CHALMERS

Design and Implementation of a Wireless CAN Proxy System

Master of Science Thesis in Communication Engineering

NIKLAS BERGGREN

Report # EX 084/2010

Department of Signals and Systems
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden, November 2010

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.

The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Design and Implementation of a Wireless CAN Proxy System

Niklas Berggren

© Niklas Berggren, November 2010.

Examiner: Erik Ström

Chalmers University of Technology

University of Gothenburg

Department of Signals and Systems

SE-412 96 Göteborg Sweden

Department of Signals and System

Göteborg, Sweden November 2010

Abstract

The purpose of this thesis project is to determine whether two Controller Area Networks (CAN) in a motor vehicle can be bridged over a wireless link, and if so to implement said bridge. The motivation for the project is to ease the work of automotive engineers and mechanics working with the programming of vehicle microcontrollers, removing the need for cabling when testing and diagnosing. The project involves both hardware and software design, resulting in a prototype board with an NXP LPC2364 microcontroller handling two CAN busses and a breakout board for a popular and widely available radio chip, nRF24L01+ by Nordic Semiconductor, over an SPI bus. Other technologies were considered such as bluetooth, however it was determined that technology of such complexity would likely be unnecessary, motivating the choice for a simpler chip.

A communication protocol was designed and implemented to facilitate communications between the nodes, allowing them to automatically balance the load depending on the incoming and outgoing data. A hard limit was found on the radio chip at 2500 messages of 32 bytes per second.

The resulting product can handle high load 500 kb/s buses and 1000 kb/s buses with medium to high loads with little to no error. The lack of available live testing facilities reduced possible testing, no proper results currently exist for live vehicle testing.

Preface

This document is the final report of a Master's thesis in Electrical Engineering at the department of Signals and Systems at Chalmers University of Technology in Göteborg, Sweden. The examiner has been Erik Ström at the department of Signals and Systems. The chalmers' tutor has been Mohammad Reza Gholami at the department of Signals and Systems. The thesis work was conducted at the request of Björn Bergholm at Broccoli Engineering AB, Gothenburg, where the majority of the thesis work was conducted.

Special thanks go to Björn Bergholm for providing the thesis opportunity itself and to Martin Persson for pointing me in the Broccoli direction. The facilities at ETA at Chalmers were invaluable for the assembly of the prototype boards with the surface mount tools available. Special mention also goes to Thomas Sporrong of IAR Systems AB, who kindly provided a J-link lite ARM probe for the project.

Contents

1 Background **7**

1.1 Purpose . . . 7

1.2 Delimitations . . . 8

1.3 Method . . . 8

1.4 Similar or related technologies . . . 8

2 System model **8**

2.1 Controller Area Networks . . . 9

2.1.1 Data frames . . . 9

2.1.2 Remote frames . . . 10

2.1.3 Error frames . . . 10

2.1.4 Overload frames . . . 11

2.1.5 Interframe spaces . . . 11

2.1.6 CAN 2.0B . . . 11

2.1.7 Error handling . . . 11

2.1.8 Arbitration . . . 12

2.2 GFSK . . . 13

2.3 Serial Peripheral Interface . . . 13

2.4 Consequences of bridging CAN over a wireless link . . . 14

2.5 Radio . . . 15

2.6 Physical connectors . . . 15

2.7 Proxying of multiple buses . . . 16

3 System hardware description **16**

3.1 Wireless communications module . . . 16

3.2 Microcontroller . . . 16

3.3 CAN Tranceiver . . . 17

3.4 CAN Wireless Proxy Prototype . . . 17

4 System software description **19**

4.1 Overview . . . 19

4.2 Peripheral initialization . . . 21

4.3 nRF24L01+ initialization . . . 21

4.4 Communication states . . . 22

4.4.1 Communication protocol 22
4.4.2 Negotiation 24
4.4.3 Listening mode 25
4.4.4 Transmitting mode 26

5 Results **27**
5.1 nRF24L01+ chip testing 27
5.2 Frame loss statistics 27

6 Discussion **29**

7 Appendix **31**
7.1 CAN Proxy Prototype gerber images 31
7.2 nRF24L01+ abstraction layer 33
7.3 Microcontroller software source code 33

List of Figures

1 Basic system mockup for the proposed solution to bridging CAN busses 7
2 CAN 1.2/2.0A Data frame with its seven bit fields 9
3 CAN 1.2/2.0A Data frame with its six bit fields 10
4 CAN Error frame with its two bit fields 10
5 CAN Overload frame with its two bit fields 11
6 Logical levels of multiple transmitting nodes 13
7 Basic operation of SPI with a master and a single slave 14
8 OBD2 connector as defined by SAEJ1962 15
9 Top copper layer of the CAN Proxy prototype board 17
10 Bottom copper layer of the CAN Proxy prototype board 18
11 Photograph of the prototype board with components mounted. . . 19
12 Software overview flowchart 20
13 Flowchart for the negotiation mode 24
14 Flowchart for the listening mode 25
15 Flowchart for the transmitting mode 26
16 Gerber file, top copper layer 31
17 Gerber file, top solder resist layer 31

18 Gerber file, top silk layer 32
19 Gerber file, top paste 32
20 Gerber file, bottom solder resist 33
21 Gerber file, bottom copper layer 33

List of Tables

1 OBD2 Pinout table 15
2 Test statistics for asymmetric load on the CAN bus 27
3 Test statistics for symmetric load on the CAN bus 28

List of Algorithms

1 C struct definition for a negotiation message 22
2 C struct definition for a handover message 22
3 C struct definition for a CAN frame 23
4 C struct definition for a CAN payload message 23

1 Background

A Controller Area Network (CAN) is a form of communication network developed by Bosch[1] and is used by various microcontrollers in larger products in order to facilitate communication with each other as well as with external equipment. Notably for the purposes of this thesis project, CAN networks are found in modern motor vehicles. By plugging into a CAN socket of a modern motor vehicle, a mechanic or engineer can read a wealth of information from the vehicle such as fuel consumption, temperatures and engine health.

Certain special testing applications require the vehicle to be running while connected to the diagnostic equipment, this is done for purposes such as evaluating the efficiency of new microcontroller programming. This form of testing is cumbersome, as it requires the addition of cables in the vehicle to connect to the mechanics' diagnostic equipment, which are both in the way and a potential safety problem while driving. The solution proposed by Björn Bergholm at Broccoli Engineering AB is to design a wireless proxy system for the On Board Device (OBD) connector, which can contain one or two CAN busses as well as signaling specified in ISO 9141 or SAE J1850.

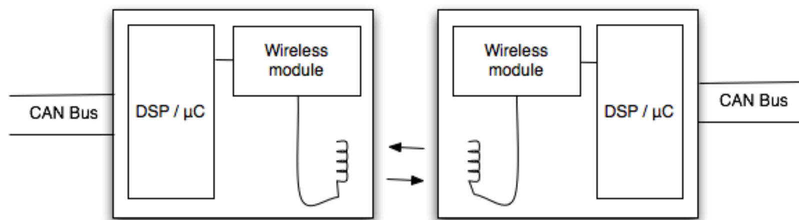

Figure 1: Basic system mockup for the proposed solution to bridging CAN busses

The system mockup in figure 1 shows two physically separated CAN busses connected via a wireless link, controlled by a DSP or microcontroller.

While most modern vehicles have a CAN bus present, certain manufacturers such as Volvo add a secondary CAN bus or an ISO 9141 bus for special purposes such as for exhaust-specific data. Because most car and truck manufacturers use different solutions, the wireless proxy can be extended through software to handle the various manufacturer-specific additions.

1.1 Purpose

The goal for this thesis project is to design and implement a wireless proxy between two physically separated CAN busses. The OBD connector typically found in a vehicle often holds more than a single CAN bus, a secondary CAN network can be present, as can an ISO 9141 and/or an SAE J1850 bus, it is therefore desirable to bridge these busses as well.

1.2 Delimitations

Because different vehicle manufacturers use different technologies in their vehicles it is difficult to design a proxy that can universally handle all forms of communication. The main focus for the project is therefor to transparantly connect two CAN networks, as is typically found in modern Volvo cars.

1.3 Method

The development project is divided into five parts. The first part constituted the bulk of the research required into the theory of CAN, microcontrollers and wireless communications methods. The second part was dedicated to prototype hardware development. Originally this part was intended to be short, with a microcontroller being mounted on a breakout board, however the requirements of clock crystals and decoupling capacitors being mounted close to the chip required the development of a prototype PCB. The third part was dedicated to the development of the software running on the microcontroller. The fourth part was the development of a complete PCB for the product, the time invested in this part was significantly shortened as the bulk of the work was completed in the second and third parts. The fifth and final part of the project was dedicated more thorough testing of the product and the testing of limitations.

1.4 Similar or related technologies

The idea of transmitting the frames on a CAN bus over a wireless link is not new, several products exist to transmit CAN data directly to a computer via WLAN or bluetooth. Kvaser is an example of a company with several products that can transmit CAN content over 802.11b/g networks directly to a computer[6], while ESD electronics produces similar products for linking CAN to a computer via bluetooth[7].

Researchers at the university of Singapore have proposed the use of DSRC / 802.11p WAVE, a WLAN technology developed specifically for vehicle communications, however the purpose of this technology is again not to bridge physically separated CAN networks, but to transmit data directly to monitoring devices. DSRC / 802.11p technology currently lacks widespread adoption among vehicle manufacturers as the IEEE specification has at the moment of writing just recently been released[8].

No products or technologies were found dedicated to bridging two physically separated CAN networks over a wireless link.

2 System model

The purpose of this section is to introduce the specifications of CAN as well as other technologies used in this project. Because CAN is likely to be an unheard of technology for many readers of this document, special attention is paid to introduce it.

2.1 Controller Area Networks

Controller Area Networks are used, as the name implies, to facilitate communications between microcontrollers in a larger device and is standardized as ISO 11898-1[2]. The technology was originally developed in the early 1980s for use in motor vehicles which is also the primary environment considered for this project.

CAN is a data link layer protocol and can be run at speeds from a minimum of 10 kb/s up to a maximum 1 Mb/s as of ISO 11898-2, which is the most common physical specification[2]. The bit rate is fixed on a given CAN bus and is thus never varied or negotiated between nodes[1]. Any node on the bus may start transmitting once the bus is deemed to be idle and uses an arbitration system in order to resolve conflicts of two or more nodes attempting to transmit simultaneously. Communication is serial over a single channel which utilizes the concept of "dominant" and "recessive" bits. A "dominant" bit is a logical zero while a "recessive" bit is a logical one. This is important for bus arbitration, documented in section 2.1.8, with the practical consequence that if two devices are transmitting simultaneously, any conflict is resolved at the first bit where the transmitters differ, as the unit attempting to transmit a recessive one will be overridden by a dominant zero.

CAN is non-addressed, meaning that individual nodes lack a specific identity to addressed as. Because of this everything on the bus is broadcast to all of the nodes on the network, what one node receives all nodes should receive. Another consequence of CAN being non-addressed is that nodes require no configuration, and can be added to the network at any time. Identifiers are used when requesting data. Rather than identifying individual nodes, identifiers identify what kind of data is sent or requested. Any node that holds the data targeted by the identifier may respond to a request. Any node that receives a frame acknowledges (ACKs) it. This is done by setting a specific bit slot in the received frame, detailed in sections 2.1.1 and 2.1.2.

CAN specifies four different frame types that can be transmitted on the bus; data frames, remote frames, error frames and overload frames.

