NTNU
Det skapende universitet

Implementasjon og testing av en åpen bussprotokoll for armproteser

Andreas Nordal

Master i teknisk kybernetikk
Innlevert: Juni 2012
Hovedveileder: Øyvind Stavdahl, ITK

Norges teknisk-naturvitenskapelige universitet
Institutt for teknisk kybernetikk

Prosthetics Device Communication Protocol for AVR – High-level Layer

by Andreas Nordal

June 25, 2012

0.1 Preface

This is my master thesis in Engineering Cybernetics at Norwegian University of Technology and Science (NTNU), Trondheim. I chose this subject because I am interested in real-time programming and free/open source software. I hope that my contribution to open source technology will change the world for the long term benefit of humanity, if only changing a small part of it.

I would like to thank my supervisor, Øyvind Stavdahl, for advice and enthusiasm that kept me going, and Andrzej Zamojski, who I cooperated with on this project, for his patience, especially in the final stressful days of testing.

0.2 Abstract

The prosthetics industry is shaped by incompatible products and manufacturers that develop their own standards. At the same time, an international research society (where NTNU is taking part) is in progress of developing an open bus protocol for use among sensors, actuators and other prosthetics components.

This thesis has looked at implementing this protocol on the AVR microcontroller architecture, with a goal to be compatible with a Canadian PIC based implementation. Results were not entirely successful, but the basic functionality is in place and has been shown to work.

0.3 Forord

Proteseindustrien preges av inkompatible produkter og produsenter som utvikler sine egne standarder. Samtidig er et internasjonalt forskningsmiljø (der NTNU er delaktig) i ferd med å utvikle en åpen bussprotokoll til bruk mellom sensorer, aktuatorer og andre protesekomponenter.

Denne oppgaven har sett på implementation av denne protokollen på AVR-prosessorarkitekturen, med et mål om kompatibilitet med en Canadisk PIC-basert implementasjon. Resultatene har ikke vært utelukkende gode, men grunnleggende funksjonalitet er på plass og er vist å fungere.

Contents

Chapter 1

Introduction

The prosthetics industry is characterized by incompatible products and manufacturers that develop their own standards. My supervisor, Øyvind Stavdahl at NTNU, and Yves Losier, at University of New Brunswick, Canada are at the forefront of research on an open bus protocol for use among sensors, actuators and other prosthetics components, the so called Prosthetics Device Communication Protocol (PDCP) protocol.

Their first draft of the protocol was available in 2009, and more revisions are underway as of 2012; it is intended that experience gained during this implementation work will be valuable for subsequent revisions of the protocol.

The idea for this project is to implement the PDCP protocol for the AVR microcontroller architecture, which is already in wide use in the prosthetics industry.

The project is a collaboration between Andrzej Zamojski, a Polish exchange student, and me. We chose to split the task horizontally, meaning that he did a low-level hardware abstraction layer, on which I implemented the protocol on top of. These layers have been termed the Hardware Abstraction Layer (HAL) and the High Level Layer (HLL) respectively.

Our source code will eventually be made available as free software. This, and the fact that a (proprietary) implementation also exists for the PIC microcontroller architecture, should hopefully make for a great incentive for the industry to adopt this open protocol that is PDCP.

Chapter 2

Theory

2.1 CAN protocol

Controller Area Network (CAN) is an industrial network protocol developed by Bosch in the 1980s, implementing the two lowest layers of the ISO OSI model, namely the *physical* layer and the *link* layer.[Catsoulis(2005)]

2.1.1 Arbitration

Nodes connect to the two wires of a CAN network in a wired-AND fashion. As figure 2.1 shows, this makes it physically impossible to send a high bit on the bus while a low bit is being sent by someone else.

This is the basis of the *arbitration* mechanism of the CAN network; the way colliding messages are dealt with: the sender of the high bit is obliged to give up sending the rest of the message, and may retry later. This contrasts Ethernet style collision detection (CSMA/CD), where both nodes need to back out and retry.

Node A sends	Node B sends	Bus value
0	0	0
0	1	0
1	0	0
1	1	1

Figure 2.1: Result of two nodes trying to send differing bits at once on a wired-AND network: Low bit wins.

2.1.2 Addressing

The transmitted payload data is prepended by, among other things, the *standard identifier field* of 11 bits [1]. [Losier(2012b)] The CAN standard specifies no explicit addressing of nodes, but the standard identifier field can be used cleverly for this purpose:

- Unicast — the standard identifier field can be used as destination address.
- Multicast — the standard identifier field can be used as source address.

Together with its following RTR (remote transmission request) bit, the standard identifier field makes up the *arbitration field*. Due to zero being the dominant bit, the arbitration field is also the priority of the message. All fields in CAN are transmitted in the order of more significant bits first, so numerically, the lower the arbitration value, the higher the priority.

2.1.3 Message filtering

Contemporary CAN controllers commonly implement hardware filtering of messages based on the standard identifier field. Although no standard specifies how, the way it's simply done™ is according to the equation of figure 2.2, where `filter` and `mask` are the variables to configure. [Natale(2008)] This is also true for the MCP2515 controller we have used. [mcp(2005)]

```
1 (arbitration_value & mask) == (filter & mask)
```

Figure 2.2: Boolean equation of the filter/mask mechanism, as expressed in C

[1] or 29 for the extended variant. Only the 11 bit variant will be considered in this report, since that is what the PDCP protocol is based on

Hardware filtering is a power saving feature, as it avoids interrupting the main microcontroller unnecessarily, possibly waking it from a deep sleep state. Indeed, battery longevity is a major concern in prosthetics. [Losier(2009)]

2.2 PDCP protocol

PDCP is a networking protocol based on CAN developed by the University of New Brunswick, Canada. [Losier(2012b)]

It further divides and defines the meaning of CAN message fields, specifically the standard identifier field and the 0–8 data bytes.

2.2.1 Division of the Standard Identifier Field

- Message Priority — 2 bits
- Message Mode — 1 bit
- Node Id — 8 bits

The PDCP protocol has similarities to the network layer in the OSI model; it defines a top layer node addressing, and the *arbitrator* node performs the same basic task as a DHCP server, in an operation called *binding*. Unlike the OSI model however, PDCP is not a layer of addressing on top of CAN, it just specifies one of many possible CAN addressing schemes.

2.2.2 Division of the Data Field

The first data (payload) byte is defined to be the *Function Code*. Other data bytes are function specific — their interpretation depends on the function code.

