NTNU – Trondheim
Norwegian University of
Science and Technology

Term Project

TTK 4550 - Medical Cybernetics

Possibility Study of Implementing Device Profile Layer in PDCP

Author:
Øyvind Rønningstad

Supervisor:
Terje Mugaas

Norwegian University of Science and Technology

Faculty of Information Technology, Mathematics and Electrical Engineering

Department of Engineering Cybernetics

Fall 2012

Abstract

This report is the result of a term project at the Department of Engineering Cybernetics at NTNU. It discusses the implementation of a device profile layer for the Prosthetic Device Communication Protocol (PDCP), and open-standard bus protocol for use in powered prostheses. The purpose of the profile layer is to standardize communication between components, to ease building and configuration of individual prosthesis solutions, and to maximize interoperability of compatible parts.

The report includes a background study into existing powered prosthesis solutions, the existing version of PDCP, and into how device profiles are used in other protocols. Later, the task of adding device profiles to PDCP is discussed, and through the elaboration, the outline of one possible implementation is presented. The concepts of configuration storage and standardized message formats are found to be essential in a profile layer for prostheses.

NTNU
Norges teknisk-naturvitenskapelige universitet

Fakultet for informasjonsteknologi, matematikk og elektroteknikk
Institutt for teknisk kybernetikk

Prosjektoppgave

Kandidatens navn: Øyvind Rønningstad

Fag: Teknisk Kybernetikk – Biomedisinsk Bevegelse

Oppgavens tittel (norsk): Mulighetsstudie for Implementasjon av et Profil-lag i PDCP.

Oppgavens tittel (engelsk): Possibility Study of Implementing Device Profile Layer in PDCP.

Oppgavens tekst:
PDCP (Prosthetic Device Communication Protocol) has been developed at UNB, in cooperation with NTNU with the goal of reducing cost, increasing reliability, streamlining system design, as well as easing the transition to more advance prosthesis systems. PDCP is maturing, and has recently been hardware implemented. It is now desirable to look at the possibility of extending PDCP with a device profile layer to further interoperability and interchangeability of prosthesis parts from different manufacturers.

The following tasks shall be addressed:

1. Acquaint yourself with the types of prostheses (including combined prostheses) which exist and what functions and what modes of control they support, and how they are controlled today.
2. Familiarize yourself with the Prosthetic Device Communication Protocol.
3. Look into how different communication protocols use profiles to represent information in message exchange.
4. Discuss how profiles can represent information on the PDCP bus, with respect to prosthesis control.
5. If time allows, outline a suggestion for implementation of a profile layer in PDCP, with the aim of achieving interoperability between prosthesis parts from different manufacturers.

Oppgaven gitt: 2012-08-20

Besvarelsen leveres: 2012-12-21

Utført ved: Institutt for teknisk kybernetikk

Veileder: Terje Mugaas, SINTEF

Trondheim, 2012-12-18

Geir Mathisen
Faglærer

Preface

Applying my education in engineering cybernetics to the medical domain is both exciting and rewarding. I hope this report can be a useful contribution to the field of medical engineering.

Acknowledgements

First, I want to thank my advisor, Terje Mugaas, for our encouraging meetings with good discussions, and for taking the time to read my report, several times, and suggesting improvements.

Secondly, I want to thank Yves Losier of UNB, Canada for taking the time to answer my many questions about PDCP, giving me a more complete comprehension of the foundation of my project.

Further, I would like to thank Øyvind Stavdahl for the idea for the project and for his inspiring passion and optimism; and my supervisor, Geir Mathisen, for handling the administrative aspects of the project.

I want to thank Ole Morten Haaland for his comments and constructive criticism on my report.

And finally, I want to thank my wife for being awesome, and also for proofreading my report.

Note on Process

My original advisor went on sudden leave for the majority of the semester. This meant that no one at the institute had much expertise on the PDCP protocol. I was handed some documents, but these were incomplete. I was not, however, aware of this until I, relatively late, came in contact with Yves Losier, the main author of the protocol, who provided me with additional information. Because of this, some of the report is written based on thoughts I had when my information was incomplete.

Øyvind Rønningstad

December 20, 2012
Trondheim

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The purpose of this report is to look at how to introduce a device profile layer into the PDCP communication protocol.

1.1 Background and Motivation

Today's market of powered prostheses consists almost exclusively of proprietary solutions even though different manufacturers provide functionally very similar systems. Existing systems are also not very scalable. Typically, each sensor requires its own set of wires to the terminal device, which makes it very expensive to modify the system, and each wire is a component that can fail, especially when they must lead through moving joints.

The PDCP protocol has been developed at UNB[1], in cooperation with NTNU[2] as a solution to these and other problems with existing systems. The goal of creating a standardized bus protocol is, in the words of Stavdahl and Mathisen[1]:

- *Reduced wiring and thus production cost and hardware failure rate.*
- *Advanced coordinated control schemes with a large number of sensor signals and control variables.*
- *Remote adjustment, fault diagnosis and software upgrades.*

[1] University of New Brunswick, Fredericton, Canada

[2] Norwegian University of Science and Technology (Norges Teknisk-Naturvitenskapelige Universitet), Trondheim, Norway

- *Interoperability and thus improved interchangeability of different devices.*

The biggest motivation for extending the PDCP protocol with a device profile layer is found in the last point: To further improve interoperability between, and interchangeability of, parts from different manufacturers or with different functions. The goal is to have different parts of a prosthesis, such as sensors and terminal devices, function in a "plug-and-play" fashion, so that systems can be built, and parts can be replaced with minimal or no need for configuration.

1.2 Focus of this Project

The first aim of this report is to investigate what a device profile layer is and how other device profile layers are implemented, and also to give an introduction to how prostheses work and to the PDCP protocol as it is at the time of writing.

The second aim of this report is to investigate the possibility of incorporating a device profile layer into PDCP, and outline one possible way to construct such a layer. A profile layer will help decrease the amount of configuration needed to make parts cooperate. Constructing a profile layer involves further standardization of functions codes, parameters, and classification of the functional roles different parts of a prosthesis play, to minimize the amount of basic implementation details that need to be decided by vendor.

