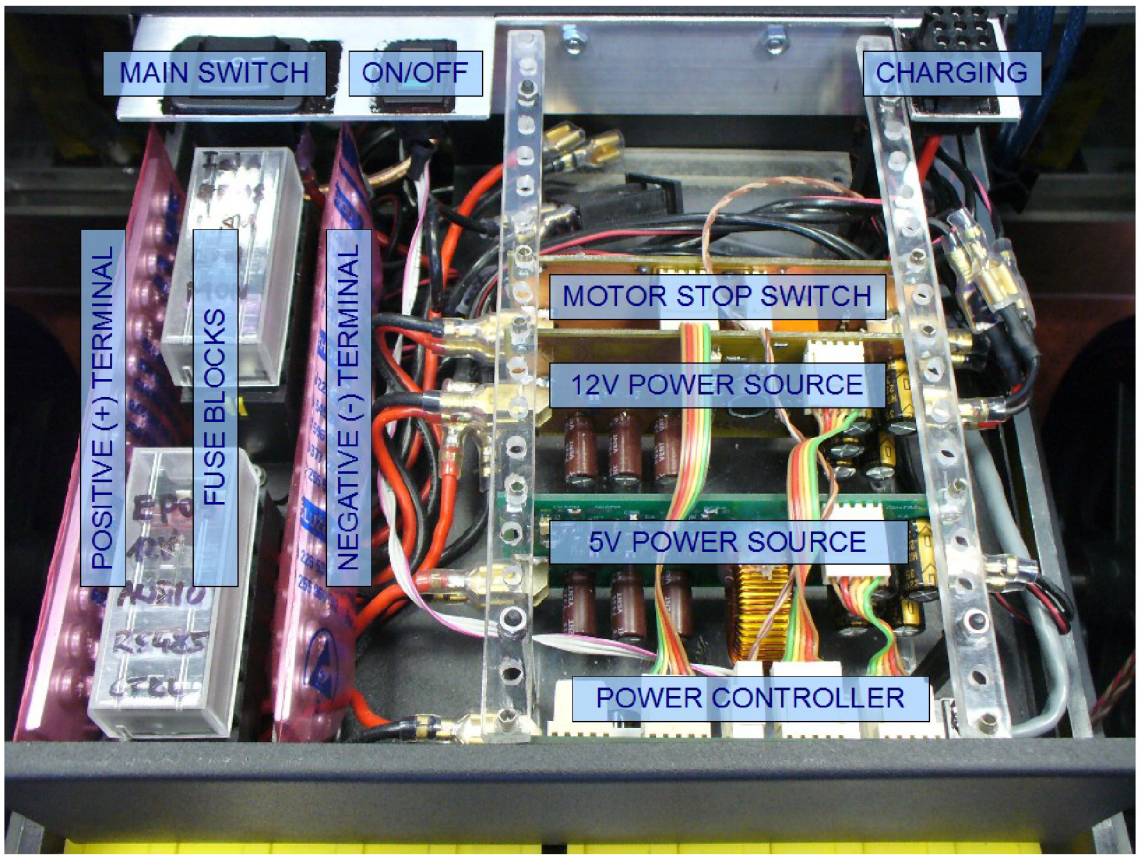


Fig. 4.2: Power box with some of power modules fitted

Some of the hardware means were available to be bough (e.g. the computers, the drives with control units etc.) but significant part of the hardware had to be custom designed. These modules can be easily recognized as their names follow a simple naming convention: prefix **BR-** concatenated to a abbreviation of module function.

4.1 Power subsystem

The whole robot is powered by a 8-cell LiYFePO4 battery pack. Each cell provides capacity of 40 Ah at voltage of 3.65 V (fully charged, operational voltage is lower but minimally 2.5 V), which gives total exploitable energy of approximately 1100 Wh.

As some components cannot be run from this relatively high voltage, a modular power source subsystem was developed. It is comprised of an 8-channel power controller **BR-PSC1** (abbreviation of Power Source Controller) and a range of slave modules (Figure 4.2 shows them in the power box):

BR-PSM1 Power Source Module — complex power source module with over-current and under-voltage protection built around the LM5116 synchronous buck controller.

BR-PSM2 Simple and cheap power source module for auxiliary purposes; built using the LM2596 buck converter.

BR-PSS1 Power Source Switch — dual SSR-based switch for power control of the touch monitor, audio amplifier and computers.

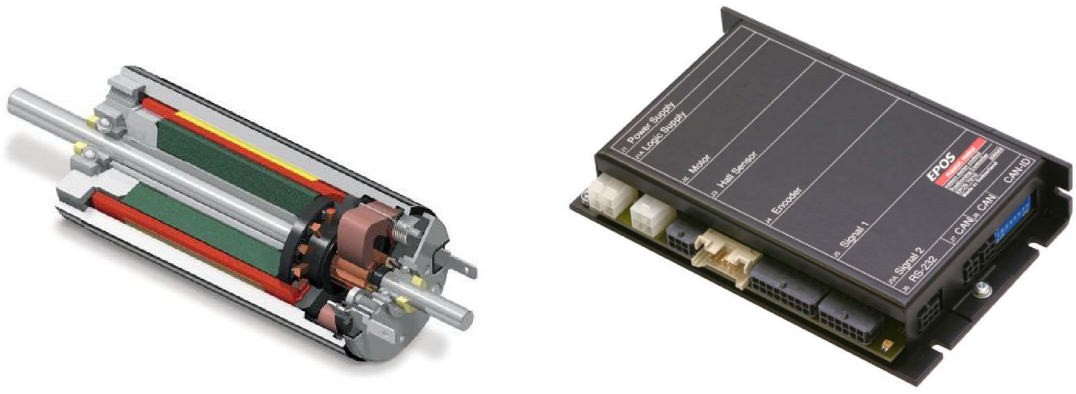
BR-PSS2 Bistable-relay switch used to engage and disengage motor power feed. An external logical signal `PAUSE` is provided that can literally pause motion of the robot in case of any danger. Both bumpers and both **BR-SD1** units can set the signal; a wireless security stop button is considered to be built.

All modules except the PSS2 module feature galvanic isolated current sensing that allow to monitor power consumption of almost all modules. Both power sources have transient voltage protection.

The Power Source Controller is a key component as it directs the whole power of the robot. Start-up and halting sequences are implemented to the controller to achieve deterministic procedure of applying power to individual subsystems. The start-up procedure proceeds from simple to complex devices — it guarantees that a slave component is being initialized before a master component.

4.2 Motion subsystem

Motion of the robot is ensured by two actuators: the drive motor and the steering motor. The selection process of both of them is described in [20]; generally speaking, power, torque, size, weight and communication interface were lead parameters.