Chapter 3

Design

3.1 Overall design

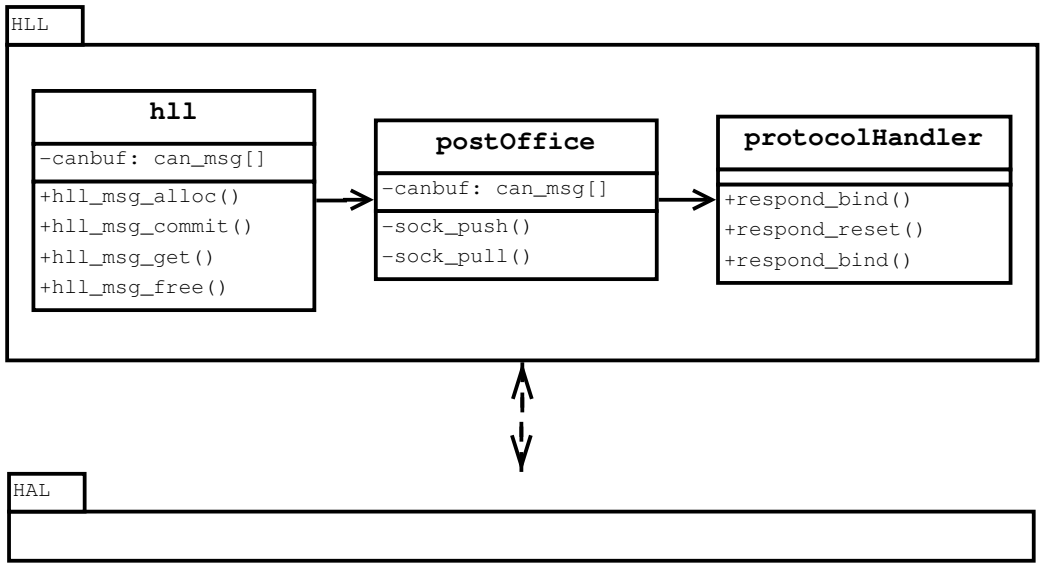

Figure 3.1: Class diagram of HLL. A possible source of confusion is the `hll` class inside the `HLL` package.

The idea is to divide the high level layer in two modules, `postOffice` and `protocolHandler`, with thin wrapper functions in `hll`, a class to encapsulate all functionality in the HLL package. While the first provides routing of CAN messages between other software modules, the second takes care of certain communication mandated by the PDCP protocol that is possible to automate, in order to relieve the application developer of this. The `postOffice` has no knowledge of the PDCP protocol, but hooks into `protocolHandler` for sorting CAN messages.

3.2 Buffering

FIFO buffered CAN transmission was found beneficial; the low level layer needs buffer space to read/write into, and this need to be handed asynchronously to the application. Three designs were considered. They are listed in figure 3.2 along with their advantages and disadvantages.

Concept	Advantages	Disadvantages
Chunked ring buffer	Stores variable length chunks compactly, exploiting that CAN messages know their own length. Requires no dynamic memory allocation.	The possibility of a chunk being wrapped around the ring's discontinuity point necessitates encapsulation of chunk memory access, involving extra copying (if a leaky interface is to be avoided).
Linked list	Avoid copying data by linking nodes into and out of the list.	Exposes the linked list container. Requires dynamic memory allocation.
Linked list of pooled containers	Abstracts away the linked list container. Memory pool(s) can be statically or dynamically allocated.	Must provide special allocators and deallocators.

Figure 3.2: 3 buffer designs, with advantages & disadvantages

Judging by the table of advantages and disadvantages in figure 3.2, the last

option seems the most flexible, and has the smallest disadvantages. [1] Thus, it is decided to go for «linked list of pooled containers». In figure 3.1, this functionality is planned to be implemented in the `postOffice` class.

The `postOffice` hands some messages to `protocolHandler`, the rest to the application, as shown in figure 3.3.

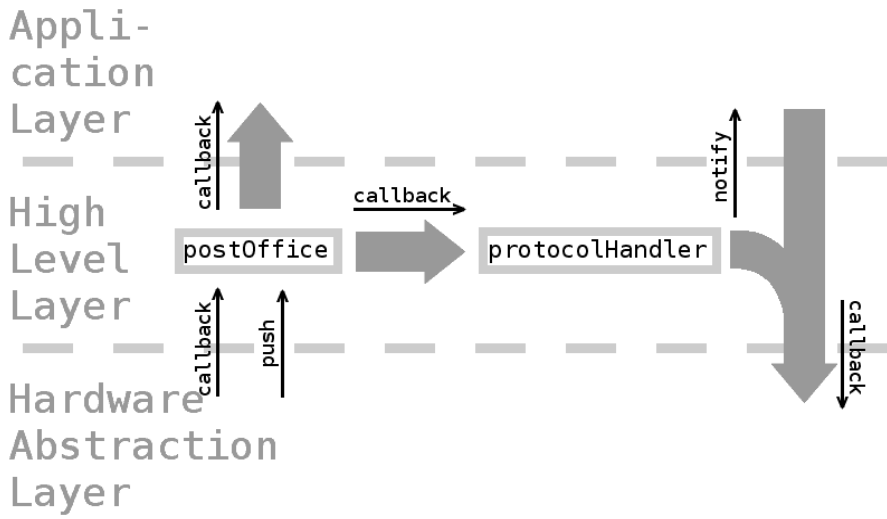

Figure 3.3: Flow of CAN messages between modules. Thick gray arrows are FIFO buffers, thin black arrows are function calls.

3.3 Middle layer interface

In order for Andrzej and I to split the task between us, we developed an interface between our parts; the high and the low layer, also known as HLL and HAL. We reached consensus on this interface before any implementation was done. The final interface was made in the form of an illustration by Andrzej and a description by me. These are reproduced below, in figure 3.4 and the following three pages.

[1] The last option was actually invented to overcome inconveniences of the first two.

The names of header files to declare this interface was left unspecified, but in retrospect, it should suffice to include `hll.h`, `functions.h` and `irqHandlers.h`.

