NTNU – Trondheim
Norwegian University of
Science and Technology

Device Profile layer for PDCP

Øyvind Rønningstad

Master of Science in Engineering Cybernetics
Submission date: July 2013
Supervisor: Øyvind Stavdahl, ITK
Co-supervisor: Anders Fougner, ITK
Terje Mugaas, SINTEF

Norwegian University of Science and Technology
Department of Engineering Cybernetics

NTNU
Det skapende universitet

Device Profile layer for PDCP

Øyvind Rønningstad

Master thesis
Spring 2013

Assignment

The PDCP protocol is an emerging standard for the interconnection of electronically controlled prosthetic components. The present protocol version lacks functionality for hot-swapping of devices, as well as general interoperability, without the need for explicit reconfiguration. The goal of this project is to establish a Device Profile layer for the PDCP that facilitates this functionality.

1. *Give an overview of the theoretical aspects of interoperability, hot-swappability and other data network properties relevant to this assignment.*
2. *Describe a suitable architecture and detailed design of a Device Profile layer for PDCP.*
3. *Implement selected parts of the design, and demonstrate and assess the related aspects of interoperability*

Advisors

Øyvind Stavdahl
Anders Fougner
Yves Losier[1]
Terje Mugaas[2]

1: UNB, 2: SINTEF

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Abstract

The Prosthetic Device Communication Protocol is a CAN-based protocol developed specifically for use within prosthetic devices, for connecting the components of the device, most importantly sensors and actuators. The lower levels of this protocol have already been developed. This thesis focuses on developing a device profile layer, to bring plug-and-play functionality to the protocol.

Specifically, this thesis contains a proposed device profile layer, a description of an implementation of the profile layer, and testing of this implementation. Testing results were largely successful, most of the desired functionality was achieved in the implementation.

Preface

It has been my privilege to work on this project for a year, and I feel I have accomplished what I set out to do. I hope that it will come to use.

I have learned quite a lot, both about the subject, and about working on such a large scale project. I am very glad to have had the experience.

I have very much liked working in the field of medical technology. It has a very wide range of interesting concerns of interweaved human and technical nature, and it is also rewarding to be a part of something that might make someone's life a bit easier.

Many people have been a part of this work, and I would first like to thank Øyvind Stavdahl for his enthusiastic and highly encouraging approach to this project. He had the idea, and has been following my work with interest despite his busy schedule. He has also taken the time for lengthy discussions that have been highly fruitful. He has been of great help and encouragement. Secondly, I would like to thank Anders Fougner for many good conversations. He has also taken the time out to help a great deal. Thirdly, I want to thank Yves Losier and Adam Wilson for great cooperation. They have gone out of their way to make sure I had what I needed for my work. They have, in addition to many helpful e-mails, taken the time for Skype conversations and provided me with their bus arbitrator implementation and two extra boards for my testing. Their help has been invaluble. Thanks also to Terje Mugaas for his help this fall and for making himself available this spring for questions.

Thanks to Ole Morten Haaland for his thoughtful and constructive criticism on my report, and thanks to my wonderful wife for all love, encouragement and help.

Øyvind Rønningstad

July 2, 2013
Trondheim

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Prosthetic Device Communication Protocol is an emerging, open source digital communication protocol for use in prosthetic devices. The motivations behind the creation of the protocol are many; among them are:

- Moving prosthetic device technology into a more digital world.
- Standardization of prosthesis function to the point where components from different manufacturers are interchangeable.
- Lowering the cost of prostheses by simplifying and streamlining design, thereby also making it easier to make custom solutions.
- Making prostheses more robust by decreasing the amount of wiring.

The low level layers of the protocol are maturing, and the focus of this thesis is on implementing higher level functions, specifically *device profiles*. The motivation behind adding device profiles is interchangeability of components that perform equal or similar functions, and interoperability of components that complement each other, but aren't explicitly set to communicate.

The work conducted for this thesis is a continuation of work the author did in the fall of 2012. This previous work consisted of a report with a theoretical discussion of how to implement a device profile layer in PDCP. The main chapter of that report is reproduced in appendix B, and a background section on lower-level PDCP is also reproduced in chapter 2.

This thesis contains three main chapters: (1) A chapter (chapter 4) expanding and refining the thoughts from appendix B into the beginnings of a device profile layer specification. (2) A chapter (chapter 5) describing a

simple software and hardware implementation. And (3) a chapter (chapter 6) describing a suite of tests conducted on the implementation and its results. In addition, chapter 3 examines more generally the concept of interoperability in communication protocols and in device profiles. Appendix A is a glossary which explains how different expressions are used in the report.

The report should come with an attachment file, the contents of which are itemized in appendix C.

Main Findings

The main findings of the work are that making a useful device profile layer for PDCP is likely possible without too much complexity - much of the desired functionality is present in the rough implementation - but that the implementation deviates from a production-grade actual prosthesis implementation in several important ways.

Chapter 2

The Prosthetic Device Communication Protocol

This is chapter 2.2 of "Possibility Study of Implementing a Device Profile Layer in PDCP." by the author. It describes the lower levels of PDCP. It is reproduced here because of its relevance for understanding the rest of the report.

The Prosthetic Device Communication Protocol (PDCP) is a bus communication protocol designed specifically for use in prostheses. PDCP is built on top of the Controller Area Network (CAN) protocol. Figure 2.1 is an example of a fully configured PDCP network.

PDCP is still a work in progress and the lower layers of the protocol, which a device profile layer will build on, are not completely specified. There is also no definitive document containing the current specification. The PDCP documentation consists, as of now, of:

- *Prosthetic Device Communication Protocol for the AIF UNB Hand Project*, which describes basic function, and PDCP's relationship to CAN.
- *PDCP Info (2011 05 04)*, which describes the implementation of data channels.
- *AIF2 System Data Capture (2012 02 21)*, which is a sample capture of the bus communication during setup after power-on.[1]

[1]It should be noted that the values used in *AIF2 System Data Capture (2012 02 21)* for "Device Type and Profile" and "Channel Type and Profile" are dummy values that do not have a specified meaning as of yet.

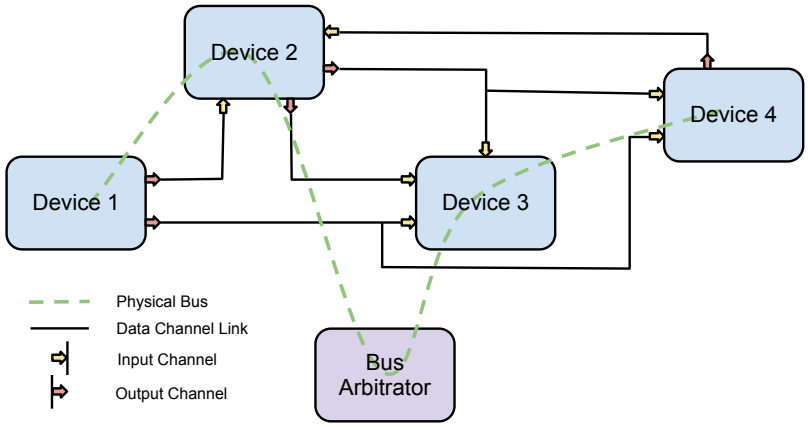


Figure 2.1: Example network structure of PDCP, including both physical and logical links. Not drawn: All devices have an inherent logical link to the Bus Arbitrator.

This section is based on these documents, in addition to correspondence with one of the protocol's creators, Yves Losier of UNB.

2.1 Bus Arbitrator

PDCP specifies that one node must have the role of "bus arbitrator", which oversees, and largely controls, the communication on the bus.

The bus arbitrator handles requests to join the bus, distributes node IDs, and sets up logical links ("data channel links") between devices. No device can communicate on the bus without being joined[2] to the bus by the bus arbitrator[3], and devices can only communicate directly with one another after the bus arbitrator has set up a data channel link between them.

The bus arbitrator role can be filled by a dedicated device or by a device that already performs some other function, e.g. the prosthetic hand.

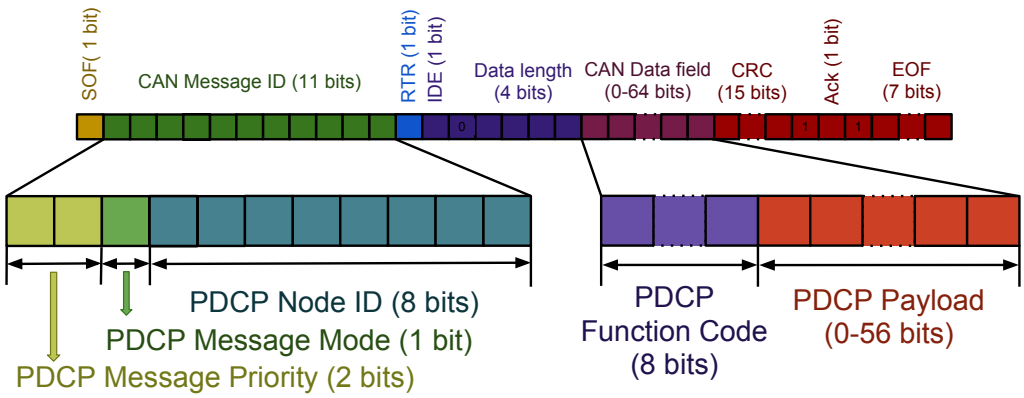


Figure 2.2: Bit layout of a PDCP message[2], in the context of a CAN message[5]. The function code field is only used in communication with the bus arbitrator.

2.2 Message Structure

PDCP uses the CAN base frame format, which means that it uses the standard 11 bit message identifier field. The message ID is divided into 3 parts: The **message priority**, **message mode**, and **node ID** fields, as seen in figure 2.2.

The **message priority** is a number (0-3) that specifies the urgency of the message. 0, 1 and 2 mean "high priority", "medium priority" and "low priority" respectively, while 3 is used only when a device attempts to perform the binding process. Except for level 3, priorities can be used at will, but the common practice is to use high priority for data streaming on channel links and for device beacons, and medium priority for everything else[4].

The **message mode** specifies if the message originates from the bus arbitrator (1) or from another device (0).

The **node ID** specifies either the target device, the sending device, or the source output data channel. See table 2.1.

CAN's arbitration scheme works in such a way that if two devices start transmitting at the same time, the message with the *lowest* message ID gets through without delay, while the other message must be aborted and retransmitted at a later time. This means that PDCP messages are pri-

[2] "Binding"
[3] The exception being *Bind Device Requests*.
[4] This is reflected in [4].

From	To	Message Mode Field	Node ID Field
Bus arbitrator	Device	1 (from bus arbitrator)	(Target) device
Device	Bus arbitrator	0 (from device)	(Source) device
Device	Device	0 (from device)	(Source) output channel

Table 2.1: Use cases of message modes and node ID types.

oritized first by priority, then by message mode, then by node ID.

In addition to this, in device-bus arbitrator communication, the first byte of the CAN data field is reserved for a PDCP function code. An overview of the predefined function codes can be found in table 2.2, with corresponding response codes in table 2.3.

In device-device communication using data channels, the format of the entire 8 byte data packet can be tailored to the needs of the channel, i.e. there is no function code.

Function Code	Description	Message Size (bytes)	Sender	Recipient	Response Function Code
0x01	Bind Device Request	7	Device	Bus Arbitrator	0x81
0x03	Get Device Parameter	3	Bus Arbitrator	Device	0x83
0x04	Set Device Parameter	4-7	Bus Arbitrator	Device	0x84
0x08	Suspend Device	3	Bus Arbitrator	Device	0x88
0x09	Release Device	1	Bus Arbitrator	Device	0x88
0x0A	Device Beacon	1	Either	Either	N/A
0x0B	Reset Device	1	Bus Arbitrator	Device	0x8B
0x0C	Configure Get Bulk Data Transfer	5	Bus Arbitrator	Device	0x8C
0x0D	Configure Set Bulk Data Transfer	5	Bus Arbitrator	Device	0x8D
0x0E	Bulk Data Transfer	3-8	Either	Either	0x8E
0x0F	Update Data Channel	2	Device	Bus Arbitrator	0x8F

Table 2.2: List of all function codes of the PDCP protocol[2]. Missing function codes are deprecated.See table 2.3 for a list of responses. Tables 2.4 to 2.6 contain the message structures of some of the message types.

Response Function Code	Description	Message Size (bytes)	Sender	Recipient
0x81	Bind Device Request Response	7	Bus Arbitrator	Device
0x83	Get Device Parameter Response	3	Device	Bus Arbitrator
0x84	Set Device Parameter Response	4-7	Device	Bus Arbitrator
0x88	Suspend Device Response	3	Device	Bus Arbitrator
0x89	Release Device Response	1	Device	Bus Arbitrator
0x8B	Reset Device Response	1	Device	Bus Arbitrator
0x8C	Configure Get Bulk Data Transfer Response	5	Device	Bus Arbitrator
0x8D	Configure Set Bulk Data Transfer Response	5	Device	Bus Arbitrator
0x8E	Bulk Data Transfer Response	3-8	Either	Either
0x8F	Update Data Channel Response	2	Bus Arbitrator	Device

Table 2.3: List of all response function codes of the PDCP protocol[2]. Missing function codes are deprecated.

Function Code 0x01 - **Bind Device Request**							
Data0	Data1	Data2	Data3	Data4	Data5	Data6	
0x01	Device Vendor ID		Device Product ID		Device Serial Number		

Function Code 0x81 - **Bind Device Request Response**							
Data0	Data1	Data2	Data3	Data4	Data5	Data6	Data7
0x81	Node ID	Device Vendor ID		Device Product ID		Device Serial Number	

Table 2.4: Structure of data field of message type 0x01 - Bind Device Request and 0x81 - Bind Device Request Response.

Function Code 0x03 - **Get Device Parameter**							
Data0	Data1	Data2					
0x03	Parameter ID	Channel Index					

Function Code 0x83 - **Get Device Parameter Response**							
Data0	Data1	Data2	Data3	Data4	Data5	Data6	Data7
0x83	Response Code	Parameter ID	Channel Index	Parameter Value (1-4 bytes)			

Table 2.5: Structure of data field of message type 0x03 - Get Device Parameter and 0x83 - Get Device Parameter Response.
Response codes: 0: Failure, 1: Success, 2: Use Bulk Data Transfer.

Function Code 0x04 - **Set Device Parameter**							
Data0	Data1	Data2	Data3	Data4	Data5	Data6	
0x04	Parameter ID	Channel Index	Parameter Value (1-4 bytes)				

Function Code 0x84 - **Set Device Parameter Response**							
Data0	Data1	Data2	Data3	Data4	Data5	Data6	Data7
0x84	Response Code	Parameter ID	Channel Index	Parameter Value (1-4 bytes)			

Table 2.6: Structure of data field of message type 0x04 - Set Device Parameter and 0x84 - Set Device Parameter Response.
Response codes: 0: Failure, 1: Success, 2: Use Bulk Data Transfer.

Function Code 0x0F - **Update Data Channel Request**							
Data0	Data1						
0x0F	Channel Index						

Function Code 0x8F - **Update Data Channel Response**							
Data0	Data1	Data3					
0x8F	Response Code	Channel Index					

Table 2.7: Structure of data field of message type 0x0F - Update Data Channel Request and 0x8F - Update Data Channel Response. the "Channel Index" must refer to an input data channel.
Response codes: 0: Failure, 1: Success.

2.3 Devices, Data Channels and Node IDs

All devices on the PDCP bus have a number of input and output "data channels". A node ID is given (by the bus arbitrator) to each device as well as to each *output* data channel[5]. The bus arbitrator then connects output channels to input channels by giving the input channel the node ID of the source output channel. The device then knows what node ID to listen for. Several input channels can be connected to one output channel.

Each data channel has a channel index local to the device. This index is used to reference the channel on the device. If a device has n channels, they must have indices 1 to n.

Using data channels is the only way for devices to communicate directly to other devices.

2.4 Parameters

Each device must store a number of parameters, both for itself as a device, and for each of its data channels. These parameters are meant to help the bus arbitrator set up the bus.

The bus arbitrator can access these parameters through the "Get/Set Device Parameter" function codes, as described in tables 2.5 and 2.6. The device-wide parameters are stored under channel index 0, while the parameters for each data channel are stored under its corresponding channel index (1-255).

Tables 2.8 to 2.10 list the interpretations of parameter IDs of devices, input channels, and output channels.

The size of these parameters is limited to 4 bytes, but the larger data can be manipulated by using the "Bulk Data Transfer" commands.