(a) RE-family construction (b) EPOS 70/10

Fig. 4.3: Parts of the Maxon drive system [27]

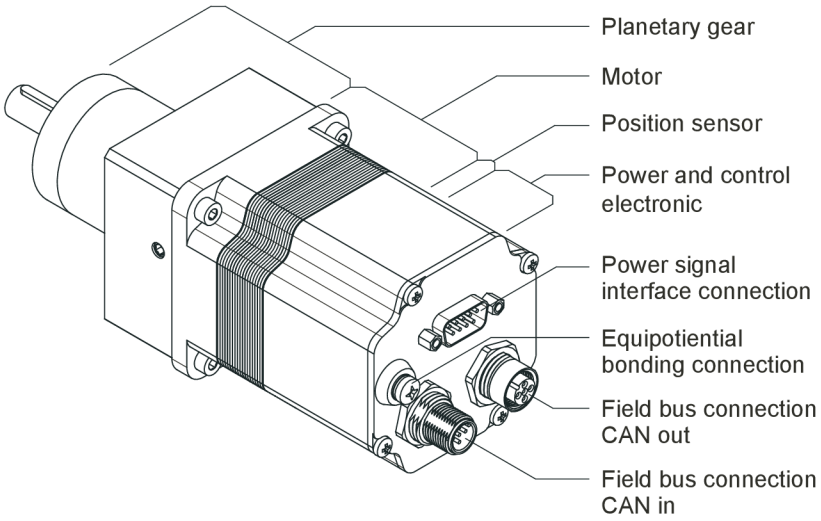


Fig. 4.4: Berger Lahr IclA N065 Intelligent Compact Drive drawing [3]

The propulsion unit is formed by a Maxon RE50 brushed DC motor with a GP52C 26:1 planetary gearhead and a HEDL 5540 500ppr encoder. It is governed by a Maxon EPOS 70/10 position controller equipped with CANopen-compliant CAN-bus interface.

For steering purposes, a Berger Lahr IcIA DC024 1-040 Intelligent Compact Drive is employed. It is an all-in-one device that also features a CANopen-compliant CAN-bus interface.

4.3 Sensory equipment

Perception of the surrounding environment is supplied to the robot by individual parts of the sensory subsystem. The middle-level software equipment makes use of two types of information: data used for position estimating and data used for obstacle detection.

Position estimates are built by module **estimator** that runs an Extended Kalman Filter (EKF). Input data are again of two types: odometry readings and beacon responses. Front wheels (that are not driven and thus are not a subject to slippage) are equipped with incremental rotary encoders (IRCs) that convert rotary motion to a sequence of impulses. These impulses are decoded and counted by a **BR-SO1** (Sensor Odometry) board, polled through a EIA-485 bus and converted to general speeds by the **driver_rs485a**. Resulting translational and rotational speed is filled to a `lowlevel_data_odo_t` message and placed to the LCM bus.

Absolute position is computed using angular signals from infrared (IR) beacons

received by a **BR-SBS1** (Sensor Beacon[1] Scanner) module. Polling for the data (EIA-485 bus again) and placing them to a `lowlevel_data_beacons_t` message is provided by module **driver_rs485b**.

Obstacle avoidance of the robot i assured using 16 ultrasonic range finder modules SRF-08 placed all around the chassis approximately 15 cm above terrain. This guarantees that even low-profile obstacles are detected and accordingly handled. The range finders form two groups of eight devices interconnected by a I^2C bus that are each managed by a **BR-SD1** (Sensor Distance) board. This board also samples distance readings from two IR range finders Sharp GP2Y0A41 placed in corners of the robot and measuring the distance to ground. During normal operation, the distance should remain approximately constant — in case of markedly different results the motion of the robot is stopped.

The last-instance collision avoidance devices are two bumpers placed in the front and rear parts of the robot. As being attached movable, four switches on each bumper activates the `PAUSE` signal and motion of the robot is interrupted by disconnecting the drive motor from power.

Advee also disposes of a HD camera built into his eye and used to find faces in the picture — this sensor is however fully handled by the high-level computer and will not be discussed here.

4.4 Computer box

The computational and control "heart" of the robot is enclosed to a acrylic-glass box that is easily removable from the robot and in case of failure also interchangeable for a replacement one. Air circulation through the computer box is provided by two 92mm fans — the fans run at circa 20 % of their maximal power due to energetic efficiency of the used computers.

The computers are formed by the TS-7800 embedded Linux computer and one or two Mini-ITX form factor high-level computers. Single Zotac IONITX-F-E with a M2-ATX-HV power source and OCZ Agility 30GB SSD disc is currently exploited. Gigabit Ethernet switch Netgear GS105 provides high-speed interconnection.

The box exposes single power connector (a 12-pin socket Souriau Trim Trio SMS-QIKMATE) that is used to bring supply voltage to all hosted components. All computers use the full battery voltage (maximally slightly under 30 V), the switch and the fans are provided with 12V supply.

[1] Do not confuse with bacon — although both of them were essential for Advee's development...

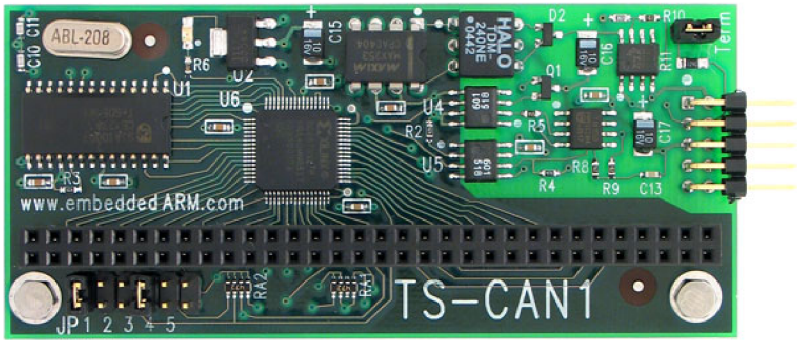


Fig. 4.5: TS-CAN1 PC/104 CAN-bus interface board [24]

4.4.1 TS-7800 peripherals

While the TS-7800 single board computer provides almost whole needed interfaces, there are a few peripherals that supply the missing features.

The most important add-on is the TS-CAN1 PC/104 CAN-bus interface board [24] that utilizes the ISA bus on TS-7800 through the standard PC/104 header. It provides optically isolated CAN-bus interface with the help of the TJA1050 CAN transceiver driven by the SJA1000 CAN controller; the SJA1000 is mapped to the ISA bus using a XC9500XL-family CPLD.

The computer box fans are controlled by a **BR-FAN1** fan controller. It employs a SMBus-interfaced MAX6615 (Dual-Channel Temperature Monitor and Fan-Speed Controller with Thermistor Inputs) to measure temperature in two independent zones and control the fans to keep the temperature at chosen level. The integrated circuit monitors the fans and outputs logical fitness signal; in addition it can maximize power of a functional fan in case of the other fan failure.

The TS-7800 board provides the bus interfaces in the form of PCB headers — modules **BR-CON1A** and **BR-CON1B** provide standard bus connectors mountable inside the computer box. **BR-CON1A** exposes 4P4C[2] connectors for two EIA-485 channels; additionally it provides EIA-422 bus driver for communication with the power controller. There are also two 10-pin headers providing power control of up to two Min-ITX computers inside the box. **BR-CON1B** board features a DB9 CAN-bus socket and a Maxim 1-Wire bus driver (the 1-Wire bus is planned to host auxiliary functionalities - temperature measurement or light control).

[2] 4P4C denotes a telephone-like connector with 4 positions and 4 contacts; also RJ9, RJ10, RJ22

Chapter 5

EIA-485 subsystem

The EIA-485 (formerly RS-485) standard has been exploited already on the preceding mobile robotic platform Bender 2 as the main communication bus. Because Advee's circuitry is partly based on modules originating in B2 and because the bus was well proven, it has been decided to employ the bus also to the new platform.

5.1 Bus description

The standard EIA-485 bus belongs to fundamental communication busses that lay fundaments of industrial data communication [28]; in fact, it is the base of CAN-bus physical layer. The bus occurs both in half-duplex (two-wire) and full-duplex (four-wire) mutations allowing data transfer at rates of 100 kb/s (up to 1200 m) or even 10 Mb/s (up to 10 m).