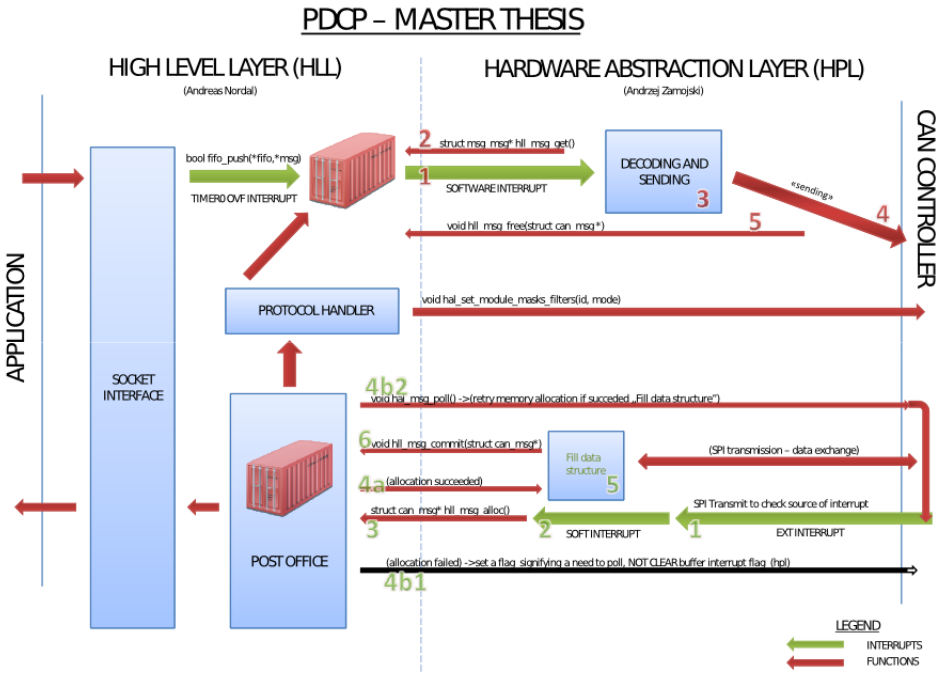

Figure 3.4: Illustration of the middle layer interface

PDCP for AVR – middle layer interface

OVERVIEW

This document describes an implementation of the *Prosthetic Device Communication Protocol* (PDCP) consisting of 2 layers:

1. **`hll`** high level layer
2. **`hal`** hardware abstraction layer

The purpose of this document is to detail the interface between these layers.

SYNOPSIS

```
struct can_msg;

void hal_set_filter(uint8_t filter);

void hal_set_mask(uint8_t mask);

void hal_msg_poll();

struct can_msg* hll_msg_alloc();

void hll_msg_commit(struct can_msg* msg);

struct can_msg* hll_msg_get();

void hll_msg_free(struct can_msg* msg);
```

DESCRIPTION

`hal_set_filter` shall set the filter configuration of the CAN controller to that specified in *filter*.

`hal_set_mask` shall set the mask configuration of the CAN controller. The bits in *filter* masked by *mask* must match the corresponding bits in the incoming message's Node Id for it to be received by the CAN controller.

hal_msg_poll shall retry fetching any incoming CAN messages left in the CAN controller by simulating an interrupt from the CAN controller.

hll_msg_alloc shall reserve unused memory in the high level layer for an incoming CAN message. This memory shall contain a **can_msg** structure, to which a pointer shall be returned to the caller. The caller must initialze the structure and send it back by calling **hll_msg_commit**.

hll_msg_commit shall pass the CAN message referenced by *msg* to the high level layer's input queue.

hll_msg_get shall look for an outgoing CAN message, and if found, return the contained can_msg structure. Otherwise, it shall return **NULL**. The caller must signify when it is finished with the pointed-to memory by calling **hll_msg_free**.

hll_msg_free shall mark the memory in use by the CAN message containing *msg* as unused.

FAILURE MODES

If no memory for an incoming CAN message is found, **hll_msg_alloc** shall return NULL.

If no outgoing CAN message is found, **hll_msg_get** shall return NULL.

These functions shall not fail:

- **hll_msg_commit**
- **hll_msg_free**

NOTE

- The hardware abstraction layer may leave CAN messages in the CAN controller in the event of failure of **hll_msg_alloc**. Therefore, the high level layer must call **hal_msg_poll** when it expects **hll_msg_alloc** to become possible.
- Incoming CAN messages may only be lost due to full receive buffers in the CAN controller. Assuming the high level layer calls **hal_msg_poll** whenever its internal «receive buffer» is no longer full, this situation only arises when the latter buffer is already full, and is still filled faster than emptied.
- Outgoing CAN messages are not lost. The high level

layer may be temporarily unable to handle outgoing CAN messages only when its internal «send buffer» is full.

- The high level layer's «send» and «receive» buffers may be implemented as a shared memory pool, with no space reserved for any direction. Thus, the capacity of one buffer may be zero due to the other buffer being full.

Chapter 4

Implementation

4.1 Socket module

The socket module provides basically what *channels* do in the Go programming language; [go(2012)] asynchronous message passing between software components that may be running concurrently. Interrupt handlers are the source of concurrency in our microcontroller.

There is, however, some specialization in the socket module and associated infrastructure that distinguishes it from a standard message passing channel:

- The datatype being transported is a `can_msg`. The content of the data does not matter, but it must be preceded by a linked list header, as found in the containing `canel` structure.
- The HLL module contains a few preconnected sockets accessible by *socket numbers*, akin to file descriptors in UNIX, thereby the name *socket*. [1] Two of these sockets, numbered as `SOCK_STDIN` and `SOCK_STDOUT`, are hooked into the physical CAN bus via HAL.
- A configuarble *receive hook* function is invoked whenever something is inserted into a socket, in the context of the insertor.
- To maintain FIFO behavior among interrupts that interrupt each other, the interrupting interrupt must not call the callback function while it is

[1] File descriptors are used for accessing sockets in UNIX.

already running in another context, and the interrupted interrupt must check for additional messages left by that interrupting interrupt, to run its callback in the context of itself. These checks are enabled by configuring `CONFIG_SERIALIZE_CALLBACKS` to nonzero. This would not be an issue if concurrency was in the form of threads.