2.1.1 Data frames

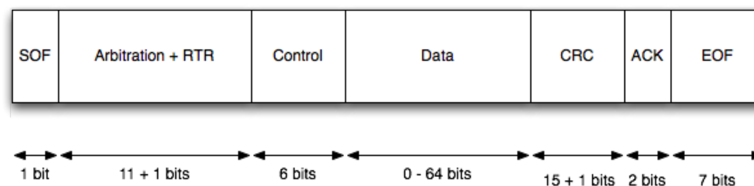

Figure 2: CAN 1.2/2.0A Data frame with its seven bit fields

CAN Data frames are used to transmit payloads between nodes and is structured as seen in figure 2[1]. A CAN 1.2/2.0A data frame consists of seven bit fields; SOF, arbitration, control, data, CRC, ACK and EOF. The start of frame field

is a single bit long and consists of a dominant bit. The arbitration field is used to determine which node gains control of the bus, the field is 12 bits long, consisting of an 11-bit identifier and a single dominant bit. The control field is 6 bits long, containing a 4-bit long data length code and two reserved dominant bits. The data field is 0-64 bits long and contains the payload of the frame. Data can only be sent in whole bytes, meaning the field is either 0, 8, 16, 24, 32, 40, 48, 56 or 64 bits long. The CRC field is 16 bits long, containing a 15-bit CRC and a single recessive delimiter. The ACK field is a two bit field, with an one bit ACK slot and a recessive delimiter. The ACK field is used by receiving nodes, where a dominant bit is set to acknowledge the reception of the frame. The end of frame field holds seven recessive bits to signal the end of the frame.

2.1.2 Remote frames

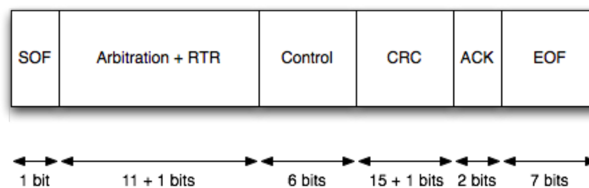

Figure 3: CAN 1.2/2.0A Data frame with its six bit fields

CAN Remote frames are used to request data from any nodes possessing the requested data and are structured as seen in figure 3[1]. A remote frame is identical to a data frame with the data field removed. Because there is no data field the control field can hold any data length code as it is ignored.

2.1.3 Error frames

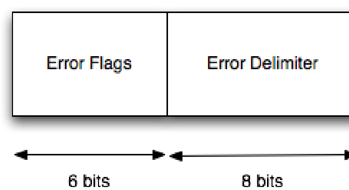

Figure 4: CAN Error frame with its two bit fields

Error frames are simple frames with two bit fields, as seen in figure 4. The error flags are either six consecutive recessive or dominant flags, depending on which state the transmitting node is in[1]. The delimiter field consists of eight consecutive recessive flags. An error frame is transmitted the moment an error is detected, interrupting any frame already being transmitted on the bus, forcing a retransmission[4].

2.1.4 Overload frames

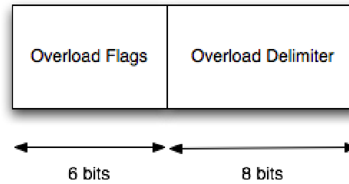

Figure 5: CAN Overload frame with its two bit fields

The purpose of the overload frame is to signal that the sending node is not yet ready to receive new frames[5]. The frame is structurally identical to the error frame, seen in figure 5, the distinction lies in when the frame is transmitted. While an error frame will interrupt another frame that is being transmitted, an overload frame is sent when the bus is free, thus no error counters will increase on the nodes. An overload frame is not preceeded by an interframe space[1]. Several overload frames can be transmitted in a series until the node in question is ready.

2.1.5 Interframe spaces

Data frames and remote frames are separated by a short bit field called interframe spaces[1]. An interframe space consists of three recessive bits, ending with a field called bus idle which is an arbitrarily long sequence of recessive bits. The result of this is that any data or remote frame is separated from subsequent frames by at least three recessive bit times until a new frame may be transmitted.

2.1.6 CAN 2.0B

CAN 2.0B is an extension of CAN 2.0A which is incompatible with CAN 1.2/2.0A. The main feature of the extension is an extended identifier field that is 32 bits long, of which 29 bits are identifier bits[1], with the remaining bits signaling the CAN 2.0B format. A CAN 2.0B controller is capable of handling CAN 1.2/2.0A frames, whereas a CAN 2.0A controller can either identify and ignore CAN 2.0B frames or fail to handle them.

2.1.7 Error handling

CAN is very error robust and detects errors through five primary methods; bit errors, stuff errors, CRC errors, form errors and acknowledgement errors[1].

A transmitting unit always monitors the level of the bus it transmits on. If a difference is detected between the bus level and the expected transmitted level from the transmitter, a bit error is triggered. An exception to this error is the ACK slot in a data or remote frame, as their purpose is to be overwritten by another node.

The CAN protocol implements bit stuffing for frames with long sequences of dominant or recessive bits, stuffing is injected every five bits of the same logical level. Should the rules of bit stuffing be broken on the bus, a bit stuffing error is triggered.

Should the received CRC value not match the locally calculated CRC of a frame a CRC error is triggered.

A form error is triggered when a fixed form field contains errant bits.

An acknowledgement error is triggered when a recessive bit is detected in the ACK slot of a data or remote frame. Every node receiving a proper data or remote frame should according to the CAN specification signal a dominant bit in the ACK slot, signalling that the frame was received without error. An acknowledment error can be triggered when a transmitting node is alone on the bus, as there are no nodes to acknowledge the frame.

CAN introduces the concept of "error active" and "error passive" devices[1]. The exact specification of error handling in CAN is deemed outside the scope of this project and therefore not fully described.

2.1.8 Arbitration

CAN implements a simple system for determining which node may transmit, should multiple devices attempt to transmit simultaneously[3]. Arbitration relies on the concept of dominant and recessive bits. Two nodes starting transmission simultaneously will both be transmitting the same logical levels on the bus until a conflict occurs. A device that transmits a dominant bit will monitor a dominant bit on the bus and continue transmitting. A device that transmits a recessive bit will monitor a dominant bit on the bus and at that point immediately stop transmitting until the bus is detected as idle again, at which point the node will attempt to retransmit the frame.

Because the first field of a data or remote frame is the identifier field, the node that is allowed to transmit is the one transmitting a frame with the highest amount of leading dominant bits. The result is that the identifier field functions as a priority on the bus, with lower priority number being of the highest importance as a logical zero is dominant on the bus as described in section 2.1.

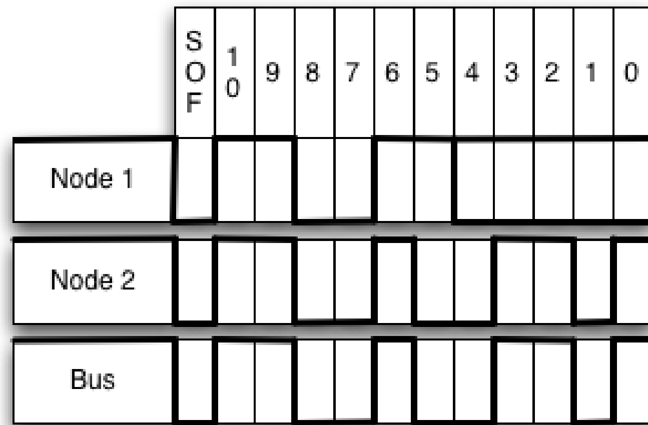

Figure 6: Logical levels of multiple transmitting nodes

Figure 6 shows an example of two nodes transmitting a frame simultaneously. Both nodes signal the start of frame, followed by the identifier. A conflict occurs on bit 5, where node 1 transmits a recessive bit while node 2 transmits a dominant bit. The conflict is resolved by node 1 backing of, allowing node 2 to continue the full transmission of its frame.

2.2 GFSK

GFSK is a form of Continuous Frequency Shift Keying (CFSK) modulation where a gaussian filter is introduced at the pulse train, shaping the pulses[11]. The reason for doing this is to reduce any power in the side bands, resulting in increased spectral efficiency and reduced disturbance on neighboring frequency bands at the cost of increased receiver complexity. GFSK is employed in techonologies such as Bluetooth and DECT telephone technology[12], while the related technology Gaussian Minimum Shift Keying (GMSK) is employed in GSM[11].

2.3 Serial Peripheral Interface

The Serial Peripheral Interface (SPI) or 4-wire interface was developed by Motorola in order to facilitate communication between microcontrollers and other chips. SPI operates in a master-slave configuration, where the master unit generates a clock signal (SCK) used by both master and slave to communicate synchronously. SPI operates with four signals; Master-In Slave-Out (MISO), Master-Out Slave-In (MOSI), Chip Select (CS) and Serial Clock (SCK). The MISO signal is the input to the master and is generated by the slave, however the master chip still indirectly controls this signal as it controls the clock. The MOSI signal is the input of the slave and is generated by the master. The Chip Select signal is used by the master to signal a specific chip that it should send

and/or receive data, which allows multiple chips to be connected to the same MOSI/MISO lines[13].

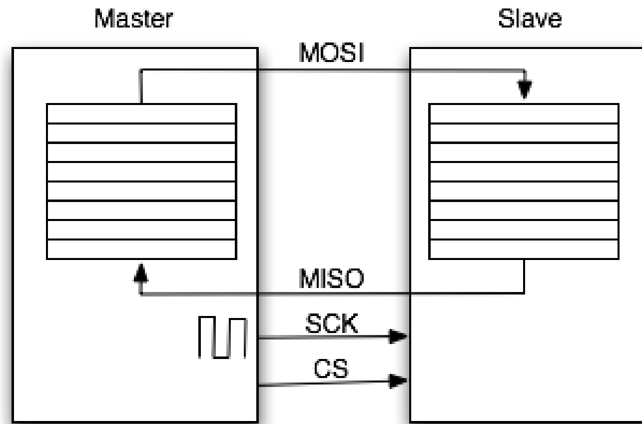

Figure 7: Basic operation of SPI with a master and a single slave

Data transmission between two chips is done through two shift registers, as depicted in figure 7. Each chip writes a byte to their respective registers and transfer one bit at a time synchronized with the SCK signal. On a clock pulse, the end of both registers have their bits sent to the register on the other end, the circuit as a whole forming a long shift register with the last position connected to the first. Due to this arrangement, transmission and reception of data can be done simultaneously. In the case that a slave has new data to transmit to the master, it will have to signal the master by different means as it cannot drive the clock or chip select pins. Additional pins are commonly added to slaves wishing to initiate a transfer, signaling an interrupt which prompts the master to active the SPI interface.

2.4 Consequences of bridging CAN over a wireless link

CAN has multiple timing requirements that result in potential problems for a wireless replacement of the bus. Whenever data is transmitted wirelessly, a delay is introduced that is significant compared to the wired equivalent which is specified at a maximum of 5 ns/m at a maximum of 40 meters for high speed CAN[2]. The result of this specification is that the absolute maximum delay between two nodes on a CAN network is 200 ns. Timing is important in CAN as receiving nodes of a data or remote frame are expected to acknowledge frames by setting the ACK slot in the frames at the appropriate time.

At data rates of 1 Mb/s the bit time is 1 μs, which is difficult to accurately time with a wireless solution as a frame has to be sent over the wireless link to the receiving nodes and the ACK flag has to be sent in response back over the link. The wireless proxy will therefore have to act intelligent, flagging ACKs and handling error frames locally. A consequence of this is that bad data one

side of the CAN proxy will not be transmitted over the wireless link, burdening the second CAN.