The line is differential — two wires (referred to as A (-) and B (+)[1]) are used to carry "single" signal. The resulting logic value is the determined from the voltage difference between these two wires. Positive difference $B - A > 200\,\mathrm{mV}$ denotes

[1]This naming convention is widely broken, using A for the non-inverting (+) and B for the inverting (-) signal.

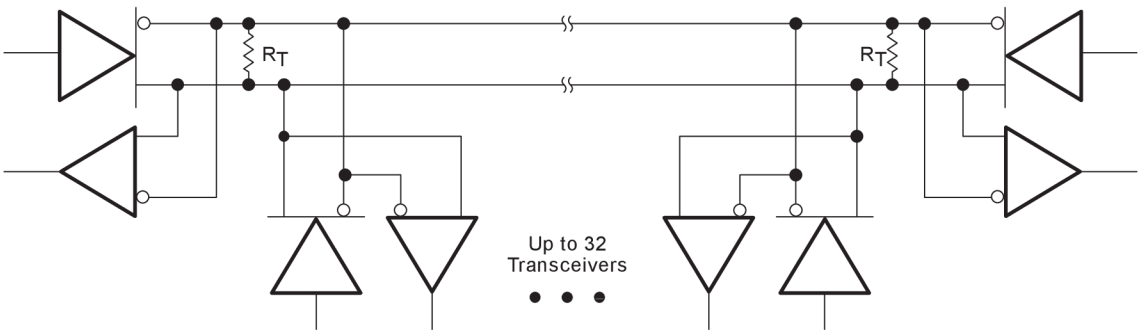


Fig. 5.1: EIA-485 typical application [26]

logic 1 and is called Mark; negative difference $B - A < -200\,\mathrm{mV}$ denotes logic 0 and is called Space.

The standard defines the physical media as a twister pair that makes the bus largely resistent to electromagnetic disturbances. In addition, the twisted pair forms a balanced line — suitable termination resistors with resistivity equal to the cable impedance help to suppress reflections that can otherwise cause data corruption. The typical resistivity of termination resistors is $120\,\Omega$.

A pair of resistors is also used to define quiescent levels on both wires [19]. This prevents the wires from floating in the time when there is no device transmitting. Values in the range of $<470;\ 1000>\,\Omega$ are usually recommended.

The EIA-422 standard can be thought of as of a simpler version of the EIA-485 destined primarily for point-to-point applications. While only one driver is allowed on the bus, up to ten receivers can listen.

5.1.1 Communication protocol

Because the EIA-485 standard does not specify any communication protocol, a simple one has been developed already for utilization on the Bender 2 platform. More features have been added during the time, the result is depicted on Figure 5.2. Message acknowledging is used as well as error detection through a cyclic redundancy check byte (CRC).

Each device on the bus disposes of its unique address; the address `0xFF` is reserved for broadcast purposes. The next byte is the instruction identificator. Again, two standard values are predefined and implemented in every device: `0xFE` to get firmware version and `0xFF` to reset the device. Meaning of the flag byte is different for request and response messages. In the request message the slave device can be asked to resend last message or to not answer with an acknowledge message.

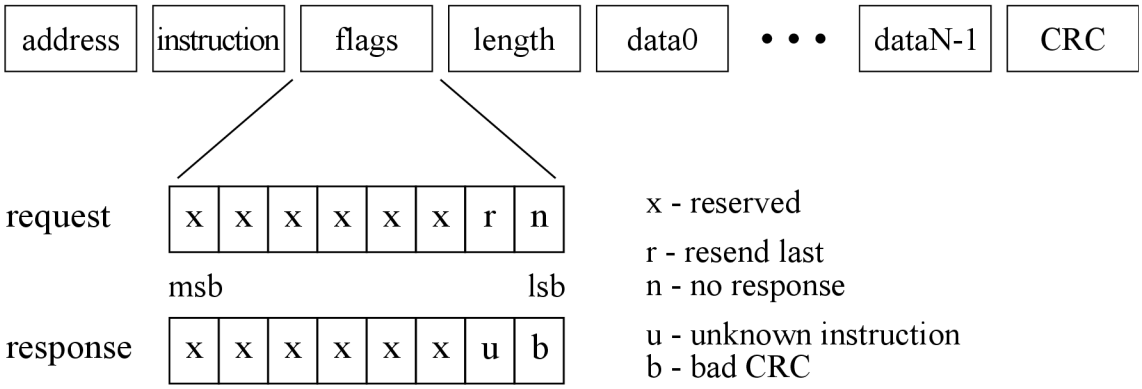


Fig. 5.2: Custom EIA-485 / EIA-422 communication protocol message structure

To reliably mark the start of a message, 9-bit communication mode is employed — the ninth bit is set for address bytes and cleared for the other bytes. The protocol provides capacity of up to 255 data bytes limited by byte-range of the length field. As noted, the data (together with the message header) are secured using the CRC field. Because the CRC is only one-byte and to prevent bus jamming, it is advisable to use rather short messages.

5.2 Kernel driver TS-UART

Both EIA-485 channels are handled by the extension serial ports implemented in the on-board FPGA [25]. Each such port is accessed using two 16-bit registers `STAT` and `DATA`. The former one defines parameters of the port (bitrate, 9-bit mode) and provides transmission / reception flags. The latter serves as data buffer.

While the operation of the TS-UART port is indeed straightforward, the manufacturer of the TS-7800 (Technologic Systems Inc.) provides a kernel-space driver `tsuart` that maps all ports to standard POSIX char devices listed in `/dev`. Devices `ttts0` to `ttts9` provide standard 8-bit access, devices `tt8s0` to `tt8s9` allow to use the 9-bit mode[2].

The driver follows the *termios* API so that the standard system calls can be employed: `open()`, `read()`, `write()`, `select()` or , `close()`. For port configuration, the structure `termios` and functions `tcgetattr()` and `tcsetattr()` (defined in `termios.h`) are to be used.

The utilization of this driver was however not flawless; as noted above, the communication protocol uses the 9-bit port mode. The data could be not read using the 9-bit device while the transmission worked as supposed. It has been found that the driver mishandles reading from the `DATA` register. Both devices share the interrupt number so that both get noticed when some data arrive; however, the data were read correctly for the (unused) 8-bit device and the 9-bit device found only empty buffer. A fix incorporating data sharing between devices had to be implemented; the misbehavior has disappeared. This case shows how advantageous the open-source development model can be — the user can modify and fix system components without time-consuming interaction with the manufacturer, particularly when there is a deadline to meet.

[2]The 9-bit characters have to be coded into two-byte sequences.

5.3 User-space library `librs485`

The library encapsulates the communication protocol described above and provides an easy way to extend the support to a new hardware module. Its functionality is implemented mostly in `rs485.c` and declared in its header file `rs485.h`; each device type is declared in dedicated files.

To use the EIA-485 bus, one has to establish a connection — this is done by calling the function `rs485_init()`. It accepts the serial port device path, a pointer to `termios` structure where the old port configuration should be stored, communication speed and the obbligato double pointer to the error record:

```
1 rs485_connection_t * rs485_init(char *dev_name, struct termios *saved, speed_t ↩
      speed, error_t **err)
```

Firstly it assures GLib threads initialization and allocates a structure of type `rs485_connection_t`. The device is then opened and its file descriptor saved to the connection structure; if this fails, the error is set and the function returns `NULL`. When successful, the function continues with port configuration and switching the peripheral to EIA-485 mode by clearing the 15th bit on address `0xE800000C`. Remaining fields of the connection structure are the initialized and the structure pointer is returned. The line is now ready to be used.