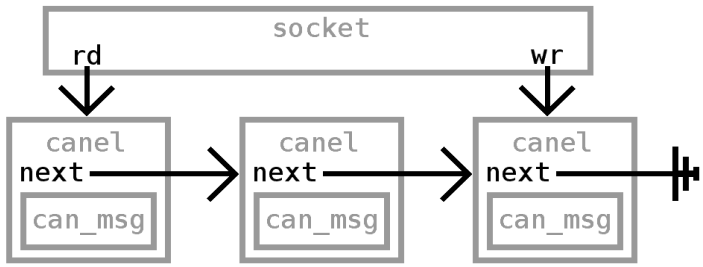

Figure 4.1: Each socket is a FIFO queue, implemented as a singly-linked list.

4.2 HLL module

The HLL module is what its name implies — an encapsulation of everything that is in the High Level Layer. It sets up socket communication channels between the hardware abstraction layer (HAL), the application layer (APP), and its own protocol housekeeper, *pdcphandler*.

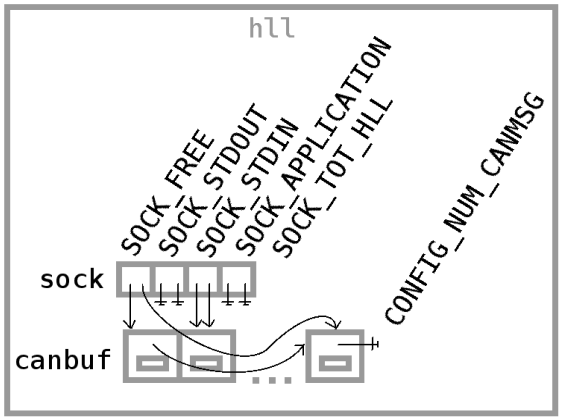

Figure 4.2: The «canel» elements are statically allocated in «canbuf».

4.3 Storing which nodes has bound to the arbitrator

A datastructure has been implemented to index node information by Node Id. Only the arbitrator has this datastructure. When binding, the arbitrator looks it up to see which keys are available. When binding is successful, a `nodeinfo` object is stored on this key, marking it as taken. For now, the `nodeinfo` object contains the identifiers sent from the newly bound device, which I, for the purpose of this discussion, will call «Device Personality». Storing the Device Personality enables later recognition of it — if for some reason a device asks for the same Node Id it already has, the arbitrator will detect this and respond by sending the same Node Id back.

A trie was initially chosen for this datastructure, for the reason that tries store leading common key subsequences compactly. As discussed in section 7.2.1, a sensible use of the Node Id bits would be as encoding of a decison tree that can group devices in several ways. As I think this is the smartest use, this is what I want to optimize for. Assuming bits are used this way, it can be expected that entropy is concentrated in certain bits, which makes a (surprise) decision tree suitable for storing the encoded decision tree. However, decision trees do not support insertion, due to not storing the whole key. A decision tree that stores the whole key in a predictable order is a trie. If assigning the bits in the order of higher entropy to the lesser significant bits, the longest common

subsequences would be at the front, and my dumb trie would do mostly the same as a smartly constructed decision tree.

My trie implementation has a configurable node size, and as it turns out, the address space is small enough that only one node suffices, making it effectively an array. Indeed, it seemed when debugging like large parts of this code was optimized away, as it was impossible to put a breakpoint in large parts of it.

If the reverse mapping was needed, from Device Personality to Node Id, the key would be much bigger, and a trie, or even a decision tree, would make more sense. This would be needed if Node Ids were to be pre-configured in the arbitrator.

4.4 Why there is no protocolHandler in the source code

There is! At least, its functions (see figure 3.1) exist. They were put in files named `hll_arbitrator.c` and `hll_device.c`, depending on whether they should be compiled exclusively for the arbitrator or device, respectively.

For the sake of the *don't repeat yourself* [rep(2012)] principle, the function that was supposed to be named `protocolHandler` was merged with `postOffice`. Consequently, the planned flow of CAN messages (figure 3.3) was simplified to that of figure 4.3.

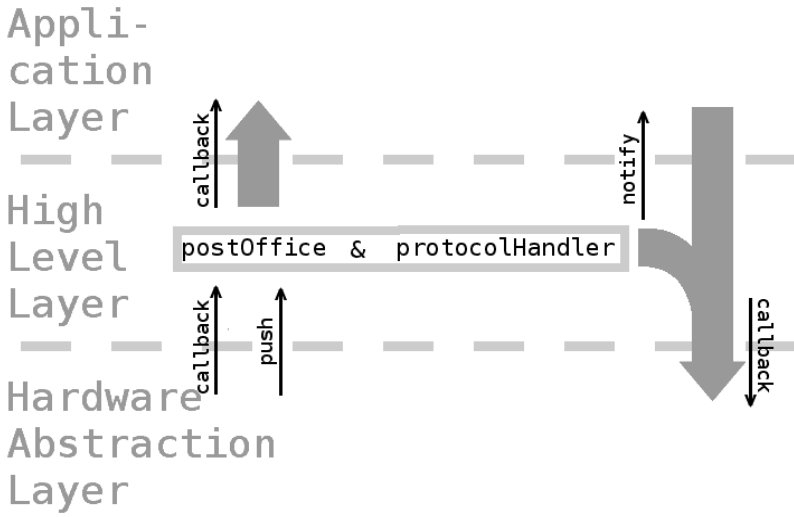

Figure 4.3: Flow of CAN messages between modules, as it finally became. Thick gray arrows are instances of my `socket` class, thin black arrows are function calls.

To see why the *don't repeat yourself* alarm was triggered by the planned design, and how it lead to this simplification, consider how the Function Code `switch` logic currently found in `postOffice` would need to be replicated in `protocolHandler` too:

```
/**
 * Handles some messages, sorts rest into SOCK_APPLICATION.
 */
void
post_office(struct socket *so){
    struct can_msg *msg = sock_pull(so);

    switch(msg->function_code){
        case REQUEST_BIND:
            respond_bind(msg);
            break;
        case RESPONS_SUSPEND:
        case REQUEST_BEACON:
        case RESPONS_RESET:
            //TODO
        default:
            hll_msg_push(msg, SOCK_APPLICATION);
            return;
    }
    hll_global.notify(msg->function_code);
}
```

Figure 4.4: Arbitrator's postOffice

```
/**
 * Handles some messages, sorts rest into SOCK_APPLICATION.
 */
void
post_office(struct socket *so){
    struct can_msg *msg = sock_pull(so);

    switch(msg->function_code){
        case RESPONS_BIND:
            respond_bind(msg);
            break;
        case REQUEST_RESET:
            respond_reset(msg);
            // should not return
        default:
            hll_msg_push(msg, SOCK_APPLICATION);
            return;
    }
    hll_global.notify(msg->function_code);
}
```

Figure 4.5: Device's postOffice

The `postOffice` would need to know which function codes are to be handled by the `protocolHandler`. These would need to be separated out and sent to the `protocolHandler` through a socket. Then, at the `protocolHandler`, *the same* function codes would need to be further sorted.

