NTNU – Trondheim
Norwegian University of
Science and Technology

Design, Implementation and Testing of Low-level Layers of the PDCP for the AVR Platform

Andrzej Zamojski

Master of Science in Engineering Cybernetics

Submission date:	June 2012
Supervisor:	Øyvind Stavdahl, ITK
Co-supervisor:	Kamil Grabowski, DMCS at TU of Lodz (POLAND)

Norwegian University of Science and Technology
Department of Engineering Cybernetics

NTNU
Norwegian University of Science and Technology

Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Engineering Cybernetics

Master's Thesis

Student's name:	Andrzej Zamojski
Field:	Engineering Cybernetics
Title (Norwegian):	Design, realisering og testing av nedre lag i PDCP på AVR-platformen
Title (English):	Design, Implementation and Testing of Low-level Layers of the PDCP for the AVR Platform

Description:
The prosthetics industry is dominated by companies that primarily use their own proprietary standards. This is an increasing hindrance to interoperability of components from different vendors.
Following an initiative by NTNU in 2005, international actors have started the development of a standardized bus interface that will enable such interoperability. The protocol is mainly developed at the University of New Brunswick, Canada, and currently only implemented on the PIC (Microchip) controller platform. This implementation is proprietary. We now want to implement the protocol on the Atmel AVR platform and release the code so that all vendors and researchers can use it. The software will be implemented for the ITK AVR controller card.
This assignment primarily concerns the lower levels of the protocol stack, while the upper layers will be developed in a parallel activity.

1. Give a concise description the CAN bus standard in relation to the ISO OSI model.
2. Familiarize yourself with the present PDCP draft standard. Provide a brief overview of its main features related to the OSI model. Point out properties that relate explicitly or implicitly to hardware resources and that is relevant for the implementation's portability to other HW platforms primarily based on the AVR controller family and secondarily on other controller architectures.
3. Suggest a suitable software architecture for the implementation of the protocol stack, and provide a detailed description of the interface between the lower and the higher levels of the software stack, respectively. This task should be carried out in close cooperation with the student responsible for the higher levels to ensure compatibility between the two.
4. Perform a detailed design and implementation of the lower level software on the given hardware.
5. As far as the time permits, test the results with the higher-level AVR implementation and with a node based on the Canadian implementation.

Advisor(s): Associate Professor Øyvind Stavdahl, NTNU

Trondheim, January 2012

Øyvind Stavdahl
Supervisor

Abstract

The industry engaged in prostheses production is dominated by companies that offer products using their own propriety standards. This results in either impossible or significantly impeded cooperation between modules from different manufacturers within one device. Looking for solutions to this problem in 2005 engineers started working on standardization of communication interface. Outcome of the University of New Brunswick (UNB) Hand Project, founded by Atlantic Innovation Fund (AIF), is still improved interface *Prosthetic Device Communication Protocol* for internal communication of prosthesis hand modules.

This paper has been devoted to the matter of design and implementation of the lower layers of PDCP on AVR Platform, hugely popular in systems of prostheses. An important aspect was to maximize software portability between different models of AVR family microcontrollers and as far as possible between microcontrollers from different manufactures. Software should be well documented and understandable for engineers enabling further development.

Hardware layer used in this project was based on AT90USB1287 Atmel microcontroller, external CAN controller MCP2515 and CAN transceiver MCP2551 (both manufactured by Microchip). A part of the project was to design Printed Circuit Board giving a visualization of the software portability.

The software was designed in close cooperation with the student responsible for the higher layers of the interface, therefore corresponding interface had to be designed. Lower layers of PDCP are based on interrupt generated by the CAN controller chip. Thanks to this solution maximum capacity was ensured while providing CPU time for upper layers of interface and application-specific tasks.

Streszczenie (Abstract in Polish)

Gałąź przemysłu zajmująca się produkcją protez jest zdominowana przez firmy, które w oferowanych produktach stosują własne, prawnie zastrzeżone, standardy. Wskutek tego, współpraca modułów różnych producentów w obrębie jednej protezy często jest niemożliwa lub znacząco utrudniona. Szukając rozwiązań tego problemu inżynierowie w 2005 roku rozpoczęli prace nad standaryzacją interfejsu komunikacyjnego. Owocem projektu *AIF UNB Hand Project* (Atlantic Innovation Fund University of New Brunswick) koordynowanego przez wspomniany uniwersytet w Kanadzie jest wciąż udoskonalany interfejs PDCP (z ang. *Prosthetic Device Communication Protocol*), służący komunikacji modułów wewnętrznych protez ręki.

Niniejsza praca została poświęcona projektowi i implementacji niższych warstw w/w protokołu w oparciu o kontrolery firmy Atmel cieszące się dużą popularnością w układach protez. Ważnym aspektem była maksymalizacja przenośności kodu między różnymi modelami mikrokontrolerów z rodziny AVR i na ile to możliwe między mikrokontrolerami różnych producentów. Opracowane oprogramowanie powinno zostać szczegółowo udokumentowane aby umożliwić dalszy rozwój interfejsu.

Wykorzystywana w projekcie warstwa sprzętowa została oparta na mikrokontrolerze AT90USB1287 firmy Atmel, układzie kontrolera magistrali CAN MCP2515 oraz kontrolera warstwy fizycznej oznaczonego symbolem MCP2551, których producentem jest firma Microchip. Częścią projektu było wykonanie płytki drukowanej, za pomocą której zaprezentowana została przenośność oprogramowania, dzięki zastosowaniu innego mikrokontrolera.

Oprogramowanie zostało zaprojektowane w ścisłej współpracy z innym studentem (Andreasem Nordalem) odpowiedzialnym za wyższą warstwę interfejsu, w oparciu o przerwania generowane przez układ kontrolera CAN. Dzięki takiemu rozwiązaniu została uzyskana maksymalna przepustowość interfejsu przy maksymalizacji czasu pracy procesora dla wyższych warstw interfejsu oraz aplikacji specyficznych dla poszczególnych modułów systemu.

Preface

This thesis is submitted in fulfillment of the degree of Master of Science at the Norwegian University of Science and Technology, Department of Engineering Cybernetics.

Working on this project occurred a really challenging experience. I hope that end effect will be useful for students and cybernetic engineers in the future work on the standardized communication protocol for prostheses. As I have heard from a physically challenged man: “*Engineers, you are our future*”. I believe that this sentence somehow shows us how important our work is in fact.

Taking the opportunity I would like to express my gratitude to Øyvind Stavdahl for being my enthusiastic and excellent supervisor, as well as Kamil Grabowski for being my very helpful co-supervisor from my home university in Łódź, Poland.

Table of Contents

1 Introduction

The best way to experience how important in human life are hands is to ask physically challenged people a question about comfort of their life. Only a few parts of human body are as complex and important as our hands. Handling everyday tasks is feasible thanks to splendid interplay of the nervous system, tendons, over 20 bones, muscles and joints. Despite technological progress recreation of hand functions is still a great challenge, both medical and mechanical, electronic and control.

One of the most important aspect that this master project concerns is prostheses modularity and communication between its modules. Many previous and current designs of the commercial arm prostheses do not support the modular approach, which can influence degree of adaptation to user's needs and disability. Frequently whole device consists of 2 modules: the integrated palm and EMG electrode module. The division in case of any part failure leads to the necessity of whole module replacement. Therefore over years researchers have still been trying to customize the prosthesis to user's needs by developing the idea of modularity. Some of past project are worth noticing.

The background for many subsequent solutions was SVEN hand project developed during the 1970s in Sweden. Apart from a mechanical aspect the most significant contribution was control system, which used EMG recordings from six electrodes located on the residual limb while the patient performed basic hand movements like finger flexion or wrist extension. EMG signals could then be used to control the prosthesis by pattern-recognition technique implemented in analog EMG processor system. Another solution was Edinburgh Arm System that introduced a new mechanical solution, but also was controlled with conventional analog electronic controllers.

A name that is often mentioned in discussion about upper-limb prostheses is Otto Bock. In 2000 the group of 2 people (*R. Obermaisser, A. Kanitsar*) together with O. Bock presented implementation of TTP protocol for master-slave application, which could be treated as a basis for the modular system.

Time-Triggered Architecture (TTP) is a real-time protocol using Time-Division Multiple Access (TDMA) scheme to provide collision-free transmission. Data communication is organized in TDMA rounds, which layout is defined a priori and known to all nodes in the system. Every round is divided into time slots associated with individual system nodes that are obliged to send frames in every round. A few rounds (usually with different messages inside) are combined into one cluster cycle, which is repeated over time. Data protection is provided by CRC sum. To proper operation system needs clock synchronization, which is done by each node by measuring the difference between known expected and observed arrival time of a correct message to compute the difference between sender`s and receiver`s clocks. This information about time shift is indispensible to keep node clock in synchrony with time frame in cluster. Previously mentioned authors have attempted to implement this protocol to the system consisting of one master and up to 7 slave devices, which was first industrial application of TTP/A. To fulfill time restrictions required by motor control some more sophisticated designs of rounds and cluster had to be performed. That project proved that TTP/A protocol is suitable for time-critical industrial applications providing efficient data transmission and error handling. More details about the aforementioned implementation can be found in the literature [2].

Another important step in upper-limb prosthetics was *"Totally Modular Prosthetics Arm with high Workability" (ToMPAW)* consortium founded by the European Union in 2007. The major concern in the provision of each limb prosthesis is to design and produce a solution that is most appropriate to the user needs. Such properties as level of losses, the strength and the needs and abilities of the user must be taken into account. All these factors complicates prosthesis design and makes prosthesis as individual as its wearer. Therefore, to simplify this process, modular approach should be adopted, which involves necessity of interchangeability and interoperability between modules provided by commercial suppliers. With the development of microprocessor-based controllers researchers received tool that enabled selecting suitable control strategy and enabling fast, secure and easy way to test it on patients. ToMPAW consortium addressed all these problems presented above.

The result of that project was total arm system providing function separation (modularity) and simplified way of upgrades and modifications. The distributed system ensured that single failure did not stop the whole device. Moreover, the system could be assembled from the set of standardized components (both from the electronic and mechanical points of view), which was undoubtedly an advantage. The design of control system took into account reliability and modularity, which was made by a decreasing amount of interconnections (*Fieldbus* has built-in network communication support and data protection mechanism). The design of ToMPAW system enables simple adding of additional joints and functional units with decreased amount of changes in the system.

As can be easily noticed, the development in prosthetics technology goes in the direction of increased modularity, number of motors and controlled joints, which involves more sophisticated control algorithms and communication requirements. To meet these expectations researchers more often think about standardization and first results of this process are already visible.

Under the name of consortium, which has long sought to standardize prosthetic control system, *the Standard Control Interface for Prosthetics (SCIP)*, Yves Losier from

University of New Brunswick, Canada, has posted a draft of CAN-based standard for *AIF UNB Hand Project*. After some time the designed interface has gained a name: *Prosthetics Device Communication Protocol (PDCP).* To date (as far as it is known to author of this thesis) only one implementation on PIC (Microchip) controller platform was performed. The designed draft protocol is based on CAN messages transmission. This standard, established for automotive communication, increases modularity possibilities of prosthesis with protection mechanism against data corruption and simple electronic structure. It should be noted that with development of prosthetics more and more sophisticated prostheses are available for researchers. The effective control of joints within the device is both clinical goal as well as the challenging research, because of an increasing number of physical connections and data flow. The designed interface should therefore facilitate this process providing fast, reliable and simple transmission medium.

Researchers do hope that manufacturer or researcher group will in the future benefit from published standard and related API or source code to ensure compatibility with any arm joint, hand or intrinsic hand joint.

2 Low-level protocols

There are numerous commercial off-the-shelf low-level protocols providing basic mechanisms for transmission and robustness. Designing completely new protocol from scratch could not make sense when existing protocols are proven and give good results in many industrial applications. Instead of that a complete protocol for prostheses could be built on top of one of technologies like I^2C, CAN or ZigBee. However, some features specified below should be achieved.

Reduced wiring – simpler connections and less interference into external environment, also reduced production cost and failure rate;

Availability – immediate availability of hardware and software components. Standardized interface should be made out of components widely available on the market;

Reduced risk – technology proven in many previous applications. It reduces probable problems at start-up;

Reduced complexity – only higher-level functionality should be included in protocol specification. The majority of hardware issues could be acquired by low-level protocol software.

High capacity of the system – prostheses systems desire dense data exchange, which should be provided firstly by low-level protocol, secondly by software basing on that.

The comparison of the mentioned technologies was presented in Table 1.

Table 1 Comparison of commercial low-level protocols

Property	CAN	I2C	ZigBee
Differential transmission	YES	NO	NO
Wiring	1-2 wires	2 wires	wireless
Bit rate	0-1 Mbps	0-3.4 Mbps	20-250 kbps
Range	40 m	Not defined	10-70 m
Power consumption	10 mA (transmission)	Extremely low	To 30 mA (transmission)
Required external components	Transceiver chip	None	Transceiver chip

From above interesting conclusions can be drawn. Every technology provides advantages. Some of them like wireless communication seem to be very good from prosthesis point of view. However, it follows slower data transmission and higher sensitivity to interference from other devices. The tendency to wiring reducing leads to focusing on CAN bus, which provides differential transmission using only two wires and relatively high bit rate. A range aspect in case of prosthesis does not play a significant role because of rather small distances between communication nodes. There is no doubt that I^{2}C protocol wins in the category of required external components. Almost every microcontroller provides support for this protocol, decreasing cost of production and size of electronic circuits.

Taking into account all above factors for the purpose of the PDCP CAN interface was chosen.

3 CAN interface

Controller Area Network (CAN) was designed by Robert Bosch in the mid-1980s for automotive applications as a response to the increasing need for more reliable, safe and fuel-efficient automobiles while decreasing complexity and wiring weights. CAN protocol gained widespread popularity in many areas of industry like medical engineering, automotive electronics, engine control units, sensors or mobiles machines.

3.1 Physical layer

Physical layer of CAN interface was presented in Fig. 1.

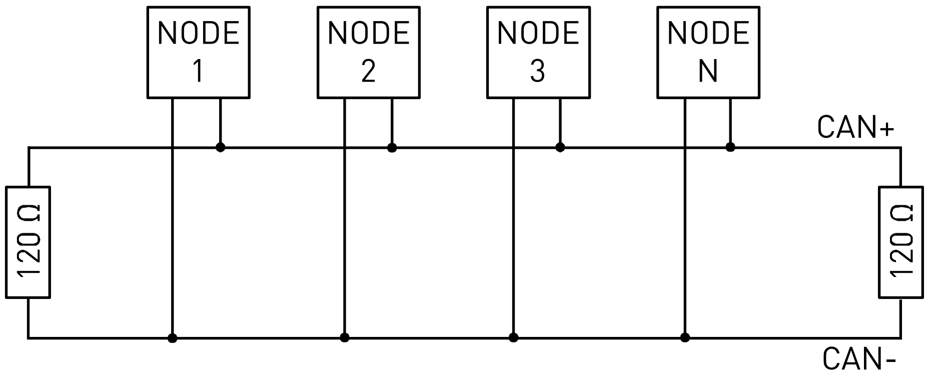


Fig. 1 CAN interface - physical layer

CAN standard uses differential transmission on 2 wires (signed CAN+ and CAN-). Additionally, all interface in the bus should have the same ground potential therefore at least ground line (GND) should be also provided. Moreover, noise immunity is achieved by maintaining impedance on the bus with low-value resistors 120 Ω at each end of the bus.

To reduce susceptibility to interference and minimize RF emission CAN bus wires should be carried in twisted pair wires – this aspect might be very important for reliable prosthesis operation in a noisy environment (mobile phones, computers and all other electric and electronic equipment).

3.2 Data frame

CAN protocol is a message-based protocol. Every node in the system receives every message and compares arbitration field with node identifier, filters and mask determines if message should be discarded or kept to be processed. CAN is often described as CSMA/CD protocol. Carrier Sense Multiple Access means that every node within the system must monitor the system bus for periods of no activity before trying to send any message. Moreover, multiple access means that every node on the bus has the same opportunity of message transfer in case of bus non-activity.

Structure of CAN Message Data Frame was presented in Table 2.

Table 2 CAN Message Data Frame

SOF	**Arbitration Field**	**Control Field**	**Data Field**	**CRC Field**	**Ack Field**	**EOF**
1b	12b or 32b	6b	0-8B	16b	2b	7b

Data frame consists of fields that provide information about transferred message and enable correct arbitration. *Start of Frame* marks the beginning of data frame by single dominant bit. Next time slot is *Arbitration Field* consisting of 12 or 32 bits depending on whether *Standard* or *Extended Frames* (Standard or Extended Identifier Field) are being utilized (for purpose of this project only Standard Frames are used). The value of arbitration field defines the priority of the message. Arbitration in CAN protocol uses logic bit 0 as a dominant bit state that wins over recessive bit state. This implies the lower the value in the *Message Identifier*, the higher the priority of the message. In case of arbitration node trying to transmit message with lower priority it is forced to wait for the next period of no activity on the bus. Thanks to this solution risk of data loss is significantly reduced. Next time slot takes *Control Field* that contains knowledge about the size of *Data Field* (from 0 up to 8 bytes). The *CRC Field* consists of a 15-bit CRC field and delimiter. This field is used by the recipient to determine if transmission errors have occurred. The *Acknowledge Field* is utilized to indicate correctness of message reception – the recipient after correct message reception puts a dominant bit on the bus in ACK slot time. End of Frame is marked with 7 recessive bits.