Each device on the bus is denoted by its node record of `rs485_node_t` type. It contains a pointer to the connection structure, address of the node, actual transmit-receive operation and more support items:

```
1 typedef struct rs485_node_t
2 {
3     rs485_connection_t *conn;   //> connection to the bus
4     int address;                //> device address
5     rs485_txrx_data_t *txrx;    //> transmit-receive data and state
6     GCond *cond;                //> response waiting condition
7     int errcnt;                 //> CANbus-like error count (inc-dec)
8     GMutex *lock;               //> node data lock
9 } rs485_node_t;
```

Participation of a device on data traffic on the bus is expressed using a subscription record of type `rs485_subscription_t`. This record is usually filled in a device initialization function (discussed below) and contains the device address, a pointer to received message handler function and a `void*` field used to point to custom data.

To send a message without waiting for response, function `rs485_send_msg()` (or its non-locked variant `rs485_send_msg_unsafe()`) is to be called. It simply places the message data to a 16-bit wide buffer array, adds the CRC and writes the data to the device. Defined enumeration types `rs485_req_flag_t` and `rs485_resp_flag_t`

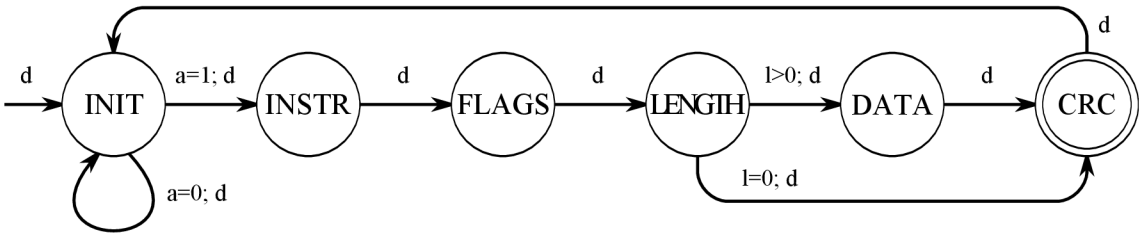


Fig. 5.3: Reception finite state machine diagram

reflect the flag values for request and response messages, the message is defined as structure `rs485_msg_t` holding all fields introduced above as `uint8_t` or `uint8_t *` members:

```
1 typedef struct rs485_msg_t
2 {
3     uint8_t address;
4     uint8_t instr;
5     uint8_t flags;
6     uint8_t length;
7     uint8_t *data;
8     uint8_t crc;
9 } rs485_msg_t;
```

The reception is not that simple. It runs a finite state machine (FSM) inside the thread created and started by the first call to `rs485_subscribe()`. The states of the FSM are enumerated in `rs485_recv_state_t` and the current state of the FMS is held as an item the `rs485_fsm_t` structure. Because of the master-slave nature of the communication, just one FSM is needed per bus — the FSM structure record is thus held as a member of the connection structure.

The finite state machine is fed by data gained using the standard call `read()`; it is called only in the case that a preceding call to `select()` tells that there are pending data. The FSM is driven by the `fsm` member of the connection structure. The received bytes are saved to corresponding fields of a statically allocated message buffer on dependence of the FSM state. For example, in the `INIT` state the automaton waits for a 9-bit word marked with ninth bit set; when such word arrives, the algorithm saves it to the message address field and transitions to the state `INSTR`. The state `LENGTH` can transition either to the state `DATA` (where the data words reception takes place) or directly to the state `CRC` depending on data length. After receiving the last CRC word, the CRC is computed also from received data to be compared. The message is then handed over to the handler function pointed from the corresponding subscription record. The automaton finally transitions to the `INIT` state.

The typical usage pattern of the bus from the master device point of view is

issuing a request message, waiting for an answer and returning the processed answer. This approach is implemented by the function `rs485_send_rec_msg()`. It uses another structure type to hold its data:

```
typedef struct rs485_txrx_data_t
{
    rs485_msg_t msg;            //> message buffer
    char pending;               //> pending tx-rx operation
    rs485_msg_status_t status;  //> transmit-receive state
} rs485_txrx_data_t;
```

The structure member `status` can take one of the values defined in the enumeration `rs485_msg_status_t`:

```
typedef enum rs485_msg_status_t
{
    RS485_TXRX_STATE_OK = 1,        //> operation OK
    RS485_TXRX_STATE_ERROR = 0,     //> sending/reception error
    RS485_TXRX_STATE_TIMEOUT = -1,  //> response timeout
    RS485_TXRX_STATE_BADCRC = -2,   //> bad CRC
    RS485_TXRX_STATE_UNKINSTR = -3  //> unknown instruction
} rs485_msg_status_t;
```

The routine starts with filling the `msg` container with the request message — this is fully controlled by the device driver or any other user calling the function `rs485_send_rec_msg()`. The function is then called; it sets node's `txrx` member with the structure and sends the message in its `msg` buffer. After successful transmission it begins to wait for the node condition (but maximally for `RS485_TIMEOUT` milliseconds). In case the waiting does not timeout, the `msg` buffer is now filled with the response message and can be processed by the caller of the function.

Whilst the `rs485_send_rec_msg()` function waits for the condition to assert, the above described reception FSM processes the response data into a message structure. The message handler function of the device driver is then called; for now, this handler is mostly reduced to calling the default handler `rs485_handle()` common to all devices. This function just copies message data to the `msg` buffer of the device's `txrx` structure and signals the condition.

Beyond this basic functionality, the library provides also helper routines for conversion between a byte array and a standard type (e.g. `rs485_bytes_int16()` or `rs485_bytes_int16()`). Also functions wrapping standard instructions of a device are implemented: function `rs485_get_fw_version()` allows to fetch firmware version (instruction `0xFE`), `rs485_reset_node()` send reset command to a node (instruction `0xFF`) and `rs485_reset_bcast()` broadcasts the reset request to all device on the bus (instruction `0xFF` to address `0xFF`).

5.3.1 Device drivers

Each device that should be connected to the EIA-485 bus has to have its driver that forms a wrapper around the functionality of the device. It defines a device structure type (e.g. `sd1_device_t`) that contains a pointer to an underlying `rs485_node_t` record. The only obligatory function is the message handler whose address is a part of the subscription record. In fact, this function is mostly reduced to retyping the universal `void` pointer to the concrete device record type pointer and passing its node record to the generic handle function `rs485_handle()`.

Almost every driver also provides a init function (e.g. `sd1_init()`) and free function (e.g. `sd1_free_device()`) to allocate the device record and subscribe to reception of its messages. However, the freeing is usually not needed as the system frees allocated memory of a terminated process automatically.

The most frequently used functionality of the driver are the wrapped functions that provide access to the device features (e.g. `sd1_get_srf_data()`). The pattern of writing these functions is almost always the same: the first thing to do is to define a `rs485_txrx_data_t` structure and fill its message buffer with the request message (at least node address and instruction is needed). The TXRX structure is the handed over to the mentioned function `rs485_send_rec_msg()` that completely handles both transmission and reception (and block execution until the reception is complete or a timeout occurs). Then the `status` member value of the TXRX structure is examined the function is terminated if the status is not `OK`. The response messages is found in the same message buffer and can be used to extract its data — the layout of the data is device- and instruction-specific.

