



# Improved traffic safety by wireless vehicular communication

Dimitrios Vlastaras  
mail@dimme.net

Daniel Lestón  
daniel@leston.me

Department of Electrical and Information Technology  
Lund University

Advisor: Fredrik Tufvesson

June 17, 2013

Printed in Sweden  
E-huset, Lund, 2013

---

# Abstract

---

In tomorrow's vehicle industry vehicles will have the ability to communicate and cooperate with each other in order to avoid collisions and provide useful information to each other. However, for this cooperation to be possible all vehicles will have to be equipped with compatible wireless 802.11p modules that implement the ITS-G5 standard. During the implementation phase of the system there will be plenty of older vehicles without such equipment.

This thesis addresses this problem by developing the hardware and software for a road side unit called *Drive ITS*. It consists of a universal medium range radar that detects older vehicles, a 802.11p modem that forwards their position and speed vectors to newer vehicles and an embedded system that utilizes and integrates those two parts.

The hardware for the embedded system is divided in two main parts; a microcontroller board and a single-board microcomputer. The software is written in two programming languages; C++ for the microcontroller and Java for the microcomputer.

Tests have been performed by comparing Drive ITS results to results from other vehicles that already implement the ITS-G5 standard and it has been confirmed that the system works as it was intended to.

This solution will prevent potential accidents of newer ITS-G5 vehicles with older ordinary vehicles thus saving human lives.



---

## Acknowledgements

---

We would like to thank Linus Conradson, Daniel Horstmark and Mikael Nilsson for their help while performing tests at Volvo Cars in Gothenburg. We are also grateful to Martin Nilsson for helping us with the manufacturing process of the PCB. Last but not least we would like to thank Taimoor Abbas for his help during the project, our families and girlfriends for their support and of course our advisor Assoc. Prof. Fredrik Tufvesson for guiding us through the project.



---

## Division of Labour

---

Dimitrios Vlastaras' main responsibility was the development of the software for the system, in particular:

- Development of the software running on the AVR ATMEGA328P-PU microcontroller.
- Development of the daemon running on the Raspberry Pi.
- Software configuration of the Raspberry Pi to enable serial communications.

Daniel Lestón's main responsibility was the development of the hardware for the system, in particular:

- Development of the CAN-bus hardware shield for the Raspberry Pi.
- Design of the case enclosure for the system.

Outside of their main responsibilities both Dimitrios and Daniel have cooperated and helped each other in their respective fields.





---

# Table of Contents

---

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
1.2	State of the art . . . . .	3
<b>2</b>	<b>Objectives</b>	<b>5</b>
2.1	Goal . . . . .	5
<b>3</b>	<b>Hardware</b>	<b>7</b>
3.1	CAN-bus interpreter . . . . .	8
3.2	Microcontroller to Raspberry Pi interface . . . . .	19
3.3	Ethernet over USB interface . . . . .	22
3.4	Programmability . . . . .	22
3.5	Prototypes . . . . .	22
<b>4</b>	<b>Software</b>	<b>27</b>
4.1	Reading data from the radar . . . . .	27
4.2	Creating a model . . . . .	33
4.3	Creating Cooperative Awareness Messages . . . . .	34
4.4	Broadcasting Cooperative Awareness Messages . . . . .	36
<b>5</b>	<b>Case enclosure</b>	<b>39</b>
5.1	Design . . . . .	39
5.2	Production and materials . . . . .	40
<b>6</b>	<b>Tests and results</b>	<b>43</b>
6.1	Test session at Volvo Cars . . . . .	43
6.2	Final product . . . . .	44
<b>7</b>	<b>Discussion</b>	<b>47</b>
7.1	Conclusions . . . . .	47
7.2	Future development . . . . .	48
	<b>References</b>	<b>51</b>

<b>A</b>	<b>UML diagrams</b>	<b>53</b>
<b>B</b>	<b>Programming manual</b>	<b>57</b>
B.1	AVR microcontroller . . . . .	57
B.2	Raspberry Pi . . . . .	61
<b>C</b>	<b>Installation manual</b>	<b>63</b>
C.1	Placement and configuration . . . . .	63

---

## List of Figures

---

1.1	Example of the Drive ITS system. . . . .	2
1.2	Applications within the ITS framework <sup>1</sup> . . . . .	3
3.1	Block diagram of the system. . . . .	7
3.2	Block diagram of the shield. . . . .	8
3.3	Architecture of CAN [2]. . . . .	9
3.4	Drive ITS CAN bus topology. . . . .	9
3.5	Possible connections of the Rs pin [2]. . . . .	10
3.6	Bit timing diagram [4]. . . . .	14
3.7	Block diagram of the level shifter. . . . .	19
3.8	N-FET 3.3V → 5V circuit. . . . .	20
3.9	Voltage divider to perform the 5V → 3.3V conversion. . . . .	21
3.10	Breadboard prototype. . . . .	24
3.11	Perfboard prototype. . . . .	25
3.12	PCB elements. . . . .	26
4.1	An 802.11p packet dump as shown in Wireshark. . . . .	35
5.1	3D model of the case enclosure. . . . .	40
6.1	Captured packets from Drive ITS and Drive C2X systems. . . . .	44
6.2	General view of the Drive ITS prototype. . . . .	45
6.3	Lateral view of the Drive ITS prototype, featuring the shield connector. . . . .	45
6.4	Photography of a final assembly inside the case enclosure, produced using FDM. . . . .	46
A.1	UML diagram for the radarp package. . . . .	54
A.2	UML diagram for the radarp.its package. . . . .	55
A.3	UML diagram for the radarp.kinematics package. . . . .	56
A.4	UML diagram for the radarp.serial package. . . . .	56
C.1	Placement of the radar. . . . .	63



---

## List of Tables

---

3.1	Register CNF1 overview. . . . .	15
3.2	Register CNF2 overview. . . . .	15
3.3	Register CNF3 overview. . . . .	15
3.4	Register CANCTRL overview. . . . .	17
3.5	Register TXRTSCTRL overview. . . . .	17
3.6	Description of SPI commands [3]. . . . .	18
4.1	CAN frame requesting real traffic. . . . .	29
4.2	CAN frame requesting simulated traffic. . . . .	29
4.3	CAN frame setting the installation height to 5 meters. . . . .	29
4.4	Frame identifiers for tracked objects. . . . .	30



# Introduction

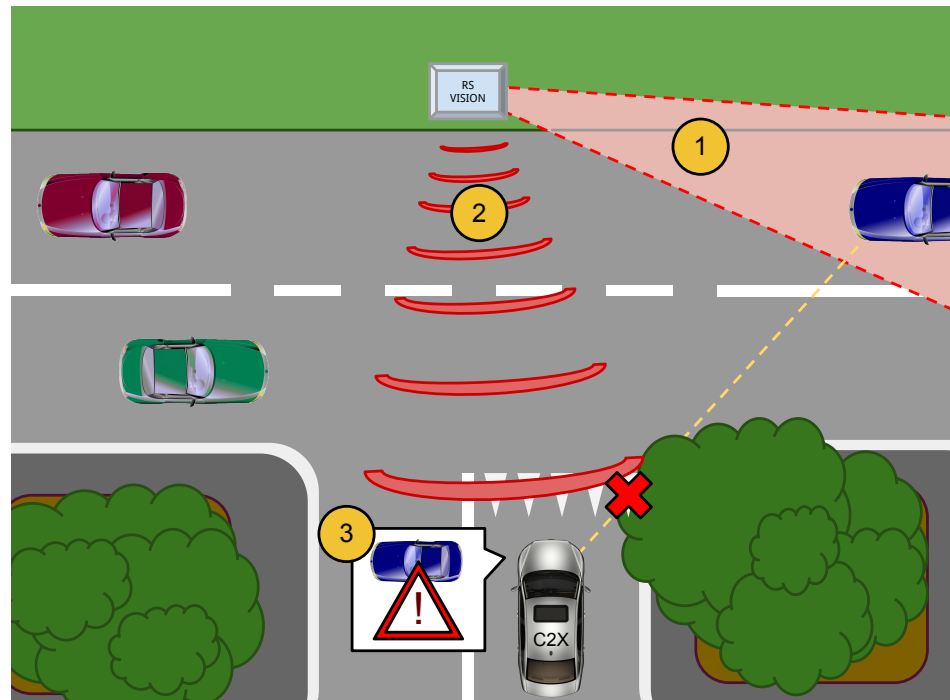
---

## 1.1 Background

Intelligent Transportation Systems (ITS) can be defined as a group of technological solutions in telematics designed to improve the security of terrestrial transportation. Using the IEEE 802.11p standard, which adds vehicle wireless capability, possibilities of improving cooperative traffic safety are almost unlimited. One of these possibilities is to broadcast the position, speed and size of each individual vehicle so other nearby vehicles can collect this information and use it for safety purposes.

The problem that this technology will face comes during the implementation phase, as just a few of the vehicles in circulation will actually be equipped with onboard devices. This project focuses on addressing this problem by implementing a static device, called *Drive ITS*, that scans for vehicles and emulates them as if they had their own 802.11p onboard transmitter modules. This has been done using another 802.11p wireless module in conjunction with a radar.

In the example portrayed in Figure 1.1, the intelligent car (grey in the example) reaches an intersection where there is known risk of colliding with other vehicles due to the low visibility of the cars that approach from the east.



**Figure 1.1:** Example of the Drive ITS system.

If all the cars in the example had a car-to-car (C2X) communication system, they would broadcast their position and speed and therefore the vehicle that tries to access the main road would be aware of the risk of impact. Nevertheless, a more realistic scenario would be that where the majority of the vehicles are not equipped with this system.

Drive ITS, the device that was developed in the present master's thesis, can be placed in the intersection, monitoring all the vehicles that approach from the east and broadcasting their position and speed as if they had their own onboard devices.

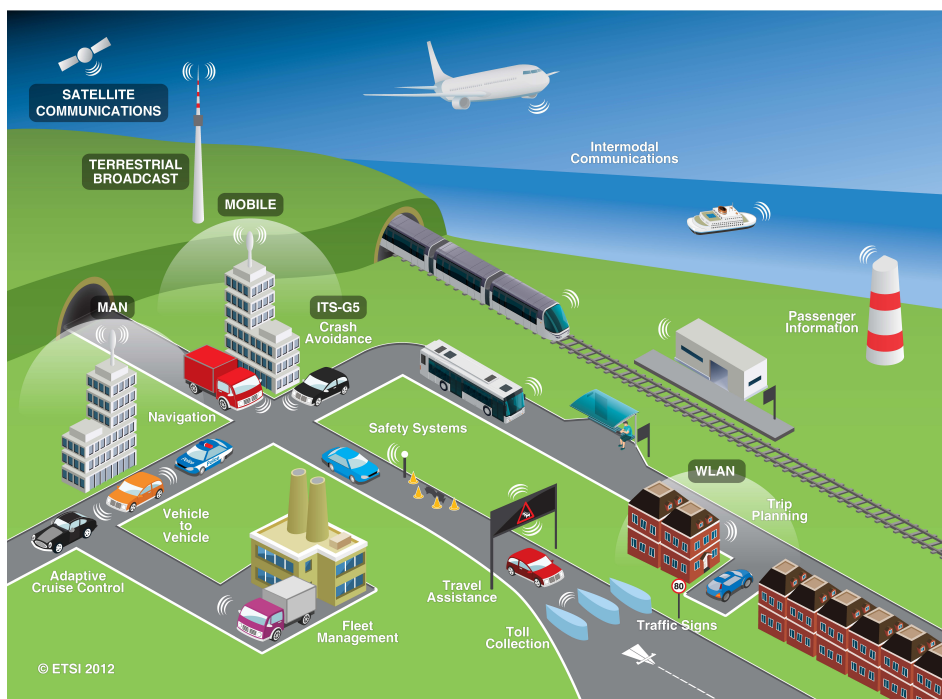
In the example, the blue car is not equipped with a C2X system and is driving at a high speed. The driver in the grey car, equipped with C2X, is not capable of detecting it since there are some bushes blocking the line of sight. Since Drive ITS is placed in the intersection, it will see the blue car (1) and will transmit instant values of its position and speed to the grey car (2). The C2X vehicle will then be notified about the approaching car and will alert the driver of the risk of collision if he decides to enter the main road (3).



## 1.2 State of the art

Intelligent Transportation Systems englobe a wide range of applications, such as car navigation, traffic signal control systems, toll collection, container management systems, variable message signs, automatic number plate recognition, efficient evacuation systems, advanced speed limiting, weather information and cooperative vehicle communication.

Figure 1.2 depicts these applications.



**Figure 1.2:** Applications within the ITS framework<sup>1</sup>.

The last application is the one targeted in the present master's thesis. The current status of cooperative communication between vehicles is still a work in progress, with protocol specifications and wireless technologies not completely defined. Nevertheless, the wireless standard that is dominating this field for short-range communications (less than 450 m) is accomplished using 802.11p/WAVE, which is an approved amendment to the IEEE 802.11 standard to add wireless access in vehicular environments (WAVE).

<sup>1</sup>Permission for use of image given by the European Telecommunications Standards Institute (ETSI). Source: <http://www.etsi.org/technologies-clusters/technologies/intelligent-transport>

In Europe, 802.11p is used as a basis for the ITS-G5[1] standard, supporting the Geonetworking protocol for vehicle to vehicle / vehicle to infrastructure communications. ITS-G5 and Geonetworking is being standardized by ETSI ITS<sup>2</sup>.

Cooperative systems on the road can include all possibilities between cars and infrastructures. This is used to create a network where vehicles act as mobile sensors, that transmit localized data such as weather conditions, speed or frequent braking activities, among others, to central servers that process this information and transmits it back to other vehicles, but also between the vehicles themselves, where position and speed data of nearby objects can be decisive to prevent a collision.

---

<sup>2</sup><http://www.etsi.org/index.php/technologies-clusters/technologies/intelligent-transport>

## Objectives

---

### 2.1 Goal

The goal of this thesis is to address the problem described hereinabove by developing a device that is able to obtain data from a Universal Medium Range Radar, process it regarding the standards of car-to-car communication and broadcast it using a 802.11p modem.

The present master's thesis will be divided into two main parts, hardware and software.