2.5 Radio

The primary purpose of this project is to bridge two wired networks over a wireless link inside of a moving vehicle. With this basic premise, parameters such as range and carrier frequency become less important as the maximum physical distance between the units is unlikely to be more than a few meters. As for carrier frequency, the Industrial, Scientific and Medical (ISM) band of 2.4-2.5 GHz is useable as it is internationally permitted and the band is unlikely to be occupied by more than one or two devices such as WLAN or bluetooth at a time in a moving vehicle.

2.6 Physical connectors

The OBD2-connector is defined by SAE J1962 [9] and is present in the majority of modern (post 1996) cars, in some cases by law.

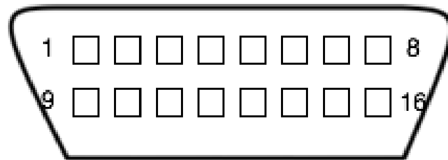

Figure 8: OBD2 connector as defined by SAEJ1962

Table 1: OBD2 Pinout table

Pin	Description
1	-
2	Bus positive line of SAE J1850 PWM and SAE 1850 VPW
3	-
4	Chassis ground
5	Signal ground
6	CAN high
7	K-line of ISO 9141-2 and ISO 14230-4
8	-
9	-
10	Bus negative line of SAE J1850 PWM and SAE 1850 VPW
11	-
12	-
13	-
14	CAN low
15	L line of ISO 9141-2 and ISO 14230-4
16	Battery voltage

Figure 8 shows the OBD2 connector, with table 1[10] detailing which signal is found on which pin. Certain manufacturers extend the connector with a secondary CAN bus, found on pins 3 and 11. Other manufacturer-specific additions may exist on the free pins.

2.7 Proxying of multiple buses

As multiple buses can be transmitted over the wireless link, a wrapper protocol is required to identify which bus a specific frame belongs to. The wrapper contains the ID of the CAN controller as well as an accompanying CAN frame so that the receiver will place the incoming frame on the correct outgoing bus. The protocol is documented in detail in section 4.4.1.

3 System hardware description

This section will detail the hardware used in the implementation of the wireless CAN proxy.

3.1 Wireless communications module

The Nordic Semiconductor nRF24L01+ chip was selected for the wireless communication based on its support for data rates up to 2 Mb/s in the 2.4-2.5 GHz band, transmitted to a microcontroller over an SPI interface. Concidered alternatives were bluetooth based chips with Enhanced Data Rate support. While a bluetooth based solution could be modified to interface directly with a computer, the added costs, complexity and PCB space requirements were difficult to justify.

The nRF24L01+ implements the GFSK modulator and demodulator, described in section 2.2, filters and amplification elements required for wireless data transmission and reception. The chip also implements the technology required to transmit data to a microcontroller over an SPI interface. Required peripherals to the chip is an antenna for the 2.4-2.5 GHz band and an impedance matching network. A clock crystal of 16 MHz is also required for the internal controller as well as oscillators.

3.2 Microcontroller

The requirements on the microcontroller are the presence of at least two CAN controllers as well as an SPI interface to communicate with the nRF24L01+ chip. While CAN controllers are common on most modern microcontrollers, having two controllers on the same chip is more rare, limiting the selection.

The first choice of microcontroller was the Atmel AT91SAM7A3, which satisfies the above mentioned requirements and operates at 60 MHz, and was chosen primarily due to previous programming experience with Atmel products. Because the AT91SAM7A3 is an aging design, availability in Sweden is low, with

a preliminary delivery date in January 2011 from one of the largest electronics suppliers in the country. Waiting six months for a microcontroller was, of course, unacceptable, so the first prototype PCB and much of the associated work had to be thrown out.

The second choice of microcontroller was the NXP LPC2364, which provides a similar set of features as the Atmel AT91SAM7A3. The chip can operate at frequencies up to 72 MHz and placed fewer requirements on the PCB design as fewer decoupling capacitors for the power lines were required as well as the lack of an external PLL filter.

3.3 CAN Tranceiver

While the microcontroller integrates two CAN controllers, two tranceivers are required to convert signals to the appropriate voltage levels and to implement the logic of dominant and recessive bits. Two Microchip MCP2551 tranceivers were purchased to act as the bridges between CAN busses and microcontroller.

3.4 CAN Wireless Proxy Prototype

For a prototype board the nRF24L01+ chipset with peripherals were not directly integrated. Instead, breakout boards were purchased in order to speed up the development process. The reason for this is due to the nRF24L01+ chip being sold in a QFN20 package, which is difficult to solder by hand.

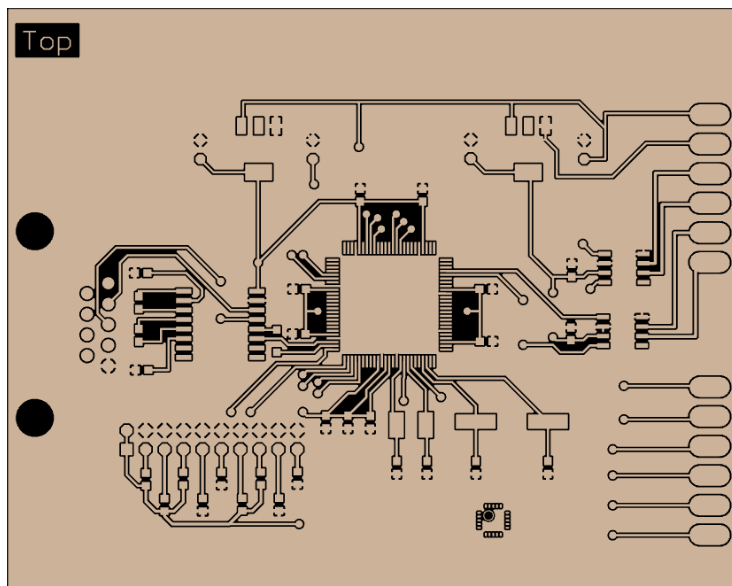

Figure 9: Top copper layer of the CAN Proxy prototype board

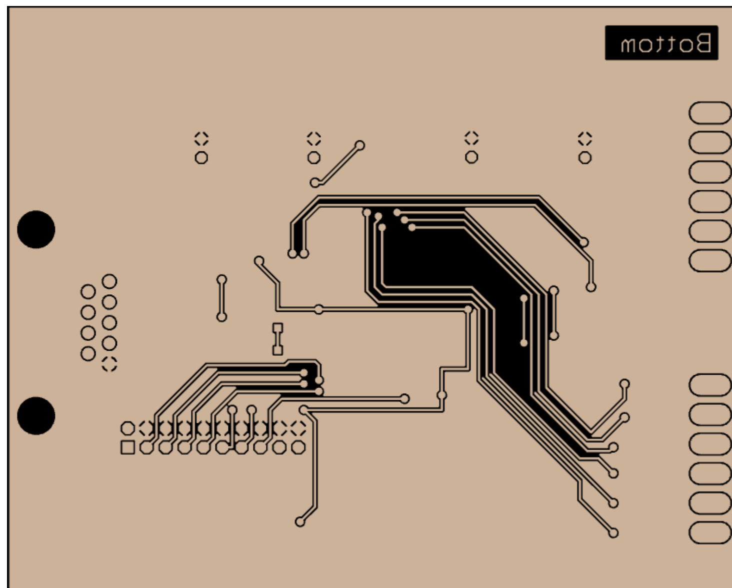

Figure 10: Bottom copper layer of the CAN Proxy prototype board

Figures 9 and 10 show the top and bottom copper layers of the prototype PCBs that were ordered to serve as a platform for software development.

On the top layer the microcontroller is placed in the middle of the board. The left part of the board is populated with a JTAG interface for a programmer/debugger as well as a MAX3232 chip for an RS232 serial interface to a computer. The MAX3232 chip as well as the 9-pin D-sub contact are not required for normal use, but were included for debugging purposes. The top side holds two voltage regulators to transform a car battery voltage of 14V to 3.3V and 5V for the microcontroller and CAN tranceivers respectively. The right hand side of the board contains the CAN tranceivers as well as two 6-pin connectors. The top connector holds pads for the required pins from the OBD2-connector; car battery voltage, signal ground and the four wires for two CAN busses. The bottom connector holds pads for the SPI signals going to the nRF24L01+ breakout board, as well as an interrupt pin. The bottom side of the board holds two clock crystals for the microcontroller and is prepared to integrate the nRF24L01+ chip, matching network, clock crystal and antenna for a final product. Surrounding the entire top copper layer is a ground plane.

The bottom copper layer of the board holds no components and is used to route certain signals and power. The JTAG connector and SPI signals are partially routed through the bottom copper layer, as is the 3.3V supply voltage to the microcontroller.

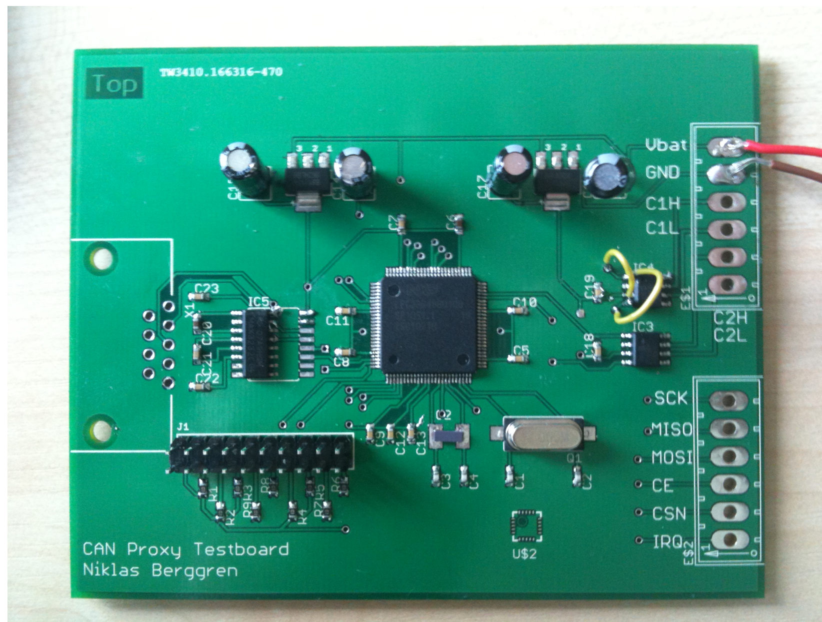

Figure 11: Photograph of the prototype board with components mounted.

Figure 11 shows a photograph of the first prototype board with components mounted. Due to a mistake identified after the gerber files were sent for manufacture, the secondary CAN tranceiver had its transmission and reception pins swapped. This was rectified by soldering the yellow air wires to their correct respective signal and not to the pads designated for the IC. The top right connector is meant to be replaced by an OBD2-connector which holds battery voltage and the CAN bus in the final board. A secondary mistake later identified was the omission of a terminating resistance over the CAN busses, which was rectified by soldering a pair of 120Ω resistors over the high and low lines.

Complete gerber images of the prototype board, including silk and solder mask layers as well as drill specifications, can be found in the appendix.

4 System software description

This section will describe the various parts of the software running on the microcontroller, forming a complete system. The source code for the microcontroller software is included in the appendix of this report.

4.1 Overview

Before software development began in ernest, the software was planned out in the form of a flowchart. The decision was made to prioritize messages coming in from the wireless link over the CAN messages in order to reduce dropped packages that had been processed on the other end.