The design plan 3.1 specifies that «The `postOffice` has no knowledge of the PDCP protocol, but hooks into `protocolHandler` for sorting CAN messages.» However, separate modules or not, a method of sharing such ontology between functionally different pieces of code was not found. [2] It is easier and less error prone (especially with unfinished code) to have *one* `switch` statement do all sorting, as in figures 4.4 and 4.5.

4.5 Compilation switches

Constants that are tweakable, or enable optional features, are named with the prefix `CONFIG_` and placed in a separate file (`config.h`). This was decided early in the implementation phase, after the need first arose.

This is similar to the way the Linux kernel is configured, except its `.config` file is integrated into the build system instead of being directly included. In order to not depend on a particular build system, and specifically to compile our codebase with AVR Studio, our config file is included in the source files that need it.

Unfortunately, in order to compile both the arbitrator and device images, one has to manually edit the `CONFIG_BUS_MODE` setting in between. For this reason, this setting can be overridden, as done by my `Makefile`, which compiles both images in parallel. It is nevertheless voluntary to use the makefile build system.

[2] I believe this is what custom code generators and preprocessors are for; the `switch` in `postOffice` could be generated from the `switch` in `protocolHandler`.

Chapter 5

Testing

5.1 Unit testing

Besides the `src` (source) directory in our code repository, there is a `test` directory containing small programs that are made to exercise individual software modules. These are not meant for use on the AVR, but can be run on the build host. As a replacement for interrupts, and synchronization by disabling thereof, POSIX threads and mutexes have been used.

POSIX threads and mutexes are available natively in Linux 2.6 and newer, and with limited functionality as a library for Windows. Only Linux was used in this testing.

5.1.1 socktest.c

This is for testing the socket module (socket.c). The socket module is all about asynchronous message passing. It implements a minimal HLL interface and application layer. The latter runs in its own thread. The program reads from standard input into CAN messages, sends them asynchronously to the application thread, which prints them to standard output.

The application thread has a 1-second sleep in its loop, so to make it easy to test all the corner cases of FIFO buffering (full/empty) when run interactively. In no event shall its output differ from its input, in order for the test to pass. The makefile of the directory verifies this with random input.

Although the program exposes FIFO logic and concurrency, it does unfortunately not expose the need for serializing callbacks.

5.1.2 offsetof.c

This is a silly test of the `offsetof` macro, specifically testing it on relevant datastructures in the socket module.

5.2 Testing on proper hardware

The Nimron P1000 [nim(2012)] was developed for the course TTK4155 «Industrial and Embedded Computer Systems Design» at NTNU, [1] featuring:

- AVR at90usb1287 microcontroller
- MCP2515 CAN controller
- MCP2551 CAN transceiver

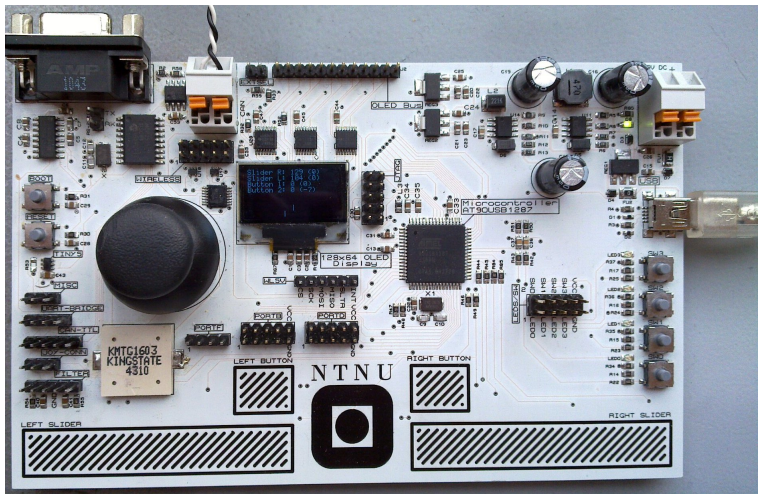

Figure 5.1: A Nimron P1000 running its default firmware. This was our primary development board.

[1] better known as «byggern»

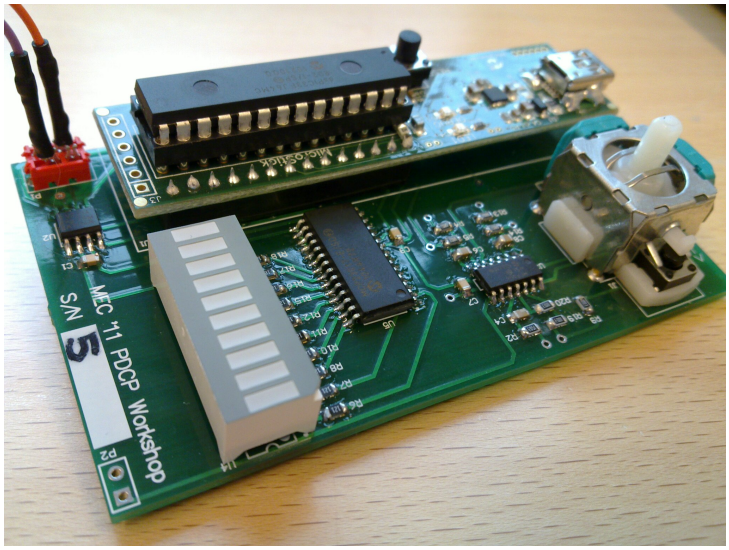

Figure 5.2: The Canadian implementation — a PIC microcontroller board. This was our black box test.