2.5 Joining the Bus (Binding)

When a device wants to join the PDCP bus (e.g. after a power-on or reset), it sends the "Bind Device Request" message seen in table 2.4

[5]The distinction between *device* node IDs and *data channel* node IDs is important: Messages with message mode 0 (from device) and a device node ID are (implicitly) bound for the bus arbitrator, while messages with mode 0 and a data channel node ID are (also implicitly) bound for devices with input channels connected to the sending channel. See table 2.1.

Device Parameters	
Parameter ID	Parameter Name
1	Device VID and PID
2	Device Serial Number
3	Device EAN13L
4	Device EAN13H
5	Device FW and HW ver.
6	Device Type and Profile
7	Device Descriptor
8	Device Node Id
9	Number of Data Channels
10	Beacon Interval
11	Time to Wait for Acknowledgement
12	Bind Request Timeout
13	*Not yet specified*
⋮	⋮

Table 2.8: Device-wide parameters.

Input Channel Parameters	
Parameter ID	Parameter Name
1	Channel Type and Profile
2	Channel Descriptor
3	Transfer Type (1: “input”)
4	Data Transfer Enabled Flag
5	Source’s VID and PID
6	Source’s SN and Channel Index
7	Source’s Node Ids
8	*Not yet specified*
⋮	⋮

Table 2.9: Input channel parameters.

Output Channel Parameters	
Parameter ID	Parameter Name
1	Channel Type and Profile
2	Channel Descriptor
3	Transfer Type (2: "output")
4	Data Transfer Enabled Flag
5	Channel Node Id
6	*Not yet specified*
⋮	⋮

Table 2.10: Output channel parameters.

(function code `0x01`) using the requested node ID together with message mode 0 and priority 3 (see table 2.11 for the full data frame).

The bus arbitrator's response (function code `0x81`) can also be found in table 2.4. If the node ID in Data1 of the response does not match the requested ID, the device must send another request using this suggested ID. An ID is not properly granted until the requested ID matches the ID in the response. See also table 2.12 for the frame format of the Bind Request Response.

The *Update Data Channel Request* (table 2.7) includes the channel index of an input channel, and is used to allow the bus arbitrator to find an output channel for the given input channel.

Priority	3
Message mode	0 (*from* device)
Node ID	The requested node ID
Function Code	`0x01` ("Bind device request")
Payload	VID (16 bits)
	PID (16 bits)
	Serial number (16 bits)

Table 2.11: The "bind device request" packet structure.

The setup performed by the bus arbitrator follows these steps:

1. Collect desired parameters from device, such as *Device Type and Profile*.

Priority	e.g. 1 (typical)
Message mode	1 (from *bus arbitrator*)
Node ID	The requested node ID
Function Code	`0x81` ("Bind device request response")
Payload	The suggested/granted node ID (8 bits)
	VID (16 bits)
	PID (16 bits)
	Serial number (16 bits)

Table 2.12: The "bind device request response" packet structure.

Priority	e.g. 0 (typical)
Message mode	0 (from *device*)
Node ID	The node ID of the source output channel
Payload	e.g. sensor readings

Table 2.13: The packet structure of a data channel link packet.

2. Get *Number of Data Channels* parameter.
3. Get *transfer type* (direction) of each channel.
4. For each output channel:
   - Collect desired parameters from channel, such as *Channel Type and Profile* or *Channel Descriptor*.
   - Assign an available node ID.
5. For each input channel addressed by a *Update Data Channel Request*:
   - Collect desired parameters from the channel, such as *Channel Type and Profile*, *Channel Descriptor*, *Source's VID and PID*, or *Source's SN and Channel Index*.
   - Decide which output channel (if any) to assign to this channel[6] and update *Source's Node Ids* with the appropriate device ID and output channel ID.

[6]The only way to do this now, is if the *Source's VID and PID* and *Source's SN and Channel Index* are available.

- If the channel is to be used, set *Data Transfer Enabled Flag* to 1.

6. Set *Data Transfer Enabled Flag* to 1 on all output channels that have an input channel connected to it. This will start data transfer.

2.6 Exchanging Information

The only way for devices to send data directly to one another is through data channel links. When a device sends a packet on a data channel link, it will use the output channel's node ID. The format of the data field is not globally specified, so the meaning of bytes transferred on data channel links can be tailored to the channel. Channel links are uni-directional; packets are sent from an output data channel to one or more input data channels. Table 2.13 shows the packet structure of data channel link packets.

When all data channel links have been set up, devices with output channels can at any time send data to the connected input channels.

Chapter 3

General Protocol Interoperability

This chapter will attempt to regard the problem of creating a fully interoperable profile layer in general terms.

3.1 Existing Theory

Literature is sparse on the topic of interoperability in contexts relevant to this work. The word is used in many fields, both technological and not. Relevant research focuses mainly on interoperability through the use of fieldbuses[1].

3.1.1 Interoperability Testing

Benkhellat, Siebert, and Thomesse[6] have this to say about interoperability:

> *In general, the interoperability is the capability of N heterogeneous pieces of equipment to communicate and to cooperate correctly. The conformance of this equipment to the same profile is a necessary condition to achieve interoperability, but not sufficient. Thus an interoperability definition is as follows: The interoperability is the capability of N heterogeneous*

[1] PDCP can, of course, easily be considered a small-scale fieldbus.

> *systems to conform to a set of OSI standards of the same profile and to communicate according to these standards in an environment representative of the reality.*

Interoperability is essential for a profile to do its duty, thus arose the need for interoperability testing. Methods for conformance testing were already well standardized, but it was found [6] that interoperability "in an environment representative of reality" is not assured through conformance testing, which typically tests if a single device (or even a single layer) conforms to specifications.

In this sense, interoperability is less related to profiles[2] and more related to a holistic or pragmatic way to view any communication protocol regardless of the presence of a profile layer.

No actual interoperability testing procedures were found while searching the literature.

3.1.2 Interoperability Verification

"Producing Compliant Interactions: Conformance, Coverage, and Interoperability"[7] addresses particularly the problem of implementations deviating from specifications. Such deviations can be harmless or harmful. They can for example involve adding an extra step to a particular procedure, or removing a procedure if another procedure exists to reach the same state. In all these cases, a protocol that in its original form ensures both conformance and interoperability can be broken in either conformance or interoperability or both in an implementation. When an implementation is broken, information does not flow as easily or not at all.

In [7], a protocol is represented as a set of states, where agents take actions which cause state transitions. Different agents can perform different roles, and thus have different actions and different states. The complete protocol is then derived from sets of states and sets of actions. In short, if this state model is available, one can determine whether the agents are interoperable by checking for things such as deadlocks. When you view agents as threads, the problem of interoperability becomes a problem of concurrency.

The conformance of an implementation, according to [7], is verified

[2]The "profile" mentioned in the quote is not quite the same as the subject of this thesis.

through comparison with the protocol itself, while interoperability is verified through comparison with other implementations.

3.2 Interoperability Related to Device Profiles

How does the concept of interoperability, as presented in the theory, relate to the concept of device profiles?

The central theme of interoperability seems to be "enabling agents in a network to communicate meaningfully". This is also the purpose of device profiles, but it is really the purpose of any communication protocol. What then is the specific purpose of a device profile layer?

An answer to this is "automation", or in other words, "interoperability in additional circumstances", specifically circumstances of little or no manual configuration.

The definition of device profiles is not set in stone, but in this thesis, the purpose of the device profile layer is to provide something resembling plug-and-play functionality.

Let's define "link" as a medium for interoperation. A prerequisite for meaningful communication, and thus for a link, is a shared context. It is not sufficient to have a physical connection, there must be an agreement on a number of values, or a *shared state*. A device profile layer is a more comprehensive such agreement, which reaches a higher level of abstraction.

The theory of interoperability seems to be more geared towards the lower levels of a protocol, the individual messages. But the essence of interoperability is also relevant in the profile layer.

Translating interoperability to the profile layer, the enemy of interoperation is not the inability to send messages, but rather the inability to establish links. This means that a successful profile layer is one that can provide agents with all the context required for communication. The context required depends on the situation.

In addition to providing context, the profile layer must also provide a framework for comparing, or relating, devices. And in the case of PDCP, it must also sometimes make intelligent decisions regarding the connections to set up. Making these decisions requires a way to compare agents.

These two concepts, context and relation, are separate, though they can be linked, such as if the context is implicit from the relational information. In a profile layer, there needs to be a deciding entity that, using

context and relation, decides which links to set up. In some cases there also needs to be information about dependencies, because in some cases, as in the case of PDCP, there needs to be webs of interconnected devices that together perform all the necessary duties.

3.2.1 Hot-Swapping

Another relevant term, referenced in the assignment text, is "hot-swappability". This is a more specific term which means that a devices can be removed and added without shutting down the system. Removed devices will be registered as gone, and steps will be taken to handle this gracefully. Added devices will be recognized and will start functioning without the system restarting.

Hot-swappability is really just plug-and-play with the additional constraint of not resetting.

Chapter 4

Specification of a Device Profile Layer for PDCP

This chapter can be viewed as a continuation and slight revision of the 3rd chapter of [1], which is also reproduced in appendix B. This continuation will aim to more completely and unambiguously specify a profile layer for PDCP, but will also present some discussions and alternative solutions.

The device profile layer will provide the following behaviors:

1. Storing the usual configuration of the system and setting this up automatically each time the system powers on. This includes redundant storage inside the bus arbitrator.
2. Automatic, intelligent reconfiguration when parts are removed and/or exchanged.
3. Some amount of automatic configuration when the system is not manually configured.

The 3rd point above warrants some rationalization. It could seem unreasonable to implement the most advanced funtionality for the rarest occasions. It is, however, the author's opinion that, firstly, developing this functionality will help bring a deeper understanding of the roles of the different parts of a prosthesis into the protocol. Secondly, solutions to the problems in the third point are likely to be relevant also to the second point. Thirdly, intelligence for automatic configuration can be incorporated into the process of manual configuration to give suggested

configurations and/or set up "obvious" links in increasingly complex systems.

The profile layer will also provide a framework for information about the devices on the bus. This includes

1. Context, or meta-information about the information transmitted on the links.
2. Relational information that allows quantification of the similarity between two channels. This is to pick suitable channels to connect to.
3. Information to infer what links need to be set up to allow expedient information flow.

The device profile layer builds on top of the lower layers (described in chapter 2) developed by Yves Losier, Adam Wilson et. al. See fig. 4.1 for an illustration of the protocol layers involved. Some small changes to the lower layers will be proposed.

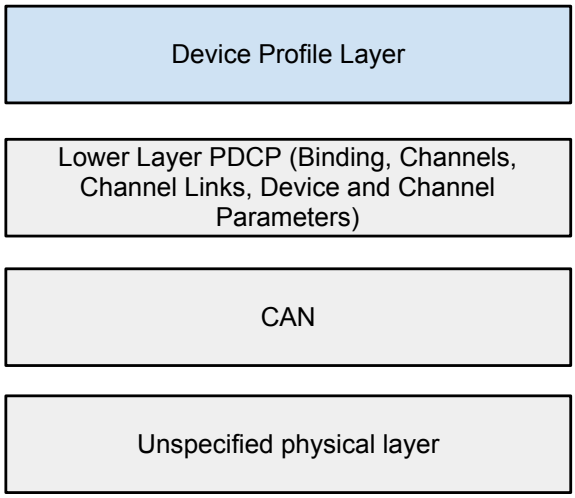


Figure 4.1: Overview of the layers of PDCP, including the profile layer described in this thesis.

In the terms of section 3.2, the entity deciding what channel links to set up will be the bus arbitrator, and the carriers of context and relational information will be the the type and the profile parameters respectively. The carriers of dependency information will be new parameters called

"Required Input Channels", "Desired Input Channels", and parts of the type parameters.

4.1 System Architecture

As presented in [1], the profile layer assumes the structure seen in fig. 4.2. This is a simplification of a general structure for prosthesis control that is described in [8]. An illustration of this from [8] is reproduced in fig. 4.3.

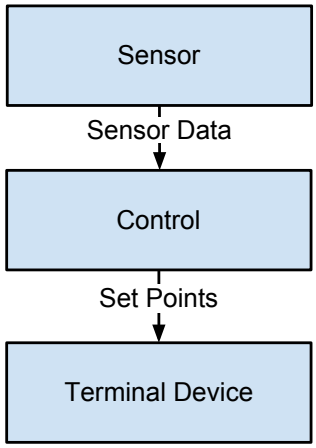


Figure 4.2: Generalization of prosthesis control for use in the device profile layer.

Figure 4.2 shows 3 roles and 2 signal types. In the profile layer, the signal transfer is performed by PDCP channel links. These roles are implicit - implied from the signal types in the following manner.

- A device performs the **Sensor** role if it presents one or more **Sensor Data** output channels.
- A device performs the **Control** role if it presents one or more **Sensor Data** input channels and/or **Set Points** output channels.
- A device performs the **Terminal Device** role if it presents one or more **Set Points** input channels.

Multiple roles can be filled by the same device as illustrated in figs. 4.4 and 4.5. In the case of fig. 4.4, input and output channels for set points are not needed. Still, adding set point input channels would make the device more flexible, allowing the use of other control devices. In fig. 4.5,

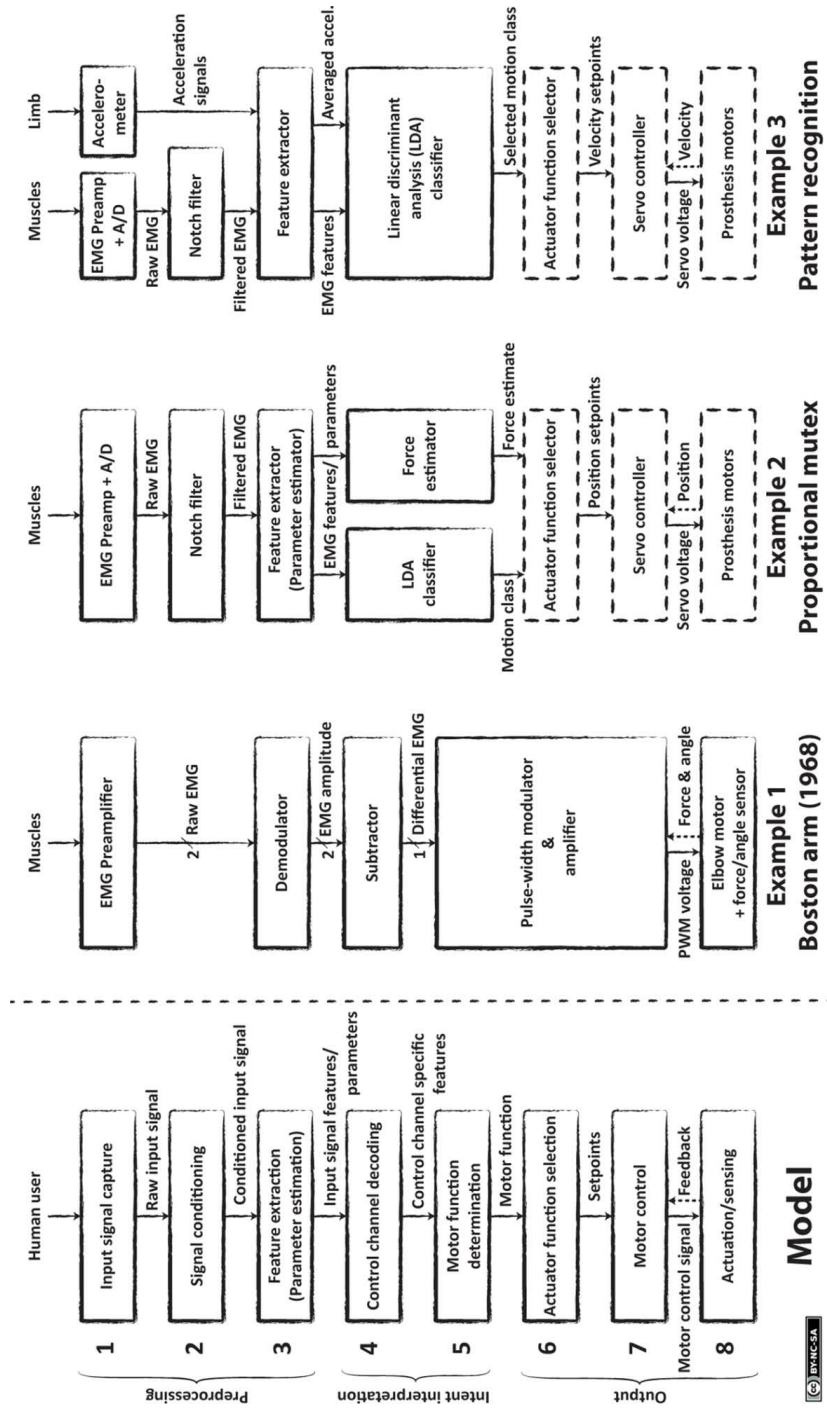


Figure 4.3: Conceptual model of a general prosthesis control scheme, as proposed by Fougner et al. [8].

