CHALMERS

In Vehicle Infotainment Demonstrator

Master of Science Thesis Software Engineer and Technology

Master of Science Thesis Intetgrated Electronic System Design

HANS SANELL
GÖRAN SAMUELSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, March 2011

In Vehicle Infotainment Demonstrator.
Research report on how to create an In Vehicle Infotainment system using existing hardware and Linux OS.

Hans Sanell
Göran Samuelsson

Examiner: Jan Jonsson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden March 2011

ABSTRACT

In modern cars the information systems have increased in the last decade. To handle this information flow to the driver a stable and intuitive computer system is needed. These systems are currently built on purpose specific hardware running purpose specific software, which is quite expensive to develop. This thesis describes a project where a multipurpose computer on module running a Linux distribution can act as an infotainment demonstrator. The report details how such demonstrator can be built and programmed to receive communication packages via a CAN bus.

PREFACE

This thesis was written for Chalmers University of Technology, Göteborg, Sweden as part of the Master programs Software Engineer and Technology and Integrated Electronic System Design. This Thesis work was performed by Hans Sanell and Göran Samuelsson at QRTECH, Göteborg. Examiner and supervisor at Chalmers for this project was Jan Jonsson at the Department of Computer Science and Engineering. Supervisor at QRTECH was Anders Runeson.

We would like to thank our supervisor at QRTECH for the dedication and help in our project.

TABLE OF CONTENTS

VOCABULARY

CAN	Controller Area Network
COM	Computer On Module
CS	Chip Select
DLC	Data Length Code
ECU	Electronic Control Unit
FMS	Fleet Management System
GUI	Graphical User Interface
IC	Integrated Circuit
IDE	Integrated Development Environment
IVI	In-vehicle-infotainment
IRQ	Interrupt ReQuest
MISO	Master Input Slave Output
MOSI	Master Output Slave Input
OS	Operating System
RPM	Revolutions Per Minute
SPI	Serial Peripheral Interface bus
SCK	Serial Clock
XML	eXtended Markup Language

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

In modern cars there has been an increase in information flow to and from the driver. To handle this information easy to use systems have to be developed. These systems are currently quite costly to manufacture due to specialized parts

In order to address this problem QRTECH assembled a master thesis with a goal to create an infotainment system using off-the-shelf hardware combined with a Linux kernel.

1.1. BACKGROUND

This section will describe the need for this thesis, its addresses problem and its delimitations, as well as the consultant firm that requested the thesis.

1.1.1. ABOUT QRTECH

QRTECH is a consultant firm that specializes in electronic and software development in a series of high tech areas where their main business areas are Product Development, Advance Engineering and Product Supply & Support. Their long experience in the automotive business has given them knowledge to develop and industrialize cost-efficient products designed for applications where the demand relies heavily on quality and environmental durability. QRTECH has knowledge of the whole life cycle of the product, from development and industrialization to support and refining.

Since they strive for excellence and to become leaders in their area of expertise all new areas of technology and solution have to be investigated, hence this master thesis.

1.1.2. PROBLEM DESCRIPTION

Development of hardware and software for a specific platform and purpose is known to very expensive, a specific hardware has to be designed and developed to fulfill the functionality of the software. The development of the software often includes writing everything that will match a certain hardware and purpose which often have to be rewritten next time if a new hardware setup is used. The basis for this thesis was to see if it is possible to achieve the same kind of performance but build the whole system on premade hardware and open source solutions. To that end this thesis set out to use GumStix hardware combined with the MeeGo Linux OS(operating system) to see if it is possible to connect the system to a CAN(Controller area network) network for receiving packages. The outcome of this thesis is supposed to lead to a fully functioning demonstrator built on MeeGo that will receive CAN-frames and display them appropriately on screen, which leaves very much for interpretations. It was not stated how far this thesis would be taken more than the investigation and how-to’s.

As both the desired hardware and OS were already given the strict research part was diminished quite early on and limited to research concerning the hardware and the specific Linux distribution.

1.1.2.1. DELIMITATIONS

Due to this just being a demonstrator to see what the hardware and software combinations can achieve several delimitations in the form of security and fault tolerance were set. Those issues would be considered later. The purpose of the thesis was instead mainly to investigate the systems ability to connect to the CAN-protocol, receive simple CAN-frames and display those accordingly. Since the project has a limited supply of displays everything will not be resolution independent, but all development will have that aspect in mind as far as possible.