Specifying a device profile layer is a very large task, and this report will not nearly finish it, so the first priority will be to provide a basic framework that can later be built on. Also, I am a student with limited experience both in the field of prosthetics, and of standardization, so in many places, I will discuss different approaches instead of giving a definitive decision. A significant portion of the specification should be left to a group of experts. The scope of this report is more about exploring the possibilities.

Chapter 2

Background

In this chapter, I provide information relevant to the project. I try to give introductions to:

- Prostheses, specifically powered prostheses, including control strategies.
- The Prosthetic Device Communication Protocol
- Device profiles, and examples of how they are used in the USB and Bluetooth 4.0 protocols.

2.1 Powered Prostheses

A prosthesis is anything that acts as a substitute for a missing body part. The purpose of a prosthesis can be entirely cosmetic, or it can provide a *functional* replacement as well. A powered prosthesis has joints with motors, usually to emulate the function of the limb it is replacing. The purpose of powered upper limb prostheses is to return to an amputee some of the ability to grasp, hold and manipulate objects.

To control the prosthesis, the intent of the person wearing it must be estimated. The most common way to accomplish this is through myoelectric sensors, which can sense contractions in muscles. Electrodes can be placed on the skin outside of muscles on the remaining part of the limb allowing the person to control the prosthesis by contracting these muscles.

Figure 2.1 shows a generalization of most prosthesis control schemes, along with some examples of both commercially available and research-only control schemes which fit the model.

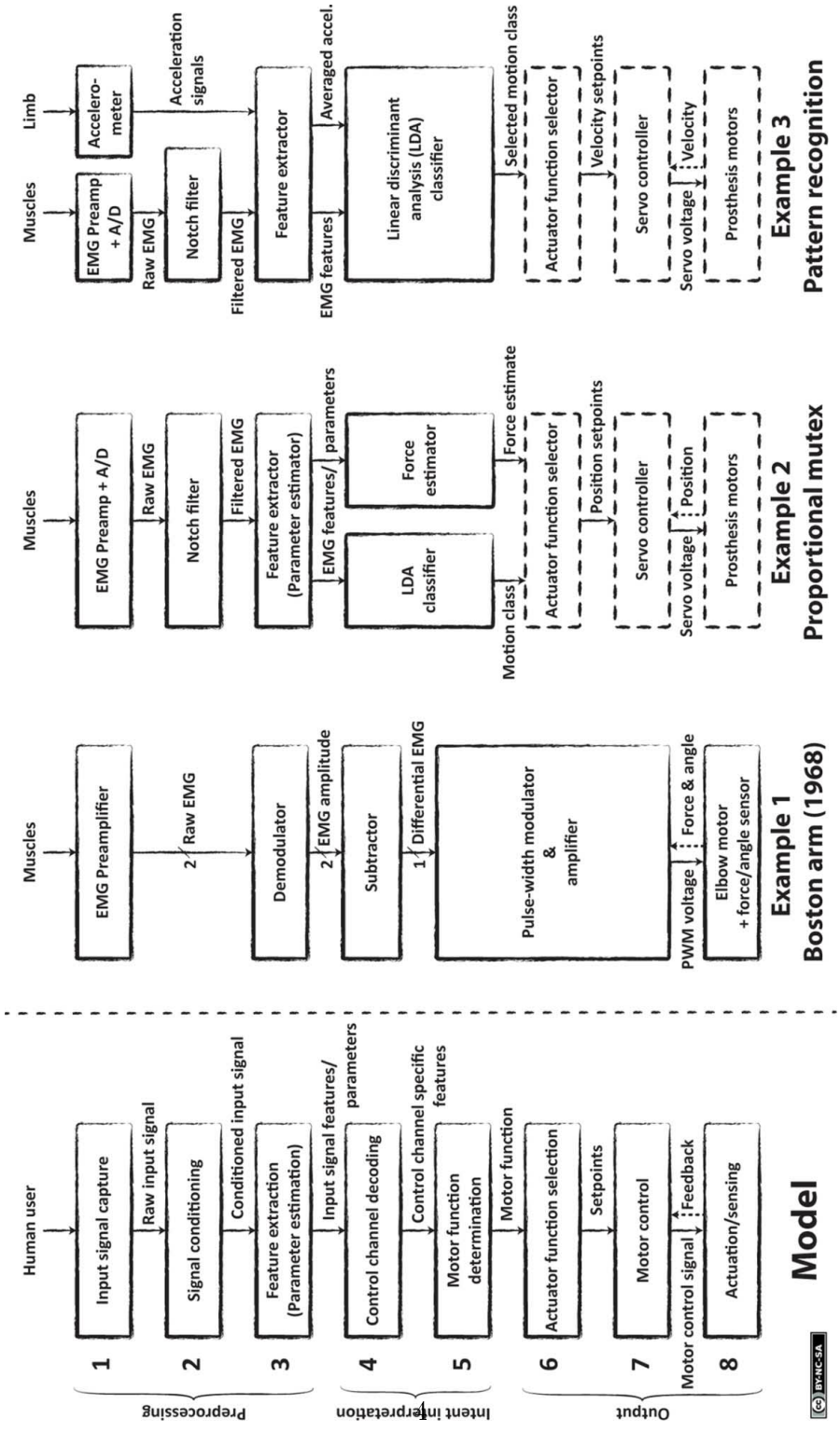

Figure 2.1: Conceptual model of a general prosthesis control scheme, as proposed by Fougner et al. [2].

2.1.1 Myoelectric Signals

The myoelectric signal is measured by placing an electrode on the skin outside a muscle. The signal measured is a combination of the action potentials sent to all motor units in the muscle. The reason why it is measured on the muscle and not on the nerve which carries the signal from the brain is that, in the muscle, the signal is branched and multiplied, effectively amplifying the signal so it is easier to detect. Also, a single nerve will most often carry many signals, to other muscles as well, which will be hard or impossible to separate.

The use of electrodes to measure myoelectric signals carries a few challenges. The amplitude of the signal depends on the connection between the electrode and the skin, which means that if pressure is applied to the electrode, it will register increased activation. It also means that sweat on the skin will affect the signal. Also, if the electrode moves across the skin, this will create noise on the signal.

Another challenge when using myoelectric signals to control prostheses is that, often, the muscles being measured on are also the muscles that are used to support the arm and prosthesis, which means that the signal will change depending on the orientation of the arm and prosthesis.