Chapter 6

Test results

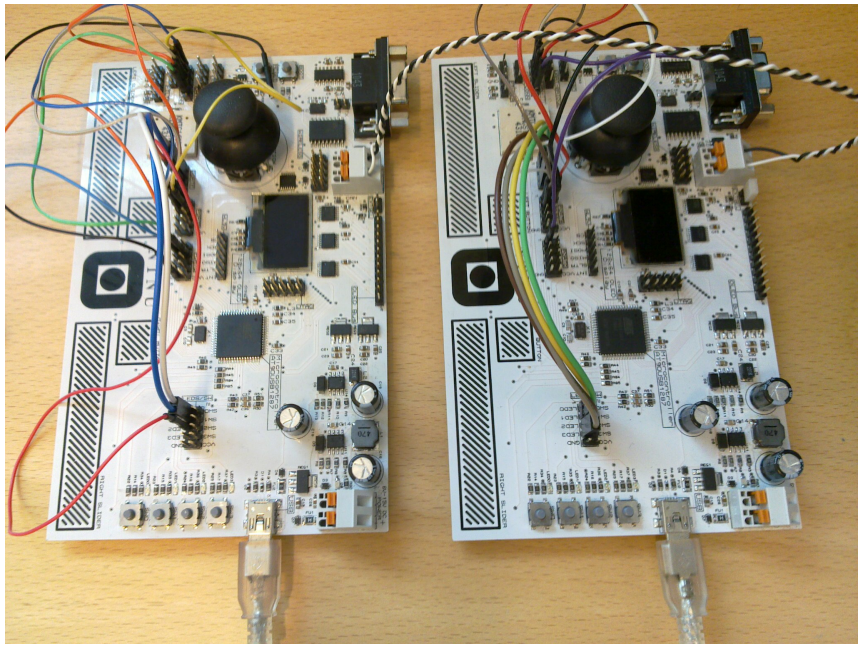

Figure 6.1: Test setup — two Nimrons in connection. The rat's nest wiring is for UART and LEDs.

In the main setup, two Nimron cards were connected, one acting as arbitrator, the other as device. It was also tested to replace the device with the Canadian equivalent.

6.0.1 Bind sequence

Binding was found to work:

```
PROGRAM START

PID = 1, VID = 2, SID = 3
Want Device ID ff
Arbitration field of sent msg: 1ff
NodeID of sent msg: 1

PID = 1, VID = 2, SID = 3
Want Device ID 1
Arbitration field of sent msg: 101
NodeID of sent msg: 1
Node number 1: PID = 1, VID = 2, SID = 3
```

Figure 6.2: Debug output of binding sequence captured by UART. The device first asks for the illegal value 0xff, but gets 1 instead. After asking anew with this value of 1, binding is complete. The last line was output after finding node number 1 in the trie datastructure, and its associated info was printed.

6.0.2 Reset sequence

The arbitrator was found able to send reset requests, which the device responded to, but no reset response was sent by the device. The reason was not pursued, due to lack of time. It is possible that the CAN controller did not have time to send it before it got reset.

6.0.3 Bind sequence with Canadian device

Our arbitrator unfortunately reset itself for unknown reasons after having filled the CAN message buffer. There was no indication of an unhandled interrupt.

[1] When our buffer had capacity of 2 messages, we received 2 bind requests before resetting, and when our buffer had capacity of 8 messages, we received 8 requests before resetting. This behavior does not necessarily mean that the buffer logic is wrong, as our buffer should never overflow anyway.

One difference between our device and the Canadian, is that the latter was sending several bind requests in quick succession, while we had previously only tested with sending one, and then wait for the response. The PDCP specification [Losier(2012b)] includes a flow chart of a device node during binding, and it has a timeout (which is not specified elsewhere). When reaching the timeout, the diagram specifies to resend the request. In relation to this diagram, our timeout was infinite, while the Canadian was short, seemingly too short for our arbitrator to handle. How long the timeout *should* be, is unspecified.

[1] We have implemented the `BADISR_vect` default interrupt handler, as suggested by [avr(2012)], to make it print «Default handler fired!» whenever an otherwise unhandled interrupt triggers. We know from previous debugging that this function works.

Chapter 7

Discussion

7.1 C versus C++

Some parts were found tempting to write in C++ rather than C. The `avr-gcc` toolchain has a C++ compiler, so this is possible. However, it was deemed that something as untraditional as C++ for microcontroller programming would reduces immediate ease of use and portability. Some possibilities were nevertheless explored.

7.1.1 Reduce scope of constants

Problem: Make the function code constants *belong* to their associated message structures.

Each *function* in the PDCP protocol has a function code constant and an associated message data structure. In order to match the spec more closely, C++ would provide a more fitting abstraction.

In C, constants can not be defined in structs [1].

[1] constants can be *declared* in structs, but that is useless for our purpose of defining its value

```c
#include <stdio.h>

typedef unsigned char u8;
typedef unsigned short u16;

struct request_bind{
    static const int FUNCTION_CODE = 0x01;
    u16 vendor_id;
    u16 product_id;
    u16 serial_number;
};

struct respons_bind{
    static const int FUNCTION_CODE = 0x81;
    u8  node_id;
    u16 vendor_id;
    u16 product_id;
    u16 serial_number;
};

#define printtype(T) \
printf(#T "::FUNCTION_CODE = 0x%02x, ", T::FUNCTION_CODE); \
printf("sizeof(" #T ") = %lu\n", sizeof(T));

int main(){
    printtype(request_bind)
    printtype(respons_bind)
    return 0;
}

```

Figure 7.1: Program demonstrating how function codes could have scope local to their own message structure. This syntactic sugar is valid C++, but not C.

```
request_bind::FUNCTION_CODE = 0x01, sizeof(request_bind) = 6
respons_bind::FUNCTION_CODE = 0x81, sizeof(respons_bind) = 8
```

Figure 7.2: Output of the above program. The constants are not contained in their message structures, as ruled out by the structures' sizes. Uncareful compilation (without the `-fpack-struct` gcc option) allowed the compiler to align the 2-byte fields on 2-byte boundaries here.

7.1.2 Parameterized datatypes

While *templated functions* are probably the most widely known use of C++ templates, the feature I find priceless, is *templated classes*. [2] Whenever there is an array in a structure, that when used for different purposes need different sizes, one better be programming in C++ rather than C. Examples from this project include `struct hll` and the trie datastructure. Luckily, these are not used more than one place, but suppose one wants to use the trie datastructure for two different things, e.g. not only mapping the Node Id to Device Personality, but also vice versa — two keys of different length… To overcome this in C, one has to manually calculate sizes and offsets in one's structures.

```
#include <stdio.h>

template <unsigned s>
struct class_with_buffer{
    static const unsigned size = s;
    char data[s];
};

#define printtype(T) \
printf(#T "::size = %u, ", T::size); \
printf("sizeof(" #T ") = %lu\n", sizeof(T));

#define STRINGIFY(EXP) #EXP
#define TOSTRING(EXP) STRINGIFY(EXP)
template <class T>
void accessor(T *obj){
    printf(TOSTRING(T) "::size = %u, ", T::size); \
    printf("sizeof(" TOSTRING(T) ") = %lu\n", sizeof(T));
}

int main(){
    printtype(class_with_buffer<0>);
    printtype(class_with_buffer<3>);
    class_with_buffer<0> a;
    class_with_buffer<3> b;
    accessor(&a);
    accessor(&b);
}

```

Figure 7.3: C++ program demonstrating a datatype with a parameter.