3.3 Layer model

CAN is a serial communication protocol that implements most of the lower two layers of ISO Open Systems Interconnection (OSI) Network Layering Reference Model was presented in Table 3 and Table 4.

Table 3 ISO OSI model

Layer	Description
Application (7)	Network process for application
Presentation (6)	Data representation, encryption and decryption
Session (5)	Interhost communication
Transport (4)	Reliability, flow control
Network (3)	Logical addressing
Data link (2)	Physical addressing
Physical (1)	Electrical and physical specification

This model gives the prescription of characterizing and standardizing the functions of communication systems in terms of abstraction layers. Functionalities, which play a similar role in the system, are grouped together into logical layers. Particular layers serve one another being responsible for individual tasks within the system in the order of abstraction.

Table 4 CAN specification in relation to ISO OSI model (literature[6])

Layer	Detailed description
Data link	Logical Link Control (LLC): • Acceptance Filtering; • Overload Notification; • Recovery Management; Medium Access Control (MAC): • Data Encapsulation/Decapsulation; • Frame Coding (Stuffing/Destuffing); • Error Detection; • Serialization/Deserialization;
Physical	Physical Signaling (PLC) • Bit Encoding/Decoding • Bit Timing/Synchronization Physical Medium Attachment (PMA) • Driver/Receiver Characteristics Medium Dependant Interface (MDI) • Connectors

To optimize the communication protocol on multiple media and increase possibilities of adaptation to certain conditions BOSCH company did not specified the CAN protocol in a very strict way. *International Standard Organization with Society of Automotive Engineers* has defined protocols based on CAN containing specified features which should be fulfilled like: differential signal transmission and speed of transmission (up to 1Mbps). Moreover, issues of coding and timing bound with synchronization were included there. All of them are related to the physical layer of the OSI model. The CAN specification also contains issues associated with serialization, error detection, frame coding and data capsulation which are definitely related to the data link layer of ISO/OSI model. Message filtering was also described within the CAN specification. The rest of the layers of the ISO/OSI model are left to be implemented by the software designer – this might include distribution of node id`s, determination of messages structure and/or providing error handling routines.

4 Prosthetic Device Communication Protocol

The aim of this chapter is to give some overview of Prosthetics Device Communication Protocol features that is based on CAN bus described in the previous section. Firstly, some specification of identifier field were presented. Secondly, message exchange system with functions description were outlined.

4.1 Identifier field

One of the most significant elements of the CAN message frame from the PDCP point of view are the identifier field together with the data field. Message addressing and prioritization is executed thanks to *Standard Identifier Field* that for the purpose of this protocol was divided into 3 subsections. A short description was presented in Fig. 2.

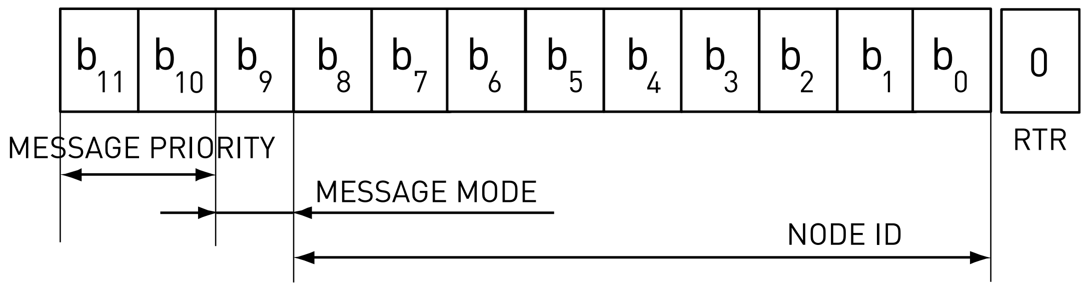


Fig. 2 Division of Arbitration Field for PDCP

Message Priority Field – these 2 bits of Standard Identifier Field are used to assign a priority to an outgoing message. The lower value, the higher priority message has (0 = High Priority, 1 = Normal Priority and 2 = Low Priority). The last value (3) is used by the device while attempting to bind itself to the bus. While more than one device is trying to send message at the same time, arbitration logic will give the bus access to device with higher priority. In case of the same priority level, the message arbitration will decide about control basing on field described below;

Message Mode Field – this field determines a source of message and takes the value of 1, when message originates from Bus Arbitrator (*Bus Arbitrator Message Mode*) or 0, when from some other device (*Standard Message Mode*). The result is the fact that in case of attempt of simultaneous transmissions of two or more messages (with identical *Message Priority Fields*) arbitration logic will give bus control to device with lower value of *Message Mode Field*, so to the message with *Standard Message Mode*. In case of the same priority level and message mode value decision about bus control is made basing on node identifier value;

Node Identifier Field – the value of this field depends on the *Message Mode Field* value. If bus arbitrator is going to send a message, then this field is assigned the NodeId of intended recipient. Otherwise, if the sender is a normal device, then this field is assigned with NodeId of the transmitting device.

It is worth noticing that the capacity of the system reaches value of 255. Node ID value 0 was reserved for broadcast messages within entire bus system. Message filtering is done using masks and filters available to control by internal registers of CAN controller. Therefore, different nodes within one system may work as arbitrators and receive all messages, while others are specified to accept only messages addressed to themselves.

4.2 Protocol functions

Message transmission implemented in the designed Prosthetic Device Communication Protocol is based on request-response message exchange model. The system module, which sends a message, expects to receive a response unless this functionality does not expect a response to be returned (Device Beacon function). PDCP contains 15 function code (with 4 deprecated) so far and code left for future system commands and module-specific commands. Table with function codes was presented in Table 5.

Table 5 List of function codes of PDCP (- deprecated functions)*

Function Code	Function Code Description	Sender	Recipient	ISO/OSI layer
0x01	Bind Device Request	Device	Bus Arb	3,4
0x02	Get Device Info (*)	Bus Arb	Device	*
0x03	Get Device Parameter	Bus Arb	Device	3
0x04	Set Device Parameter	Bus Arb	Device	3
0x05	IntGetDeviceParameter (*)	Device	Bus Arb	*
0x06	IntSetDeviceParameter (*)	Device	Bus Arb	*
0x07	SetNodeId (*)	Bus Arb	Device	*
0x08	Suspend Device	Bus Arb	Device	6,7
0x09	Release Device	Bus Arb	Device	6,7
0x0A	Device Beacon	Dev or Arb	Arb or Dev	5 or 3/4
0x0B	Reset Device	Bus Arb	Device	2,3,4,6,7
0x0C	Configure Get Bulk Data Transfer	Bus Arb	Device	3,4
0x0D	Configure Set Bulk Data Transfer	Bus Arb	Device	3,4
0x0E	Bulk Data Transfer	Device or Arbitrator	Arbitrator or Device	3
0x0F	Update Data Channel	Device	Bus Arb	3

For better understanding of request-response model, some details of one function code were presented in Table 6. For this purpose Bind Device Request function was chosen.

Table 6 Parameters of Bind Device Request function code

Parameter	Value
Function code	0x01
Response function code	0x81
Description	Function is sent just after power-on or reset. The bus arbitrator respond consists of available *NodeId* that has not been allocated to another device within the system. The device is successfully bound with the system if *NodeId* used by the device and received *NodeId* are identical. Otherwise, the device is obliged to send new Bind Device Request using received *NodeId*.
Sender	Any device
Recipient	Bus Arbitrator
Data bytes (request)	7
Data bytes (response)	8

In Table 7 and Table 8 the structure of CAN messages data field for Bind Device Request was presented.

Table 7 Request message format (data field) - Bind Device Request function

BYTE0	BYTE1	BYTE2	BYTE3	BYTE4	BYTE5	BYTE6	BYTE7
0x01	Device Vendor ID		Device Product ID		Device Serial Number		(empty)

Table 8 Response message format (data field) - Bind Device Request function

BYTE0	BYTE1	BYTE2	BYTE3	BYTE4	BYTE5	BYTE6	BYTE7
0x81	NodeId	Device Vendor ID		Device Product ID		Device Serial Number	

In Table 9 and Table 10 one example of binding message transfer was presented. More details and explanation were placed below tables.

Table 9 Bind Device Request - example

Standard Identifier Field							
Priority		**Message Mode**		**Node ID**		**DLC**	
3		0		**6**		7	
BYTE0	**BYTE1**	**BYTE2**	**BYTE3**	**BYTE4**	**BYTE5**	**BYTE6**	**BYTE7**
0x01	0x03	0x00	0xA9	0x00	0x17	0x00	-

Table 10 Bind Device Request response - example

Standard Identifier Field							
Priority		**Message Mode**		**Node ID**		**DLC**	
1		0		1		8	
BYTE0	**BYTE1**	**BYTE2**	**BYTE3**	**BYTE4**	**BYTE5**	**BYTE6**	**BYTE7**
0x81	0x06	0x03	0x00	0xA9	0x00	0x17	0x00

To illustrate mechanism of message exchange between 2 modules of prosthetic system *Bind Device Request* function has been chosen. As can be easily noticed, response code for every code function is logical sum of function code and value 0x80. In case of binding request function, data field consists of device vendor id, device product id and device serial number. Inside of the identifier and beside of priority and message mode, one of the most important part is *NodeId* (in this example bold value 0x06). If this *NodeId* is not assigned to any other device within the system, the bus arbitrator in response message sends back this particular value of *NodeID* (in this case 0x06). Otherwise, arbitrator sends back another *NodeId* and the procedure has to be repeated. It is worth noting that this functionality may be assigned both to link and transport layer of the ISO/OSI model because it concerns matters of logical addressing and flow control as well.

The last column of the table presenting description of the functions contains the attempt to assign particular functionalities into layers of the ISO/OSI model. So far there has been little discussion about it, probably because of the early stage of development and first software implementations of designed interface. Such division might help in protocol standardization in the future.

Protocol functions implement mainly 3 and 4 layer of ISO/OSI model, network and transport layer accordingly. While determining function attributes, an emphasis was placed on feature distribution in ISO/OSI model. However, one has to realize that it is often quite hard, if not impossible, to give strict information which functionalities should be assigned to which model layer. Functions, which are not related to setting parameters within a system node (*Get Device Parameter* or *Set Device Parameter*, *Update Data Channel*), may be assigned to third layer of ISO/OSI model (link layer). The functions that are responsible for more than setting or getting some parameters but also trigger some logical connection between nodes were assigned both to the link layer and the transport layer. This applies mainly to *Bind Device Request* described in the previous section that establishes the connection between system nodes and enables bus configuration. *Device Beacon* is the function that provides control over nodes in the system by checking their connection to the system. If the node does not send Beacon messages to Arbitrator in specified time intervals, might be reset. There has been a big discussion to which layer this function should be assigned. One, which seems to be accurate, is the session layer (interhost communication, session managing between applications, which may be understood as session maintaining between one node and other system nodes). From another point of view this functionality could be ascribed to link and especially the transport layer, as e.g. *Bind Device Request*, because of flow control mechanism maintaining features. Very similar problem refers to functions responsible for device suspending and releasing. Depending on whole application and interface layout, this function may concern higher or lower layers of ISO/OSI model. *Reset Device* function, which follows from the idea of reset, was assigned both to the data link layer (while reset also CAN controller should be reset) and higher model layers – software reconfiguration.

5 PDCP interface layout

This section begins with the description of the Prosthetics Device Communication Protocol division into two parts carried out parallel with another student. Next, the broadest subsection is devoted to software layout of the low level part of the interface, which is the main topic of this work. Finally, hardware aspects and design of PCB board have been outlined.

5.1 HAL – HLL interface

The PDCP implementation was divided into 2 parts called as follows:

- HLL – High Level Interface
- HAL – Hardware Abstraction Layer

<u>Acronyms mentioned above are consequently used in the next part of this dissertation.</u>

High Level Layer contains implementation of binding and interface functions without any consideration about hardware issues. As mentioned above, this software was designed by Andreas Nordal [master thesis on *Design, Implementation and Testing of High-Level Layers of PDCP for AVR,* under publication], but interface between HLL and HAL was carried out with strong cooperation between two designers of the PDCP for AVR implementation.

The interface between HLL and HAL consists of following data structures and functions:

struct can_msg – contains following fields: identifier, number of data bytes of message frame and data bytes;

#define CONFIG_BUS_MODE – pre-processor directive defining node type (ARBITRATOR or DEVICE);

void hal_set_mask (uint8_t id) – sets mask configuration of CAN controller providing message reception conditions;

void hal_set_filter (uint8_t id) – sets filter configuration of CAN controller providing message reception conditions;

void hal_msg_poll () – retries fetching an incoming message left in CAN controller in case of no memory to assign message to pointer for HLL;

struct can_msg* hll_msg_alloc() – reserves unused memory for incoming CAN message. Pointer to *can_msg* structure should be returned to caller (HAL);

void hll_msg_commit (struct can_msg* msg) – invokes HLL`s processing of CAN message referenced by *msg* pointer, function called by HAL after message reception;

struct can_msg* hll_msg_get() – looks for messages to send in HLL and if found, return can_msg structure to caller. Otherwise, returns NULL;

void hll_msg_free (struct can_msg* msg) – marks the memory used for CAN message as unused, additionally after message sending checks whether any message is waiting for reception from CAN controller.

As can be easily noticed, some name convention was used. Explanation was presented in chapter *5.2.11 Name convention*. Function usage was described precisely in chapters treating about message sending and receiving *5.2.6 Messages sending* and *5.2.7 Messages receiving*, respectively.

5.2 Software layout

This chapter precisely describes design and implementation of the Hardware Abstraction Layer pointing out information about factors influencing modularity, portability capabilities between microcontrollers from AVR family and as far as possible between microcontrollers from different microcontroller vendors. Firstly, an introduction and hardware resources used in the project have been described. Secondly, aspects of message sending and receiving have been outlined. Then, other functionalities implemented within the HAL have been described and at the end, the project file and code structure have been presented.

5.2.1 Hardware architecture

In Fig. 3 NIMRON board, used for protocol implementation, was presented.

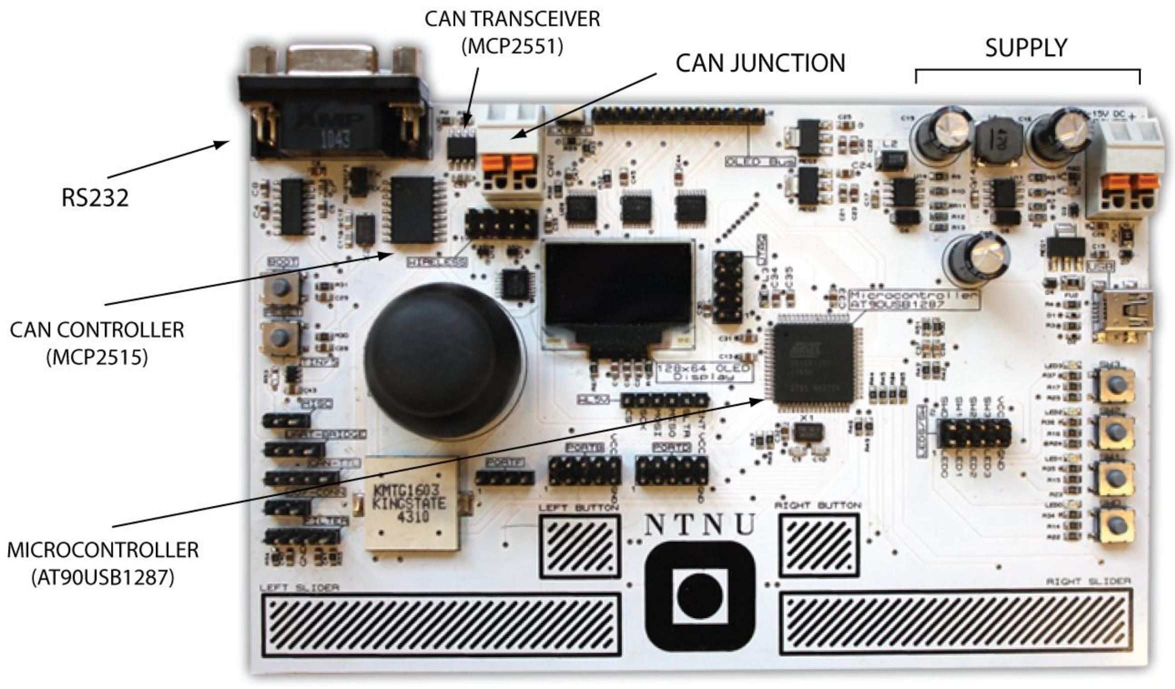


Fig. 3 NIMRON board layout

Printed Circuit Board from Fig. 3 was designed by *Ole Johnny Borgersen* and *Marius Lind Volstad* as *USB Multifunction Board – NIMRON*. Now it is used in programming courses by students of Cybernetic Engineering at NTNU. Thanks to many external peripherals it is very useful providing platform for code startup and programming learning. From PDCP point of view hardware listed below and placed on the board is essential.

Microcontroller: AT90USB1287 – High Performance, Low-Power AVR 8-bit Microcontroller with 128kB of ISP Flash and USB controller;

CAN controller: MCP2515 (Microchip) – Stand-Alone CAN controller with SPI Interface;

CAN transmitter: MCP2551 (Microchip) – High-Speed CAN transceiver.