Research is currently being conducted into the use of pressure sensors and accelerometers in myoelectric sensors to cope with these problems.

2.1.2 Today's Commercially Available Prostheses

Most commercially available myoelectric prostheses use one or two electrodes and can control only a single degree of freedom (DOF) at a time, e.g. grasp or rotation. Many of these allow just 3 states: One movement speed in each direction, and a rest state. This control strategy is called "**on-off**" control and makes it possible to control the prosthesis with only one signal (single-site system). In single-site systems, a processed myoelectric signal (PMES)[1] from an electrode is used to decide which of the three states to use, see figure 2.2 for an example.

With two electrodes (two-site systems), two different muscles are measured, and used to control opposite movement directions (open/close, pronate/supinate, flexion/extension). Activating one muscle will cause

[1] The processing usually consists of rectification and filtering (smoothing), giving the average amplitude.

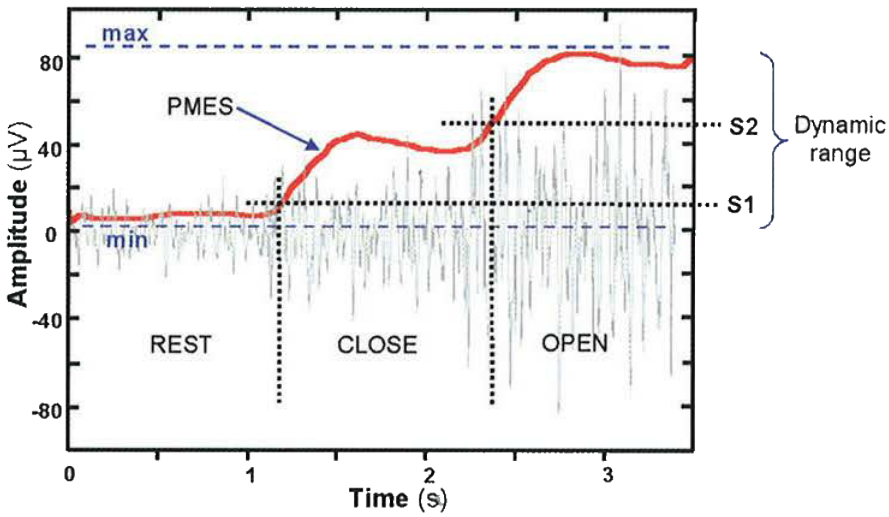

Figure 2.2: An example of a single-site (single electrode) control strategy for a prosthetic hand. When measured muscle activation is low, the prosthesis will not move; when activation is moderate, the hand will close with a set speed; when activation is high, the hand will open with a set speed. Taken from Muzumdar [3, p. 45, figure 3.10].

movement in one direction, while activating the other muscle will cause movement in the opposite direction.

Two site systems can be used for both **on-off** control and **proportional** control. In **proportional** control, the difference in PMES across the two electrodes is used to decide both movement *direction* and movement *speed*, which allows for both fast and delicate movements. Because of the amount of noise in myoelectric signals, the difference must usually be above a certain threshold to trigger movement. This MES noise is also one reason why **on-off** control is as common as it is, even in two-site system, since the on-off control strategy behaves more predictably than more complicated strategies.

When a two-site system is used to control a prosthesis with more than one DOF, only one DOF will be controlled at a time, and the active DOF can be switched, for example by co-contracting the two muscles. In this way, very complicated prostheses can be controlled with two-site systems through the use of movement classes such as "hand rotation", "pinch grip", "power grip", etc. Co-contractions will then cycle through the available movement classes. These complicated systems can often become *too* complicated to be practical, and often, one-DOF systems are preferred over even a two class (rotation, grasp) system, because the more complex system offers little improvement, and can also be heavier, less wieldy, and more fragile.

2.1.3 The Future of Myoelectric prosthesis Control

As mentioned, there is research being conducted on the use of other sensors to supplement the myoelectric control signals. These involve pressure sensors and accelerometers inside electrodes, rotational sensors inside joints, and pressure, texture, and slip sensors on terminal devices. There is also ongoing research into the use of additional myoelectric sensors to control more DOFs simultaneously, for example by using pattern recognition. Pattern recognition systems will often extract multiple features from each EMG signal, giving a larger number of total features to perform the pattern recognition on. [2] list a number of features that can be extracted from an EMG signal; the list is reproduced in table 2.1

2.2 Prosthetic Device Communication Protocol

The Prosthetic Device Communication Protocol (PDCP) is a bus communication protocol designed specifically for use in prostheses. PDCP is

MES Features
Mean Absolute Value
Mean Square Value
Myo-pulse
Number of Turns
Root-mean Square
Slope Sign Changes
Willison Amplitude
Windowed Fourier Transform
Waveform Length
Wavelet Packet Transform
Wavelet Transform
Zero-Crossings

Table 2.1: Features that can be extracted from a MES. [2].

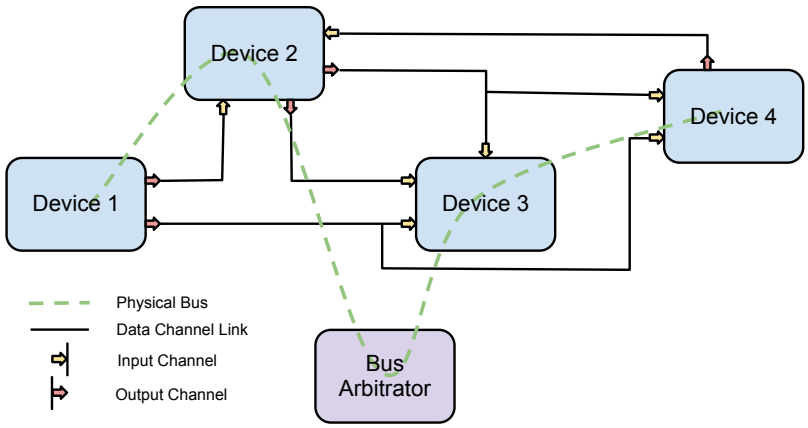

Figure 2.3: Example network structure of PDCP, including both physical and logical links. Not drawn: All devices have an inherent logical link to the Bus Arbitrator.

built on top of the Controller Area Network (CAN) protocol. Figure 2.3 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.[2]

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

2.2.1 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[3] to the bus by the bus arbitrator[4], 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.