1.1.2.2. TARGET PLATFORM

The targeted platform for the development is Gumstix Overo Fire COM (Computer on moduel) (1) which will run the MeeGo Linux OS (2). In the last phase the GumStix will be transferred to an in-house developed motherboard that will supply necessary amount of input and output pins. This assumes that the motherboard will be delivered on time. The reason for the use of an in-house developed motherboard is to remove unnecessary components, minimizing the board size and integrating the extra modules needed for CAN. In the meanwhile all development will be done on a Chestnut43 (3) expansion board for the GumStix. This will be connected to a 4.3 inch touch display from LG for both input and output. In addition to facilitate the development also a USB keyboard and mouse was connected to enable easier manipulation directly to the hardware. These have to be connected through a powered USB hub, because the USB port on the expansion board does not have a power supply.

1.1.3. PREVIOUS WORK

The early research on the subject showed that a hardware that communicates by CAN is not that hard to develop. Although most of the investigated work were based on an ATMEGA processor where the user has full control of all registers and I/O ports this thesis assumes that these will go through a MeeGo Linux Kernel. There has also been some previous work on GumStix, our hardware of choice, but not as extensive as we hoped. MeeGo, on the other hand, has successful been integrated into a so called Beagle board which has a similar hardware setup as the GumStix. With some modification it worked on the GumStix as well, mostly because the Beagle board is also built on ARM architecture.

2. THEORY

The theory that belongs to the basis for this thesis is described in this section, such as the chosen hardware and software.

2.1. MEEGO

A request in the thesis specification was to use existing hardware and software and since MeeGo's target group is handheld devices and is starting to get a foothold in the IVI business. Both due to it is quite new and they had a lot to learn about it as well as if MeeGo gets a real foothold in the IVI (In vehicle infotainment) business then QRTECH might be a bit ahead of different competitors. The MeeGo project started as collaboration between Intel and Nokia. Both of them have two separate Linux projects, Moblin from Intel and Maemo from Nokia. The whole MeeGo project is open source and it utilizes ideas found in both Moblin and Maemo, which is the OS currently in use in Nokia's N900. (4) The MeeGo distribution is still in the testing phase, currently at version 1.1, so it doesn't exists that many commercial products yet that uses it but it shows a great promise. The distribution is available for download at the MeeGo site (2) and for different platforms, depending what the target system is i.e. a handheld, and guides on how to install the system. Since it is still in development there are some problem with the user interface which will be discussed more in the Discussion section. Even though MeeGo targets smaller system it still has a complete Linux Kernel which needs some configuration before it can be installed in the target system for example drivers for peripherals and such.

2.2. GUMSTIX

GumStix is the name of the hardware that was used in this thesis project. The name GumStix comes from the size of the computer-on-module chip, dimensions 17mm x 58mm x 4.2mm (1), which is about the size of a chewing gum. It has a 720 MHz ARM processor which is quite useful both due to the speed and the architecture; most handheld devices today are built on ARM. An extension board was used as well together with a 4.3 inch touch screen to be able to develop the system. The upside of using GumStix is that it is fairly easy to develop a motherboard on your own and install the OS on a SD card and boot from the stick. The extension board comes with integrated mini USB connector (serial communication with a standard PC), an Ethernet connector for network services and a slave USB port. Also, but not relevant for this thesis, the expansion board has a stereo output port as well as a microphone in port. All in all the GumStix with the expansion board gave a good start for the thesis with communication possibilities, display device and the extra I/O-pins easily available.

The GumStix is delivered with an Angstrom Linux distribution on the onboard memory, but since the request was to use MeeGo this was overlooked.

One of the reasons for the use of MeeGo is the so called GENIVI alliance (5). This alliance is a collaboration of different car manufactures, embedded system developers and semiconductor manufacturer like Intel. The goal is to build a foundation for an IVI system that can shorten the development time for new IVI's and hence reduce the development costs based on open source software. GENIVI has said that they will start to develop for the MeeGo platform due to its optimization for the handheld devices.

2.3. CAN

In a modern car there might be up to 70 ECU for different subsystems (6); those ECU's(Electronic control unit) needs to communicate with each other. The CAN bus may be used in vehicles to connect the engine control unit to the dashboard for instance, or control the door locks and climate control.

In a modern car the cruise control is a good example. The velocity of the car must be regulated, but also the breaks and automatic fuel injection which are needed for more power. There are probably a lot more functions which need to be combined for a feature like cruise control. At a first glance it might seem simple but it is a more difficult task than one might think.