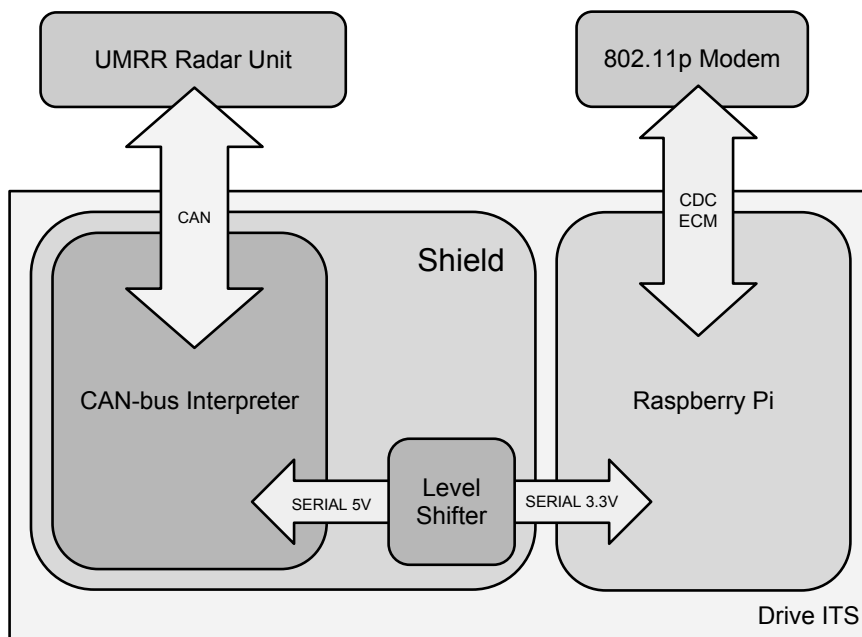
The hardware must be able to communicate with both the UMRR radar and the 802.11p modem. Since the radar was designed to be an onboard unit for vehicles, it provides a Controller Area Network (CAN) bus interface for communicating with it. CAN is a vehicle bus standard (ISO 11898) designed to allow high-speed communication between microcontrollers and devices within a vehicle without a host computer. The modem on the other hand was designed to be used together with personal computers, and therefore provides an Ethernet implementation over the USB protocol.

The software must be able to receive the stream of information provided by the UMRR radar, process it in such a way that it filters out irrelevant data and transmit it to other nearby vehicles using the 802.11p modem. Since the radar uses the CAN protocol a CAN library will have to be implemented. The software also must be able to implement the most current version of the standard ITS-G5 to encode the vehicle information data. It must also be able to communicate with the 802.11p modem using Ethernet over USB.



This chapter describes the hardware part of the present master's thesis. The device is divided into three modules, the CAN-bus interface, the microcontroller to Raspberry Pi interface, and the Ethernet over USB interface, integrated in the Raspberry Pi, which allows its connection to a 802.11p modem.

Figure 3.1 depicts these modules.

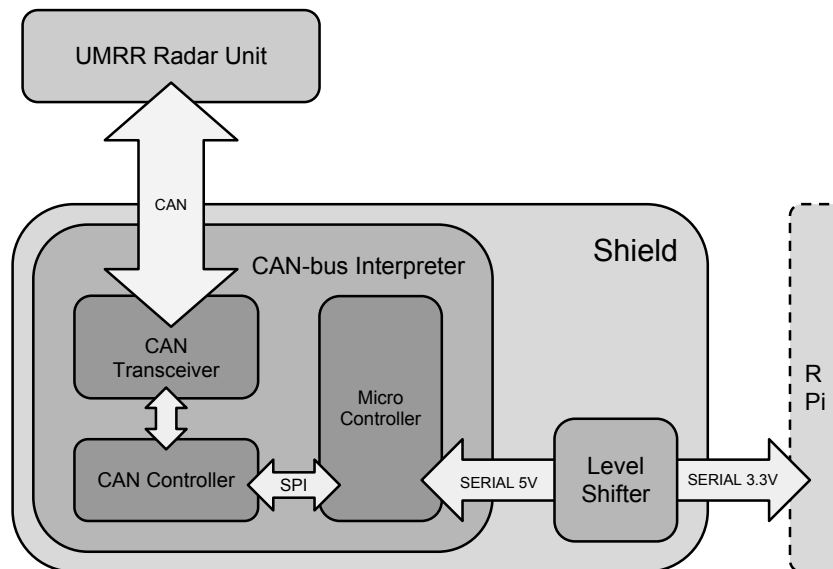


**Figure 3.1:** Block diagram of the system.

### 3.1 CAN-bus interpreter

In order to interface with the CAN-bus connection of the radar, a hardware shield for the Raspberry Pi (see Section 3.3) was designed, implemented and produced. This shield consists of a CAN transceiver (PCA82C251 or MCP2551), a CAN controller (MCP2515) and a microcontroller (AVR ATMEGA328P-PU) for programmability and serial communications with the Raspberry Pi, featuring a transceiver that acts as a voltage level shifter.

Figure 3.2 portrays the disposition of the elements in the shield, which includes the CAN-bus Interpreter, subject of this section.



**Figure 3.2:** Block diagram of the shield.

#### 3.1.1 The CAN architecture

The CAN bus is a serial communication protocol with bus topology, initially intended for vehicle environments, as a means to connect the different sensors with the ECU. In CAN networks there is no addressing, master or slaves. Each node sends messages to the rest of the nodes and, based on the identifier attached to the message, each node decides to process the message or not.

Even if the initial scope of CAN was the automotive sector, it has gained popularity in other applications such as domotics, industrial networks, automation, etc.

The scope of ISO 11898 specifies the data link layer and physical layer of the communication link, along with their own sublayers. This CAN bus interface implements both layers, as shown in Figure 3.3, and adds an application layer embedded in a microcontroller.

Specification	OSI Layer		Implementation
To be specified by the system designer	Application layer		AVR ATMEGA328P-PU
CAN-protocol specification	Data Link Layer (DLL)	Logical Link Control (LLC)	CAN Controller (MCP2515)
		Medium Access control (MAC)	
		Physical Signalling	
	Physical Layer (PHY)	Physical Medium Attachment	CAN Transceiver (PCA82C251 or MCP2551)
		Medium Dependent Interface	
Scope of ISO 11898	Transmission medium		

Figure 3.3: Architecture of CAN [2].

The bus topology is portrayed in Figure 3.4, with an example of a possible Drive ITS setup.

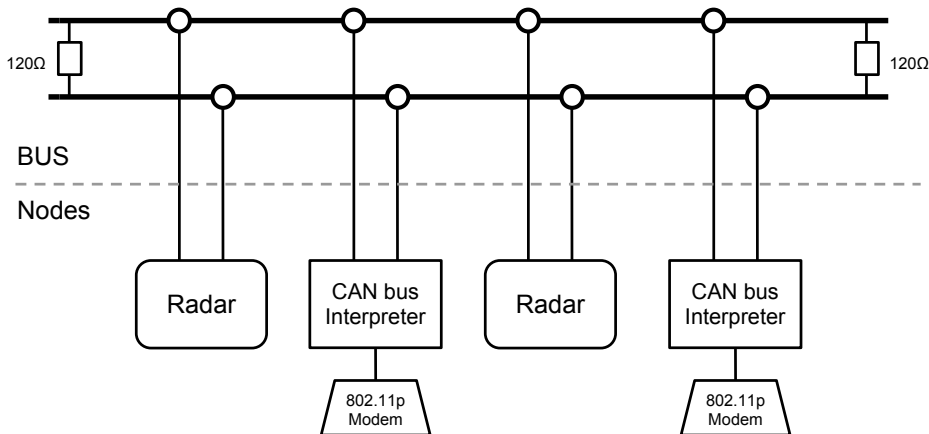


Figure 3.4: Drive ITS CAN bus topology.

### 3.1.2 The CAN transceiver

The PCA82C251 transceiver provides an interface between the physical transmission line and a protocol controller. It is capable of transmitting and receiving data with a bit rate of up to 1Mbit/s over a two-wire differential voltage bus line. A dominant bit will occur when the CAN\_H wire takes 3.5V and CAN\_L takes 1.5V. A recessive bit will be read if both wires take 2.5V.

#### 3.1.2.1 Operating modes

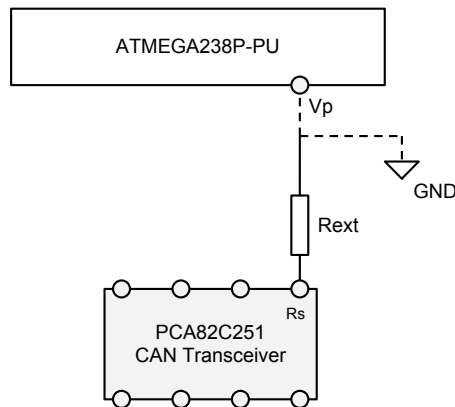
This transceiver can operate in three modes:

**High-Speed Mode** is suitable to achieve a maximum bit rate and/or bus length.

**Slope Control Mode** provides compatibility with unshielded bus cables by decreasing the slew rate of the bus signal in order to reduce electromagnetic emission.

**Stand-By Mode** drastically reduces the system's power consumption.

The transceiver operating mode is selected via the Rs pin, using an external resistor connected either to ground or a pin in the microcontroller. Only in this last case it will be possible to select the stand-by mode. See figure 3.5.



**Figure 3.5:** Possible connections of the Rs pin [2].

The values of  $R_{ext}$ , together with the state of the microcontroller pin determine the selected mode.



For  $V_P = V_{CC}$ , stand-by mode is selected no matter what value  $R_{ext}$  assumes. For  $V_P = 0V$ , high-speed mode is selected if  $R_{ext} \in (0, 1.8)k\Omega$  while the transceiver will operate in slope control mode if  $R_{ext} \in (16.5, 140)k\Omega$ .

In this setup, a conservative value of  $R_{ext}$  was selected by connecting a  $4.7k\Omega$  resistor from Rs to ground, somewhere in between the high-speed mode and the slope control mode. The drawback is the loss of the stand-by functionality, which will be considered in future implementations, as described in section 7.2.

### 3.1.2.2 Limitations

The maximum bus length in a CAN bus is determined by three physical effects [2]:

- Loop delays of the connected bus nodes, such as the CAN transceiver and the CAN controller and the delay of the bus line.
- The differences in bit time quanta length due to the relative oscillator tolerance between nodes.
- The signal amplitude drop due to the series resistance of the bus cable and the input resistance of bus nodes.

For the selected bit rate (250kbaud), the maximum bus length that can be achieved with the CAN transceiver is 250m, as depicted in the Bit Rate/Bus Length Relation table of the application note [2].

The maximum number of nodes per bus that the PCA82C251 is capable of driving is as much as 100 [2]. Therefore, being such a big figure, it can be concluded that the CAN transceiver does not limit the number of radars connected to the solution targeted in this report.

### 3.1.3 The CAN controller

The MCP2515 CAN controller implements the CAN specification as in figure 3.3. It is capable of transmitting and receiving both standard and extended data, as well as remote frames.

It has two acceptance masks and six acceptance filters, which are used to filter out unwanted messages from the nodes in the bus, thus reducing the host microcontroller overhead.

The CAN controller interfaces with the microcontroller unit via Serial Peripheral Interface (SPI).

The device is divided in three main blocks; the CAN module handles all functions for receiving and transmitting messages on the CAN bus (CAN protocol engine, masks, filters and TX-RX buffers), the control logic which configures the device and its operation, and the SPI port which interfaces with the microcontroller unit.

### 3.1.3.1 Operating modes

The MCP2515 has five modes of operation:

**Configuration mode** which is used to initialize the device and modify the configuration registers and the filter and mask registers.

**Sleep mode** which minimizes current consumption of the device, providing automatic wake-up upon detecting activity on the bus, and in  $8\mu\text{s}$  (128 times the oscillation period) the communication is resumed in listen-only mode.

**Listen-only mode** which provides means for the MCP2515 to receive all messages transmitted in the bus, including messages with errors. It features auto-baud detection, thus making a perfect operating mode for a CAN bus eavesdropper. No messages are transmitted in this state, not even acknowledge signals or error flags.

**Loopback mode** which allows internal transmission of messages from the transmit buffers to the receive buffers. It is used for testing purposes.

**Normal mode** which is the standard operating mode, where the MCP2515 monitors all bus messages generating acknowledge bits and error frames and enables transmission capabilities.

The operation mode is selected by writing to the CAN control register (*CANCTRL*), which will be further described later in this document.

### 3.1.3.2 Bit timing settings

The documentation of the UMRR Traffic Management Sensor [15], specifies the data communication settings regarding the CAN bit timing and baud rate.

- CAN bit rate: 250 kBaud
- Phase Segment 1: 8 TQ
- Phase Segment 2: 7 TQ
- Synchronization Jump Width: 4 TQ

Each of the segments that make up a bit time are divided in integer units called Time Quanta ( $TQ$ ), which is based on the oscillator period ( $T_{OSC}$ ) and the Baud Rate Prescaler ( $BRP$ ) which is programmable from 1 to 64 to satisfy the demanded baud rate.

$$TQ = 2 \cdot BRP \cdot T_{OSC} \quad (3.1)$$

The Nominal Bit Rate ( $NBR$ ) is defined as the number of bits per second transmitted by an ideal transmitter with no resynchronization, as seen in equation 3.2.

$$NBR = f_{bit} = \frac{1}{T_{bit}} \quad (3.2)$$

Being  $T_{bit}$  the nominal bit time, defined as the summation of the following segments:

$$T_{bit} = T_{SyncSeg} + T_{PropSeg} + T_{PhaseSeg1} + T_{PhaseSeg2} \quad (3.3)$$

The synchronization segment ( $T_{SyncSeg}$ ) is used to synchronize the nodes on the bus. It is fixed to 1 TQ.

Propagation segment compensates for physical delays between nodes, as twice the sum of the signal's propagation time on the bus line and the delay in the bus driver. It is programmable from 1 to 8 TQ. In this case the minimum quantity of 1 TQ has been selected, since the bus is quite short and there are no significant delays expected.

Substituting equations 3.1 and 3.2 in equation 3.3 and factoring:

$$\frac{1}{250 \cdot 10^3} = (1 + 1 + 8 + 7) \cdot (2 \cdot BRP \cdot T_{OSC}) \quad (3.4)$$

The design features a 16 MHz crystal oscillator giving the following expression of  $T_{OSC}$ :

$$T_{OSC} = \frac{1}{16 \cdot 10^6} \quad (3.5)$$

When substituting  $T_{OSC}$  in equation 3.4 and solving for BRP, the following value is obtained:  $BRP = 1.882$

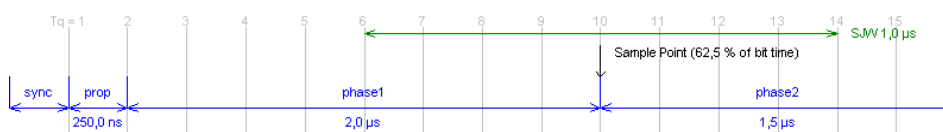
As this is an impossible value, given that  $BRP$  is in the domain of the integers, it can be concluded that achieving 250kbps with the specified phase segments is

impossible. It was opted to lower the second phase segment by one unit, thus being  $T_{PhaseSeg2} = 6 TQ$  and equation 3.4 now changes to:

$$\frac{1}{250 \cdot 10^3} = (1 + 1 + 8 + 6) \cdot (2 \cdot BRP \cdot T_{OSC}) \quad (3.6)$$

Solving equation 3.6 for BRP, a valid value was found:  $BRP = 2$

The bit timing diagram is thus as shown in figure 3.6



**Figure 3.6:** Bit timing diagram [4].

Having fixed these parameters, now the associated programmable registers have to be written to reflect the desired bit timing configuration.