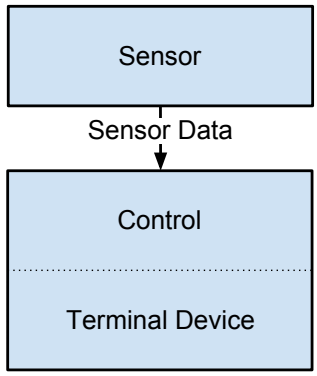


Figure 4.4: Variation of fig. 4.2 where the control and terminal device roles are performed by the same device.

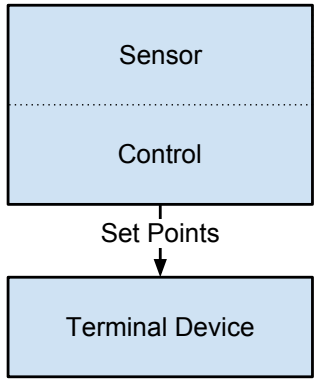


Figure 4.5: Variation of fig. 4.2 where the sensor and control roles are performed by the same device.

the Sensor/Control device is required by the profile layer to present sensor data output channels. The motivation behind this requirement is to increase interoperability by ensuring that the data is available on the bus in the rawest state, and by preventing situations where e.g. a Sensor/ Control and a Control/Terminal Device cannot communicate even though they have all roles between them. Any device that performs the Sensor role, must provide one or more sensor data output channels.

4.1.1 Terminal Channels

To allow for systems like that in fig. 4.4, there is the concept of a **Terminal Channel**. A terminal channel is an input channel that receives data that is used to control a terminal device. A set point input channel is always a terminal channel, and a sensor data input channel is a terminal channel if the device is a combined Control and Terminal Device, and the data received on the channel is used to control the actuation.

4.1.2 The Philosophy Behind the Channel

The channel is the primary unit of communication, the *agent*, in the profile layer. The system is primarily seen as a collection of channels. How many devices are present and which owns which channel is of little importance in practice[1].

4.2 Channel and Device Parameters

4.2.1 Device Type and Profile

The device type and profile parameters are not utilized.

4.2.2 Channel Profile

In the profile layer, the channel profile is determined from a tree structure as explained in [1][2]. An example of such a tree structure can be found in fig. 4.6. Each child node is a specialization of its parent node. This tree structure is used by the profile layer to determine the similarity between two nodes for the purposes of finding replacements for missing channels

[1] Device roles are a concept that is of most use in thinking about the protocol, but does not play a part in implementation.

[2] See appendix B.4.3

and for autoconfiguration. In section 4.3.1, the concept of similarity is explained in more detail.

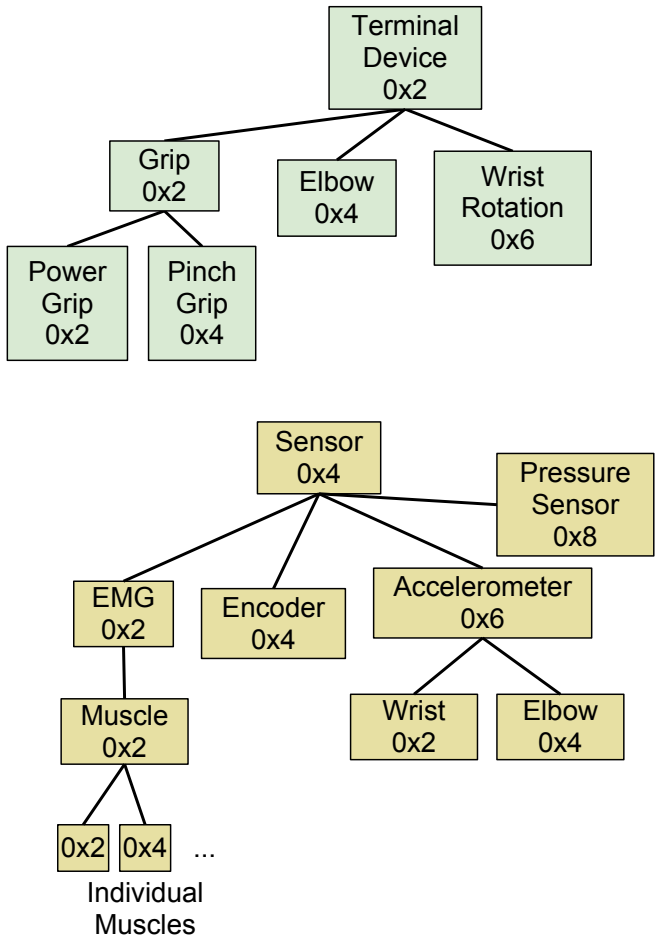


Figure 4.6: An example tree structure for the PDCP profile layer.

Values

Each node is given a 4 bit value, called a "profile component", and a profile value is constructed by concatenating the components from the root node to, and including, the profile node. For example, in fig. 4.6, the "Wrist" node has the profile value 0x462 because the "Sensor", "Accelerometer" and "Wrist" nodes have the values 0x4, 0x6, and 0x2 respectively.

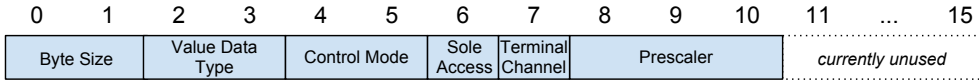
This profile scheme implies that the profile value will be variable length, and PDCP does support this. However, a fixed length value has advantages in implementation, and a 16 bit value has been used previously. In fixed length values, the remaining bits can be padded with zeros at the end (e.g 0x4620). A fixed length of 16 bits will be used throughout this thesis, but is not mandatory.

The profile value of 0x0 is invalid and can be used as a null profile in implementation.

In fig. 4.6 the values of the nodes are all even numbers. The odd numbers can be reserved for vendor use or future expansion. The reason why the used values are not bunched together is that a lower number implies higher priority and will determine the order in which channels are connected. This will be relevant if the system for example does not have enough sensor signals to serve all terminal devices. In such a case, the lower profile channels will be attempted connected first. The spread scheme thus gives more flexibility to custom values.

4.2.3 Channel Type

In the profile layer, the channel type is used as a collection of flags. They are currently defined as follows.



- **Bits 0-1** *"Byte Size"* - byte size of values transmitted/preferred by this channel.
  - 0b00: 4 bits
  - 0b01: 8 bits
  - 0b10: 12 bits
  - 0b11: 16 bits

  The purpose of this is both to know how to extract the data from the payload, and to know the range of the value. 4 and 8 bit values take up 8 bits; 12 and 16 bit values take up 16 bits. A value's full range is used, e.g. 0-4095 for unsigned 12 bit integers.

- **Bits 2-3** *"Value Data Type"*.
  - 0b00: signed integer
  - 0b01: unsigned integer
  - 0b10: float
  - 0b11: boolean

The float type has not been specified yet, in terms of which standard to use, and how to view its range. Boolean values are *false* when all bits are 0, and *true* otherwise.

- **Bits 4-5** *"Control Mode"*
  - 0b00: position
  - 0b01: velocity
  - 0b10: acceleration
  - 0b11: other

  The purpose of these flags is to inform of the level of differentiation of the values on this channel. These are especially relevant for set points.

- **Bit 6** *"Sole Access"* (For input channels)
  - 0b1: This channel does not allow other input channels with this flag set to be connected to its source output channel.
  - 0b0: This channel can be connected to any output channel, subject to profile value.

- **Bit 7** *"Terminal Channel"* (For input channels)
  - 0b1: This input channel directly controls a terminal device.
  - 0b0: This input channel does not directly control a terminal device.

  The purpose of this flag is to give the autoconfigure algorithm starting points. See also listing 4.7.

- **Bits 8-10** *"Prescaler"*. Scale the received value by an amount.
  - 0b000: 1
  - 0b001: 10^{-6}
  - 0b010: 10^{-3}
  - 0b011: 10^{-1}
  - 0b100: 10^{1}
  - 0b101: 10^{2}
  - 0b110: 10^{3}
  - 0b111: 10^{6}

  These prescaler values are suggestions, and further research will likely find more useful values.

In all cases, the input channel cannot force some format on the output channel. The output channel operates independently of the input channels connected to it. Thus, for the input channel, flags such as the byte size, data type, and control mode are to be regarded as the "desired"

format (see listing 4.9). The values it receives might have another format. The input channel will be required to "understand" all possible formats of the values it receives.

The available flags favor using relative values and scaling them to fill the range. However, this issue has not been thoroughly explored by this author, and further work ought to determine if these flags should be tailored differently.

Timing

There will probably also be a need to specify the frequency of the data sent on the bus, because the timing information will likely be used in some feature extraction algorithms. The best way to represent this information has not been investigated in the course of this work, and will need to be addressed in later work.

4.2.4 New Parameters

Device

No new device-wide parameters (channel index 0) have been defined.

Output Channel

The following additional parameters have been defined for output channels:

9. *"Required Input Channels"*: A value with length up to 256, where each bit corresponds to a channel index. If an input channel's bit is set, this means that the device uses data received on that input channel to calculate the data sent on this output channel. The required input channels can then be connected. Only bits corresponding to extant *input* channels should ever be set to 1. The most significant bit refers to channel index 0.

10. *"Desired Input Channels"*: The same as "Required Input Channels", except that the indicated channels are not required for the operation of this channel, but will enhance the functionality. This can, for example, be used for secondary feedback sensors which, when present, can help a control unit give more close control.

Input Channel

The following additional parameters have been defined for input channels:

8. *"Source's Type and Profile"*: The type and profile of the source output channel. Value is 0 (null) when not connected.
9. *"Required Input Channels"*: A value with length up to 256, where each bit corresponds to a channel index. If an input channel's bit is set, it means that that input channel must be connected for the current one to work. Only bits corresponding to extant *input* channels should ever be set to 1. The most significant bit refers to channel index 0.
10. *"Desired Input Channels"*: The same as "Required Input Channels", except that the indicated channels are not required for the operation of this channel, but will enhance the functionality. This can, for example, be used for secondary feedback sensors which, when present, can help an effector align itself.
11. *"Terminal Profile"*: The profile of the set points for the terminal device that this channel controls. For set point channels, this is identical to its regular profile.

The concept of terminal profile demonstrates the dual purpose of the profile of a channel. The first is to determine *similarity*, and the second, which is sometimes given to the terminal profile, is to communicate *function*.

Examples

Here are some examples to illustrate the purpose of the added parameters and flags.

Figure 4.7 shows an example system with a sensor, a control unit and a terminal device. In this case, the terminal device's channels will be designated terminal channels, and their terminal profile will be the same as their profile. But there is no point in connecting the terminal channels to the control unit without connecting the sensor to the control unit, and this is the purpose of the "Required Input Channels" parameter. The control unit's output channels' "Required Input Channels" parameter will specify one or more of the control unit's input channels. When the bus arbitrator then connects an input channel to one of the control

unit's output channels, it knows to connect more channels, and can follow this chain as far as it goes. If it is unable to complete a connection, it may need to go back and tear down links. In the future, terminal devices may also contain sensors, allowing for feedback loops. The "Required Input Channels" parameter can then help set up complicated webs of channel links in a recursive manner. The "Desired Input Channels" parameter works in almost the same way, except that desired channels are not mandatory for proper operation so that connections can be set up even if one or more desired channels cannot be connected.

Figure 4.8 shows an example system with two sensors, and a device that is both a control unit and a terminal device. In this case, there will be no set point channels in the system, but the sensor input channels are used to control the terminal device. These channels will therefore be designated terminal channels (The terminal channel flag will be set in the channel type). The terminal profile will be the profile a set point channel for the terminal device would have.

Input channels (not just output channels) also have a "Required Input Channels" parameter. This is so that if, for example, channel 1 and 2 on the control unit are used to control opening and closing of a hand, it would make no sense to connect just one of them. In that case, these channels can reference each other in their required channels.

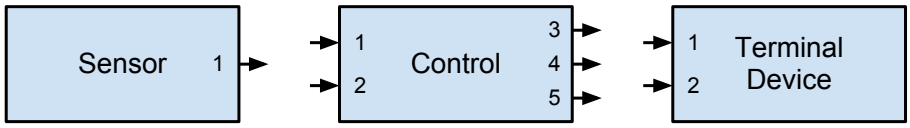


Figure 4.7: Example system 1. The numbered arrows represent input and output channels.

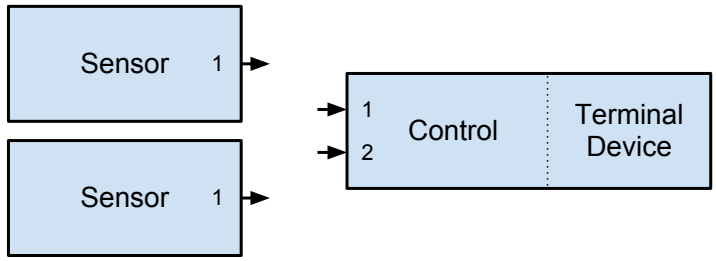


Figure 4.8: Example system 2. The numbered arrows represent input and output channels.

4.3 Device Behavior

This section will describe how devices will behave differently with the profile layer.

4.3.1 Bus Arbitrator

The bus arbitrator profile behavior will mostly occur right after a system reset.

The bus arbitrator will perform the following procedure (listing 4.1) when the system starts up:

Listing 4.1: Pseudo code for bus arbitrator startup procedure

```
1  if (there is a configuration in memory)
2      //Retrieve configuration from memory.
3      load_configuration()
4  else
5      //Retrieve configuration from individual devices.
6      retrieve_configuration()
7
8  //Connect according to retrieved configuration.
9  implement_configuration()
10
11 if (configuration involves missing devices/channels)
12     //Try to find alternate channels using type/profile.
13     find_replacements()
14
15 if (there are unconnected terminal channels)
16     //Try to connect them according to type/profile.
17     autoconfigure()
```

This means that the central configuration stored in the bus arbitrator takes precedence over the distributed configuration. This prevents devices configured in other systems to throw the system off with their foreign configuration.

Further, the following procedure (listing 4.2) can be performed regularly to provide hot-swapability.

Listing 4.2: Pseudo code for bus arbitrator hot-swapability duty cycle

```
1  if (devices/channels have disappeared since last cycle)
2      //Try to find alternate channels using type/profile.
3      find_replacements()
4
```

```
5       if (there are unconnected terminal channels)
6           //Try to connect them according to type/profile.
7           autoconfigure()
```

`load_configuration()`

Retrieves a configuration from non-volatile memory.
See section 4.5 for more information about storing configurations.

`retrieve_configuration()`

Retrieves a configuration by reading the Source VID, PID, serial number, channel index, type and profile from all input channels in the system.

`implement_configuration()`

Constucts the links described in the configuration.

Listing 4.3: Pseudo code for making links based on a retrieved configuration.

```
1 function implement_configuration()
2     for (every link in configuration)
3         if (input channel is not terminal channel)
4             // connect link, respecting sole access
5             connect(link)
6         else
7             // connect link, respecting sole access,
8             // required and desired channels
9             intelligently_connect(link)
```

Listing 4.4: Pseudo code for making links in a way that respects sole access.

```
1 function connect(link)
2     if (it does not violate sole access)
3         set up link
```

Listing 4.5: Pseudo code for making links in a way that respects required channels desired channels and sole access.

```
1 function intelligently_connect(link)
2     array created_links
3     connect(link)
4     if (failed)
5         return (failure)
6     else
```

```
7           add link to created_links
8       for (each channel that is required or desired)
9           if (not already connected)
10              find suitable output channel
11              intelligently_connect() //recursive
12              if (failed)
13                  if (input is required)
14                      remove all links in created_links
15                  return (failure)
16      return (success)
```

`find_replacements()`

Tries to find replacements for channels that are in the configuration, but are missing from the system.

