I. Declarations / Erklärungen

This thesis is not supposed to be available for (university internal) publicity. The author, *Micro-Key BV* as well as its customers prohibit the transfer of this paper to anyone who is not directly involved into this thesis or its assessment. This also includes the use of this paper as (university internal) example material. The use of all figures and references of hard- and software in this thesis is permitted by *Micro-Key BV* and/or its customers.

Diese Diplomarbeit ist nicht für die (hochschulinterne) Öffentlichkeit bestimmt. Der Autor, *Micro-Key BV* sowie ihre betroffenen Kunden untersagen jegliche Weitergabe dieser Arbeit an Personen oder Stellen, die nicht direkt mit der Durchführung und/oder Bewertung beauftragt sind. Dies untersagt auch die Nutzung dieser Arbeit als (hochschulinternes) Anschauungsmaterial. Die Verwendung von Abbildungen und Hard- und Software für diese und in dieser Arbeit ist durch *Micro-Key BV* und/oder ihre betroffenen Kunden genehmigt.

I declare that I have written this thesis on my own, only and solely with the help of the named/denoted resources.

Ich erkläre, die nachfolgende Diplomarbeit selbstständig und nur unter Zuhilfenahme der angegebenen Hilfsmittel und Quellen angefertigt zu haben.

Place, Date / Ort, Datum	Matthias Mitscherlich Diplomant / Diplomand

II. Preamble

This thesis was written between the 15.October 2008 and the 19. January 2009 in cooperation with the company *Micro-Key BV*. The title is “Hard- and software development of a test environment of a maritime stabilisation system”.

I would like to thank *Micro-Key BV* for the offer to graduate my studies with this thesis at their company, the special offer to extend this thesis to the hardware development and the great support during the whole time. Special thanks for support goes to *Thomas Lantinga* in the hardware development and *Frank van der Schoot* in the software development, implementation and integration. Furthermore to *Jos Pasop* for the support with the testee device and *Cor Peters* and *Peter Loef* for help with the embedded Linux operating system and the Linux scripting language.

Finally, I would like to thank *Prof. Dr. H. Ortleb* and *Prof. Dr. W. Pohl* for the support before and during this thesis.

III. Abstract / Zusammenfassung

This thesis deals with the development, the assembly and the application of a test environment for a custom hardware design. This specific test device is able to send appropriate stimuli to the test item and in return to receive and read outgoing signals. Special test programs afford the active search for wrong-placed components, short-circuits and open wires. Furthermore, the analogue in- and outputs will be calibrated.

The development of the tester concerns both the hardware design as well as the implementation of an independent operation software, started with the hardware configuration, the operating system integration, a communication protocol and the creation of the special test programs. In addition to that, software will also be created for the testee in order to react on incoming commands.

At Last, the operation of the complete environment is assembled. With this, all occurring problems will be analysed and solved. The full-finished vicinity requires an analysing and reporting software, which also is created in this part of the thesis.

Diese Diplomarbeit behandelt die Entwicklung, den Aufbau und die Verwendung einer Testumgebung für ein speziell angefertigtes Gerät. Die spezifisch auf dieses Gerät ausgerichtete Testhardware ist in der Lage, passende Stimuli an den Prüfling zu senden und umgekehrt abgehende Signale auszulesen, durch spezielle Testprogramme falsch plazierte Bauteile, Kurzschlüsse oder offene Leitungen auf dem Prüfling zu erkennen und die analogen Ein- und Ausgänge des Gerätes zu kalibrieren.

Die Entwicklung des Testgerätes betrifft sowohl das Hardwaredesign als auch die Realisierung eines eigenständigen Betriebes von der Hardwarekonfiguration über Betriebssystemintegration, Die Implementierung eines Kommunikationsprotokolls bis zu den spezifischen Testdurchläufen. Zusätzlich wird auch für den Prüfling Software entwickelt, die auf die entsprechenden Kommandos des Testers reagieren soll.

Zuletzt wird der Betrieb der Testumgebung hergestellt. Dabei werden vor allem die auftretenden Probleme analysiert und behoben. Zum vollständigen Testbetrieb gehört eine Auswertungssoftware, die ebenfalls in diesem Abschnitt erarbeitet wird.

IV. Table of content

V. List of figures

VI. List of tables

VII. List of code sequences

VIII. Abbreviations

Within this thesis, several abbreviations are used to increase the abridgement of the text. These Abbreviations mostly describe technical expressions, which are more or less common in the field of technical engineering. Table 1 shows all used abbreviations in alphabetical order.

Table 1: Comparison of abbreviations and their meaning

Abbreviation	Meaning
ADC	*Analogue-digital converter*
API	*Application programming interface*
ARM	*Advanced RISC machine*, formerly *Acorn RISC machine*
CAN	*Controller area network*
CE	*Chip enable*
CF	*Compact flash*
DAC	*Digital-analogue converter*
EEPROM	*Electrically erasable programmable read-only memory*
FAT	*File allocation table*
FreeRTOS	*Free real-time operating system*
GCC	*GNU compiler collection*
GNU	*GNU is not UNIX (recursive acronym)*
GPIO	*General purpose input output*
GPL	*General public license*
IDE	*Integrated development environment*
IIC, I²C or I2C	*Inter-integrated circuit*
ISP	*In-system programming*
JTAG	*Joint test action group*
MAC	*Media access control*
MCI	*Memory card interface*
MMC	*Multimedia card*
OpenOCD	*Open on-chip debugger*
OSI	*Open system interconnection*
PCB	*Printed circuit board*
PHY	*Physical layer*
PowerPC	*Performance optimization with enhanced RISC performance chip*
RAM	*Random access memory*
RISC	*Reduced instruction set computing*
(R)MII	*(Reduced) media independent interface*

Abbreviation	Meaning
RS	*Recommended standard*
RTC	*Realtime clock*
SD	*Secure disk*
SPI	*Serial peripheral interface*
SRAM	*Static random access memory*
UART	*Universal asynchronous receiver transmitter*
YAGARTO	*Yet another GNU ARM toolchain*
USB	*Universal serial bus*

2. Task

The main task of this thesis is to create an environment which is able to assure the quality of a mass-produced hardware. The test procedure is based on a multiple device vicinity in which one device is the test item coming from production and one other device is used as tester. The idea behind this test principle is to have a specific designed hardware that is able to individually test every wanted part of the testee. For the mass-production use, the vicinity is as far as possible automated.

The main focus of attention in this thesis is on the hard- and software development of the specific tester, although it will also be necessary to work on and with the test item as well. The whole thesis can be graduated in five sections, which are both, chronological and logical:

- General considerations about the mode of operation
  - Analysis of the test item
  - Creation of a test scheme based on technical and economical considerations
  - Choice of specific tester hardware (processor, specific circuits)
- Hardware development of the specific tester
  - Hardware schematic research and development
  - Hardware layout design
- Software development for the specific tester
  - Implementation of basic structures (operating system, drivers)
  - Acquiring of specific test applications
- Software development for the testee
  - Acquiring of scripts and software interacting with the tester test applications
- Consolidation of tester and testee
  - Appliance of test applications to complete test environment
  - Development of test assessment applications

3. Introduction

This thesis deals rather with an current problem for engineering companies than with scientific research. The development of a specific test device is, for a company, one of the most extreme methods of quality assurance, mainly because of its costs on the one hand for the device itself but on the other hand also for the engineer, who develops it. Quality assurance is of continuously increasing importance and gets still more important depending on the level, on which these devices interfere with the security of human life. Apart from that, quality is always a good sales argument, no matter how big and international the manufacturer may be. Especially small firms with a curt budget (that have to keep to the budget) are within a quandary; only products with a high amount of devices allow a non-uneconomical use of specific testers or hiring of test companies, but every risk taken with the quality assurance on smaller series increases the probability of claims by the customers and so rises costs and/or service work.

This paper shows the complete development of a test environment based on a specific designed test device. This specification will be achieved by analysing the test item both from an economical as well as a technical point of view. With the help of these results, a tester hardware will be developed, assembled and afterwards used in the complete environment with the units under test.

I have chosen this topic for my thesis, because it involves both hardware as well as software development parts and deals with an current problem with which I was faced before, during my practical semester. The creation and the design of a multilayer hardware was the more exciting part and forced me to a well defined and strict time management, because the production and assembly of the board took almost five weeks, which is nearly the half of the thesis time. Because of this, most of the software was created theoretically or was tested on alternative devices without any warranty that it will work like it was supposed to. Concerning the software, the most interesting part was the interaction of two totally different systems, one running on an embedded Linux operating system, the other running on a different processor and on an embedded operating system with a self-made command-line interpreter.

Nowadays it is common to develop hard- and software simultaneously. This approach is mainly caused by costs and simplifies time management. Furthermore, many parts of a software can be written and developed theoretically and are checked when the corresponding hardware becomes

available. On more complex software developments, demonstration boards or alternative devices are used to be able to prove the developed software.

This thesis was originally based on the simultaneous development, as mentioned above, due to the extreme time limits prescribed and assessed by the examination regulations for diploma students. Because of delivery-bottlenecks of the component distributors, it was not possible to assemble the specific tester device developed and designed within this thesis. In order to complete the task as far as possible, the consolidation part was moved to an already existing, alternative hardware based on the same processor and as far as possible similar to the tester board. As it will be shown in chapter XYZ SW_HARDWARE_CONFIGURATION on page ZYX, the software development was based on multiple devices and because of that equipped with software switches. So it is easily possible to switch the complete software to the alternative board. Nevertheless, the use of the alternative hardware is only a workaround caused by the delivery problems and does not allow to finish the development of the test environment.

4. Used software in this thesis

During this thesis, several programmes, packages and software are used. The frequently used and/ or special software will be shown and explained in this chapter.

4.1. Altium Designer package

The Altium Designer package is a powerful and for commercial and industrial purpose created software package. It contains special subprogrammes for every step of the hardware development and is furthermore able to be comfortably used in network vicinities by one or more designers. It is a commercial software and needs to be licensed by the user.

4.1.1. Altium schematic editor

The schematic editor allows the creation and development of electrical circuits. It can be used just to draw circuits (for instance for documentation) but is commonly used for creation of library bases for PCB[1] designs. The user is able to develop circuits from every available or self-composed component on multiple sheets. Furthermore, the components can be marked as driving sources, power supplies, inputs, outputs and the like, with which the drawn schematics can be simulated as an electric circuit. Moreover, a very detailed rule-management allows a complete schematic rule check for almost every possible error or carelessness like open pins, short-lines and much more. A special advantage of the schematic editor is the multi-sheet ability. It is possible to insert already drawn circuits from the actual or older projects as a component and so save development and drawing time. Additionally, the documentation gets smaller with this. Furthermore, the editor is able to annotate the component designators. This function allows to automatically order and rename the components, so that during the development process every sheet can contain a number one of every kind of component but afterwards the components are all named and numbered unique.

When the schematics are finished and free of errors, the complete schematic hierarchy can be compiled. With this, the drawn components and connections are split into a component list, a pin list and a netlist, which are then used by the *PCB* design editor.

1 PCB: Printed circuit board

4.1.2. Altium PCB editor

The *PCB* editor is for the development and design of (multilayer-) *PCBs* and is usually based on the lists composed by the schematic editor before. From this, normally the *PCB* is created by exporting all available circuits to a new *PCB*. After setting the properties of the board (e.g. number of layers, size), all used components from the schematic are added to the *PCB* design including their connections to each other.

The *PCB* editor is a powerful tool mainly based on extensive graphics. Nevertheless it is also usable on older PCs, although with lower performance. Its advantage lies in many additional graphical or organisational functions and routines, which allows keeping an overview over the design even on big multilayer design with less processor stressing. Similary, the graphic engine is completely adjustable, so that every layer and component can be set to every colour and almost every behaviour. The design editor is based on the same rule engine like the schematic editor, which allows to set different priorities and warning or error levels for certain rule violations. The *PCB* design rules mostly contain isolator spaces, distances between nets, widths of wires and the like. They are adjustable for general use as well as for certain nets or components.

The design is all time update-able. This means that if changes in the schematics are necessary after the design process already started and components or nets are added or deleted, the *PCB* designer can easily be updated from the new compiled schematic component- and netlists. Even changes in the inverse directions are possible. Figure 1 shows the *PCB* design editor window with focus on the central processor of the tester board, where red wires are on the bottom and blue wires are on the top signal layer.

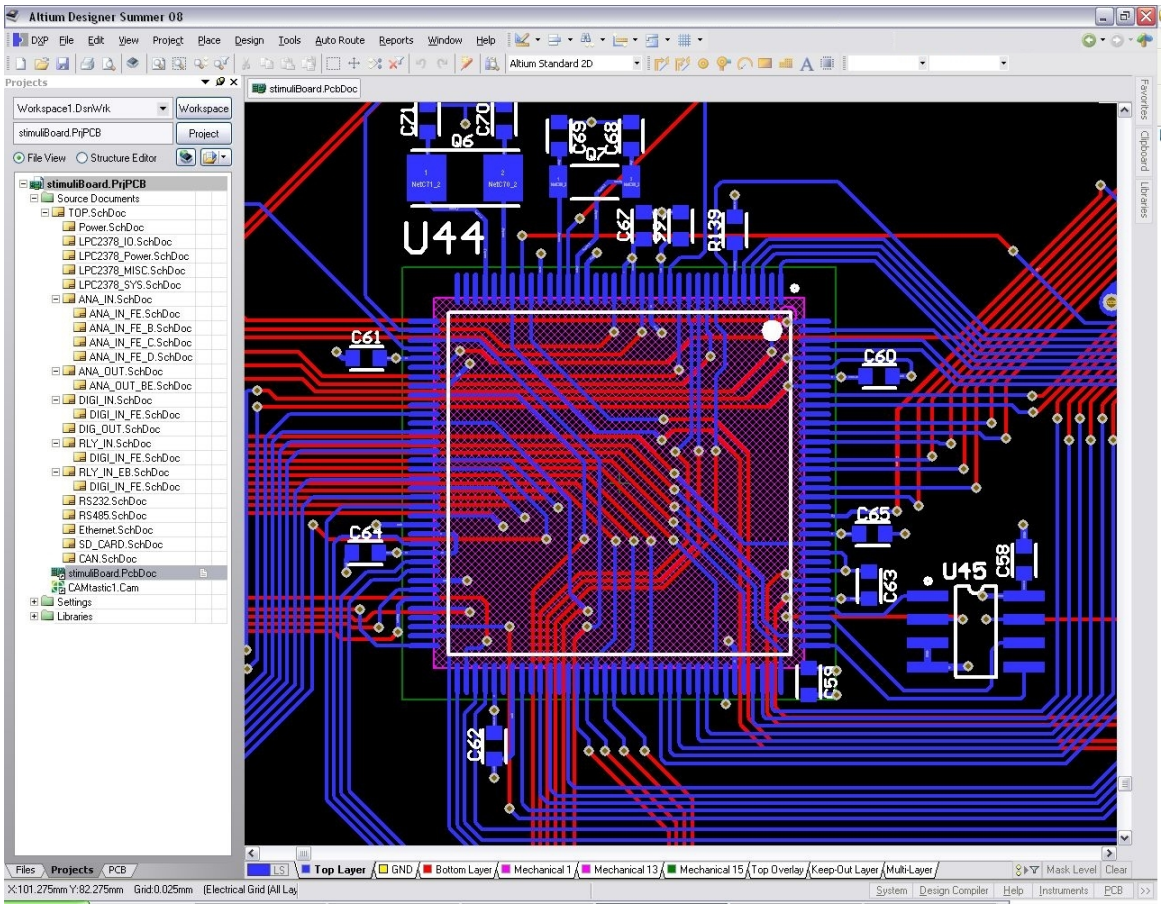


Figure 1: Altium PCB Designer with focus on the processor routing on all layers

4.1.3. Altium component editor

Although the *Altium* package already contains several thousand common components like resistors, capacitors, standard ICs, processors, diodes, *LEDs* connectors et cetera, it will be necessary to create or update components within the design process. In order to do this, *Altium* created an own component editor, which allows to create schematic and layout symbols and combine them.

A schematic symbol is an abstract drawn component which contains every used pin. Furthermore it is supposed to contain important internal connections and a logical structure for the pin order (for instance power pins above, input pins left, output pins right, *GND* pins below). The symbol is combined with a library containing additional information, like the manufacturer name, order codes, datasheets and more. Lastly it is possible to attach one or more layout symbols to it. These so-called footprints are exact mechanical and graphical descriptions of the component and displays every mechanical existence on every used layer. Mostly this concerns the pins of a component on the top-

layer, component names and drawn shapes on the overlay (or text) layer and on through hole components also the hole with its corresponding drill. Most footprints are normed to certain sizes and already included in *Altium Desinger*, so that in most cases just a new schematic symbol needs to be created and then combined with an existing footprint.

Figures 2 and 3 show examples of schematic and *PCB* symbols, both created with the component editor. They both stand for the same component, which is a *LM2678* step-down voltage regulator. In figure 2 the logical pin order mentioned above is visible. The footprint in figure 3 complies with the *TO-263* footprint standard.

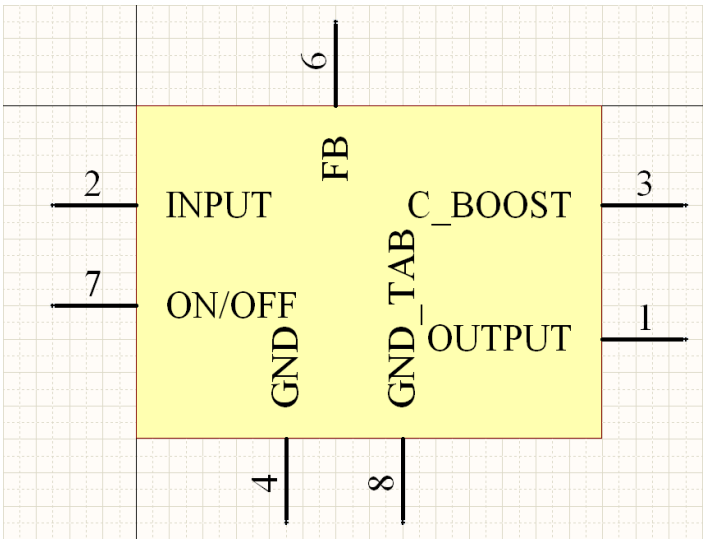


Figure 2: Schematic symbol example

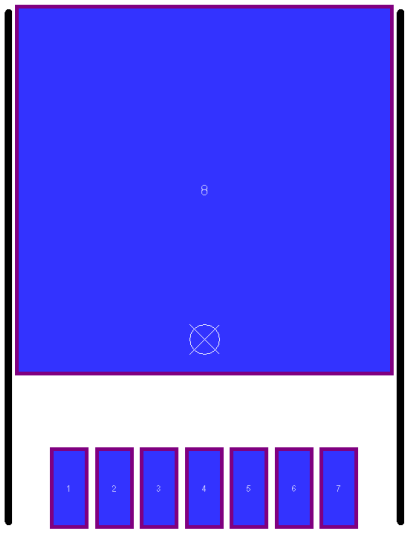


Figure 3: PCB symbol example (TO-263)

4.2. YAGARTO GNU-ARM Eclipse toolchain

The YAGARTO[2] is an ARM[3]-based cross-compiler toolchain based on open source projects, which all are freely available on the internet. The complete package is based on the GNU[4]-license, which allows to use and change the corresponding software both in private and commercial environments without any payments or restrictions. The rights of the original remain at the author in every case.

YAGARTO is composed of a number of individual programmes. The package installer already

2 YAGARTO: Yet another *GNU ARM* toolchain
3 ARM: Advanced RISC machine
4 GNU: *GNU* is not UNIX (recursive acronym)

combines these programmes to a complete toolchain environment in which manual settings are hardly required. The IDE[5] is optimised and set to the included compiler. No further software is required to develop and build executable software for *ARM7*-based processors.

The YAGARTO toolchain contains:

- the IDE Eclipse (open source, special license of the Eclipse foundation)
- the GNU-licenced GCC[6]
- the OpenOCD[7]

4.2.1. Eclipse IDE

Eclipse is an *IDE* based on *JAVA* and originally just supposed for *JAVA* development. Within the last years, Eclipse became the number one open source *IDE* because of its quality to be easily extended with plugins. In this way it is for example now possible to use Eclipse also for development of software in *C* and *C++*. This huge popularity could be ascribed to Eclipses possibilities of handling big projects, work with more then one opened project once and combine a clear, structured surface with helpful additional functions like auto-complete or a mouseover display. Moreover, Eclipse is unlike many other *IDEs* able to be set up in almost every way the user wants it to be. For example the text editor can be configured in many different ways of code style, just like the user or the corresponding company wants it to be. Figure 4 shows the surface of the Eclipse *IDE*.