Figure 12: Software overview flowchart

The flowchart as seen in figure 12 depicts the basic structure of the software running on the microcontroller. When starting up, the software initializes the on-board peripherals of the microcontroller used for the project; PLL, GPIO directions, SPI module, CAN controller and UART. Once the SPI module is initialized it is possible to send the initial commands to the nRF24L01+ chip setting it in receive active mode, monitoring for incoming packets.

Once the system initialization steps are complete the program enters its main loop attempting to negotiate with another nearby node to start communication,

the negotiation phase is documented in section 4.4.2. Once negotiation has finished the nodes enter either listening or transmitting mode, documented in sections 4.4.3 and 4.4.4 respectively, in order to exchange CAN payloads. Should any process ever fail or terminate abnormally the node reverts to negotiation mode.

The software is written nearly entirely in the C programming language, the exception is a certain initialization step for initializing the low-level startup PLL, which is the first thing done on startup. This can easily be transfered into the C program if desired.

4.2 Peripheral initialization

The microcontroller peripherals to be initialized are, in order, PLL, powerup of the CAN controllers, pin mode selection, pin direction setting and finally peripheral clock configuration. Once these base components are set, clocks and initial settings are made for the SPI and CAN interfaces.

The PLL drives the CPU core clock from the 12 MHz crystal on the PCB up to 72 MHz. The increased performance of the microcontroller up to 72 MHz from the microcontrollers' internal 4 MHz oscillator is the reason why it is initialized first. On microcontroller reset the CAN controllers and acceptance filter are unpowered, while peripherals such as UART, timers and SPI are powered by default. The second step of initialization is therefore to power up the CAN peripherals. The pin selection connects the programmable I/O pins to their intended peripheral, such as SPI or CAN. The direction setting sets the programmable I/O pins for Chip Select Not (CSN) and Chip Enable (CE) as outputs, which are used as control signals beside SPI for the nRF24L01+ chip. Finally, the peripheral clocks are set to be equal to the CPU core speed, the reason for this being that calculation of individual clock peripherals become slightly less complex as the default peripheral clock is the CPU core speed divided by 4.

Initiation of SPI is accomplished by setting the clock divider to 12, producing an SPI clock of $\frac{72}{12} = 6$ MHz, which is 2 MHz from the maximum clock the nRF24L01+ is capable of. Once the clock is set, the microcontroller is set as the master of the SPI bus.

The initiation of the CAN controller sets the controller in reset mode, which allows the control registers to be written to. Once the controller enters reset mode, the bus timing register is set to sample the CAN bus at 83% of the bit time at the bit rate of the CAN bus the device is connected to. Current supported modes are 1 Mb/s, 500 kb/s, 250 kb/s and 125 kb/s.

4.3 nRF24L01+ initialization

The nRF24L01+ chip is controlled through the SPI interface and in the program code through an abstraction layer. The initialization sets the chip in active listening mode and activates the ShockBurst functionality, meaning that the chip will automatically acknowledge any received messages, offloading the microcontroller.

4.4 Communication states

The software can enter three distinct states or processes, as documented in section 4.1, namely negotiation, listening and transmitting. In order to communicate efficiently a rudimentary communication protocol was developed, consisting of three basic message types. It is important to note that incoming CAN frames on the local buses are always buffered to memory, no matter which mode the node is in.

4.4.1 Communication protocol

The communication protocol is based on three types of messages; the negotiation message, the CAN payload message and the handover message. Due to the lack of full duplex communications over the radio link, the nodes alternate between transmitting and listening modes. If a node is in transmitting mode with no CAN messages in the buffer to transmit, a handover will be sent in order for the two respective nodes to simultaneously switch modes. For both listening and transmitting modes, timers are used to ensure that no node is stuck in either mode. The length of this timer is configured as a compromise between latency and frame loss. With the proxy mainly being intended to connect diagnostic equipment to motor vehicles, the transmission to receive ratio is likely to be highly skewed as the vehicle bus will constantly be occupying the bus while the diagnostic equipment transmit very little into the bus. The use of handover messages ensure that both nodes are able to transfer information two-ways, and also automatically balance the listen to transmit times to reflect the bus load.

Algorithm 1 C struct definition for a negotiation message

```
typedef struct w_negotiation_message {
  unsigned char pkgtype;          // Message type flag, 'n'
  unsigned char nodeID;           // ID of the local node
} w_negotiation_message;
```

The implementation of the negotiation message is shown as Algorithm 1. The information transmitted is a message type indicator present in all C structs and the node ID of the transmitter. This message type is exclusively transmitted in the negotiation phase, however it is always looked after in order to assertain whether the partner node has reverted into negotiation mode.

Algorithm 2 C struct definition for a handover message

```
typedef struct w_handover_message {
  unsigned char pkgtype;          // Message type flag, 'h'
  unsigned char nodeID;           // ID of the local node
} w_handover_message;
```

The handover message, defined in C as Algorithm 2, is nearly identical to the negotiation frame, the difference lies in the message type flag. The handover

message is transmitted when a node in transmitting mode has either run out of its alotted time or the local CAN message buffer is empty. The message is used as a flag that a transmitting node has ended properly and will be entering listening mode as soon as the handover message is transmitted. A listening node receiving a handover message will transition to transmitting mode as soon as the handover message was received.

Algorithm 3 C struct definition for a CAN frame

```
typedef struct can_message {
  unsigned int ID : 29;           // 29 bit ID field
  unsigned int X : 1;             // Extended format flag
  unsigned int DR : 1;            // Data/Remote flag
  unsigned int DLC : 4;           // Data Length Code field
  unsigned int data1 : 32;        // Data field one, 4 bytes
  unsigned int data2 : 32;        // Data field two, 4 bytes
} __attribute__ ((packed)) can_message;
```

Algorithm 4 C struct definition for a CAN payload message

```
typedef struct w_can_payload_message {
  unsigned char pkgtype;          // Message type flag, 'p'
  unsigned char nodeID;           // ID of the local node
  unsigned char ctrl1 : 2;        // Controller ID of msg 1
  unsigned char ctrl2 : 2;        // Controller ID of msg 2
  can_message msg1;               // The actual CAN messages
  can_message msg2;
} __attribute__ ((packed)) w_can_payload_message;
```

The CAN payload message is implemented in C as a pair of structs, defined as in Algorithms 3 and 4. The CAN message struct contains information about the sent and received CAN frames and is used by the CAN abstraction layer to read or write to the bus. The CAN payload struct is used to transmit CAN frames over the radio link, together with CAN interface data. As always, a message type indicator and the ID of the transmitting node is included.

4.4.2 Negotiation

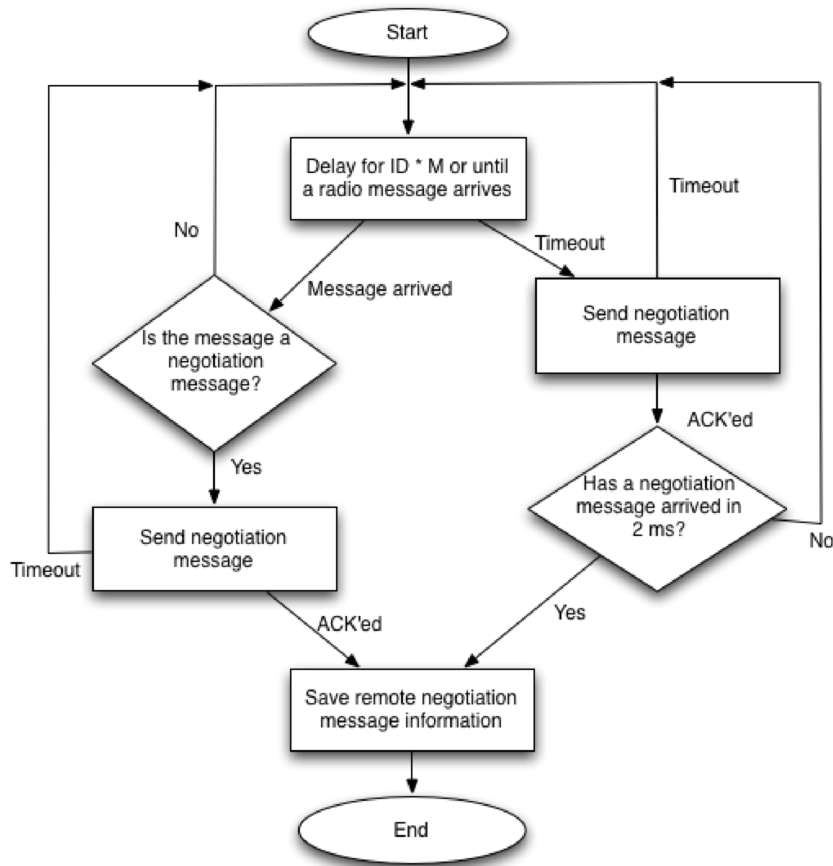

Figure 13: Flowchart for the negotiation mode

The negotiation process is charted in figure 13 and represents the handshake process of the system. The process is called on system startup and whenever either the listening or the transmitting processes fail or encounter an unexpected scenario. A delay based on the node ID is introduced to de-sync two nodes who might otherwise collide attempting to negotiate. If a negotiation message arrives before the delay runs out, a responding negotiation message is transmitted in response and the ID of the remote node is saved for future use. Should no message arrive before the timer runs out, a negotiation message is transmitted. If no responding message arrives within two milliseconds or the transmission times out, the process restarts. If a responding negotiation frame arrives, the remote ID is saved and the process terminates.

4.4.3 Listening mode

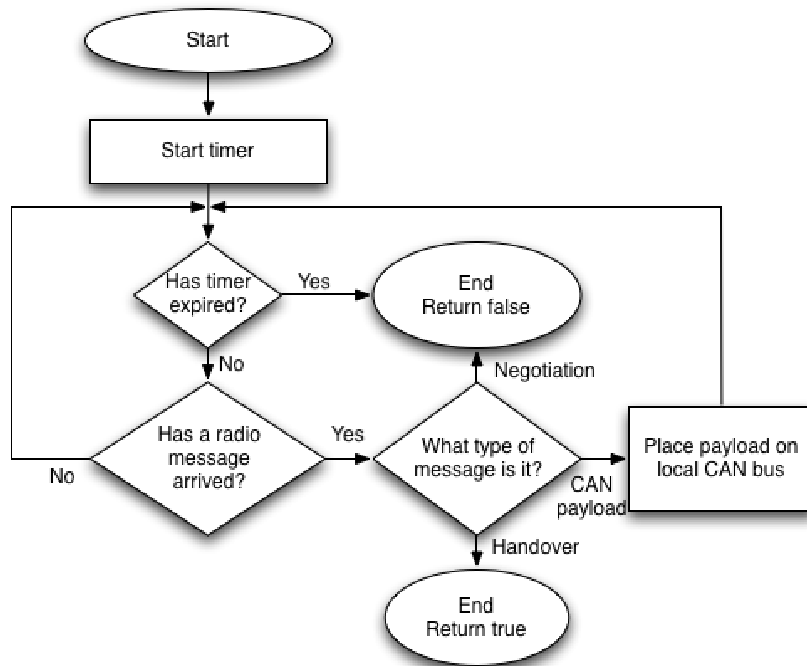

Figure 14: Flowchart for the listening mode

The objective of the listening mode, charted in figure 14, is to listen to the radio link for incoming messages while buffering local CAN messages. Incoming CAN payloads over the radio link are immediately transmitted on the local CAN busses until one of the following events occur; the timer expires, a negotiation message arrives or a handover message arrives. The proper way for the process to terminate is to process an incoming handover message, while having the timer expiring or a negotiation message arrive will imply that the transmitting node has failed.