Listing 4.6: Pseudo code for finding replacements for missing channels and connecting them.

```
1  function find_replacements()
2      array repaired_links
3      for (each link with exactly one missing channel)
4          // find replacement based on type and profile
5          //                              of missing channel
6          find_similar_channel(missing channel)
7          if (succeeded)
8              add link to repaired_links (do not connect)
9      sort (repaired_links) by similarity between
10                                  input and output
11     for (each link in sorted repaired_links)
12         intelligently_connect(link)
```

`autoconfigure()`

Tries to connect channels in a sensible way if system is unconfigured or partially configured. A system is fully configured when all its terminal channels are connected.

Listing 4.7: Pseudo code for autoconfiguring an unfigured or partially configured system.

```
1  function autoconfigure()
2      array unconnected_chans
3      for (each unconnected terminal channel)
4          add channel to unconnected_chans
```

```
5       sort (unconnected_chans) by terminal profile
6                                       (low to high)
7       for (each channel in sorted unconnected_chans)
8           // find suitable output channel based on
9           //                          type and profile
10          find_similar_channel(channel)
11          intelligently_connect()
```

`find_channel()`

Several procedures call for finding channels (replacements or candidates for connection) based on type and profile.

Listing 4.8: Pseudo code for finding similar channels for replacement or connection.

```
1  function find_similar_channel(channel)
2       for (each candidate)
3           // candidates have the correct direction (input or output)
4           // and are free to be connected to.
5           determine_similarity(channel, candidate)
6       return (the most similar candidate)
```

`determine_similarity()`

Returns a number describing the similarity between two channels, based on the type and profile of those channels.

Listing 4.9: Pseudo code for determining the similarity between two channels.

```
1  function determine_similarity(channel 1, channel 2)
2       variable similarity
3       similarity <- 0
4       for (profile component from front of profile)
5           if (component is equal in
6                           channel 1 and channel 2
7                           and component is not 0)
8               similarity <- similarity + 8
9           else
10              exit for loop
11      if (channels have equal control mode in type)
12          similarity <- similarity + 4
13      if (channels have equal data type in type)
14          similarity <- similarity + 2
15      if (channels have equal byte size in type)
```

```
16          similarity <- similarity + 1
```

This means that the similarity is determined primarily from the profile, while control mode, data type, and byte size are used as tie breakers.

4.3.2 Regular Devices

Devices other than the bus arbitrator are responsible for presenting the parameters and that they have the correct value. It should be noted that the device is expected to retain both static and writable parameter values in non-volatile memory. Writable values include *source's VID*, *source's PID*, *source's serial number*, *source's channel index*, *source's type*, and *source's profile*.

Other than this, devices should behave like before.

The *update data channel* message will have little or no function because the bus arbitrator always has the final say in which links to implement.

4.4 Notes on the Bus Arbitrator Device

This thesis assumes generally that the bus arbitrator is a separate device, which has no other roles. However there are other possible constellations:

- The bus arbitrator is part of a device performing the sensor, control, or terminal device role. This should be avoided, because these devices are more likely to be replaced.
- The bus arbitrator is part of another utility device such as a battery pack, or a memory node. This is a better choice, because these devices are more bound to the particular prosthesis.

Generally, a constellation which ensures that the bus arbitrator remains with a prosthesis as permanently as possible is preferred. In terms of cost, it would probably also be beneficial to consolidate all such permanent parts into one device. Examples of such permanent parts are: Bus arbitrator, battery socket, memory, display, buttons etc.

4.5 Storing Configurations

In the context of the profile layer, "configuration" means "the way the system is connected". The profile layer calls for sentralized storage of

configurations so that these can be reproduced. There is therefore a need for a standardized way to describe a configuration. A natural way to think of a configuration is a collection of channel links. This has therefore been chosen as the approach to creating configuration descriptions.

This specification does not define constraints other than non-volatility on where to store the central configuration. It can either be stored locally, or a memory node can be a separate device on the bus.

4.5.1 Structure

The configuration is stored as a collection of links, and each link contains an input channel and an output channel. The channels each contain these 8 values:

- type
- profile
- channel index
- owner's VID
- owner's PID
- owner's serial number
- owner's type[3]
- owner's profile[3]

4.5.2 Semantics

The configuration is stored in memory as a string of bytes, and each value takes up 3 bytes. The first byte of a value is a code that specifies what the following 2 byte value means. The layout is as shown in fig. 4.9. These codes are defined as shown in table 4.1.

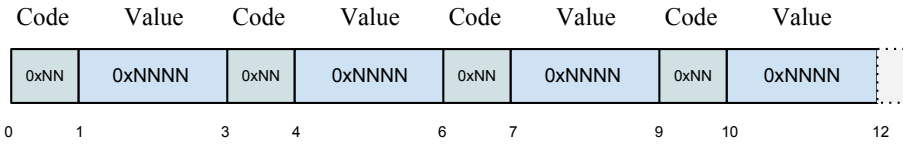


Figure 4.9: An illustration of the layout of a stored configuration string.

"Configuration", "Channel link", "Input channel", and "Output channel" have declarative roles as well, meaning that:

- A configuration declaration starts with "Configuration" and ends with "Configuration" or "End".

[3]Unneeded at this point, but included for future use.

Code	Name	Value
1	# of configurations	The number of configurations in this collection.
2	Configuration	The index of this configuration.
3	# of links	The number of channel links in this configuration.
4	Channel link	The index of this channel link.
5	Input channel	*No defined meaning, use 0x00*
6	Output channel	*No defined meaning, use 0x00*
7	Channel type	The type of the current channel.
8	Channel profile	The profile of the current channel.
9	Channel index	The channel index of the current channel.
10	Owner VID	The vendor ID of the device containing the current channel.
11	Owner PID	The product ID of the device containing the current channel.
12	Owner serial number	The serial number of the device containing the current channel.
13	Owner type	The type of the device containing the current channel.
14	Owner profile	The profile of the device containing the current channel.
15	End	*No defined meaning, use 0x00*

Table 4.1: This is the definition of the code/value pairs used to store configurations.

- A channel link declaration starts with “Channel link” and ends with “Channel link”, “Configuration” or “End”.
- An input channel declaration starts with “Input channel” and ends with “Output channel”, “Channel link”, “Configuration” or “End”.
- An output channel declaration starts with “Output channel” and ends with “Input channel”, “Channel link”, “Configuration” or “End”.

Figure 4.10 shows an example of a set of stored configurations.

This framework allows storing multiple configurations with multiple links, and since each value is explicitly described with a code, other values can be added at a later stage without disrupting previously made configurations.

The framework is similar to the attribute system in Bluetooth LE [1]. JSON was also considered as a framework, but was discarded because of the complexity of making a parser. With such short values (no strings), JSON also has a high storage overhead.

4.6 Further Ideas

The following should be considered when continuing work on this specification.

4.6.1 Profile Hierarchy

The final profile hierarchy will need to be carefully constructed, bearing the following in mind:

- The profile number is also a priority, so numbering is not arbitrary. Previous sections mention using only even numbers, but other intervals are also possible.
- Each standardized profile “node” should come with a set of requirements to standardize the behavior expected from different profiles. The more specialized the profile (the farther it is from the root node), the more unambiguous the requirements.

	Code	Value	
Byte n	1	2	# of configurations = 2
Byte n+3	2	1	Start of configuration 1
Byte n+6	3	2	Configuration 1 contains 2 links
. . .	4	1	Start of channel link 1
	5	0	Start of link 1's input channel
	7	10	Link 1's input channel has type 10
	8	20	Link 1's input channel has profile 20
	9	1	Link 1's input channel has channel index 1
	10	30	Link 1's input channel's owner's VID is 30
	11	40	Link 1's input channel's owner's PID is 40
	12	50	Link 1's input channel's owner's serial number is 50
	13	60	Link 1's input channel's owner's type is 60
	14	70	Link 1's input channel's owner's profile is 70
	6	0	Start of link 1's output channel
	7	10	Link 1's output channel has type 10
	8	20	Link 1's output channel has profile 20
	9	1	Link 1's output channel has channel index 1
	10	30	Link 1's output channel's owner's VID is 30
	11	40	Link 1's output channel's owner's PID is 40
	12	50	Link 1's output channel's owner's serial number is 50
	13	60	Link 1's output channel's owner's type is 60
	14	70	Link 1's output channel's owner's profile is 70
	4	2	Start of channel link 2
	5	0	Start of link 2's input channel
	..	...	
	6	0	Start of link 2's output channel
	..	...	
	2	2	Start of configuration 2
	3	1	Configuration 2 contains 1 link
	4	1	Start of channel link 1
	..	...	
	15	0	End

Figure 4.10: An example configuration storage.

4.6.2 Algorithm intelligence

Some of the algorithms described here can be modified, and made more intelligent. An investigation into the level of intelligence which is most sensible when considering all things such as hardware and software constraints, usability, practicality, and interoperability will probably be needed.

4.6.3 An Alternative Approach to Multiple-Role Devices

An alternative (conceived of in the later stages of this work) to the way of handling multiple-role devices described above is to require that no channels be dropped, and data is only allowed to flow between two roles if the channels are connected, even if the roles are in the same device.[4] The advantage of this approach is that it gives more flexibility to the bus arbitrator, and that the "terminal profile" parameter[5] may be unnecessary. The disadvantage of this approach is a slight increase in logic in devices (more channels, and inter-role communication), and increasing the complexity of the system by introducing more channels. All things considered, this method seems a good alternative, and would have been pursued further if the idea had come earlier in the thesis work.

4.6.4 More Roles

The 3 roles introduced in section 4.1 were developed in [1], as a simplification of 5 roles (see fig. B.1 and fig. B.5). Adding these roles would greatly increase the flexibility of the system. However, if they are added, stronger guidelines would be needed to ensure interoperability. As an example, a guideline could be that some roles cannot be split into individual devices, and that channels of some profile must always be present on the bus. If section 4.6.3 is realized, it might help interoperability, but also bring new challenges if devices disagree on what roles are available. Perhaps a compromise between the two alternatives would be best.

4.6.5 Expanding

It is the author's opinion that the component of PDCP least robust to expansion is the node ID. The system-wide upper limit of 256 node IDs can quickly be too small as new classes of sensors are being put to use. If

[4] In this case, it would, of course, be wise to require that the device does not physically send the data to itself on the bus.

[5] Described in section 4.2.4.

additional roles are added, this would also drive up the number of used node IDs.

One solution to this problem is to increase the size of the node ID value, and use the extended msg ID of CAN, which will provide an additional 18 bits. Not all of these will be needed, and some might even be made freely usable as extra payload room.

4.6.6 Timing

As mentioned in section 4.2.3, information about the period of values transmitted will need to be accomodated. As an extension of this, one can envision two different types of signals: Periodic streams, and sporadic, event-like messages.

Additionally, requirements for sensor signals and set points are somewhat different. There is little reason to send more than one set point in the same message because they are used without consideration of past set points. But sensor signals can be processed in many different ways, and multiple samples can therefore be sent in each message to conserve bandwidth.

Chapter 5

Implementation

This chapter describes an implementation by the author of the profile layer for testing purposes. All software should be available as an attachment. The attachment contents are inventoried in appendix C.

The testing procedure is described in chapter 6.

5.1 The Hardware

The test setup is shown in fig. 5.1.

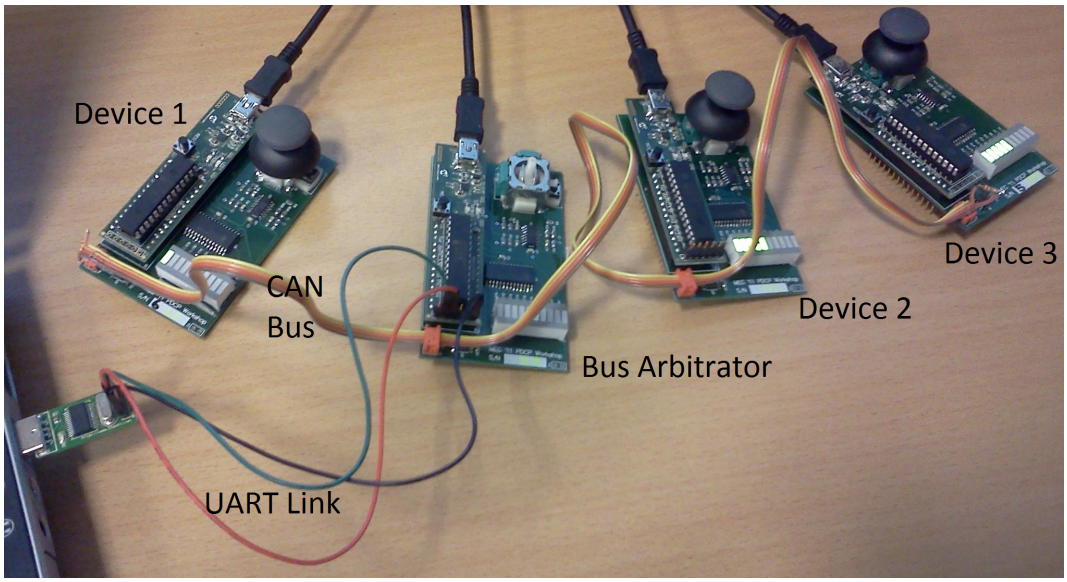


Figure 5.1: The test setup

The software produced for this test was run on a PC, while the rest of the bus arbitrator functionality resided on a PIC microcontroller[1]. The profile layer (on the PC) communicated with the lower layer bus arbitrator functionality (on the dsPIC33) through a UART link, using the provided API. The reason the profile layer was implemented on a separate device was that, due to licensing, the particular implementation of the bus arbitrator lower levels used was closed source, and was provided as a pre-compiled *hex* file.

The test system consisted of four identical nodes. These nodes were created at UNB for a PDCP workshop at the MEC'11[2] conference and consist of a Microstick[3] connected to a circuit board created at UNB. The UNB board contained a CAN tranceiver, connector for CAN bus, a joystick and an array of LEDs. The microstick contained a single freely controllable LED.

The CAN bus was a four-wire ribbon cable carrying Vcc and GND in addition to the two CAN signals CANL and CANH. CANL and CANH were terminated in both ends by 120Ω resistors.

The UART link consisted of 3 wires carrying RX, TX, and GND, connected to a brandless USB to UART dongle.

The nodes were powered by the same USB link used to program them. When the tests called for adding and removing devices from the bus, the USB power was plugged or unplugged, and each device, starting with the bus arbitrator were reset (push button reset).

5.2 The Software

The lower layer functionality was been implemented by Yves Losier and Adam Wilson at UNB, and they have also implemented an API accessible through a UART link.

The bulk of the test software was developed for a PC running Linux[4]. It was written in C, and modularized, to ease later porting to a microcontroller alongside low level functionality, if this is desired.

The complete test setup contains both code produced by the author, and code produced by Yves Losier and Adam Wilson.

Figure 5.2 shows some of the software modules involved in the test

[1] Model no. dsPIC33FJ64MC802

[2] "MyoElectric Controls Symposium 2011" in Fredericton, New Brunswick, Canada.

[3] Development board for the dsPIC33 by Microchip

[4] Lubuntu 12.10

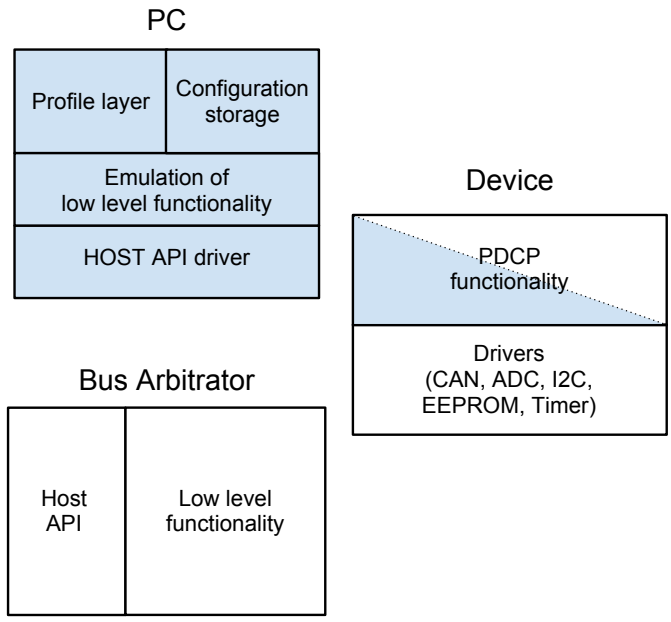


Figure 5.2: An outline of the software modules. Modules highlighted blue have been developed by the author.

setup. Here follows a more detailed explanation of each device and module.