5 IDE: Integrated development environment
6 GCC: GNU compiler collection
7 OpenOCD: Open on chip debugger

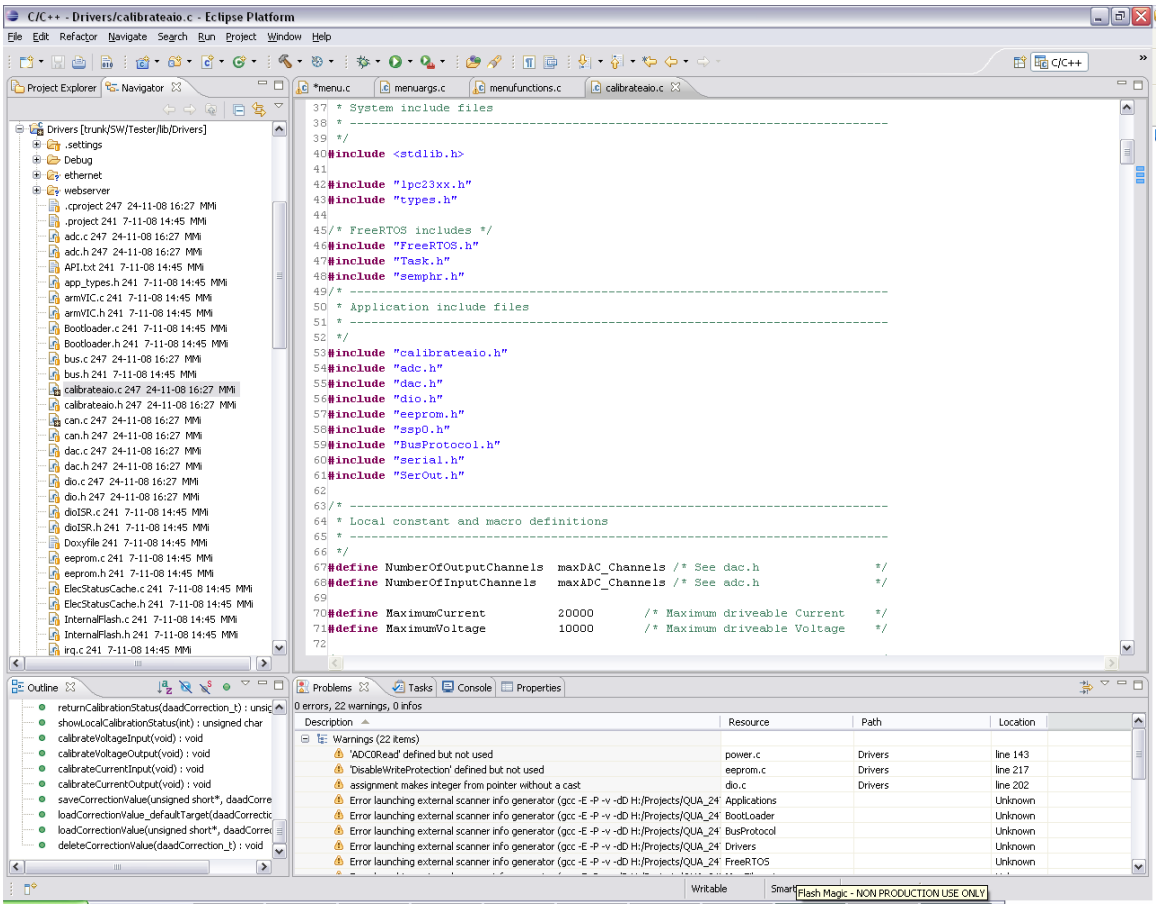


Figure 4: Eclipse IDE with opened projects

4.2.2. Open OCD

The Open OCD is a small tool that allows a comfortable debug communication with the processor via the JTAG[8] interface. Required for this is a supported *JTAG* controller. The tool creates an environment that enables a connection to the board via a telnet connection and provides a UNIX-like API[9]. With this vicinity, it is for instance possible to read or manipulate every available register or memory address. For every action, the processor must be halted. With this method, even one-time readable registers, which lose their content after reading in normal processing mode, are readable multiple times.

4.3. Visual Studio C# Express 2008

The *Visual Studio* package is a professional development environment created by the company

8 JTAG: Joint Test Action Group
9 API: Application programming interface

Microsoft (MS). The full Studio package contains *IDEs* for the languages *C*, *C++*, *C#*, *Virtual Basic* and *J++* and provides own compiler for each of them. The Visual Studio series is designed for software development for x86 systems, so usually personal computers, and is hardly applied to embedded systems.

Microsoft provides two different versions of *Visual Studio*; either the complete package, which must be licensed, or single, limited versions of each *IDE* for free use, which are marked with the additional name *"Express"*. Although the used version of *Visual Studio* is limited to more basic interaction with the operating system, it is sufficient for the purpose of this thesis and does not require a new license on *Visual Studio*. The environment is used to create the window test application control. Figure 5 shows the surface of the visual Studio *IDE*.

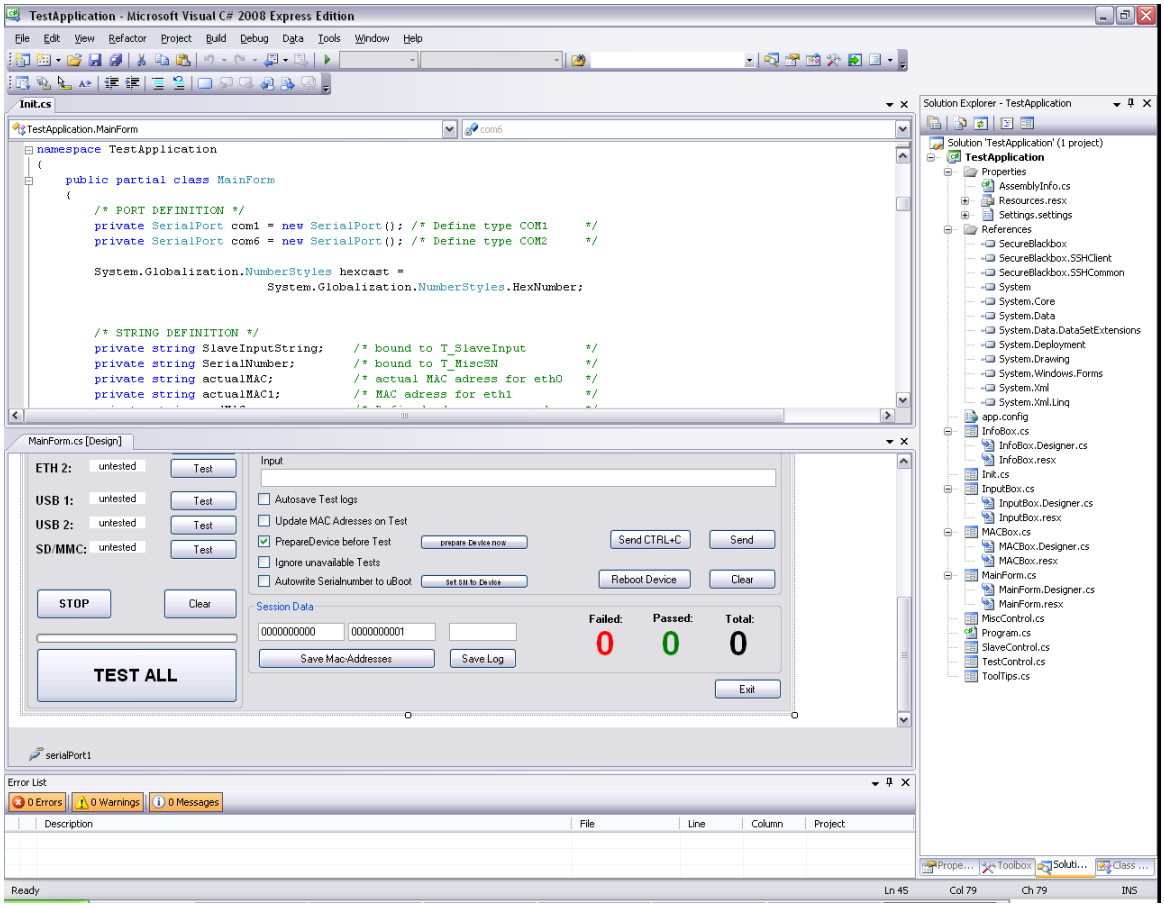


Figure 5: Visual Studio C# Express 2008 IDE surface with opened project

4.4. Notepad++

Notepad++ is an easy-to-use open source text editor, published with the GPL[10]. Its advantage is the automated keyword- and text highlighting for different programming languages and a teachable behaviour. Some functions like auto-completion or mouseover-display are similar to the ones already mentioned with the *Eclipse IDE* (see chapter *4.2.1. Eclipse IDE* on page *8*), which makes it easier to switch between them or work with both programmes at the same time. In this thesis, *Notepad++* is mostly used to develop the linux applications for the testee, which can either be scripts or small *C* programmes. A server-based version of the *GCC* is used to compile the *C* code written in *Notepad++*.

4.5. Putty

Putty is an open source terminal, *ssh*- and *telnet* client for *Windows* operating systems. It is used to establish almost every connection between the PC and the used boards, normally on the serial ports. The connection to the *Open OCD* is also established with *Putty*. It is usable multiple, which means that several instances of this application can be opened and used at the same time. Although the *Windows* operating systems contain an own terminal client called *Hyperterm*, *Putty* is preferred due to its smaller size and its better adjustability. The practical use showed also that *Hyperterm* is less reliable.

10 GPL: General Public License

5. Test item

The testee is the device that should be tested. As mentioned in the task, it is created and designed to be used as a ship stabilisation system. The principle of this stabilisation is rather new and not well known to the general public. It in fact consists an artificial horizon device, weather and atmospheric sensors shared all over the ship and several fins and heavy weights built to the ship hull. With this the controlling device is able to compensate wave as well as wind influences and to balance the ship to a cross-movement of just 0,7° to each side. Before this was more than 10°. This balancing is done by moving the weights near to or away from the hull. Moreover, the system is able to compensate movements ahead or sidewards by moving and rolling the attached fins. The weights as well as the fins weigh several tonnes per part, so they are moved by strong engines, which are controlled but not directly powered by the testee device.

The main purpose of the system is the stabilisation of luxurious offshore yachts. Because of the high-water resistance of the weights, they can only be used when the ship stands still or moves very slow. The fins are designed to be used also while sailing and to reduce the rolling of the hull on heavy sea. The main reason to implement a system like this to an offshore yacht was to solve the problem of safely landing a helicopter on a strong and incalculable moving and rolling ship. Also, yachts from that size often are too big to directly enter a port, which forces the owner to anchor outside in the unguarded sea. In this situation the stabilisation system increases the comfort on the ship extensively. The figures 6 and 7 show the attached weights (figure 6) and fins (figure 7) in action. The pictures are from demonstration and public relation video animations created by the customer company.

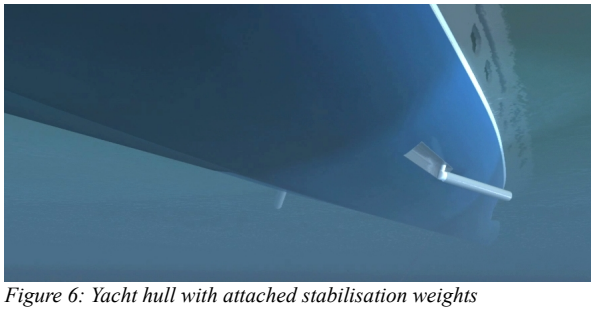


Figure 6: Yacht hull with attached stabilisation weights

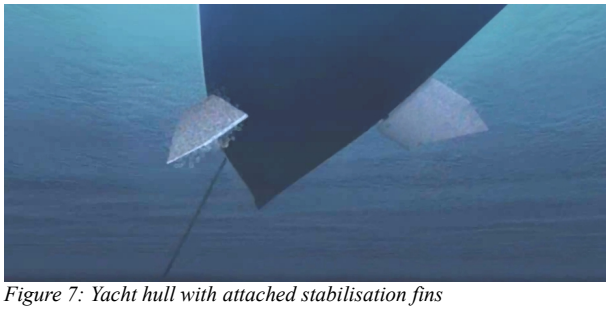


Figure 7: Yacht hull with attached stabilisation fins

Apart from the main purpose of the device, there are several more functions it fulfils. Because luxurious yachts are equipped with a ship-internal network and attached to that a high-level multi-

media system, the stabilisation system board can be integrated to this network up to a certain level, where the stabilisation of the ship is in every situation highest prioritised.

The complete controlling software is developed by the customer. The device is produced and afterwards delivered with a basic linux operating system that supports all available periphery. The test application environment developed within this thesis is placed in the production chain between the production and the delivery of the devices. Because the controlling software is not available, the purpose of the device is of minor importance for the further hardware design test.

The stabilisation system device consists of three different boards. This is mainly caused by the requirements of the customer for a better adjustment of the product to the future purpose; there is no technical need for division. Figure 8 shows the complete device with all three parts.

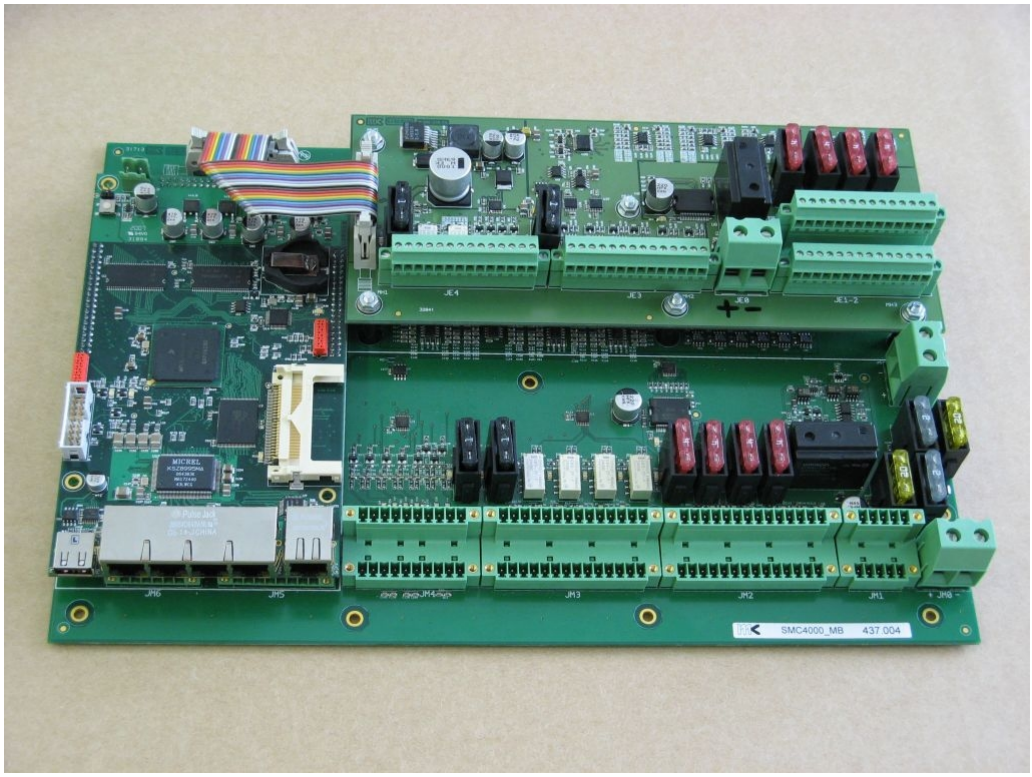


Figure 8: Overview to the complete testee device with all extensions

5.1. Processor board

The processor board is the only “intelligent” part of the device. It contains every component directly required by the processor. This includes memory devices as well as interfaces.

As main processor, a *Freescale MPC 5200* based on the PowerPC[11] architecture is used. *Freescale* (formerly *Motorola*) is, along with of *Apple* and *IBM*, one of the developers of this architecture and is nowadays one of the leading manufacturers for *PowerPC* chips. These are often used in embedded systems because of their high performance and the ability to calculate with floating points in double correctness. This is interesting for measurement devices like the actual system. The *PowerPC* processors nowadays available are normally based on 64-bit architecture, only *IBM* supports some 32-bit chips. In addition to the embedded purpose, *PowerPCs* are used in almost all actual gaming consoles (Nintendo Wii, Microsoft Xbox), in all *Apple* computers produced before the year 2006 and in high performance server stations and supercomputer grids (mostly from *IBM*).

Additionally to of the main processor, a co-processor is used to control the inputs and outputs of the device. Although this causes a discharge on the *PowerPC*, the main reason for the second processor is the real-time behaviour. The linux operating system on the main processor is a non-real-time system, which means that the software can not warrant the exact time of input values. In contrast to that, the co-processor only contains an cyclic, interrupt-based software that warrants the compliance of requested timings. The communication between the processors is mainly implemented with SPI[12], where the main processor represents the master device. Inputs as well as outputs are driven in the same cycle. The applied co-processor is a *NXP LPC 2108*, based on the *ARM* chip architecture. These chips became famous in the embedded and mobile sector within the last years because of their power saving features and the consequent 32-bit design. According to general estimates, more than 75% of the used embedded 32-bit *RISC* processors are based on an *ARM* core. Famous example devices based on *ARM* processors are the *iPhone* and the *iPod* (both developed and traded by *Apple*) and the mobile gaming console *DS* (developed by *Nintendo*).

Except for the two processors, the most important components are the memory devices. The board possesses 128 MB SRAM[13] and 64 MB FLASH, where the FLASH is used to store the original software code, which is copied to and afterwards executed from the *SRAM* on a boot-up. Furthermore, the processor board consists of a eight-port ethernet switch, a ethernet uplink port, an *USB* host interface connector and a *CF*[14] card reader. Finally, the processor board contains several connector interfaces for different purposes; for instance a *JTAG* connection for debugging, RS 232[15] connections to programme and communicate with the processors and three connector rows at the

11 PowerPC: Performance optimization with enhanced RISC performance chip
12 SPI: Serial peripheral interface
13 SRAM: Static random access memory
14 CF: Compact Flash
15 RS 232: Recommended standard 232

bottom side of the board to establish connections to the peripheral devices on the other boards.

5.2. Main board

The main board is the biggest part of the stabilisation system device and is used to mount the system to a rack and the processor as well as the extension board to itself. Furthermore, it contains the basic input and output connections available on the system and provides the complete power supply of the board including the step-down stages and fuses.

Most parts of the connector periphery consists of the analogue and digital inputs and outputs, out of which each eight channels are available. Only the analogue inputs contain six channels. The digital connections work with industrial standard logic of 24 V, the analogue connections are only used to drive and read current between 0 and 20 mA. In addition to that, four relay throughputs are implemented to the main board, which allow to switch an incoming signal directly to an outgoing one. Finally a CAN[16] and two *RS 485* ports complete the connection section. Because the additional extension board can be mounted to the main board as well as the processor board, there is one more signal connector available to attach the extension board to the system.

The main board contains an 64 kB EEPROM[17], which is to store the analogue calibration values. This is mainly caused by the consideration of being able to use the board in different vicinities with different processor boards without the need to recalibrate the processor to the new analogue connections. Figure 9 provides an overview of the whole connectivity. As shown, apart from the *CAN* and the *RS 485*, the different kinds of connections are not grouped to own connectors but are adjusted to the future purpose. This assembly is arranged by the customer. Also shown is the *EEPROM* in the upper left corner.

16 CAN: Controller area network
17 EEPROM: Electrically erasable programmable read-only memory (also E²PROM or E2PROM)

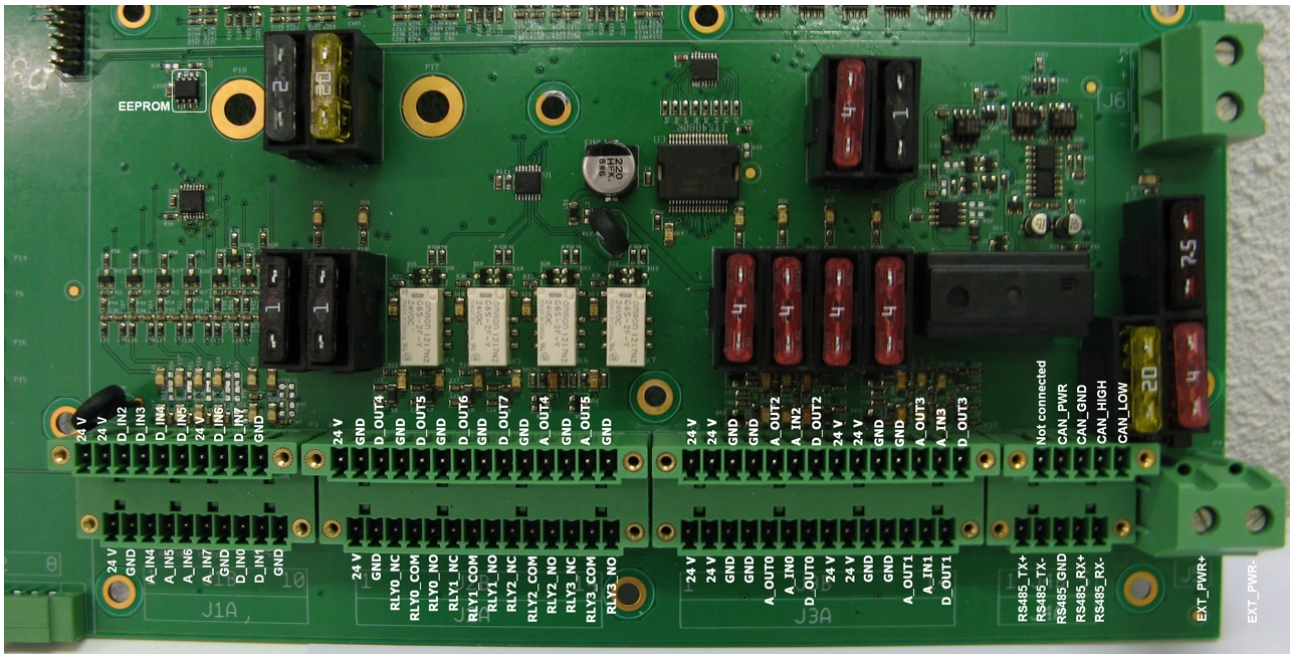


Figure 9: Overview of the mainboard connectivity

5.3. Extension board

The extension board is an additional component of the stabilisation system to extend it with more analogue and digital connections and relay throughputs. In fact, the extension board contains eight more analogue inputs, six more analogue outputs and both, four more digital in- and outputs. Furthermore, two more relay throughputs are available with this extension.