These registers are CNF1, CNF2 and CNF3, accessible through their respective addresses; 0x2A, 0x29 and 0x28.

The next section will describe these configuration registers and their values.

### 3.1.3.3 Configuration and control registers

The MCP2515 provides multiple operation modes, bit timing possibilities and other configuration parameters that have to be set within some registers. In this section the most critical configuration registers will be described in order to set their values for the final product.

- CNF1 (address 0x2A)
- CNF2 (address 0x29)
- CNF3 (address 0x28)
- CANCTRL
- TXRTSCTRL

CNF1, CNF2 and CNF3 store the bit time settings as well as other miscellaneous parameters.

In section 3.1.3.2 the following parameters were determined:

- $BRP_{REAL} = 2$
- $PHSEG1_{REAL} = 8$
- $PHSEG2_{REAL} = 6$
- $PROPSEG_{REAL} = 1$
- $SJW_{REAL} = 4$

Note that for BRP, SJW, Phase and Propagation Segments the following rule applies:

$$Value_{REG} = Value_{REAL} - 1 \quad (3.7)$$

Where  $Value_{REG}$  is the value stored in the register (in binary) and  $Value_{REAL}$  is the real value of the segment.

Tables 3.1, 3.2 and 3.3 show the configuration registers CNF1, CNF2 and CNF3 respectively.

**Table 3.1:** Register CNF1 overview.

Element	SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0
Value	1	1	0	0	0	0	0	1

**Table 3.2:** Register CNF2 overview.

Element	BTL MODE	SAM	PHSEG12	PHSEG11	PHSEG10	PRSEG2	PRSEG1	PRSEG0
Value	1	0	1	1	1	0	0	0

**Table 3.3:** Register CNF3 overview.

Element	SOF	WAKFIL	-	-	-	PHSEG22	PHSEG21	PHSEG20
Value	0	0	0	0	0	1	0	1

As well as fixing these parameters to these values, the configuration registers set four other parameters:

**BTLMODE** phase segment 2 bit time length mode.

- 1 = Length of PhSeg2 determined by PHSEG22:PHSEG20 bits of CNF3.
- 0 = Length of PhSeg2 is the greater of PhSeg1 and 2 TQ.

**SAM** Sample Point Configuration. The sample point takes place in the end of the phase segment 1 and is the point in the bit time where the logic level is read and interpreted.

- 1 = Bus line sampled three times at the sample point, determining the value of the bit by a majority decision.
- 0 = Bus line sampled once at the sample point.

**SOF** Start-of-Frame signal. It depends on the CANCTRL.CLKEN bit. If this bit is set to 0, the CLKOUT/SOF pin is in a high-impedance state. If it is set to 1, then the SOF bit determines the operation of the CLKOUT/SOF pin.

- 1 = SOF signal is generated on the CLKOUT/SOF pin at the beginning of each CAN message detected on the RXCAN pin.
- 0 = the CLKOUT/SOF pin transmits the system clock, allowing subsequent devices to get this signal from the MCP2515.

**WAKFIL** The MCP2515 can be programmed to apply a low-pass filter function to the RXCAN pin while in sleep mode, preventing the device to wake up due to short glitches in the bus. CNF3.WAKFIL enables or disables this filter.

- 1 = Filter enabled
- 0 = Filter disabled

The CANCTRL register (with address 0xXF) controls the basic functionality of the MCP2515. Table 3.4 depicts the register's elements, which are defined as following:

**REQOP** request operation mode.

- 000 = Normal operation mode
- 001 = Sleep mode
- 010 = Loopback mode
- 011 = Listen only mode
- 100 = Configuration mode
- 111 = Power-up default value, which has to be changed in order to operate the device.

**ABAT** Abort All Pending Transmissions

- 1 = Request abort of all pending TX buffers.
- 0 = Terminate request to abort all transmissions.

**OSM** One-Shot mode.

- 1 = The message will only be sent once.
- 0 = If requested, the message will be transmitted again.

**CLKEN** CLKOUT Pin enable.

- 1 = Pin enabled for CLKOUT/SOF functionality, as defined by the CNF3.SOF bit.
- 0 = Pin set to high-impedance state.

#### CLKPRE CLKOUT Pin Prescaler

- 00 = System clock / 1
- 01 = System clock / 2
- 10 = System clock / 4
- 11 = System clock / 8

**Table 3.4:** Register CANCTRL overview.

Element	REQOP2	REQOP1	REQOP0	ABAT	OSM	CLKEN	CLKPRE1	CLKPRE0
Value	0	0	0	0	0	0	0	0

TXRTSCTRL, with address 0x0D, serves both as a control register and as a configuration register for the TXRTS pins. A detailed overview can be found in table 3.5.

**Table 3.5:** Register TXRTSCTRL overview.

Element	-	-	B2RTS	B1RTS	B0RTS	B2RSTM	B1RTSM	B0RTSM
Value	0	0	0	0	0	0	0	0

The bits corresponding to BxRTS are read-only and provide the state of the TXRTS pins when enabled as digital inputs.

The writable bits BxRTSM configure the TXRTS pins to work either as request-to-send inputs (1), providing an alternative means of initiating the transmission of a message, or as standard digital inputs (0).

#### 3.1.3.4 SPI interface

The MCP2515 features Serial Peripheral Interface, a very common port featured in a wide range of microcontrollers, such as the one included in the ATMEGA328 that this system incorporates.

The commands and data are sent to the MCP2515 via the MOSI pin, being the data clocked on the falling edge of the system clock, while all the information that is requested from the MCP2515 (typically messages on the CAN bus) is driven out via the MISO pin, on the falling edge of the system clock. Every time an operation is performed, the Chip Select pin must be held low.

The SPI commands used in the software are described in Table 3.6

**Table 3.6:** Description of SPI commands [3].

Instruction Name	(BIN)	(HEX)	Description
RESET	1100 0000	0xC0	Resets internal registers to default state and sets configuration mode.
READ	0000 0011	0x03	Reads data from register beginning at the selected address.
WRITE	0000 0010	0x02	Writes data to register beginning at selected address.
READ_RXBUF0_ID	1001 0000	0x90	Reads the state of the RX buffer 0.
READ_RXBUF1_ID	1001 0100	0x94	Reads the state of the RX buffer 1.
READ_RXBUF0_DATA	1001 0010	0x92	Reads the data from the RX buffer 0
READ_RXBUF1_DATA	1001 0110	0x96	Reads the data from the RX buffer 1
LOAD_TXBUF0_ID	0100 0000	0x40	Loads the state into the TX buffer 0.
LOAD_TXBUF1_ID	0100 0010	0x42	Loads the state into the TX buffer 1.
LOAD_TXBUF2_ID	0100 0100	0x44	Loads the state into the TX buffer 2.
LOAD_TXBUF0_DATA	0100 0001	0x41	Loads the data into the TX buffer 0.
LOAD_TXBUF1_DATA	0100 0011	0x43	Loads the data into the TX buffer 1.
LOAD_TXBUF2_DATA	0100 0101	0x45	Loads the data into the TX buffer 2.
RTS_TXBUF_	1000 0xyz	0x8_	SPI Commands to send the content in buffers 0, 1 and 2. If x, y or z take the value 1, the data in the corresponding buffer will be transmitted. Only one buffer can be transmitted at a time.
READ_STATUS	1010 0000	0xA0	Quick polling command that reads several status bits for transmit and receive functions.
RX_STATUS	1011 0000	0xB0	Quick polling command that indicates filter match and message type (standard, extended and/or remote) of received message.
BIT_MODIFY	0000 0101	0x05	Allows the user to set or clear individual bits in a particular register.



### 3.1.4 The Microcontroller Unit (MCU)

As a first approach, the first prototypes were based on the Arduino platform, more specifically Arduino UNO R3, which provided a simple way to get started without worrying about difficult setups or faulty connections.

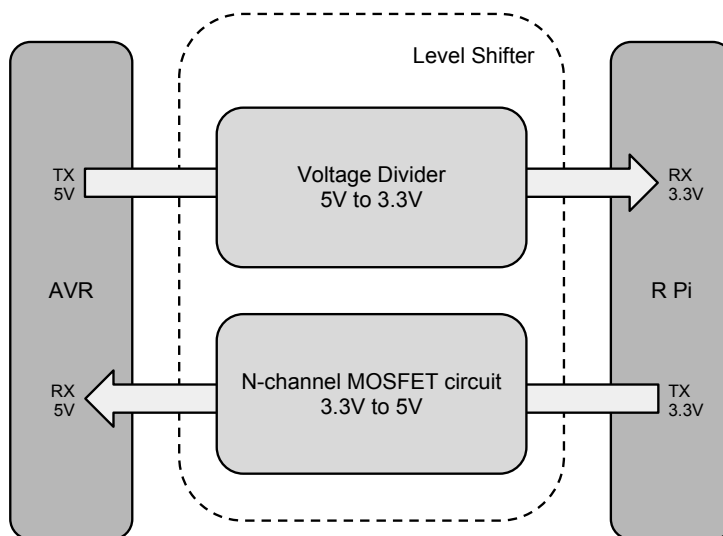
This board features an AVR ATMEGA328P-PU microcontroller unit, which was kept until the final version of the system.

## 3.2 Microcontroller to Raspberry Pi interface

The communication between the AVR microcontroller and the Raspberry Pi is achieved with the serial port included on both devices. In a normal setup, this would require to connect the TX of the AVR with the RX in the Raspberry Pi and vice versa, while in this case a transceiver is needed.

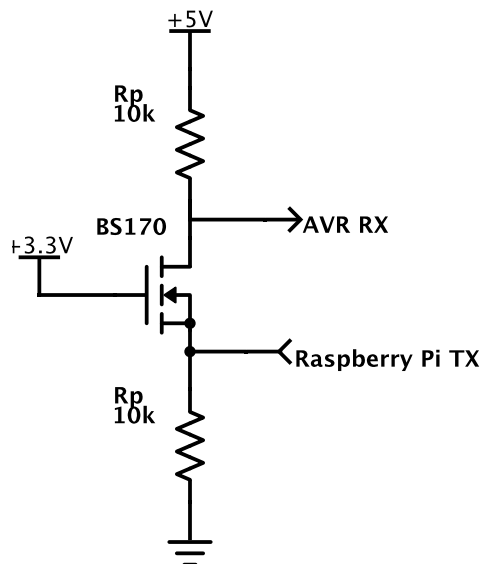
Since the shield's components are powered with 5V while the Raspberry Pi's use just 3.3V, the logic levels in the serial communication are also different. Therefore, there is a need for an intermediate device that acts as an interpreter between the AVR and the Raspberry Pi, avoiding possible malfunctioning because of overpowering the Raspberry Pi with 5V, which could harm the device.

This simple transceiver is based on a voltage divider for the *AVR* → *Raspberry Pi* line and a N-channel MOSFET level shifter circuit for the *Raspberry Pi* → *AVR* line, as depicted in Figure 3.7.



**Figure 3.7:** Block diagram of the level shifter.

The N-channel MOSFET circuit can be found in Figure 3.8, which is obtained from Herman Shutte's work on Bi-Directional Level Shifters [8]. It allows the 3.3V TX pin from the Raspberry Pi to raise its level to the AVR RX pin's 5V.



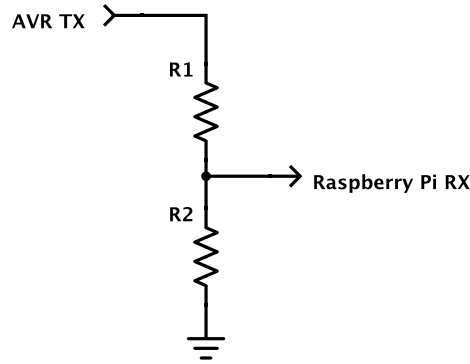
**Figure 3.8:** N-FET 3.3V → 5V circuit.

In the hereinabove mentioned document, Shutte explains the three states that can be considered, which are adapted to this particular solution:

- State 1** The TX line of the Raspberry Pi is pulled up by its pull-up resistors  $R_p$  to 3.3 V. The gate and the source of the MOS-FET are both at 3.3 V, so its  $V_{GS}$  is below the threshold voltage and the MOS-FET is not conducting. This allows that the RX line at the AVR side is pulled up by its pull-up resistor  $R_p$  to 5V. So the lines of both sides are HIGH, but at a different voltage level.
- State 2** A 3.3 V device pulls down the Raspberry Pi TX line to a LOW level. The source of the MOS-FET becomes also LOW, while the gate stays at 3.3 V. The  $V_{GS}$  rises above the threshold and the MOS-FET becomes conducting. Now the bus line of the AVR RX side is also pulled down to a LOW level by the 3.3 V device via the conducting MOS-FET. So the lines of both sections become LOW at the same voltage level.
- State 3** (*Not applicable*) A 5 V device pulls down the AVR TX line to a LOW level. Via the drain-substrate diode of the MOSFET the *lower voltage* section is in first instance pulled down until  $V_{GS}$  passes the threshold and the MOS-FET becomes conducting. Now the TX line of the Raspberry Pi

section is further pulled down to a LOW level by the 5 V device via the conducting MOS-FET. So the lines of both sections become LOW at the same voltage level. Since it is not operating on a bi-directional line, typical from bus topologies (such as I<sup>2</sup>C or SPI), the circuit will never be in this third state.

The other line, *AVR* → *Raspberry Pi* is much easier since it is based on lowering the voltage. This can be easily achieved via a voltage divider, as depicted in Figure 3.9.



**Figure 3.9:** Voltage divider to perform the 5V → 3.3V conversion.

Resistor values are chosen from the voltage divider's formula.

$$V_{RPi_{RX}} = V_{AVR_{TX}} \cdot \frac{R_2}{R_1 + R_2} \quad (3.8)$$

Substituting the values of  $V_{RPi_{RX}}$  and  $V_{AVR_{TX}}$  in HIGH level state and solving for  $R_1/R_2$ :

$$\frac{R_2}{R_1 + R_2} = \frac{3.3}{5} = 0.66 \quad (3.9)$$

Therefore, if the values of  $R_1 = 10k\Omega$  and  $R_2 = 20k\Omega$  are assigned, equation 3.9 is satisfied.

### 3.3 Ethernet over USB interface

The specification of the present master's thesis required the usage of a 802.11p modem which was originally designed to be connected to a personal computer without the needs of manufacturer's drivers.