2.2.2 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

[2] 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.

[3] "Binding"

[4] The exception being *Bind Device Request*s.

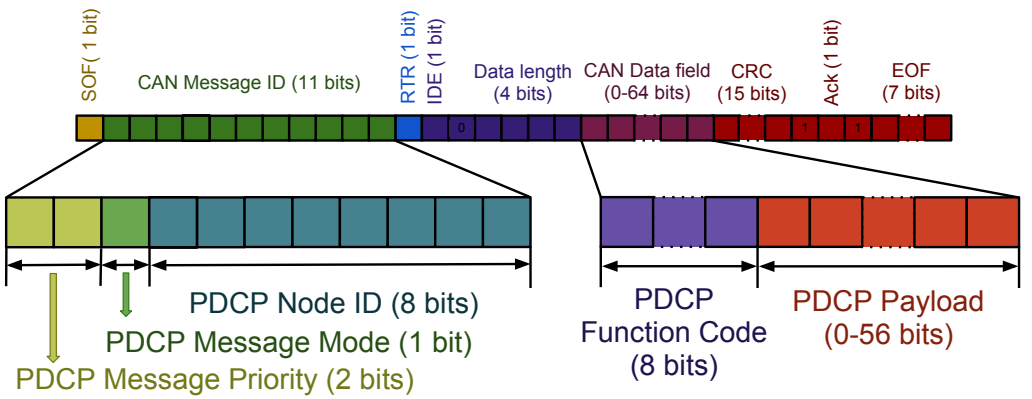

Figure 2.4: Bit layout of a PDCP message[4], in the context of a CAN message[7]. The function code field is only used in communication with the bus arbitrator.

parts: The **message priority**, **message mode**, and **node ID** fields, as seen in figure 2.4.

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[5].

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 2.2.

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 2.2: Use cases of message modes and node ID types.

CAN's arbitration scheme works in such a way that if two devices

[5]This is reflected in [6].

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 prioritized 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 2.3, with corresponding response codes in table 2.4.

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 2.3: List of all function codes of the PDCP protocol[4]. Missing function codes are deprecated.See table 2.4 for a list of responses. Tables 2.5 to 2.7 contain the message structures of some of the message types.

2.2.3 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[6]. 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

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 2.4: List of all response function codes of the PDCP protocol[4]. 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 2.5: 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 2.6: 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 2.7: 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 2.8: 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.

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.

2.2.4 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 2.6 and 2.7. 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 2.9 to 2.11 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.

2.2.5 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 2.5 (function code `0x01`) using the requested node ID together with message mode 0 and priority 3 (see table 2.12 for the full data frame).

The bus arbitrator's response (function code `0x81`) can also be found in table 2.5. 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 2.13 for the frame format of the Bind Request Response.

[6] 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 2.2.

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 2.9: 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 2.10: 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 2.11: Output channel parameters.

The *Update Data Channel Request* (table 2.8) 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 2.12: 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*.
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.

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 2.13: 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 2.14: The packet structure of a data channel link packet.

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[7] and update *Source's Node Ids* with the appropriate device ID and output channel ID.
   - 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.

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

2.2.6 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 2.14 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.

2.3 Device Profiles

A device profile layer is a system of context for information exchanged via a communication protocol.

The purpose of a digital communication protocol is, essentially, to transport bits over some distance. Different protocols provide different amounts of context for the information.

Some protocols provide only the barest form of context, for example: RS232 provides no more context than the grouping of bits into bytes.

Other protocols allow for contextualization, but don't define the exact meaning. An example of this is the CAN protocol, which provides a message ID, but does not provide any specific *meaning* to the values the ID can take. Most protocols are of this kind.

Still other protocols give specific meaning to the transmitted information, and often they will also provide some kind of organization or hierarchy to the information. This last layer of specification can be understood as a device profile layer, though it is called different names in different protocols.

A profile layer for PDCP should give the necessary context to allow independently manufactured devices to communicate, and understand each other.

The following sections will look at how a device profile layer is implemented in USB and Bluetooth 4.0.

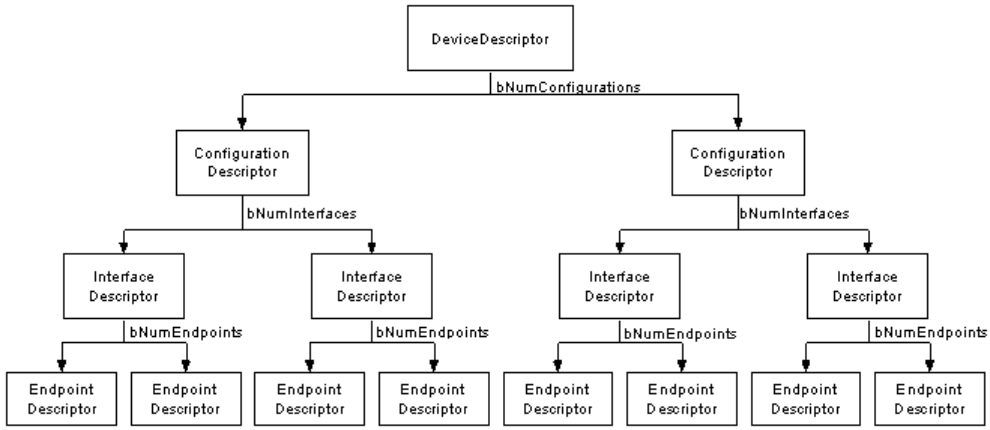

Figure 2.5: The hierarchy of USB descriptors. Taken from *USB in a Nutshell* [8, page 5, Ch. 5: "USB Descriptors"]

2.4 Device Classes in USB

USB (Universal Serial Bus) is already ubiquitous in the domain of personal computers, and is making its way into other areas, such as embedded computing. It is characterized by being very easy to use; it "just works".

2.4.1 Descriptors

When a USB device connects to a host, the host will ask the device to describe itself. The description of a device consists of a number of *descriptors*. There are 4 kinds of descriptors: **Device**, **configuration**, **interface**, and **endpoint**.