All in-, out- and throughputs are attached directly connected to the co-processor on the processor board through a connector placed on the main board. The power supply is done by a special power connector from the main board.

As well as the main board, the extension board contains an own 64 kB *EEPROM* to store the calibration values of the analogue connections. Figure 10 provides an overview of the connectivity. As well as already seen on the main board, the different kinds of connections are not grouped to own connectors but are adjusted to the future purpose. Also shown is the *EEPROM* and the connector to attach the extension board to the CPU board.

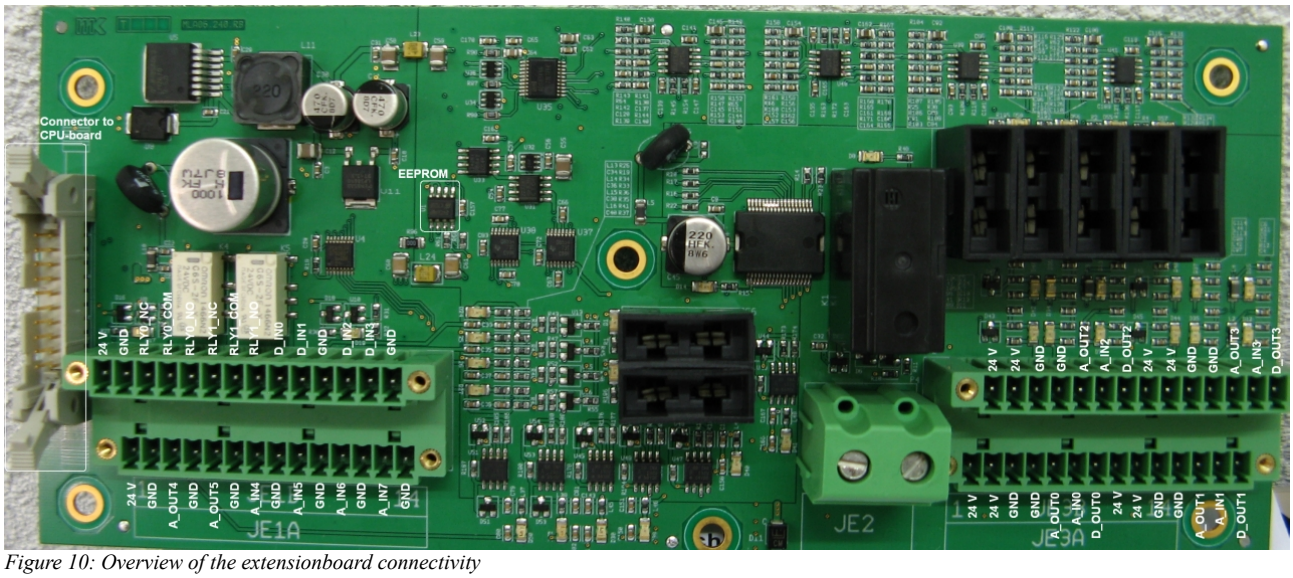


Figure 10: Overview of the extensionboard connectivity

6. Considerations

The considerations are the first step in developing of test environments. In general, they consist of a list of components, interfaces, connections and software parts that should be checked on the test item and corresponding to that a list of possible methods to apply these tests. The test item was introduced in chapter TESTEE on page XYZ. From this and other influences like already existing resources among others, one or several options to design a specific test hardware are to be created. Within these considerations, not only technical but also economical issues will be mentioned. Finally, the usability of the hardware must be considered, for instance the possible usage of operating systems or development and debugging software. With this information, the hardware assortment and afterwards the hardware design can be done.

6.1. Test scheme

The test scheme shows the individual parts of the testee that should be checked and in addition to that possible test methods for each part. Furthermore, a general, logical test order sequence must be acquired.

The testing of the several single-channel connections is one of the biggest parts of the whole test procedure. This section is split into the three different kinds of connections, which are digital, analogue and relay channels. For all, the test procedure is nearly similar; each channel must be proved to be able to drive the full signal range and not to be shortened to other channels. On the digital and relay channels, the full range consists of the two easy distinguishable BOOLEAN stats *LOW* and *HIGH* (digital channels) and *OFF* and *ON* (relay channels) respectively. The range of the analogue channels is prescribed by the converter-circuit, but except from the simple functionality, offset and deviation of the converter and converter-circuit must be considered, too. Non-calibrated converters should be tested with wide tolerances in order to simply check their functionality. In order to test these single-channel connections, every kind of connection will get an opposite on the tester hardware in the same quantity. Because the testee is expandable to more in- and outputs and the test environment should remain highly automated, the tester must contain also additional connections. This could either be done by adding more analogue, digital and relay circuits to the tester board or by multiplexing the already existing channels between different connectors. In order to reduce costs, both, on components as well as on the size of the board, the multiplexing method is advisable, also

because the channels will not be used all at the same time.

To test the *USB* and *CF* host controller and interfaces, it is sufficient to attach an appropriate memory device and mount it in the Linux operating systems. This is caused by the fact, that the design test should only assure that all connections between an interface and its controller are routed correctly. So, a special, so-called “golden file” can be copied to the memory and afterwards be read back and compared with the original. If the comparison failed, it is difficult to suggest a possible reason, because both interfaces are more complex constructions. The easiest workaround is to check all connections manually.

The two *EEPROM* components located at the main as well as on the extension board are connected to the processor by the *I²C* bus interface. Several, individual test options are possible. In general, a memory attachment is tested by a systematic check of all address and data lines. Because of the special *I²C* link-up, this check is not required. To test the connection and functionality of the memory, it is sufficient to write and afterwards read back a test string. Depending on the size, the connection speed and the future purpose of the *EEPROM*, it is recommended to systematically write to and read from the complete memory to find defect bits. If it is known that just certain areas of the memory will be used it is sufficient to just scan this sector for broken bits. Because of the size of 64 kB, a complete memory test will be applied. If both memory devices are not functional, it is obvious that the *I²C* interface is broken.

The *CAN* interface is easily testable either, with an self-test mode or with an so-called “Hello world” test. The self-test does not require a second device that sends or listens to the bus which in turn only assures that the devices transmit and receive connections work properly. One of the advantages of the *CAN* interface is to register it to certain addresses that are sent with every message. With this it is possible to group certain devices. With the “Hello world” test, the testee *CAN* is registered to a user-defined address and programmed to directly send back the incoming message. Afterwards, another *CAN* device sends a test string to this address. If the received string is the same as the sent one, the interface works. The test name is based on the fact that test strings in the information technology are commonly and traditionally “Hello world”.

The *RS 485* interfaces can be tested with two different methods. On the one hand, it is possible to connect them to each other and send test strings from one to another. This method is most economic but, in case of an error, does not allow any suggestion, which interface is broken. The second option is to attach two *RS 485* interfaces also to the tester board and connect them to the test items. Be-

cause of the low costs of *RS 485* transceivers, the higher comfort of two own *RS 485* interfaces outweighs the economic advantages.

The remaining connections like the *JTAG* or the *RS 232* interface can be tested by several different methods. The *JTAG* and one of the eight ethernet switch ports are necessary to copy the software to the board. The only method to test their functionality before any code is copied is to measure the wires manually. Programming the board is the most complex part for debugging because it is almost impossible to suggest any reason for a certain functionality disorder. From this one emanates that the connections will be fine if the copying is successful. Nevertheless it is possible but not necessary to implement a special test routine for the *JTAG* interface. The *RS 232* of the main processor is used as standard input and standard output of the Linux terminal. Although more complex tests with different transfer speeds can be created, it is adequate to test if the terminal of the board is accessible with the standard serial port settings already implemented in the Linux operating system, which are set to the maximum transfer rate.

The ethernet interface can be tested in two different ways, which depend on the surrounding. If available, the testee should be connected to an already installed network with its uplink port. So that it is possible to execute a ping command to an known server, which, if available, answers with a pong. The readback of this action can be evaluated. Although the functionality of the switch components is already assured by the software copying, it is recommended to test all available ethernet ports. In order to to do so, the switch can also be connected to the network, but it is important to only connect one port simultaneously, because it is otherwise not possible to get to know which port the device pinged. A more advisable method is an ethernet connection from the tester to the testee switch. Because of the uplink port connection to the network, the tester is able to reach the network through the switch. In order to automate this, it is possible to insert an 8-way multiplexer into this connection to test all eight switch ports one after another. Although this method is rather comfortable, it causes high costs, because of the high signal frequencies of the ethernet connection that must be multiplexed. To reduce the costs, the single line connection method is chosen and moreover, the software will force the user to switch the cable between the eight ports manually.

6.2. Tester hardware

the tester hardware is chosen by considerations of many different influences. As main processor and base of the test hardware, an 32-bit *ARM7* microcontroller of type *LPC 2388*, manufactured by

NXP, is chosen. This decision is mainly based on already existing resources and experiences within the company that supports this thesis, but also on the controller specifications of 64 kB RAM[18], more than 80 *GPIOs* and the applied interfaces, which are the advantages of this processor.

6.3. Test procedures

Table 2 shows all interfaces, connections, components and software parts, which should be tested, with their corresponding test method, controlled by the specific test hardware.

Table 2: Comparison of test items and their corresponding test methods

	Item to be tested	Test method
Processor board		
	Compact Flash reader	• Mount an attached memory device • Test write and read on memory
	Ethernet switch	• Ping to a known server through the uplink port for each port individually
	Ethernet uplink	• Ping to a known server
	SRAM	
	USB host controller	• Mount an attached memory device • Test write and read on memory
Main board		
	Analogue input	Stimuli read by tester board • Read single channel low value • Read single channel high value • Check if all other channels remain low
	Analogue output	Stimuli written by tester board • Read single channel low value • Read single channel high value • Check if all other channels remain low
	CAN interface	• Receive and return a test string from and to tester board
	Digital input	Stimuli written by tester board • Read single channel low value • Read single channel high value • Check if all other channels remain low
	Digital output	Stimuli read by tester board • Write single channel low value • Write single channel high value • Check if all other channels remain low
	EEPROM	• Test write and read on all memory section
	Relay	• Switch incoming voltage; readback on tester

18 RAM: Random access memory

	RS 485 interface	• Receive and return a test string from and to test board
Extension board		
	Analogue inputs	Stimuli read by tester board • Read single channel low value • Read single channel high value • Check if all other channels remain low
	Analogue outputs	Stimuli written by tester board • Read single channel low value • Read single channel high value • Check if all other channels remain low
	Digital inputs	Stimuli written by tester board • Read single channel low value • Read single channel high value • Check if all other channels remain low
	Digital outputs	Stimuli read by tester board • Write single channel low value • Write single channel high value • Check if all other channels remain low
	Relay	• Receive and return a test string from and to tester board

7. Hardware development

7.1. Schematic development

The hardware development usually begins with the design of the schematics. They contain every electrical component and connection which should be prospectively available on the *PCB*. On projects with a higher amount of electrical devices and wirings, the schematic drawings are among others usually divided to contiguous circuits like the boards power supply, the input- and output section of the main processor et cetera. Furthermore, multiple-used circuits or circuit parts should be split to an own schematic to prevent of repeated drawing.

In order to achieve a better abridgement, the schematics for this project should also be divided as mentioned above.

7.1.1. General information for schematic designs

The schematic design represents the actual research part of the hardware development. Because of that, this activity usually takes a higher amount of time than the *PCB* design afterwards. Depending on the future purpose of the device, the designer is not only responsible for the pure electrical circuits but also for the compliance of (international) norms and/or the costs of the device. These topics are normally more important on mass-produced and public-used products and force the designer to search and compare several and partial a lot of different devices to make the circuit fit as much as possible to the wanted target. But although the costs or norms are minor motivations on a prototype or single device development, the choice of the components takes some time. In addition to that, the use of non-common devices or ICs mostly forces the designer to create new component symbols in the design environment. Although only a few different symbols are used in the schematics, these components may differ on the further *PCB* in size (e.g. common resistors and power resistors, which both got the same schematic symbol but need size-different *PCB* symbols). On *SMD* devices, these symbols need to be very accurate. In a commercial facility, it is usual to supplement the components with the manufacturer name, the manufacturer code and order codes of the common used device distributors. Sometimes alternatives are also necessary.

These tasks rather concerning organisation parts are mostly important in commercial environ-

ments, where usually more then one designer designs several *PCBs* and is commonly called “library care”. The more accurate these libraries are created and supported, the easier is the future use of it for all designers, who all then can access new libraries and/or components.

7.1.2. Introducing information

The schematic drawings will contain several components and circuits with different purpose, so the division of the schematic to more sheets is advisable. The following chapters will to explain the created circuits and their purpose.

The schematics are organised in a hierarchical way with three layers. The lowest of these are circuits, which are used several times and are included to other circuits. The mid layer contains all one-time used circuits which are brought together on the highest layer, which is called *top sheet*.

7.1.3. Designators, values and synonyms in schematics

Every component in the schematics must be explicit assignable and as far as possible specifiable. For this, normally three different specifiers are used, either in the schematics but also later on in the *PCB*. These specifiers are: designator, value and synonym. Figure 11 shows the usage of these specifiers, where the letter “C” is the synonym, “22uF_16V” is the value, and the number “C92” is the designator, which is in fact a combination of the synonym and an individual component number. Where this number can exist several times on different components, its combination with a synonym is unique.

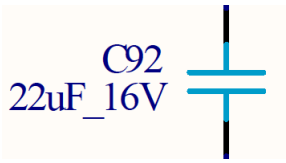


Figure 11: Example of the usage of the different specifiers

The synonyms are used to show the kind of the component on first sight. They are not normed but normally used akin. In table 3, the synonyms used in this thesis with their component meanings are explained.

Table 3: Overview of used synonyms and their component-meanings

Synonym	meaning
C	Unipolar capacitor, Electrolytic-capacitor
D	Diode, Schottky-diode, LED
F	Fuse
L	Inductor, Choke
P	Connector, Pin-grid, Header
Q	Transistors, Crystal
R	Resistor, Power-resistor
S	Switch
U	ICs, processors, any kind of integrated devices

7.1.4. Power supply section

The whole board will only be supplied with a constant power source of 24 Volts. Nowadays, this is the most common power supply available in developmental and industrial vicinities.

Although there are some circuits that are designed to work on 24 Volts supply, most of the circuits and components are to be supplied with a much lower voltage. It is advisable to chose the components that way that they fit as far as possible on the same voltage. In the actual schematic, seven different voltages are needed, which are mentioned in table 4 with their schematic names, values and main purposes.

Table 4: Table of used voltages with schematic-used names, values and general purpose

Name	Value	Purpose
VIN	24 V	• Unfiltered voltage from the external supply • Supply of VIN_DIGI
V24	24 V	• Switching voltage of multiplexers • Further transformation to VCC
VIN_DIGI	24 V	• Reference voltage of digital in- and outputs
VCC	5 V	• Supply of integrated circuits like level changer • Supply of status LEDs • Further transformation to VIO and VCC_ETH
VCC_ANA	5 V	• Extra filtered 5 Volts supply for analogue in- and outputs
VIO	3.3 V	• Supply of main processor • Supply of integrated circuits • Supply of JTAG interface
VCC_ETH	2.5 V	• Supply of external ethernet PHY circuit

As protection, the power supply circuit contains several special components. Directly parallel to the power supply connector (P19), a supressor diode (D26) protects the following circuits against overvoltage. In row-connection to the following circuits, a 1.35 A fuse (F1) protects the boards against high current flow, for instance caused by short lines. In connection to that a common diode (D25) prevents of fault connection on the main power supply connector. Lastly, a choke reduces possible noise of the supply. The protection part of the power supply circuit is shown in figure 12:

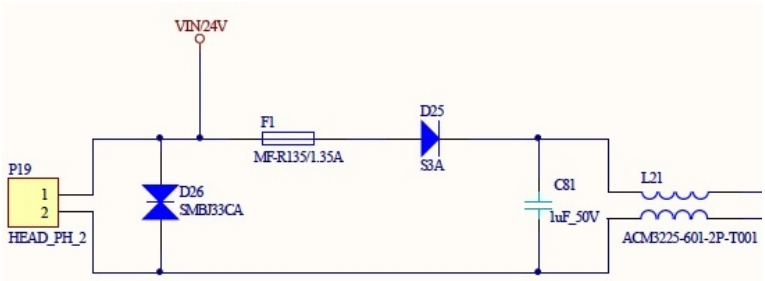


Figure 12: Power protection circuit on power supply

The filtered 24 V behind the choke is used to supply the multiplexers with a high switching reference voltage as well as also supply further step-down regulators to lower voltages. The first of these step-down regulator takes voltage down to 5 V. This transform-stage is mainly responsible for the correct supply of the whole board, because it supplies several devices directly as well as it also drives a second step-down stage to 3.3 V and a third stage to 2.5 V. Because of this requirements, a

high quality and stable step-down regulator with high current throughput is necessary. Since the possible high current, it is highly recommended to adhere the impedance of the regulator and its wiring. The second step-down stage to 3.3 V is directly driven by the first one and is mainly to supply the central processor as well as several internal devices like level changers and memory interfaces. These devices are all not highly sensitive against voltage-fluctuation, but nevertheless a regulator with high output should be chosen since otherwise a consumption peak may cause voltage drops, which in turn may cause a unmeant restart of the processor. The third and last stage only partly supplies the ethernet *PHY* and must not be chose as a high quality device due to the *PHYs* purpose.

As seen in table 3, the digital and analogue in- and outputs are supplied with an own voltage. This is mainly caused by their purpose. The supply for *VIN_DIGI* is gripped directly from the external supply in front of any filters or security components like fuses. In future use, this voltage supplies the digital outputs, which are mainly designed to drive optocoupler-circuits on a second, unknown device. Because the functionality of these inputs cannot be warranted and may be short-lined for any reason, they can cause a much higher current than the outputs are designed to drive. To prevent the whole board of the risk of damage, this supply voltage is separated and secured by an own fuse right in front of the main fuse of the board. This is shown in figure 13, where P19 is the connector for the external power supply (Pin 1: 24 V; Pin 2: *GND*).

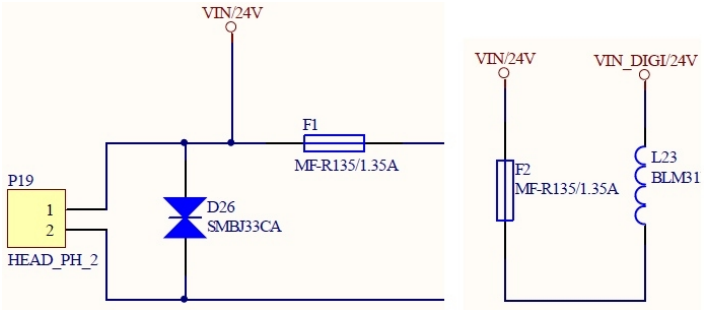


Figure 13: Scheme of the separation of the digital output power supply

From the already to 5 Volts transformed voltage *VCC*, the additional supply *VCC_ANA* is split for the analogue in- and outputs. This is mainly caused by noise reduction on the power lines in order to achieve a more stable reference voltage on the analogue-digital and digital-analogue converters as less as possible influenced by the current consumption of all other *VCC*-supplied components. To achieve this, *VCC_ANA* is filtered and stabilised from *VCC* as shown in figure 14.