Therefore, the modem manufacturer decided to incorporate a Ethernet over USB connectivity as the only interface in their device. This means that it uses the physical layer of USB, with TCP/IP layers on top. If it is plugged to a Ethernet over USB host, such as a personal computer, both devices will be connected and ready to communicate.

Since there are no available solutions for Ethernet over USB hosting on AVR microcontrollers, it was opted to use a Raspberry Pi, that is capable of running this protocol out of the box.

### 3.4 Programmability

The embedded microcontroller, an AVR ATMEGA328P-PU, is programmed in the Arduino Development Environment.

It features a program that receives, filters and forwards the data received from the radar and sends it to the Raspberry Pi, which will be further described in Chapter 4.

To control the CAN-bus interpreter, this program uses a CAN library which is based on the open-source CANduino project [7]. This library has been modified to write the previously mentioned registers in Section 3.1.3 with the required values. The program also uses the in-built SPI library from Arduino to access the MCP2515 via this communication interface.

In order to communicate with the Raspberry Pi, the Arduino Development Environment also features a serial communication library that simplified the task of data forwarding.

### 3.5 Prototypes

Up to the completion of the present master's thesis, three prototypes of the system were built, each of them incorporating new functionalities.

The desire of the authors was to achieve a final prototype that was sturdy and provide a neat feeling, all in a single package that could be easily assembled and held without the danger of damaging its elements.

### 3.5.1 Breadboard prototype

The first approach to this project was to use the Arduino platform to interface between a CAN-bus interpreter and a Ethernet over USB communication module, which could be part of the Arduino board itself, since it already features an USB port onboard.

Therefore, the microcontroller would take the following tasks:

- SPI interface with CAN-bus interpreter
- Process all information received through CAN-bus
  - Extract the position, speed and size of the tracked objects.
  - Calculate the CAM parameters as specified by the ETSI ongoing standards.
  - Build the CAM messages
- Encapsulate the CAM messages into Ethernet frames.
- Interface with the Ethernet over USB communication module.

Nevertheless, the Ethernet over USB interface was very difficult to achieve since it would demand to develop the AVR driver for it. Since the protocol is quite hermetic and given that this task would be very time-consuming, it was opted to find a platform that natively supports Ethernet over USB. As a result of it, the second prototype was developed to interface with a Raspberry Pi. The decision was taken considering that the main task was developing a fully working product at the end of the thesis, no matter which platform was used.

This left the breadboard prototype as a means to develop and test the CAN-bus interface and to further expand the available drivers for similar CAN-bus solutions already developed.<sup>1</sup>

Figure 3.10 depicts the first design of the CAN-bus interpreter connected to the Arduino UNO R3 board.

---

<sup>1</sup>The CAN-bus interpreter was based on the models available in ModelRail[5], SparkFun[6] and libraries available in the CANduino project[7].

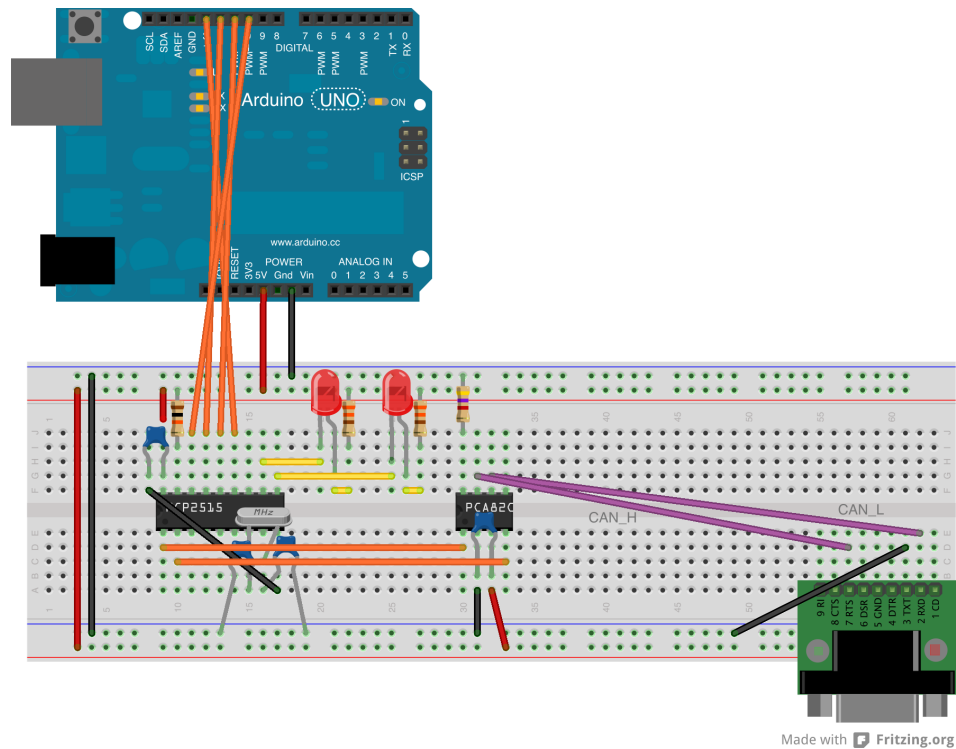


Figure 3.10: Breadboard prototype.

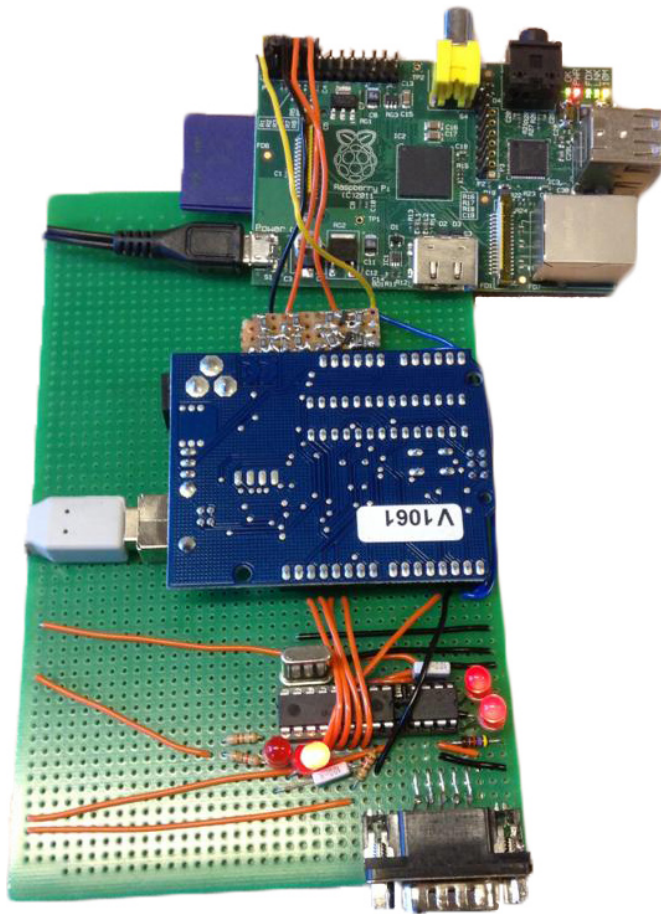
### 3.5.2 Perfboard prototype

Once the CAN-bus interpreter was fully developed on a breadboard, it was decided to move it into a robust platform which involved soldering the components into a bakelite perfboard. It also provided a dock for the Arduino UNO R3 board, so it could be easily mounted and extracted without the need of a cable harness.

This prototype features an interface between the Arduino board and a Raspberry Pi via their serial port (RS-232). In order to get both devices to communicate, a transceiver was developed to shift the 5V level of the Arduino into the 3.3V level of the Raspberry Pi and vice versa.

This device was primarily used to test the communication with the Raspberry Pi and to develop the filtering and formatting of the object information tracked by the radar in order to avoid an overload of data in any point of the system.

Figure 3.11 depicts the perfboard prototype, the first featuring all the elements that would be included in the final product.



**Figure 3.11:** Perfboard prototype.

### 3.5.3 PCB prototype

Once all communication was achieved and all further development was only software-based, the next step into a real-life solution was the production of a printed circuit board (PCB) where the components could be soldered with no need for tin tracks or cables.

The PCB, entirely designed in the CAD software EAGLE, focused on creating a neat package out of the assembly of both the PCB and the Raspberry Pi. Hence, it was decided to mount one on top of the other, by connecting them through a 26-pin header, which serves as the power supply for the PCB components, getting powered via the Raspberry Pi, but also as the serial communication interface.

In this prototype, the Arduino platform was dismissed, stripping out all unnecessary components, just leaving the AVR microcontroller, the external oscillator, a

protecting circuit and a programming port. In this way, the component cost was notably reduced, while preserving all the functionality of the previous prototype.

The programming port allows an AVR dragon device to program the device just by flipping a switch and dumping the code into the microcontroller.

An overview of the main elements of this PCB is depicted on Figure 3.12.

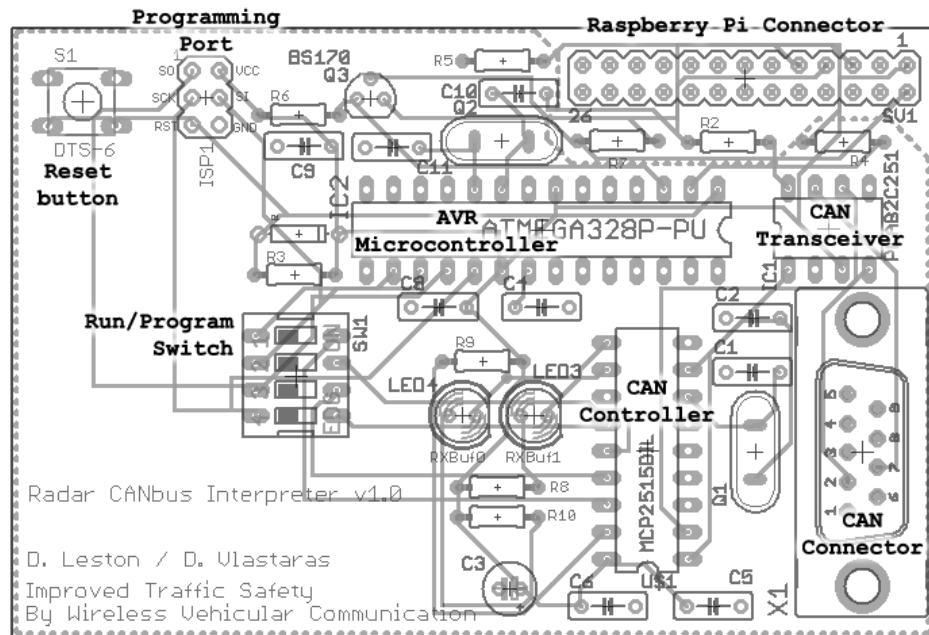


Figure 3.12: PCB elements.

At the moment of publishing the present master's thesis this was the final version of the Drive ITS system, and as such it can be found in Section 6.2, dedicated to the final product.



The software running on the product is divided in two main parts. The first part is the code running on the AVR microcontroller, which is written in C++ taking advantage of the Arduino SDK for SPI and Serial (RS-232) communications. The second part is the software running on the Raspberry Pi. That part of the code is written in Java and this is where the main algorithms are running.

In this chapter it will be explained further how each part of the code works and how the system is put together as one following the flow of data from a software point of view.

## 4.1 Reading data from the radar

In order for the radar to start scanning for vehicles and to provide the system with data it has to be initiated with certain commands. After it has been initiated the received data needs to be forwarded to the rest of the system. The CAN-bus shield is responsible for these tasks. The Raspberry Pi is responsible for running the rest of the system.

### 4.1.1 CAN-bus shield

#### 4.1.1.1 RS-232 baud rate

When the system is powered up the initiation procedure has to take place. The first thing to happen is setting the baud rate for the RS-232 port. It has been observed that the radar never sends more than 170 CAN frames per second onto the CAN bus. Also each CAN frame forwarded onto the serial port contains 22

bytes (see section 4.1.1.4) but since serial packets have  $1 + 1 = 2$  bits for header and trailer,  $8 + 2 = 10$  bits need to travel through serial for each byte from the CAN bus in the worst case. From that the required speed for RS-232 has been calculated as following:

$$\frac{170 \text{ frames}}{s} \cdot \frac{22 \text{ bytes}}{\text{frame}} \cdot \frac{8 \text{ bits} + 2 \text{ bits}}{\text{byte}} = 37400 \text{ bits/s} = 37400 \text{ baud} \quad (4.1)$$

The closest RS-232 configuration to that baudrate is 38400 baud. However since 37400 baud is very close to 38400 baud and the serial port should not be overloaded in case of incoming spikes, the baud rate selected is the next available one at 57600 baud. It is being set in code using `Serial.begin(57600)` in `setup()` function.

#### 4.1.1.2 Bit timing settings

Next the CAN controller needs to be set to the correct bit timing settings as described in section 3.1.3.2. That procedure is taking place in function `CANClass::baudConfig()` of `CAN.cpp`<sup>1</sup>. The registers are getting set according to tables 3.1, 3.2 and 3.3 as following:

```
setRegister(CNF0, 0xC1);
setRegister(CNF1, 0xB8);
setRegister(CNF2, 0x05);
```

#### 4.1.1.3 Radar initiation

Now that there is a working CAN configuration, communication with the radar is possible. First frames to be sent to the radar are initiation frames which set the operation mode and the installation height of the radar. However, since this information has to come from the configuration file on the Raspberry Pi, a byte has to be read from serial port of the Raspberry Pi first. The byte that arrives is being received using the following code:

```
byte data = Serial.read();
bool simulated = (bool) (data >> 4);
byte height = (byte) (((byte)(data << 4)) >> 4);
```

<sup>1</sup>A modified version of the CANduino library. <http://code.google.com/p/candduino/>

By shifting the incoming byte 4 bits to the left, the operation mode is found. By removing the 4 leftmost bits and keeping the 4 rightmost bits of the byte, the installation height is found.

With this information given, according to the radar documentation [15] the following CAN commands need to be sent to the radar in order to set the operation mode and installation height:

**Table 4.1:** CAN frame requesting real traffic.

byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7
0xFF	0x00	0x00	0x00	0x00	0x00	0x00	0x00

**Table 4.2:** CAN frame requesting simulated traffic.

byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7
0xFF	0x00	0x00	0x00	0x08	0x00	0x00	0x00

**Table 4.3:** CAN frame setting the installation height to 5 meters.

byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7
0xFF	0x01	0x00	0xAC	0x05	0x00	0x00	0x00

In tables 4.1 and 4.2 it can be seen that byte 4 defines the operation mode, 0x00 defines normal operation mode and 0x08 defines simulated operation mode. In table byte 4 defines the installation height. According to [15] installation height can be 1m (0x01) minimum and 10m (0x0A) maximum.