A device can only have *one* **device descriptor**. The device descriptor will specify the number of configurations the device can have, and each configuration must have its own **configuration descriptor**[8]. The configuration descriptor specifies the number of interfaces associated with it, and each of these must have an **interface descriptor**. Lastly, the interface descriptor specifies the number of endpoints it has, and each endpoint has an **endpoint descriptor**. In other words, the descriptors form a sort of hierarchy, with the **device descriptor** at the top and **endpoint descriptors** at the bottom, as shown in fig. 2.5.

[8] In practice, devices usually have only one configuration[8, page 5], and for the purpose of this explanation, we will assume a device has only one configuration.

Video Streaming Interface		
Base Class	Subclass	Protocol
`0x0E` (Video)	`0x02` (Streaming)	`0x01`[9]

Image Capture Interface		
Base Class	Subclass	Protocol
`0x06` (Image)	`0x01` (Still Image Capture)	`0x01`[9]

Table 2.15: USB Example: Interfaces of a webcam which can also capture still images

2.4.2 Device Classes (Profiles)

A USB device supports one or more device classes. Device classes consist of a base class, a subclass and a protocol, creating a hierarchy of classes. The supported class(es) can either be specified in the device descriptor, if only one class is supported, or in the interface descriptors, allowing each interface to represent its own class. Custom-made classes must have the device class `0xFF`. More information about available classes can be found in *Approved Class Specification Documents* [9].

Example

A webcam which can also take stills could have the interfaces described in table 2.15.

Since the interface descriptor will contain classes, the device descriptor will not, or the class will be ignored.

For each of these classes, the number of endpoints and their function will be specified in the device class documentation.

2.4.3 Interfaces

Each interface has 32 endpoints identified by a number, 0-15, and a direction, IN or OUT. After the connection has been initialized, the host can access a specific value by addressing a particular interface, end point number and data direction.

[9] This is the only choice of Protocol available for this class.

2.5 Device Profiles in Bluetooth 4.0

The Bluetooth 4.0 protocol is also called "Low Energy", and is aimed particularly towards small devices sharing small bits of data with each other. The device profile layer of Bluetooth 4.0 is called GATT[10].

2.5.1 Master/Slave, Server/Client

Bluetooth connections are established when a master device discovers a slave device and asks to be connected to it. To exchange data, at least one device must act as a server, and the other as a client, although both devices can be both server and client. The server is the keeper of the information, and decides in which way this information may be manipulated by the client (read, write etc.).

2.5.2 Accessing the Information

There are several ways in which the information on the server can be read and modified. A selection of these can be found in table 2.16.

Value[11]	Name	Details
`0x01`	Broadcast	The value is broadcast without any specific recipient.
`0x02`	Read	Regular read.
`0x04`	Write Without Response	Unacknowledged write.
`0x08`	Write	Acknowledged write.
`0x10`	Notify	The server sends the new value to the client when the value changes. The client does not need to poll the server.

Table 2.16: A selection of read and write modes supported by the Bluetooth 4.0 protocol.

2.5.3 Structure

The information is kept in a hierarchy of profiles, services, characteristics and attributes, see fig. 2.6. A Bluetooth 4.0 server supports a number

[10] GATT: Generic ATTribute profile.
[11] See table 2.18

of profiles, a profile contains services, a service contains characteristics, and a characteristic contains a single value and optionally, descriptors. The *characteristic value* contains the actual piece of information (e.g. temperature, heart rate, battery level) which is offered by the server, while *descriptors* contain meta-information (e.g. value type, human-readable description).

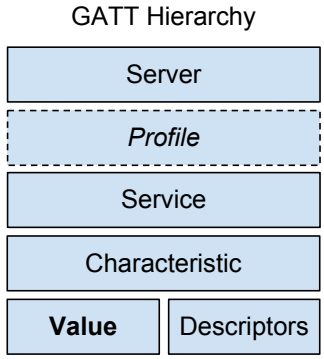

Figure 2.6: GATT hierarchy. "Profile" is dotted because profiles are only implemented implicitly, through the services offered by the server. Thus, a device will automatically support a profile if it offers the right services. "**Value**" is bold because this is where the actual information is located.

2.5.4 Attributes

All information, meta-information, and structural information is contained in "attributes". An attribute contains four things: A **handle**, a **value**, a **type**, and a set of **permissions**.

The attribute **handle** is a 16 bit "address" that is unique to the device, used to unambiguously access the attribute. The attribute **type** is a 128 bit UUID[12]. All standardized Bluetooth UUIDs can be represented as 16 bit shorthand versions. 16 bit UUIDs can be converted to full 128 bit UUIDs by replacing the '`x`'s in the Bluetooth Base UUID (hexadecimal representation):

```
0000xxxx-0000-1000-8000-00805F9B34FB
```

The attribute **permissions** govern what read and write operations are allowed on the value.

[12]UUID: Universally Unique IDentifier.

The contents of the attribute **value** depends on the attribute **type**.

2.5.5 Attributes as Declarations

All services, characteristics, characteristic values and characteristic descriptors are "declared" by attributes. This means that the entire hierarchy can be inferred from the attributes. All services and characteristics also have a unique UUID. Again, standard services and characteristics can be identified by their 16 bit shorthand versions. See *GATT Specifications* [10] to see the standardized Bluetooth 4.0 profiles, services and characteristics.

A service is declared by setting the attribute **type** to the service declaration UUID `0x2800`. The attribute will be on the form found in table 2.17. Every attribute with attribute **handle** in the range between one service declaration and another is grouped with the preceding service. Thus, if service S is to contain characteristic C, then C must be declared after S and before the next service.

Attribute Handle	Attribute Type	Attribute Value	Attribute Permission
`0xNNNN`	`0x2800`	UUID of service	Read only

Table 2.17: An attribute containing a service declaration.[11, Vol. 3, Part G, Ch. 3.1]

A characteristic is declared by setting the attribute **type** to the characteristic declaration UUID `0x2803`. The attribute will be on the form found in table 2.18. Every attribute with attribute **handle** in the range between one characteristic declaration and another is grouped with the preceding characteristic. Thus, if characteristic C is to contain descriptor D, then D must be declared after C and before the next characteristic. All characteristics must contain a characteristic value which must be declared immediately after the characteristic itself (see table 2.19).