Undoubted benefit of using this particular board was almost complete platform for code testing. Using wires connection firstly between microcontroller and CAN transceiver and secondly between 2 nodes could be easily reached. Board is also equipped in RS232 junction, which may be used in communication with PC. However, increasingly smaller amount of computers is equipped with this interface. USB gains in popularity since a couple of years. For more comfortable usage (without a need of binding CAN controller with microcontroller, debugging diodes or other chips with single wires) and further interface development new hardware platform was proposed in chapter *5.3 New hardware platform*.

More information about NIMRON board can be found in literature[10] or in the Internet[1].

[1]*http://www.nimron.no/P1000/*

5.2.2 Programming language and software platform

Microcontroller programming in case of AVR processors may be done using different programming languages like: *Assembler*, *Bascom* or *C*. Definitely the most efficient is low-level Assembler. However, complex application desires expanded code structure, which implies rather big code volume. Therefore, one of the most user-friendly and common programming language for this purpose is C. Implementation of the PDCP was programmed using that C language.

One of the most common software platforms for programming of AVR microcontrollers are Eclipse and AVR Studio. Because Eclipse does not contain build-in plug for AVR, some additional one has to be installed. For this reason, AVR Studio 4.0 dedicated to AVR controllers has been chosen.

Moreover, in the software development AVR LIBc[2] library has been used. It provides a high quality C library for use with GCC compiler on AVR microcontrollers, while licensing under so-called modified Berkeley license compatible for example with GPL. That, thanks to code structure described in the next subchapters, allowed to avoid tedious process of coding of input-output port addresses or other essential low hardware issues while maintaining high level of portability between microcontrollers from the same and similar families.

[2] Source: *http://www.nongnu.org/avr-libc/*

5.2.3 CAN controller – initialization

MCP2515 stand-alone CAN bus controller implements standard CAN2.0B with transmission speed up to 1Mb/s. Communication with host is executed through 4-wires SPI interface with speed up to 10Mb/s. Mechanism of message reception bases on two acceptance masks and six acceptance filters that are used to filter out unwanted messages, reducing microcontroller overhead.

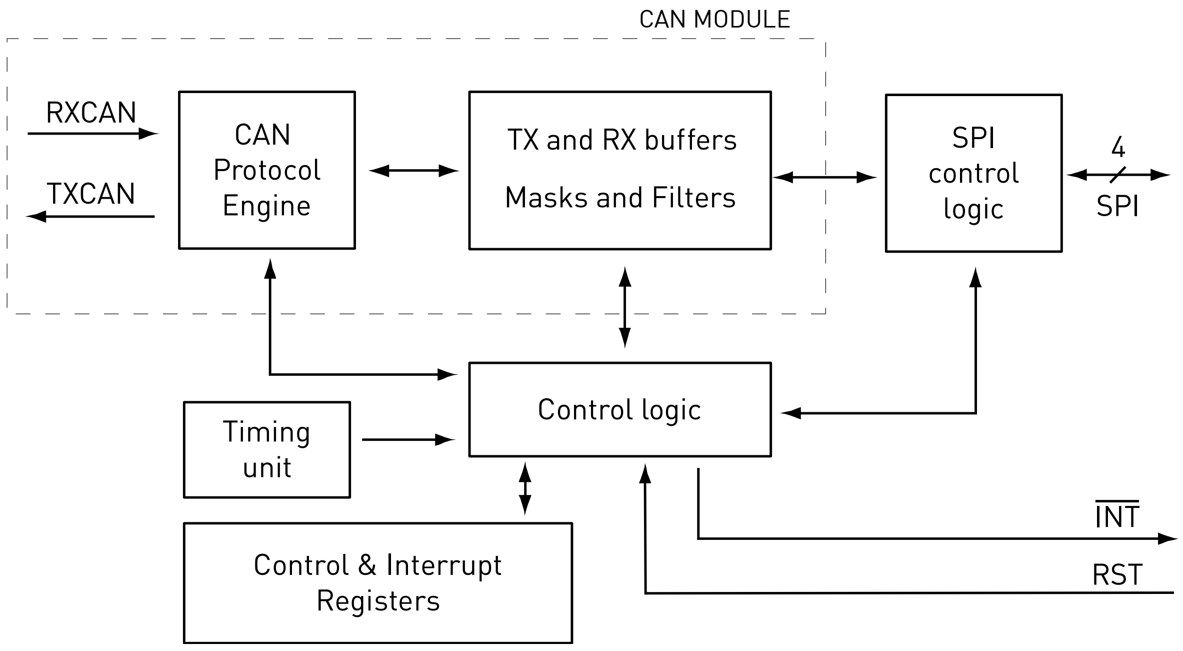


Fig. 4 MCP2515 simplified block diagram

The diagram presented in Fig. 4 shows elements of CAN controller mediating in the mechanism of data transfer. Before any CAN message transmission, initialization has to be performed. This process concerns setting mentioned flags and masks depending on whether node is supposed to a DEVICE or an ARBITRATOR.

The controller contains also registers responsible for the bit rate. Because the CAN protocol uses *Non Return to Zero* coding, which does not encode a clock within the data stream, therefore the clock of receiver node has to be synchronized to the transmitter`s clock.

An experiment showed that with default speed of transmission all registers associated with bit timing can have default (unchanged) value. However, with increased speed of transmission (up to 1Mbps) CAN bus stops operating or operation is dependent of inexplicable plexus case. In a word, the system becomes unpredictable, which is undesired.

The second aspect is the possibility of different clock frequencies of individual nodes, which should not damage the system. Therefore, following a documentation of MCP2515[8]: "*the bit rate has to be adjusted by appropriately setting the baud rate prescaler and number of time quanta in each segment*".

Because of technological barriers or oscillator mismatch phase shifts may occur during transmission. To prevent transmission errors each CAN controller within a system must be able to synchronize to the relevant signal edge of the incoming message.

Taking into account all above factors, to provide the best performance registers containing relevant data for bit timing and synchronization should be well adjusted to each other. For this purpose CAN bus timing calculator available in the Internet[3] has been used. The highest speed together with high level of reliability was achieved at speed of 1 Mb/s and SPI transmission at the level of 4MHz (literature does not recommend speed faster than 0.25 * frequency of microcontroller operation because of the risk of instability).

[3]*http://www.kvaser.com/en/support/bit-timing-calculator.html*

5.2.4 Microcontroller resources

The Hardware Abstraction Layer should provide quick and reliable data transmission between upper software layers and hardware while minimum use of microcontroller operation time and its resources. The latter might be needed for other purpose of either the PDCP or application not necessarily known right now, but may be used in the future. Moreover, simpler microcontrollers contain poorer peripheral resources, however protocol operation together with even simple application should still be possible.

The designed HAL benefits from the following microcontroller resources:

1 external interrupt – event indication from the external CAN controller;

2 external interrupts – playing role of software interrupts for message sending and receiving;

SPI interface – for communication between microcontroller and CAN controller;

USART interface –for debugging purposes.

All other resources have been left for the upper layers of the PDCP protocol and application designer.

5.2.5 Overall system structure

In Fig. 5 overall software design of the Hardware Abstraction Layer was depicted on UML sequence diagram. All diagrams concerning code structure in this paper were generated in Visual Paradigm for UML. For readability only main calling between interrupts and functions were presented. The procedure of sending and receiving (including references between the HLL and the HAL) was described in details in next chapters.

The HAL beside microcontroller and CAN controller initialization provides mechanism of message sending and receiving and error indicating in the form of message structure which is readable from level of the HLL or the application.

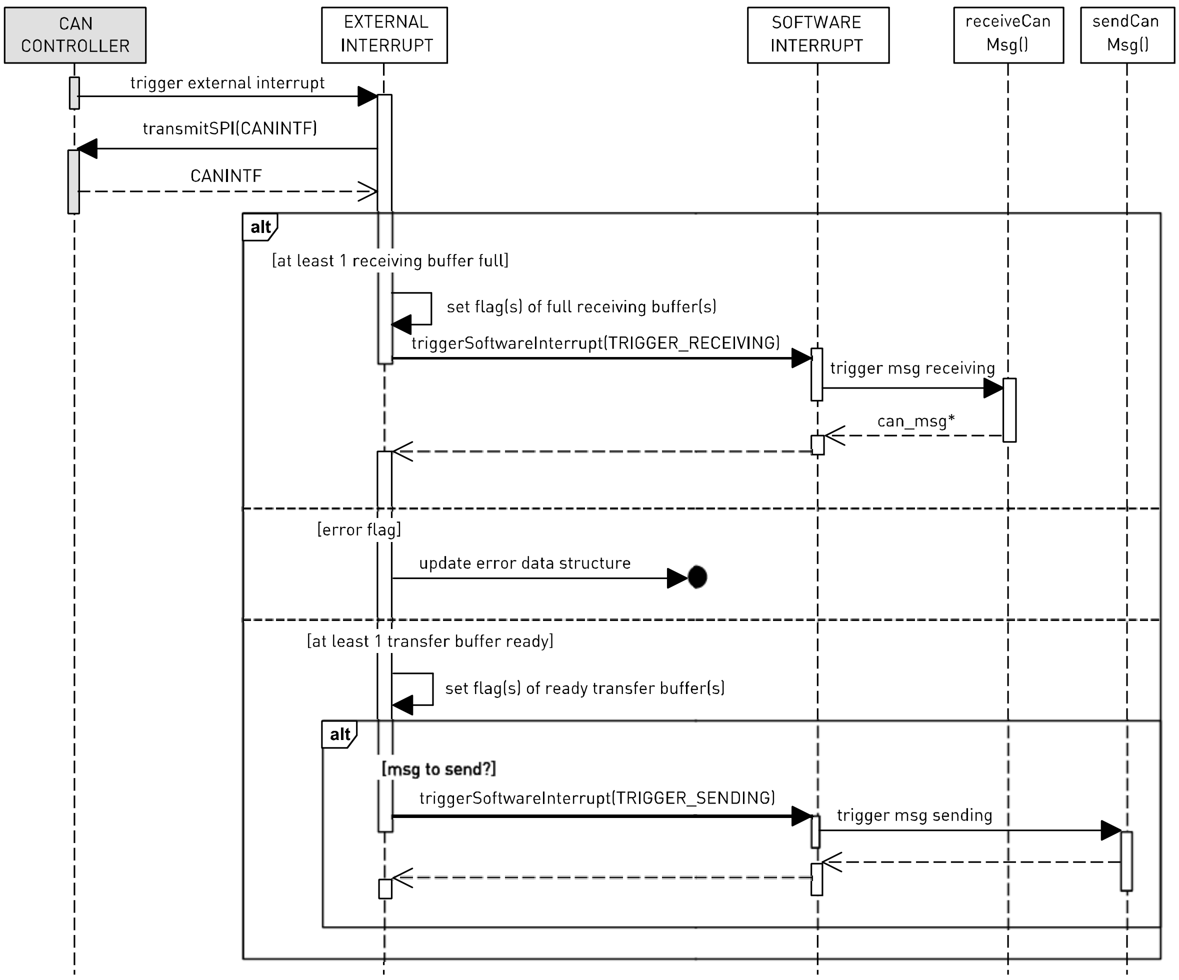

CAN CONTROLLER
EXTERNAL INTERRUPT
SOFTWARE INTERRUPT
receiveCan Msg()
sendCan Msg()
trigger external interrupt
transmitSPI(CANINTF)
CANINTF
alt
[at least 1 receiving buffer full]
set flag(s) of full receiving buffer(s)
triggerSoftwareInterrupt(TRIGGER_RECEIVING)
trigger msg receiving
can_msg*
[error flag]
update error data structure
[at least 1 transfer buffer ready]
set flag(s) of ready transfer buffer(s)
alt
[msg to send?]
triggerSoftwareInterrupt(TRIGGER_SENDING)
trigger msg sending

Fig. 5 Overall HAL program structure – UML diagram

5.2.6 Messages sending

The procedure of messages transfer from the HAL point of view is closely related to upper layers of interface (HLL). The CAN controller (MCP2515) exchanges data with microcontroller using SPI interface. Choice of implementation of the PDCP using AVR microcontroller without built-in CAN controller has its advantages. One of them is the fact that some information about bus capacity and speed of transmission for system with external CAN controller may be obtained. Any medium between two electronic circuits slows down speed of transmission of overall system. Firstly, because some computations indispensible for proper communication between circuits has to be performed and secondly, because of data exchange between controller and microcontroller which is definitely more time-consuming than updating internal registers.

Such information can be useful because even taking into account miniaturization and integrated peripherals in many simple prosthesis modules some very simple and cheap microcontrollers will be used.

On the other hand, the proposed software structure should be very simple for adaptation to microcontrollers with internal CAN controller, much simpler than in the opposite direction. Also time needed for computation and preparing data to transmission (indirectly also overall transmission time) between modules should be shortened.

To illustrate the mechanism of messages sending UML diagram was presented in Fig. 6.

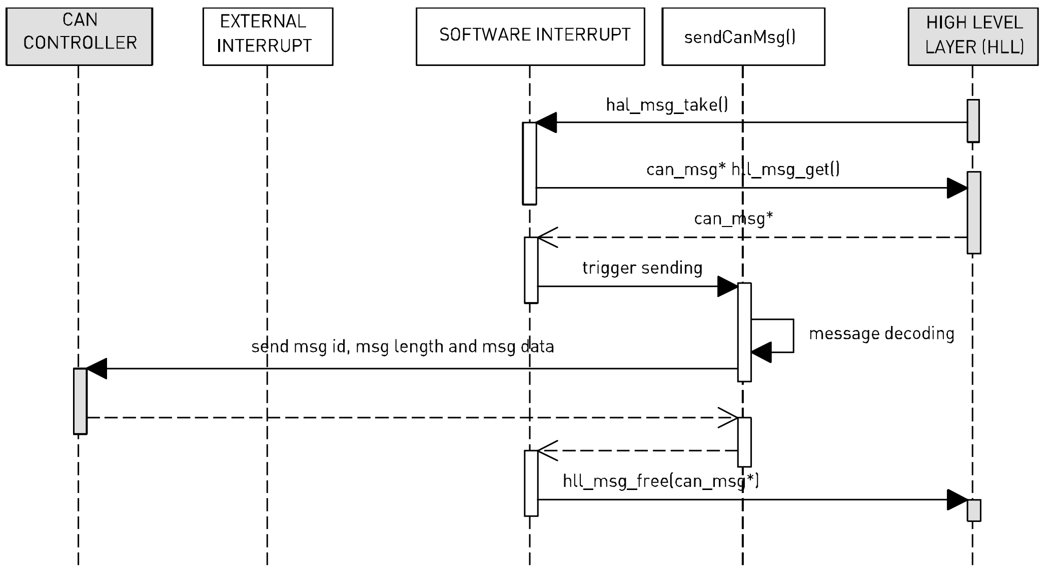


Fig. 6 Sending procedure - UML diagram

As follows from Fig. 6 sending has to be initiated by the High Level Layer by calling a function: *hal_msg_take()* which calls function *triggerSoftwareInterrupt()* with an argument *TRIGGER_SENDING*. As the name suggests, this function implemented inside of the HAL software is responsible for triggering software interrupt.

After triggering software interrupt, a pointer to the message is received from HLL by one of designed methods called *hll_msg_get()*. Afterwards, depending on which transmitting buffer is ready (which should be done by CAN controller before transmission by triggering external interrupt and setting appropriate flag), the software calls *sendCanMsg()*. This function is responsible for decoding message structure and sending all essential parameters like the message id, the number of transmitted data bytes and data bytes itself to the CAN controller. After that software should clear flags of empty transmitting buffers. For better memory usage *hll_msg_free()* function should be called with the pointer as the argument to mark memory containing CAN message as unused.

Mentioned *hll_msg_free()* function should also check whether any message is ready to reception and, if necessary, use declared pointer to this purpose (more details about this aspect may be found in the chapter concerning message receiving).

The mechanism of messages sending is relatively less important in priority than receiving. Node can wait with sending messages without significant losses, while receiving should be handled as soon as possible in order not to overflow receiving buffers. Therefore, interrupt handling message sending should have lower priority than interrupt of message reception. In case of only one interrupt devoted to the whole transmission mechanism priority will be the same, of course.

The whole mechanism of the Hardware Abstraction Layer is event-triggered and uses external interrupts both for communication with CAN controller (event indication) and also for triggering software interrupts by changing one of microcontroller pins. In this way, the triggered interrupt has a priority of one of external interrupts – lower only from reset interrupt, which might be desired in case of important message transmission. This mechanism has been used instead of Pin Change Interrupt (delivered by microcontroller AT90USB1287) because simpler microcontrollers do not support such interrupts. Electronic circuits market review showed that typical amount of external interrupts supported by AVR microcontrollers reaches 2 in case of small ones, and 8 in case of bigger ones. Taking into account the designed interface uses one external interrupt for CAN controller and 2 interrupts for sending and receiving, code may be simply adopted to only two external interrupt (one for controller and one for interface).

5.2.7 Messages receiving

Procedure of message receiving was presented in Fig. 7.