Also byte 0 needs to be set to 0xFF for the receivers to be all the available radars on the CAN bus and the CAN frame ID needs to be set to 0x3F2 so that the radar recognizes the frame as a command [15]. Either one of those configurations can be sent to any radar on the bus as following:

```
byte opmode_data[] = { 0xFF,0x00,0x00,0x00,0x00,0x00,0x00,0x00 };
byte height_data[] = { 0xFF,0x01,0x00,0xAC,height,0x00,0x00,0x00 };
if (simulated) opmode_data[4] = 0x08;
uint32_t frame_id = 0x3F2;
byte length = 8;
CAN.load_ff_0(length, &frame_id, opmode_data, false);
CAN.load_ff_0(length, &frame_id, height_data, false);
```

#### 4.1.1.4 Receiving, filtering and forwarding data

Once a stream of data from the radar has begun only the tracked object data should be kept and forwarded to the rest of the system. In this subsection it is explained how this is achieved.

Traffic coming from the CAN bus resides within the first or second RX buffer of the CAN controller depending on whether the first RX buffer is full or not. Which RX buffer the data resides on is determined as following:

```
bool buffer0 = CAN.buffer0DataWaiting();
bool buffer1 = CAN.buffer1DataWaiting();
```

And the frame data is read as following:

```
if (buffer0)
{
    CAN.readDATA_ff_0(&length,frame_data,&frame_id,&extended,&filter);
}
else if (buffer1)
{
    CAN.readDATA_ff_1(&length,frame_data,&frame_id,&extended,&filter);
}
```

Where `length` is the length of the CAN frame, `frame_data` is the CAN frame data and `frame_id` is the CAN frame identifier. The identifier is what defines the kind of a received CAN frame. For tracked objects the identifiers are defined as following [15]:

**Table 4.4:** Frame identifiers for tracked objects.

Object 0	ID0 0x510	ID1 0x590	ID2 0x610	ID3 0x690
Object 1	ID0 0x511	ID1 0x591	ID2 0x611	ID3 0x691
⋮	⋮	⋮	⋮	⋮
Object 63	ID0 0x54F	ID1 0x5CF	ID2 0x64F	ID3 0x6CF

Using table 4.4 it is possible to filter out relevant CAN frames by looking at the value of `frame_id`. However, each CAN frame originating from the radar contains 8 bytes of encoded data. This data needs to be decoded and encoded again in a way friendly to the serial port before forwarding it to the Raspberry Pi. Decoding the data requires the use of some bit shifting and it is done the following way:

```

byte object_id = frame_data[7] >> 2;
byte object_length = frame_data[7] << 14 >> 8 | frame_data[6] >> 2;
int16_t y_velocity = frame_data[6] << 14 >> 5 | frame_data[5] << 1
    | frame_data[4] >> 7;
int16_t x_velocity = frame_data[4] << 9 >> 5 | frame_data[3] >> 4;
y_velocity = y_velocity < 0 ? -(1024 + y_velocity) : 1024 - y_velocity;
x_velocity = x_velocity < 0 ? -(1024 + x_velocity) : 1024 - x_velocity;
int16_t y_position = (frame_data[3] << 12 >> 2 | frame_data[2] << 2
    | frame_data[1] >> 6) + 8192;
int16_t x_position = (frame_data[1] << 10 >> 2 | frame_data[0]) + 8192;

```

This data gets then re-encoded into an ASCII string representing hexadecimal values in order to be sent onto the serial port. That string always starts with the character G, it contains 20 bytes of payload characters ranging from 0 to F and it ends with a character ranging from H to Z. The last character is a checksum verifying that the payload has been transmitted correctly.

The following piece of code takes care of encoding and transmitting the serial frame as well as computing its checksum:

```

#define FRAME_LENGTH_WITHOUT_CHS 21

char checksum(const char * str)
{
    uint8_t sum = 0;
    for (uint8_t i = 0; i < FRAME_LENGTH_WITHOUT_CHS; i++) {
        sum += (uint8_t) str[i];
    }
    return 'H' + sum % 16;
}

char frame[FRAME_LENGTH_WITHOUT_CHS];
sprintf(frame, "%02X%02X%04X%04X%04X", object_id, object_length,
        x_position, y_position, x_velocity, y_velocity);
Serial.print(frame);
Serial.print(checksum(frame));

```

#### 4.1.2 Raspberry Pi

The Raspberry Pi is a credit-card sized single-board computer. In this system it runs the Raspbian<sup>2</sup> GNU/Linux distribution. The serial port in Raspbian is by default attached to a command console. In order to receive data the serial port

---

<sup>2</sup><http://www.raspbian.org>

needs to be detached so that it can be used by another process. For that to happen two files have to be edited.

From file `/boot/cmdline.txt` the following line needs to be removed:

```
console=ttyAMA0,115200 kgdboc=ttyAMA0,115200
```

Also the second line below, at the bottom of `/etc/inittab`, needs to be commented out:

```
#Spawn a getty on Raspberry Pi serial line  
#T0:23:respawn:/sbin/getty -L ttyAMA0 115200 vt100
```

As mentioned earlier the Raspberry Pi software is written in Java. In order to enable serial communications in Java the RXTX<sup>3</sup> library has been used. Figure A.4 depicts the UML diagram for the classes that use the RXTX library.

Information that arrives to the serial port is being read by the `SerialReader` thread. The thread is responsible for validating the received payload by calculating its checksum and comparing with the received checksum. The checksum computation algorithm is exactly the same as for the AVR and its implementation in Java is the following:

```
Config.FRAME_LENGTH_WITHOUT_CHS = 21;  
  
char checksum(byte[] frame) {  
    short sum = 0;  
    for (short i = 0; i < Config.FRAME_LENGTH_WITHOUT_CHS; i++) {  
        sum += (short) (frame[i] & 0xFF);  
    }  
    return (char) ('H' + sum % 16);  
}
```

After a valid frame has been received its data is being stored in a local synchronized<sup>4</sup> `MessageBuffer<byte[]>` (figure A.1) instance as an array of bytes for further interpretation.

---

<sup>3</sup>[http://rxtx.qbang.org/wiki/index.php/Main\\_Page](http://rxtx.qbang.org/wiki/index.php/Main_Page)

<sup>4</sup>Thread safe.

## 4.2 Creating a model

### 4.2.1 Radar interface and implementation

A basic Java interface, `Radar`, has been created to describe the required behaviour of any radar attached to the system and its implementation. The only requirement is for the implementation to contain a method called `getMovingObject()` that returns an object of type `MovingObject` for every object that is being detected. If there are no objects to be detected then this method should be blocking.

The current implementation of the `Radar` interface can be found in the `AVRRadar` class. Inside the `getMovingObject()` method the object raw data is being fetched from the local `MessageBuffer<byte[]>`. This data is then being parsed and a `MovingObject` is being created and returned.

### 4.2.2 MovingObject model and kinematics

The basic model of a tracked vehicle or pedestrian is described by a `MovingObject` instance. It contains the following attributes:

- The identification number of the object as provided by the radar which is a number between 0 and 63.
- The length of the object.
- The position vector of the object, which is described by the `Position` class.
- The velocity vector of the object, which is described by the `Velocity` class.
- The kind of the object; pedestrian, car, etc, which is determined from the length of the object and a definitions table that can be found in [15].

The `Position` and `Velocity` classes have more intelligent functionality than just storing a position and a speed. A `Position` can be created either by giving the absolute geographic coordinates of an object or by giving its  $x$  and  $y$  coordinates in relation to the radar. A `Velocity` can return the speed of an object in m/s or km/h and the direction of an object in degrees from the North Pole counting clockwise.

### 4.2.3 Usage of a global synchronized `MessageBuffer<T>`

The whole process of fetching data from the radar to creating a `MovingObject<T>` is running within a thread of type `RadarThread`. In order to forward the created `MovingObjects` to the rest of the system a global synchronized `MessageBuffer<MovingObject>` is being used.

The `MessageBuffer<T>` implementation is made in such a way that it is generalized. Just like an `ArrayList<T>`, a `MessageBuffer<T>` can contain any kind of object and safely communicate it to other threads without having to worry about deadlocks. That is why it can be used early communicating arrays of bytes between two threads and later communicating `MovingObjects`.

## 4.3 Creating Cooperative Awareness Messages

Cooperative Awareness Messages are those messages that are responsible for transmitting information about the size, speed, direction, origin and position of a moving object. The classes required for creating Cooperative Awareness Messages can be found in figure A.2. From now on, Cooperative Awareness Messages will be called CAM messages.

CAM messages belong to the last of the three layers required to transmit them. The two layers below are the Basic Transport Protocol (BTP) layer defined in [10] and the GeoNetworking (GN) layer defined in [9]. Also the CAM layer is defined in [11].

Those definitions have been used during the development the protocol stack. However, since all three protocols are still under development and their specifications have not yet been finalized, a reference from another user of the same technology had to be used. The packet dump illustrated in figure 4.1 has been used for that reason, it was opened in Wireshark using dissectors from AMB Consulting<sup>5</sup>.

---

<sup>5</sup><http://www.amb-consulting.com/en/#downloads>



```

⊞ Frame 1: 83 bytes on wire (664 bits), 83 bytes captured (664 bits)
⊞ Ethernet II, Src: Kapsch_00:bb:15 (00:e0:6a:00:bb:15), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
⊞ 802.1Q Virtual LAN, PRI: 0, CFI: 0, ID: 5
⊞ GeoNetworking (TSB Single Hop)
  ⊞ Common Header
    0000 .... = Version: 0
    .... 0001 = Next Header: BTP-A (1)
    0101 .... = Header Type: TSB (5)
    .... 0000 = Header Subtype: Single Hop (0)
    Reserved: 0
  ⊞ Flags: 0x00
    0000 00.. = Reserved: 0x00
    .... ..0. = Station Type: vehicle ITS station (0)
    .... ...0 = Reserved: 0x00
    Payload Length: 29
  ⊞ Traffic Class: 0x3a
    0... .... = Reserved: 0
    .011 .... = Relevance: 3
    .... 10.. = Reliability: Medium (2)
    .... ..10 = Latency: Medium (2)
    Hop Limit: 1
  ⊞ Sender Position Vector
    ⊞ GN Address: 0x169000e06a00bb15
      0... .... = Assignment: Automatic (0)
      .0.. .... = Station Type: vehicle ITS station (0)
      ...01 0... = Station Type Details: Car (2)
      .... .1.. = Station Subtype: Private (1)
      .... ..10 1001 0000 = Country code: 656
      Link-Layer Address: Kapsch_00:bb:15 (00:e0:6a:00:bb:15)
      Timestamp: 4294966296
      Latitude: 00°00'0.00"N (0)
      Longitude: 00°00'0.00"E (0)
      Speed: 0.00 m/s | 0.00 km/h (0)
      Heading: 0.0° (0)
      Altitude: 0 m (0)
      0000 .... = Timestamp Accuracy: 1 ms (0)
      .... 0000 = Position Accuracy: Unspecified (0)
      000. .... = Speed Accuracy: Unspecified (0)
      ...0 00.. = Heading Accuracy: Unspecified (0)
      .... ..00 = Altitude Accuracy: Unspecified (0)
  ⊞ Basic Transport Protocol (Type A)
    Destination Port: 2001
    Source Port: 2000
⊞ CAM
  ⊞ CamPdu
    ⊞ header
      protocolVersion: 0
      messageID: 0
      generationTime: 1341100800000 (0x1383fd63800)
    ⊞ cam
      stationID: 2130706433
      ⊞ stationCharacteristics
        ...0 .... mobileItsStation: False
        .... 0... privateItsStation: False
        .... .1.. physicalRelevantItsStation: True
      ⊞ referencePosition
        ⊞ longitude
          hemisphere: east (0)
          degree: 16434540
        ⊞ latitude
          hemisphere: north (0)
          degree: 48215126
          elevation: 1560

```

**Figure 4.1:** An 802.11p packet dump as shown in Wireshark.

Based on figure 4.1 three classes were created; GN.java, BTP.java and CAM.java. They were designed in such a way that all the required information can be provided from the already existent model. E.g. the CAM(byte messageID, long

`stationID`, `Position referencePosition`) constructor gets a `Position` object as parameter for the `referencePosition` part of the protocol. The `GN(MovingObject.Kind kind, byte[] macAddress, Position position, Velocity velocity)` constructor receives a `MovingObject.Kind`, a `Position` and a `Velocity` object for the respective parts of the protocol where this information is required.

Each class is required to define a `byte[] getBytes()` method. This method is responsible for taking the model data and returning a byte array with all the required bytes for wireless transmission of a frame. In this method the main technique used for setting the bytes is bit shifting. As an example, this is the implementation of the `getBytes()` method for the BTP layer:

```
public byte[] getBytes() {
    byte[] data = new byte[size];

    data[0] = (byte) (destinationPort >> 8);
    data[1] = (byte) destinationPort;
    data[2] = (byte) (sourcePort >> 8);
    data[3] = (byte) sourcePort;

    return data;
}
```

## 4.4 Broadcasting Cooperative Awareness Messages

Before broadcasting CAM messages the network interface for the modem must be configured. The following lines have to be appended to `/etc/network/interfaces` of the Raspberry Pi:

```
allow-hotplug usb0
iface usb0 inet static
address 192.168.11.1
netmask 255.255.255.0
network 192.168.11.0
broadcast 192.168.11.255
```

The class responsible for broadcasting CAM messages is `KapchModem` which implements the `Modem` interface. It runs within a `ModemThread` and it receives `MovingObjects` from a global synchronized `MessageBuffer<MovingObject>` for transmission (figure A.1).

`ModemThread` is responsible for transmitting frames with the right frequency. According to [11] CAM messages should not be transmitted more often than 1 message per 100 ms. The algorithm implementing this functionality in the `run()` method of `ModemThread` is illustrated below:

```
modem.initiate();

MovingObject obj;
long[] movingObjectLastTransmitted = new long[64];
while ((obj = globalBuffer.fetch()) != null) {

    // Get the current time
    long currTime = System.currentTimeMillis();

    // Check if at least 100 ms have passed
    if (currTime > movingObjectLastTransmitted[obj.id] + 100) {

        // If yes transmit it and set the new time
        modem.transmitMovingObject(obj);
        movingObjectLastTransmitted[obj.id] = currTime;
    }
}
```

Any modem class implementing the `Modem` interface is required to implement two methods; `initiate()` which is responsible for setting up a connection with the 802.11p modem and `transmitMovingObject(MovingObject obj)` which transmits a triple of GN-BTP-CAM byte data for any given `MovingObject`.

The class `KapchModem` uses a Kapsch ETTE 5.9 802.11p modem [16] for implementing the `Modem` interface. From the modem documentation and reverse engineering a couple of provided example scripts in python<sup>6</sup> the functionality of the modem was implemented.