4.4.4 Transmitting mode

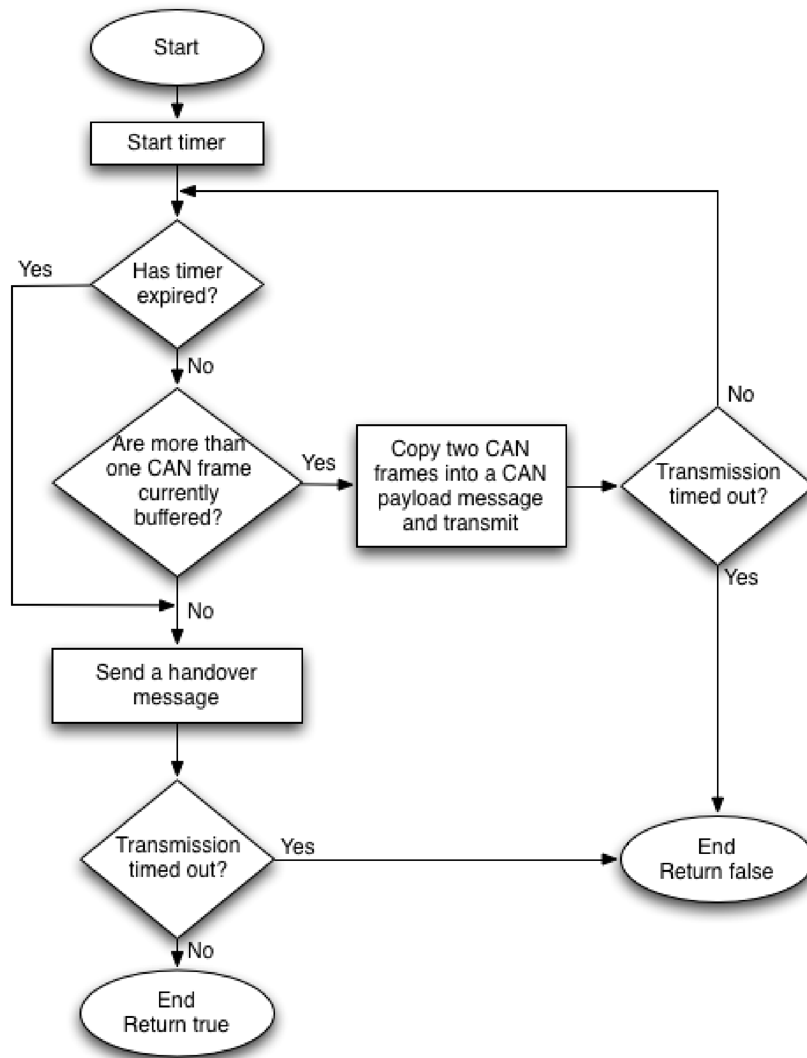

Figure 15: Flowchart for the transmitting mode

When put in transmitting mode, charted in figure 15, the node will transmit CAN messages from the local buffer until either the timer expires or there are no more messages to transmit. Should the timer expire, a handover message will be transmitted in order for the nodes to switch modes. In the case that the transmission of a payload or a handover times out, the process will terminate and the application will revert to negotiation mode.

5 Results

5.1 nRF24L01+ chip testing

Rudimentary testing of the nRF24L01+ chip revealed a non-documented limit of 2500 messages per second with a payload size of 32 bytes. This results in a hard throughput limit of $2500 * 32 * 8 = 640000$ kb/s, which at a glance will not accomodate a 1 Mb/s bus. However, the CAN payload is 99 bits long, as seen in the CAN message C struct in Algorithm 3, which is shorter than a CAN frame on the bus which with bit padding can be as long as 142 bits long. Effectively, the radio link employs data compression, compensating for the lower bitrate over the link.

With two CAN frames being transmitted within a payload message the proxy can theoretically transmit 5000 frames per second, which while not maximum load still represents a significant load on a 1 Mb/s bus and by a wide margin saturates a 500 kb/s bus, which is the most commonly encountered bitrate within motor vehicles.

5.2 Frame loss statistics

Statistical tests were conducted on the prototype board, using three settings of bitrate; 250, 500 and 1000 kb/s and varying bus loads. The tests were made using a script in Vector CANalyzer to test varying bus conditions and measuring frameloss. The script was set to transmit 500000 frames, at which point the script was halted and the number of dropped frames were counted.

Table 2: Test statistics for asymmetric load on the CAN bus

Bus load [%]	Bitrate [kb/s]	Runtime [s]	Frame loss	Frame loss per s
63	250	454	0	0
95	250	301	0	0
46	500	322	0	0
62	500	237	0	0
88	500	169	0	0
28	1000	272	0	0
39	1000	191	0	0

Table 2 shows the results of stress testing with a listening mode to transmitting mode ratio of 1:200, meaning that one node spends the majority of its time in transmitting mode while the other node spends an equal amount of time in listening mode. The software was configured with a maximum mode timer of 10 ms, meaning that the absolute maximum latency between transmission and reception of a CAN frame is 10 ms, though likely much shorter than that. The results show that out of ca 500000 transmitted frames, not a single one was lost or dropped in transition. Generating high bus loads on a 1 Mb/s bus was difficult as the PC card on the PC started to overflow as it could not generate frames quickly enough, however the communication appears to break down near the

60% bus load point. The reason for this breakdown is that frames are inserted into the CAN buffers faster than they can be transmitted over the radio link, leading to an ever-increasing buffer.

Table 3: Test statistics for symmetric load on the CAN bus

Bus load [%]	Bitrate [kb/s]	Runtime [s]	Frame loss	Frame loss per s
98	250	300	0	0
86	500	167	0.0076%	0.23
50	1000	149	0.0054%	0.18

Table 3 contains the results of stress testing at a listening mode to transmitting mode of 1:1, putting the bus load equal at both ends to determine the viability of placing the proxy in the middle of a busy CAN bus. As per the previous experiment, generating high loads on a 1 Mb/s bus was difficult with the controlled testing tools available. While the frame loss is no longer zero of a sample size of 500000 frames, the loss is low to the point of negligible.

6 Discussion

The project is still a work in progress, however the core functionality is firmly in place and functioning well.

Future development includes revising the PCB using smaller components and placing them in a more compact manner. The reason for using larger than necessary passive components and generous spacing is to ease the mounting of components on the board, speeding the process up slightly. Unfortunately, the use of small components is also a problem in the production as the mounting of the nRF24L01+ chip in its 3mm x 3mm QFN20 package is nearly impossible without a microscope and scalpel above the other necessary tools for surface mounting. For any production Broccoli Engineering AB is left with two major options; The continued use of breakout boards for the nRF24L01+ chip mounted directly on the PCB or ordering the PCBs with components mounted. Other size-cutting measures would be to replace the aluminum electrolytic capacitors with tantalum based capacitors, which are more expensive but also much smaller. The remaining size-cutting measure would be the exclusion of the JTAG interface, as it is by far one of the more area demanding components, the significant downside of this would be the requirement of pre-programmed microcontrollers mounted on the boards and the loss of ability to patch the software.

As for the software, one major desired functionality is the ability to configure a node via CAN messages. This would be implemented by reserving a CAN ID used for node configuration, this ID would not be proxied as other nodes. By setting specific payloads the node could be configured by for example letting the first byte be a configuration instruction and the second byte be an argument, allowing one to for example configure CAN bitrates on the fly. Another possible use of this reserved ID would be to query the nodes for diagnostic data on the proxies themselves, such as error counters. The functionality was never implemented as the instruction set for configuration would have to be determined more thoroughly, however the software was written with this functionality in mind and the implementation should be very easy.

It should be noted that the board is not necessarily exclusive to the use of motor vehicles, CAN busses are found in several applications and as the solution is very generic it can bridge any CAN networks outside of motor vehicles.

References

[1] BOSCH. CAN Specification Version 2.0. Accessed July 19, 2010. Available from http://www.semiconductors.bosch.de/pdf/can2spec.pdf

[2] CAN in Automation. Basic information on the CAN physical and data link layer by CAN in Automation (CiA). Accessed July 19 2010. Available from http://www.can-cia.org/pg/can/additional/can_basics_print.pdf

[3] CAN in Automation. CAN protocol. Accessed July 20, 2010. Available from http://can-cia.org/index.php?id=518

[4] Softing. CAN bus Error Handling - Error frame. Accessed July 26, 2010. Available from: http://www.softing.com/home/en/industrial-automation/products/can-bus/more-can-bus/error-handling/active-error-frame.php?navanchor=3010501

[5] Softing. CAN bus Error Handling - Overload frame. Accessed July 26, 2010. Available from: http://www.softing.com/home/en/industrial-automation/products/can-bus/more-can-bus/error-handling/overload-frame.php?navanchor=3010518

[6] Kvaser. Kvaser wireless CAN solutions. Accessed July 29 2010. Available from: http://www.kvaser.com/en/products/can/wireless.html

[7] ESD Electronics. CAN Interface for bluetooth. Accessed July 29, 2010. Available from: http://www.esd-electronics.com/german/products/CAN/can-bluetooth.htm

[8] Huaqun Guo, Lek Heng Ngoh, Yong Dong Wu, and Joseph Chee Ming Teo. Secure Wireless Vehicle Monitoring and Control. IEEE Asia-Pacific Conference on service computing, pp. 81-86, Singapore, 2009.

[9] SAE. Diagnostic Connector Equivalent to ISO/DIS 15031-3. Accessed Aug 25, 2010. Available from http://standards.sae.org/wip/j1962

[10] Interfacebus. OBDII Bus. Accessed Aug 25, 2010. Available from http://www.interfacebus.com/OBDII-pinout-signal-assingment.htm

[11] John B Anderson. Digital Transmission Engineering. 2nd ed. IEEE Press; 2005. pp. 133-135.

[12] Tai-Cheng Lee and Chin-Chi Chen. A mixed-signal GFSK demodulator for Bluetooth. Circuits and Systems II: IEEE transaction on express Briefs, vol. 53, no. 3, pp.197-201, March 2006.

[13] John Catsoulis. Designing embedded hardware. 2nd ed. O'Reilly Publishing; May 2005. pp 160-162.