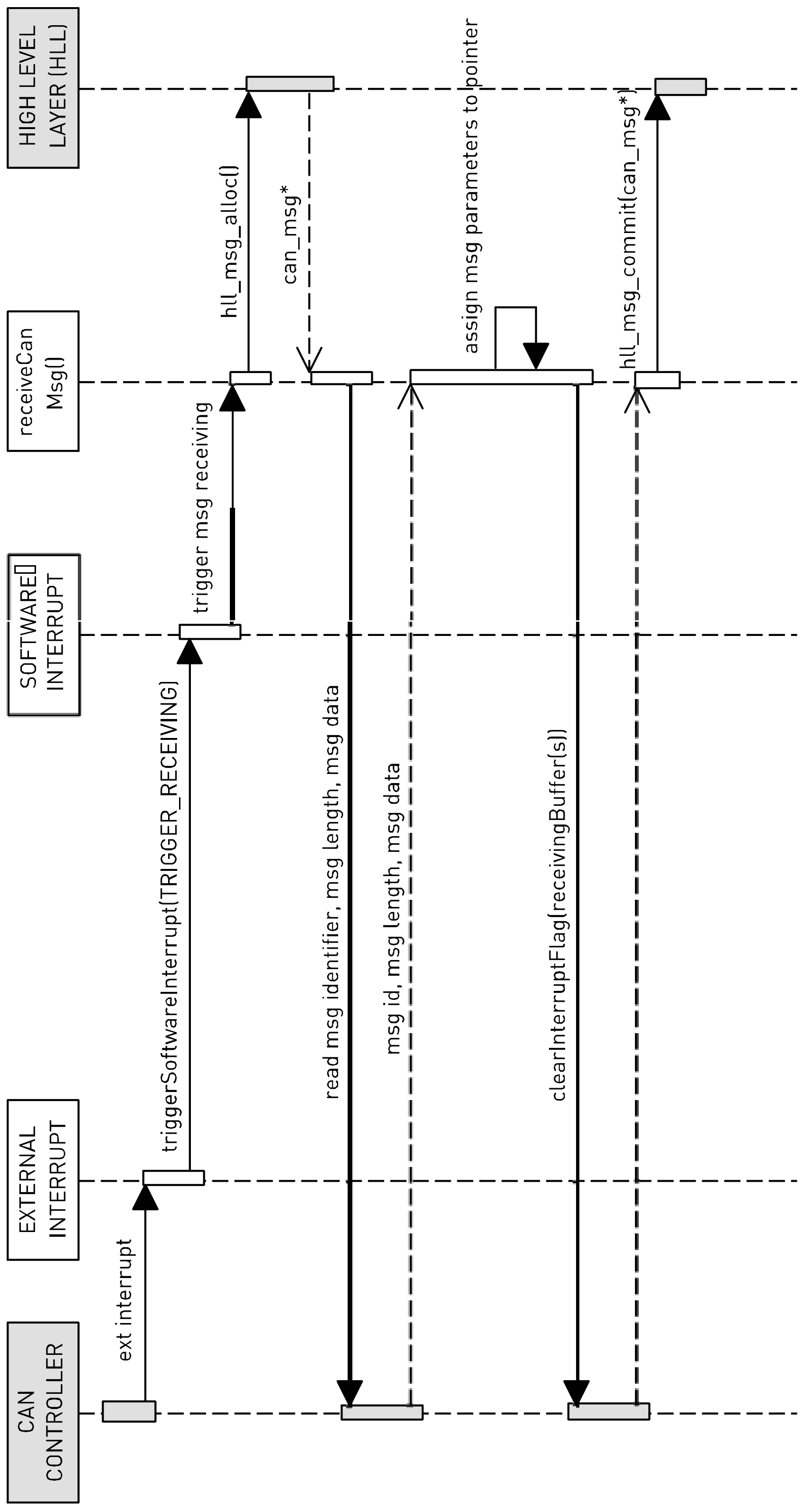

HIGH LEVEL LAYER (HLL)
receiveCan Msg()
SOFTWARE INTERRUPT
EXTERNAL INTERRUPT
CAN CONTROLLER
ext interrupt
triggerSoftwareInterrupt(TRIGGER_RECEIVING)
trigger msg receiving
hll_msg_alloc()
can_msg*
read msg identifier, msg length, msg data
msg id, msg length, msg data
assign msg parameters to pointer
clearInterruptFlag(receivingBuffer(s))
hll_msg_commit(can_msg*)

Fig. 7 Message reception mechanism (UML diagram)

Message receiving is triggered by the CAN controller whenever the valid message is received. External interrupt handler checks the source of interrupt and sets appropriate flags (full receiving buffer flag, in case of received message). Just after that suitable software interrupt is triggered by calling function *triggerSoftwareInterrupt(TRIGGER_RECEIVING)*. Basing on the saved number of full receiving buffer*receiveCanMsg*() function with appropriate parameters is called. The mentioned function calls *hll_msg_alloc*() which allocates memory for received message in the HLL. If allocation is not succeeded (internal stack is full), the message is left in the controller with remaining receiving the flag set. Also, the flag of interrupt inside of controller is not cleared (which preserves against message overwrite) – edge triggered interrupt allows to go out of interrupt routine, process some other data, release memory for waiting message(s) and retry message reading. This is executed thanks to the mechanism of sending – function *hll_msg_free*() called from sending routing should check, whether any messages are ready to fetch from controller.

If allocation has been succeeded, program executes the block of transmissions with CAN controller to read message id, number of data length and data itself and assigns them to pointer reached from allocation. Afterwards, interrupt flag clearing is executed by calling a function *clearInterruptFlag()* with an appropriate flag of interrupt inside of the internal flag register CANINTF. Then the pointer is committed to the HLL by calling a function *hll_msg_commit()* and the receiving mechanism in the HAL is completed. Message is then processed inside of the HLL.

5.2.8 Additional functionalities

The designed software contains additional functionalities, which either provide resources handling, increase code portability or help in software debugging while startup at different AVR platform or further software development. The implemented functions were shortly described in the next part.

EEPROM handling - nodes of prosthesis system contain much information about their vendor, serial number, transmission channels and other relevant parameters. Some of them are used only during program operation and there is no need to keep them in memory. However, some of them have to be saved in the non-volatile memory either external or internal one. Because the used microcontroller is equipped with internal EEPROM (2kB) functions for EEPROM handling have been implemented. For some microcontrollers internal memory might not be big enough and external memory is needed – in this case good solution could be memory with SPI interface (SPI transmitting function is implemented for communication with the CAN controller). Functions for EEPROM handling are:

void writeEEPROM (uint8_t address, uint8_t data)

uint8_t readEEPROM (uint8_t address)

Error handling – although message sending in MCP2515 is retried up to 255 times, sometimes transmission errors may occur. Application working with the PDCP should be informed that error(s)happened, therefore error structure was implemented. Every error indicated by interrupt from controller increments the value of specified elements of this structure. The application can read the whole error structure by calling function:

checkTransmissionErrors(struct errorStr).*

USART debugging – inside of the HAL USART handling was included, which occurred a very useful tool while debugging because it enabled printing text messages in the form of strings on serial port. In conjunction with designed PCB board (which uses simple USB port and integrated circuit *FT232RL* emulating RS232 port) and serial terminal program installed in the computer, application designer is able to print both text and variable values on the screen. The latter is possible thanks to *itoa() or sprintf()* function provided by *stdlib* library of AVRLIBc. Function for string printing is:

printUsart(uint8_t).*

5.2.9 File structure

Project implementation was physically divided into several files, which structure was presented in Fig. 8.

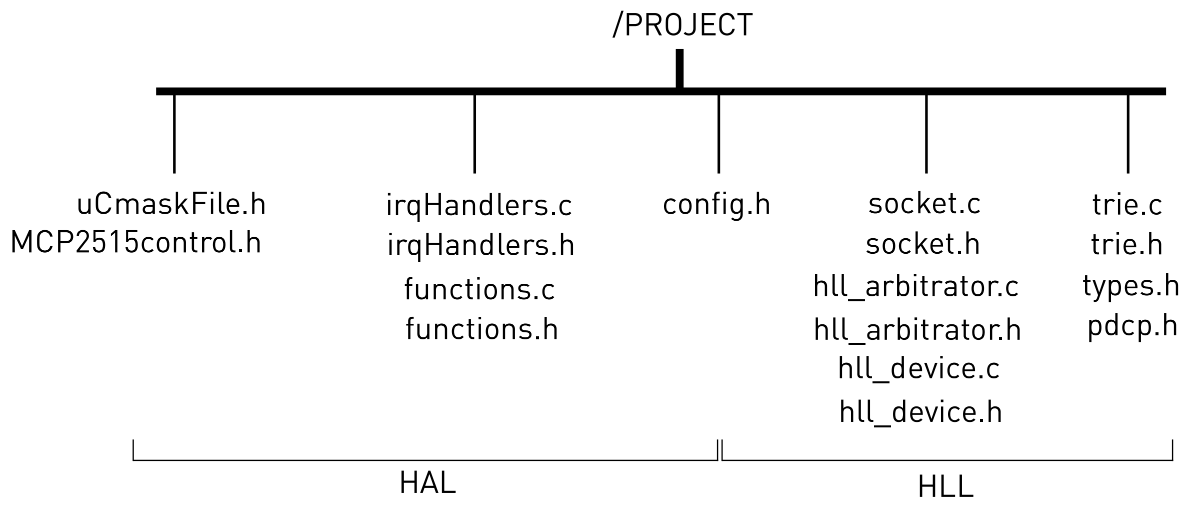


Fig. 8 Files structure

The Hardware Abstraction Layer software was separated into 3 groups of files:

Hardware configuration files: *uCmaskFile.h*, *MCP2515control.h*

uCmaskFile.h – contains preprocessor directives which are used to mask AVR LIBc library.

Example: *#define EXT_INT_PCIFR PCIFR*

At a glance, such solution might seem to be useless. However, microcontroller exchange desires only replacement of its header file and refreshing hardware configuration directives (located in *config.h*) without a need to touch “sensible software body”, which (without knowledge how the interface is designed or simply by accident) could be real danger for interface operation.

<u>This solution increases portability of designed software by the decreased number of changes in the software in case of hardware exchange</u>.

MCP2515control.h – contains library for MCP2515 CAN controller.

All essential register addresses and enumerated types facilitating controller handling have been contained.

Interface implementation (HAL):

irqHandlers.c, *irqHandlers.h, functions.c, functions.h*

The above files contain external and software interrupt handlers and function declarations responsible for providing the mechanism of message sending and receiving by communication with CAN controller. All software responsible for microcontroller input-output port initialization, peripherals configuration and handling may be found within *functions.c* file. Within *irqHandlers.c* interrupt routines were defined together with functions responsible for message transmission.

Software configuration file:

config.h

This file contains all software configuration code for setting bus mode for particular node, enabling debugging modes or defining important parameters of the HLL.

It should be strongly emphasized that content of this file should be kept under control every time software is compiled and uploaded to microcontroller. Its content is shared between both the HAL and the HLL of the designed PDCP.

5.2.10 Code structure

To show modularity and portability capabilities code structure is shortly described. Firstly, the designed software has been divided into the blocks of functions which enable simple redesign in case of hardware change. Adjusting software to hardware change thanks to structural approach is relatively easy and not time-consuming. To handle the lowest hardware issues (like port or SPI peripherals initialization), the specified functions are implemented and only these ones have to be redesigned in case of hardware exchange. That division provides also transparency of lower layers in the relation to upper layers of the PDCP protocol and application designer. Definitely, it is desirable that designers of all layers above the HAL do not have to and cannot directly handle hardware issues. Function blocks have been presented below.

Microcontroller hardware – block of functions responsible for microcontroller hardware initialization. It consists of following implemented functions:

void initIntPorts(void) – the function responsible for initialization of ports responsible for external interrupt triggering;

void initSPIPorts(void) – the function responsible for initialization of SPI ports;

void initIO_appDesigner(void) – the empty function left for the application designer for I/O ports initialization;

void initIOPort (void) – the calling functions: void *initIO_appDesigner*(void), *initSPIPorts()* and *initIntPorts()*;

void initInterrupts_appDesigner(void) – the empty function left for the application designer for interrupt initialization (timers, external interrupts or others);

void initInterrupts(void) –the function setting HAL interrupts and calling *initInterrupts_appDesigner ()* function;

SPI_INIT_STATUS initSpi(void) –the function initializing SPI interface of microcontroller.

CAN controller communication – the block of methods that specify communication with external CAN controller:

CAN_INIT_STATUS initCan (void) – the function responsible for CAN initialization;

uint8_t mcp2515TransmitData (uint8_t mode, uint8_t reg, uint8_t val, uint8_t setClearFlag) – the function is responsible for transmitting data to specified register within CAN controller. As arguments it takes controller mode of operation, register, value and information about flag clearing or setting (only Bit Change Mode);

void clearMCP2515InterruptFlag (uint8_t reg, uint8_t flag) – the function that is used for clearing flag after external interrupt from CAN controller. The arguments are register address and flag (bit) to clear.

Interface communication – this block contains function used in interrupt handlers for message sending and receiving:

void receiveCanMsg (uint8_t regRec) – the function that fetches received message from register pointed by regRec CAN controller and assign value to pointer to can_msg struct which is committed to HLL;

void sendCanMsg (struct can_msg msgToSend, uint8_t regToSend)*–the function that sends message (pointed by pointer msgToSend) to register pointed by regToSend.

Other functions

void forceReset (void) – the function that triggers software reset;

void wdtDisable() – the function that disables watchdog timer;

void wdtEnable(uint8_t time) – the function that is responsible for watchdog initialization.

Two functions implemented in the "Microcontroller hardware" block have been left empty for application designer. This allows to avoid problems coming from probable insufficient knowledge about interface. The application designer does not need to know where some additional hardware initialization should be done so as not to damage the PDCP. Instead, he/she is advised to fill blank functions, which are called in places in the code proper and safe for the interface. This solution should reduce the risk of unintentional errors.

The important aspect has been code documentation. One of the most popular and common program supporting generation of documentation is *Doxygen*. It is a standard that specifies style of code comments, on which generator builds ready-to-use .*html* files. The performed documentation both the HAL and the HLL of the PDCP is available on the CD attached to this dissertation

5.2.11 Name convention

In the agreement with the student responsible for the High Protocol Layer of the PDCP name convention has been used. The short description was outlined below.

CONFIG_xxx	Both the HAL and the HLL contain part of software dedicated to special functionalities, which either are used only for debugging or for some configuration. To facilitate the use of the software compilation and running of the part of the code can be manually enables or disabled by changing *#define* preprocessor directives. All the directives which are related to bus device mode or define debugging modes begin with the prefix CONFIG_. Examples: *CONFIG_BUS_MODE* – ARBITRATOR/DEVICE
HW_xxx	Taking into account the above introduction, the lower layers of interface, in contrast to the upper layers, contain low level hardware initialization. To improve portability all ports relevant for proper operation of HAL are signed with HW_ prefix. These directives should be absolutely refreshed after the microcontroller exchange. Example: *HW_MISO_MCP2515* – port MISO for SPI transmission with CAN controller.

The second aspect that should be described concerns name convention within interface between the HAL and the HLL of the PDCP. Naming, described in next part, was established.

hll_x_y_z	The functions, implemented in the upper layers of the PDCP, are called from lower layers (HAL) and have prefix *hll*. Individual names are separated by "_". These functions are responsible for memory allocation and messages handling. Example: *hll_msg_alloc*() – the function responsible for memory allocation
hal_x_y_z	The functions that are implemented in the lower layers of the PDCP and are called from upper layers (HLL) have prefix *hal*. The individual names are separated by "_". These functions are responsible for module mask and filter setting and message polling. Example: *hal_set_mask*() – the function responsible for module CAN receiving configuration

Within Hardware Abstraction Layer of the PDCP CamelCase convention is used. It is the practice of writing compound words, in which elements are joined together without a space or underscore ("_")character.

5.3 New hardware platform

Although the main idea of this project was the PDCP software implementation, there is no other way to check portability than to try to implement code on different microcontroller. Another reason for the new hardware design is a lack of AVR PCB board projects dedicated to this particular interface. Board used during this implementation described in chapter *5.2.1 Hardware architecture* was very comfortable solution especially at the project beginning, but necessity of continuous wiring exchange was irritating. Taking into account above factors, a block diagram of the new design was presented in Fig. 9.