The modem uses Ethernet over USB which is a protocol for Ethernet communications on top of the USB protocol. In order to transmit a message it has to be embedded within a UDP datagram first. Then this UDP datagram is being sent to the modem's IP address and finally the modem unpacks the datagram and broadcasts the provided payload using the 802.11p protocol. To initiate a connection with the modem a UDP message containing the string *Hi there!* must be sent to the modem. If the modem replies with the string *TS3306 is here!* then transmission of messages to the modem can begin.

They key feature here is that `transmitMovingObject(MovingObject obj)` uses the previously described `getBytes()` method of the GN, BTP and CAM classes to

---

<sup>6</sup><http://www.python.org>

translate the `MovingObject` into a byte array for transmission.  
`transmitMovingObject(MovingObject obj)` implements that as following:

```
void transmitMovingObject(MovingObject obj) {
    byte[] data = new byte[GN.size + BTP.size + CAM.size];

    GN gn = new GN(obj.kind, localModemMACAddress,
        obj.position, obj.velocity);
    BTP btp = new BTP((short) 2001, (short) 2001);
    CAM cam = new CAM((byte) 0, (long) obj.id, obj.position);

    // Add the GN layer to the datagram
    byte[] gnData = gn.getBytes();
    for (int i = 0; i < GN.size; i++)
        data[i] = gnData[i];

    // Add the BTP layer to the datagram
    byte[] btpData = btp.getBytes();
    for (int i = 0; i < BTP.size; i++)
        data[GN.size + i] = btpData[i];

    // Add the CAM layer to the datagram
    byte[] camData = cam.getBytes();
    for (int i = 0; i < CAM.size; i++)
        data[GN.size + BTP.size + i] = camData[i];

    sendPacket(data);
}
```

## Case enclosure

---

This chapter describes the possibilities of a fully integrated casing for the hardware solution, so it can be protected from impacts and weather exposure.

### 5.1 Design

The design was entirely made in SolidWorks 2013, which provided simplicity to create the 3D model, assembly, dimensioned 2D drawings and STL files.

#### 5.1.1 SolidWorks 2013

SolidWorks 2013 is a computer-assisted design (CAD) program for mechanical modeling. It is developed by Dassault Systèmes SolidWorks Corp.<sup>1</sup>

It allows to model separate parts and assemblies beginning with sketches on a base plane and extruding them (i.e. by adding or subtracting material) towards the final shape. Once a final product has been achieved, the program automatically outputs the 2D drawings and the STL files necessary for 3D printing or production.

Additionally, the program features a finite element analysis engine that can calculate component displacements, strains and stresses under internal and external loads. [12]

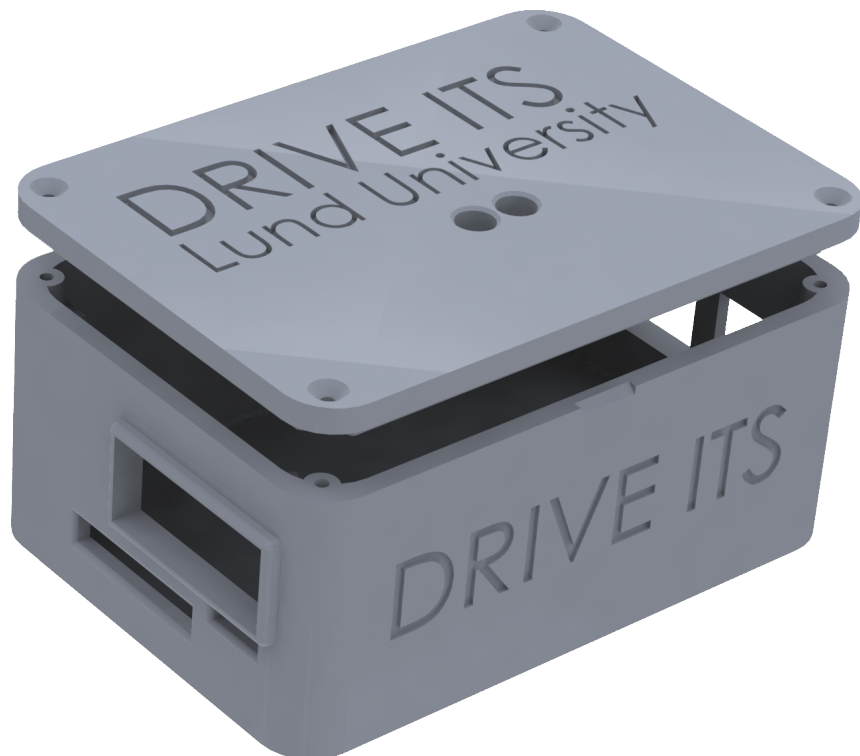
---

<sup>1</sup><http://www.solidworks.com>

### 5.1.2 Drawings and STL files

SolidWorks 2013 automatically draws any orthogonal view of the designed piece or assembly, thus saving time for the designer. Nevertheless, there is still the need of properly dimensioning the piece in order to allow a correct interpretation of the design.

The STL file format, which stands for *STereoLitography* is widely supported in rapid prototyping machines that describes the surface geometry of a 3D object.



**Figure 5.1:** 3D model of the case enclosure.

## 5.2 Production and materials

### 5.2.1 Rapid prototyping

A prototype can be defined as the limited representation of a system's or product's design, which allows the human team involved in its creation to experiment their usage, explore the design and present the product. The goal when creating a

prototype is to validate the essential requirements of the product while accepting any chance of modification. At the moment the prototype is sufficiently perfected, the team can consider the industrial production of the final product.

Rapid prototyping consists of obtaining physical models out of a 3D CAD system in a short time. Among the multiple advantages of rapid prototyping, the following can be listed:

- It allows the validation of final geometry.
- It reduces the designing process time.
- It allows the verification of assemblies.
- Verification of the design during the development phase.
- Prototypes don't require multiple machines to be produced, independently of the geometry.
- Detection of design errors before production, decreasing the production costs.

Nevertheless, rapid prototyping is limited to parts made out of plastic and their resolution, though it can be high, it doesn't allow the production of superfinished surfaces or extremely accurate geometries.

In the past, the availability of these machines was limited to big industries due to their size and price. However, in the last few years, the prices of 3D printers are being reduced drastically as a result of the growing interest on home-made parts.

Technologies for rapid prototyping include polymerization techniques, such as *Stereolithography (SLA)* and *Solid Ground Curing (SGC)*, sintering techniques (i.e. *Selective Laser Sintering, SLS*) and the most common among the 3D printers, *Fused Deposition Modeling (FDM)*, where a thermoplastic material (typically ABS or polyamides) is extruded through a hot nozzle. The material is deposited layer by layer, with a typical resolution of 0.1mm.

For the prototypes that have been implemented during the thesis research, *acrylonitrile butadiene styrene (ABS)* plastic has been the chosen material. It is a classic thermoplastic among FDM prototyping as its glass transition temperature is high enough to reduce unwanted deformation and it is a good material for finished products, given its good mechanical properties, such as impact resistance, toughness and heat resistance. ABS polymers are resistant to aqueous acids, which is an important property when considering a direct exposition to rain in contaminated areas.

### 5.2.2 Industrial production

If decided to produce the case in industrial quantities, 3D printing should be dismissed as it is a time-demanding and expensive production process. Instead, a less expensive, mass production oriented process should be selected.

Given the shape of the piece, after adapting the case design in order to simplify the mould geometry, the selection of process should head to vacuum thermoforming for a relatively low number of units or injection molding, only for a big number of units given the high cost of the tooling and equipment and the high-production rate [13].

Possible polymers for the final product include *acrylonitrile butadiene styrene (ABS)*, *high impact polystyrene (HIPS)* or *rigid polyvinyl chloride (uPVC)*, given their excellent properties against impacts, weather degradation and impermeability.

A thorough study of water leakage into the case should also be considered, since the current prototype only provides a small protection. Possibilities include the introduction of rubber around the open-air elements in order to assure their isolation against weather elements and avoid the degradation of the internal components.



## Tests and results

---

### 6.1 Test session at Volvo Cars

Two different tests were performed at Volvo Cars in Gothenburg. The first test, communication test, was to see whether the Kapsch Modem [16] could successfully communicate with Drive C2X<sup>1</sup> equipment or not. The second test, verification test, was to see whether Drive ITS CAM implementation followed the CAM specification [11] by comparing packets from another CAM source, e.g. a Drive C2X car, to packets generated by Drive ITS.

#### 6.1.1 Communication test

Drive ITS was set to simulation mode according to Appendix C, it was powered up and it started transmitting simulated traffic messages. The Drive C2X receiver was configured, powered up and any received traffic was saved to a file. The file was inspected in Wireshark<sup>2</sup> and the acquired packets could successfully be interpreted by the Wireshark dissectors<sup>3</sup>. That meant that Drive ITS was generating valid Cooperative Awareness Messages with correct format and that the 802.11p Kapsch modem was compatible with Drive C2X equipment.

---

<sup>1</sup><http://www.drive-c2x.eu/project>

<sup>2</sup>Wireshark is a free and open-source packet analyzer. <http://www.wireshark.org>

<sup>3</sup><http://www.amb-consulting.com/en/#downloads>

### 6.1.2 Verification test

During this test there was a setup with 3 main components; Drive ITS, a Drive C2X receiver and a Drive C2X Volvo car with capabilities of broadcasting its own CAM messages.

Drive ITS was installed and configured next to a road according to Appendix C. The Drive C2X receiver was also installed and configured in the same spot. The Volvo car was driven on the road next to the radar a few times. The external Drive C2X receiver was capturing packets during this whole time.

Once the test was completed the captured packets were analyzed and compared side by side in Wireshark. As seen in figure 6.1 the packets were almost identical which means that the test was a success. The only deviation was the car heading. However, it was soon realized that the heading reported by Drive ITS was in relation to the direction of the radar while the heading reported by the Drive C2X car was in relation to the North.

Latitude: 57°44'2647.69"N (577354698)	Latitude: 57°44'2647.97"N (577355470)
Longitude: 11°51'3102.12"E (118616989)	Longitude: 11°51'3102.42"E (118617823)
Speed: 10.44 m/s   37.58 km/h (1044)	Speed: 10.00 m/s   36.00 km/h (1000)
Heading: 5.4° (54)	Heading: 240.0° (2400)
(a) Drive ITS	(b) Drive C2X Volvo Car

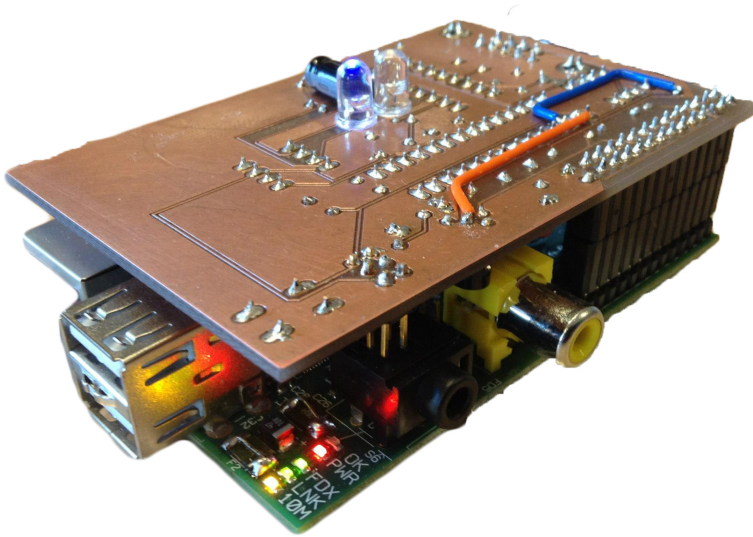
**Figure 6.1:** Captured packets from Drive ITS and Drive C2X systems.

The radar bearing was measured with a smartphone to  $45^\circ$  from the North advancing clockwise. However, the car was moving in the opposite direction which means that the back side of the radar was facing  $45^\circ + 180^\circ = 225^\circ$  from the North advancing clockwise. After compensating for the bad heading  $225^\circ + 5.4^\circ = 230.4^\circ$  heading for the car was calculated, which is much closer to  $240^\circ$  and within an acceptable error margin for a smartphone. The algorithm was corrected and no more deviations existed.

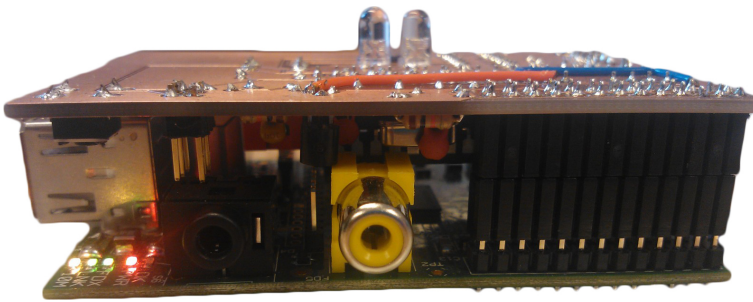
## 6.2 Final product

This section presents the final product, which is the result of combining the hardware and software part of the master's thesis. It is delivered as a neat package that can be easily assembled and placed at any location, given its reduced size and sturdy feel.

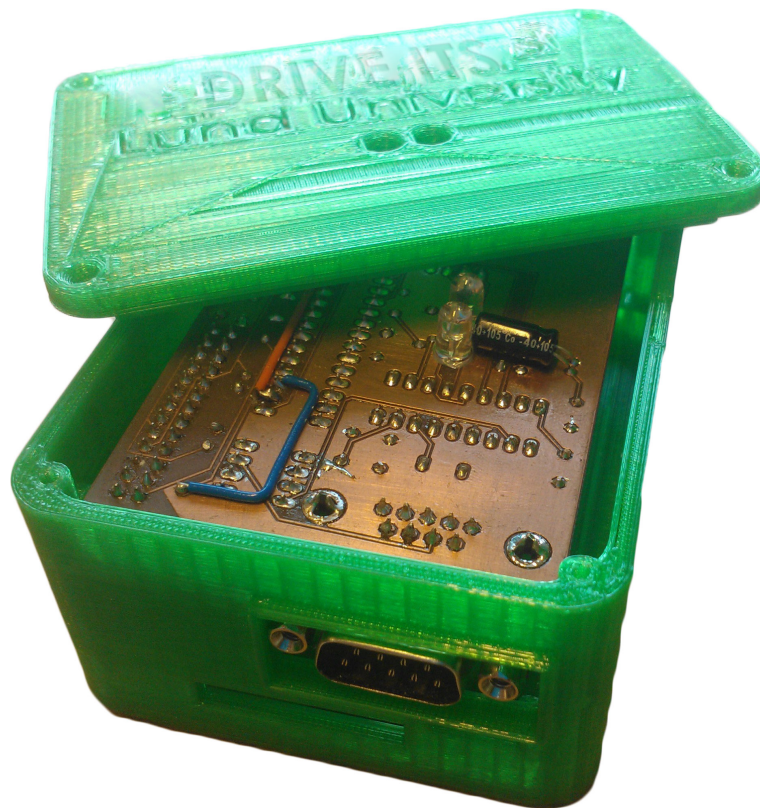
Figures 6.2 and 6.3 depict actual photographs of the Drive ITS product. Figure 6.4 depicts the case enclosure containing the Drive ITS prototype.