5.2.1 The Bus Arbitrator

The bus arbitrator software was provided precompiled, a "black box". Interaction with it happened through a UART API called the Bus Arbitrator API. The API is documented in [9]. The API defines a message format including a message ID field, a length field, and a variable length payload field. The host (in this case the PC) sends a request and receives a reply from the bus arbitrator. Communication is always initiated by the host. The message ID specifies the request type. Message IDs come in threes: request (n), success response ($n+1$), and failure response($n+2$). With these messages, the host can inquire about the devices, channels, and channel links of the system, read and write parameters manually, and configure channel links. There are also options for resetting devices and for sending a raw CAN packet onto the bus ("passthrough"), but these were not used in the course of this work.

Details of Functionality

From interaction through the Bus Arbitrator API, the following could be inferred about the inner details of the bus arbitrator:

- It will always accept bind requests while running.
- It will not set up a link without being asked to by a device sending an Update Data Channel Request, or by the host using the API.
- It will not let devices bind unless they send regular beacons.
- If a device disconnects from the bus, the bus arbitrator will still report it as present through the API.[5]

Hot-Swappability

Hot-swappability was not implemented because of idiosyncracies in the low level bus arbitrator implementation, as described above.

5.2.2 The Other Devices

The software for the non-bus arbitrator nodes was adapted from the code used at the workshop the nodes were originally developed for.

The changes involved adding support for new parameters, and input and output functionality for use in tests. In addition some refactoring was done to facilitate the implementation of the different test cases.

The software consists of these components:

- `ADC1`[6]- Driver for the analog to digital converter. Used to read joystick input.
- `CAN1`[6] - Driver for CAN.
- `DEE Emulation 16-bit`[6] - Driver for EEPROM (non-volatile memory).
- `I2C`[6] - Driver for I2C. Used to control LED array.
- `Timer`[6] - Interface for timers.
- `PWM` - Driver for PWM module. Used to fade the LED.
- `PDCP` - PDCP specific functionality.

[5]This fact disallowed implementation of hot-swappability.
[6]Unmodified by the author.

- `Main` - Main program flow.

Most PDCP functionality is provided by the `ProcessMessage()` function in `PDCP.c`. This function interprets an incoming CAN message and responds accordingly by sending a parameter value, updating a parameter value, or storing a value received on a channel link.

The `main()` function has three parts:

1. A setup part which includes retrieving values from EEPROM,
2. A bind sequence part, and
3. An infinite loop performing three main functions:
   (a) Sends a beacon every beacon interval.
   (b) Reads and processes CAN messages.
   (c) Performs channel-specific behavior (e.g. sending joystick values on a channel).

The code needs to be compiled with a "test number" macro (`TEST_NUM`) and a "device number" macro (`DEVICE_NUM`); these are set in `PDCP.h`. These macros dictate parameter values and channel behavior, and are used to quickly switch between compiling the different test devices. The list of possible parameters are found in `PDCP.c` and the list of channel functions are found in `Main.c`.

The software was compiled and loaded using MPLAB X v1.80, the standard IDE provided by Microchip.

5.2.3 The PC (Host) software

The PC software is the most substantial part of the software authored in the course of this thesis work.

The software consists of a number of modules (also illustrated in fig. 5.3):

- Primary modules:
  - `pdcp_driver`
  - `pdcp_low_level`
  - `pdcp_configurations`
  - `pdcp_profiles`

- Helper modules:
  - `pdcp_error`
  - `pdcp_malloc`
  - `pdcp_names.h`
- Runtime modules:
  - `pdcp_run`
  - `main.c`

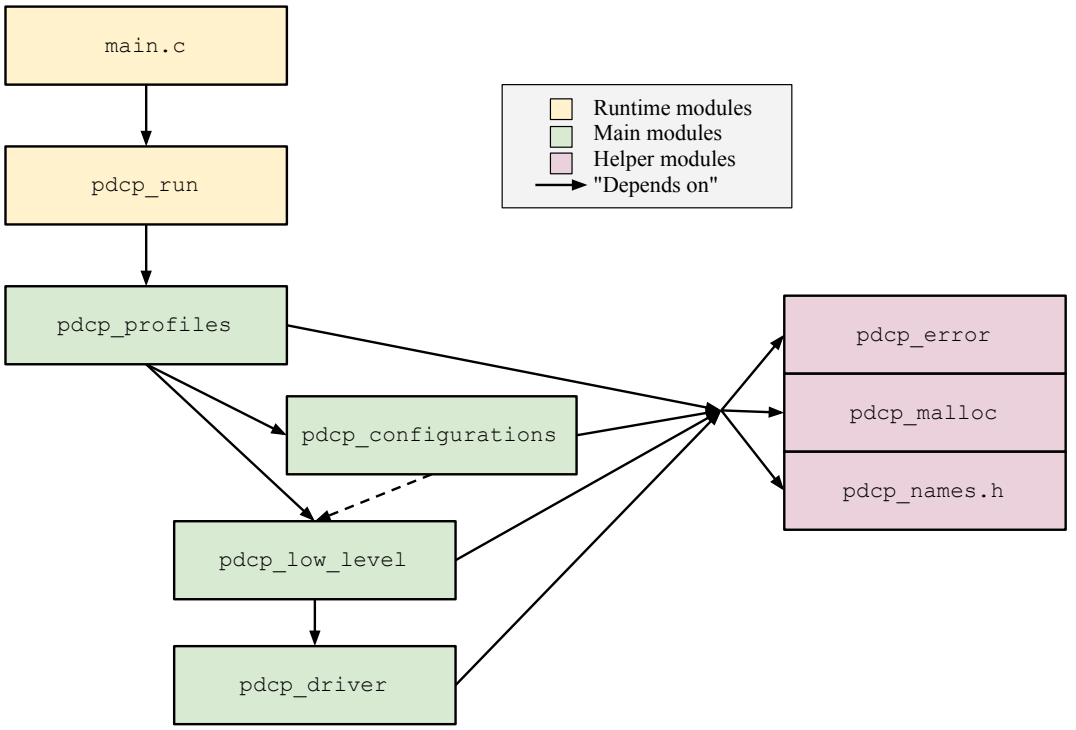


Figure 5.3: The software modules of the host program run on the PC. The configuration module's dependency on the low level module is dotted. This is because it only uses the struct definitions, not function calls.

Procject Structure

Each of the software modules includes a source file (`.c`) and a header file(`.h`) (except `main.c` and `pdcp_names.h`). In addition, many modules

have unit test files meant to be used with the Unity test framework [10].

The code can be found in the attachment, in the "hostkode" folder. Header files are in "include", source files in "src", and test files in "test". The "obj" folder holds intermediate object files during compilation.

The "test" folder also contains a "test_runners" folder, containing files autogenerated by Unity.

The "hostkode" folder contains four test files(`test_conf.file`, `test_read.file` and `test_write.file`) used by the unit tests. It also contains a makefile, the file containing stored configurations (`configurations.file`), and a script (color_output.sh) used to color console output. Usage:

`make all |& ./color_output.sh` or
`make test |& ./color_output.sh`

Makefile

The makefile can be used both to compile and run tests, and to compile and run the main program. Run tests with `make test`, and main with `make all`. The makefile will automatically compile (if needed) and run the program (with test: multiple programs). `make clean` will remove object files.

Module *pdcp_driver*

This module provides an interface to the UART link by providing data structures (`uart_msg_struct`, `uart_msg_array` and `uart_conn`), and a send/read interface:

```
driver_send_msg(uart_conn* conn, const uart_msg_struct* in)
driver_blocking_read(uart_conn* conn, uart_msg_struct* out)
```

`driver_blocking_read` will not return before it has received a message, unless a timeout occurs.

The fields of `uart_msg_struct` struct correspond to the message structure of the bus arbitrator API. The `uart_conn` struct contains file descriptors and other meta information.

`driver_init` must be called before the driver module can be used.

Module *pdcp_low_level*

The low level module performs the emulation of low level funtionality by use of the bus arbitrator API. It provides the data structures for storing

system data, provides function for all the API features, and auxilliary functions pertaining to the system info, such as printing to stdout, and comparing variables.

At its core is the `pdcp_system_info` struct. This contains arrays of all the devices, channels and channel links on the system. It has an array called `missing_devices`. These are devices that are referenced by input channels as source output channels, but are not present on the bus. Some links in the `channel_links` array may also involve a missing device. In that case, the `link_status` field of the channel link struct will be set to `LINK_STATUS_INACTIVE` (= 1).

`low_level_init` must be called before the module can be used, and will fetch all system info. To update the system info later, `explore_system_info` can be used.

See `pdcp_low_level.h` for additional documentation.

Module *pdcp_configurations*

The configurations module allows for storing and loading a single configuration[7]. The module provides its own data structures tailored for storing configuration (`pdcp_configuration`, `pdcp_channel_link_configuration` and `pdcp_channel_configuration`), and the following interface:

```
pdcp_configuration* load_configuration()
void save_configuration(const pdcp_system_info* system_info)
```

and some functions for comparing this module's structs to the low level module's structs.

Module *pdcp_profiles*

This module contains the profile logic. The algorithms described in section 4.3.1 are implemented here.

Listing 4.1 is performed by `profiles_first_setup`. Listing 5.1 shows pseudo code for this procedure, as it is implemented.

Listing 5.1: Profile layer startup procedure as implemented.

```
1 Wait until no bind requests for x ms.
2
3 if (there is a configuration in memory)
4     Retrieve configuration from memory.
5     Remove all channel links requested by devices
```

[7]The parser supports multiple configurations, but full support for this was not prioritized.

```
6       Connect according to retrieved configuration.
7   else
8       Allow all channel links requested by devices.
9
10  if (configuration involves missing devices/channels)
11      Try to find alternate channels using type/profile.
12
13  if (there are unconnected terminal channels)
14      Try to connect them according to type/profile.
```

Most of the algorithms in section 4.3.1 are implemented under the same name, except

- `retrieve_configuration()`: This happens naturally through the low level actions of the bus arbitrator and devices. The devices will send update data channel requests and the bus arbitrator will set up the channel links automatically. From there, the profile layer can decide whether to keep them or not. If there is a configuration stored centrally, this will be implemented, and the automatic links will be removed.
- `connect(link)`: The sole access checks mostly happen inline, and a low level function named `make_channel_link` is used afterwards.[8]
- `intelligently_connect(link)`: The implementation is named `intelligently_connect_links`. Another function, `intelligently_connect_link`, finds an appropriate output channel for an input channel, then calls `intelligently_connect_links` on the two channels.
- `find_replacements()`: This functionality is covered by `find_replacements`, which finds replacements for channels reported as missing by the bus arbitrator, and `find_ochan_replacement` and `find_ichan_replacement` which are used when channels in the configuration saved to file are missing.[9]
- `find_channel()`: This is performed inline.

The module is used as shown in listing 5.2

Listing 5.2: Adapted code from `pdcp_run.c` which demostrates the correct use of the profile module

[8] `make_channel_link_raw` is the same, only with different arguments.

[9] The functionality is split because of idiosyncrasies in the bus arbitrator implementation.

```
1     profiles_init();
2     while (!profiles_ready_to_run_first_setup()){}
3     profiles_first_setup(autoconfig);
```

`profiles_first_setup` takes a boolean value specifying whether to run autoconfiguration.

Module *pdcp_error*

This module provides the following functions:

```
void check_ret_value(int ret_value, int line_num);
void check_return_code(return_code ret, int line_num);
```

in addition to the global variable `pdcp_errno`, which works in the same way as the standard `errno`. `pdcp_errno` is meant to report error codes pertaining to the bus arbitrator API. However, these error codes are not yet implemented, so `pdcp_errno` is unused at the moment.

The functions, however, are in use.

`check_ret_value` is used with values returned by standard Linux library calls, such as `read` and `write`, where a negative value indicates an error. The line number argument is for debugging purposes. The function prints an error message to stdout.

`check_return_code` is used with the `return_code` type defined in `pdcp_names.h`. The function prints an error message to stdout, in case of atypical values (i.e. not `OUT_VAR_FILLED`).

Module *pdcp_malloc*

This module provides wrappers for the standard functions `malloc`, `realloc`, and `free`. The wrappers mainly count the number of allocated memory areas, so the unit tests can verify that there is no memory leakage.

Module *pdcp_names.h*

This header file contains a number of `#defined` values used by multiple modules.

This file also contains the definition of the enumerated type `return_code`:

```
typedef enum{
        NO_ACTION,
        OUT_VAR_FILLED,
        BAD_POINTER,
        TOO_SMALL,
        ERROR_CONDITION
} return_code;
```

The `return_code` type is used throughout the host software to tell how the execution of a function went.

All values except `ERROR_CONDITION` have to do with output arguments. The function will take a pointer as argument, and attempt to set the variable (usually a struct or array) it is pointing to.

- `NO_ACTION` means that there was nothing to fill the variable with. E.g. if `driver_blocking_read` times out, it will return `NO_ACTION`.
- `OUT_VAR_FILLED` is the standard "success" response.
- `BAD_POINTER` means the output variable pointer is `NULL`.
- `TOO_SMALL` means an array among the arguments is too small.
- `ERROR_CONDITION` means some other error happened.

Module *pdcp_run*

This module executes the program based on command line arguments:

`./main.out [argument]`

- "autoconfig" or no argument: Run profile layer normally (with autoconfiguration).
- "noautoconfig": Run profile layer without autoconfiguration.
- "save": Save the current system configuration
- "print": Print the long system info (using `print_system_info` from `pdcp_low_level`)
- "info": Print the short system info (using `print_system_info_short` from `pdcp_low_level`)
- "clear": Remove all channel links in the system.
- "enforce": Remove channel links that break sole access rules.

Module *main.c*

Contains only this:

Listing 5.3: Contents of main.c

```
1 #include <stdio.h>
2 #include "pdcp_run.h"
3
4 int main(int argc, char* argv[]){
5     pdcp_run(argc, &argv[0]);
6     return 0;
7 }
```

Chapter 6

Testing and Evaluation

6.1 Method

The implemented system was tested using 5 small tests involving 4 devices each. For the purpose of the tests, a profile hierarchy was constructed, shown in fig. 6.1. The testing was an attempt to verify different aspects of the profile layer by observing what channel links were created under different circumstances.

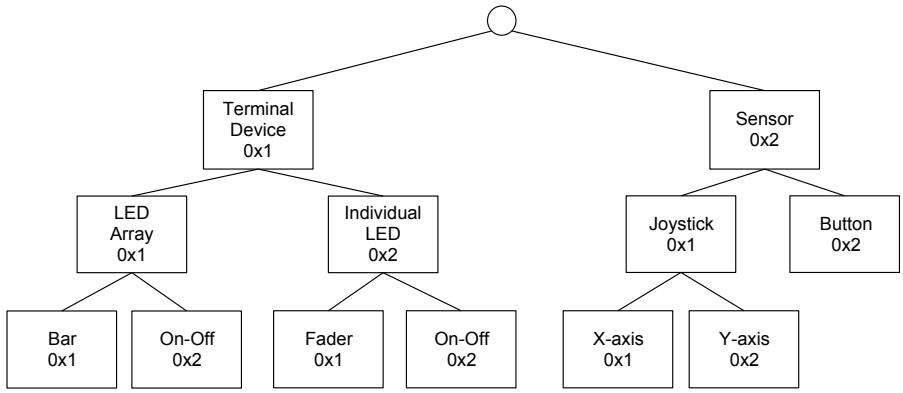


Figure 6.1: The profile hierarchy used in the tests.

6.1.1 Setup

The following explains how the testing environment was set up.

The hardware setup is displayed in fig. 5.1. The four devices were

connected by a ribbon cable, the bus arbitrator was connected to the Linux PC via UART, and all four devices were powered through USB. The three non-BA devices were connected to a separate PC running Windows and MPLAB X v1.80.

The tests utilized the 10-LED array on the MEC board and LED1 on the Microstick for output. The LED array was used as a bar graph (the On-Off profile was unused), while LED1 was used as an binary output (the fader profile was implemented using PWM, but remained unused.). The joystick's two directions in addition to its push button were used as inputs.

The non-BA device software included device specific functionality for all tests (1-5)[1] and all devices (1-3) in functions called `dutyCycle[test][device]()` (e.g. `dutyCycle32()` for test 3, device 2). The device specific parameters were set and chosen by preprocessor checks on the value of the macros `TEST_NUM` and `DEVICE_NUM`. For each test, each device was programmed individually with the appropriate `TEST_NUM` and `DEVICE_NUM`.