2.3.1. HISTORY

In early 1980s the engineers at Bosch were revising the existing serial bus systems and how they could be used in passenger cars but since none of the current network protocols fulfilled the requirement they started developing on a new bus type in 1983. Quite early in the specification phase Mercedes-Benz got involved and also Intel as the potential main semiconductor vendor (7). So in February 1986 the CAN was presented at a SAE (Society of Automotive Engineers) congress in Detroit (8). The protocol defines a standard for efficient and reliable communication between sensors, actuator and other nodes in a real-time application. The idea was that the bus would take care of all messages without a central bus master by giving access to the bus to the message with the highest priority. The Bosch CAN specification, which was at version 2.0 at the time, was submitted to the international standardization in the early 1990s. In 1993 CAN got the ISO standard 11898.

2.3.2. THE PROTOCOL

CAN is a so called multi-master broadcast serial bus for connecting ECUs, meaning that every node is able to send and receive messages although not simultaneously. Each message consist first of an identifier that is used to present the priority of the message. If the bus is free any node can feel free to start transmission, but if two or more nodes begin at the same time the message with the more dominant id will supersede the ones with lower priority. A message with smaller id-value has higher priority and is transmitted first.

A CAN data frame consists of either an 11 bits identifier (standard format or CAN 2.0A) (9) (10) or a 29 bits identifier (extended format or CAN 2.0B), made up from the base identifiers first 11 bits and an 18-bit extension. An IDE (Identifier extension bit) bit is used to distinguish between the two types of frames where the bit is dominant in the 11-bits case and recessive in the 29-bit case. The CAN protocol supports 4 different frame types but this thesis will only utilize the standard and extended data frames. All the frame types are listed below. Since only the data frames are of interest for this thesis they will be shown in a more detailed way in table 1 and 2.

- Data frame: a frame containing the node data for transmission.
- Remote frame: a frame requesting the transmission of a specific identifier.
- Error frame: a frame transmitted by any node detecting an error.
- Overload frame: a frame to inject a delay between data and/or remote frame.

BASE DATA FORMAT (6)

Field name	Length(bits)	Purpose
Start-of-frame (SOF)	1	Denotes the start of the frame transmission
Identifier (ID)	11	An identifier for the data which also serves as the message priority
Remote transmission request (RTR)	1	0 for a data frame
Identifier extension bit (IDE)	1	Must be dominant (0)
Reserved bit (r0)	1	Reserved bit
Data length code (DLC)[1]	4	Number of bytes of data (0-8 bytes)
Data Field	0-8 bytes	Data to be transmitted, decided by the DLC
CRC	15	Cyclic redundancy check
CRC delimiter	1	Must be 1
ACK slot	1	Transmitter sends recessive (1) and any receiver can assert dominant (0)
ACK delimiter	1	Must be 1
End-of-frame (EOF)	7	Must be 1

Table 1

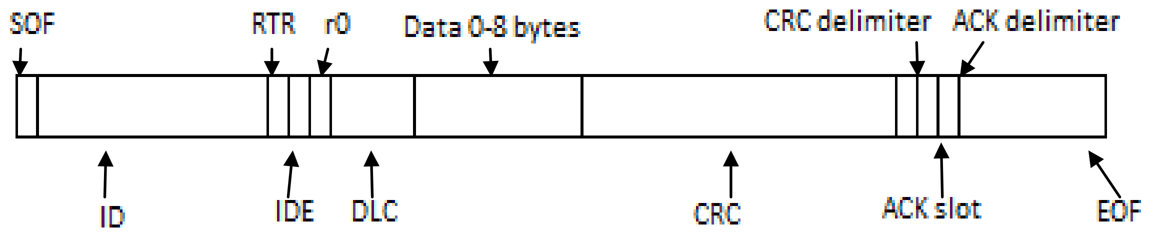


Figure 1 CAN-frame, standard format

[1] It is physically possible to address values up to 15 with the 4 bits but the data is still limited to 8 bytes maxium.

EXTENDED DATA FORMAT (6)

Field name	**Length(bits)**	**Purpose**
Start-of-frame (SOF)	1	Denotes the start of the frame transmission
Identifier A (ID)	11	First part of the identifier for the data which also serves as the message priority
Substitute remote request (SRR)	1	Must be 1
Identifier extension bit (IDE)	1	Must be 1
Identifier B (IDB)	18	Second part of the identifier for the data
Remote transmission request (RTR)	1	Must be 0
Reserved bits(r0, r1)	2	Reserved bits, set to 0
Data Length code (DLC)[2]	4	Number of bytes of data (0-8 bytes)
Data Field	0-8 bytes	Data to be transmitted(length decided by
CRC	15	Cyclic redundancy check
CRC delimiter	1	Must be 1
ACK slot	1	Transmitter sends recessive (1) and any receiver can assert dominant (0)
ACK delimiter	1	Must be 1
End-of-frame (EOF)	7	Must be 1