7 Appendix

7.1 CAN Proxy Prototype gerber images

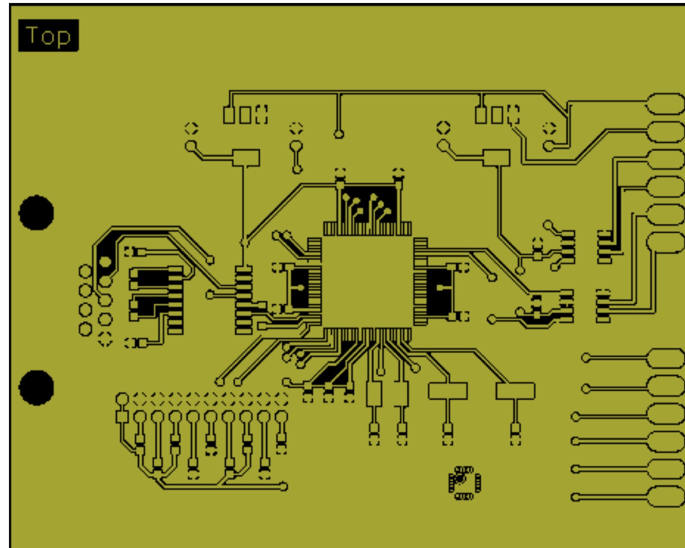

Figure 16: Gerber file, top copper layer

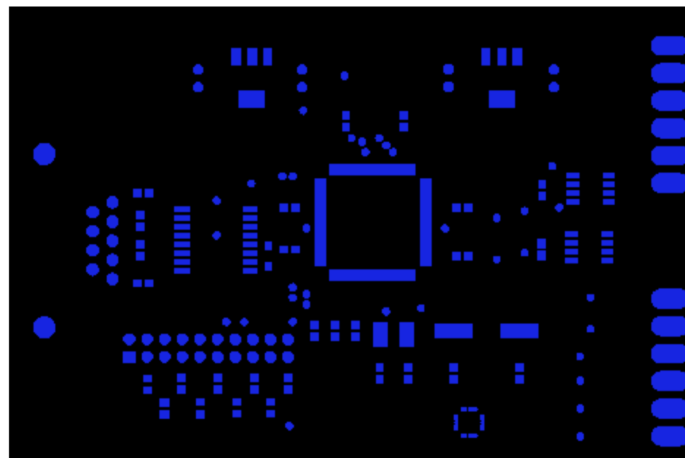

Figure 17: Gerber file, top solder resist layer

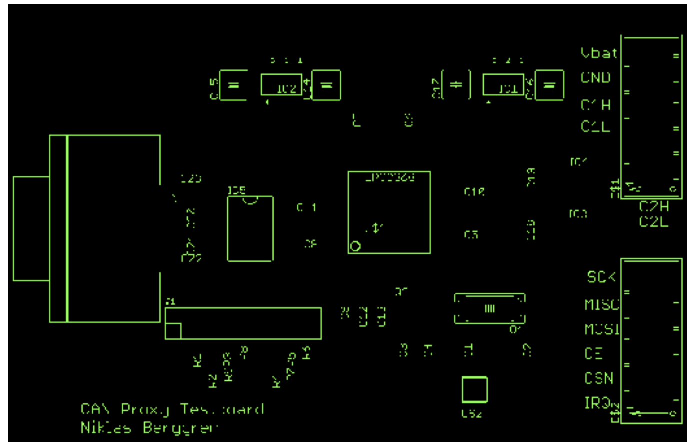

Figure 18: Gerber file, top silk layer

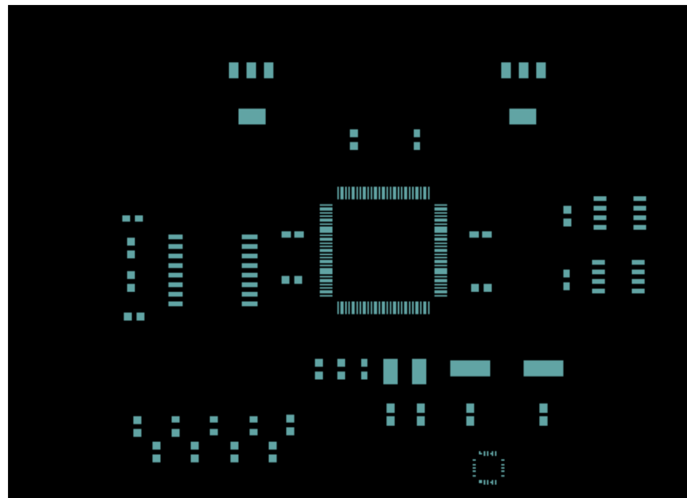

Figure 19: Gerber file, top paste

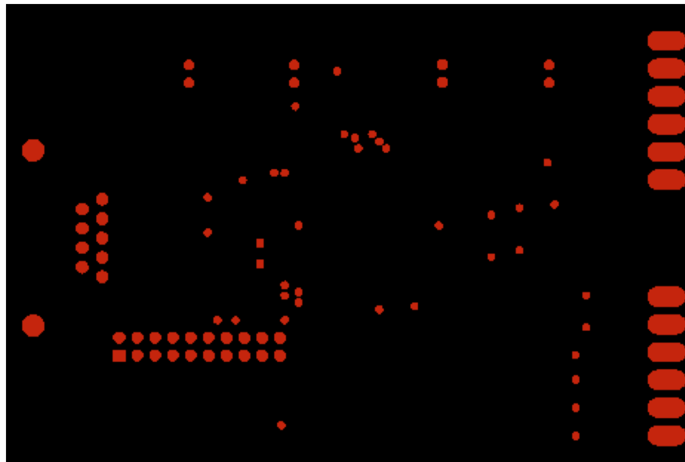

Figure 20: Gerber file, bottom solder resist

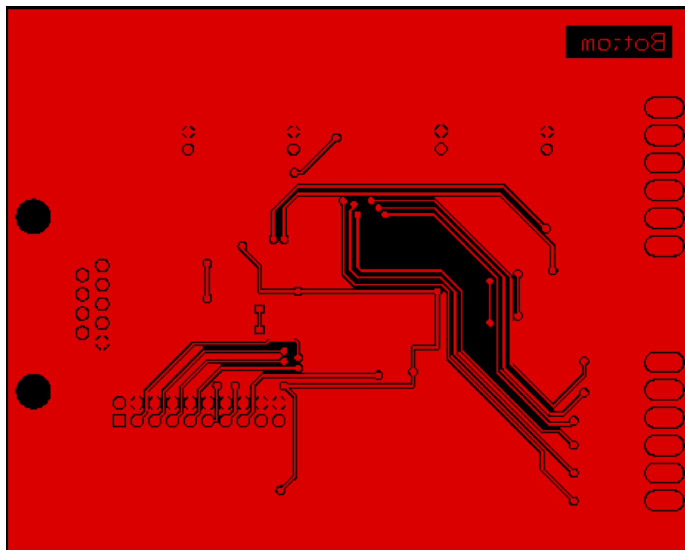

Figure 21: Gerber file, bottom copper layer

7.2 nRF24L01+ abstraction layer

The Wireless CAN Proxy utilizes a freely available abstraction layer for the nRF24L01+ chip from http://blog.diyembedded.com which is written for the LPC2100 platform. By minimal modification the layer works perfectly with the slightly more modern LPC2300 used for this project.