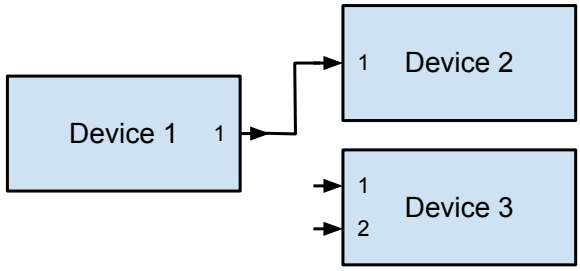
The profile layer software was run on the Linux PC, using the command line arguments documented in section 5.2.3.

[1]Note: There is also a Test 7, but it was used only for debugging.

6.2 Tests and Results

6.2.1 Test 1

Description



Profiles

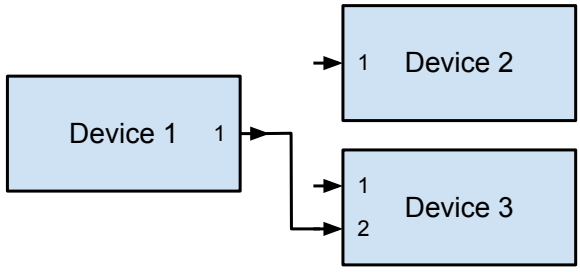
Device 1	Device 2	Device 3
Ch. 1: 0x2110	Ch. 1: 0x2110	Ch. 1: 0x2200 Ch. 2: 0x2120

Device 1's and Device 2's channels will be connected. Then Device 2 will be replaced by Device 3 to test the profile layer's ability to find replacements for missing input channels.

A video of this test can be found in the attachment.

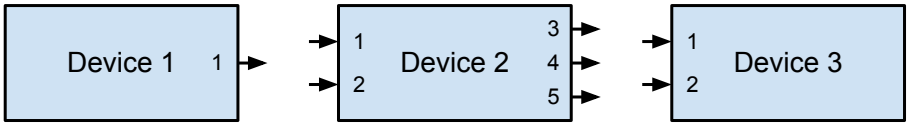
Result

The test succeded. The final configuration was:



6.2.2 Test 2

Description



Profiles

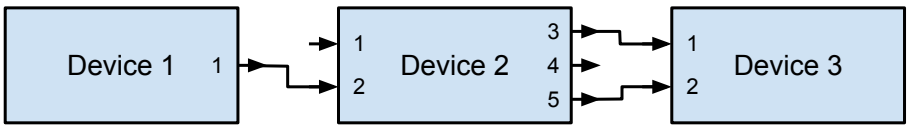
Device 1	Device 2	Device 3
Ch. 1: 0x2120	Ch. 1: 0x2200	Ch. 1: 0x1110
	Ch. 2: 0x2110	Ch. 2: 0x1220
	Ch. 3: 0x1100	
	Ch. 4: 0x1210	
	Ch. 5: 0x1220	

The system will be turned on without any configuration, testing the autoconfiguration capabilities of the profile layer.

A video of this test can be found in the attachment.

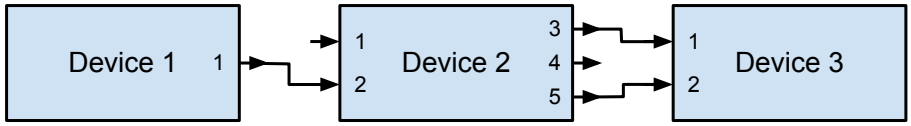
Result

The test succeded. The final configuration was:



6.2.3 Test 3

Description



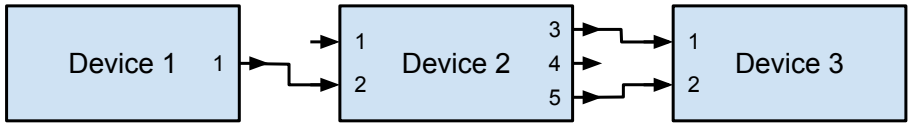
Profiles

Device 1	Device 2	Device 3
Ch. 1: 0x2120	Ch. 1: 0x2200	Ch. 1: 0x1110
	Ch. 2: 0x2110	Ch. 2: 0x1220
	Ch. 3: 0x1100	
	Ch. 4: 0x1210	
	Ch. 5: 0x1220	

The system will be configured like above, and the configuration will be saved. The system will then be reset, to test the profile layer's ability to recall the configuration, and reimplement it during power-up.

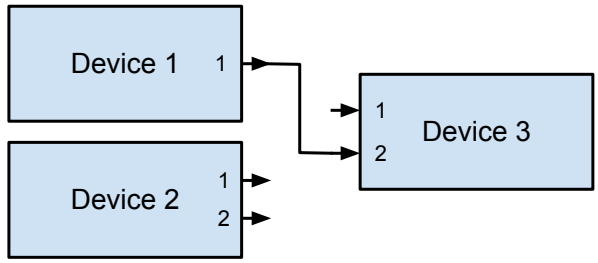
Result

The test succeded. The final configuration was:



6.2.4 Test 4

Description



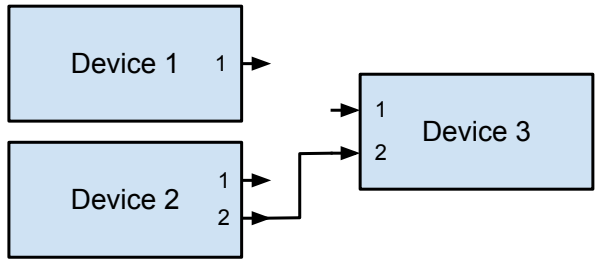
Profiles

Device 1	Device 2	Device 3
Ch. 1: 0x2110	Ch. 1: 0x2200	Ch. 1: 0x2200
	Ch. 2: 0x2120	Ch. 2: 0x2110

Device 1's channel and Device 3's channel no. 2 will be connected, Device 1 will be replaced by Device 2, to test the profile layer's ability to find replacements for missing output channels.

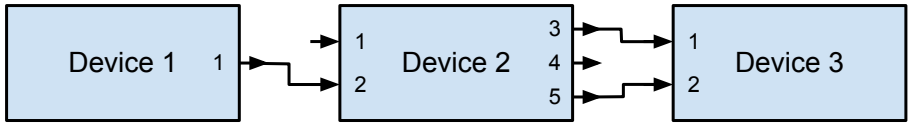
Result

The test succeded. The final configuration was:



6.2.5 Test 5

Description



Profiles

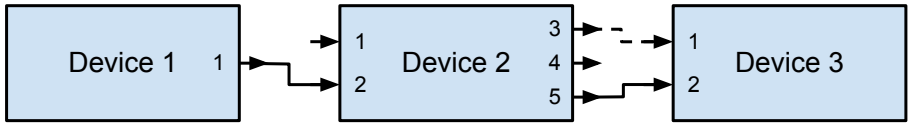
Device 1	Device 2	Device 3
Ch. 1: 0x2120	Ch. 1: 0x2200	Ch. 1: 0x1110
	Ch. 2: 0x2110	Ch. 2: 0x1220
	Ch. 3: 0x1100	
	Ch. 4: 0x1210	
	Ch. 5: 0x1220	

The system will be configured, and subsequently reset with no saved configuration to emulate a replaced bus arbitrator. This is to test the profile layer's ability to recall the configuration from just the information on the individual devices.

A video of this test can be found in the attachment.

Result

The test failed partially. Channel 1 of device 3 was reported as connected, but it did not work (LED array did not update) after resetting. This result is discussed in section 7.3.1. A printout of the system as reported to the host at the end of this test can be found in appendix D. The final configuration was:



Chapter 7

Discussion

7.1 Specification

The tasks of a device profile layer for PDCP were determined to be:

Behavior

1. Storing the usual configuration of the system and setting this up automatically each time the system powers on. This includes redundant storage inside the bus arbitrator.
2. Automatic, intelligent reconfiguration when parts are removed and/or exchanged.
3. Some amount of automatic configuration when the system is not manually configured.

Information Framework

4. Context, or meta-information about the information transmitted on the links.
5. Relational information that allows quantification of the similarity between two channels. This is to pick suitable channels to connect to.
6. Information to infer what links need to be set up to allow expedient information flow.

Does the specification address all these tasks satisfactorily?

Behavior

1. The storage of configuration boils down to the markup language described in section 4.5.2. The string of bytes can be stored in any chosen, non-volatile way. The flexibility of the markup system should be enough to store everything needed to recreate links.

2. The reconfiguration functionality is provided by listing 4.6. With this algoritm, some amount of reconfiguration will happen, which is also shown by the tests. There are many ways to achieve this functionality, and no "best solution" is presented.

3. The autoconfiguration functionality is provided by listing 4.7. With this algoritm, some amount of autoconfiguration will happen, which is also shown by the tests. There are many ways to achieve this functionality, and no "best solution" is presented.

Information Framework

4. Context is provided by the type parameter, and this is probably inadequate to cope with all the different values that could be sent on the bus. Already, timing information has been identified as missing. The final design of the context should be done very carefully.

5. The primary relational information is contained in the profile and terminal profile value. The tree structure paradigm gives a way to compare not only identical agents but also more or less similar agents. The use of the type value for additional filtering gives more correct calculations of similarity. For the goals set for this work, the profile value and tree structure perform satisfactorily.

6. The "desired input channels" and "required input channels" parameters, in addition to the "terminal profile" parameter and the "sole access" and "terminal channel" flags of the type parameter provide information about how to set up webs of links. These values together address all of the issues identified in appendix B.4.4[1]. This task is fulfilled for the purpose of all scenarios the author has conceived of during a sizeable amount of consideration. For example, this setup will accomodate feedback loops. Nonetheless, the tests did not include more complicated systems than two-level linear links, so further testing might bring up issues.

7.1.1 The Alternative

The alternative presented in section 4.6.3 is intriguing.

The advantage is more streamlined behavior by further abstracting the channels away from devices. One disadvantage is how to handle more roles (discussed in section 4.6.4).

The best way might be middle ground. When a terminal channel is explicitly indicated by setting the flag in the type parameter, the algorithms should be able to handle it even if the different channels have different root profile components. The terminal profile parameter can then likely be dispensed of.

The author's opinion is that this avenue should be investigated in future work.

7.2 Implementation

The implementation includes several parts, not all of them pertaining directly to the device profile layer. The actual profile layer (`pdcp_profiles`) module is less than 700 code lines, and thus not overly complex. For production, this size would likely increase, but given good design, a single integrated bus arbitrator node can, in the author's opinion, be acheived without a prohibiting level of complexity. Establishing this was one of the goals of the work, and has thus been accomplished.

Some of the complexity and idiomaticity of this implementation is due to the design of the bus arbitrator API. Much of the development of the host software was completed with only the API documentation as a guide, without actual devices to test on. This meant that some design choices were based on misunderstandings of the actual behavior of the bus arbitrator.

As an example, the representation of missing devices and channels, and the implementations of `find_replacements` became cumbersome because of specific implementation details in the bus arbitrator.[1]

[1] When devices request to be connected to a device that is not present, the bus arbitrator will remember this and realize the connection if the device appears later. The bus arbitrator will also report these unrealized links to the host, indicating their inactive state with a "link status". On the other hand, when the host requests a link be made, and one or both devices is not present, the request will fail and no "inactive" link will be set up. This detail was not discovered until testing the completed code on the physical system, and missing channels needed to be handled differently depending on whether they were specified in the central configuration or on the devices themselves.

The specification also allows for flexibility regarding complexity. Specifically, the algorithms, especially `find_replacements` and `autoconfigure` can be made as intelligent or simple as desired to accomodate limitations and requirements.

7.2.1 Possibilities for Further Use

The purpose of the implementation was to create a proof of concept of a profile layer for PDCP. This has largely been accomplished. The software was written in C. This was partly in the hopes that it could be reused in a single-chip implementation. However, the profile implementation is probably too much shaped by the API to be of use without it.

Perhaps the biggest use for this software is to use the `pdcp_low_level` module as C bindings for the bus arbitrator API. The low level module is the largest one (> 1500 lines of code) and developing it took considerable time, so this implementation can be useful for others making host software using the API.

The API, though not ideally suited for a device profile implementation, can be very useful for developing tools for manual configuration.

7.3 Testing and Results

The testing can be likened to interoperability testing as described in section 3.1.1. The likeness comes from the fact that the tests tested the ability of the devices to actually communicate meaningfully and achieve the intended functionality (controlling LEDs using joysticks and buttons).

As such, the tests were definitely interoperability tests of the *implementation*. Yet, as for interoperability testing of the protocol itself, it can be argued whether the testing conditions were "an environment representative of reality" for the purposes of use in prosthetics.

The main differences between the testing environment and a prosthesis use case are:

- The use of the bus arbitrator API and host software instead of a single-device bus arbitrator.
- The tests involved neither mechanical actuators nor the type of sensors that would be found in a prosthesis, specifically myoelectric sensors whose sensor readings have a completely different character than joystick readings.

- None of the control schemes used in the tests needed timing information about samples sent on the bus. In prostheses, timing information is usually needed to a certain extent, to process noisy EMG signals.
- All devices (except the bus arbitrator) used mostly the same software, on the same model of microcontroller and CAN transceiver. The tests did therefore not show whether devices developed on different hardware and by different developers can interoperate.
- The tests involved no more than 3 regular devices, which, though maybe realistic for typical prostheses available nowadays, is nowhere near the theoretical capacity of the profile.

The issue of timing has already been identified. The specification in chapter 4 is of course just a first draft, and the specification cannot be said to be complete yet. All of the above points can turn out to be issues that need to be addressed in later revisions of the profile layer.

7.3.1 The Failure of Test 5

Because of time constraints, the cause of the failure was not investigated thoroughly, but the fact that the host appeared to think the link was there, seems to indicate a small bug in the low level bus arbitrator implementation. The error does not appear to be indicative of a serious problem with the profile layer itself.

In summary, the testing suite can be said to be sparse, and not rigorous enough for more than a first proof of concept.

That said, the tests do cover most of the functionality described in the specification, and presuming that the error of test 5 is not the symptom of a deeper issue, the rough principles of the profile layer can be said to be confirmed.

Chapter 8

Conclusion

The work conducted for this thesis has produced a first draft of a specification for a device profile layer for the Prosthetic Device Communication Protocol. A proof-of-concept implementation has been made and tested, proving that adding such a profile layer to PDCP is feasible.

An investigation into the concept of interoperability was conducted, and an attempt was made to adapt the concepts found in the literature to the effort of creating a device profile layer. Interoperability was found to be a general term pertaining to the ability of agents in a network to communicate meaningfully.

The implemented system consisted of 4 nodes with identical hardware, where one was used as a bus arbitrator. This bus arbitrator node communicated with the other three over PDCP, and with a host computer over UART. The implemented profile layer code was run on the computer. This setup was found to slightly diminish the value of the implementation as an example implementation. Nevertheless, the merits of the profile layer can be said to have been duly demonstrated.

The specification was found to be sufficient to achieve the desired functionality in the implemented system. 5 tests were conducted, of which 4 were successes and the 5th was a partial failure. The failure was, however, attributed to problems largely unrelated to the profile layer implementation.

Chapter 9

Further Work

The creation of a profile layer for PDCP is only a small part of the undertaking that is PDCP.

In addition to refining and expanding directly on the work of this thesis (See section 4.6), future work in this area can include the following.

9.1 Manual Configuration

The purpose of the profile layer is to increase the usability of prostheses using PDCP. However, manual configuration will always be necessary, and a standardized framework for this is essential to the further increased usability of PDCP. The bus arbitrator API used in this thesis is a big step towards this, but much remains.

9.1.1 Software

There should be software tools for configuring the prosthesis from a PC or another type of computer. The scope of such software could be anything from simple command line instructions to a full GUI with drawable links.

9.1.2 Connection

The bus arbitrator API uses UART, but this is not the only possibility. Work is being conducted into allowing wireless configuration as well. In any case, physical connectors would benefit from standardization.

9.2 Hardware

Many aspects of hardware can be considered in relation to the development and deployment of PDCP. The addition of other classes of sensors (pressure, acceleration) is being considered. PDCP will greatly reduce the cost of additional devices, and new sensors is just one possible use of these new possibilities. Other possibilities are human interface devices such as buttons or displays.

There is also work to be done in standardizing the hardware aspects of PDCP itself, specifically the bus wire and connectors.

Bibliography

[1] Øyvind Rønningstad. "Possibility Study of Implementing a Device Profile Layer in PDCP." 2012.

[2] *Prosthetic Device Communication Protocol for the AIF UNB Hand Project*. By Yves Losier. University of New Brunswick. 2012.