Table 2

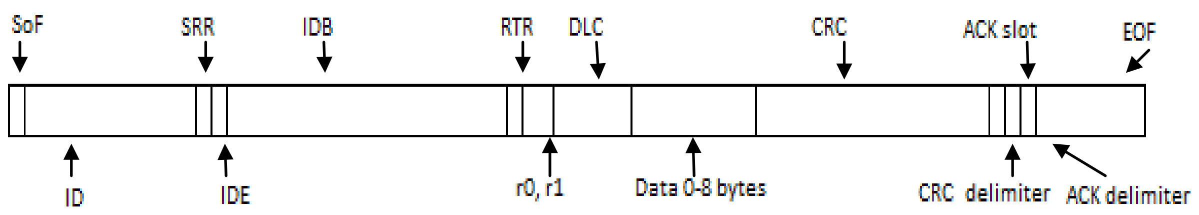


Figure 2 CAN-frame, extended format

[2] It is physically possible to address values up to 15 with the 4 bits but the data is still limited to 8 bytes maximum.

3. METHOD

The method that was used was a modified iterative V-mode (11); where each part of the system was developed through first specifies what was needed to be done, followed by implementing this specific feature. After the implementation and integration was done the feature was tested to verify its correctness. All features designed and implemented had testing close in mind. If the feature did what was intended the cycle starts over with a new feature. This process was used out of necessity since there was no clear picture what was needed in the program or the system itself. During the development some changes were necessary.

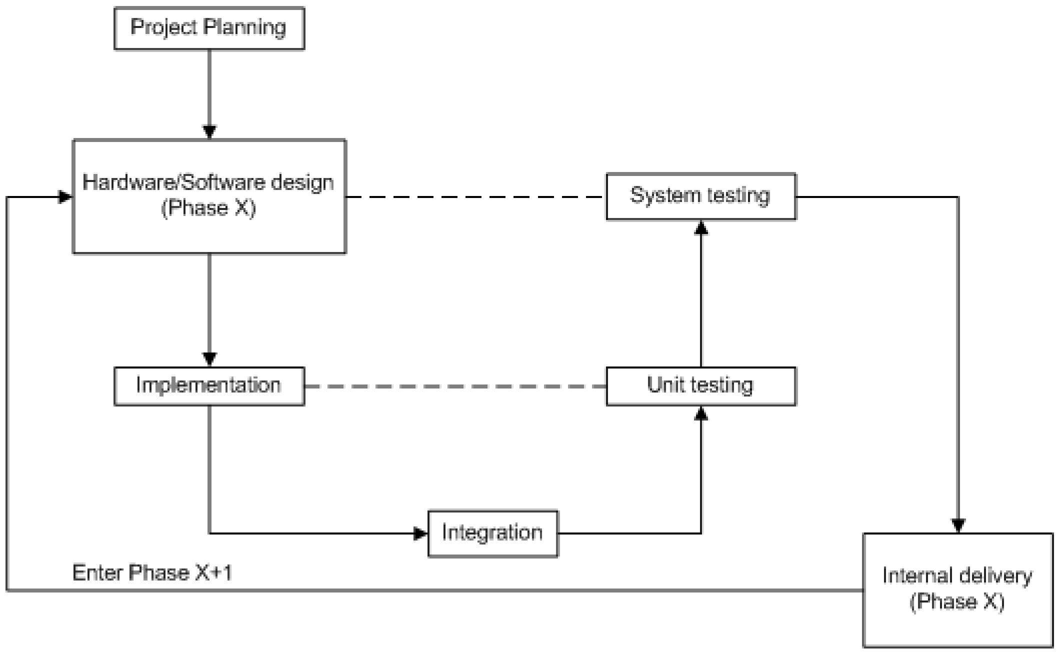


Figure 3 Method model

The complementary time plan can be found in Appendix 1.

4. DEMONSTRATOR DESIGN AND IMPLEMENTATION

The demonstrator will have some specific requirements for the design and these will be followed in the implementation phase. The implementation section will only describe what was done during a successful integration; all problems on the way are discussed in the result and discussion sections.