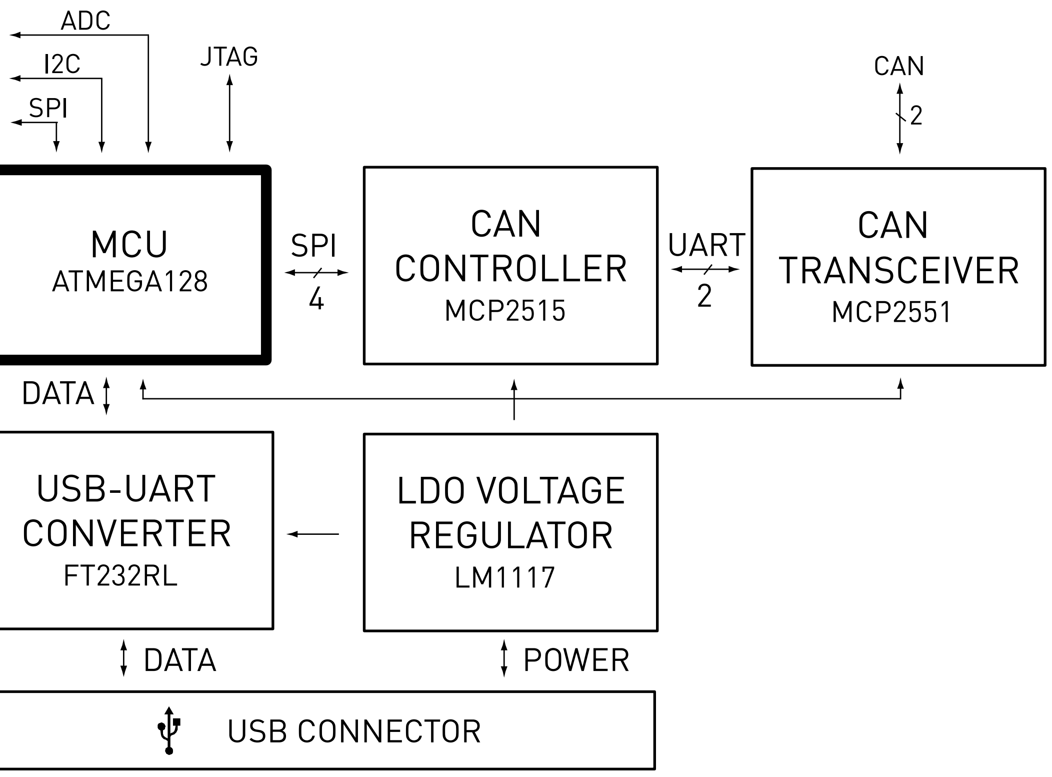


Fig. 9 Designed PCB board - block diagram

Electronic circuits on the board are supplied with voltage from USB junction that is converted to 3.3 V by Low Dropout voltage regulator – it is the most common and sensible solution to have stable voltage supply from USB port with relatively the lowest power loss. The board contains also external supply possibilities without using USB by external junction (max. 5V).

Microcontroller benefits from USB and using FTDI circuit (which converts USB port for UART)it is able to communicate directly with computer using installed terminal program – there is no need of using RS-232 junction.

As mentioned before, the software has been enriched with functions handling USART communication, so debugging process while components (e.g. microcontroller) exchange should be much simplified.

The board is equipped also with CAN controller and CAN transceiver (the same IC as hardware given to project). Analog-Digital Converter is very important from prosthesis point of view, therefore together with other interface ports (like SPI, I2C) have own junctions on the board. This solution simplifies connection of electrodes or other sensors or actuators into the microcontroller and enables testing not only interface itself, but almost the whole prosthesis system.

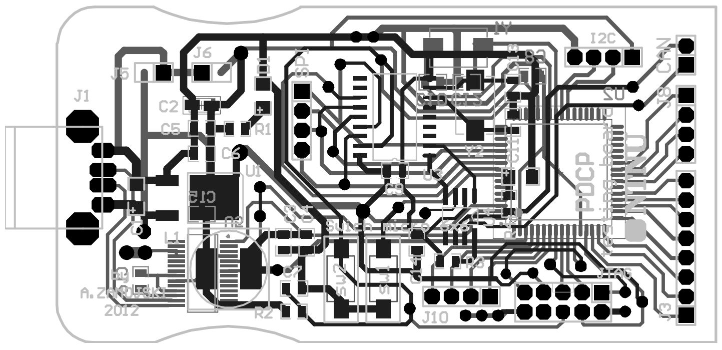


Fig. 10 Printed Circuit Board layout

In Fig. 10 layout of PCB board was presented. The double-sided board has a shape of rectangle (88 x 45 mm) with USB-A connector which allows direct communication with PC. On the connector side comfortable mild notches are placed. The board contains also 2 buttons (one for reset, one for other purposes defined by the programmer), JTAG junction. I/O ports, especially analog-to-digital converter and interfaces like SPI and I^{2}C are situated at the edge of board enabling simple connection. The designed platform is dedicated for further development of the standardized protocol for prosthesis.

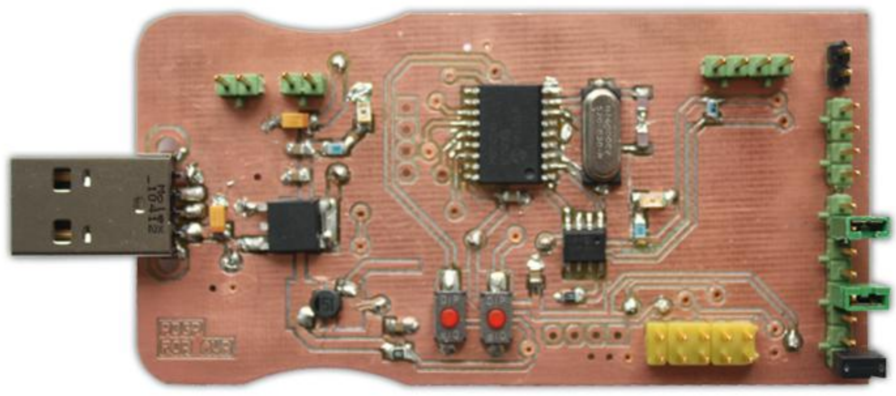


Fig. 11 New hardware design - after assembling

Assembled card described in the previous part was presented in Fig. 11.

One can ask question, why new hardware design was proposed? Comparing proposed new hardware together with NIMRON board undoubted advantage of the new design is simplicity of connections on the board and avoidance of additional wires desired for PDCP start-up. Board contains UART – USB converter and USB plug which facilitate software development. Moreover, board was equipped with external pins of the most essential interfaces and ports from prosthetic point of view, like ADC or I2C. Thanks to this solution external components may be very easily connected to the board and tested. On NIMRON board not all needed ports were delivered in form of external pins. Additionally, amount of wires used for simple testing complicates comfortable usage.

NIMRON board is great base for code start-up. New designed hardware gives possibility of node assembling with reduced number of external wiring and simplicity of connections.

6 Testing

6.1 Introduction

One of the most important part of this dissertation is the discussion of testing results. Content of this chapter treats about code testing of the designed HAL and interface between the HAL and the HLL.

As it was mentioned before implementation of the PDCP was divided into two parts, what definitely improved modularity and portability. However, this solution influenced also testing complexity at start-up, because part of codes designed by different programmers are threatened with negative interference, what was the case within this project and was described in the next parts.

The Hardware Abstraction Layer, which is described in this thesis, is to some extent independent from the HLL, therefore testing only the HAL was definitely easier than the HLL alone. The HAL was tested under following terms:

Assuring maximization of code portability and modularity – code structure should maximize portability and decrease amount of changes in case of hardware exchange;

Providing mechanism of message sending and reception – the designed HAL should provide efficient mechanism of data exchange between software and internal registers of CAN controller. This action should be executed in tread-off between the least possible usage of hardware resources and decreased time of operation;

Providing data lossless transmission – designed software should fulfilled assumption of lossless transmission between nodes of the system;

Providing error detection mechanism – because CAN controller, used in the project, provides mechanism of error detection information about actual bus state should be transferred to the HAL and/or application (because architecture of application is not known in this stage of protocol development, therefore and/or statement was used).

Main features pointed above were tested and described in the next parts.

6.2 Code portability – testing

The only way of testing code portability is a hardware change and objective assessment of the effort made to adjust code to this new hardware. For testing code portability new hardware design described in *5.3 New hardware platform* with ATmega128 working as the arbitrator and one NIMRON card, working as the device, were used.

At the beginning, basing on configuration files (*uCmaskFile.h* and *config.h*) code was adjusted to differences between microcontrollers (*AT90USB1287* and *ATmega128*). This process was not so trivial as expected, although both microcontrollers were from the same AVR 8-bit family. Names of many registers or even initialization of particular hardware resources differed slightly, what resulted in a couple of hours spent for finding these differences. After software error elimination many problems with hardware were encountered, which probably disabled proper operation of the device. Tedious process of looking for nonconductive vias, checking termination resistance at the nodes, physical connections between CAN transceivers or transmission between chips on the board using oscilloscope didn`t help in finding reasons of wrong transmission between arbitrator and device - message was not delivered to the device, what was indicated by errors of transmission from CAN controller.

Aspects of code portability were presented in the chapter *5.2.10 Code structure*, because it is directly related to the way of coding, specifying functions and code structures. However, after portability testing a few aspects, inadvisable in term of portability, should be emphasized. These features were listed below.

Data types with not precisely specified size –data types like *int* should be avoided and replaced with types *uint8_t* or *uint16_t*. Specified size allows microcontroller to interpret data type correctly and avoid problems with different meaning of the data type in different microcontrollers;

Usage of functions specified for particular model or family of microcontrollers – code should be as far as possible independent from not universal external libraries or implement solutions masking this library to increase portability (in case of this project *uCmaskFile.h* is an example of such masking file of libraries delivered with *AVRLIBc*);

Not clear code structure – if code is expected to be portable should be well structured and documented. Complexity of the project increases time of adjusting software to exchanged hardware.

Features impacting portability related to hardware:

Code part related strictly to hardware implemented within other code – in case of any hardware exchange designer is forced to look for every code snippets within project responsible for hardware handling. This activity should be avoided;

Unpopular hardware resources or peripherals – software should as far as possible use hardware resources (like SPI, timers, small amount of external interrupts, efficient usage of built-in EEPROM memory etc.) common for almost all microcontrollers from different families and vendors.

6.3 Mechanism of message sending and receiving – testing

The only way for message exchange testing between two nodes is sending a message from one node, receiving in another one and checking accordance of data fields. This testing is somewhat associated with testing data lossless transmission described in chapter *6.4 Data lossless transmission – testing*, because message exchange mechanism should work reliably.

CAN transmission may be easily previewed using an oscilloscope . Example of data transfer in designed interface between two nodes was presented in Fig. 12 and Fig. 13. From physically point of view Arbitrator and Device (in the PDCP notation) are almost identical. The only difference is hidden in the mask and filter configuration, what cannot be observed in electric signal presented below. As it was observed after change of Device Id and filter reconfiguration message, arbitration field had to be also refreshed, otherwise message transmission stopped working. Therefore it`s important for upper layers of the PDCP to bear in mind necessity of filter reconfiguration every time *NodeId* has been changed (for example process of node binding to system).

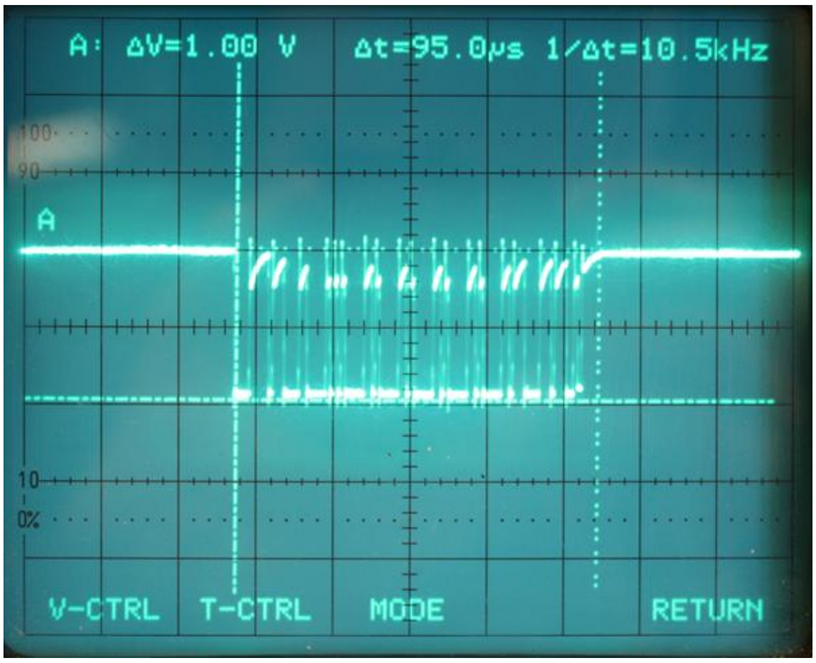


Fig. 12 Single data frame (oscilloscope)

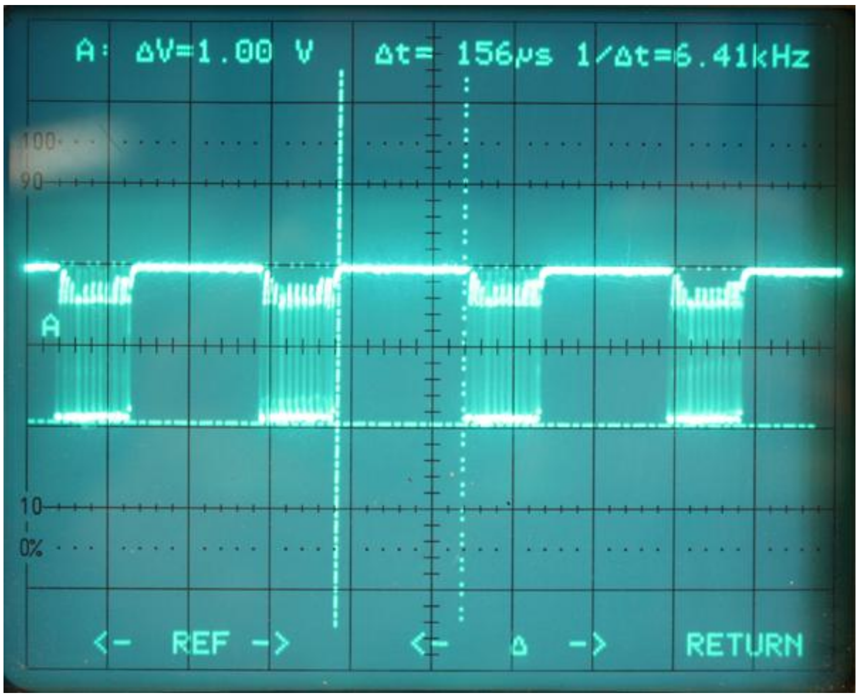


Fig. 13 Measurement of the delay between data frames (oscilloscope)

Using cursors of oscilloscope time and quantitative parameters of CAN bus signal were measured.

CAN bus signal takes differential values 0 up to 1V. Time of transmission of single data frame with 8 byte data field takes 95 μs while time distance between frames equals 156 μs, what gives bus utilization at the level of 37%. Taking into account the test involved only 2 nodes (only one sending) that situation was acceptable. Data transmission between microcontroller and CAN controller takes time, which the delays come from. From measurements can be concluded that majority of time bus is inactive, but for real prosthesis system consisting of many nodes, working independently from each other and exchanging messages in various time moment through the same bus, utilization should increase.

Simplified diagram of testing message transmission between 2 nodes was presented in Fig. 14. Photo of simple system under tests was depicted in Fig. 15.