2.5.6 GATT Declaration Example

An example heart rate monitor device supports the services "Heart Rate" (thus supporting the "Heart Rate" profile) and "Battery Service". The "Heart Rate" service contains 2 characteristics: "Heart Rate Measurement" (supports notification) and "Body Sensor Location". Both characteristics

Attribute Handle	Attribute Type	Attribute Value	Attribute Permission
0xNNNN	0x2803	Read/write mode (see table 2.16), Attribute handle of characteristic value, and UUID of Characteristic	Read only

Table 2.18: An attribute containing a characteristic declaration.[11, Vol. 3, Part G, Ch. 3.3.1]

Attribute Handle	Attribute Type	Attribute Value	Attribute Permission
0xNNNN	UUID of the characteristic (16 bit or 128 bit)	Characteristic value	Specific to the characteristic

Table 2.19: An attribute containing a characteristic value declaration.[11, Vol. 3, Part G, Ch. 3.3.2]

contain a value, and "Heart Rate Measurement" also contains a descriptor called "Client Characteristic Configuration" (CCCD)[13]. The Battery Service contains one characteristic, "Battery Level". The attributes could then be as in table 2.20.

2.6 Similarities and Differences Between the Profile Layers of USB and Bluetooth 4.0

Both profile layers allow devices to have multiple independent profiles. Both put the data into a hierarchy. We see that much of the functionality is the same, but implementation differs. If we compare USB's interfaces and endpoints to Bluetooth 4.0's Services and Characteristics, and USB's interface descriptors and endpoint descriptors to Bluetooth 4.0's service declarations and characteristic declarations, it is clear that they serve much the same purpose, although the framework around is at times quite different.

USB also employs a hierarchy in device classes, with classes, subclasses and protocols, whereas all Bluetooth 4.0 profiles and services are intrinsically independent, although they can in specific implementations be used hierarchically.

[13]The CCCD is used when the characteristic supports notification or indication. If a client device wants to enable notification or indication on the characteristic, it must write a '1' or '2', respectively, to the CCCD.

Attribute Handle	Attribute Type	Attribute Value	Attribute Permission
0x0001	0x2800 (Service)	0x180D (Heart Rate)	Read
0x0002	0x2803 (Characteristic)	0x10 0003 2A37 (Heart Rate Measurement)	Read
0x0003	0x2A37 (Characteristic value)	0xNNNN	Read
0x0004	0x2902 (CCCD)	0x0000	Read, Write
0x0005	0x2803 (Characteristic)	0x02 0006 2A38 (Body Sensor Location)	Read
0x0006	0x2A38 (Characteristic value)	0xNNNN	Read
0x0007	0x2800 (Service)	0x180F (Battery Service)	Read
0x0008	0x2803 (Characteristic)	0x02 0009 2A19 (Battery Level)	Read
0x0009	0x2A19 (Characteristic value)	0xNNNN	Read

Table 2.20: An example set of attributes in a GATT server.

Bluetooth 4.0 essentially allows all UUIDs that are not derived from the Bluetooth base UUID to be freely used for proprietary services and characteristics, whereas vendor proprietary USB device classes are confined to a single value[14].

[14]Strictly speaking, Vendor ID (VID), Product ID (PID), and Serial numbers can also be used to identify a device.

Chapter 3

Elaboration on the Implementation of Device Profiles in PDCP

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.

3.1 System Architecture

Controlling a prosthesis is essentially the problem of converting sensor data into motor input. Fougner et al.[2] divides the problem into a series of steps, as illustrated in fig. 2.1. 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 3.1 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.

Figures 3.2 to 3.4 are some examples of systems that conform to these specifications. The figures are meant to show the diversity of setups available with the proposed architecture.

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. 3.5. 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".

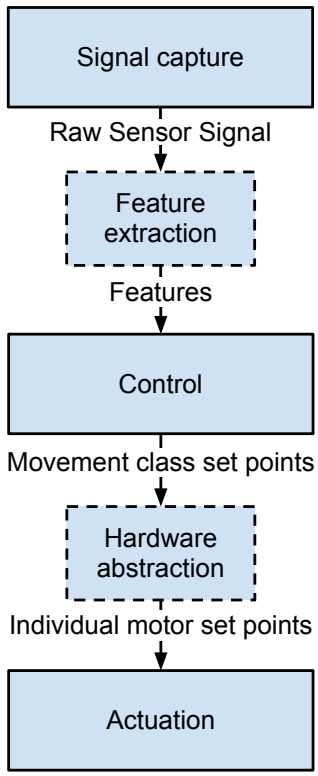

Figure 3.1: This is the division of roles I suggest, and the signals that can travel on the bus.

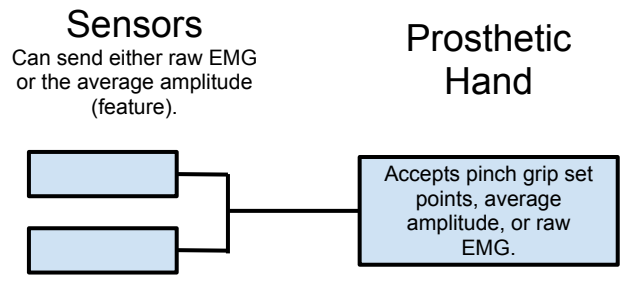

Figure 3.2: Example 1 of a prosthesis system conforming to the proposed specifications. Here, the control role is performed in the hand itself. If the sensors send average amplitude (PMES), this system will be a digital equivalent to modern two-site systems.