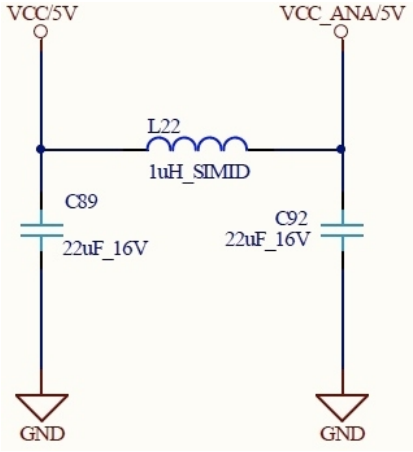


Figure 14: Filtering of analogue supply voltage

7.1.5. Central processor

The central processor is, according to its name, one of the most important components in a digital circuit. Most common designs are based on only one central processor to do all work, where bigger or more complex designs divide several tasks like communications or special memory algorithms to so-called co processors, which are in turn controlled by one or more, to a cluster connected central processing unit(s). The actual design contains one central processor to do the whole work.

The *LPC 2388* in the applied version with 144 pins causes a quite big schematic symbol. Due to the abridgement, this symbol should be divided into two or more single signs, which then can be split to several schematic drawings. It is recommended to split the symbol according to the purpose of the pins, e.g. GPIOs[19], power connection and the debug interface. The processor is able to use most of its pins as a general in- or output, but also use them for instance as special communication or memory interfaces. In case of the *LPC 2388*, most of the pins support four different purposes. These different possible usages should be mentioned in the schematic symbol for each pin. Secondly, it must be decided, in which order the pins are printed. All applicable pins are grouped to five banks with up to 32 pins. These pins are not compulsorily consistent with the external pin number, which then allows a printout either according to the external pin numbering, which is more interesting for the further *PCB* design, or the logical bank order allowing a better overview for the software development. In this paper, the second option will be used due to the design software which already contains a ready-to-use symbol for the *LPC 2388* with the specified order. Figure 15 shows a partial pin printout. It shows the pins ten to 26 of the first pin bank (bank 0) with their different possible purposes. It is observable that the pins used for the digital inputs are set to the general purpose where the pins concerning the MCI[20] mostly are set to a special interface function. Moreover, the figure shows the difference of the two ordering methods mentioned above where the external pin numbers are mentioned outside of the component above their corresponding pin.

19 GPIO: General purpose input output
20 MCI: Memory card interface

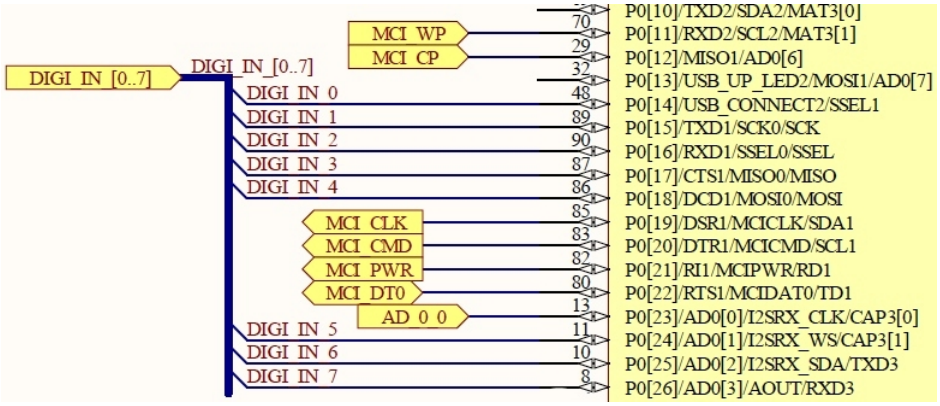


Figure 15: Partial drawing of a pin bank of the central processor

Except from the adaptable pins, the *LPC* contains several more connections to the outside which are also grouped to four more schematic symbols. The first of these is the power connect block, which contains every pin used to supply the processor. Because the *LPC* contains several, independent interfaces, it is possible to supply some of them individually, for instance the build-in converters. Also, reference voltages can be connected separately. At least, the processor is able to be partly supplied with a battery. With this, the internal RTC[21] on a special part of the memory is still on power, so that for instance the clock does not need to be set up every time the processor is switched on. In the actual design, a battery supply is not designated, so that all of the supply pins are directly connected to the power supply of 3.3 V. Except for the battery supply pin, all power connections need to be buffered with an own blocking capacitor of 100 nF.

The next schematic symbol contains the crystal connections. The *LPC 2388* must be driven at least by one crystal, commonly with 12 *MHz*. Without this clock, the processor is not runnable. Although the internal circuits of the processor need different clock speeds, the whole component only needs to be supplied with one external crystal. The *LPC* contains special, individual clock divider registers for the interfaces and circuits that must be driven with different clock speeds. A second, much slower crystal with 32,768 kHz exclusively supplies the *RTC*. As mentioned above this is because the *RTC* is driven by an own supply-pin either on the normal power connection or on battery so that even in battery mode the *RTC* crystal is supplied.

The fourth symbol contains the *JTAG* debug interface connections. This interface allows comfortable debugging directly on the chip and is a powerful tool especially on prototype development. The interface connections are ready-to-use and only need to be added with pull-up resistors on the data and a pull-down resistor on the clock connection. Although the *JTAG* interface only contains six

21 RTC: Realtime clock

connections, it is common to use a 20 pin connector with two times ten pins in a row. This is because the *JTAG* controller outside the board (for instance a PC) also gets its power from the board via two pins and is able to act on some more pins with special actions. In the current design, the *JTAG* controller can also drive the global *RESET* signal on the board.

The last symbol contains every non-connected pin of the *LPC*. Although there are of course no connections to this symbol, it should be mentioned in the schematics to show which pins are not involved in the design.

7.1.6. In- and outputs

The design contains several kinds of in- and outputs with each several pins or channels. Because of this, this chapter will be split-up into these different kinds.

7.1.6.1. Digital

The digital in-and output section is mainly designed to be connected to industrial sensors or devices. Digital means that each output contains two states, which are either *HIGH* or *LOW*. In industrial vicinities it is nowadays common to use 24 V logic, because electrical signals than can be carried on long distances. Furthermore, each channel of both the in and outputs should contain an own *LED* that shows the present state, where an illuminated *LED* means *HIGH*. Each eight digital in- and outputs are needed, from which each four must be switchable to an alternative connector.

To achieve in- and outputs fulfilling these requirements, on both ports some special circuits are necessary. On the outputs, the first step is to raise the *LPC* output level from 3.3 V to the required 24 V. For this, special so-called high-side switches are available. This device is designed to a certain input logic and raises this to a certain output level. The high-side switch is supplied by the special secured digital 24 V voltage. The output signal from the switch is then filtered with a *CLC* low-pass to protect the output against chattering and reduce noise. At least, the signal is wired to a connector to be able to grip the signal externally. Before, the required *LED* is connected to the signal between the *CLC* and the connector with a series resistor to ground. The four-way multiplexer is a special component that allows to switch four digital outputs between two connectors. It is controlled by only one line coming directly from the central processor. The signals the multiplexer should switch are gripped between the *CLC* filter and the *LED* connections.

The digital inputs are working the inverse way to the outputs. The only difference is the method

to reduce the incoming voltage of 24 V to a level the processor can handle. This is done by using a optocoupler circuit that also realises a complete electrical detachment between the incoming signal from a foreign device and the own board, which is shown in figure 16. The optocoupler is secured with a special diode (D3) component that prevents of misconnection and wired to a *LED* (D2) showing the momentary signal status. The coupler (U2) output creates a inverted signal with a pull-up resistor (R5) pulling the output to *HIGH* when the coupler is not active and pulling down the signal to 0 V when the coupler becomes active. This inversion must be considered in the software. Afterwards, eight of the input signals are wired to a multiplexer to use each four of them. The multiplexer is already known from the digital outputs.

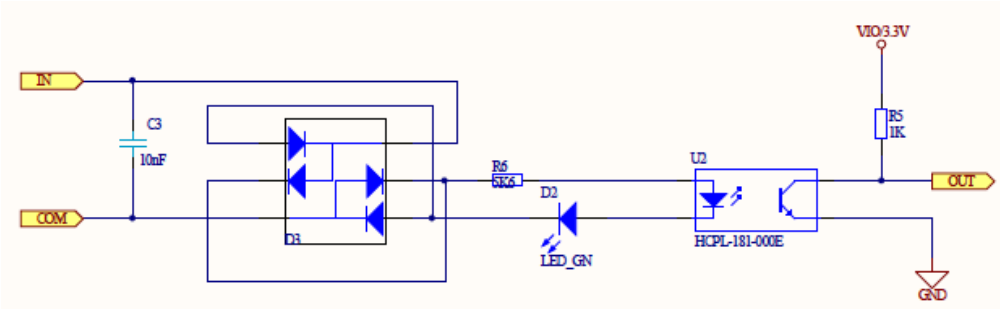


Figure 16: schematic of the optocoupler input circuit

7.1.6.2. Relay

The relay inputs are working the same way as the digital inputs, which means that in fact they consist of the optocoupler circuit shown in figure 16. The only difference to the digital inputs mentioned before is the circuit on the testee, which does not only drives 24 V to an output but switches an incoming signal, so that it has no power consumption of this action. Because of that, every relay-input got its own 24 V output signal, that should be switched by the testee. In addition to that, the incoming signal must not refer to 24 V, which is just the maximum the relais is able to switch.

7.1.6.3. Analogue

The analogue in-and output circuits are the most complex circuit parts in the design. This is mainly caused by three specific requirements to these circuits. Firstly, the internal converters of the central processor should not be used due to their insufficient accurateness which causes external devices inclusive their required wirings. Secondly, the in- and outputs should be able to read both voltage and current, which requires a special circuit combination, because converters are only able

to read voltages. Thirdly and due to debugging, the values from the converters should be accessible with a measure device, which causes an additional pin-row to grip the signals. Eight analogue inputs and six analogue outputs are needed. As on all in- and outputs, at least a few of the signals must be switchable to a alternative connector. In case of the analogue connections, all channels are involved, which requires two four-way multiplexers per port.

On the analogue inputs, the multiplexers are the first devices involved into the circuit. This is mainly to reduce the further needed components and based on the principle that all channels are switched at the same time. Also based on this is the connection to the pin-grid (see figure 17). It is wired the way that it is possible to grip voltage as well as current. For this, each of the eight channels is wired to one input pin on the grid and one output pin. With this, a current measure device can be connected in series to the circuit as well as a voltage measure device can grip the voltage parallel between both pins shorted (for instance with a jumper) and *GND*. Afterwards, The signal is led to a special circuit that allows the detection of either voltage or current, which is shown in figure 18. This is basically realised by a *MOSFET* transistor (Q1) and a series power resistor (R2) in front of a four-way operation amplifier (U1A), on which each one channel uses one amplifier. The *MOSFET* is controlled by an output (CURRENT_IN_A) from the *LPC*. When inactive, the signal is led directly to the amplifier. When the control line becomes *HIGH*, the transistor connects the power resistor to *GND* and so terminates the current throughput. This causes a voltage drop on the amplifier, which is linear to the current though the resistor.

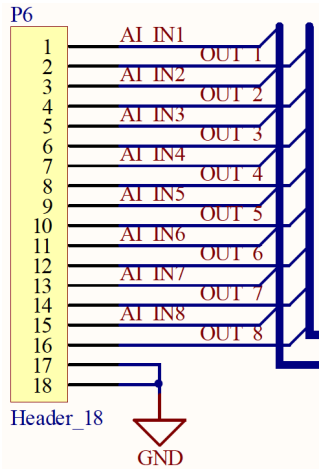


Figure 17: Pin-grid example

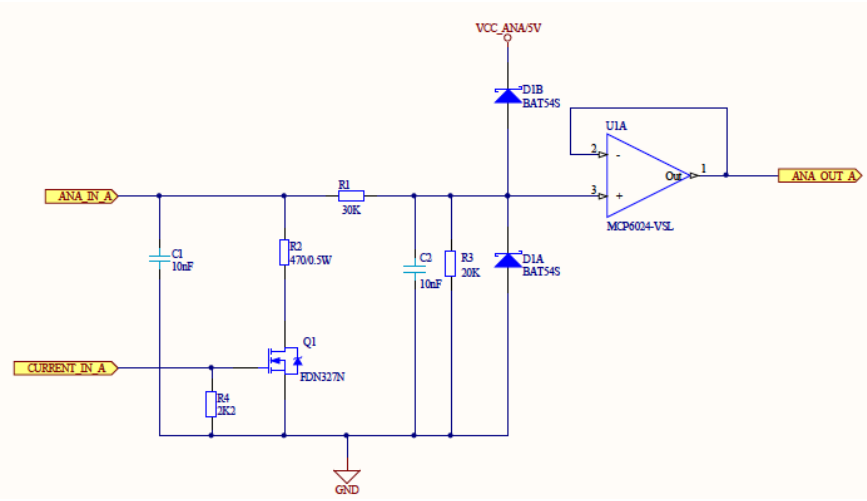


Figure 18: Analogue input frontend with current detection

The applied converter works with an accurateness of twelve bits and is connected to processor via the SPI. It contains eight individual analogue-digital converting channels, so that only one device is needed. To reduce the measurement inaccuracy, a high precision, low power reference device stabilises the reference voltage of the converter.

The analogue outputs work the inverse way of the inputs with the difference, that three two-way digital-analogue converters are needed and because of this three CE[22] connections are required to be able to control every converter individually. The converters are also connected to the *SPI*, so that except for the *CE*-wired no further connections to the processor is needed. As well as the analogue-digital converter, a high precision reference device stabilises the reference voltage. The outlet of the converters are connected to the specific analogue out circuit allowing to differ between voltage and current output, which is partly shown in figure 19. It consists of two divided circuits based on a two-way operation amplifier (U3) where one of these simply amplifies the incoming voltage and the other controls a power transistor (T1) which realises a current flow depending on the output of the amplifier. Both circuits are brought together in a two-way multiplexer (U4) where both channels are controlled by only one connection from the processor (CONFIG). This is because one of the two switches in inverted and so only one of the two amplifying circuits becomes active.

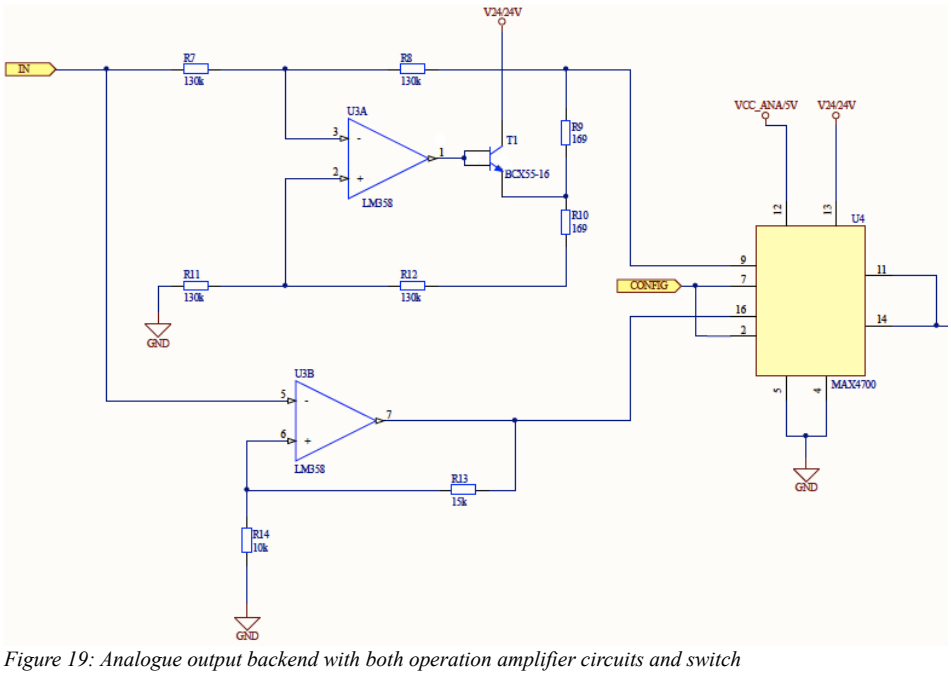


Figure 19: Analogue output backend with both operation amplifier circuits and switch

Like on the analogue inputs, the signals should be accessible on a pin-grid for measuring with the

22 CE: Chip enable

same connection principle. Afterwards, the six signals are connected to two multiplexers to be able to switch them between two connectors.

7.1.7. Communication

7.1.7.1. RS 232

The RS 232 interface is the most used communication interface in the design. It will be used both to control the program flow of the processor, and to write program code into the processor-internal memory.

The *Recommended Standard 232* describes the level and way of connection of a certain serial interface. *RS 232* is the most commonly used serial interface and realisable with a *LPC*-intern UART[23] and a *RS 232* certified level changer. To communicate with the processor, these two connections (*TX*: transmit; *RX*: receive) are sufficient. The Standard prescribes more connectivities, for instance flow-control mechanisms to communicate, for example with modems, which are not necessary in the present context. The two connections should be filtered with a *CLC* filter to reduce high-frequency noise. The values of this *CLC* part must be chosen carefully, because the filter limits the transfer rate of the connection due to the buffer-effect of the capacitors.

In order to be able to write programme code into the processor-internal memory, the serial connection must be switched in some way. To prevent the processor of unintended reprogramming, this change must be done by the user manually with a jumper. So, the global *RESET* signal can be pulled to *GND* with a reset signal from the programming device (e.g. a PC) which in turn resets the processor to the starting address. On another wire the programming device then can start programming the processor via the ISP[24] interface. The *RS 232* connection with the attached *ISP* interface is shown in figure 20. On the *RS 232* connector (J1), the lines 2 and 3 are for the serial communication with the processor, where line 2 represents *RX* and line 3 *TX*. The figure also shows the manual accessible jumper (P1) to enable the *ISP* interface. Lastly, the whole board is reset-able via a switch (S1) that pulls down the *RESET* signal to *GND*.

23 UART: Universal asynchronous receiver transmitter
24 ISP: In-system programming

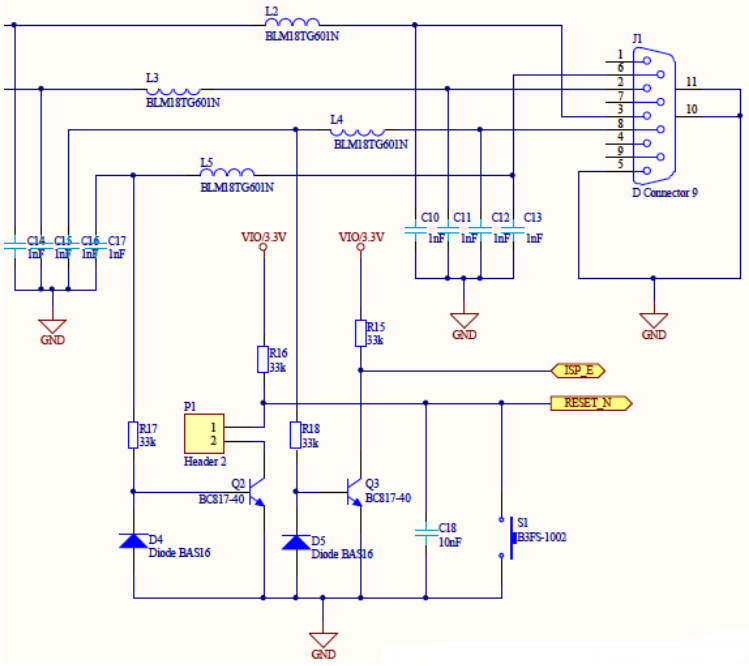


Figure 20: RS 232 interface with attached In-System Programming interface

7.1.7.2. RS 485

The *Recommended Standard 485* as well as the *232* describes the level and way of connection of a serial interface. In difference to the *232* standard, the levels on the *485* are higher so that both interfaces are not directly connectable. The principle behind both interfaces is similar, so that also the *RS 485* is also supplied by a *LPC*-internal *UART* with an attached level changer.

On the current design, the *RS 485* connections are used to create a parallel bus structure. This means that the whole bus consists in fact of two wires, on that several devices can be attached parallel. This principle can be seen in figure 21.

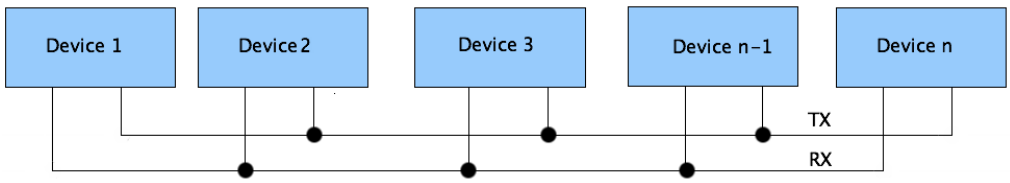


Figure 21: Scheme of a parallel bus structure

The design contains two *RS 485* interfaces and two external connectors to attach them, so that two individual bus structures are possible. Both connectors contain both interfaces, so that the connectors are not bound to one bus. Normally, a device within the bus requires two connectors to be

build into the existing structure, because on every side other members are available. The only exceptions are the master device at the beginning and the last slave device at the end of the bus. The design can represent both one device within the bus or up to two devices on one of the two ends. Also, the device is able to be attached to both busses at the same time.