**Figure 6.2:** General view of the Drive ITS prototype.



**Figure 6.3:** Lateral view of the Drive ITS prototype, featuring the shield connector.



**Figure 6.4:** Photography of a final assembly inside the case enclosure, produced using FDM.

## Discussion

---

### 7.1 Conclusions

Throughout the present master's thesis, a device named *Drive ITS* has successfully been developed. It is able to emulate the car-to-car communication of non-intelligent vehicles within the sight of the radar, thus providing a better integration of ITS vehicle cooperation in the future.

The hardware part succeeds on connecting the radar and the 802.11p modem, providing an intelligent device in the middle that is powerful enough to perform all needed calculations.

The software part is tailored to successfully understand the radar's frames and translate them into messages that intelligent vehicles can process and use to improve active safety on the roads.

With this technology, intelligent transportation will be realized even during the early implementation phase, where just a small portion of the vehicles on public roads will feature a cooperative communications system. It is also a long-term solution, since older vehicles will keep driving on public roads for a long time.

By implementing the system in blind spots, intelligent vehicles will benefit from extra active safety as they will be warned about surrounding vehicles.

## 7.2 Future development

### 7.2.1 Web server

Given the availability of an Ethernet interface on the Raspberry Pi, it would be possible to connect the Drive ITS device to Internet, creating a web server that can provide traffic statistics (traffic congestion, real-time vehicle monitoring on a map, etc...) and could also be means of remote configuration.

A solution involving an encrypted Virtual Private Network, such as OpenVPN<sup>1</sup>, should be considered since the system should only be available to the operators of the network and not the general public.

### 7.2.2 Communication with traffic lights

The radar needs to be placed somewhere high and with good visibility over the road. Since traffic lights are already positioned in places like that they make the perfect spot to install a radar.

The radar on the other hand can see the traffic and it is possible to collect statistics about it. Having such statistics from all the radars all over a city each radar could provide its individual traffic light with recommendations about when to turn green and when to turn red.

Such a feature would require an extra connection for the traffic lights and further development of software for collection of statistics and generation of recommendations.

### 7.2.3 Multiple radars and devices over CAN

The current system only allows for one Radar and one Drive ITS device on the CAN bus, but it could be possible to access multiple radars with a single Drive ITS device, or even plug more than one Drive ITS device that would allow to have multiple 802.11p emitters.

---

<sup>1</sup><http://openvpn.net>

### 7.2.4 User interface

Currently, all configuration of the device is done via an SD card that contains a settings file, requiring to modify it with a visual text editor, either by connecting a screen to the Drive ITS device or plugging the SD card to a personal computer.

It could be possible to develop a user interface via an LCD screen and a small keypad, allowing easy and quick configuration of the device.

### 7.2.5 Beaglebone Black

A Beaglebone Black<sup>2</sup> replacing the Raspberry Pi would eliminate the need for an SD-card since there is 2GB flash memory on it. Also it features a smaller form, which would be useful in reducing the overall device size. A major advantage is that it features 5 UART Serial ports. That could open up new possibilities such as adding an onboard GPS module to the system for positioning and time synchronization.

It is also more inclined towards embeddedness and the hardware design is released under an open source license making it the perfect platform to change and base ones product on.

### 7.2.6 Ethernet over USB driver

One suggestion is to develop an Ethernet over USB driver for the AVR platform in order to eliminate the need for a Raspberry Pi and provide a more embedded solution.

Such an improvement would require an estimate of 6 months to develop, since it consists of developing the hardware, an Ethernet over USB driver and porting the software running on the Raspberry Pi to the AVR platform using a multitasking solution.

### 7.2.7 Standby operation

For low-traffic situations, or in case of temporary closed roads, it could be interesting to support a standby operation, with drastically lower energy consumption. Both the CAN transceiver and the CAN controller support it, but the Drive ITS device currently does not profit from this feature.

---

<sup>2</sup><http://beagleboard.org/Products/BeagleBone%20Black>





---

## References

---

- [1] European Telecommunications Standards Institute (2009), *Final draft ETSI ES 202 663 V1.1.0*. ETSI, France.
- [2] H. Eisele, E. Jöhnk (1996), *PCA82C250 / 251 CAN Transceiver Application Note*. Philips Semiconductors - Product Concept & Application Laboratory Hamburg, Germany (Currently NXP).
- [3] Microchip Technology Inc. (2007), *Stand-Alone CAN Controller With SPI Interface (MCP2515 Datasheet)*. Chandler, Arizona, USA.
- [4] Intrepid Control Systems Inc., *Microchip Controller Area Network (CAN) Bit Timing Calculator*. <http://www.intrepidcs.com/support/mbtime.htm> - Last visit 25-05-2013
- [5] Modelrail.Otenko, *Arduino + Controller Area Network (CAN)*. <http://modelrail.otenko.com/arduino/arduino-controller-area-network-can> - Last visit 25-05-2013
- [6] Sparkfun Electronics, *CAN-BUS Shield* <https://www.sparkfun.com/products/10039> - Last visit 25-05-2013
- [7] Crockett Engineering, *CANduino - Controller Area Network shield for Arduino compliant devices*, <https://code.google.com/p/candduino/> - Last visit 14-04-2013.
- [8] H. Schutte (1997), *Application Note. Bi-directional level shifter for P<sup>2</sup>C-bus and other systems*. Philips Semiconductors Systems Laboratory Eindhoven, The Netherlands (Currently NXP).
- [9] European Telecommunications Standards Institute (2011), *ETSI TS 102 636-4-1 V1.1.1* ETSI, France.
- [10] European Telecommunications Standards Institute (2011), *ETSI TS 102 636-5-1 V1.1.1* ETSI, France.

- 
- [11] European Telecommunications Standards Institute (2011), *ETSI TS 102 637-2 V1.2.1* ETSI, France.
  - [12] Dassault Systèmes SolidWorks Corp. *Finite Element Analysis (FEA) Overview*, <http://solidworks.com/sw/products/simulation/finite-element-analysis.htm> - Last visit 03-04-2013.
  - [13] S. Kalpakjian, S.R. Schmid (2009) *Manufacturing Engineering and Technology*. (6th ed.) Pearson Education
  - [14] Sveriges Ingenjörer, *Ingångslöner*, <http://www.sverigesingenjorerer.se/0m-forbundet/Sa-tycker-vi/ingangslon/>. Last visit 02-04-2013.
  - [15] Smart Microwave Sensors GmbH (2012), *UMRR Traffic Management Sensor Full Documentation*
  - [16] Kapsch TrafficCom AB (2012), *Product Specification, ETTE 5.9 model, Doc No. ETTE-00-001, Version A6*
  - [17] International Telecommunication Union (2001), *List of Mobile Country or Geographical Area Codes (Complement to ITU-T Recommendation E.212 (11/98))*



## UML diagrams

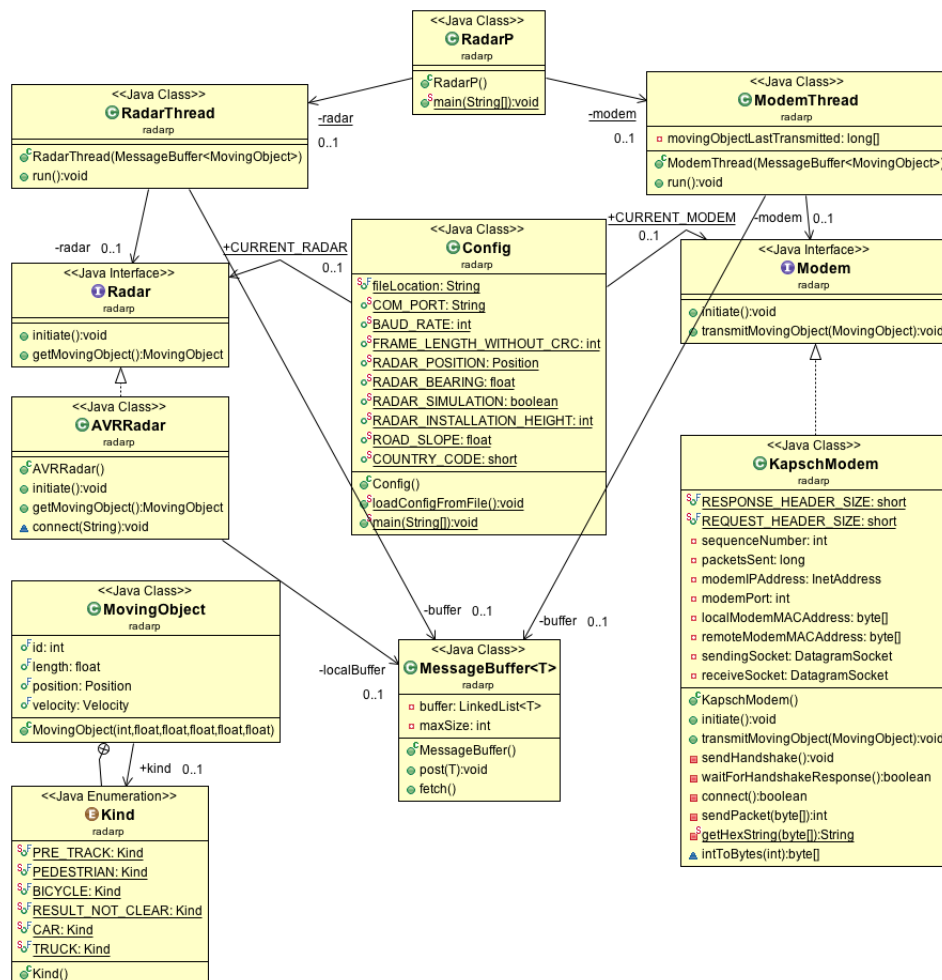


Figure A.1: UML diagram for the radarp package.

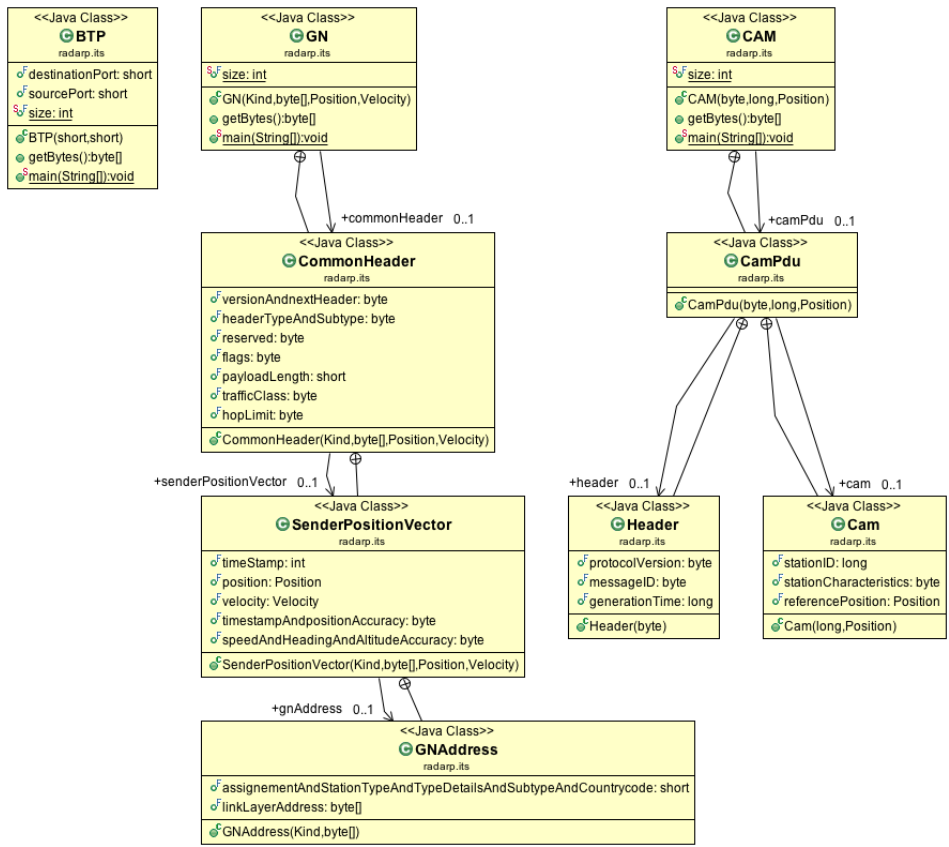
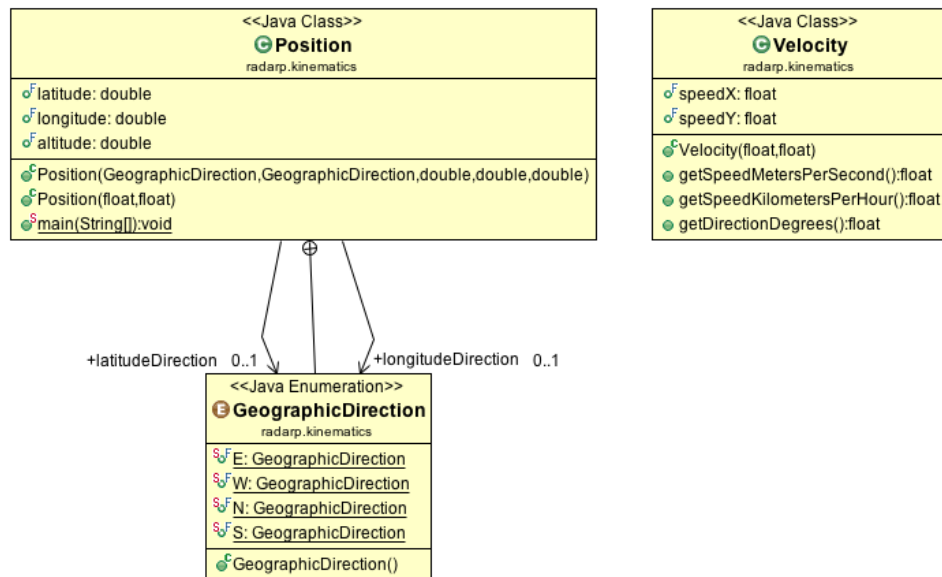
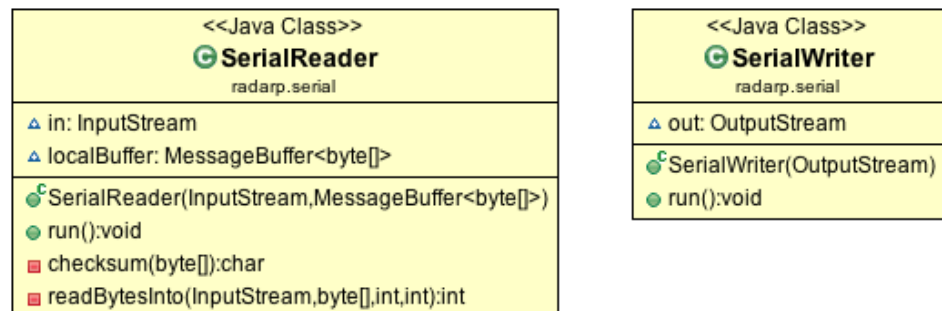


Figure A.2: UML diagram for the radarp.its package.