7.3 Microcontroller software source code

```
/**
 * CAN control functions
 */
#include "lpc2364.h"
#include "CAN.h"
#include "helpmacros.h"

// Initialize the can controller denoted by the variable controllerNo
void can_open(unsigned char controllerNo, unsigned int mode) {
  AFMR = 0x03; // Set the acceptance filter in bypass mode

  if(controllerNo == 1) {
      CAN1MOD = 0x01; // Set controller in reset mode
      CAN1BTR = mode; // Set the bit timing register according to the constants in CAN.h
      CAN1MOD = 0x00; // Initiation done, bring the interface up from reset mode
  }
  else {
      CAN2MOD = 0x01;
      CAN2BTR = mode;
      CAN2MOD = 0x00;
  }
}

// Transmit the CAN message msg on CAN interface controllerNo
void can_send_message(unsigned char controllerNo, can_message *msg) {
  if(controllerNo == 1) {
    //Wait here until at least one transmission buffer is available
    //Check bits 2, 10 or 18 in the status register for a free buffer
    while( (CAN1SR & 0x40404) == 0 ) {}

    // Copy message data to the available transmission buffer
    if(CAN1SR & 0x04) {
   // Copy frame information
      CAN1TFI1 = ((msg->X) << 31) | ((msg->DR) << 30) | ((msg->DLC) << 19);
      CAN1TID1 = msg->ID;       // Copy ID
      CAN1TDA1 = msg->data1; // Copy the first 4 bytes ...
      CAN1TDB1 = msg->data2; // ... and the second 4 bytes
      CAN1CMR = 0x21;     // Transmit the frame!
    }
    else if(CAN1SR & 0x400) {
      CAN1TFI2 = ((msg->X) << 31) | ((msg->DR) << 30) | ((msg->DLC) << 19);
      CAN1TID2 = msg->ID;
      CAN1TDA2 = msg->data1;
      CAN1TDB2 = msg->data2;
      CAN1CMR = 0x41;
    }
    else {
      CAN1TFI3 = ((msg->X) << 31) | ((msg->DR) << 30) | ((msg->DLC) << 19);
      CAN1TID3 = msg->ID;
      CAN1TDA3 = msg->data1;
      CAN1TDB3 = msg->data2;
      CAN1CMR = 0x81;
    }
  }
  else {
    // Identical code for using CAN controller 2:
    while( (CAN2SR & 0x40404) == 0 ) {}

    // Copy message data to the available transmission buffer
    if(CAN2SR & 0x04) {
      CAN2TFI1 = ((msg->X) << CAN_EXTENDED_FRAME) | ((msg->DR) << CAN_DATA_REMOTE) |
((msg->DLC) << CAN_DLC);
      CAN2TID1 = msg->ID;       // Copy ID
      CAN2TDA1 = msg->data1; // Copy the first 4 bytes ...
```

```
      CAN2TDB1 = msg->data2; // ... and the second 4 bytes
      CAN2CMR = 0x21;     // Transmit the frame!
    }
    else if(CAN2SR & 0x400) {
      CAN2TFI2 = ((msg->X) << CAN_EXTENDED_FRAME) | ((msg->DR) << CAN_DATA_REMOTE) |
((msg->DLC) << CAN_DLC);
      CAN2TID2 = msg->ID;
      CAN2TDA2 = msg->data1;
      CAN2TDB2 = msg->data2;
      CAN2CMR = 0x41;
    }
    else {
      CAN2TFI3 = ((msg->X) << CAN_EXTENDED_FRAME) | ((msg->DR) << CAN_DATA_REMOTE) |
((msg->DLC) << CAN_DLC);
      CAN2TID3 = msg->ID;
      CAN2TDA3 = msg->data1;
      CAN2TDB3 = msg->data2;
      CAN2CMR = 0x81;
    }
  }
}

// Investigate if a new message has been received on interface controllerNo
bool can_new_message(unsigned char controllerNo) {
  if(controllerNo == 1) {
    return (CAN1GSR & 0x01);
  }
  else {
    return (CAN2GSR & 0x01);
  }
}

// Retrieve the new message from the CAN buffer
void can_get_message(unsigned char controllerNo, can_message *rec) {
  if(controllerNo == 1) {
    // Copy the received message to the provided memory
    rec->ID =        CAN1RID;
    rec->X =         (CAN1RFS >> CAN_EXTENDED_FRAME) & 0x01;
    rec->DR =        (CAN1RFS >> CAN_DATA_REMOTE) & 0x01;
    rec->DLC =       (CAN1RFS >> CAN_DLC) & 0x0f;
    rec->data1 =     CAN1RDA;
    rec->data2 =     CAN1RDB;

    // Release the received data for the next message
    CAN1CMR = 0x04;
  }
  else {
    // Copy the received message to the provided memory
    rec->ID =        CAN2RID;
    rec->X =         (CAN2RFS >> CAN_EXTENDED_FRAME) & 0x01;
    rec->DR =        (CAN2RFS >> CAN_DATA_REMOTE) & 0x01;
    rec->DLC =       (CAN2RFS >> CAN_DLC) & 0x0f;
    rec->data1 =     CAN2RDA;
    rec->data2 =     CAN2RDB;

    // Release the received data for the next message
    CAN2CMR = 0x04;
  }
}
```

```
/**
 * CAN control functions
 */
#ifndef _CAN_H_
#include "helpmacros.h"

// Define the divider for various CAN speeds, sample as close to Tseg1 = 8, Tseg2 = 1 as
possible.
// Assume the perhepial divider is 1, so that perhepial clock is the CPU clock (72 MHz)
// Formula: Bitrate = CCLK / (VPDIV * (BRP+1) * ( (Tseg1+1) + 1 + (Tseg2+1) ) )
#define CAN1000KBIT72MHZ 0x180005
#define CAN500KBIT72MHZ  0x18000B
#define CAN250KBIT72MHZ  0x180017
#define CAN125KBIT72MHZ  0x18002F

// Define which bits in the CAN registers control what, used for bit shifting
#define CAN_EXTENDED_FRAME 31 // Bit 31 indicates whether a frame is extended or not
#define CAN_DATA_REMOTE    30 // Bit 30 indicates wheter a frame is data or remote
#define CAN_DLC            19 // Bits 19-16 give the data length code for a frame

// Define a struct describing a CAN frame
typedef struct can_message {
  unsigned int ID : 29;      // 29 bit ID field
  unsigned int X : 1;        // Extended frame format flag
  unsigned int DR : 1;       // Data/Remote flag
  unsigned int DLC : 4;       // Data Length field
  unsigned int data1 : 32;    // Data field one
  unsigned int data2 : 32;    // Data field two
} __attribute__ ((packed)) can_message;

// Initialize the can controller denoted by the variable controllerNo
void can_open(unsigned char controllerNo, unsigned int mode);

// Transmit the CAN message msg on CAN interface controllerNo
void can_send_message(unsigned char controllerNo, can_message *msg);

// Investigate whether a new message has been received on interface controllerNo
bool can_new_message(unsigned char controllerNo);

// Retrieve the new message from the CAN buffer, write it into rec
void can_get_message(unsigned char controllerNo, can_message *rec);

#define _CAN_H_
#endif
```

```
/**
 * Delay functions, implemented in delay.c through timers
 */
#include "lpc2364.h"
#include "delays.h"
#include "helpmacros.h"

// Microsecond delay
void delayUS(unsigned int microseconds) {
    T0TCR = 0x02;                      //Reset timer
    T0PR  = 0x00;                      //Set prescaler to zero
    T0MR0 = microseconds * (72 -1); //Wait for uSeconds/1000000*CPU freq
    T0IR  = 0xff;                      //Reset all interrupts
    T0MCR = 0x04;                     //stop timer on match
    T0TCR = 0x01;                      //Start timer

    //Loop until the timer is done
    while (T0TCR & 0x01);
}

// Millisecond delay
void delayMS(unsigned int milliseconds) {
    T0TCR = 0x02;                          //Reset timer
    T0PR  = 0x00;                          //Set prescaler to zero
    T0MR0 = milliseconds * (72000 -1);   //Wait for milliseconds/1000*CPU freq
    T0IR  = 0xff;                          //Reset all interrupts
    T0MCR = 0x04;                         //stop timer on match
    T0TCR = 0x01;                          //Start timer

    //Loop until the timer is done
    while (T0TCR & 0x01);
}

// Second delay
void delayS(unsigned int seconds) {
    T0TCR = 0x02;                      //Reset timer
    T0PR  = 0x00;                      //Set prescaler to zero
    T0MR0 = seconds * (72000000 -1);//Wait for seconds*CPU freq
    T0IR  = 0xff;                      //Reset all interrupts
    T0MCR = 0x04;                     //stop timer on match
    T0TCR = 0x01;                      //Start timer

    //Loop until the timer is done
    while (T0TCR & 0x01);
}

// Non-blocking timer running on TIMER1
void start_timer(unsigned int microseconds) {
    T1TCR = 0x02;                      //Reset timer
    T1PR  = 0x00;                      //Set prescaler to zero
    T1MR0 = microseconds * (72 -1); //Wait for uSeconds/1000000*CPU freq
    T1IR  = 0xff;                      //Reset all interrupts
    T1MCR = 0x04;                     //stop timer on match
    T1TCR = 0x01;                      //Start timer
}

// Function to check wheter TIMER1 has run out or not
bool time_is_up() {
  if(T1TCR & 0x01)
    return false;
  else
    return true;
}
```

```
/**
 * Delay functions, implemented in delay.c through timers
 */
#include "helpmacros.h"

#ifndef _DELAY_H_
#define _DELAY_H_

// Delay functions
void delayUS(unsigned int microseconds);
void delayMS(unsigned int milliseconds);
void delayS(unsigned int seconds);

// Timer functions
void start_timer(unsigned int microseconds);
bool time_is_up();

#endif //_DELAY_H_
```

```
/**
 * Various helpful macros and definitions to achieve slightly cleaner code
 */

#ifndef _CANPROXY_MACROS

#define bool unsigned char // Simple boolean
#define false 0
#define true !false

#define _CANPROXY_MACROS
#endif
```

```c
#include <string.h>
#include "lpc2364.h"
#include "SPI.h"
#include "CAN.h"
#include "nrf24l01.h"
#include "wcom.h"

// Define the ID and Active/Passive of this particular node
#define NODE_ID 4

void InitializeIO();

void main() {
  // Various initializations
  InitializeIO(); // Initialize IO
  spi_open(); // Open the SPI and initialize the interface for master mode
  can_open(1, CAN500KBIT72MHZ);  // Open CAN interface 1
  can_open(2, CAN250KBIT72MHZ);  // Open CAN interface 2

  // Initialize the nRF24L01+ chip
  delayMS(20);
  nrf24l01_initialize_debug(true, WSIZE, true); // Initialize the nRF24L01+ chip

  // Start by setting the local ID
  w_set_ID(NODE_ID);

  // Main loop
  while(1) {
    // Find a node to communicate with
    w_negotiate();

    // Low ID gets to start in transmit mode, high ID in listen mode
    while(1) {
      if(w_get_remote_ID() < NODE_ID) {
        if(w_listen_mode() == false) {
          break;  // w_listen_mode() required re-negotiation, break the loop
        }
        if(w_transmit_mode() == false) {
          break;  // w_transmit_mode() required re-negotiation, break the loop
        }
      }
      else {
        if(w_transmit_mode() == false) {
          break;  // w_transmit_mode() required re-negotiation, break the loop
        }
        if(w_listen_mode() == false) {
          break;  // w_listen_mode() required re-negotiation, break the loop
        }
      }
    }
  }
}

// Initializes the various IO functions of the board
void InitializeIO() {
  SCS |= 0x03; // Use fast GPIO

  // CAN controllers are unpowered on reset, turn them on
  PCONP |= 0b110000000000000;

  // Setup pin functions, defaults to GPIO
  // P0:
  PINSEL0 = 0xC0000A55; // 0b11000000000000000000101001010101 CAN0, CAN1, UART0, SCK of
SPI
```

```
  PINSEL1 = 0x0000003F; // 0b00000000000000000000000000111111 MISO, MOSI, SSEL of SPI

  // Kill the damn embedded trace module!!1one
  // Angry rant aside, PINSEL10 reseting to an undefined value was the cause of vast 
headaches
  PINSEL10 = 0x00;

  // No special setup required for P1, P2, P3
  // Setup pin directions, 0 = input, 1 = output
  // List of outputs:
  //                  SPI CE on P2.6
  //                  SPI CSN on P2.5
  FIO2DIR = 0x60; // 0b00000000000000000000000001100000 Set direction on GPIO pins CE/
CSN/IRQ

  // Set perhepial clocks to use the CPU core clock, rather than CPU core clock/4
  // Clocks set: CAN Acceptance filter, CAN1, CAN2, SPI, UART0, TIMER0 and TIMER1
  PCLKSEL0 = 0x54010054; // 0b01010100000000010000000001010100
}
```

```c
/**
 * SPI control functions
 */
#include "lpc2364.h"
#include "SPI.h"
#include "helpmacros.h"

// Open the SPI interface here
void spi_open() {
  S0SPCCR = 0x0C; // Divide CPU clock by 12 for 6 MHz SPI clock (Max 8)
  S0SPCR = 0x20;  // Set master mode
}

// Read from the SPI data register, abort if overrun.
int spi_read() {
  if( S0SPSR & 0x08 ) // Check for read overrun
    return -1;
  else
    return S0SPDR;
}

// Transmit a byte over the SPI
unsigned char spi_write(unsigned char byte) {
  int rec;

  S0SPDR = byte; // Place the payload on register for transfer
  while(!(S0SPSR & 0x80)) {} // Wait until done

  // Because a transmission of a byte implies a reading of a byte:
  rec = spi_read();
  while( rec == -1 ) {
    rec = spi_read(); // Keep reading until success
  }

  return (unsigned char)rec;
}
```

```
/**
 * SPI control functions
 */

void spi_open();
int spi_read();
unsigned char spi_write(unsigned char byte);
```

```c
/**
 * UART (RS232) related functions
 */
#include "lpc2364.h"
#include "helpmacros.h"
#include "uart.h"

//Open for UART communications
void uart0_open() {
  U0LCR = 0b10000011;
  U0DLM = 0b00000001;
  U0DLL = 0b11010101; // Calculate divider to achieve the desired bitrate
  U0FCR = 0b00000001;
  U0FDR = 0b00010000; // Don't use prescaler
  U0FCR = 0b00000001;
  U0TER = 0b10000000;
  U0LCR = 0b00000011; // Set 8-bit character length
  U0SCR = U0RBR;
}

//Simple transmission of a string
void uart0_send_string(char *data) {
  unsigned short iterator = 0;