7.1.7.3. Ethernet

The ethernet connection is based on the multilayer principle of the OSI[25]-model. The specific layers are designed in two different hardware devices, firstly the PHY[26] and secondly the MAC[27] .

The *PHY* is responsible for the physical connection between the outside network and the central processor. It is automated as much as possible and only contains a few registers to be configured. The physical layer is the first, bottom layer of the *OSI*-model. The *MAC* as second layer is the data link layer and responsible for a reliable, as far as possible error-free data transfer. On the *LPC*, the *MAC* is integrated as a hardware interface and communicates with the *PHY* with a special interface called RMII[28]. This interface is designated for the communication between these two *OSI* layers and can either be in full transfer (non-reduced) mode with four bits parallel (MII) or in reduced mode with two bits parallel (RMII).

The ethernet layers are supplied by an own crystal and the *PHY* is partly supplied by an special power supply of 2.5 V.

7.1.7.4. CAN

The CAN is a common used bus in industrial or automotive vicinities. The *LPC 2388* processor contains two independent *CAN* interfaces, which both consists of each a receive and transmit line. To access a *CAN* bus with the processor, an external *CAN* receiver transmitter device is required to change the receive and transmit lines to the specified levels. Depending on the transfer speed, the number of connected devices and the distance to the bus, a 120 Ω resistor is required to terminate the bus between its high and low signal wires. The complete *CAN* connections circuit is shown in figure 22, where P14 represents a jumper to manually terminate the bus connection as mentioned above if necessary.

25 OSI: Open system interconnection
26 PHY: Physical layer
27 MAC: Media access control
28 RMII: Reduced media independent interface

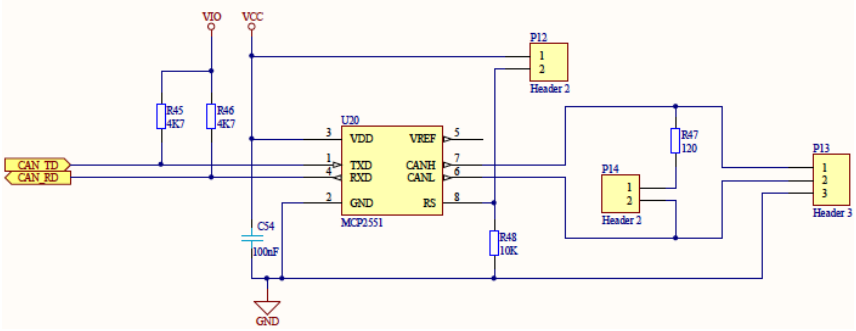


Figure 22: CAN connection circuit

7.1.8. Memory

7.1.8.1. EEPROM

The EEPROM is accessed via SPI by the processor. To warrant low latency, the memory is attached to an own serial interface to which no other devices are connected. Due to its properties as a non-volatile memory, it is used to store important variables or data concerning the device or a special part of a programme, that belongs to this device and should be accessible even if the original program code in the processor was updated or changed. The advantage of this memory is the fact that it is accessible byte-by-byte, which allows a comfortable and fast access and storage of single variables or smaller arrays. In contrast to that, most memories are accessed in a certain page-size which demands to read or write always a whole block, mostly of 512 bytes, which is more comfortable on working with bigger data sizes, but is not given in this case.

The *EEPROM* is used to store board-specific data like serial numbers, certain *MAC*-address spaces or special values like calibration multipliers. Because these data belong to the device and not to the specific programme running on it, it is more comfortable to store them externally. To attach the memory device to the processor, no further devices like level changers are needed. It is directly connected to the first *SPI* of the *LPC*.

7.1.8.2. MCI

The MCI is used to attach removable memory to the processor. It can either work with MMC[29] or SD[30]. The *LPC* contains a implemented *MCI*, that can be set-up with just a few registers. It works

29 MMC: Multimedia cards
30 SD: Secure disk

with a certain set of commands, that mostly fit in both kind of cards. Only one command on the initialisation and a few commands for multiple access to the card differ. With the initialisation difference it is possible to scan which kind of card is inserted to the slot. In hardware, the *MMC* is only reachable with one data line, where an *SD* can be attached either to one or four data lines parallel, which allows a much higher transfer rate up to 5 MB/s. Due to interoperability, all four data connections will be realised. In addition to that, the *MCI* contains one clock line and one command line. Furthermore, two switches allow scans whether a card is inserted and if its write-protected (*SD* only). This protection is based on a mechanical switch build to the *SD* card, which can easily be overridden by software. It will be implemented and used as a kind of notification to prevent of accidentally overwriting.

The data lines all need to be pulled up to 3.3 V as well as the both switch lines. An own power line (MCI_PWR) from the *MCI* is used to switch the power supply of the card. This line is automatically controlled by the interface. The power status becomes visible with an included *LED*. This supply control circuit is shown in figure 23.

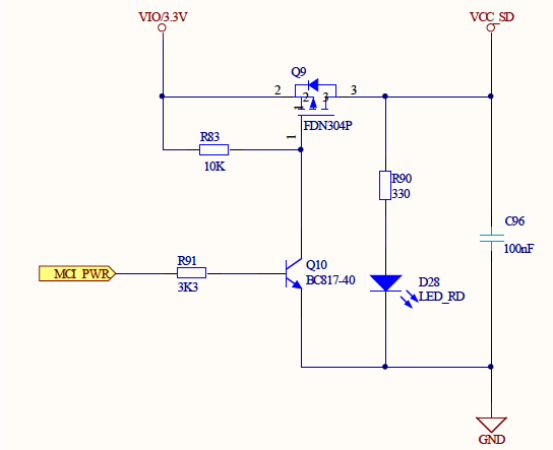


Figure 23: Power supply and control of the MCI

7.2. Printed circuit board design

The design of a printed circuit board is the more complex part of the hardware development. Although there is hardly research action left to do, there are a lot of possible errors in this step of development.

7.2.1. General information on the printed circuit board design

Printed circuit boards are common devices, available and used almost everywhere in normal life, from the coffee machine up to the mobile phone or the notebook. A board is not only used to support electrical connections between components but also to hold them. So aside from electrical attributes, the mechanical and even thermal properties are important for the design. In some cases, a board holds bigger and heavier components or is used to conduct lost heat from components like processors or integrated power circuits. These and probably several more issues must be considered on a *PCB* design before the first connection is drawn.

The starting point of a prototype board like in the current design differs from a commercial, maybe massed-produced product. The first difference is given with the size of the board, which is mostly prescribed on commercial products by the used enclosure. In most cases also the positions of some special components like switches or *LEDs* are given due to their accessibility. These specifications decrease the possibilities of the designer, which may be an advantage in the beginning but can also be turned to a drawback in the last connections. On a prototype development with no specifications about positions and sizes like this design, the first step normally is to chose positions for attachable components like connectors, interfaces, *LEDs* and switches. Moreover, it is common to place the central processor in the middle of the board in order to reach it from every side. The size of the board is at first chosen a bit bigger than necessary and is firstly fitted to a possible size when the mentioned positioning is done. Many further positions and design parts of components and circuits result from this. The last decision before the designing process begins is the definition of the used layers. Generally, the designer choses between a one, two or a multilayer design. The difference between these is mainly given in the power supply and the costs. *PCBs* with inner layers are more expensive but allow inner power and *GND* plains, which permits a low-impedance power supply at every position of the board without getting problems to route the signal layers. Smaller designs are normally designed on two or even just one layer. On larger or more complex designs, usually a multilayer design is chosen, because the advantage of easier routing and the lower imped-

ance outweighs the higher costs. Multilayer *PCBs* must always contain an even number. It is common to start with four layers and add more signal layers only if necessary. Normally, the inner layers are used for *GND* and power and the outer layers for the signal routing. This is mainly caused by the fact that all components are mounted on the outer layers and must be attached to them. Also it is recommended to route the signals of both layers 90° staggered to prevent of crossed connections on one layer.

The different layers are connected to each other by so-called vias, which are small, metallised holes, that establishes a connection between at least two layers. A via can be build in two ways, where the first is a fully metallised hole through the complete board that connects every layer. The second kind only contains conducting material between the layers that should be connected to each other. The first possibility requires an isolating area around the via on every layer, that should not be connected to the signal. The second method only involves the layers that should be connected to each other. However it is more complex in production and therefore more expensive than the full metallised via.

The not-used areas on the signal layers can either be left out free or filled with plains connected to *GND*. Filling reduces the impedance and saves vias to the *GND* layer. It also reduces noise and interaction between signal lines and can cause a more stable system.

7.2.2. Designators, values and synonyms in PCB designs

The designators, values and synonyms already described in chapter *7.1.3.* on page *24* are also used in the *PCB* design. It is common to just write the designator next to the component on the board surface. The reason why values are normally not added to the board is that values are usually mentioned on the component by the manufacturer. The designators are helpful additions to the design both for the person that places the components on the board and for the designer checking and eventually debugging the design. It is advisable to keep one or at most two different writing directions on the board and also to place the designators as near as possible to the component. Only if a lack of space occurs, designators should be left out.

7.2.3. Component positioning

A complete overview on the layout of the *PCB* with the positions of every component is shown in

figure 24.

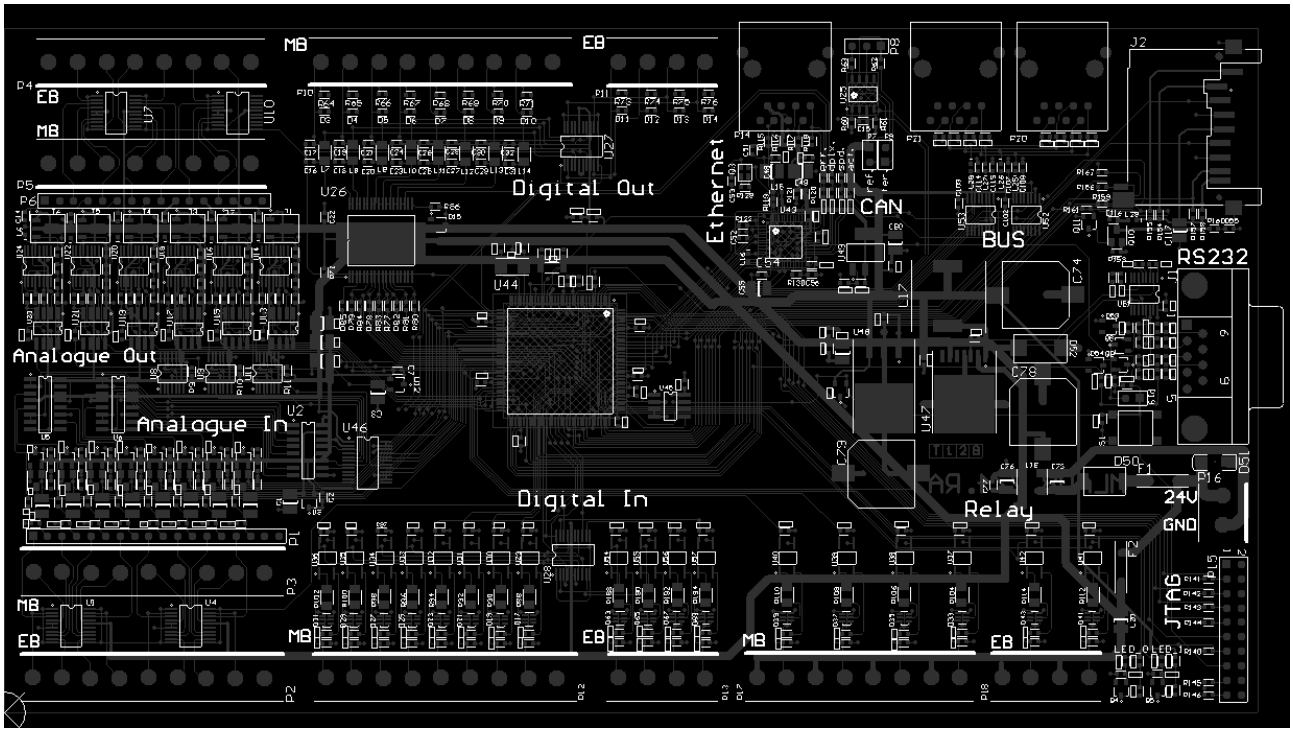


Figure 24: Overview of the component positions

As observed, the analogue and the digital in- and outputs are routed to a adversed, similar design. The outputs are all located at the top edge and the inputs at the bottom edge of the board. This is mainly caused by a better overview and to minimise the risk of wrong connections. All these in- and output sections plus the relay inputs consist of two connectors, where one is routed to the basic settings and the second one can be used instead of several basic channels. In case of the analogue circuits, all channels are switches, either between the main board (MB) or the extension board (EB) connector. On the digital circuits, only four of the eight existing channels are switched. These are the first four channels from the left on the main board connector. The last four channels are applicable in both modes. On the relay inputs, all channels are reachable without a switch in between. The analogue circuit connectors are placed in a special way with the pin-grid, where the pin wired to the connector is always routed in a row to its corresponding pin on the connector, and the other pin of the grid wired to the converter circuits is placed in the gap between two connector pins. This placement is exemplary shown in figure 25 by the connectors of the analogue outputs.

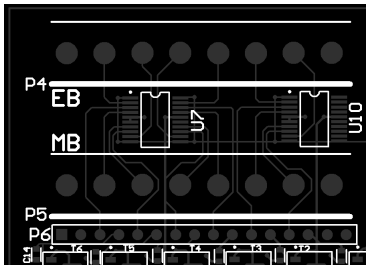


Figure 25: Pin-grid placement of the analogue outputs

On the top edge next to the outputs, all circuits concerning the communication with the testee are located. This affects the complete ethernet circuit including the *PHY*, the *CAN* circuit and the *RS 485* bus assembly. To reduce impedance, the last step-down voltage regulator stage for the ethernet circuit is located directly next to the *PHY*.

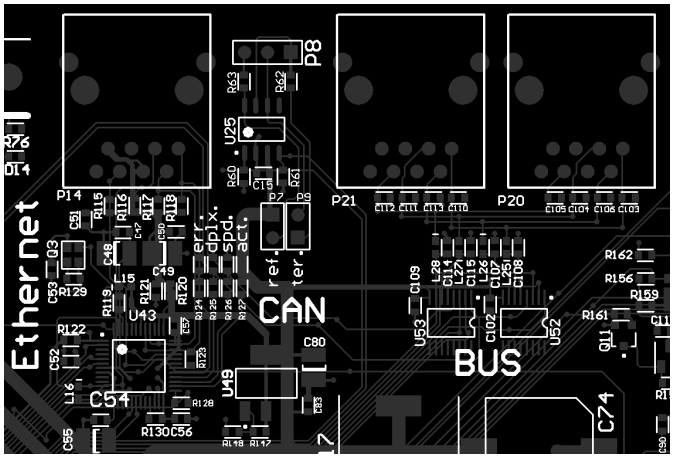


Figure 26: Communication circuits (ethernet, CAN, bus)

Figure 26 shows the layout of the communication circuits. Because the ethernet and the bus assembly use the same connector, they are separated from each other with the *CAN* interface to reduce the risk of wrong connections which may harm the *RS 485* level changer or the ethernet *PHY*. Also displayed in this figure is the possibility to manually add text strings to the board. This is done on some components like jumpers or special *LEDs*. In case of the *PHY*, the four available *LEDs* show different stats of the current ethernet connection, which are rather readable with their logical meaning than just with their designators. The same holds for the two jumpers of the *CAN* interface, where one of them activates the bus termination and the other one disables the transceiver.

On the right edge of the board, the more user-specific components are placed. This mostly concerns the *RS 232* and the *JTAG* interfaces, with which the user controls the board. Also, the power connector, the reset switch and the *MCI* cardholder are placed in this area. This is chosen this way, because the surrounding circuits (bus, *CAN*, ethernet at the top, relay inputs at the bottom) leave

much free space to place the relatively big power circuit components like step-down regulators or capacitors. To reduce the impedance of the supply circuits, the lengths of the wires between them must be kept as short as possible, which is easiest done in the free space of the board. Figure 27 and 28 demonstrates the impedance-relevant connections on the power supply circuits in the schematic (figure 27) and in the *PCB* (figure 28) with red highlights.

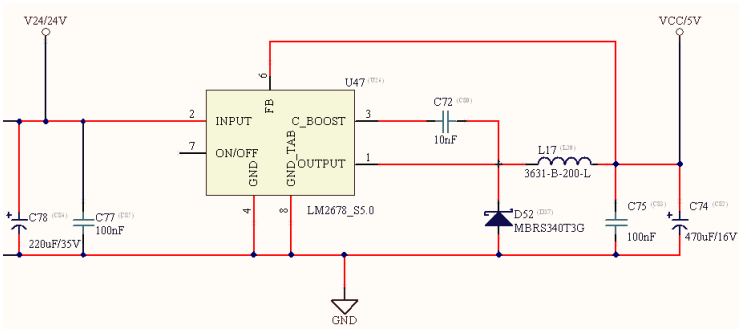


Figure 27: Impedance-relevant connections on power supply in schematics

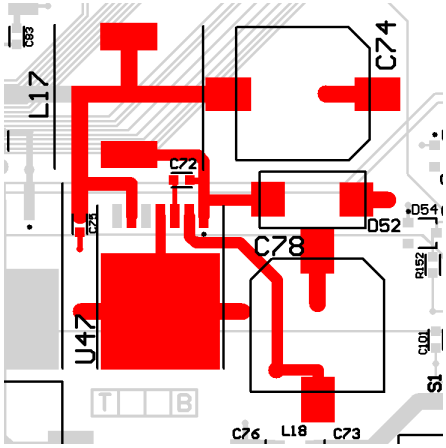


Figure 28: Impedance-relevant connections in PCB

At last, two free usable *LEDs* are located at the bottom right of the board. They are designated to show the momentary status of the processor, for instance a normal running mode with a blinking *LED* and an active procedure with a permanent illuminated *LED*.

8. Software development

The complete software used on the tester board and the test item written in *ANSI C* and Linux script with only a few exceptions on some register commands and interrupt handling that can only be done in *Assembler*. The software is basically split into the two different subchapters *8.1. Tester* and *8.2. Test item*, which are both divided to several sub-projects.

8.1. Tester

The software for the tester device is split into several sub-projects, which is on the one hand caused by a better abridgement but on the other hand makes the individual software parts much easier portable to other projects based on the same processor architecture. Some parts like the file-system library are totally platform-independent.

The following chapters explain the individual software parts. The annotations are kept to the more general functionality of the code and will not be too detailed. Only a basic approach used in the whole project (for instance register manipulation) or very special programme sequences will be clarified with example code lines or the corresponding, special code section. Also, special approach of code-style is shown with short examples in the chapter, where the example belongs to.

As far as possible, already written code or software parts, which are only assumed to this thesis, will be marked as so. Every code file in the addendum got a file header, in which the initials of the author or authors and the date of the file creation are mentioned. Foreign sources such as open source projects contain own file headers with their licence agreement, which regulates the private and commercial use of this code.

Most parts of the software mentioned and explained in the following chapters were created and tested on the target hardware. Nevertheless, the complete software is based on the considerations from chapter CONSIDERATIONS on page XYZ. Occurring problems with this software during the development and testing/debugging of the complete test environment with the testee will be mentioned, explained and if possible solved in chapter MERGE on page ZYX.

8.1.1. Type definitions and code style

The company supporting this project uses an own coding standard as well as own data type defin-

itions. This is due to the use of several different platforms and architectures with the different projects of the company. The own data types are to keep a maximised level of portability and readability of the code. The main problem of data types in C-code is, that some of them differ in size, depending on the processor architecture on which they are applied. For instance, the common type `int` differs between sizes of two and eight bytes. Also, common data types can be used signed or unsigned, which is also important on embedded systems. Code fragment 1 displays the type definition used on the 32-bit *ARM* processor.

```
#define UINT8       unsigned char          /* Definition of type unsigned 8 Bit              */
#define UINT16      unsigned short         /* Definition of type unsigned 16 Bit             */
#define UINT32      unsigned int           /* Definition of type unsigned 32 Bit             */

#define pUINT8      unsigned char *        /* Definition of pointer to type unsigned 8 Bit   */
#define pUINT16     unsigned short *       /* Definition of pointer to type unsigned 16 Bit  */
#define pUINT32     unsigned int *         /* Definition of pointer to type unsigned 32 Bit  */

#define INT8        char                   /* Definition of type signed 8 Bit                */
#define INT16       short                  /* Definition of type signed 16 Bit               */
#define INT32       int                    /* Definition of type signed 32 Bit               */

#define pINT8       char *                 /* Definition of pointer to type signed 8 Bit     */
#define pINT16      short *                /* Definition of pointer to type signed 16 Bit    */
#define pINT32      int *                  /* Definition of pointer to type signed 32 Bit    */

#ifndef NULL                               /* If NULL was NOT defined up to now              */
#define NULL       (0)                     /* Definition of NULL                             */
#endif

typedef enum                               /* Define own enumeration type BOOLEAN            */
{
  FALSE = NULL,                            /* Set FALSE to NULL                              */
  TRUE                                     /* Set TRUE to 'NOT FALSE'                        */
} BOOLEAN;

typedef enum                               /* Define own enumeration type RESULT             */
{
  ERROR = 0,                               /* Set ERROR as a Result member                   */
  OK                                       /* Set Result OK to 'NOT ERROR'                   */
} RESULT;
```

Code 1: Own data type definitions

The coding standard describes the way code has to be formatted within the code files. On the one hand this increases the serviceability of the software as well as the readability, but on the other hand decreases the dependency to the software author. For instance, the coding standard regulates in which way comments have to be written within the code files. Code 1 shows a more simple example of these regulations, where all comments start and end at the same column. Also shown is that all macro definitions are written bold in order to be able to differ between variables and macros.