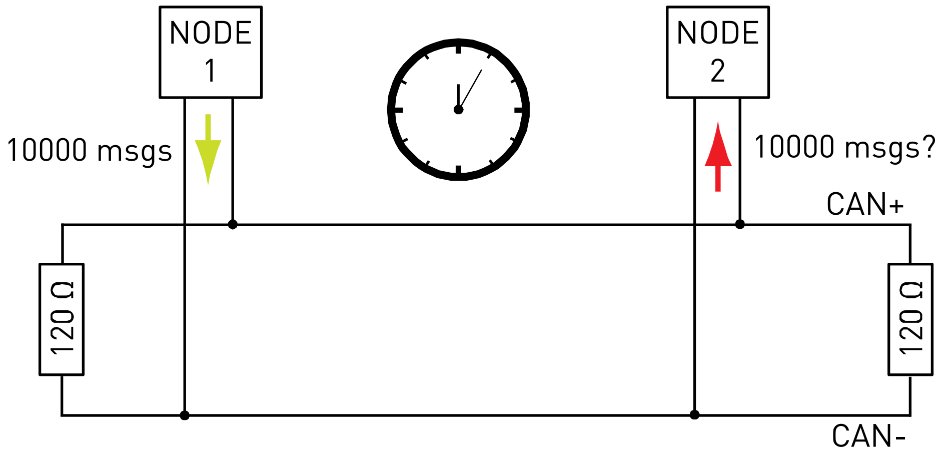


Fig. 14 Test of data exchange mechanism

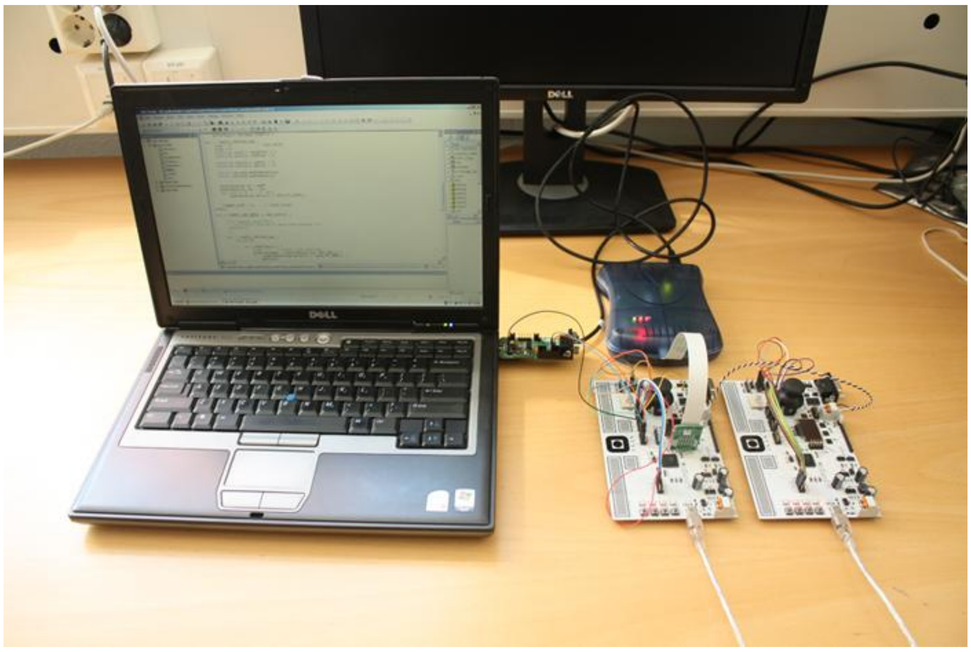


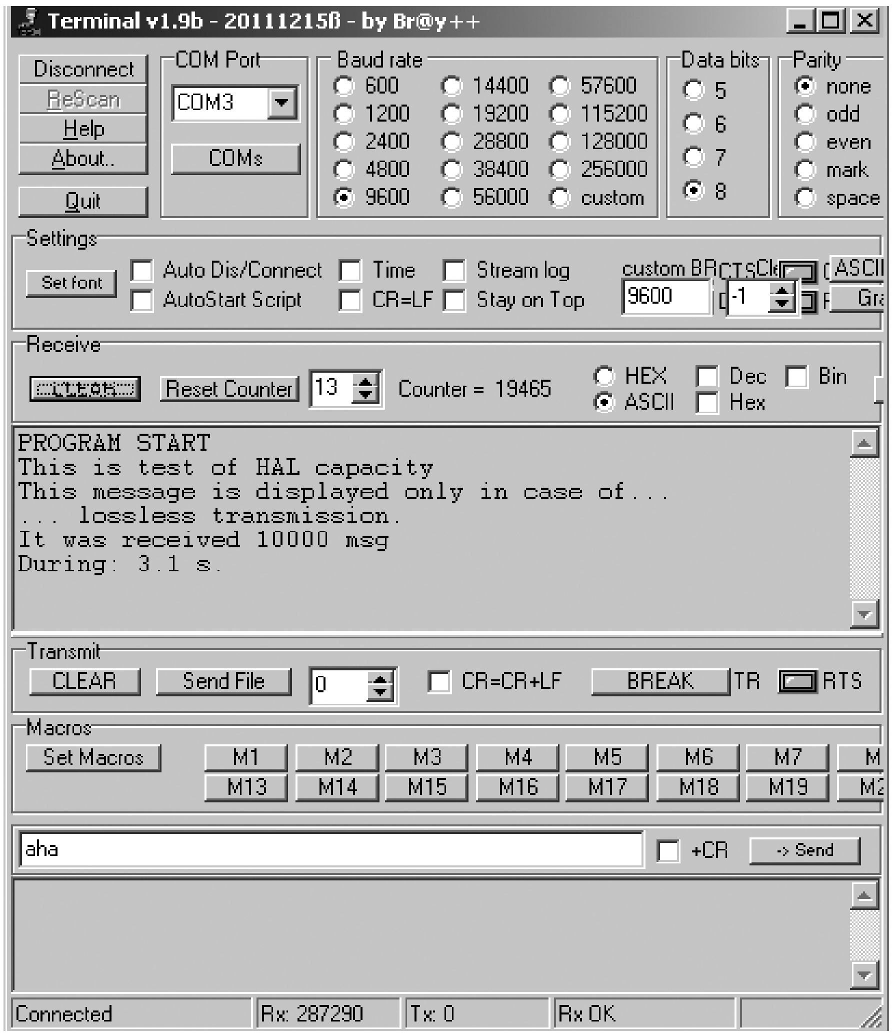
Fig. 15 System consisting of one Arbitrator and one Device under tests

From one node (configured as a Device) benchmarking program sends messages to second node (configured as an Arbitrator) one after another in *while* loop. To reach the greatest possible accuracy and not introduce undesired delays just after indication of empty transmitting register next message is shifted into CAN controller. Control over timing is kept by timer overflow interrupt which increments counter variable every 100 ms. This accuracy seems to be high enough for estimating average time of transfer of certain amount of messages.

6.3.1 Impact of SPI speed on the HAL interface capacity – testing

AT90USB1287 contains SPI interface which may be configured to operate with frequencies dependent of microcontroller frequency with following dividers from /2 up to /128. From stability point of view, it is advised to use dividers equal or higher than /4 (more information can be found in literature[7]).

To illustrate result of testing UART-USB converter together with *Terminal v1.9b*[4] was used. In the Fig. 16 screenshot from one the tests was presented.


Terminal v1.9b - 20111215ß - by Br@y++

COM Port: COM3
Baud rate: 9600
Data bits: 8
Parity: none
custom BR: 9600
Receive: ASCII
Counter = 19465

```
PROGRAM START
This is test of HAL capacity
This message is displayed only in case of...
... lossless transmission.
It was received 10000 msg
During: 3.1 s.
```

aha

Connected | Rx: 287290 | Tx: 0 | Rx OK

Fig. 16 Testing output from RS232 terminal

[4]*https://sites.google.com/site/terminalbpp/*

Short description of benchmarking parameters for testing impact of SPI interface between microcontroller and CAN controller on interface capacity was presented in Table 11.

Amount of messages – 10 000

Size of data field – 8 byte

SPI speed – variable

Table 11 Impact of SPI speed on HAL interface capacity

SPI divider	/64	/32	/16	/8	/4	/2
Time [s]	*	*	5.9	4.1	3.1	3.1

One may read out from Table 12 that with decreased divider shorter time of transmission is obtained. With the asterisk (*) two slowest frequencies were signed – in the case of nodes working with this particular divider for these two divider values testing under stress caused data loss. It was checked that transmission is successful if sender operates with slower SPI than receiver. Slower SPI beside higher divider may be result of slower microcontroller frequency of operation, what can be imposed by lower voltage supply level. This property can be used for nodes which either mostly send messages (for example electrodes, but from the other hand electrode nodes are going to desire high density of data exchange) or rarely communicate with others (e.g. supply node).

Having measured time of transmission of 10 000 messages short analyze of CAN bus usage can be easily made. Taking into account that preferred dividers were equal or higher than /4 and frames with 8 bytes data fields were transferred:

SPI divider: / 4 (CPU frequency – 16 MHz)

Amount of bits in single message: 108 (together with 8B data field, [*Table 2 CAN Message Data Frame Table 2*])

Amount of messages: 10 000

Generated traffic: $10\,000 \cdot 108b = 1\,080\,000\ b\ \cong 1.08\ Mb$

Time of traffic generation: 3.1 s

CAN bus speed: 1 Mb/s

Average measured capacity: $\frac{\mathbf{1.08}\boldsymbol{Mb}}{\mathbf{3.1}\boldsymbol{s}} = 0.35 Mb/s$

Bus usage: $\frac{measured\ capacity}{teoretical\ capacity} = \frac{0.35\ Mb/s}{1\ Mb/s} = 35\ \%$

Measured bus usage differs only slightly from the one, measured in *6.3 Mechanism of message sending and receiving – testing*, what proves correctness of the computation. Mentioned level of the bus usage may be considered as low. This is the result of many data transfer between microcontroller and CAN controller before complete message is shifted into the CAN bus – before any value may be loaded into an internal register, special command deciding either *writing*, *reading* or *bit set mode* must be sent. In case of *bit set mode* additional transfer of mask is desired. Taking into account that for single CAN message average 20-25 (in the worst case up to 32) SPI transfers have to be carried out, single SPI transmission is time-critical and determines time of whole data transfer between nodes.

6.3.2 Impact of CAN speed for HAL interface capacity – testing

Second stage of testing was checking the extent to which CAN speed configuration influences time of transmission between nodes. Result of testing (with method described in chapter 6.3.1) was presented in Table 12.

Table 12 Impact of CAN speed on HAL interface capacity

CAN speed [kbps]	250	500	1000
Time [s]	6.9	4.4	3.1

As it can be noticed with speed of SPI decreased 4 times in relation to maximum speed, time of transfer increases 2.2 times. Definitely for providing high capacity of the whole PDCP the highest possible CAN speed should be used.

6.3.3 Impact of size of data field on HAL interface capacity – testing

Important aspect of data transmission is size of the data field of a single message. The PDCP protocol specifies functions differing from each other with content, therefore usually data field is smaller than 8 bytes. This should influence the time of transmission. Result of testing in term of size of the data field was presented in Table 13.

SPI divider: /4 (CPU frequency – 16 MHz)

CAN bus speed: 1 Mb/s

Amount of messages: 10 000

Table 13 Impact of size of data field on time of transmission

Amount of data bytes of data field	8	7	6	5	4	3	2	1
Time [s]	3.1	3	2.7	2.5	2.3	2.1	1.9	*

As expected time of transmission depends highly on message content. In case of transfer of one data byte benchmarking program was too fast in comparison with designed system and caused overflow – messages were sent too fast by *while* loop and not received correctly by the second node. However, this was kind of a stress test. In real such situations are not expected.

6.4 Data lossless transmission – testing

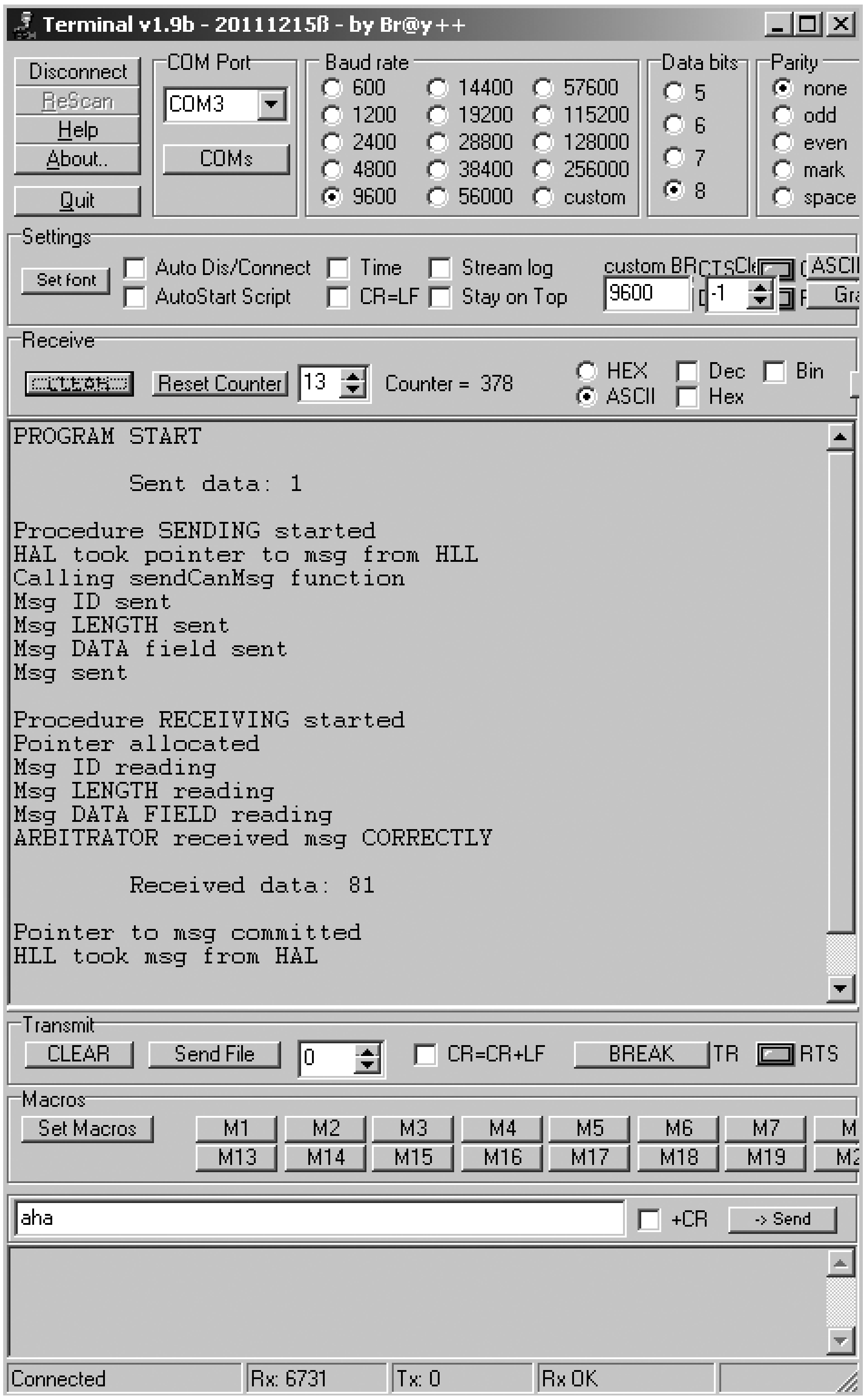
One of the way of testing of data lossless transmission between 2 nodes of a system is sending specified number of data from one node and checking amount of correct received messages by second node. The testing method was described in chapter 6.3 Mechanism of message sending and receiving – testing. Results presented there show, that system works without losses, otherwise time measure wouldn`t be possible. In case of any problems with message delivering, CAN controller retries sending procedure up to 255 times informing HAL about it by triggering external interrupt and refreshing the designed error structure.

6.5 HAL – HLL interface testing

Proper operation of the HAL – HLL interface is based on good cooperation between these two separate parts of code. In brief, the HAL takes responsibility for hardware resources and communication with CAN chip implementing all configuration and data exchange functions, while the HLL deals with message handling and data processing for the PDCP. Description in the chapter *5.1 HAL – HLL interface* and also in chapters *5.2.6 Messages sending* and *5.2.7 Messages receiving* indicates that interface is pretty easy in operation. However, detection of problems encountered during tests and described in the next part was really time-consuming.

The biggest problem, during first stage of combining the HAL and the HLL layer, was microcontroller reset. Debugger (which proved to be indispensible device in prototype testing and error debugging) indicated that a part of code located before *while* loop of benchmarking program was executed infinitely many times, what definitely was a sign of some mistake. First thoughts led to memory stack overflow, but it was hard to prove the suspicion. Fortunately, AVR microcontrollers were equipped with dedicated interrupt handler (called BADISR) called every time an Interrupt Service Routine (ISR) fires with no accompanying ISR handler. Using debugger and elimination of particular parts of the code, it managed to find wrong function calling order which generated “vicious circle”, stack overflow and microcontroller reset. The task of finding the source of described reset was hard, because MCU status register (MCUSR) didn`t indicate any unforeseen watchdog reset (which was used in software reset forcing), brown-out detection reset or other common sources of reset, which firstly were considered.

For interface testing similar method to these from previous chapters was used. In Fig. 17 output terminal from one the tests was presented.



COM3
Baud rate: 9600
Data bits: 8
Parity: none
Counter = 378
ASCII

```
PROGRAM START

        Sent data: 1

Procedure SENDING started
HAL took pointer to msg from HLL
Calling sendCanMsg function
Msg ID sent
Msg LENGTH sent
Msg DATA field sent
Msg sent

Procedure RECEIVING started
Pointer allocated
Msg ID reading
Msg LENGTH reading
Msg DATA FIELD reading
ARBITRATOR received msg CORRECTLY

        Received data: 81

Pointer to msg committed
HLL took msg from HAL
```

aha

Connected | Rx: 6731 | Tx: 0 | Rx OK

Fig. 17 HAL-HLL interface testing (terminal output)

System consisting of one Arbitrator and one Device was configured to test proper operation of designed interface between the HAL and the HLL. To make the test more realistic arbitrator was designed to send one message with one byte of the data field with value 0x1 (this corresponds to binding request function code). Device node was expected to receive message correctly, toggle diode, read data byte and sum it logically with value 0x80 (what corresponds to mechanism of response function code generation). Afterwards, Device was intended to send message back to Arbitrator, which checked received value and compared it with 0x81. Correctness of that sequence of data exchange, which involved big part of the HLL mechanism and almost all the HAL mechanism, was indicated by result of mentioned comparison.

The most important stages of data exchange within interface were commented and presented in terminal output in Fig. 17. As it was expected procedures of sending and receiving successfully cooperate with the HLL mechanism designed by *Andreas Nordal*[5]*.* Message received by the Arbitrator was logic sum of data sent and value 0x80, which firstly indicated proper the HAL operation (filter, id settings etc.) and secondly proved proper collaboration of these two layers of interface. Pointers (mentioned in Fig. 17) were used to distribute access to memory containing message fields between two separate HAL and HLL. As a result of that HAL is deprived of buffers sacrificed for messages buffering, which probably would introduce undesired delays and increase memory usage. Layers, using pointers, were able to exchange data between each other in the most efficient way.

[5] master thesis on *Design, Implementation and Testing of High-Level Layers of PDCP for AVR,* under publication

6.6 Canadian implementation – compliance testing

To check correctness of designed software, test based on two nodes, NIMRON card working as the Arbitrator and Canadian board (presented in Fig. 18) as the Device trying to bind itself to the system, was performed.