A complete overview of implemented instructions and corresponding driver functions for individual devices is given in the Appendix.

5.4 Driver module `driver_rs485a`

The EIA-485 bus A is the "faster" one as it interconnects devices polled at frequency of 4 Hz. These modules are two instances if the **BR-SD1** distance measurement board and one **BR-SO1** odometry board. It is implemented in a single file (as the whole code has only about 500 lines) structured in the following way:

Inclusions The first thing in the source code are inclusions of all needed header files. Besides several standard headers, also `advee.h`, `navitools.h`[3], `error.h`, LCM-related and EIA-485-related header files are used.

[3]The `libnavitools` library encompasses localization- and navigation-related functionality.

Macro definitions Important parameters of the driver (as for example the bus baud rate, serial port device path etc.) are digestedly defined as preprocessor macros, which assures that an eventual modification will be done on one place in the code only.

Type definitions The driver usually needs to hold some structured data — defining custom data types greatly improves readability of the code. In this case, a structure type `srf_t` is defined to hold ultrasonic sensor information loaded from the configuration file `srfs.txt`.

Variable declarations The driver uses a number of global variables that are shared among all its functions and thread. E.g. the module state and its mutex, LCM instance, EIA-485 connection and devices have to be global.

Function prototypes There are two possibilities how to achieve visibility of a function to another function: to define the former function before the latter or to use a function prototype. The driver uses a combination of both approaches.

LCM handlers The declaration of LCM handler functions is clarified by using the `LCM_HANDLER` macro defined in `messaging.h`.

Main function The function `main()` provides an entry point for the module execution. It handles all initialization tasks — it prepares for use the LCM, creates notification pipes, loads information about the ultrasonic sensors from the config file, initializes the EIA-485 bus and all device drivers and creates the data-polling thread. Then it enters a pseudo-infinite loop where the `lcm_handle()` gets called. As soon as the module approaches its termination, it cleans up used resources and ends.

Helper functions Several utility functions are needed — the `SIGTERM`, `SIGQUIT` and `SIGINT` signal handler, a function to safely change the module state etc.

Thread function The function `read_thread_run()` provides similar functionality to the thread as function `main()` to the whole process — it provides both initial setup of some specific resources and the pseudo-endless loop that runs for the rest of the time.

5.4.1 Interruptible waiting

Because the loop in the thread function provides periodic polling, its structure can be divided into two logic parts: it executes the communication task and then suspends for a specific amount of time and yields to the system task planner. However, the

waiting is highly desirable to be interruptible — in the case that the module receives an order to terminate, the waiting should break immediately and let the process end as soon as possible.

This behavior is achieved by using the standard function `select()` to wait for an event on a special notification pipe. This waiting has however specified timeout that end the waiting in case no event happens. The `select()` function works similarly to `sleep()` except it can by interrupted any time by writing a character to the pipe.

The use of a pipe is close to a "hack" — since Linux core version 2.6.27 there is a specialized resource *eventfd* that would fit better. It creates an event object disposing of a file descriptor needed to be used by `select()`. Unfortunately, the last version of the modified TS kernel is 2.6.21 so that this resource is not available.

5.4.2 Data-polling functionality

The effective part of the thread loop begins with a check the module is not in the `STOP` state: in such case the data polling is skipped and the loop continues with another iteration.

Both the front and consequently the rear **BR-SD1** module can be then asked for data using the function `sd1_get_both_data()` that, when successful, allocates and fills buffers for the ultrasonic and infrared distance readings and returns counts of gathered readings. The odometry data polled by the function `so1_get_ticks()` from the only front **BR-SO1** module is handled similarly.

```
1 // front
2 char num_srf = 0, num_sharp = 0;
3 sd1_srf_data_t *data_srf = NULL;
4 double *data_sharp = NULL;
5 sd1_get_both_data(dist_f, &num_srf, &data_srf, &num_sharp, &data_sharp, &err);
```

After the hardware gets polled, it is checked that some data were successfully gained and the LCM messages are prepared. The distance measurements are put into a message instance of the type `lowlevel_data_srfs_t` (both the front and read readings into one message) and placed to channel `LL_DATA_SRFS`.

The odometry data have to be processed first — to maintain the chosen level of abstraction in the robot design, sampled ticks are converted into the form of general speeds with the help of function `speeds_odo2ml` defined in `libnavitools`. The translation and rotational speed is then assigned to an instance of the `lowlevel_-data_odo_t` and transmitted to channel `LL_DATA_ODO`.

5.5 Driver module `driver_rs485b`

The "slower" EIA-485 bus B host only one **BR-SBS1** beacon scanner module that is polled for data at 1 Hz. The driver follows principles described hereinbefore so that its general architecture is almost the same except it currently handles single module.

The only addition is that the module listens to messages `lowlevel_data_active_beacons_t` on channel `LL_DATA_ACTIVE_BEACONS` that carry information on infrared beacon IDs that should be scanned. The data polling done by function `sbs1_get_beacon_data()` is thus once for a while prepended by a call to function `sbs1_set_beacon_ids()` that tries to force the beacon scanner to ping specified beacons.

Gained data are then filled to the `lowlevel_data_beacons_t` message instance and published to channel `LL_DATA_BEACONS`.

Chapter 6

CAN-bus subsystem

To enable utilization of industrial-grade components in the development of the mobile robot Advee, at least one of the standard industrial communication field-busses had to be implemented. Probably the best interoperability among devices made by various manufacturers is nowadays guaranteed by the Controller Area Network (CAN) bus. Its standardization is handled by a wide consortium of manufacturers CAN in Automation (CiA) [4].

6.1 Bus description

The CAN standard accepts several physical layer standards that allow to choose the most propriate transfer medium for the most of the situations. The most frequently used is the ISO11898-2-conforming physical medium: it defines a two-wire differential bus with specific impedance of 120 Ω similar to the EIA-485 standard. On the condition of maximal bus data rate of 1 Mb/s the bus can be up to 40 m long. Commonly for all the physical layer standards, the Non Return to Zero (NRZ) bit encoding is utilized.

The other layer defined by the CAN bus ISO11898 standard is the data link layer that sits on the top of the physical layer. It is divided into two sublayers: the Medium Access Control (MAC) and the Logical Link Control (LLC). The former one provides the lower function related to frame marshalling, error detection or bit stuffing whereas the latter one handles message filtering, overload control and recovery management.

The communication is done using so called *frames*; four types of the frame are introduced: data frame, remote frame, error frame and overload frame. The protocol is not byte-oriented — the communication is divided into several groups of bits that do not have any standard length. The data frame used for the most of the operational

data exchange can hold up to 8 data data bytes — this restriction allows for an efficient hardware implementation and prevents bus contention.

6.2 CANopen higher-layer protocol

As the Controller Area Network defines only a relatively low-level part of the communication, a higher protocol is needed to provide device standardization means. Tens of such protocols can be found — with CANopen being one of the most successful and supported. It is being administered by the CiA organization as well.

CANopen protocol specifies a standard interface of the device that is managed by several *services*; the common functionality is mostly specified by the DS-301 standard. The lifecycle of each device is driven by a standard state machine controlled by the bus master using the *network management* (NMT) service. All configuration and run-time parameters are implemented as objects identified by a 16-bit ID and held in the *object dictionary*. Several index ranges are defined to accommodate entries of specific meanings.