8.1.2. Hardware configuration

The hardware configuration of the processor is split into four different parts and several different files of the project. The most elementary part is the linker script, which is used by the compiler to manage the memory parts of the processor. The used *LPC 2388* processor contains several different memory sections, like the *FLASH* to store the software code and a general RAM section on which variables are created and commands are executed. Moreover, the ethernet and the *USB* interfaces contain own, small *RAM* sections to store the corresponding stacks. These sections are all mentioned in the linker script.

The second part is the basic processor configuration, which mainly consists of the different clock settings. Although the processor component is attached to only two different crystals, the internal devices and interfaces need several different clock cycles. This is done by internal clock prescalers, which are to be configurated, set and enabled as one of the first steps done with the processor boot up. In fact, the complete clock management depends on the external 12 MHz oscillator. The second crystal of 32,768 kHz is exclusively used by the internal realtime clock.

As the first two parts of the processor set-up are more general settings which can be used on almost every project based on the *LPC 23xx* family and are set before any general application can be called, the other two steps depend much more on the application(s) executed on the processor. The next step is the enabling and configuration of used internal devices like the interrupt vectors, the memory acceleration module or the watchdog module. The last hardware set-up part is the pin configuration. These settings depend only on the application and the attached periphery and are set just in the application using these pins.

Any hardware setting must be written to the corresponding registers of the processor. Because of the strict 32-bit architecture of the *ARM* cores, all registers are accessible on 32-bit addresses and are 32-bit wide. As already mentioned in HW_DEVELOPMENT on page XYZ, the *GPIOs* are bundled to five banks with up to 32 pins, which can be set with writing a logical '1' to the corresponding bit in the banks *SET* register and are deleted by writing a logical '1' to the corresponding bit in the banks *CLR* register. Writing a logical '0' to one of these registers has no effect. The procedure of using two different registers to set and delete a pin is not common but allows an interrupt-stable programming and write- or read-access to all to the corresponding register attached pins in only one clock cycle. On regular (non-*GPIO*) registers, setting and deletion of a bit is done in the same register. Beside the *GPIO* configuration, the processor contains a lot more set-up registers for the sev-

eral interfaces, which partially need much configuration data. The general access to the registers is firstly possible directly on the 32-bit address or secondly with pointer variables to these addresses. The first method is easier and faster usable because no additional file containing all the pointers must be written. On bigger projects, the second method is commonly used because the code becomes much more read- and serviceable if the pointers are named like the corresponding registers in the processor user manual. Appropriate files containing already defined pointers are available for most *ARM* processors in example projects of development board manufacturers. In this paper, this file is called *LPC23xx.h* and contains individual pointers to all accessible registers.

The application of pointers allows direct access to the registers but also requires detailed knowledge of bit-manipulation. It is always advisable to only change the actual bit(s) with logical *AND* or *OR* assignments. Doing so, it is not possible to set and clear bits in a register within one step. This must be done successively. The only exception is to change the complete register value coevally, which should only be used if the non-actual bits do not matter or are not used. Otherwise, certain initialisation or function sequences could be disturbed by setting foreign bits in wrong moments. The following code fragment (code 2) shows the definition and usage of register pointers to set and delete a pin with an attached LED.

```
/* Fast Input/Output pin setup registers in LPC23xx.h                                */
#define FIO_BASE_ADDR        0x3FFFC000
#define FIO4DIR         (*(volatile unsigned int *)(FIO_BASE_ADDR + 0x80))
#define FIO4PIN         (*(volatile unsigned int *)(FIO_BASE_ADDR + 0x94))
#define FIO4SET         (*(volatile unsigned int *)(FIO_BASE_ADDR + 0x98))
#define FIO4CLR         (*(volatile unsigned int *)(FIO_BASE_ADDR + 0x9C))

/* LED Mask definitions in LED.c driver file                                         */
#define LED0_MASK    0x00000010         /* LED is attached to pin 4 (Bit 4) of bank 4 (P4.4)   */

    FIO4PIN |= LED0_MASK;               /* Set pin 4 of bank 4 as output                       */
    FIO4SET  = LED0_MASK;               /* Switch on LED by setting corresponding pin          */
    FIO4CLR  = LED0_MASK;               /* Switch off LED by deleting corresponding pin        */
```

Code 2: Example of register manipulation to set and clear an external pin

The next code fragment (code 3) shows a typical register set-up with predefined bit-masks and afterwards logical assignments to set or clear certain bits in a register. As already shown, the first writing to the register overwrites every available bit, because this special register is not used before this statement and all other important bits are changed afterwards. This style of code-writing is more extensive but on the other hand easier readable and serviceable, because the use of the mask names shows at first sight, what part of the configuration is affected and the used method of writing the operator shows directly if the corresponding bit(s) are set or deleted. To get to know the meaning of a set or cleaned bit in a register, the mask definitions are commented with their effects to the

processor.

```
/* static RAM access register configuration in LPC23xx.h                    */
#define EMC_BASE_ADDR       0xFFE08000
#define EMC_STA_CFG0        (*(volatile unsigned int *)(EMC_BASE_ADDR + 0x200))

/* EMC CONFIGURATION REGISTER MASKS in EMC.c                                 */
#define EMC_BANDWIDTH_8     0x00000000     /* Bit 0-1. 00: 8 Bit             */
#define EMC_PAGE_MODE       0x00000008     /* 1: Async mode, 0: disabled     */
#define EMC_CS_POLARITY     0x00000040     /* 1: Active HIGH, 0: active LOW  */
#define EMC_EXTENDED_WAIT   0x00000100     /* 1: EW enabled, 0: disabled     */
#define EMC_BUFFER_CONTROL  0x00080000     /* 1: Buffer enabled, 0: disabled */
#define EMC_WRITE_PROTECT   0x00100000     /* 1: protected, 0: not protected */

/* Set up Static Memory Configuration Register in EMC.c                     */
    EMC_STA_CFG0  = EMC_BANDWIDTH_8;       /* 8 Bit Buswidth and Register Reset */
    EMC_STA_CFG0 |= EMC_PAGE_MODE;         /* Async Page Mode                */
    EMC_STA_CFG0 &=~EMC_CS_POLARITY;       /* Active LOW Chip Select         */
    EMC_STA_CFG0 &=~EMC_EXTENDED_WAIT;     /* Extended Wait disabled         */
    EMC_STA_CFG0 &=~EMC_BUFFER_CONTROL;    /* Write Buffer enable            */
    EMC_STA_CFG0 &=~EMC_WRITE_PROTECT;     /* No write protection            */
```

Code 3: Example of the register manipulation codestyle with set and clean statements

If a mask is used in Eclipse, the mouseover display shows its comment, although the definition is outside of the actual screen or even the file. This special behaviour is shown in figure 29, where the mouse (not visible) points to the `EMC_WRITE_PROTECT` mask.

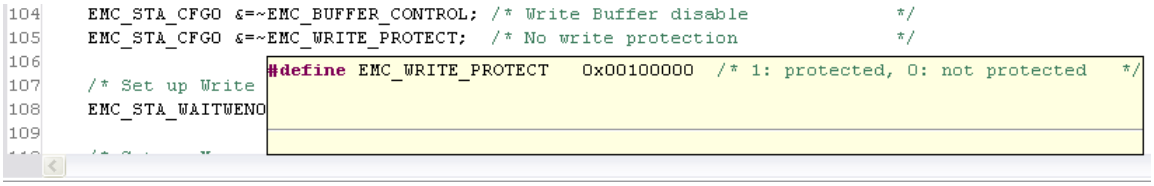


Figure 29: Example of the mouseover-display function in Eclipse IDE

A last option to manipulate the processor registers is to build structures of every register with members for all mated bits. This method allows an easier usage, because the members can be used as normal variables and so set and clean bits within one logical step. The disadvantage of this method is the high effort to create the structures as well as the fact that the compiler will build the same bit manipulation commands like the method shown in code 3, with the difference that the structure method does not show exactly what the compiler will generate.

Because the software development is done for most parts on other devices than the target tester board, the individual hardware-using software segments are switchable between three different hardware environments by changing only the switch. These environments are firstly a development board from the manufacturer *Olimex*, a company-intern product called *IO-Controller*, which is in several parts similar to the tester board, and lastly the tester board itself. Code 4 shows the switch

definition and the general usage within this thesis. The keyword \MARK is a part of the *Eclipse task function*, which allows to mark codelines or -sections with several, self-created keywords like for instance \TODO, \NOTICE or \MARK. The slash identifies the following word as a task keyword. All lines marked this way are mentioned in the Eclipse task window and allow a direct step to this line or section. This allows a better abridgement.

```
/* Pinset Switch declaration and definition in LPC23xx.h                                    */
/* For Olimex DemoBoard            : PINSET_TESTER = 0
 * For Pinsettings of IO-CTRL REV_C: PINSET_TESTER = 1
 * For QUA_2475_TESTER             : PINSET_TESTER = 2
 */
#define PINSET_TESTER               2

/* PINSET_TESTER switch usage in example file (dac.h)                                       */
// \MARK NEW PINSETTINGS FOR TESTER (2)
#if (PINSET_TESTER == 2)
#define maxDAC_Channels             6      /* If actual board is tester, set 6 DAC channels   */
#else
#define maxDAC_Channels             4      /* If other board than tester, set 4 DAC channels  */
#endif
```

Code 4: Definition and usage of the pinset switch

8.1.3. FreeRTOS operating system

The FreeRTOS[31] is an open source operating system for embedded systems and is available for many different processors. It is freely usable for private as well as commercial applications, but without any warranty. The operating system is already ported to the *LPC23xx* processor family, which allows an easy usability and requires just minor customising to the tester application, such as memory allocation. *FreeRTOS* contains realtime behaviour, and in addition to that multitasking functionality, time basics and semaphores.

The multitasking principle is in fact just an illusion, because every conventional processor can only execute a single command per cycle. The trick is to rapidly switch between single tasks and make it appearing like multiple tasks running concurrently. This is shown in figure 30, where the upper graph displays the subjective and the lower graph displays the real behaviour of the system. The order, in which the individual tasks are executed one after another can be affected by the priority assigned to every new task.

31 FreeRTOS: Free realtime operating system

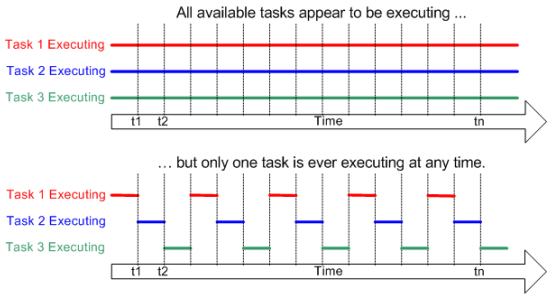


Figure 30: Multitasking principle of FreeRTOS (source: FreeRTOS.org)

FreeRTOS requires at least one task running in an endless loop. This is caused by the fact that the complete operating system consists of a simple application that is ended if no more code is available to execute. As long as the so-called *idle_task* runs, the system remains actively and new functions and tasks can be called or created (for instance by interrupts or remote calls). The *idle_task* of *FreeRTOS* is comparable with the idle task of other operating systems like *Windows* or *Unix*. New tasks are created with `xTaskCreate` using parameters like the function name, function arguments, the task priority and the stack size. The task management system is based on the principle of endless tasks, which means that tasks that end because their code ends are not deleted. They can be started again without any need to recreate them. So of course they also block memory in the stack if they are just ended and not deleted with the command `xTaskDelete`.

Time basics are of great importance in real time systems. The operating system is able to create a quite simply usable time stamp in milliseconds. This allows to directly handle times within the code. The command `vTaskDelay` in combination with a time in milliseconds as parameter forces the system to hold the containing process or task for the given time. Thus, almost 100 per cent of calculation time becomes available for other processes.

Semaphores guarantee the exclusivity of functions. Even in stressed and critical situations, a real time operating system needs to keep its ability of forecasting. In many cases, more than one process are able to access a certain function or variable. In critical situations, these parallel-accessible functions must be secured of a simultaneous access. Usually, this is done by flags behaving like normal variables which are changed and requested for their value. Although this method is the simpler one, it contains an important disadvantage: the setting of a normal variable may take more than just on calculation cycle, and with this the exclusivity of a certain, secured software sequence is not warranted. The more technical form for this is *devideable*. In contrast to that, semaphores are *undevidable*. In *FreeRTOS*, semaphores are from an own type and are taken and released instead of set and

cleaned like flags. Before semaphores may be used, they must be created. This is done with the command `vSemaphoreCreateBinary`. The function `xTakeSemaphore` returns a numeric value that indicates whether the semaphore was taken or not. The software then is able to request for this and act further. Exceptions are numeric semaphores; they are able to be taken multiple times up to a defined maximum. With `xReleaseSemaphore`, a semaphore is freed for a new access. If multiple processes try to access an already taken semaphore, *FreeRTOS* will not create some kind of queue list for it. The access will be given to the first process in stack. Also a queue list based on priorities is not possible, which turns the securing process based on semaphores into a serious exercise.

8.1.4. Drivers

The drivers sub-project contains all functions necessary to access internal or external attached periphery, which also includes special hardware configurations (see chapter *8.1.2. Hardware configuration* on page *50*). Hereby each periphery or device got an own driver structure, which consists of a header- and a code file (see figure 31). This structure may also be called module. If a certain periphery becomes more complex (like for instance an *USB* stack), the code should be divided into more than two files and be arranged in an own sub-folder. The basic idea of this structure is a maximised dynamic use.

Each driver contains an initialisation function, which sets hardware configurations (e.g. pin settings), starts up or loads special interfaces, allocates memory and sets the periphery to a certain start condition. In the following, functions to interact with the corresponding periphery are available within the code file, for instance read and write functions on the memory drivers or special algorithms to build *SPI* commands. Each of those functions, that should be available from outside of the corresponding code file is prototyped in the drivers header file. Also, driver-specific type definitions, settings and variables, which should be used from other functions, are defined in the header. With this method it is possible to simply include the drivers header file to an application file and so get full access to all data types, functions and drivers-specific variables.

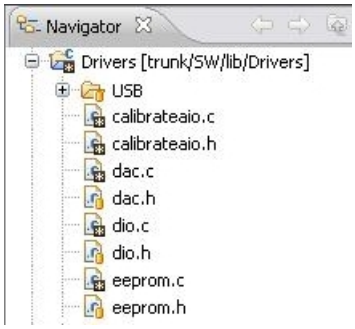


Figure 31: Driver file structure

The initialisation functions are usually called from the main function right before the operating system becomes active. In the initialisation sequence, a certain order must be complied, because

some peripheries depend on each other, as for instance the *EEPROM* depends on the *SPI* bus. In general, the whole driver sub-project should contain as few dependencies and interactions with other projects as possible. In the present software, the drivers only depend on the register-definition file *LPC23xx.h*, the custom data type definition file *types.h* and in some cases on parts of the operating system.

8.1.5. Communication protocol

The communication protocol is used to transfer commands and data between at least two environment devices and is based on the Master-Slave principle. This means that the devices within the communication are not equal but depend on the master. The advantage of this bus architecture is an easy development, the drawback is the total dependency of the communication on the master device.

The used protocol is based on a company-internal, general communication system and was adjusted to the purpose of this thesis. All devices within the bus contain an own device number between 1 and 256, whereas 1 stands for the master and all numbers above for slave devices. Moreover, every device must contain the highest used number and so know the total number of devices within the bus. According to several functions like the in- and output drivers it is possible to choose the device on which the command is to execute. This is realised by handling the device number within the function call. Although it is possible to execute a command on the own device by choosing the own device number. Device number 0 always stands for the self device and does not make use of the protocol.

The interchange of commands and data happens in cycles stimulated by the master device. Each cycle belongs to only one slave device within the bus, whereas at first the master transfers his commands and data, and afterwards receives results or function calls. The slave also transfers a *cycle finished* symbol, which enables the master to start the next cycle on the next slave device.

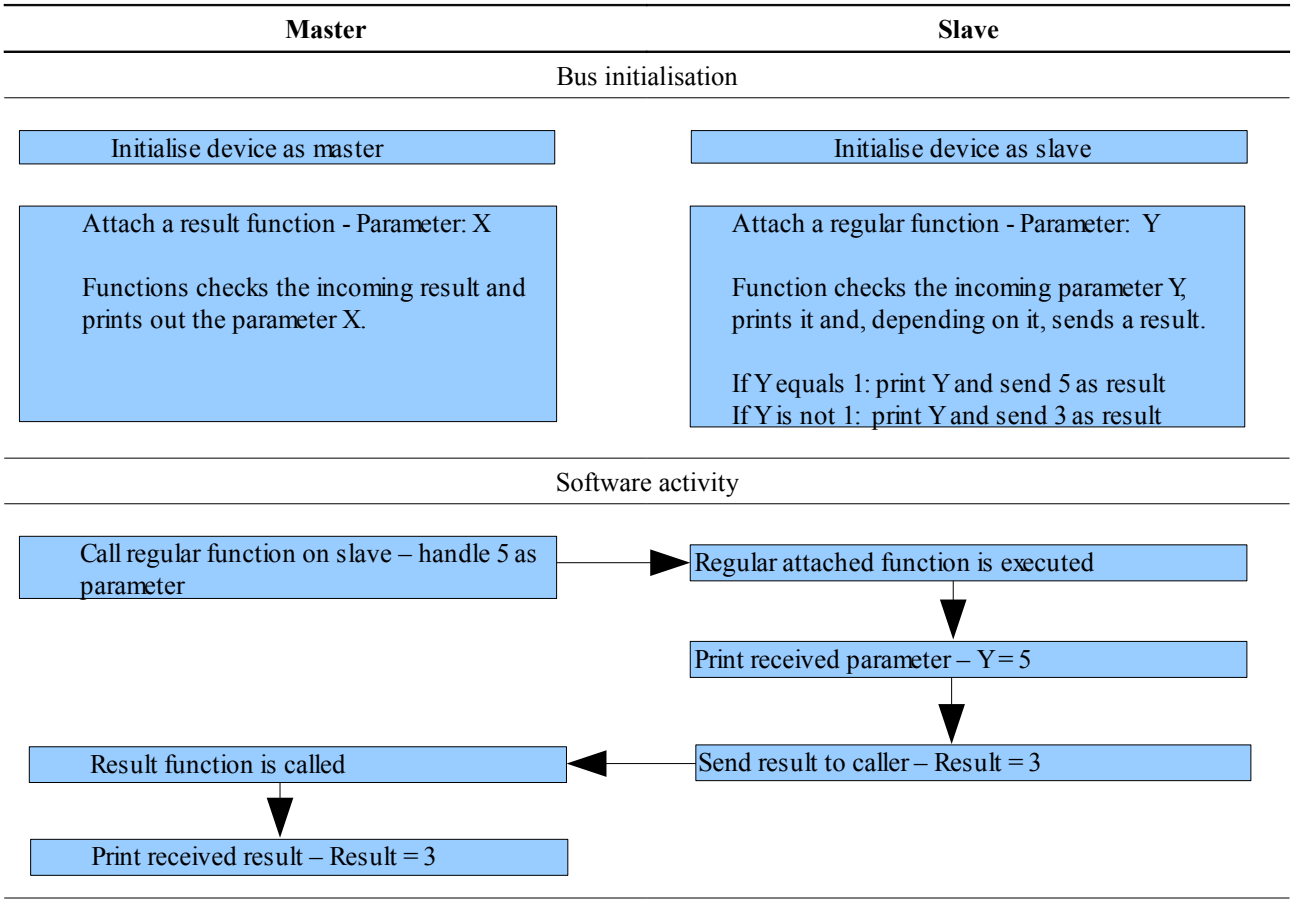
The bus architecture interferes with the operating system and initialises an own task on its start-up. All functions called from the bus are executed within this special task to prevent collisions with the regular running software. Apart from the task creation, the bus initialisation also registers a device to a certain number and thus either to be the master or a slave. The registering is also bound to a certain communication interface, which is in this thesis on of the *RS 485* front ends. In switching to a different interface, the cycle timings may be retuned. The communication protocol is mul-

tiply usable at the same time, which means that a device for instance may act as the master on one bus and as a slave on a second bus. Because of that, the binding to an interface is marked in a bus handler variable with which it is possible to identify each bus affiliation.