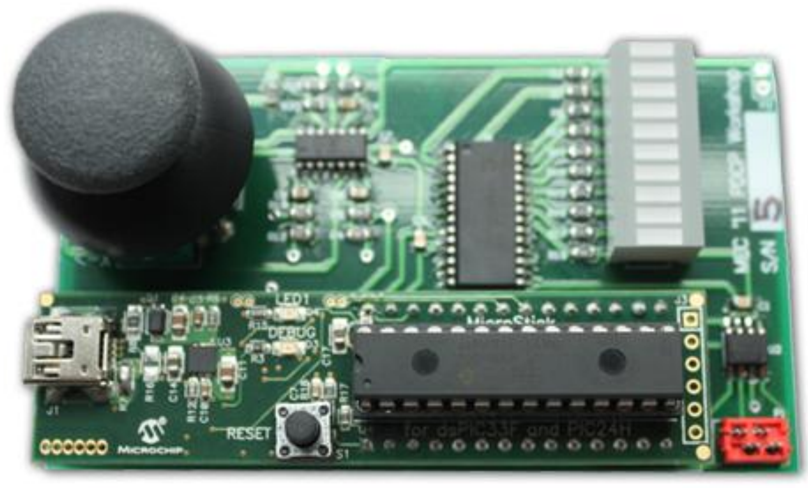


Fig. 18 Canadian PDCP board

Hardware implementation, presented above, consists of Microchip microcontroller on DIP board and extension board designed by *Yves Losier* [1] for the use of the PDCP.

Main purpose of the conducted test was to check whether the HLL worked according to assumptions, thus it was not directly related to the low-level layers of the PDCP. Before testing compliance of PDCP implementation on AVR and on Canadian Microchip node, binding procedure was checked between two NIMRON (AVR) cards. Conducted tests showed, that the device (software implemented mainly in the HLL) was successfully bound to the arbitrator. However, tests with Canadian implementation gave bad results – many bind requests were received, but none of them was handled properly by the arbitrator. Nevertheless, communication between nodes with different PDCP implementation was provided. Received bind requests were visible on terminal, what proved good hardware design, CAN controller configuration and finally, mechanism of the data reception.

7 Suggested future work

The designed software (in cooperation with the HLL part of the PDCP) opens wide possibilities for further development of communication protocol for prosthetics. Some aspects that may indicate some progress directions were listed and described below.

Increasing of interoperability – the main idea of standardization is to enable using modules from different vendors within one prosthetic. The question that should be asked is: in whose interest is the standardization? There is no doubt that people who benefit from prosthetic would support this idea, but it is hard to say if producers would be fans of that process (now the user is left to replacement parts only of prosthesis producer). However, interoperability together with modular approach may give great results. Let`s imagine a situation when one module was broken, but the rest of hand prosthesis works fine. If prosthesis does not support the modular approach and any interoperability is not implemented, the user is forced to exchange whole prosthesis module (palm or electrode module) for the new one from the same producer. Firstly, this is associated with the increased cost (definitely higher than single part of prosthesis module), secondly with a strict desire of elements fit.

If interoperability mechanism along with modularity of prosthesis design (both from electronically and mechanically point of view) were implemented, a defect of one element would not dictate the need for the whole big module exchange. Next, a great advantage could be system reconfiguration after one module exchange. Within the system one special module responsible for keeping all configuration data could be designed.

In case of some module replacement the bus arbitrator basing on configuration data could assign new module to rule played by replaced module. Using “this sort of external memory”, even the controller unit (arbitrator) could be successfully replaced.

Following this idea, one example may be outlined: simple prosthesis configuration could be used to adjust more sophisticated palm module to old control and sensors units, which should make possible simple prosthesis movement like palm flexion-extension. Similar situation could be presented in the reversed order: sophisticated control and sensor modules adjusted to a simple mechanical palm module;

Software optimization – definitely if possible effort should be done on increased speed of protocol operation and higher system capacity. The complex system desires dense data exchange, especially from sensors to control unit and then from control to actuators unit. As far as possible, the speed of data computation and transmission should be increased while reliability maintaining;

Software implementation for another processor families – the software could be implemented on more efficient 32-bits AVR microcontrollers or processors from ARM family. Moreover, taking into account that prosthesis is going to be battery supplied and time of operation is one of the most relevant aspects of convenience for the user, all components used in prosthesis should meet expectations both of processing- and energy-efficiency. Always this is a trade-off between them, however time between battery recharging or replacement should be overlong as far as possible;

Miniaturization – from electronic point of view miniaturization of PCB components may be desired for nodes operation in terms of size and cost.

8 Conclusions

Through this dissertation low-level layers of the Prosthetic Device Communication Protocol (PDCP) for AVR platform were designed, implemented and tested. Moreover, it was made an attempt of assigning PDCP functions to particular layers of ISO/OSI model (Table 5).

The PDCP implementation was divided into two parallel master projects and two layers, which during implementation were called the Hardware Abstraction Layer (HAL) and the High Level Layer (HLL). During start-up indispensible devices occurred debugger (*JTAGICE mkII*) and UART – USB converter, that was used to display variable values or messages on *RS-232* terminal program. It is hard to imagine final interface implementation without mentioned devices.

Design of the HAL was based on external interrupts generated by CAN controller and two software interrupts triggering the message exchange. Communication with mentioned chip was provided with 4-wired SPI interface. Only these hardware resources (three external interrupt pins and SPI) were used to implement the HAL. It can be concluded that system is event-based, what certainly decreases computational load of microcontroller.

For providing the highest possible level of portability and modularity of designed the HAL several steps have been taken. Firstly, file masking registers and ports, delivered by microcontroller header file, was designed. Secondly, code snippets responsible for hardware resources were collected into several functions which might be easily modified if needed (designer is not forced to browse every project file and look for code to refresh). Furthermore, all language and coding structure listed in *6.2 Code portability – testing* were avoided. Moreover, program structure gave desirable hardware transparency for the HLL and application layers.

Finally, the least possible amount of hardware resources was used for the HAL implementation, what should increase software portability and open wide possibilities for application designer.

To provide cooperation between codes implemented by two different authors the HAL – HLL interface basing on callback functions was implemented. Thanks to separation between layers related mainly to hardware and these implementing the PDCP, in case of necessity of change of one of them (hardware exchange or refreshed idea of communication protocol), only adequate layer had to be updated. This solution definitely increased software both modularity and portability.

Results of conducted tests showed, that trial of AVR implementation of the PDCP designed by *Yves Losier* was successful. Although not all functionalities were provided and tested by the HLL, but tests both the HAL, interface between these two layers and the HAL together with the HLL brought satisfactory results in the area of binding (between nodes based on NIMRON boards), which probably in the near future might be extended to full PDCP implementation. However, implementation of binding procedure and other functions (HLL) delivered by the PDCP were not task of this dissertation.

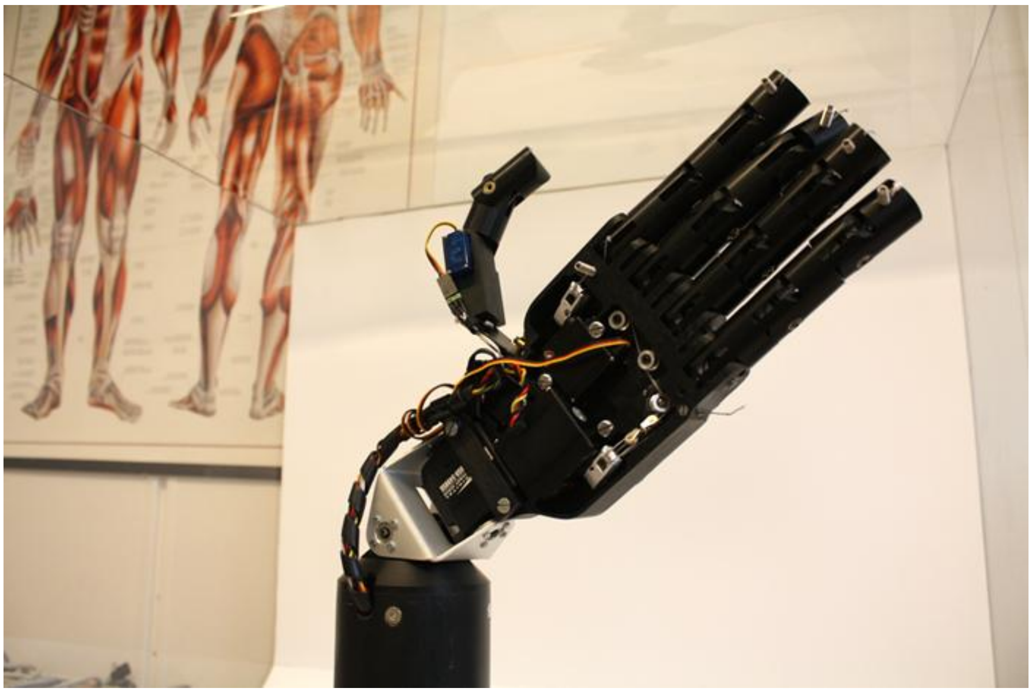


Fig. 19 Handle model - prosthesis designed at NTNU

In Fig. 19 example of prosthesis designed at NTNU was presented. Great advantage of the PDCP is versatility. It provides communication mechanism between parts of prosthesis responsible for different activities like gathering data (EMG sensors), control or supply. These parts may be almost identical for many prostheses from communication point of view, what makes it universal. Handle model from Fig. 19 is only an example of prosthesis which could benefit from designed communication protocol.

It is obvious that designed interface creates possibilities of future development in the area of hardware and software. From the hardware point of view several aspects like trade-off between power- and energy-efficiency or components limitation and miniaturization should be taken into account. Software could be developed to provide protocol implementations for other hardware platform and chips to increase range of applications. Moreover, as it was mentioned in chapter *7 Suggested future work* emphasis should be placed on prostheses modularity.

9 Bibliography

[1] Yves Losier - *Prosthetic Device Communication Protocol for the AIF UNB Hand Project (obtained from author)* [pdf]

[2] Roman Obermaisser, Armin Kanitsar - *Documentation version 1.2. TTP/A Master Slave Application. Axon Bus (obtained from authors) [pdf]*

[3] Øyvind Stavdahl, GeirMathisen – *A Bus Protocol for Intercomponent Communication in Advanced Upper-Limb Prosthesis [pdf]*

[4] Øyvind Stavdahl, Heir Mathisen – *An Intra-Prosthesis Communication Bus: Now is the Time to Standarise![pdf]*

[5] Peter J. Kyberd, Adrian S. Poulton, Leif Sandsjö, SteweJönsson, Ben Jones, Dawid Cow – *The ToMPAW Modular Prosthesis: A Platform for Research in Upper-Limb Prosthetics*JPO Journal of Prosthetics and Orthotics, Volume 19, Number 1, 2007

[6] *AN713*, Microchip – Controller Area Network (CAN) Basics [pdf] *(http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en011694, date of download: 04.05.2012)*

[7] *AT90USB1287*, datasheet [pdf] – product specification by Atmel (http://www.atmel.com/devices/at90usb1287.aspx?tab=documents, date of download 10.02.2012)

[8] *MCP2515*, datasheet [pdf] – product specification by Microchip (http://www.microchip.com/wwwproducts/devices.aspx?dDocName=en010406, date of download: 15.02.2012)

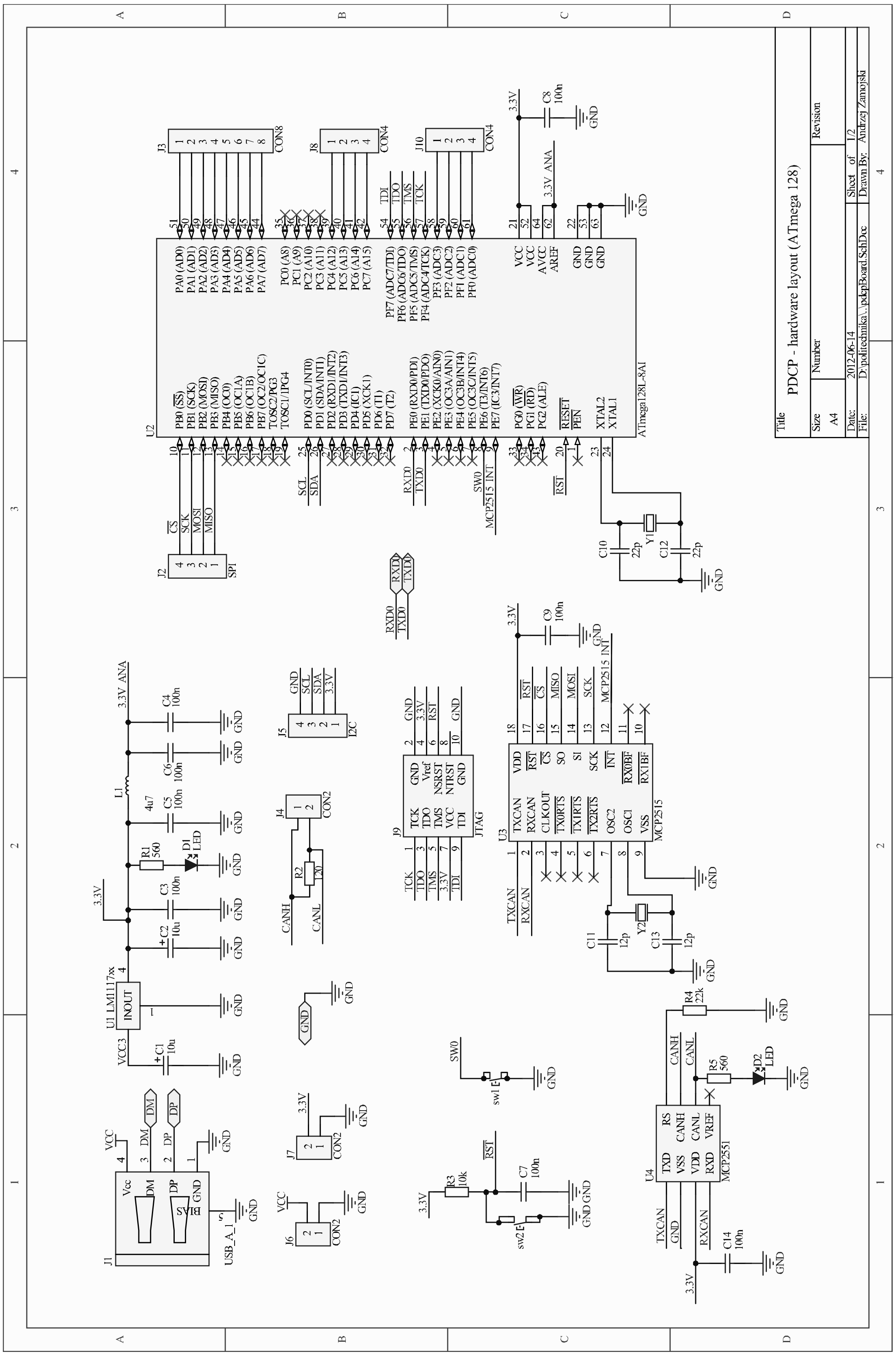
[9] *MCP2551*, datasheet [pdf] – product specification by Microchip (http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en010405, date of download: 10.02.2012)

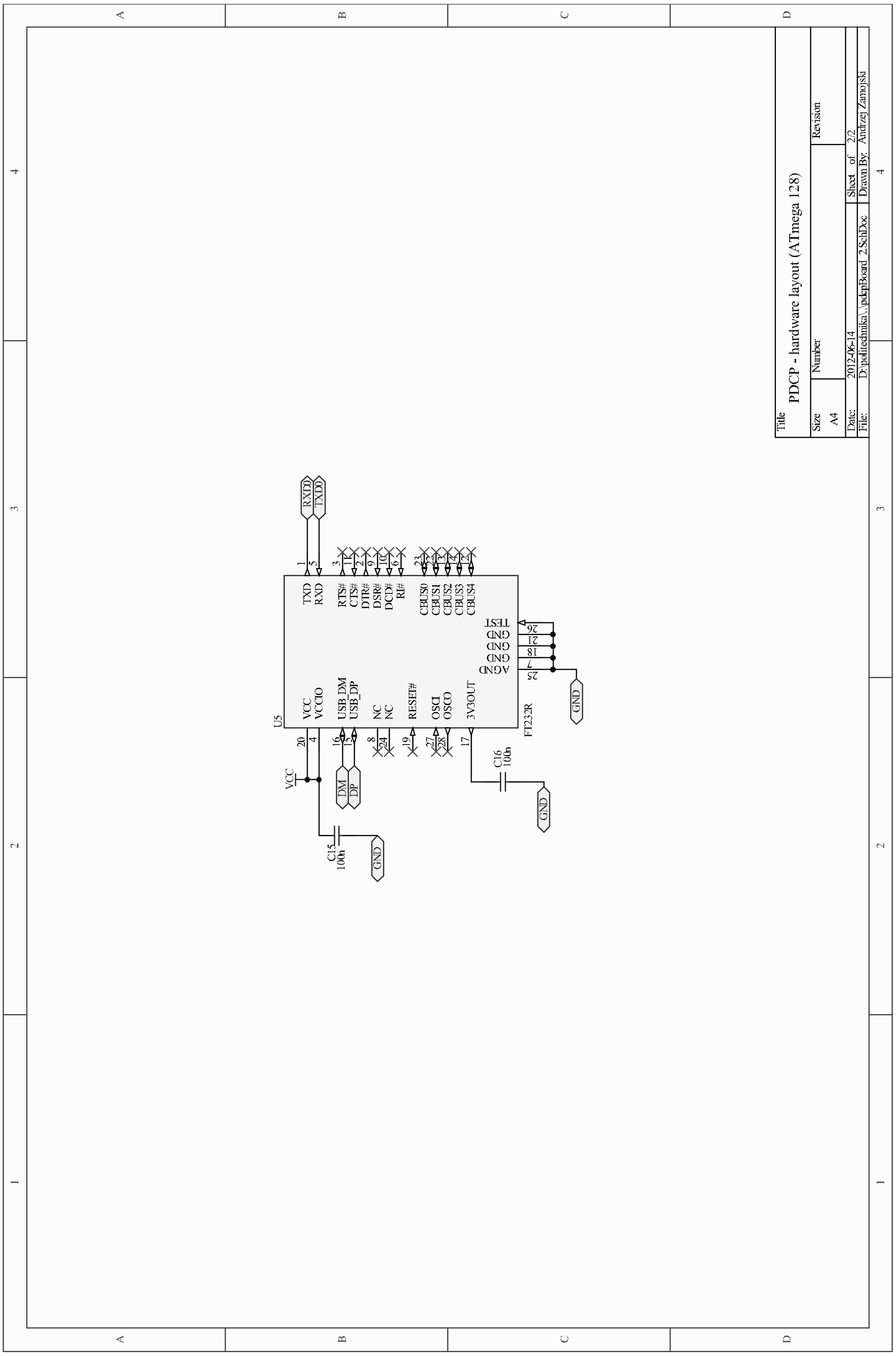
[10] Ole Johnny Borgersen, Marius Lind Volstad – *USB Multifunction Board Users Guide [pdf]* (http://www.nimron.no/P1000/, date of download: 10.02.2012)