Full access to the object dictionary is enabled using the service *data object protocol* (SDO) — a non-real time acknowledged protocol that allows device configuration. The operational access is done via transmit and receive *process data objects* (PDO) — a real-time data service. Additionally, *synchronization* (SYNC), *time stamp* (TIME) and *emergency* (EMCY) objects are defined.

The CiA also introduces about thirty device and application profiles. As both devices needed to be interfaced by the control system are motion-related, only the DSP-402 profile is covered. The specification provides a set of generic default PDOs available to all drives as well as set of specific default PDOs applicable only to a specific class of drives as servo drives, frequency inverters or stepper motors [4].

6.3 Kernel driver `lincan`

The TS-CAN1 PC/104 CAN bus interface card is supported by the *lincan* kernel driver, a part of the OCERA real-time framework [12]. The driver encompasses the whole communication with the TS-CAN1 card in the kernel space and exhibits a standard character device (usually `/dev/can0` for the first device) for utilization from the user space.

The *lincan* kernel driver ARM9 binary is supplied by the manufacturer of the TS-7800 board. Module loading by the Linux kernel has to enabled in the case that a different than supplied kernel is employed.

6.4 User-space library `libcanbus`

The user-space library `libcanbus` interfaces the kernel driver device, implements the DS-301 and DSP-402 standards and provides custom functionality both for the EPOS 70/10 controller and the IclA compact drive. Because the library will be a subject of refactoring and reimplementation soon, its description will be rather brief.

The implementation is divided by the standards: the base is the bare CAN bus functionality. The library uses the Virtual CAN API (VCA) [12] to wrap the access to the *lincan* device. The connection is managed using a custom structure type `canbus_connection_t` and handled by self-explanatory functions `canbus_init()`, `canbus_stop_reception()` and `canbus_close()`.

The participation on CAN-bus communication is started by calling `canbus_subscribe()` with filled `canbus_subscription_t` record. On the contrary to the EIS-485 support library, the communication is not direct to accommodate higher layers — so called providers are introduced to formalize the relation between layers of the library. The basic CAN bus implementation provides functions to send a message that wraps the `vca_send_msg_seq()` function. Finally, the function `canbus_msg2str()` that prints the formated message into a dynamically allocated character buffer is implemented.

6.4.1 DS-301 standard layer

The DS-301 CANopen functionality is implemented as a provider by defining the `canbus_provider_t` structure. The provider record is filled and passed to the `canbus_subscribe()` function in the `ds301_init()`. All messages received by the basic CAN bus layer are thus handed over to the DS-301 implementation. The layer further provides its own subscription mechanism that allows the CANopen nodes to use its functionality.

The present state of the library has a working support for NMT, SDO, PDO and EMCY services; the remaining SYNC and TIME protocols are going to be implemented in a near future. A set of helper function is provided to support conversion between standard data types and their serialized byte-array form. A number of standard DS-301-compliant communication objects is also supported — e.g. getting device status or value of the error register.

6.4.2 DSP-402 standard layer

The DSP-402 support code augments the support of communication objects to some more DSP-402-related objects — especially the *controlword* and *statusword* that are

essential for the DSP-402 operation.

Finally, the device drivers are implemented above this layer. These are really minimalistic as providing only access the functionality needed for purposes of the robot. The EPOS 70/10 driver consequently implements only a custom function `epos7010_set_target_velocity()` to set the target velocity using the default RPDO4. The IclA driver is richer — the driver is operated in a position mode and the position is by function `iclan065_set_target_position()` via the default RPDO2. Moreover, a more complex function `iclan065_perform_homing()` provides homing of the drive using a mechanical homing switch fitted in the robot.

6.5 Driver module `driver_canbus`

The driver module is functionally very similar to the EIA-485 driver module thoroughly described above. The main loop contains one additional operation and that is checking the LCM motion command message timeout — in case the motion command does not arrive for longer than one second, a *quick stop* command is issued to EPOS 70/10 controlling the propulsion.

The driver also at the time lacks the data-polling thread — no data are currently polled from the devices. Motion commands are received as `lowlevel_cmd_motion_t` messages on channel `LL_CMD_MOTION` and corresponding device driver function gets called directly in the LCM message handler function.

Chapter 7

Auxiliary subsystems

Besides the the main communication busses, which handle sensory data polling and motion subsystem governing, there are utilitary modules in the control system that manage resources not directly connected with the robot function — the power management and computer box cooling.

7.1 Power supply control

As already introduced and described in Chapter 4 Detailed hardware description, the power subsystem of the robot is managed by the **BR-PSC1** eight-channel power controller. Each channel can be enabled / disabled and the current flowing through the connected power module can be measured[1]. The power controller also implements a boot sequence and a power-off sequence — it guarantees that the master polling modules will be switched on after all slaves are up and running.

The power controller is communicated with using a dedicated point-to-point full-duplex EIA-485 link (sometimes denoted also as EIA-422). It provides a largely noise-proof communication channel that is not threated by jamming or contention by a malfunctional device — the power management is important enought to qualify for a dedicated line.

7.1.1 Support for BR-PSC1 in the `librs485`

The basic structure of the **BR-PSC1** device driver is that same as of the other devices. However, because of the full-duplex, point-to-point nature of the communication channel the device implements also assynchronous (non-polled) communication. Because such message can arrive virtually at any time, the driver has to provide a call-back function that gets call in the case of message reception. This

[1] On the condition that the module is equipped with the ACS712-family Hall current sensor.

is implemented using a `psc1_subscription_t` type member of the device structure `psc1_device_t` — the subscription holds a handler function pointer for each message type of the device.

The device-specific message handler function `psc1_handle()` (that is called from the reception automaton) then does not simply pass the message to the generic function `rs485_handle()`. It firstly checks whether the received message is one of the message that can come assynchronously — if positive, it further checks that the TXRX record is empty (this provents the handler function from "stealing" the data in case of a synchronous transaction). If these contitions are fulfilled, the message is considered to be assynchronous, is parsed and the call-back function gets executed.

To support the tranditional synchronous transaction, the driver implements functions `psc1_get_pwr_status()` to query the power status of individual channels and `psc1_set_pwr_status()` to force a channel to a needed state.

7.1.2 Driver module `driver_power`

The architecture of the **driver_power** module fully conforms to principles stated in Chapter 5 EIA-485 subsystem. The difference is that at the time it does not feature a data-polling thread, all communication is assynchronous. Secondly, a LCM handler listening to `lowlevel_cmd_power_t` messages on channel `LL_CMD_POWER` has been implemented — it branches its execution according to the `device` property of the message. If the target device is the Linux SBC itself, it does not communicate with the power controller at all: instead it just issues a *poweroff* or *reboot* command depending on the ordered action.

Finally, the the driver implements the `psc_paused_handler` function that is an interface to the assynchronous EIA-485 messages reporting the `PAUSE` signal engagement change. This event is just translated to a LCM message of the `lowlevel_event_stop_t` type and published to channel `LL_EVENT_STOP`.

7.2 Fan control

A reliable operation of the used computers is among others conditioned also by keeping their temperature at reasonable levels. The air circulation through the computer box is forced by two fans governed by the **BR-FAN1** module. As its System Management Bus (SMBus) interface is based on the Inter-Integrated Circuit (I^2C) bus standard[2], the `libi2c` library has been developped to simplify fan-control driver design.