The function management is based on two different bus function types, where one is a regular call and the other is a result call. The difference between them is the moment in which they are called; the regular function is always executed in the beginning of a cycle and a result function at the cycle end. So, it is possible to command a device to a certain task and to receive a result of this task within one cycle. This fact gets more important the more devices are involved in the communication structure. Functions must be shared with the bus in order to be able to execute them remotely. Each device contains two function-pointer arrays in which the attached regular and result functions are listed. A function is attached with `bpAttachRpc` (for regulars) or `bpAttachRpcResult` (for results). The attachment requires parameters, which consist of an individual function number and the number of maximal parameters that should be transferred within the remote call to this function. If no parameters are necessary, the number of parameters for attaching is `NULL`. In fact a remote function call consist of the transfer of the individual function number and of an optional list of parameters.

Table 5 shows an example use of the bus protocol. The mentioned software activity is placed in only one bus communication cycle.

Table 5: Communication protocol command and data flow example



8.1.6. Test applications

The test applications are the main adapted part within this thesis and are split into two different sections and folders. First, the tester contains several self-test applications to assure its functionality. The second kind of applications represents the remote hardware design test functions. Most of the self-test and remote test sequences are based on the same principles but are realised in different ways. This is mainly caused by the fact that the self-test must not be as automated as possible and will rarely be executed.

In order to keep an overview, the test application files creation is based on the same principle already known from the drivers project; a header file for all prototypes, global defines and type definitions and a code file for the software. Every file that directly contains test applications is marked with the prefix "*test*" within its filename. The self-test folder solely consists of files like

this. The remote test applications additionally require special driver software for the communication with the other bus device.

The individual test procedures are all based on the deliberations of chapter CONSIDERATIONS on page XYZ. In order to achieve a dynamical use, each test consist of a general execute function which controls the further test procedure. If a certain test should be executed, only this function of type BOOLEAN is to be called. All further, individual test functions as well as the test result calculation for this periphery are done within the general execute function, which at its end returns the calculated test result.

The remote test functions are partially based on complex communication with included wait states realised by semaphores. For that reason, these test applications need an own initialisation sequence to create and take the semaphores. Because the operating system will collapse if a semaphore is taken or released without its previous creation, the semaphore-using functions must be secured against non-initialised access. This is most easily done with a BOOLEAN flag. Each individual semaphore-using test sequence got an own, global flag which is set to FALSE on start up and set to TRUE if the corresponding test initialisation is finished.

Each test is totally independent from the other tests. This is done to be able to call each sequence individually. In order to achieve a highly automated test environment it is necessary to create a fully automated test routine based on the single tests. Within this it is appropriate to acquire and apply a logical order and to react on and interact with previous results. Figure 32 shows the applied test sequence order and the possible dependences on other test parts.

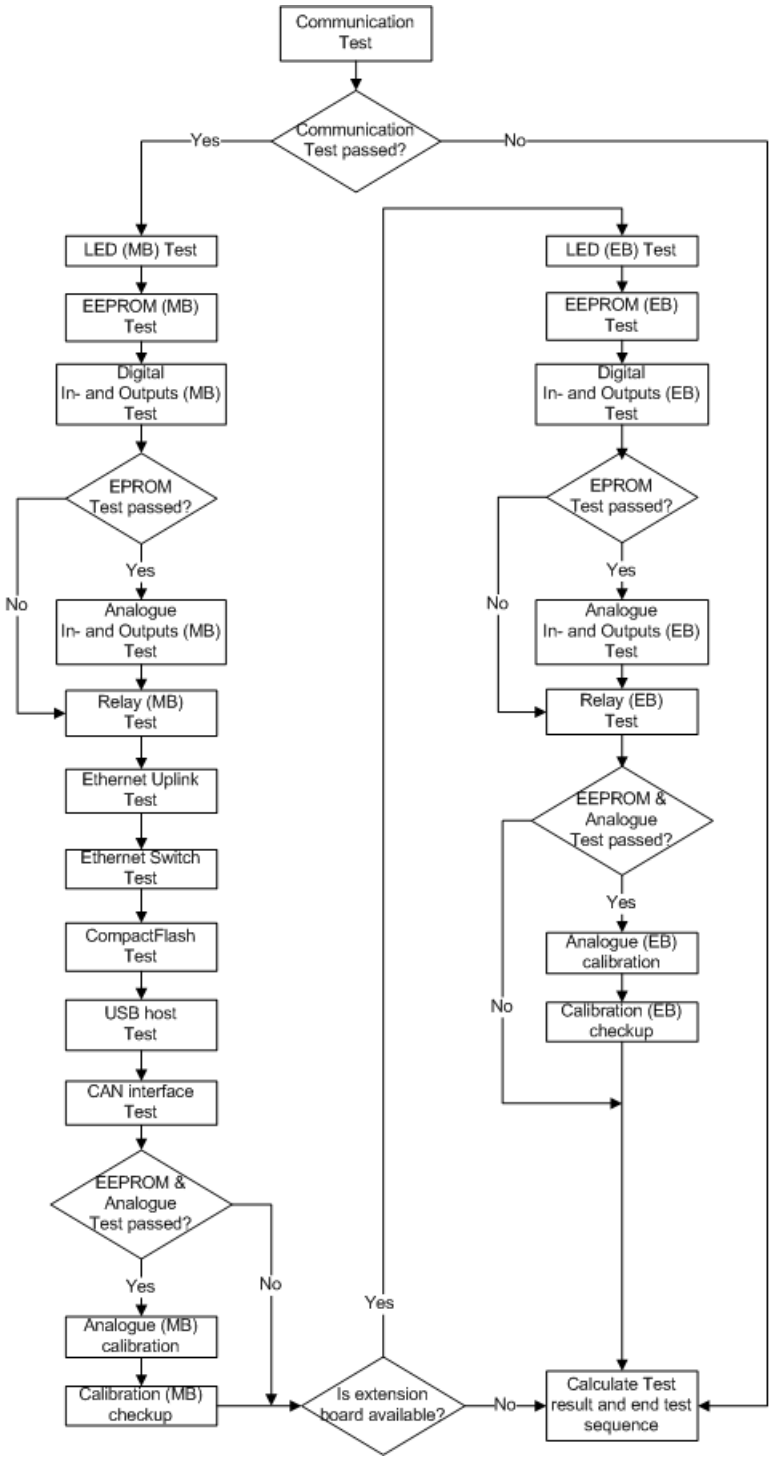


Figure 32: Flow chart of the complete test sequence

8.1.7. Additional software

Within the application project some additional software parts are also implemented to simplify the tester usage. Mainly this is the menu structure which enables the user to interact with the test en-

vironment. The menu architecture refers to a common Linux terminal structure but is strongly limited to more general abilities. The menu basically consists of four different arrays, out of which the first contains the command, the second the corresponding function pointer, the third an calculation offset and the last some optional help text strings. The recognition of a command is built with string comparison. To reduce this intensive compare calculations, the command table is ordered alphabetically. Before the complete command is processed, the first character of the command is compared with the offset table. If commands with this first character are available, the offset of this character for the command table is returned and enables the menu to start comparing the commands at the first entry with the given character. The offset table is built and calculated within a menu initialisation function, which must be called before using the menu. Figure 33 shows the flow chart of the command-input.

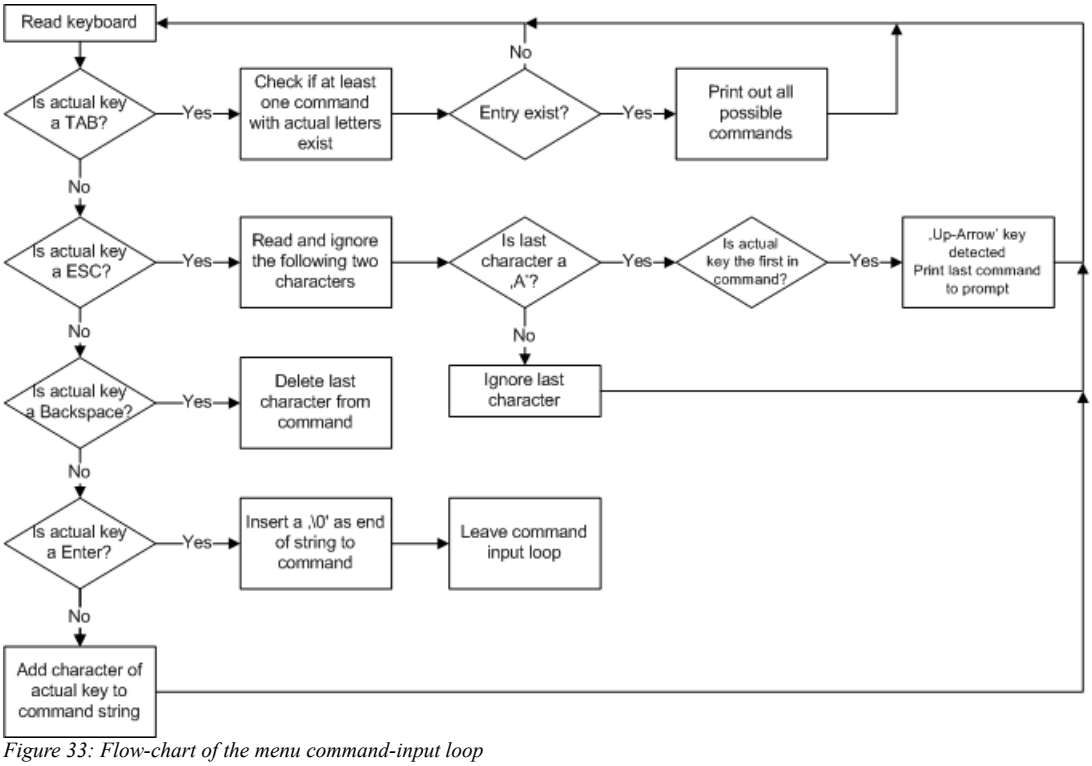


Figure 33: Flow-chart of the menu command-input loop

The menu terminal is able to handle a maximum of three 32-bit arguments which can be handled directly next to the command. As known from other command-line interpreters like the UNIX terminal, the command as well as the arguments are typed within the same call and so are written to only one string. The differentiation between command and arguments is done by the standard function `strtok` which is able to divide a string to several so-called tokens. This division is based on a character list that must be handled to `strtok` as well. Every time `strtok` detects one of the characters

given in the list, a new token will be split. The divider characters are not handled to any token. At last, it is possible to print out short help messages for each available command. This is done with attaching the suffix "_help" to a command. The printed aid shows the usage of the corresponding command and optional or required arguments. Figure 34 explains the command detection and the corresponded handling of the menu.

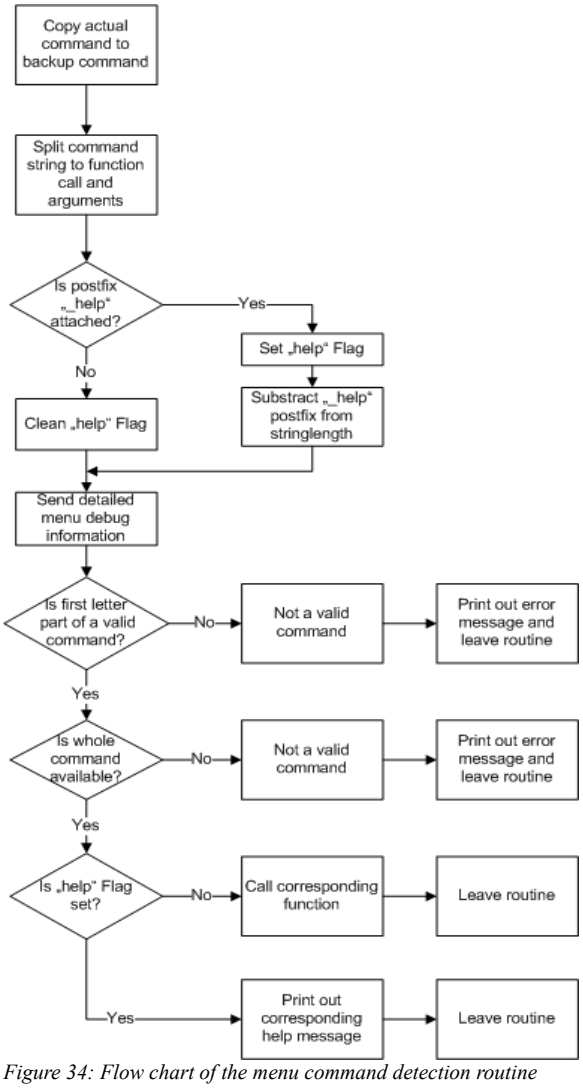


Figure 34: Flow chart of the menu command detection routine

The arguments, which are optionally handled with a command, are designated to specify the function called with it. For example it is possible to access every digital output directly from the command line. Figure 35 shows an example use of the *help*-functionality and the argument handling.

Figure 35: Example usage of the created command-line interpreter

The call `digiout` represents the actual command, the fist argument stands for the device number within the bus, the second argument for the effected channel and the last argument for the value to that the output should be written to. The order of the arguments cannot be changed by the user but is created to be consistent. For instance all commands which effect the bus handle the device number as first argument.

A second addition to the regular software is the logging tool. This function is mainly implemented to be able to save every test result of an individual test item within one text file. This allows an easier overview on possible errors of each test item board and prevent of manual notes.

The implementation of the logging algorithm is based on a FAT32[32] filesystem, which is taken from earlier projects and uses the memory card slot. So it is possible to read the logged data on other computers. The filesystem provides functions to create and edit files. In order to log every incoming and outgoing message of the tester device it is only necessary to add a write-statement to the write and read functions of the *RS 232* interface. By doing this, every message handled on this serial connection is automatically written to the log file. In addition to the message, also other information like the transfer direction, the used interface and the message urgency are transferred to the log function and afterwards written to the log file.

The logging can be en- and disables by changing a `BOOLEAN` flag implemented in the logging algorithm. The memory card must be initialised and mounted before it is able to be used.

32 FAT32: File allocation table (32-Bit based)

8.2. Test item

8.2.1. Test software integration to the existing operating system

Unlike the tester board the test item operating system is based on an actual Linux Kernel version implemented to a small and limited Linux environment. Because of that, the integration of specific test software is more complicated and requires a development environment which is especially build for the applied processor. This is done with the help of the *Buildroot* package, which is able to be set to several different processors and peripheries and so is able compile the software written for the test item.

All functions concerning the communication as well as the test applications are built into one binary file which is afterwards executable on the test item. Although it would be possible to create different binaries and let them interact depending on the demanded actions, this approach leads to a much more complex development without an appreciable gain.

8.2.2. Communication

The communication protocol source code in the test item in most parts is similar to the code used on the tester hardware. In contrast to *FreeRTOS* it is not easily possible to embed the protocol to the Linux operating system. This is because the kernel and the surrounded operating system are already compiled and built to the test item. A second difference is the interface used to communicate which is unlike the tester on the test item managed and controlled by Linux as well.

To solve these problems it is necessary to reconfigure all functions that directly handle variables and data flows between the protocol and the operating system. These are the write and read functions on the interface and the protocol-internal task management. *ANSI C* enables interaction between binary code files and the operating system, which allows an uncomplicated cooperation.

As already known from chapter *Communication protocol* (page *56*) it is necessary to run the communication within an endless loop. On the tester device this is done automatically by the operating system which is run endless as well. The binary file on the test item must be kept alive with the explicit use of such a loop. To prevent of stressing the processor, a sleep function is called within the endless loop. Both is shown in code sequence 5 .

```
  /* Macro definition of device numbers in protocolfunctions.h
*/#define THISDEVICENUMBER        2
  #define MAXDEVICENUMBER         2

  /* Main function of the test item binary
*/
  int main (void)
{
    /* Initialisation section                                                  */
      IO_CPU_Init();                                    /* Open data stream to IO_CPU
*/
    protocolInit(THISDEVICENUMBER, MAXDEVICENUMBER);  /* Initialise device as slave    */

    for (;;)                                          /* Run endless loop for protocol */
        {
        fflush( stdout );                             /* Flush FIFO of UART interface  */
        usleep ( 10000 );                             /* Free CPU time (10 ms)         */
        }

       return 0;
}
```

Code 5: Main function of the test item binary

`IO_CPU_INIT` opens and enables a data flow stream between the main- and the co-processor. This allows a simpler access to the available periphery. The test item only reacts on functions calls from the outside. This allows an relatively easy main function and does not require any development of command or text input handling. On the other hand this operation method requires an strict and secured software development to prevent of collapses and crashes. It must be taken into account that the user is only able to start and stop the execution of the binary file without any possibility to interact with it. So for instance communication errors, time-outs or other possible sources of error must be eliminated.

8.2.3. Script and C-Code development for individual test applications

Unlike the tester device software the test applications for the test item are designed to only react on function calls. Because of that, the code development is limited to a reacting behaviour based on the communication protocol. As shown in code fragment 5, the main function does not support any control mechanisms. The only statement that has an active effect to the behaviour is the initialisation of the communication. Within this all functions, which are supposed to be reached from the bus system are attached to the protocol.

The attached functions must keep a prescribed argument list given by the protocol. These arguments include the number of the caller and of the target device, the number of the called function, a request number, the number of optional handled parameters and the parameter array. Because data is

handled only within this array of type `UINT32`, it is advisable to mention their meaning in comments in detail. The function argument list and the parameter meaning comments are available in every remotely accessible function and is exemplary shown in code 6.

```
void digitalWrite ( UINT8 senderId, UINT8 targetId, UINT8 requestNr,
                    UINT8 functionId, UINT8 nrOfParams, UINT32 *params )
{
    /* nrOfParams = 2
     * params[0]: digital output channel number
     * params[1]: digital output value (0: LOW, !0: HIGH)
     */
```

Code 6: Remote-accessible function argument list and parameter-array description

Generally it is possible to access all periphery from within the compiled *C* application. To do so, the module *smc4000io* enables an easy-to-use interaction of application and periphery. This interaction is based on the data stream between main- and co-processor.

In a few cases it is advisable to create Linux scripts in order to develop a certain test. A script is a collection of individual Linux terminal commands which are executed directly after another. Unlike the command-line interpreter provided by the Linux operating system, it is possible to use loops as well as if-conditions and switch-statements. The memory as well as the ethernet interfaces are easier controllable by the operating system itself. The corresponding script is called by the standard function `system`.

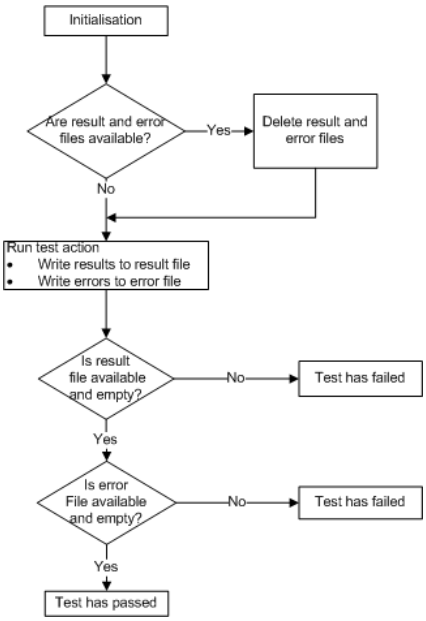


Figure 36: General flow chart of a Linux test script

Figure 36 displays the approach of a script test application. Although the tested items differ from

each other, the method is generally the same. As shown, the script tests are based on the principle of file comparison. Usually Linux commands only return messages if an unexpected behaviour appears. Only a few functions print out texts generally. Normally, this behaviour can be disabled by setting an special argument in the function call. Linux is able to canalise data flow to a file instead of the standard output port. The same holds for error messages. The code lines in fragment 7 explain the canalisation and the file handling within a test script.

```
# Define terminal type
#!/bin/sh

# File-ending hint: "ori" = ORIginal; "rb" = ReadBack; "res" = RESult; "err" = ERRor

# Mount USB stick to /mnt/usb
mount /dev/sda1 /mnt/usb

# Copy the golden file from test folder to USB stick
cpy /test/goldenfile /mnt/usb/

# Compare both files - canalise standard output (1) to result file and error (2) to error file
diff /test/goldenfile /mnt/usb/golenfile 1>> /test/result/usb.res 2>> /test/result/usb.err

# Print out the content of usb.res
echo -e "\n\rContent of usb.res"
cat /test/result/usb.res
# Print out the content of usb.err
echo -e "\n\rContent of usb.err"
cat /test/result/usb.err

# If result and error file exist and are of size NULL, test is passed
if [ ! -s /test/result/usb.res -a ! -s /test/result/usb.err ]
then
    # Test was successful
    echo -e "USB1 TEST-PASSED"
    echo "USB1 TEST-PASSED"  >> /test/result/usb1.res
else
    # Test failed
    echo -e "USB1 TEST-FAILED"
    echo "USB1 TEST-FAILED"  >> /test/result/usb1.res
fi
```

Code 7: General data flow canalisation and file handling within a test script

As shown, the *Linux* script language differs from *ANSI C*, especially in the field of combination. The argument `-a` within the if-condition stands for a logical AND. Also it is necessary to insert space characters around every parenthesis and every argument. The `fi`-statement ends the if-condition. Linux script does not support the grouping of statements using cambered parenthesis.