Figure A.3: UML diagram for the `radarp.kinematics` package.Figure A.4: UML diagram for the `radarp.serial` package.

## Programming manual

---

### B.1 AVR microcontroller

The AVR microcontroller can be programmed in two different ways. The first way involves an Arduino<sup>1</sup> platform while the second way uses an external AVR programmer. Using an Arduino has the advantage that an expensive external programmer is not required. However, in order for that to work the microcontroller must be preloaded the Arduino bootloader<sup>2</sup>. Using an external AVR programmer, in this case the AVR Dragon<sup>3</sup>, has the advantage that the chip doesn't have to be removed and it doesn't have to contain a bootloader thus all of its flash memory can be used for programming it.

#### B.1.1 Arduino platform

Since the microcontroller used here is an AVR ATMEGA328, an Arduino UNO or an Arduino Duemilanove platform needs to be utilized. The following steps must be followed:

1. Detach the *ATMEGA328* from the system.
2. Attach the *ATMEGA328* to the *Arduino UNO* or *Arduino Duemilanove w/ ATmega328* platform.

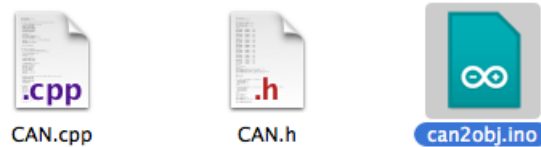
---

<sup>1</sup><http://arduino.cc>

<sup>2</sup><http://arduino.cc/en/Hacking/Bootloader>

<sup>3</sup><http://www.atmel.com/tools/AVRDRAGON.aspx>

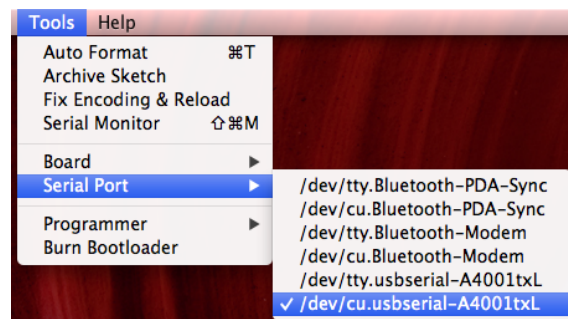
3. Start the Arduino programming environment by double clicking on the `can2obj.ino` file.



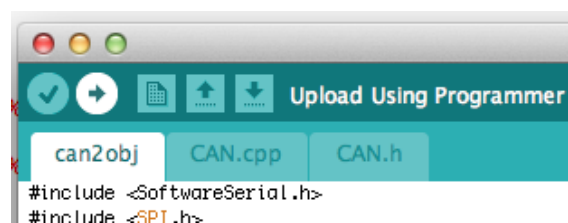
4. Make sure the *Arduino UNO* or *Arduino Duemilanove w/ ATmega328* board is selected from the *Tools* menu.



5. Make sure the correct *Serial Port* is selected from the *Tools* menu.



6. Hit the *Upload* button and wait until the process has completed.



7. Detach the *ATMEGA328* from the *Arduino UNO* or *Arduino Duemilanove* platform and attach it back to the system.



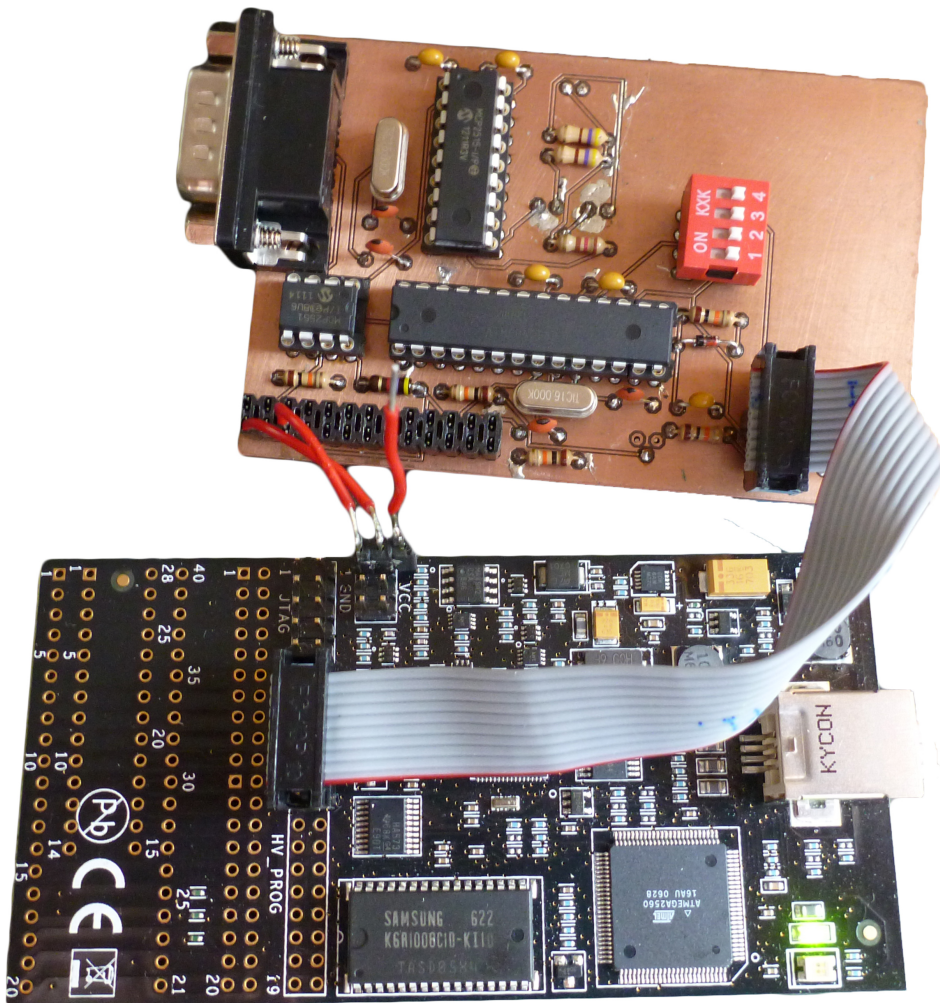
## B.1.2 AVR Dragon programmer

Before the AVR Dragon programmer can be used the Arduino programming environment needs to be tweaked in such a way that it can utilize the programmer.

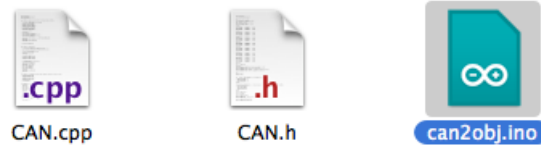
Find and edit the `hardware/arduino/programmers.txt` file by appending the following lines to the end of the file:

```
dragon_isp.name=AVR Dragon ISP
dragon_isp.communication=usb
dragon_isp.protocol=dragon_isp
```

1. Attach the *SPI* port of the programmer to the *SPI* port of the system using a ribbon cable. Also attach *VCC* and *GND* to the system accordingly.



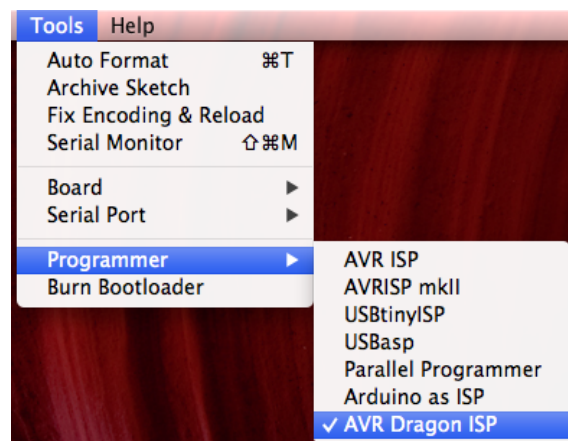
2. Start the Arduino programming environment by double clicking on the `can2obj.ino` file.



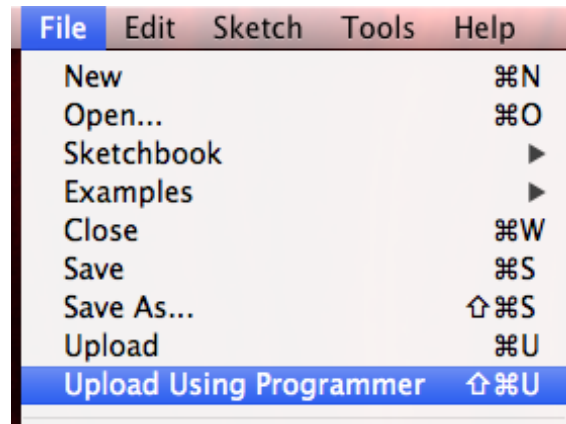
3. Make sure the *Arduino UNO* or *Arduino Duemilanove w/ ATmega328* board is selected from the *Tools* menu.



4. Make sure the *AVR Dragon ISP* programmer is selected from the *Tools* menu.



5. Hit *Upload Using Programmer* from the *File* menu and wait until the process has completed.



6. Detach the system from the programmer.

Keep in mind that programming an AVR microcontroller with an external programmer will wipe out the Arduino bootloader, thus removing the possibility of ever programming it again using an Arduino platform unless the bootloader gets flashed to the chip again.

## B.2 Raspberry Pi

The Raspberry Pi is a computer, there is no need for direct programming to be made on it. The daemon is written in Java therefore a Jar file is constructed and copied to the Raspberry Pi. To accomplish that one must follow the following steps:

1. Open the RadarP project in Eclipse<sup>4</sup>.
2. Go to *File* → *Export* → *Java* → *Runnable JAR file* and click *Next*.
3. Select *RadarP - RadarP* from the *Launch configuration*, select an *Export destination*, make sure *Extract required libraries into generated JAR* is selected and click *Finish*. Click *OK* if a warning window pops up.
4. Copy the generated Jar file to the Raspberry Pi using WinSCP<sup>5</sup> or the following command for a Unix system. When asked for the password type **thesis**:

```
scp radarp.jar pi@10.123.123.1:/home/pi/
```

---

<sup>4</sup><http://www.eclipse.org>

<sup>5</sup><http://winscp.net>

5. Connect to the Raspberry Pi through SSH and type the following commands. For Windows, Putty<sup>6</sup> can be used. The username is `pi` and the password `thesis`:

```
ssh -l pi 10.123.123.1 #Skip this step in Windows
cd /home/pi
chmod +x radarp.jar
```

---

<sup>6</sup><http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>

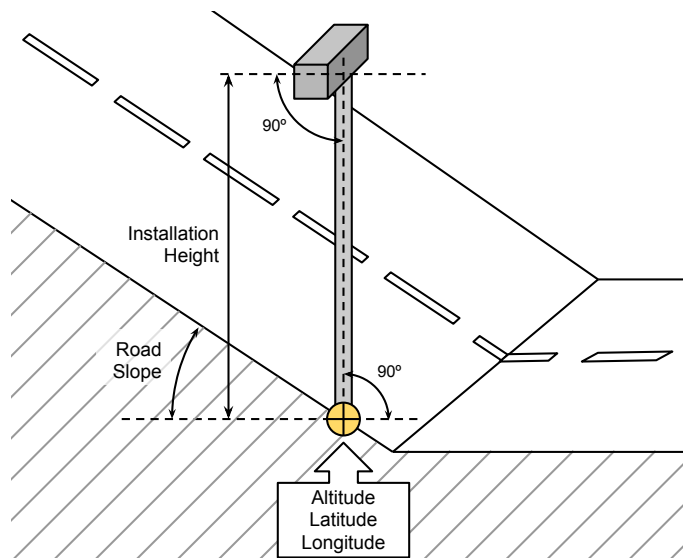
## Installation manual

---

Before installing the radar and the rest of the system one must first consider if there is good visibility of the road from the installation point. It is advisable that the radar should not have any walls in its close proximity and that trees and other structures should not block its line of sight.

### C.1 Placement and configuration

Figure C.1 illustrates the placement of the radar. Pay attention to that it should be facing 90° relative to center of the earth and not the road. This can be measured using a spirit level.



**Figure C.1:** Placement of the radar.

After the radar has been installed it has to be configured. This can be done by removing the SD card and connecting it to a computer. In the root directory of the SD card there is a file called `radarp.conf` to be found, this is the configuration file. Open the file and the following variables will be found inside:

```
# North or South (N, S)
radarLatitudeDirection = N

# East or West (E, W)
radarLongitudeDirection = E

# Latitude
radarLatitude = 55.711646

# Longitude
radarLongitude = 13.211060

# Altitude
radarAltitude = 66.43

# Facing direction of the radar in degrees. 0 degrees is north,
# advancing clockwise.
radarBearing = -73.0

# Whether the radar should simulate traffic or not
radarSimulation = false

# Installation height of the radar in meters. Min 1m max 10m.
radarInstallationHeight = 4

# Slope of the road in degrees. Positive is uphill and negative
# is downhill. 0 is flat.
roadSlope = 3.5

# Sweden according to COMPLEMENT TO ITU-T RECOMMENDATION E.212 (11/98)
countryCode = 240
```

Most of the variables are self-explanatory. However, `radarAltitude` is the altitude from the sea level at the bottom of the post, not at the installation point. `radarInstallationHeight` is the height of the post, 1m at least and 10m at most. The correct `countryCode` can be found in [17].

Edit the variables according to your wishes and save the configuration file. Place the SD card back in to the system and connect the power unit. In order to start the daemon follow the following procedure:

1. Connect the Raspberry Pi to a computer using an Ethernet cable.
2. Manually set the IP of the computer address to 10.123.123.2.
3. Use Putty<sup>1</sup> (Windows) or ssh (Unix) to connect to the ssh server of the Raspberry Pi. The IP is 10.123.123.1, the username is pi and the password is `thesis`.
4. Push the *Reset* button onboard the system shield.
5. Execute the following command: `java -var /home/pi/radarp.jar &`

The daemon should now be running, detecting vehicles and transmitting CAM messages.

---

<sup>1</sup><http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>