[3] *PDCP Info (2011 05 04)*. University of New Brunswick.

[4] *AIF2 System Data Capture (2012 02 21)*. University of New Brunswick.

[5] *Road Vehicles - Controller Area Network (CAN)*. ISO 11898-1. Norm. ISO, Geneva, Switzerland, 2003.

[6] Yazid Benkhellat, Marc Siebert, and Jean-Pierre Thomesse. "Interoperability of sensors and distributed systems". In: 37-38 (June 1993).

[7] Amit K. Chopra and Munindar P. Singh. "Producing Compliant Interactions: Conformance, Coverage, and Interoperability". In: *Declarative Agent Languages and Technologies IV*. 2006, pp. 1–15.

[8] Anders Fougner et al. "Control of Upper Limb Prostheses: Terminology and Proportional Myoelectric Control — A Review". In: (Sept. 2012).

[9] *Prosthetic Device Communication Protocol Host ¡-¿ Bus Arbitrator Communication*. University of New Brunswick.

[10] URL: http://throwtheswitch.org/white-papers/unity-intro.html.

[11] *CANOpen Network CAN Bus Cabling Guide*. By Advanced Motion Controls. URL: http://www.a-m-c.com/download/support/an-005.pdf (visited on 12/19/2012).

[12] Thomas M. Idstein et al. "Using the Controller Area Network for Communication Between prosthesis Sensors and Control Systems". In: *MEC '11 Raising the Standard*. From the MyoElectric Controls Symposium in Fredericton, New Brunswick, Canada. 2011.

[13] Andrew Szeto. *Introduction to Biomedical Engineering*. Elsevier Inc., 2005. Chap. 5: Rehabilitation Engineering and Assistive Technology.

Appendix A

Glossary

- **PDCP** - Prosthetic Device Communication Protocol.
- **"The device profile layer" or "the profile layer"** - The protocol layer constructed, specified, implemented and tested in this thesis work.
- **Terminal Device/End Effector** - Gripper etc. placed on the end of a prosthesis.
- **Control unit/Controller/Control (role)** - The role of converting sensor data into set points for terminal device(s).
- **NTNU** - Norges Teknisk-Naturvitenskapelige Universitet (The Norwegian University of Science and Technology), which this thesis is written for.
- **UNB** - University of New Brunswick in Fredericton, Canada.
- **Adam Wilson** - Creator of PDCP, professor at UNB.
- **Yves Losier** - Creator of PDCP, professor at UNB, co-advisor for this thesis.
- **Low Level PDCP** - The part of PDCP already created by Y. Losier and A. Wilson, described in chapter 2, which the profile layer of this thesis builds on.
- **PDCP Profile Layer** - Additional functionality for PDCP, developed in this thesis, with the purpose of allowing devices to be connected in an automatic, semi-intelligent way.

- **UART** - Universal Asynchronous Receiver Transmitter. A simple, ubiquitous communication protocol.
- **TTL** - Transistor-transistor logic. Usually used to specify the voltage levels used by most digital integrated ciruits (0-3.3V or 0-5V).
- **CAN** - Controller Area Network. A network protocol designed for use in noise-heavy environments, originally automobiles.
- **MPLAB X** - IDE (Integrated Development Environment) software used to create, compile, and load programs for Microchip's microcontrollers (PIC).
- **Unit Testing** - Automated testing of small units of software.
- **Powered Prostheses** - Prosthetic limbs with motors, providing different functionality, and controlled using myoelectric sensors.
- **Myoelectric** - A term designating things pertaining the study and capture of the electric nerve signals that control muscles.
- **Bus Arbitrator (BA)** - A specialized device which must be present on the PDCP bus, see section 2.1. This device performs most of the profile layer functionality.

Appendix B

Elaboration on the Implementation of Device Profiles in PDCP

This is chapter 3 of "Possibility Study of Implementing a Device Profile Layer in PDCP." by the author. It is reproduced here for the reader's convenience.

This chapter is a discussion of how to implement a profile layer in PDCP. I will try to present multiple solutions to problems, and make decisions where it is appropriate for further discussion. The result will be a general outline of one way to implement the profile layer.

The following is a set of goals I devised for the device profile layer when it is finished. These will guide the decisions made in later sections.

- Full specification and standardization of the communication needed for basic, prevalent prosthesis functions.
- Allowing vendors to implement custom functionality.
- Allowing for future extensions to the protocol and to the device profile layer.
- Allowing for backwards compatibility.
- Being practical for use in existing systems.
- Being able to serve the increase in complexity that will come with future systems.

B.1 System Architecture

Controlling a prosthesis is essentially the problem of converting sensor data into motor input. Fougner et al.[8] divides the problem into a series of steps, as illustrated in fig. 4.3. We will use this as the basis of our system architecture.

As the information moves through the steps, it takes on different values. The question is then: Which of the values in should be available on the bus, and on what form?

The sensors will need to transmit their readings in some form, and since each processing step necessarily reduces the amount of information, the raw EMG/sensor signal should always be available, to allow for the widest range of possible control schemes.

Additionally, since raw sensor signals usually have a relatively high bit rate, a processed version could also be available. The best choice seems to be the signal features/parameters, because this is the last step where the signals from different sensors are kept separate, which means the feature extraction can be done in the sensors themselves.

Further, the effector(s) could accept set points for individual motors, to allow the control intelligence to reside outside the effector.

Lastly, the effectors should also be able to accept set points for at least one generalized "movement class"[1] so that the control intelligence is not required to be able to control *all* constellations of motors.

These constraints then outline three main roles in the system in addition to two helper roles which should be filled by the others. Figure B.1 shows the roles and signals. A device is also allowed to fill more than one main role, such as a control unit in a terminal device. The device should still be allowed to be used as just one or the other, i.e. exposing all channels.

Since features and individual motor set points are not essential for an implementation of a profile layer, the rest of the chapter will focus on implementing raw sensor signals and movement class set points. The simplified architecture is shown in fig. B.5. Support for features and individual motor set points can also be added later if not part of the first version of the profile layer.

[1] For example "grasp" or "wrist rotation".

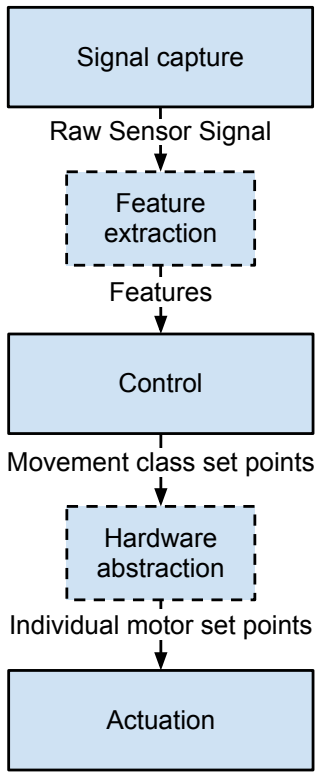


Figure B.1: This is the division of roles I suggest, and the signals that can travel on the bus.

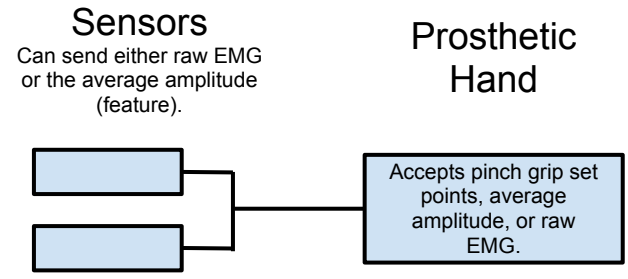


Figure B.2: Example 1 of a prosthesis system conforming to the proposed specifications. Here, the control role is performed in the hand itself. If the sensors send average amplitude (PMES), this system will be a digital equivalent to modern two-site systems.

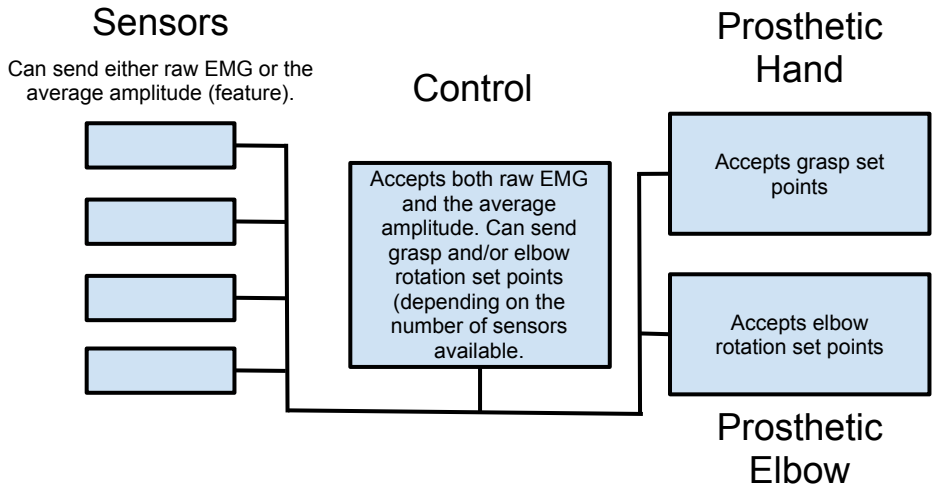


Figure B.3: Example 2 of a prosthesis system conforming to the proposed specifications. This is a thought scenario with a control unit especially suited to the combined elbow-hand prosthesis.

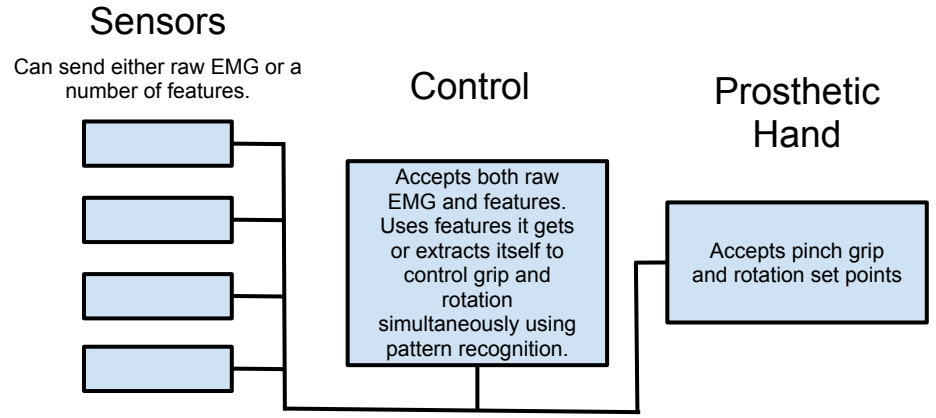


Figure B.4: Example 3 of a prosthesis system conforming to the proposed specifications. Pattern recognition systems must be trained, so the advantage of having a separate control unit is that hands can be switched without needing to train a new control unit.

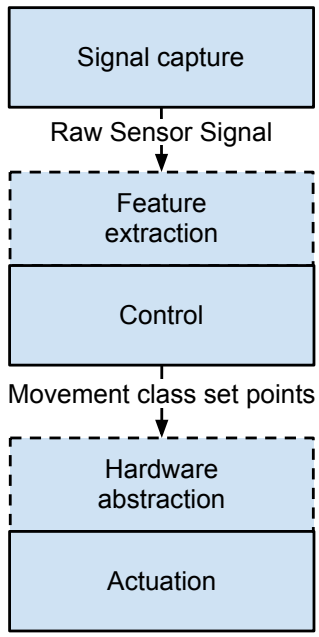


Figure B.5: This is a simplified version of the division of roles I suggest.

B.1.1 A Note on Bandwidth

The CAN protocol has a relatively high bit rate (up to 1 Mbit/s for bus lengths below 25 m [11]). Idstein et al. [12] report a bus utilization of 61% for a system transmitting 16 EMG signals with a resolution of 16 bits and a rate of 1 kHz. This means that bandwidth will not be a problem until prostheses become significantly more advanced than they are today. This also means that the possibility of sending features on the bus is not critical at this time, and could be, as mentioned, deferred to a later version of the protocol.

B.1.2 A First Implementation

The following describes the behavior of a possible "first implementation" of the above system architecture in a device profile layer. "First implementation" means the basic, most essential behavior. The following sections will go deeper into the profile layer to look at how to realize this behavior.

Behavior

A system consisting of:

- A control unit
- Enough electrodes for the control unit's control scheme
- A terminal device of any kind, and
- A bus arbitrator

all connected to the bus, will allow the terminal device to be controlled along at least one degree of freedom by way of some sort of activation of the electrodes. The system will behave the same way after every power-on unless a device is added, removed or replaced, in which case, the new behavior should be as similar as possible to the old.

B.2 Data Channels - Setup

A device will use data channels to send or receive data, so it should have a channel for each data type it accepts or provides. As an example: An electrode will provide an EMG output channel, and the control module will provide an EMG input channel.

The profile layer is responsible for specifying how to connect inputs and outputs ("Configuration") in the best possible way. The bus arbitrator sets up the actual links, so we will assume it will also decide which channels to connect to each other.

B.2.1 Configurations

One goal of the device profile layer is to be able to swap one part for another, similar part or to add or remove devices. After the change, the prosthesis should function as similarly as possible to before, but also adapt to changes in complexity.

In essence, there are four different power-up scenarios which require different kinds of configuration:

1. **First power-up**: All channels must be connected according to device and channel profiles.
2. **Restart of an already configured system**: Trivial case of reconnecting a stored connection scheme according to VIDs, PIDs, Serial numbers, and channel indices.

3. **Restart of an already configured system with devices added, removed, or replaced**: A combination of the two previous, involving mapping the functions of the removed device(s) to the functions of the new device(s) to make the new system behave similarly to the old system.

4. **Restart of an already configured system with devices added, removed, or replaced, but the new system has also been configured before**: This is a possibility if, for example, a patient owns different terminal devices for different uses, and switches between them. If the configuration is stored and can be identified, it can be reused.

B.2.2 Storing Configurations

When the configuration of the system is completed once, the configuration should be stored and reused on the next power-up. In the current protocol, each device stores its own configuration. The input channel parameters (table 2.9) contain a "Source's VID and PID" and "Source's SN and Channel Index", which can be written to after configuration, and read later.

But in the event that a device is removed, it might be desirable to know the information stored on the removed device. This would be accomplished if the bus arbitrator were to duplicate all information in its own memory. But if the bus arbitrator role is filled by another device, and this device is removed, the stored configurations are gone.

Another option is to have a dedicated memory device sitting in the socket. Configurations for a particular prosthesis would then be "permanently" available in the prosthesis.

As mentioned in appendix B.2.1, a patient may use different prosthesis parts for different situations, and thus want multiple configurations to be stored. This is certainly possible with the right data structures in the memory node.

A memory node could also store other configuration information. A control unit could use the memory node to store information about which control strategy it uses etc. If this information is stored in a standardized way, another control unit can retrieve it.

There are also other ways to make sure the information is available in the network. Storing in the bus arbitrator has already been mentioned. If the bus arbitrator were required to be a separate node, this would

be a good alternative. Other alternatives would be to distribute the information among the nodes, e.g. by duplicating the information one or more times.

B.2.3 Manual Configuration

No matter how well the device profile layer can configure the system, there should be a possibility for manual configuration. People's preferences differ, and giving the patient part in the customization of their prosthesis will help them get more out of it [13].

From *PDCP Info (2011 05 04)* [3]:

> *(The) Bus Arbitrator (is) responsible for binding devices onto (the) network and providing an interface for Software Applications to configure the devices and device interconnections on the PDCP bus system.*

This could be taken to mean that the bus arbitrator should be the point of outside access to the system. Regardless, it is a natural choice, since all manual configuration will probably reach it eventually.

Optimally, such configuration should be simple enough for the patient to use at home. In terms of human interfaces, there are many possibilities:

- On the prosthesis:
  - Buttons/knobs/joystick
  - Display
  - Touchscreen
- Computer software via wired or wireless connection to prosthesis.
- Dedicated handheld device via wired or wireless connection.
- Smartphone or tablet app, via wireless connection.

Manual configuration should allow the user to choose which electrodes to use for which movement, which control strategy to use if more than one is available, which movement classes to use and how to switch between them, and tuning of parameters such as threshold and sensitivity.

B.2.4 Control Units – Transparent or Opaque?

Consider a control unit with multiple possible control strategies. Should each strategy have its own set of input channels, or should they share input channels since only one strategy will be used at a time? In the first case, the control strategy used can be determined from the input channels used. In the second case, the control strategy would have to be stored separately if record of it is to be kept.