[2] Functions can always take an extra argument, which in presence of link time optimization and inlining, hopefully gets optimized away.

```
class_with_buffer<0>::size = 0, sizeof(class_with_buffer<0>) = 0
class_with_buffer<3>::size = 3, sizeof(class_with_buffer<3>) = 3
T::size = 0, sizeof(T) = 0
T::size = 3, sizeof(T) = 3
```

Figure 7.4: Output of the above program.

7.2 Protocol evaluation

7.2.1 Node Id

Due to the mask/filter method of filtering Node Ids (figure 2.2), there has to be a common bit pattern between Node Ids that a device is supposed to receive. Otherwise, this type of hardware filtering is not sufficient; the device will have to mask fewer bits, receive more messages, and additionally do software filtering (which we have not implemented).

Ideally, Node Ids should be assigned more like they were memory addresses of memory mapped devices, so that their bit sequences form encoded decision trees indexing the recipent. However, unlike memory mapped devices, there may be more than one recipent, and whenever that occurs, we need to severely waste address space to ensure that a unique bit pattern identifies the group of recipents. At its most extreme, the 8-bit Node Id would become the hot-one encoding of the numbers 0–7. With only one set bit per address, these 8 nodes could be set up to mask each other's addresses arbitrarily.

It is the arbitrator that ultimately decides the Node Id of devices, however as part of the Bind sequence, the device effectively suggests its own Node Id: It is inherent to the protocol that all messages originating from a device contains a Node Id, since this is baked into the standard identifier field of messages, and it is specified that the Bind sequence terminates iff this value equals the explicit Node Id field of the Bind response. This opens the question of who ``prefers'' Node Ids, if any. There are 3 alternatives:

1. Neither devices, nor the arbitrator has preferences for Node Ids in the system. The binding procedure is a race to determine Node Ids.

2. Devices have a preference of Node Id, communicated to the arbitrator upon Bind. The arbitrator approves if possible. If not possible, the arbitrator must inform of a value that is available, and the device is specified

to retry with this, eliminating further preferences it may have. However, such a permissive arbitrator *will* respect further preferences, if the device (by violation of the protocol) tries other values in order of preference.

3. The arbitrator fully determines Node Ids, without regard to devices' suggestions.

Only in case 2 can the suggested Node Ids be expected to be unique.

The arbitrator effectively has no Node Id, since the Node Id used in the standard identifier field is always that of the device (when the arbitrator is involved anyway). Furthermore, the Message Mode field only tells the node type of the *originating* node, not whether the message is *destined* for the arbitrator. Therefore, the arbitrator has no opportunity for masking away messages not meant for itself. Assuming that the majority of messages are *not* destined for the arbitrator during operation, this can be expected to be detrimental to power usage.

7.2.2 Byte order

The PDCP spec does not specify whether multibyte fields are big or little endian. Unlike many OSI network protocols, it is in fact little endian, as is evident from data captures provided by the protocol author. [Losier(2012a)] This suits our AVR architecture well.

7.2.3 Channels

The PDCP spec mentions *channels*, e.g. as content of the Node Identifier field from messages sent from device nodes, without explaining what they are.

7.3 Method evaluation

7.3.1 Separate offices → not so easy collaboration

Being a Norwegian student, I had an office at NTNU from before, while Andrzej, who is Polish, got another one. Although not far away, I knew from the start that this might be a bummer. [3] Although Andrzej and I had frequent meetings at the start and end of this project, that is, when we really needed to, I

[3] The situation resembled a student project at a summerjob I had, involving 3 students distributed on 2 offices. Lesson learnt: Collaboration is futile when on separate offices.

don't think we met as often as we should; my supervisor even offered to sponsor a pizza lunch — the fact that we missed this opportunity to me says that we failed despite good incentives.

7.3.2 Simultaneous development of layered software

Since my layer was supposed to run on top of Andrzej's layer, I could not realistically test my part before the lower layer was fairly functional (except doing unit tests). Fortunately, this did not take long, and most of my work turned out to be more or less based on things I had done before (and was confident about), so this was actually not a big problem in the end.

7.3.3 Flimsy AVR Studio → lack of testing

In the final stressful days of testing, my AVR Studio was incidentally broken. [4] This meant that, as in the early phase, I was also unable to debug my code in the final phase of testing, when connecting to the Canadian node was attempted. Consequently, only Andrzej was able to test our work at this stage. I was present, and I fixed problems in my code that we found together. Progress was unnecessarily slow because I would certainly have found a couple of those errors myself by regular integration testing. Fortunately, I had created a `Makefile` for compiling with `avr-gcc`, so at least I was able to make sure it compiled.

7.4 Implementation decisions

One implementation decision I had long forgot at the time of testing with the Candadian node, 6.0.3 that may explain the observed accumulation of incoming messages, might be the elimination of the intermediate socket buffer between `postOffice` and `protocolHandler`. The socket class is responsible for serializing callbacks, and without it, messages *will* stack up like that if the callback, in this case `respond_bind` takes a long time (it calls malloc, so probably does) and incoming messages come in too fast.

This does still not explain the crash, however. When the buffer is full, `sock_push` returns NULL. Something like a missing test for such a condition could in theory cause a reset.

[4] In a desperate attempt to debug why AVR Studio didn't want to communicate with my Nimron card, I uninstalled the wrong driver, after which AVR Studio was permanently unable to communicate with my Nimron card.

Chapter 8

Licensing

When releasing source code publicly, even though everyone can see it, copyright laws of many countries [1] still require people to ask for permission in order to use it.[Atwood(2007)] To explain in advance how people can reuse one's work, the standard practice is to apply a *license* to it. One part of my assignment was thus to suggest a suitable licensing of the source code.