10 Attachments

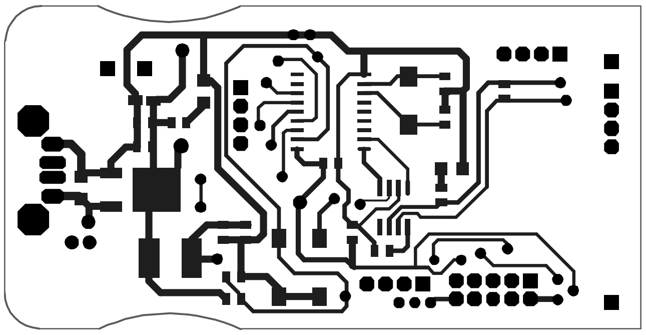
In this chapter electric schematics, layout of designed PCB and Bill of Materials was presented. Files, which these figures were generated from, were recorded and attached with CD.

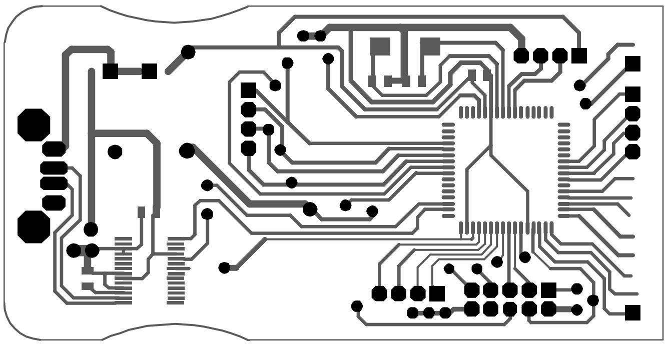
10.1 Printed Circuit Board – electric schematic

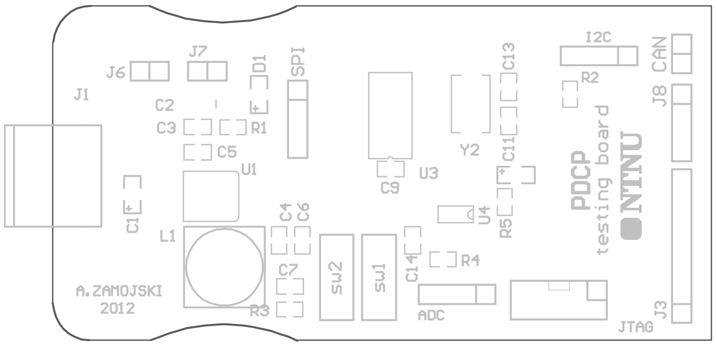



U5
FT232R
TXD
RXD
RTS#
CTS#
DTR#
DSR#
DCD#
RI#
CBUS0
CBUS1
CBUS2
CBUS3
CBUS4
TEST
GND
GND
GND
AGND
VCC
VCCIO
USB_DM
USB_DP
NC
NC
RESET#
OSCI
OSCO
3V3OUT
RXD
TXD
VCC
DM
DP
GND
C15
100n
C16
100n
Title
PDCP - hardware layout (ATmega 128)
Size
A4
Number
Revision
Date: 2012-06-14
Sheet 2 of 2
File: D:\politechnika\..\pdcpBoard 2.SchDoc
Drawn By: Andrzej Zamojski

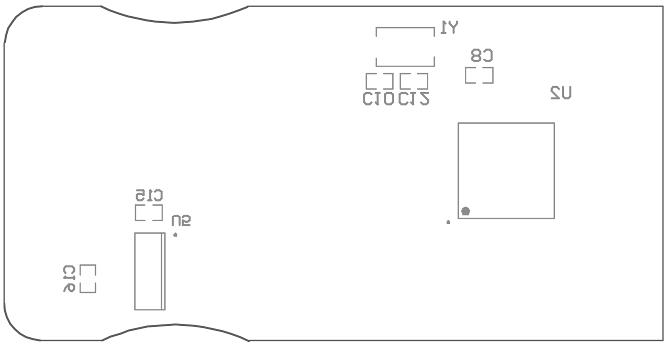
10.2 Printed Circuit Board – layers


Top layer (without polygons)


Bottom layer (without polygons)



Top overlay layer



Bottom overlay layer

10.3 Printed Circuit Board – Bill Of Materials (BOM)

List of materials needed for PCB assembling was presented below.

Description	Designator	Footprint	Quantity	PartType
Polarized Tantalum Capacitor	C1, C2	SMDA	2	10u
Capacitor	C3, C4, C5, C6, C7, C8, C9, C14, C15, C16	C0805	10	100n
Capacitor	C10, C12	C0805	2	22p
Capacitor	C11, C13	C0805	2	12p
LED	D1, D2	SMDA	2	LED
USB-A	J1	USBA-G	1	USB_A_1
Connector	J2	SIP-4	1	SPI
Connector	J3	SIP-8	1	CON8
Connector	J4, J6, J7	SIP-2	3	CON2
Connector	J5	SIP-4	1	I2C
Connector	J8, J10	SIP-4	2	CON4
JTAG Header for AVR MCU's	J9	IDC10	1	JTAG
L	L1	L_SMD_1	1	4u7
Resistor	R1, R5	R0805	2	560
Resistor	R2	R0805	1	120
Resistor	R3	R0805	1	10k
Resistor	R4	R0805	1	22k
SWITCH 6x6mm SPST-NO	sw1, sw2	TACT 6x3.5mm SMI	2	Swich micro 6x6
Voltage stabilizer 3.3V	U1	TO-252	1	LM1117xx
8-Bit AVR Microcontroller with 1	U2	64A_N	1	ATmega128L-8AI
CAN Controller	U3	SOIC18	1	MCP2515
CAN Transceiver	U4	SOIC-8	1	MCP2551
USB UART IC	U5	SSOP-28	1	FT232R
Crystal Oscillator	Y1, Y2	SMD 12SMX A	2	16MHz

11 Appendix A

For more convenient HAL handling the most important aspect were listed in Appendix A.

Property	Description	File
Bus mode	Node mode determination (CONFIG_BUS_MODE): BUS_ARBITRATOR or BUS_DEVICE	config.h
SPI transmission	Properties of SPI determined in function *SPI_INIT_STATUS initSpi().* transmission - *uint8_t transmitSpi(uint8_t data).*	functions.c
CAN transmission	CAN initialization specified in function *CAN_INIT_STATUS initCan(uint8_t id)*	functions.c
I/O configuration	Function void initIOPort() calls functions: *void initIOPort_appDesigner(),* *void initSPIPorts()* and *void initINTPorts().*	functions.c
Interrupt initialization	All interrupt initialization is executed within *void initInterrupts().* **Particular attention should be paid to settings sources of external interrupt (EICRA, EICRB and EICRC registers, configuration is not made automatically using prefixed CONFIG directives from *config.h*.** Function *initInterrupts_appDesigner()* is sacrificed for application designer. However, queue of function calling causes, that initialization improper from the PDCP point of view will be overwritten by interface initialization.	functions.c

Property	Description	File
Software interrupt	Software interrupts are based on external interrupts. Default configuration of external interrupt: TRIGGERING: any edge of pin change, PORT: specified by *INT_SENDING_NUM and INT_RECEIVING_NUM* defined in *config.h.*	functions.c irqHandlers.c config.h
CAN controller	Default ID: Arbitrator – 0x01, Device – 0xFF Default mask register settings: Arbitrator – 0x00, Device – 0xFF Default filter register settings: Arbitrator – 0x00, Device – depending on ID and 0x00 for broadcast messages (FILTER 2) Default transmitting register settings: TXB0 – highest priority, TXB1 and TXB2 - intermediate	functions.c
EEPROM handling	EEPROM reading - *uint8_t readEEPROM (uint8_t address)* EEPROM writing – *void writeEEPROM (uint8_t address, uint8_t data).*	functions.c
Watchdog	*void wdtEnable (uint8_t time)* *void wdtDisable ()* For software reset following function is used: *void forceReset ()* – this function uses watchdog timer. Program initialization should disable manually watchdog timer, otherwise reset is supposed to happen.	functions.c
Message sending	*void sendCanMsg (struct can_msg* msgToSend, uint8_t regToSend)* Code responsible for sending triggering: *void hal_msg_poll()* *ISR (CONFIG_INT_SENDING).*	irqHandlers.c
Message receiving	*void receiveCanMsg (uint8_t regRec)* Code responsible for receiving triggering: void *hal_msg_take();* *void triggerSoftwareInterrupt (uint8_t port)* and *ISR (CONFIG_INT_RECEIVING).*	irqHandlers.c functions.c

12 Appendix B

Project was designed, compiled and tested using AVR Studio 4.0 together with WinAVR library. Before code uploading it is very important to remember about following aspects:

Correctness of frequency of microcontroller with project configuration options: mentioned microcontroller frequency should be set in *config.h* file, while project configuration options for AVR Studio 4.0 are easily available in: *Project->Configuration Options-> General;*

Setting appropriate microcontroller model inside of AVR Studio (or other development environment) – for AVR Studio: *Project->Configuration Options-> General*

Refreshing microcontroller masking file – file *uCmaskFile.h;*

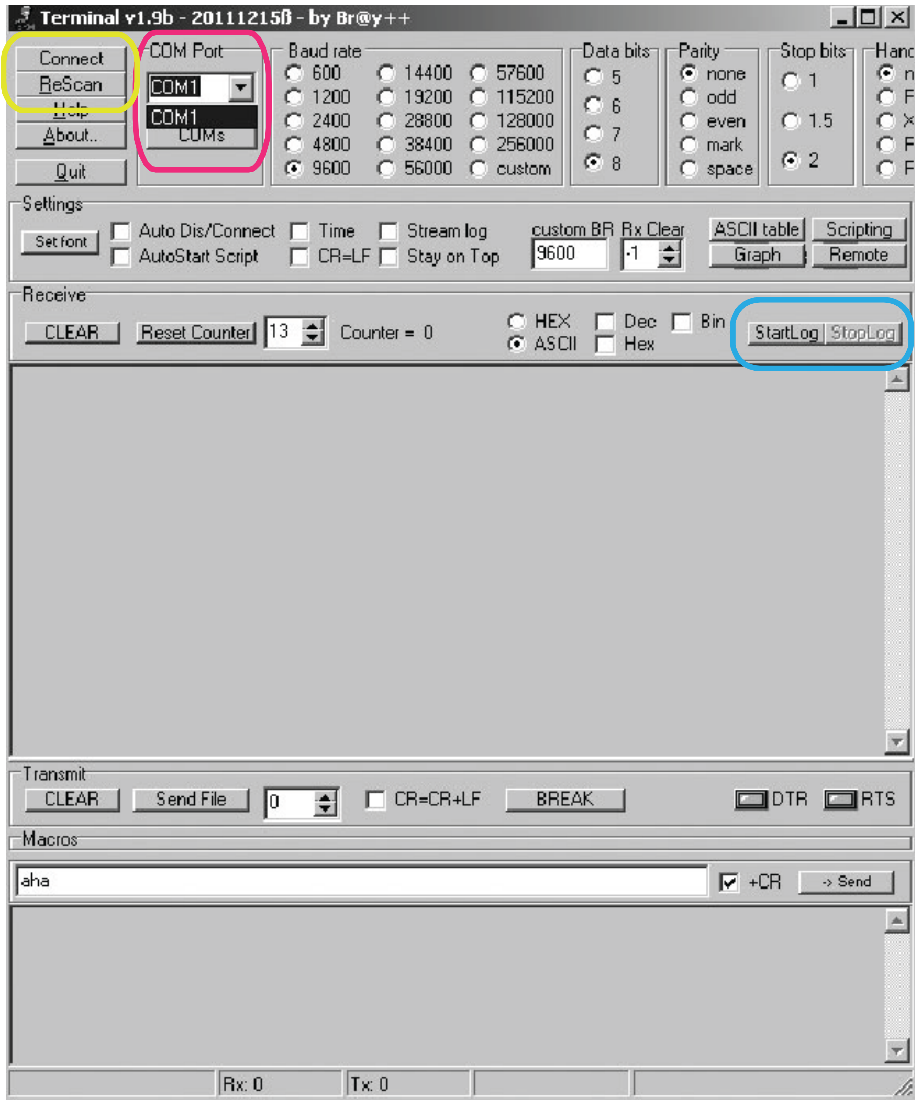
Setting all hardware pins signed with prefix HW_ - in file *config.h*;

Setting all configuration data signed with prefix CONFIG_ - in file *config.h*

Browsing code in case of problems – designing totally portable code for huge diversity of microcontrollers is almost impossible task. Trial of uploading code on the new card showed, that even very similar microcontroller models differed from each other very slightly, what made code portability not easy task.

13 Appendix C

As it was mentioned in the thesis for debugging and testing *Terminal 1.9b*[6] was used. For more comfortable usage main features of the program were shortly described in the appendix.



Baud rate: 9600
Data bits: 8
Parity: none
Stop bits: 2

[6] *https://sites.google.com/site/terminalbpp/*

After each! connection a device to USB port user should *ReScan* ports (yellow "cloud"). Just after that in *Com Port* section all available ports should be displayed and the one which is desired device should be chosen. After a port choosing, the *Connect* button may be pressed and if device is supposed to send messages which should be displayed in *Receive* section. It should be emphasized that designed PDCP software was adjusted to default settings of the Terminal program (speed, data bits, parity and stop bits). This results in statement, that if software is correctly configured, Terminal should display messages both from arbitrator and device node without any special additional effort made.

The program enables also logging, which may be very useful in case of the PDCP. To logging process two buttons (marked with blue cloud) were created. For logging start *StartLog* button <u>should be pressed before any transmission</u> from/to device. When transfer is completed, *StopLog* button should be pressed and logging file will be saved in localization pointed by the user.

List of figures

List of tables

CD description

Short description of CD content has been presented below.

\code\ - designed software

\!readme.txt

\HAL_testing_code\ - directory containing readme file, project files, *.hex* files of arbitrator and device for only HAL testing

\HAL+HLL\ - directory containing project files, *.hex* and *.elf* files ready for uploading for testing HAL together with HLL layer basing on BINDING operation

\HAL_testing_code_new_hardware\ - directory containing project files for new hardware platform – precise explanation of encountered problems in txt files

\HAL+HLL\documentation\ - Doxygen documentation of the designed software

\documents - contains pdf files, documentations

at90usb1287.pdf

axonBus.pdf

canPhysicalLayer.pdf

Kyberd2007-JPO-The_ToMPAW_Modular_Prosthesis__A_Platform_for.pdf

mcp2515.pdf

mcp2551.pdf

Obermeisser2000-Master Slave App for AxonBus.pdf

Stavdahl2005-MEC'05 POSTER-A Bus Protocol for Intercomponent Communication in Advanced Upper-limb Prostheses.pdf

Stavdahl2005-MEC'05-A Bus Protocol for Intercomponent Communication in Advanced Upper-limb Prostheses.pdf

P1000_user_guide-3.pdf

\pcbDesign - PCB projects designed in *Altium Designer Winter 2009*

\electric_schematics\ - directory containing electric schematics of designed PCB boards in *pdf* format

!readme.txt

\pcb_converter_UART_USB\ – UART-USB converter project

\pcb_pdcpBoard\ - pdcp board project (ATmega 128, ext CAN controller)

\pcb_pdcpBoard_at90can128\ - pdcp board project (AT90CA128)

\pcb_converter_UART_USB_prof\ – UART-USB converter project (with overlays)

\pdcpLibrary\ - library of components used in every following project

\thesis - thesis documents

masterAssignment_AZamojski.pdf

masterThesis_AZamojski.pdf