However, in reality both cases would use a channel parameter to hold the strategy information. The parameter would be static in the first case and variable in the other. Since there are no real disadvantages of the variable parameter, the second case, with shared channels, seems the better choice.

B.3 Data Channels - Transmission

B.3.1 Information Integrity

What safeguards, if any, should be implemented to ensure the integrity of packets sent on channel links?

Sources of transmission and reception errors include noise on the wire, high bus loads, and buffer overflows. CAN itself already has quite a system for detecting these errors, through ACKs, error flags, overload flags, and retransmissions. This makes it robust to packet loss. In most use cases for PDCP (low noise, low to medium bus utilization) packet loss should be minimal. Idstein et al. [12] report:

> *Bus utilization was, on average, 62% for the upper limb system and 73% for the lower limb system with no loss of data or perceivable latency.*

Extra measures including explicit retransmission should be unnecessary.

A sequence number can still be useful, because it will enable detection of bad transmission (packet loss, or faulty nodes transmitting the same packet over and over) which is useful to know whether or not it is acted upon. In addition it enables transmission of "special" packets, as the meaning of the packet can be dependent on the sequence number. An example of this would be defining sequence number 0 as containing configuration data such as data rate and resolution.

B.3.2 Byte Format on Data Channel Links

Sender and receiver must agree on what format the data is. This could be explicit or implicit.

As discussed in appendix B.3.1, if each frame contains a sequence number, one sequence number can be used for configuration, and thus to explicitly inform of the byte format. One disadvantage of a sequence number is that it takes up space in the payload, reducing the net bit rate. Another disadvantage is that if this configuration packet is somehow lost, the rest of the correspondence will be unintelligible.

Another option is to put format information in a channel parameter. The bus arbitrator would be required to inform input channels of the output channel's format information. The advantage of this is that the format information is explicit, while also reserving the data channel link for only data. A disadvantage is that the format information cannot necessarily be changed after the channel link has been set up.

A third option is to have dedicated channel links for metadata. Making data links in pairs would be a very flexible setup. The disadvantage of this is halving the number of possible channels in each device. An intriguing option is a broadcast channel, which can be used for metadata, but this would require support in the lower levels of the protocol, and may also be against the principles of PDCP.

A last option is that all format information is implicit. The type of the output channel would dictate the correct way to interpret the signal. This solution would, however, be troublesome, because input channels can be matched with output channels of other types. The input channel would then need to keep a record of all channel types and all possible ways of interpretation.

In any case, more detailed studies should be conducted into the optimal sampling rate and resolution of EMG (and other) signals, so that good standards for the byte format can be made.

B.4 Profiles

PDCP, as it stands now, allows devices and channels to specify their "type and profile" and "descriptor". In a device profile layer, these numbers should be the basis for configuring the network, and must therefore contain most of the metadata needed to make a configuration. Especially when reconfiguring the system after parts have been replaced, it is important that the new device can be compared to the old device by use

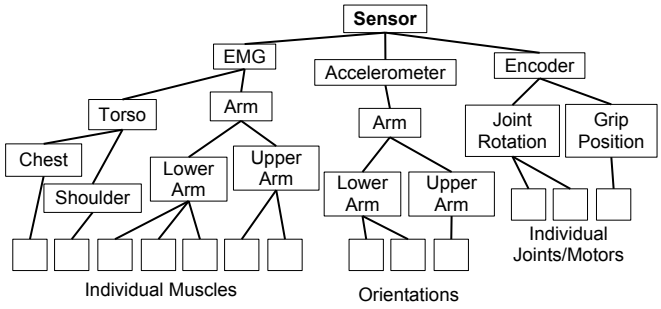


Figure B.6: An example profile hierarchy for sensor channels.

of the profile information. Also, when the system is started for the first time, the automatic configuration should be logical, even if some manual configuration will usually be done afterwards.

B.4.1 Channel Profiles

It seems natural that most of the profile layer functionality should be implemented using channel profiles rather than device profiles. This is because a physical device can perform multiple roles, while a channel has a single function. The same tendency is seen in both USB and in Bluetooth 4.0. In USB, many devices will have their classes specified in the interface descriptor rather than the device descriptor. In Bluetooth 4.0, a device can support many profiles, and profiles themselves are mostly specified in terms of individual services.

B.4.2 Device Profiles

Even though the most important profile information will reside in the channels, it would probably be useful to also utilize the device-wide "Type and Profile". One possible use is to specify whether the device is a sensor, control unit, terminal device, or a combination of these.

B.4.3 Tree Structure

A natural way to represent both information and relation is a tree structure. Examples of such tree structures for sensor channels and movement class set point channels can be seen in fig. B.6 and fig. B.7.

A channel's profile could be any of these nodes. A node can then be identified by a sequence of numbers, which we will call the "profile code".

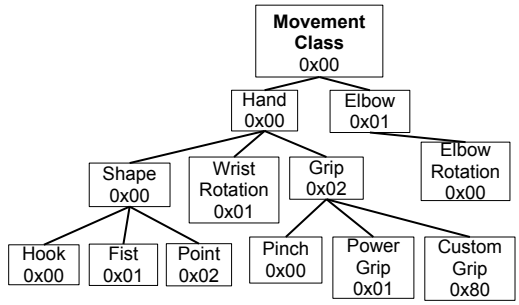


Figure B.7: An example profile hierarchy for movement class set point channels, with enumeration.

E.g., if each node is assigned a number as in fig. B.7, then a "Power Grip" movement class could be identified by the profile code `0x00000201`. A portion of the numbers, for example above `0x80`, could also be reserved for non-standard devices, as with the "Custom Grip" in fig. B.7.

The final structure of this hierarchy would have to be carefully considered, because once it is official, nodes can only be added to the tree in certain ways, so that the profile code of each node is unchanged.

The control strategy parameter described in appendix B.2.4 can also be made from a tree structure. That way, a new control unit can be matched to the old, to provide similar behavior when replaced.

B.4.4 Channel-Matching

The way channels will be matched is that when a device sends an *Update Data Channel Request* message, the bus arbitrator will find the most similar output channel and connect it. Similarity is measured by how far down in the tree structure the profiles are alike, i.e. how many bytes (from the front) in the profile code are equal. E.g., the similarity of pinch and power grip is 3 (the depth of "grip").

In PDCP, multiple input channels can be connected to one output channel, but this is not always desirable. For example, an EMG sensor can only be used once as control input. This can be solved by specifying if the input channel needs exclusive use of the output channel. Only one input channel with exclusive use can be connected to a given output channel.

Sometimes, particularly for the input channels of the control unit, either all or none need to be connected. E.g. for a two-site system, it is useless for only one of the two input channels to be connected to an

electrode. This can be solved if each channel has a number (we can call it the "channel group"), which will be the same for channels that need to all be connected.

If a control module is flexible, i.e. it can use different control schemes dependent on the terminal device and the number of electrodes available, it will need to have channels for all possibilities. To know which channels to use, it should wait until the terminal device is configured, then decide which input channels to request[2] to be connected. It can prioritize its channels by requesting the most important channel first, etc. If there are not enough sensor channels to supply its needs, the bus arbitrator can deny the requests[3].

B.4.5 Channel-Matching in Previously Configured Networks

In some cases, devices may have "Source's VID and PID" and "Sources SN and Channel Index" filled incorrectly, if:

- A channels source has been removed or replaced.
- A device has been configured in another system.

One possible way to cope with this is to make the memory node accessible to all devices, which can retrieve the correct configuration. This would, however, be problematic if the memory node has multiple configuration stored. The devices do not know which configuration to use. One possible solution is to have the bus arbitrator tell every device which of the configurations to use. It is still a slightly complicated approach, because the devices must first be granted a channel to the memory node, then be told the configuration index, then read the configuration, then be granted channels to each other.

It is also possible for the bus arbitrator to write the actual configuration to each device. This would be a more straight-forward approach.

A third possibility is to have the devices act the same in all scenarios, and have the bus arbitrator guide the setup. It would then have to deny "incorrect" *Update Data Channel Request*s and coax the right requests out of the devices.

The exact way to solve this problem will have to be elaborated on in a later study.

[2]Using the *Update Data Channel Request*.

[3]More diverse response codes would be useful for this purpose.

B.4.6 Profiles as the Basis for Message Format

A message could have a profile corresponding to a node in the tree structure. Each node could then possibly have different value format. Since channels of different kinds could be connected, what profile should the message have? To avoid having to know the value format of all nodes in the tree, a message profile should be the node where the two channel profiles branch away from each other[4]. This way, each channel is required to know the value format of each node from the root node to its own profile.

As an example (see fig. B.7), if `0x00000201` (Power Grip) set points are used to control `0x000001` (Wrist Rotation), then the signals have profile `0x0000` (Hand).

This approach would necessarily mean that details are lost when transmitting on other message formats, and conversion rules must be established, but the details lost would be details irrelevant to channels not of the same type, and thus not understandable by the input channel.

An alternative to this approach is to have a description of the format in a channel parameter, including such things as the unit (V, m/s^2, etc.), range, scale, and maybe also the byte format (signed/unsigned, int/float, bit length, etc.). The input channel would then read the output channel's message format parameter, and ideally understand messages from this. The disadvantage of this, is that channels must know of all types of formats. Some format information, such as scaling and range could be explicit, but all information on units must be standardized, and channels can only use known units. If an output channel has a custom profile, it can still not use a custom format, unless it makes duplicates of its channels, with standard profiles, so they can be used by all.

B.5 Fringe Cases

As it is not clearly defined how the bus arbitrator role will be fulfilled, there may be situations where more than one device is ready to take the role. There is no way to solve this in the current protocol. One possibility would be that all bus arbitrators must send a message as soon as it is turned on. Since no device is bound to the bus, other devices will be using priority 3. Other priority values could be used to negotiate between multiple bus arbitrators.

Prostheses which contain both a prosthetic elbow and a prosthetic

[4]This is incidentally the node that determines their similarity.

hand are commercially available. In implementations with PDCP, these systems could easily end up containing multiple control units because the elbow and hand could come with one control unit each. This would also happen if a regular system (with one terminal device) with a separate control unit was fitted with a terminal device with its own control unit. In the first example, both control units are needed, while in the second, one control unit is superfluous.

The problem with the elbow-hand example is that in most cases, they will be controlled one at a time, with one set of electrodes. The control units would need to hand over control to one another, which would need to be done through channels, or through a change in the lower layers of PDCP. A quick fix is to demand that a system must contain a control unit capable of controlling all terminal devices present in the system.

In the case of duplicate control units, the built-in control unit can avoid the collision by never setting up its set point channels on the bus. This would mean that the separate control unit never takes part on the bus because no terminal device connects to it. Or the built-in control unit can choose to always attempt to connect its set point channels first, to allow a separate control unit to take control if it is present. This might give more consistent behavior. The user could in any case later manually configure the system to use the other control unit.

APPENDIX B. ELABORATION ON THE IMPLEMENTATION OF DEVICE PROFILES IN PDCP

Appendix C

Attachment Inventory

The following is the contents of the attachment to this report.

- **Code**: Contains all the code produced in the course of this work.
  - **Host**: Contains all code for the PC.
    - **hostkode**: Contains all code produced by the author. The contents are described in section 5.2.3.
    - **unity**: Contains the *Unity* unit test framework.
  - **Nodes**: Contains all code for the PDCP nodes.
    - **Bus Arbitrator**: Contains the pre-compiled hex files for the bus arbitrator software.
      - **BusArbitrator.hex**: Bus arbitrator software. The bus arbitrator API communicates at a baud rate of 128000.
      - **BusArbitrator_57600.hex**: Bus arbitrator software. The bus arbitrator API communicates at a baud rate of 57600.
    - **Devices**: Contains the code for the other, non-bus arbitrator nodes.
      - **klientkode**: Contains the MPLAB X project.
- **Documents**: Contains relevant documents.
  - **PDCP Documentation**: Contains documentation of the low level PDCP including the Bus Arbitrator API.
  - **PDCP Nodes**: Contains documents pertaining to the hardware of the PDCP nodes.
  - **Possibility Study of Implementing Device Profile Layer in PDCP: Øyvind Rønningstad (2012).pdf**: This is [1], the report produced by the author in 2012.
- **Videos**: Contains videos of 3 of the 5 tests.

APPENDIX C. ATTACHMENT INVENTORY

Appendix D

Detailed Printout of the System in Test 5

This printout was generated immediately after the conclusion of Test 5.

```
ronningstad@oyvind:~/Dropbox/masteroppgave/hostkode$ ./main.out print
n_devices: 3
n_missing_devices: 0
n_input_channels: 4
n_output_channels: 4
n_channel_links: 3

device 0 (8A93010):
    name: D
    vendor id: 2
    product id: 2
    serial number: 4
    node_id: 0
    type: 0
    profile: 0
    # of input channels: 0
    # of output channels: 1
    input channels:
    output channels: 8A93108,

device 1 (8A93030):
    name: D
    vendor id: 2
    product id: 2
    serial number: 5
    node_id: 0
    type: 0
    profile: 0
```

```
    # of input channels: 2
    # of output channels: 3
    input channels: 8A930A0, 8A930B8,
    output channels: 8A93120, 8A93138, 8A93150,

device 2 (8A93050):
    name: D
    vendor id: 2
    product id: 2
    serial number: 6
    node_id: 0
    type: 0
    profile: 0
    # of input channels: 2
    # of output channels: 0
    input channels: 8A930D0, 8A930E8,
    output channels:

input channel 0 (8A930A0):
    channel index: 1
    type: 7200
    profile: 2200
    transfer enable: 99
    required ichans: 0
    desired ichans: 0
    terminal profile: 0
    source channel node id: 0
    source device node id: 0
    source vid: FFFF
    source pid: FFFF
    source snum: FFFF
    source ci: FF
    source type: 0
    source profile: 0
    owner: 8A93030
    channel link: 0


input channel 1 (8A930B8):
    channel index: 2
    type: 5200
    profile: 2110
    transfer enable: 99
    required ichans: 0
    desired ichans: 0
    terminal profile: 0
    source channel node id: 0
    source device node id: 0
    source vid: 2
```

```
    source pid: 2
    source snum: 4
    source ci: 1
    source type: 5000
    source profile: 2120
    owner: 8A93030
    channel link: 8A931E0


input channel 2 (8A930D0):
    channel index: 1
    type: 5300
    profile: 1110
    transfer enable: 99
    required ichans: 0
    desired ichans: 0
    terminal profile: 1110
    source channel node id: 0
    source device node id: 0
    source vid: 2
    source pid: 2
    source snum: 5
    source ci: 3
    source type: 5000
    source profile: 1100
    owner: 8A93050
    channel link: 8A931EC


input channel 3 (8A930E8):
    channel index: 2
    type: 7300
    profile: 1220
    transfer enable: 99
    required ichans: 0
    desired ichans: 0
    terminal profile: 1220
    source channel node id: 0
    source device node id: 0
    source vid: 2
    source pid: 2
    source snum: 5
    source ci: 5
    source type: 7000
    source profile: 1220
    owner: 8A93050
    channel link: 8A931F8
```

```
output channel 0 (8A93108):
    node id: 0
    channel index: 1
    type: 5000
    profile 2120
    node id 0
    transfer enable: 63
    required ichans: 0
    desired ichans: 0
    owner: 8A93010
    # of channel links: 1
    channel links: 8A931E0,


output channel 1 (8A93120):
    node id: 0
    channel index: 3
    type: 5000
    profile 1100
    node id 0
    transfer enable: 63
    required ichans: 32
    desired ichans: 0
    owner: 8A93030
    # of channel links: 1
    channel links: 8A931EC,


output channel 2 (8A93138):
    node id: 0
    channel index: 4
    type: 5000
    profile 1210
    node id 0
    transfer enable: 63
    required ichans: 64
    desired ichans: 0
    owner: 8A93030
    # of channel links: 0
    channel links:


output channel 3 (8A93150):
    node id: 0
    channel index: 5
    type: 7000
    profile 1220
    node id 0
    transfer enable: 63
```

```
    required ichans: 32
    desired ichans: 0
    owner: 8A93030
    # of channel links: 1
    channel links: 8A931F8,

channel link 0 (8A931E0):
    input channel:8A930B8
    output channel:8A93108
    link status: 2

channel link 1 (8A931EC):
    input channel:8A930D0
    output channel:8A93120
    link status: 2

channel link 2 (8A931F8):
    input channel:8A930E8
    output channel:8A93150
    link status: 2
```