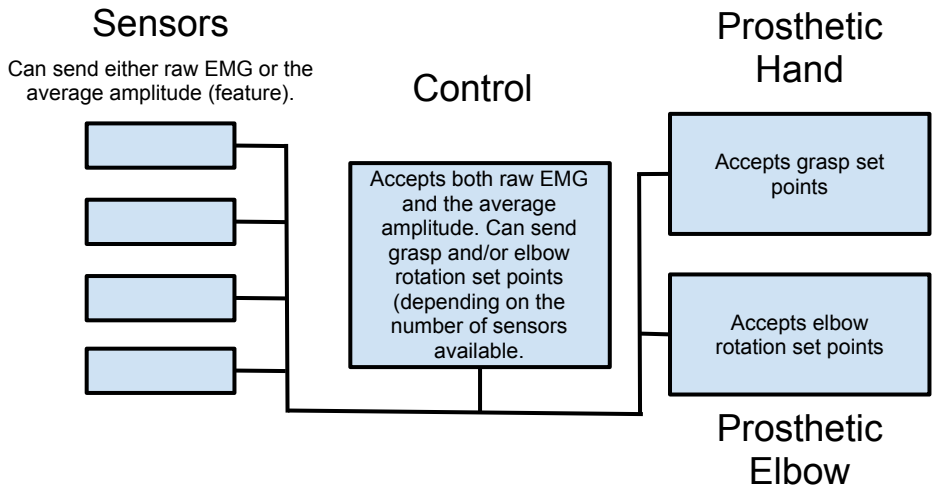

Figure 3.3: Example 2 of a prosthesis system conforming to the proposed specifications. This is a thought scenario with a control unit especially suited to the combined elbow-hand prosthesis.

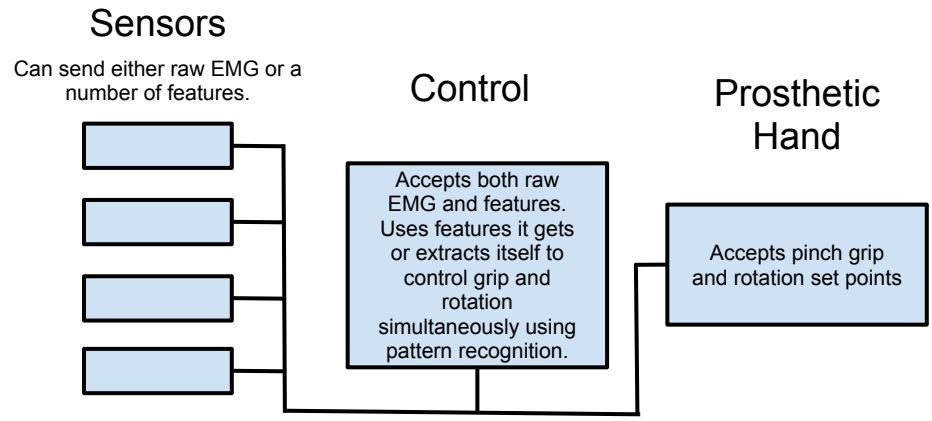

Figure 3.4: Example 3 of a prosthesis system conforming to the proposed specifications. Pattern recognition systems must be trained, so the advantage of having a separate control unit is that hands can be switched without needing to train a new control unit.

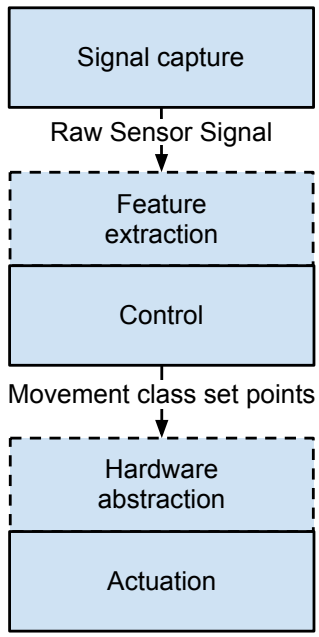

Figure 3.5: This is a simplified version of the division of roles I suggest.

3.1.1 A Note on Bandwidth

The CAN protocol has a relatively high bit rate (up to 1 Mbit/s for bus lengths below 25 m [12]). Idstein et al. [13] 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.

3.1.2 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.

3.2 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.

3.2.1 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.

3.2.2 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 2.10) 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 section 3.2.1, 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.

3.2.3 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 [14].

From *PDCP Info (2011 05 04)* [5]:

> *(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.

3.2.4 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.

3.3 Data Channels - Transmission

3.3.1 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. [13] 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.

3.3.2 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 section 3.3.1, 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.

3.4 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

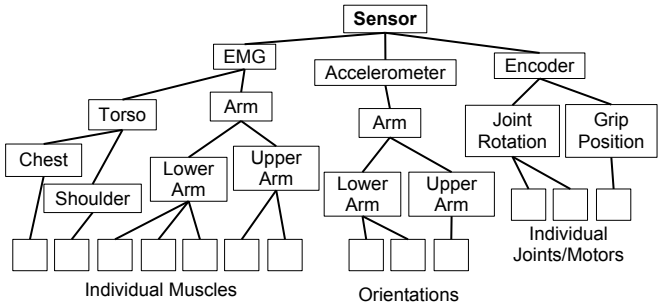

Figure 3.6: An example profile hierarchy for sensor channels.

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.

3.4.1 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.

3.4.2 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.

3.4.3 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 figs. 3.6 and 3.7.

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".

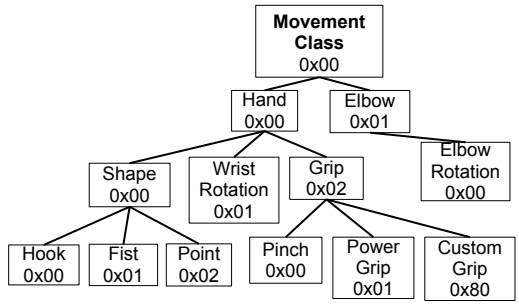

Figure 3.7: An example profile hierarchy for movement class set point channels, with enumeration.

E.g., if each node is assigned a number as in fig. 3.7, 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. 3.7.

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 section 3.2.4 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.

3.4.4 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].

3.4.5 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.

3.4.6 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. 3.7), 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.

3.5 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.

Chapter 4

Discussion

Chapter 3 already contains discussion of the details of implementation. In this chapter, I will look back on the bigger picture, and discuss some of the larger issues.

4.1 Restrict or Accommodate?

For problems like those described in section 3.5, it must be decided if they should be solved by accommodating the situation or restricting the protocol to avoid them. This also goes for e.g. deciding on a system architecture: How strictly should the roles be specified? Should a device be allowed to fill more than one main role?

For the case of multiple bus arbitrators, it could well be a good idea to restrict the role of bus arbitrator to be a separate device in the prosthesis socket. This would eliminate the problem of a bus arbitrator "hidden" in another device, and could also reduce the need for a memory node.