The compare-statement `diff` only prints out a message if the comparison fails. If both files are equal, the result file as well as the error file will be created but left empty. The print-out of the file content is mainly caused of debugging.

The interaction of the *C* application and the test script happens indirectly. As shown in the example test script (code 7), the test results and errors are written to own files and are not directly

handled to the caller. The function system holds the *C* application as long as the operating system is busy executing the script and continues running when the script is finished. Afterwards it is possible to open and read the result function directly from the test software. To do so, it is at first necessary to open the corresponding file using the command `open`. From this, a file descriptor is returned which allows access to the opened file stream. With `read`, the content of the result function is read to a local array, which is afterwards scanned for the two keyword “PASSED” and “FAILED”, which indicate the test result. If `open`, `read` or the keyword detection fail, the test is not passed as well. Figure 37 shows this approach within a detailed flow chart.

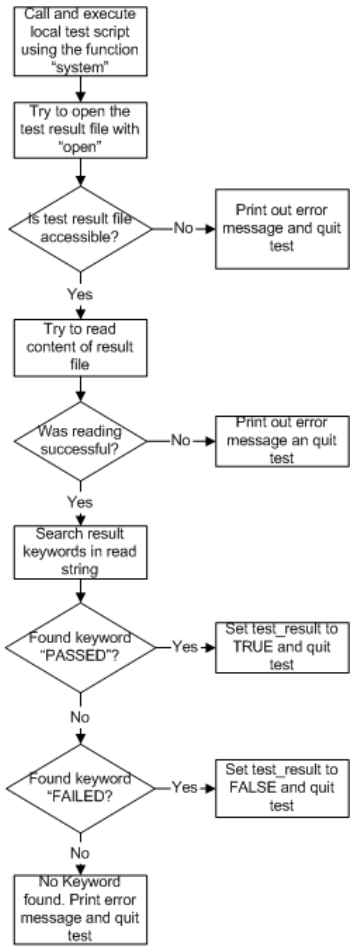


Figure 37: Flow chart of the script test result detection

9. Merging of tester and test item

The following sub-chapters will deal with the problems occurred during the consolidation. From the beginning of this thesis it was expected that this part of development will cause problems which need to be fine-tuned and/or solved. Due to the circumstance of the used alternative hardware (see chapter XYZ INTRODUCTION on page ZYX), several more problems and workarounds occur during the consolidation. As far as possible it must be taken into account that the workarounds are functional on the alternative devices.

9.1. Analysis and solution of accrued problems

9.1.1. Communication

The communication between both devices is based on an already finished and proved communication protocol. It stands for one of the major parts of the development, because the complete environment is based on this.

Originally it was expected that the implementation of this protocol will only cause minor problems concerning the task stack size and minimal timing difficulties. During the consolidation it turned out that the communication protocol is based on a half-duplex communication interface. This means that a member of the communication can either read or write but is not able to do both coevally. In general, a *RS 485* interface is built in full-duplex with five wires, of which two represent plus and minus either for transmit and receive and the fifth line represents the interface *GND* signal. The mode-reducing is effected by an special component. It is built to normally just read from the interface and only to be enabled to write by setting the *ENABLE* signal. The special component with its internal connection is shown in figure 38.

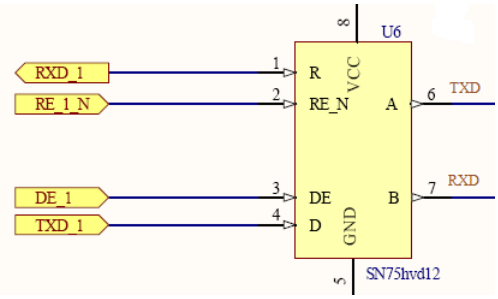


Figure 38: RS 485 full-to-half-duplex mode changer

In the considerations it was expected that the test item also contains two half-duplex *RS 485* interfaces. During the first communication attempt this mistake was found because the protocol was not able to establish a connection. During the debugging phase it turned out that the testee was either able to send or to read. But the normal protocol functionality includes confirmation as well as pure data packages which comes out to the fact that a normal operation is only possible with a two-way communication. Even without confirmation data transfer in both directions is necessary because the test item needs to send test results back to the test device.

There are five possible solution statements for the communication problems, which more or less interfere with the hard- and/or software of both devices. Firstly it is possible to change the testers *RS 485* interface to full-duplex mode as well. On first sight this seems to be the easiest solution. Beside of the complex patchwork on the test device *PCB* this possibility also enforces changes to the interface driver software as well. Another reason against this option is the fact that the *PCB* changes are not allowed to be done on the alternative hardware. Because of that, no further development would be possible as long as the tester device becomes available, which costs a lot of time. Thus this possibility is no option. The second possible solution would be the modification of the full-duplex interface on the test item. In fact this option requires at first the patching and afterwards the depatching of the *RS 485* connection and also changes to the Linux kernel system, which makes this solution even more unhandy than the first one. A third solution is an additional piece of hardware which is connected to the *RS 485* interface of the test item and reduces the transfer to half-duplex mode. Although this option does not require any patchwork it costs develop- as well as assemble time and enforces changes to the Linux kernel as well as the second option. As a forth option it is possible to change the communication interface. Basically a *RS 485* interface is driven by an *UART* which means that any other *UART* connection may be used instead of the *RS 485* connection. The test item contains one more interface driven by an *UART*, which is the *RS 232* serial connection. Unfortunately this connection is already used by the operating system as standard input and output for commands and messages. With the use of this connection for the communication it will not be possible any more to interact manually with the Linux operating system. Also it would not be possible to print out messages to a terminal. Lastly is possible to modify the protocol software on the test device to use both available *RS 485* interfaces to create a full-duplex transfer. The advantage of this method is that no patchwork is required and that this solution is runnable both on the original test device as well as on its alternatives. Furthermore the required software changes are minimal because in the end only the central read and write calls must be modified to work with the right inter-

face. Fortunately this both calls are each only once used in the protocol and are not able to collide. In order to be able to modify the interface number used to write and read, the protocol initialisation is extended with arguments to set these configuration manually.

9.1.2. test sequences

9.2. Adjustment of applications

Several parts of the software require a fine-adjustment or special additions in order to create a fully applicable test environment. This mostly concerns the communication between both devices as well as necessary wait-states on certain code locations.

9.2.1. Remote access

The communication protocol is a circular system. This means that a call from outside is usually not executed immediately but on the next cycle. This certainty causes a natural delay which must be taken into account if a two-way data transfer is requested. The easiest method to handle this problem is a blocked programming based on semaphores.

Table 6: Flow example with and without semaphore blocking

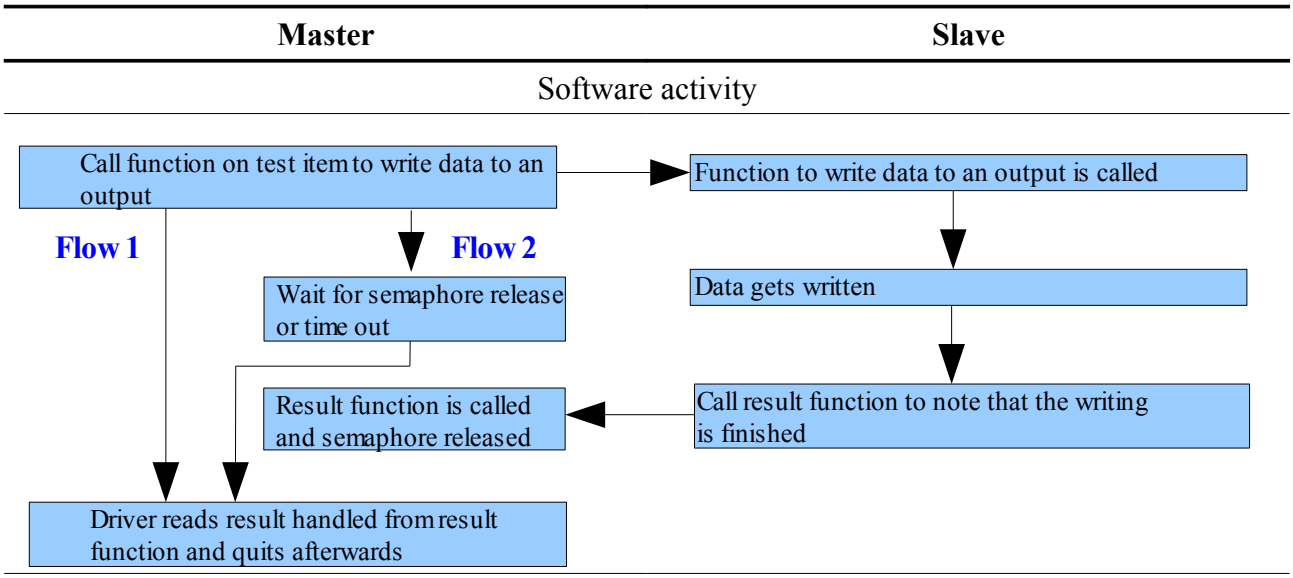


Table 6 shows the main difference between a blocked and a non-blocked two-way transfer. The non-blocked method (Flow 1) automatically reads the result created by the result function right after the remote call. The remote device is not able to execute the requested command and to call the result function within this short time. If the result function was already used before, the previous result is still active and will be used instead of the new requested one. In worst case the result is not initialised and contains meaningless information. This circumstance leads to a undesired software behaviour which is hard to debug. The blocked method (Flow 2) is extended with an semaphore which is requested right after the remote function call. The semaphore is already taken and will be released as last statement of the result function. This approach assures that the remote function was executed and finished and the result was generated. A further advantage of this method is the prevention of deadly time-outs. The request for a semaphore is able to be bound to a maximum time. In case of a fault the semaphore will not be released within the specified time and the request returns an error. The detailed semaphore approach is shown in figure 39. The mentioned three seconds in this figure are a sample value. During the merge of the devices it will be necessary to improve such values. These timings do not have to be very hazardous because the test environment has no claims to act as a high performance system. In general it is expectable that the communication is workable. The time-out represents the worst case and should only be entered if the collapse of the regular communication is definitely assured. Every time-out will automatically cause a negative test result. It must be taken into account that this in turn enforces a manual check-up which takes a lot more time then a generous calculated time-out.

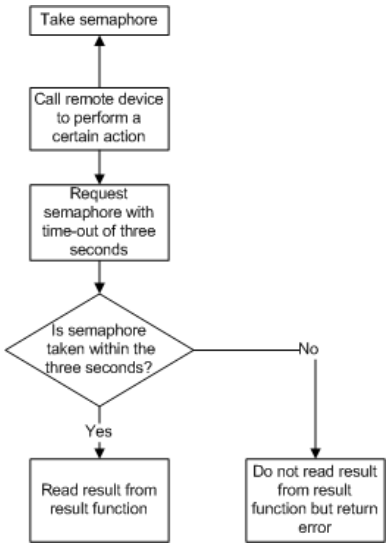


Figure 39: Flow chart of the block method

9.2.2. General wait-states

The *FreeRTOS* provides a function to hold a task for a certain time (see chapter XYZ FREERTOS on page ZYX). In several parts of the software, a so-called wait-state increases the reliability and functionality.

One of the most important wait-states is implemented to the command-line interpreter. The menu structure is designed that way to wait for incoming data the whole time. Because of that, the read-

statement is written into a loop which is only quit if a command has been inserted. Without a wait-state within here, the operating system would spend all calculation time of the processor only for this loop. This in turn implies that no calculation time is left for the circular communication protocol or other tasks running in the background. Code 8 shows the input loop mechanism. The wait-state holds the loop for ten microseconds, which frees enough calculation time for all other tasks. In turn there is no risk to miss an incoming character because no human is able to type with that speed.

```
UINT8 buffer;                                   /* Contains the momentary received character  */
BOOLEAN receive;                                /* Indicates whether a character is available */

do                                              /* Do-while loop for String input             */
    {
    if ((receive = serGet(MenuPort, &buffer) == TRUE))
        {
        /* Key detection algorithms                                                           */
        }
    vTaskDelay (10);                            /* Share Calculation Time to FreeRTOS         */
    } while (buffer != 13);                     /* Read input until ENTER-Key                 */
```

Code 8: Menu input loop with implemented wait-state

Apart from the command-line interpreter, wait-states are usually used to make assure that a previous command was executed. This is for instance used in the analogue connectivity sequences; if an in- or output is switched between voltage and current it is necessary to wait a certain time until the mode is changed. It is not possible to get any feedback to make sure that the mode-change is finished. The same holds for some more cases in which it is not possible to clamp the status of a certain command. Usually a wait-state represents a less complex method to assure the completion of a previous function. The hold-times must be adjusted during the merge. In general it is possible to use more generous values, which in turn may be unhandy if the corresponding code sequence is used more often.

10. Development of the test environment control software

The test environment developed within this paper will be used by several employees. Because of this, it is one ambition of this thesis to create an as far as possible easy-to-use application. Although the command-line interpreter allows interaction to control the environment, it needs a certain time get used to the usage. In order to simplify the control, a window application will be created.

The development of a window application for *Windows* operating system is possible in several

programming languages and environments. Because of company-internal experience the application will be developed in *C#* within the *Visual Studio* software.

Basically the *C#* code is separable to two different purposes of which the first controls the window behaviour and the second one the conventional, executable application. *C#* is a object-orientated language and is so unlike the software developed in *ANSI C* able to remain within the application without the need of an endless loop. Furthermore *C#* is able to control multiple tasks and instances which allows among others the run of an own time-out task running parallel to the common application.

The window application uses a *COM* port of the PC on which it is executed to communicate with the tester device. Basically the control mechanism consists of several function calls, which can also directly be handled to the command line. For the other direction a interpreter must be created which is able to filter keywords sent from the tester device, for instance test results or error messages.

The application allows an dynamic automation in which test sequences can be en- or disabled. Also all tests can be called individually. Furthermore the application is able to show and calculate test statistics and save the test results in log files.

Due to the circumstance mentioned in the Introduction (see page XYZ), the control application can not be completed within this paper. In order to create and show a general functionality, the digital, analogue and relay connectivity tests are controllable. The application surface is displayed in figure 40. As shown, each available test is either executable individually or within a series with all activated tests. The upper right area contains detailed test-flow options. The lower right side shows the test statistics.

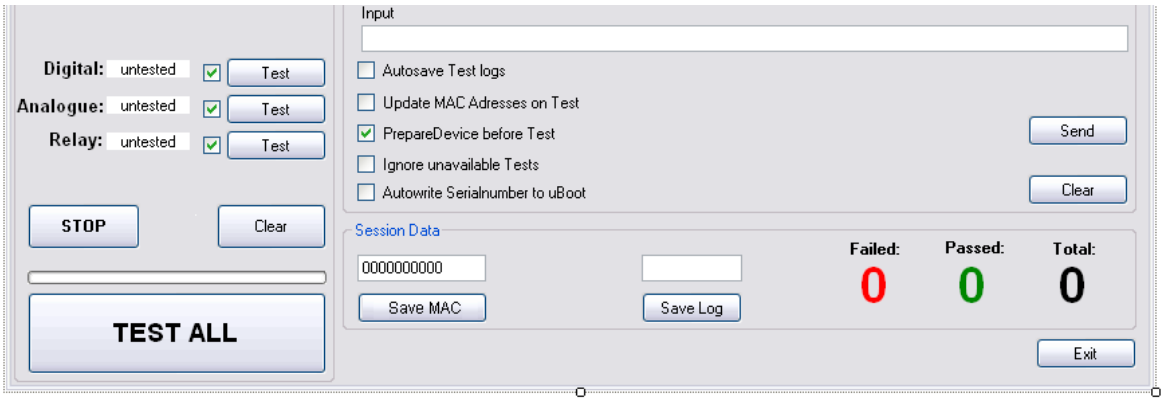


Figure 40: control application surface

11. Test application run cycle

12. Conclusion

This thesis can generally be divided to three different parts, which are firstly the basic principles, secondly the hard- and software development and thirdly the merge of the devices.

The first part is solely based on theoretical considerations. Apart from the introduction of the test item, this section deals mainly with the analysis of the testee and the future environment. Furthermore it figures out further steps and lists the possible test schemes. Lastly it faces the possible, technical test methods with their economical backgrounds and assigns the further approach. The second part is in most parts based only on this. Although several parts of the hardware design are taken from already existing hardware schematics, the combination of all parts composes an absolute new device. The same holds for the software development. The last section represents the merge, in which both, hard- as well as software, are checked and proved for the first time.

It was one ambition of this thesis to show, apart from the actual work, the general approach and the management of a common time-limited, embedded-system development project applied within a company. Due to circumstances this ambition was fulfilled more than expected, because it laid open several critical spots of the management. Firstly, it is always complicated to apply a project of that size within such a small time-frame. Generally this project was only possible to be finished somehow because it was based on several already existing resources. Otherwise the research of the hardware schematics would have taken much too much time. The same holds for the software development as well. The development of all driver-specific functions would have been a major part as well as the memory set-up. The second issue is the common approach to develop hard- and software simultaneously. This is mainly done in order to save development time but leads to the problem of only theoretically developed software. Although it is called “good style” to create code following detailed previous considerations, this method will cause more problems the more software is developed solely theoretical. The main reason for this is the fact that the huge number of changes at once to the software hardly allow a starting point for debugging. In addition to that, some parts of embedded systems do have a kind of “life of its own”, which means that some special software constructions will not work although they do in theory. Several parts of the software within this paper were developed theoretically, especially the communication driver and most functions of the test routines. Other sections like the command-line interpreter were created with the “trial and error” method, because the structure and the functionality was created not before but during the develop-

ment process and also depended on the behaviour of added parts. Lastly it was unintentionally shown that a project of this size and such a curt time-frame in most parts depends on the assumption that no problems will occur. The time management does not allow any deviation and cannot be fulfilled if one part causes problems.

13. Literature

13.1. Printed sources

Kernighan, Brian W., Ritchie, Dennis M.(1988): The C Programming Language, London: Prentice-Hall International (UK) Limited

Prinz, P, Kirch-Prinz, U.: C – Kurz & Gut, Köln: O'Reilly Verlag

Barrett, Daniel J.(2004): Linux – Kurz & Gut, Köln: O'Reilly Verlag

Albahari, J., Albahari, B.: C# 3.0 – Kurz & Gut, Köln: O'Reilly Verlag

13.2. On-line sources

Microsoft: Visual Studio 2008 Express Editions

http://www.microsoft.com/Express/

Fischer, Michael: YAGARTO – Yet another GNU ARM toolchain

http://www.yagarto.de/index.html

Ho, D., DV, Harry, Lorenz, J., Severance, Ch., Le Gall, St.: Notepad++

http://notepad-plus.sourceforge.net/de/site.htm

Tathan, S., Owen, D., Harris, B., Nevins, J.: PuttY: a free telnet/ssh client

http://www.chiark.greenend.org.uk/~sgtatham/putty/

Barry, Richard: FreeRTOS – A free RealTime Operating System

http://www.freertos.org

ANSI/VT100 Terminal Control Escape Sequences

http://www.termsys.demon.co.uk/vtansi.htm

Chann, Elm: FAT File System Module

http://elm-chan.org/fsw/ff/00index_e.html

Microsoft: FAT32 File System Specification http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx

O'Reilly Media: Linux Command Directory: stty

http://www.oreillynet.com/linux/cmd/cmd.csp?path=s/stty

14. Apendix

Due to the size and length of the software as well as the hardware development projects, the complete attachments are given on a digital medium.

Most of the code files contain a special file header in which the date of creation as well as the author or authors are mentioned. The author of this thesis can be identified by the initials *Mmi*. Shown below is the folder structure of the digital medium.

```
DVD:.
├───Dokument
│   ├───pictures
│   │   ├───Code
│   │   ├───PCB
│   │   ├───SCH
│   │   ├───SW
│   │   └───testee
│   └───visio
├───Quantum Controller
│   └───SW
│       ├───script
│       └───testapplication
└───Tester
    ├───HW
    │   ├───BOM
    │   ├───Gerber
    │   ├───PCB
    │   └───SCH
    └───SW
        ├───Applications
        │   ├───groupings
        │   ├───menu
        │   ├───remote
        │   └───testfiles
        ├───BootLoader
        ├───inc
        └───lib
            ├───BusProtocol
            ├───Drivers
            ├───FreeRTOS
            │   └───Source
            │       ├───include
            │       └───portable
            │           ├───GCC
            │           │   └───ARM7_LPC23xx
            │           └───MemMang
            └───MmcFilesystem
```