  while(iterator < sizeof(data) && data[iterator] != '\0') {
    while((U0LSR & 0b100000) == 0) {} // Wait here until ready to transmit

    if(data[iterator] != '\n') {
      U0THR = data[iterator]; // Place character on the transmission buffer
    }
    else {
      uart0_send_crlf(); // Send CRLF if a newline was detected
    }

    iterator++; // Increase the iterator; move on to the next character
  }
}

//Send CRLF
void uart0_send_crlf() {
  while((U0LSR & 0b100000) == 0) {} // Wait here until ready to transmit
  U0THR = 0x0A; // Newline
  while((U0LSR & 0b100000) == 0) {} // Wait again until ready
  U0THR = 0x0D; // Carriage return
}

//Check if data is ready for reading
bool uart0_read_ready() {
  return (U0LSR & 0b1); // Return the read bit in the Line Status Register
}

//Retrieve data
unsigned char uart0_read() {
  return U0RBR; // Just return the read buffer
}
```

```
/**
 * UART (RS232) related functions
 */
#include "helpmacros.h"

void uart0_open(); //Open for UART communications
void uart0_send_string(char *data); //Simple transmission of a string
void uart0_send_crlf(); //CRLF

bool uart0_read_ready(); // Check if data is ready for reading
unsigned char uart0_read(); // Retrieve data
```

```
/**
 * Higher level functions for wireless communications
 */
#include <string.h>
#include "CAN.h"
#include "wcom.h"
#include "delays.h"
#include "nrf24l01.h"

// Set up variables and buffers used between the functions
unsigned char spibuf[WSIZE];            // Buffer to read/write messages from the radio
chip to/from
w_negotiation_message rnmsg, lnmsg;     // Remote and local negotiation messages
can_message canbuf[CANBUF_SIZE];        // Buffer to store CAN frames in
unsigned char canctrl[CANBUF_SIZE];     // Partner array to canbuf, holds controller
information
unsigned int canbufptr = 0;             // Buffer pointer to the next empty place

// Initialization function, sets the local node ID.
// This function must be called before any other w_* functions, failure to do so will
result in bad mojo. Also null pointers.
void w_set_ID(unsigned char nodeID) {
  lnmsg.pkgtype = WCOM_NEGOTIATION_MESSAGE;
  lnmsg.nodeID = nodeID;
}

// Retrieve the node ID of the current partner
unsigned char w_get_remote_ID() {
  return rnmsg.nodeID;
}

// Probe for new CAN frames and add to buffer if they exist
void w_monitor_can_messages() {
  can_message tmpcan; // Temporary storage for CAN messages
  unsigned short ctrl = 0;

  if(can_new_message(1)) {
    can_get_message(1, &tmpcan);
    ctrl = 1;
  }
  else if(can_new_message(2)) {
    can_get_message(2, &tmpcan);
    ctrl = 2;
  }

  // INSERT CAN CONFIG FUNCTION CALL HERE!
  // can_config(&tmpcan);

  // Add new message to buffer if it exists
  if(ctrl > 0) {
    if(canbufptr < CANBUF_SIZE-1) {
      // Copy the message to the buffer, increase the value of the array pointer
      canbuf[canbufptr] = tmpcan;
      canctrl[canbufptr] = ctrl;
      canbufptr++;
    }
    else {
      // Buffer overflow. Send a CAN frame on the local net, light a LED up, or signal in
some other way
      canbufptr = 0;
    }
  }
}
```

```
// Negotiation function, blocks until a partner node is found
void w_negotiate() {
  can_message tmpcan;

  // Main loop
  nrf24l01_clear_flush();
  while(1) {
    nrf24l01_set_as_rx(true);

    // Start a timer waiting for ID*multiplier milliseconds until moving on
    start_timer( lnmsg.nodeID*1000*WCOM_NEGOTIATION_DELAY );

    // Buffer CAN while waiting for another node to show up or the timer to expire
    while(!(time_is_up() || nrf24l01_irq_pin_active())) {
      w_monitor_can_messages();
    }

    // Check if another node has sent a negotiation frame
    if(nrf24l01_irq_pin_active()) {
      // Message received, check if it's a negotiation message
      nrf24l01_read_rx_payload(spibuf, WSIZE);
      nrf24l01_irq_clear_all();
      if( ((w_negotiation_message*)spibuf)->pkgtype == WCOM_NEGOTIATION_MESSAGE ) {
        // Negotiation message received, save the negotiation message and reply with the
local negotiation message
        memcpy(&rnmsg, spibuf, sizeof(w_negotiation_message));
        nrf24l01_set_as_tx();
        delayUS(130);
        nrf24l01_write_tx_payload((char*)&lnmsg, WSIZE, true);

        // Buffer messages while waiting for the interrupt
        while(!nrf24l01_irq_pin_active()) {
          w_monitor_can_messages();
        }
        if(nrf24l01_irq_tx_ds_active()) {
          // Message was ACK'ed, break the loop and proceed
          break;
        }
        else {
          // Transmission timed out, ignore and move on
          nrf24l01_clear_flush();
        }
      }
    }

    // No one contacted this node, attempt to initiate contact
    else if(time_is_up()) {
      nrf24l01_set_as_tx();
      delayUS(130);
      nrf24l01_write_tx_payload((char*)&lnmsg, WSIZE, true);

      // Buffer messages while waiting for the interrupt
      while(!nrf24l01_irq_pin_active()) {
         w_monitor_can_messages();
      }
      if(nrf24l01_irq_tx_ds_active()) {
        // Transmission was ACK'ed, read incoming reply
        nrf24l01_irq_clear_all();
        nrf24l01_set_as_rx(true);

        start_timer(2000);
        // Buffer more CAN while waiting
        while(!(time_is_up() || nrf24l01_irq_pin_active())) {
          w_monitor_can_messages();
```

```
        }

        if(nrf24l01_irq_pin_active()) {
          // A reply was received, check that it's a negotiation message
          nrf24l01_read_rx_payload(spibuf, WSIZE);
          nrf24l01_irq_clear_all();
          if( ((w_negotiation_message*)spibuf)->pkgtype == WCOM_NEGOTIATION_MESSAGE ) {
            // Hooray, a negotiation message was received. Copy information.
            memcpy(&rnmsg, spibuf, sizeof(w_negotiation_message));
            break;
          }
          else {
            // False alarm, the incoming message was not a negotiation message, carry on
          }
        }
      }
      else {
        // Transmission timed out, ignore and move on
        nrf24l01_clear_flush();
      }
    }
  }
}

// Listening mode, buffer incoming CAN messages while placing incoming CAN payloads on
the bus
// Returns false if re-negotiation is required, true otherwise
bool w_listen_mode() {
  can_message tmpcan;              // Temporary storage for new incoming CAN frames
  w_can_payload_message *tmpwcp;   // Struct converter for incoming payloads

  // Enter listen mode
  nrf24l01_set_as_rx(true);
  //delayUS(130);
  start_timer(MODE_MAX_TIME + LISTENER_TOLERANCE); // Start the timer, include the
tolerance

  // Main loop
  while(1) {
    w_monitor_can_messages();

    if(time_is_up()) {
      // Timer reached the target time before a handover was received, require a
renegotiation
      return false;
    }
    else if(nrf24l01_irq_pin_active()) {
      // A new frame was received
      nrf24l01_read_rx_payload(spibuf, WSIZE);
      nrf24l01_irq_clear_all();

      // Check that it's not a negotiation message from our remote. If so, require a
renegotiation
      if( ((w_negotiation_message*)spibuf)->pkgtype == WCOM_NEGOTIATION_MESSAGE &&
          ((w_negotiation_message*)spibuf)->nodeID == rnmsg.nodeID) {
        return false;
      }

      // Check if it's a handover message so that this node can enter transmit mode
      else if( ((w_handover_message*)spibuf)->pkgtype == WCOM_HANDOVER_MESSAGE &&
               ((w_handover_message*)spibuf)->nodeID == rnmsg.nodeID) {
                            tmpcan.ID = lnmsg.nodeID;
        return true;
      }
```

```
      // Check if it's a CAN payload message, put contents on the local CAN bus if so
      else if( ((w_can_payload_message*)spibuf)->pkgtype == WCOM_CAN_PAYLOAD_MESSAGE &&
               ((w_can_payload_message*)spibuf)->nodeID == rnmsg.nodeID) {
        tmpwcp = (w_can_payload_message*)spibuf;

          can_send_message(tmpwcp->ctrl1, &(tmpwcp->msg1));
          can_send_message(tmpwcp->ctrl2, &(tmpwcp->msg2));
      }
    }
  }
}

// Transmitting mode, transmit as much as possible from the buffer while still handling
incoming CAN
// Returns false if re-negotiation is required, true otherwise
bool w_transmit_mode() {
  can_message tmpcan;             // Temporary storage for new incoming CAN frames
  w_handover_message hm;          // The local transmission handover
  w_can_payload_message cpm;      // Temporary storage for a CAN payload message

  hm.pkgtype = WCOM_HANDOVER_MESSAGE;
  hm.nodeID = lnmsg.nodeID;

  cpm.pkgtype = WCOM_CAN_PAYLOAD_MESSAGE;
  cpm.nodeID = lnmsg.nodeID;

  // Enter transmit mode
  nrf24l01_set_as_tx();

  // Start the timer
  start_timer(MODE_MAX_TIME);

  // Main loop, run as long as there are more than one CAN frames ready in the buffer and
the
  // timer doesn't run out
  while(canbufptr > 1 && !time_is_up()) {
    w_monitor_can_messages();

    // Prepare a message in the CAN payload struct
    canbufptr--;
    cpm.ctrl1 = canctrl[canbufptr];
    cpm.msg1 = canbuf[canbufptr];

    canbufptr--;
    cpm.ctrl2 = canctrl[canbufptr];
    cpm.msg2 = canbuf[canbufptr];

    nrf24l01_write_tx_payload((char*)&cpm, WSIZE, true);  // Transmit the message
    // More CAN buffering while waiting
    while(!nrf24l01_irq_pin_active()) {
      w_monitor_can_messages();
    }
    if(nrf24l01_irq_tx_ds_active()) {
      // Payload successfully sent
      nrf24l01_irq_clear_all();
    }
    else {
      // Transmission timed out, force re-negotiation
      nrf24l01_clear_flush();
      return false;
    }
  }
```

```
  // The buffer has been reduced to 1 or less, send a handover
  memcpy(spibuf, &hm, WSIZE);
  nrf24l01_write_tx_payload(spibuf, WSIZE, true);
  // Guess what? MORE MONITORING!
  while(!nrf24l01_irq_pin_active()) {
    w_monitor_can_messages();
  }
  if(nrf24l01_irq_tx_ds_active()) {
    // Handover successfully sent
    nrf24l01_irq_clear_all();
    return true;
  }
  else {
    // Transmission timed out, force re-negotiation
    nrf24l01_clear_flush();
    return false;
  }
}
```

```
/**
 * Higher level functions for wireless communications
 */
#include "CAN.h"
#ifndef _WCOM_H_

// Define the largest radio payload size, sizeof(w_can_payload_message)
#define WSIZE 32

// Define the maximum of how many CAN messages that are buffered. Set as high as
possible.
#define CANBUF_SIZE 150

// Define the maximum time in uS a node can stay in transmit or listen mode
#define MODE_MAX_TIME 10000 // 10 ms

// Define the extra leisure time the listener gives the transmitter until deciding
// that the other node has broken protocol and reverts to negotiation mode
#define LISTENER_TOLERANCE MODE_MAX_TIME/4

// Define possible message type flags, used in the pkgtype fields
#define WCOM_CAN_PAYLOAD_MESSAGE 'c'
#define WCOM_NEGOTIATION_MESSAGE 'n'
#define WCOM_HANDOVER_MESSAGE 'h'

// Define a multiplicant of the ID to delay in the negotiation
#define WCOM_NEGOTIATION_DELAY 5

// Define a struct describing a payload package, this struct is used as a pointer to one
of
// the CAN buffers when reading or writing data, never declared by itself.
typedef struct w_can_payload_message {
  unsigned char pkgtype;            // Message type flag
  unsigned char nodeID;             // Transmitter ID
  unsigned char ctrl1 : 2;          // ID of the controller the first message belongs to
  unsigned char ctrl2 : 2;          // ID of the controller the second message belongs
to
  can_message msg1;             // The actual CAN messages
  can_message msg2;
} __attribute__ ((packed)) w_can_payload_message;

// Define a struct used to describe a negotiation message, containing the sending node's
ID
typedef struct w_negotiation_message {
  unsigned char pkgtype;           // Message type flag
  unsigned char nodeID;
} w_negotiation_message;

// Define a struct used to describe a handover message, identical to the negotiation
message except
// the pkgtype.
typedef struct w_handover_message {
  unsigned char pkgtype;
  unsigned char nodeID;
} w_handover_message;

// Initialization function, sets the local node ID
void w_set_ID(unsigned char nodeID);

// Retrieve the node ID of the current partner
unsigned char w_get_remote_ID();

// Add a CAN message to the buffer
void w_monitor_can_messages();
```

```
// Negotiation function, blocks until a receptive node is found.
void w_negotiate();

// Listening mode, buffer incoming CAN messages while placing incoming CAN payloads on
the bus
// Returns false if re-negotiation is required, true otherwise
bool w_listen_mode();

// Transmitting mode, transmit as much as possible from the buffer while still handling
incoming CAN
// Returns false if re-negotiation is required, true otherwise
bool w_transmit_mode();

#define _WCOM_H_
#endif
```