[2]The SMBus is in fact more strict than the I^2C specification to allow detection of common device failures — e.g. it defines clock timeouts and slave address acknowledging.

7.2.1 User-space library `libi2c`

Encouraged by a manufacturer-supplied reference design of a bit-banged[3] SPI bus driver, it has been decided to implement the I²C master library in a similar way. The TS-7800 platform drivers exhibit the access to the digital input / output (DIO) pins using memory mapping: the input direction is mapped to offset `0x4` from the base address `0xE8000000` and output is offset by `0x8`. The output is assymetrical — the logic 1 is held only by a pull-up resistor whereas the logic 0 is achieved by pulling down the pin by a transistor switch. This enables easy direction change to intput by setting the propriate output register bit and reading the input register bit.

The library internally provides preprocessor macros to access the SDA and SCL pins — for SDA it enables both directions of access while the SCL support is currently limited to writing only.

Similarly to the EIA-485 and CAN-bus libraries, the `libi2c` provides functions `i2c_init()` to open the memory access and setup the port and `i2c_close()` to clean up used resources. Basic operation on the bus are supported: issuing a START condition (`i2c_start()`) and repeated START condition (`i2c_repstart()`), a STOP condition (`i2c_stop()`), writing a byte to a slave device (`i2c_write()`) and reading a byte from a slave device (`i2c_read`).

MAX551X driver

The general I²C functionality is employed by the MAX551X Dual-Channel Temperature Monitor and Fan-Speed Controller with Thermistor Inputs device driver. Again it follows conventions used in all developed libraries — the device is denoted by a custom struture type `max551x_device_t` holding its bus address and a pointer to the `i2c_connection_t` record. Its initialization is provided by `max551x_init()` and freeing by `max551x_free` functions.

The device driver further provides functions `max551x_get_reg()` and `max551x_set_reg()` that implement a minimal byte-oriented I²C / SMBus protocol.

The device functionality supported by the driver is not complete — only access to the used registers has been implemented. The temperature measurement channels 1 and 2 can be read using the functions `max551x_get_temp1()` and `max551x_get_temp2()` and corresponding over-temperature (OT) registers can be both read and written using the function `max551x_get_ot_limit1()` and others. Device configuration byte stored in register `0x02` is made accessible using functions `max551x_get_config()` and `max551x_set_config()`.

[3]Bit-banging means a method of emulating serial communication interfaces without use of dedicated hardware peripherals — the I/O pin states are set purely from the software.

Functions `max551x_get_config()` and `max551x_set_config()` utilize another custom data type defined by the driver: `max551x_config_t`. Its declaration utilizes a `union` of a bit-structure field and a `uint8_t` field; this approach allows an access to the whole configuration byte while providing an easy structural access to individual flags of the configuration:

```
typedef union
{
    struct
    {
        unsigned spinup_dis : 1;     //> Spin-up disable: 0 = enable; 1 = disable
        unsigned temp2_local : 1;    //> Temp Ch2 sources: 1 = local; 0 = remote2
        unsigned pwm_start_en : 1;   //> Min duty cycle: 0 = 0%; 1 = fan-start
            duty cycle
        unsigned pwm2_invert : 1;    //> Fan 2 PWM invert
        unsigned pwm1_invert : 1;    //> Fan 1 PWM invert
        unsigned timeout_dis : 1;    //> Timeout: 0 = enabled; 1 = disabled
        unsigned por : 1;            //> POR: 1 = reset
        unsigned standby : 1;        //> Standby: 0 = run; 1 = standby
    } bit;
    uint8_t word;
} max551x_config_t;
```

Similarly, the fan configuration read by `max551x_get_fan_config()` and written by `max551x_set_fan_config()` is defined as the `max551x_fan_config_t` union type.

7.2.2 Driver module `driver_aux`

The auxiliary devices driver module **driver_aux** is at time used only to handle the **BR-FAN1** fan controller communication. It features the architecture already known from the other driver modules, including a resources intialization phase, pseudo-endless LCM handling loop and a data-polling thread. The thread reads out temperature values, fills them into a `lowlevel_data_temperature_t` message and publishes it to channel `LL_DATA_TEMPERATURE`.

Chapter 8

Conclusion

The goal of the thesis was to design and implement a control system that would fit the needs of the autonomous mobile robot Advee. Both the hardware and software architecture should have been chosen on the basis of its mechanical construction and of the sensory / actuator equipment demands.

An industrial-grade ARM9 single-board computer TS-7800 has been selected to form the base of the control system. It disposes of almost all peripherals needed to interface the rest of the electric equipment; its PC/104 form factor allows to implement missing capabilities by using a standard expansion board. The electrical interface of the computer is adjusted to the rest of the robot by employing custom intermediary modules.

The extended host computer provides one CAN-bus interface, two half-duplex and one full-duplex EIA-485 ports and Dallas 1-Wire support. In addition, up to two Mini-ITX computers can be controlled and the computer box cooling is governed.

Efficient utilization of the hardware means and is guaranteed by using the manufacturer supplied custom GNU/Linux operating system. Several user-space libraries have been implemented on the top of the operating system to simplify and clarify the control system development. Detailed information on the software modularity principles is given.

Communication among the software modules of the system is handled by the simple yet powerful LCM library developed at the Massachusetts Institute of Technology. Its only flaw, the absence of a Microsoft .NET implementation, had been solved by creating the *lcm-dotnet* port that has been then officially adopted to the LCM project.

The thesis also describes all software equipment that is needed to handle the slave devices including the sensors connected by the EIA-485 busses and the actuators networked using the CAN bus. The power management and fan control is introduced as well.

The qualities of the system have been verified during over 500 hours of the robot operation in various environments. It has been found stable and reliable, still open to functionality modifications and extensions. The best proof of its stability is that the robot has been routinely operated by personnel with only a rough knowledge of the system functionality.

Fig. 8.1: Advee in a real environment (19th International Trade Fair AMPER 2011)

Bibliography

[1] Agner, S. *Linux Realtime-Fähigkeiten.* Luzern: Hochschule Luzern, Technik und Architektur, 2009. Bachelor's thesis.

[2] Ben Ari, M. *Principles of Concurrent and Distributed Programming.* Second edition. Addison-Wesley, 2006. ISBN 0-321-31283-X.

[3] *Intelligent Compact Drives IclA N065 Catalogue* [online]. Berger Lahr GmbH, 2006. [cited 18 May 2011]. Available from: <http://www.regulacni-pohony.cz/ftp/Katalog_IclA_N065_GB_10_2006.zip>.

[4] *CAN in Automation (CiA)* [online]. 2011 [cited 23 April 2011]. Available from: <http://www.can-cia.org/>.

[5] Gerkey, B. P., Vaughan, R. T. & Howard, A. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *Proceedings of the International Conference on Advanced Robotics (ICAR 2003).* Coimbra, Portugal, 2003. Pp. 317–323.

[6] *GLib Reference Manual* [online]. The GNOME Project. [cited 16 May 2011]. Available from: <http://developer.gnome.org/glib/stable/>.

[7] Hrbáček, J., Hrbáček, R. & Věchet, S. Modular Control System Architecture for a Mobile Robot. In Fuis, V. (ed.). *Proceedings of the 17th international conference Engineering Mechanics 2011.* Prague: Institute of Thermomechanics, Academy of Sciences of the Czech Republic, 2011. Pp. 211–214. ISBN 978-80-87012-33-8.