8.1 Purposes of licensing

Attribution

It is very typical (to my knowledge, universal) of open source licenses to demand attribution notices in source code to be retained. Common additions to this rule include:

- contributors' names may not be misused to promote other products (as in 3-clause BSD)
- subsequent contributors must add their name to the list of contributors (as in GPL)
- attribution notices in program output must be retained (as in GPL)

Disclaimer

Once contributors have got their attribution, it is time to disclaim all responsibility. To be fair, I don't know of a country where *giving* permission

[1] In Norway, that would be Åndsverkloven.

to use one's copyrighted work makes one responsible, as if *selling* it, but who wants to take that risk… If not treated collectively with attribution, the disclaimer tends to follow closely the same rules for preservation as attribution. The disclaimer is often strikingly similar between licenses, even in wording. [2]

Copyleft

Copyleft is a form of copyright that permits derivative works to be made, provided that these are distributed under the same requirements as the original. In practice, the work will be perpetually tied to that exact license, except where licenses are designed or agreeed to be compatible. [3] However, compatibility goes mostly one way, from permissive to less permissive licenses. The scope of copyleft can apply to *changes* and *additions* to the original work, which a few licenses distinguish.[epl(2004)]

8.2 Applicability to own needs

If I understand the enthusiasm of my supervisor right, it is important for NTNU to harvest all possible prestige from this project. If NTNU, as well as Andrzej and me, wants our attribution notices to be respected, we should rule out releasing the project to *public domain*. Disclaiming responsibility is probably a good idea too, by analogy to how others have done open source licensing before. So far, a BSD-style license would fit well, without being more restrictive than needed. If we also want to protect our names from misuse, why not specifically take the 3-clause BSD license.[3bs(2012)]

However, the final criterion of copyleft will be decisive and difficult: First off, our creation is supposed to be suitable for inclusion in proprietary products of capitalistic companies in fierce competition. Given no incentive to cooperate, said companies can be expected to want to keep their own code secret. Some of it, I presume, very secret. Thus, a *strong copyleft* license like GPL would be totally unacceptible to them. [4] On the other hand, a too permissive license like BSD would not require them to share improvements to our creation.

[2] A particularly *legalese* wording, that is: redundant wording with Caps-Lock on.

[3] Even translating a license is nontrivial due a to risk of infringement no one wants to take.[gpl(2012)]

[4] GPL requires combined works containing GPL-licensed code to also be licensed under GPL terms, including other constituents. This positive feedback loop is what critics call *viral*.[vir(2012)]

8.3 Verdict

Of the not-so strong copyleft licenses:

- LGPL seems unfitting due to its requirement of runtime linking in order not to fall back to GPL mode.
- Eclipse Public License[epl(2004)] is supposedly «business friendly», but is still in the «permissive» category. [Atwood(2007)] Minus point for not being GPL compatible.
- Mozilla Public License[mpl(2012)] distinguishes changes from additions on a per-file basis. Changes to MPL-licensed files fall under the MPL, whereas additions of other source files do not. I see no problem with this.

If it was up to me, I would without doubt go for Mozilla Public License. But my decision is only suggestive.

Chapter 9

Conclusion

A framework around the PDCP protocol has been implemented for the AVR microcontroller architecture, and basic protocol features work between our own nodes. Binding with the Canadian node was not successful, but we have no reason to believe this isn't just a minor fault of our own. Specifically, an untested theory might explain the accumulation of messages observed, but this does still not explain the crash.

The Canadian protocol specification is somewhat lacking, but not to the point of being a compatibility problem. In some ways, it seems not entirety thought through, but this should not limit further refinement and development.

Glossary

Andrzej Andrzej Zamojski; creator of the low level part of PDCP for AVR. 12, 37, 38, 40

CAN Controller Area Network. 5, 7

Device Personality the combination of what the PDCP protocol refers to as Vendor Id and Product Id and Serial Number. 19, 20, 35

FIFO datastructure with the *first in, first out* property; queue. 11, 17, 25, 26

HAL Hardware Abstraction Layer. 3, 12, 17

HLL High Level Layer. 3, 12, 17

my supervisor Øyvind Stavdahl; associate professor at department of engineering cybernetics at NTNU. 38, 40

NTNU Norwegian University of Technology and Science. ii, 26, 40

PDCP Prosthetics Device Communication Protocol. 3, 7

UNIX a class of operating systems identified by commonality in file APIs among other things. 17

Bibliography

[epl(2004)] *Eclipse Public License*. 2004. URL http://www.eclipse.org/legal/epl-v10.html.

[mcp(2005)] *MCP2515 CAN controller datasheet*. 2005. URL http://ww1.microchip.com/downloads/en/devicedoc/21801d.pdf.

[3bs(2012)] *3-clause BSD license*. 2012. URL http://www.opensource.org/licenses/BSD-3-Clause.

[avr(2012)] *avrlibc*. 2012. URL http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html.

[go(2012)] *go*. 2012. URL http://golang.org/doc/effective_go.html#channels.

[gpl(2012)] *Unofficial translations of GPL*. 2012. URL http://www.gnu.org/licenses/translations.html.

[mpl(2012)] *Mozilla Public License*. 2012. URL http://www.mozilla.org/MPL/2.0/FAQ.html.

[nim(2012)] *Nimron*. 2012. URL http://www.nimron.no/P1000/.

[rep(2012)] *repeat*. 2012. URL http://en.wikipedia.org/wiki/Don't_repeat_yourself.

[vir(2012)] 2012. URL http://en.wikipedia.org/wiki/Viral_license.

[Atwood(2007)] Jeff Atwood. 2007. URL http://www.codinghorror.com/blog/2007/04/pick-a-license-any-license.html.

[Catsoulis(2005)] John Catsoulis. *Embedded Hardware (Second Edition)*. 2005.

[Losier(2009)] Yves Losier. *Moving Towards an Open Standard: The UNB Prosthetic Device Communication Protocol*. 2009.

[Losier(2012a)] Yves Losier. *AIF2 System Data Capture (Formatted)*. 2012a.

[Losier(2012b)] Yves Losier. *Prosthetic Device Communication Protocol for the AIF UNB Hand Project*. 2012b.

[Natale(2008)] Marco Di Natale. *Understanding and using the Controller Area Network*. 2008. URL www-inst.eecs.berkeley.edu/~ee249/fa08/Lectures/handout_canbus2.pdf.