In the case of multiple control units, there will be a benefit from accommodating this. Otherwise, elbow-hand combinations will need specialized control units. It will also allow manufacturers to make specialized terminal devices, with specialized control units built-in, while also allowing users to own multiple, more affordable terminal devices, and control them with a control unit residing in the socket, increasing consistence of control.

These examples illustrate that there is probably no single right solution to use for all decisions.

4.2 Desired Behavior

The profile layer was designed to behave in a self-configuring manner. The behavior described in section 3.1.2 is at the heart of the proposed solution in chapter 3. It is worth discussing what behavior will be most useful for an actual user.

The behavior of my system has, roughly, three components: Standard message formats, self-configuration, and configuration memory. When I look back, I may have put too much emphasis on self-configuration, because, as I touch on in section 3.2.3, a system will need to be manually configured later anyway. So a good device profile layer could well be a standardization of message types, a configuration memory, and a well-designed manual configuration interface. Only very rarely, would you have the need to be able to use a part without having the possibility or time to manually configure it.

That is not to say that developing self-configuration is fruitless. It would be a useful (if not essential) addition to a profile layer, and the way to look at relations (here, the tree structure) can surely be put to good use, maybe in storing configurations, or displaying configurations in an intuitive way. It might also be more relevant as feedback sensors are added, and the system becomes more complex; self-configuration can allow the user to deal only with the most important devices when manually configuring.

I will again quote Stavdahl and Mathisen [1]:

> - *Reduced wiring and thus production cost and hardware failure rate.*
> - *Advanced coordinated control schemes with a large number of sensor signals and control variables.*
> - *Remote adjustment, fault diagnosis and software upgrades.*
> - *Interoperability and thus improved interchangeability of different devices.*

Interoperability and interchangeability comes from standardization of the value format in messages. Standardizing value formats is therefore very important. I value formats in sections 3.3.2 and 3.4.6, but the solution to be used in the end must be carefully considered.

Chapter 5

Conclusion

In this report, I have discussed the task of making a device profile layer for PDCP, and along the way, outlined one way to implement device profiles.

I have introduced a system architecture involving sensors, control units, and terminal devices. The devices will communicate using the data channel framework already part of PDCP. I have established that the configuration of the system will need to be retained across power-ons, and that this will most likely need to be taken care of by a device residing permanently in the prosthesis socket. I have also illustrated the need for a framework for manual configuration.

I have introduced the tree structure as a suitable format for profiles, and discussed the possibility of using profiles for channels, messages, and control strategies. I have also illustrated the use of tree-structure-based profiles for self-configuration.

Towards the end, I have discussed what sort of device profile layer would really be useful for real-world wearers of prostheses, and described what I, after having completed the project, see as the most essential parts of a device profile layer: Message format and configuration storage.

Chapter 6

Further Work

There is still a lot of work to be done on PDCP and an eventual profile layer. The following is a list of possibilities, considerations, and dilemmas that should be addressed in further work.

6.1 Profile Layer

With the ultimate goal of specifying a complete, "production ready" device profile layer, the following tasks are important.

6.1.1 Message Format

As noted in section 4.2, interoperability relies heavily on the standardization of message formats. The goal is for input channels to always understand the messages it receives. Further work should find a good way to accomplish this, and also investigate the best resolution and frame rate to use for prosthesis applications.

6.1.2 Configuration Interface

A good interface for manual configuration of the system would be immensely helpful, and is probably essential to make the profile layer useful.

6.1.3 Retaining Configuration

The need to retain configurations has been firmly established. Further work should decide on the best way to accomplish this, either through a

memory node, or other means. Data structures for keeping the configuration data will need to be developed.

6.1.4 Details

When the framework of a profile layer has been made, the details must be decided, such as the final shape of tree structures, parameter numbers for new parameters, which device profiles to include, etc.

6.2 Lower Levels of PDCP

6.2.1 Response Codes

As the devices do not know much about the network as a whole, the bus arbitrator may need to deny requests for different reasons. More diverse response codes can be helpful in these cases.

6.2.2 Information to Devices

It would be useful to have a framework for devices to acquire information about other devices on the bus. This could reduce the need for trial and error in requests, as described in the previous section.

6.2.3 Negotiation

There is a possibility of multiple bus arbitrators ending up on the same bus, as discussed in section 3.5. A protocol for negotiation between these would need to be developed, unless the bus arbitrator role is restricted in a way which guarantees that only one is present.

Negotiation between other devices such as control units would also be useful.

6.3 Hardware

6.3.1 Memory Node

A memory node has been established as a suitable way to retain concurrent information about the network. A physical implementation will need to be carried out.

6.3.2 Physical Interface

The manual configuration needs a physical interface; either through buttons and displays on the prosthesis, or through wired or wireless connection points. The possibilities are many, and wireless communication is on the rise.

Bibliography

[1] Øyvind Stavdahl and Geir Mathisen. "A Bus Protocol for Inter-component Communication in Advanced Upper-Limb Prostheses". In: *MEC '05 Intergrating Prosthetics and Medicine*. MyoElectric Controls/Powered Prosthetics Symposium. 2005.

[2] Anders Fougner et al. "Control of Upper Limb Prostheses: Terminology and Proportional Myoelectric Control — A Review". In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* (Sept. 2012).

[3] Ashok Muzumdar, ed. *Powered Upper Limb Prostheses*. Springer-Verlag, 2004.

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

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

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

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

[8] *USB in a Nutshell*. By Craig Peacock. 2010. URL: http://www.beyondlogic.org/usbnutshell/usb1.shtml (visited on 10/15/2012).

[9] *Approved Class Specification Documents*. Ed. by USB Implementers Forum. URL: http://www.usb.org/developers/devclass_docs (visited on 10/15/2012).

[10] *GATT Specifications*. Ed. by Bluetooth Special Interest Group. URL: http://developer.bluetooth.org/gatt/Pages/default.aspx (visited on 10/15/2012).

[11] *Specification of the Bluetooth System*. Ed. by Bluetooth Special Interest Group. Version 4.0. URL: http://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737 (visited on 10/15/2012).

[12] *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).

[13] 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 2011 MyoElectric Controls/Powered Prosthetics Symposium Fredericton, New Brunswick, Canada: August 14-19, 2011. 2011.

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