[8] Hrbáček, R. Simulation Based Neural Motion Planner Learning. In *Proceedings of the 17th Conference STUDENT EEICT 2011 Volume 1.* Brno, Czech Republic, 2011. Pp. 189–191. ISBN 978-80-214-4271-9.

[9] Hristu-Varsakelis, D. & Levine, W. S. (ed.). *Handbook of Networked and Embedded Control Systems.* Boston: Birkhäuser, 2005. ISBN 0-8176-3239-5.

[10] Huang, A. S., Olson, E. & Moore, D. *Lightweight Communications and Marshalling for Low Latency Interprocess Communication* [Technical Report MIT-CSAIL-TR-2009-041]. Cambridge, USA, 2009.

[11] Huang, A. S., Olson, E. & Moore, D. LCM: Lightweight Communications and Marshalling. In *Int. Conf. on Intelligent Robots and Systems (IROS)*. Taipei, Taiwan, Oct. 2010.

[12] Krakora, J., Pisa, P., Vacek, F. et al. *OCERA: Deliverable D7.4 Communication components* [online]. OCERA Consortium, February 2004. Available from: <http://www.ocera.org/archive/deliverables/ms4-month24/WP7/D7.4.pdf>.

[13] Krejsa, J. & Věchet, S. Mobile Robot Motion Planner via Neural Network. In Fuis, V. (ed.). *Proceedings of the 17th international conference Engineering Mechanics 2011*. Prague: Institute of Thermomechanics, Academy of Sciences of the Czech Republic, 2011. Pp. 327–330. ISBN 978-80-87012-33-8.

[14] *LCM: Lightweight Communications and Marshalling* [online]. 2011, Revised on 2010/11/08 [cited 18 May 2011]. Available from: <http://code.google.com/p/lcm/>.

[15] *The Linux Documentation Project* [online]. 2011, Revised on 2010/05/25 [cited 18 May 2011]. Available from: <http://mirrors.kernel.org/LDP/>.

[16] McKenney, P. *A realtime preemption overview* [online]. 2005, Revised on 2005/08/10 [cited 27 December 2010]. Available from: <http://lwn.net/Articles/146861/>.

[17] *Microsoft Robotics Developer Studio* [online]. [cited 11 May 2011]. Available from: <http://www.microsoft.com/robotics/>.

[18] Peringer, P. *SIMLIB/C++ – SIMulation LIBrary for C++* [online]. 2010, Revised on 2010/12/15 [cited 6 February 2011]. Available from: <http://www.fit.vutbr.cz/~peringer/SIMLIB/>.

[19] Poucha, P. *Přenos dat po linkách RS485 a RS422* [online]. 1999, Revised on 1999/07/25 [cited 20 March 2011]. Available from: <http://hw.cz/Teorie-a-praxe/Dokumentace/ART705-Prenos-dat-po-linkach-RS485-a-RS422.html>.

[20] Ripel, T. *Návrh a realizace konstrukce kolového mobilního robotu*. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2010. 61 pp. Diploma thesis.

[21] *RTAI - Official Website* [online]. 2010, Revised on 2010/02/16 [cited 23 April 2011]. Available from: <https://www.rtai.org/>.

[22] Simmons, R. & James, D. *Inter-Process Communication: A Reference Manual*. Carnegie Mellon University, School of Computer Science / Robotics Institute, August 2001.

[23] *TS-7800 Embedded Computer* [online]. Technologic Systems. [cited 12 April 2011]. Available from: <http://www.embeddedarm.com/products/board-detail.php?product=TS-7800>.

[24] *TS-CAN1 PC/104 CAN Bus Interface* [online]. Technologic Systems. [cited 12 April 2011]. Available from: <http://www.embeddedarm.com/products/board-detail.php?product=TS-CAN1>.

[25] *TS-7800 Manual* [online]. Technologic Systems, 2009. Revised on 2009/05 [cited 7 May 2011]. Available from: <http://www.embeddedarm.com/about/resource.php?item=393>.

[26] *SN75176A Differential Bus Transciever* [online]. Texas Instruments Inc., 1995. Revised on 1995/05 [cited 2 April 2011]. Available from: <http://focus.ti.com/lit/ds/symlink/sn75176a.pdf>.

[27] *UZIMEX: Špičkové technologie do automatizace a robotizace* [online]. [cited 27 April 2011]. Available from: <http://www.uzimex.cz/>.

[28] Vojáček, A. *Základní informace o RS-485 a RS-422 pro každého* [online]. 2007, Revised on 2007/07/14 [cited 22 May 2011]. Available from: <http://automatizace.hw.cz/zakladni-informace-o-rs-485-rs-422-pro-kazdeho>.

[29] *Xenomai: Real-Time Framework for Linux* [online]. 2011, Revised on 2011/03/08 [cited 23 April 2011]. Available from: <http://www.xenomai.org/index.php/Main_Page>.

[30] Yaghmour, K. *Adaptive Domain Environment for Operating Systems* [online]. [cited 27 December 2010]. Available from: <http://opersys.com/ftp/pub/Adeos/adeos.pdf>.

[31] Yodaiken, V. The RTLinux Manifesto. *Proceedings of the 5th Linux Expo* [online]. 1999 [cited 15 May 2011]. Available from: <http://www.yodaiken.com/papers/rtlmanifesto.pdf>.

List of symbols, physical constants and abbreviations

CISC Complete Instruction Set Computer

CLI Common Language Infrastructure

CLR Common Language Runtime

CRC Cyclic Redundancy Check

EKF Extended Kalman Filter

GCC GNU Compiler Collection

GUI Graphical User Interface

IDE Integrated Development Environment

IPC Inter-Process Communication

IRC Incremental Rotary enCoder

LED Light Emitting Diode

$LiFeYPO_4$ Lithium Yttrium Iron Phosphate

PCB Printed Circuit Board

RISC Reduced Instruction Set Computer

SoC System on Chip

SSR Solid State Relay

TCP Transmission Control Protocol

UDP User Datagram Protocol

List of appendices

A Instructions of current devices **68**

Appendix A

Instructions of current devices

This overview does not mention common intruction for getting the firmware version (`0xFE`) and ordering reset of the device (`0xFF`).

ID	Support function	Description
`0x01`	`sd1_get_srf_data`	Read SRF distances as uint16
`0x02`	`sd1_get_ir_data`	Read SHARP distances as uint8
`0x10`	`sd1_get_both_data`	Read SRF distances as uint16 and SHARP distances as uint8
`0x20`	`sd1_get_pause_src`	Read last PAUSE source
`0x21`	`sd1_clear_pause_src`	Clear last PAUSE source

Tab. A.1: Instruction set of BR-SD1

ID	Support function	Description
`0x01`	`so1_get_ticks`	Get IRC ticks in two to eight bytes
`0x02`		Reset IRC tick counters

Tab. A.2: Instruction set of BR-SO1

ID	Support function	Description
`0x01`	`sbs1_set_beacon_ids`	Send request to scan given beacons
`0x02`	`sbs1_get_beacon_data`	Get beacon measurements
`0x03`	`sbs1_get_debug_data`	Get debugging information

Tab. A.3: Instruction set of BR-SBS1

ID	Support function	Description
0x01	psc1_get_pwr_status	Get channel on/off power status
0x02	psc1_set_pwr_status	Set channel on/off power status

Tab. A.4: Instruction set of BR-PSC1