4.1. DESIGN

The software high level application would be developed in C++ with the overlaying framework QT in the given IDE (Integrated development environment) QTCreator, which is the default IDE for the QT framework (12). This makes it possible to develop graphical user interfaces in a similar way as Microsoft Visual Studio that many people are familiar with. This drag-and-drop system in combination with a CSS style sheet editor a sleek design for the user interface can be achieved easily. Since the majority of the functionality and time will be spent on the kernel and communication protocol to the CAN-controller the development time on the UI will be as minimized as possible.

4.1.1. SOFTWARE REQUIREMENTS

Some of the requirements were set before the start of the project by QRTECH and some were decided to be necessary during the development phase.

SRQ1:
Name: Speed
Description: Due to the speed of the CAN-controller a 2x sampling speed of the bus is needed to avoid anti aliasing. This is needed to ensure that all packages are received.

SRQ2:
Name: Reliability
Description: The system should be reliable to show the correct data from the can controller

SRQ3:
Name: Configurability
Description: The program should be easy to configure to receive/send another set of CAN-frames.

SRQ4:
Name: Touch screen
Description: The interface should be developed with respect that a touch screen will be used.

SRQ5:
Name: Configuration
Description: The framework for setting CAN-frame settings should be easy to extend to new protocols.

SRQ6:
Name: Concurrently

Description: Since the interrupts to the kernel is difficult to develop, threading should be used to avoid unnecessary polling.

SRQ7:
Name: OS
Description: The program should be build for the MeeGo Linux distribution

SRQ8:
Name: Framework
Description: The demo program should be build as extensively as possible in C++ with Nokias Qt-framework, available as default in the MeeGo distribution.

SRQ9:
Name: Extendibility
Description: The whole program should be developed in such a way that QRTECH can continue the development at some point without the core knowledge of the developers.

SRQ10:
Name: Understandability
Description: The program should be easy to use and understand, with regard to the other SRQ's such as small display and touch screen.

SRQ11:
Name: Resolution
Description: The program should not be resolution depended and be developed for small resolutions such as 480x272

SRQ12:
Name: Input
Description: The program should be able to perform all it is tasks without the user connecting a mouse or a keyboard, but if needed it should work with that setup as well.

SRQ13:
Name: Guidance
Description: A help function should be implemented to guide the user through the different functionalities.

SRQ14:
Name: Thread safe
Description: Since threading will be used all communications between threads and processes will be safe to avoid locking such as live lock, deadlock or starvation, around shared resources.

4.1.2. USE CASES

To easier understand what the software part of the project should be able to achieve use cases was written down to keep the development path clear.

NAME: Receive CAN frame
PRECONDITION: The program is up and running connected to correct peripheral hardware can-controller
POST CONDITION: The data from the can frame will show up on the screen under a specific category
STEPS:
1. Drivers receive the CAN-frame.
2. The middleware translates the frame to a data structure that can be used according to current CAN-configuration
3. Middleware gives the GUI application the data which sorts it out and represent it accordingly on screen

NAME: Try to receive CAN frame
PRECONDITION: -
POST CONDITION: The user gets notified that the peripheral is not connected
STEPS:
1. The middleware checks if the CAN-controller is connected.
2. The middleware sends error to the GUI application.
3. The GUI application displays this error accordingly, letting the user know that the peripheral hardware is not connected.

NAME: Send CAN data
PRECONDITION: A configuration is set and the peripherals are connected
POST CONDITION: Successful send of the CAN-frame
STEPS:
1. The user selects 'send CAN-data'
2. The user selects from a drop down menu what id/name the frame should have
3. The user inputs the data that should be sent
4. The user press 'send'
5. The user gets confirmation that the CAN-frame has been sent.

4.2. IMPLEMENTATION

To implement a fully functioning demonstrator for CAN using the GumStix hardware, the MeeGo Linux distribution and the MCP2515 CAN-controller (13), the controller of choice in this thesis due to it's simplicity and functionality, requires a couple of steps to be completed.

Some embedded CPU's have a controller integrated to receive and send CAN data, the GumStix does not have that integrated instead an external one had to be implemented, hence the MCP2515.

A CAN-controller is needed for receiving CAN-frames. It puts a frame in a buffer slot and gives the CPU an interrupt to say that there is data to be read. This happens after the CAN-controller has checked that the CAN-frame is not corrupt. The same thing could be achieved by using the GumStix but there is a case when the kernel is busy and will not have time to service the newly sent package.

All of these steps assume that the user has access to a PC with Ubuntu installed with root access. First of all the GumStix uses a microSD memory card as hard drive so one memory card needs to be acquired. Since the chosen CAN-controller had drivers that were included in the distribution this is the example that will be followed here. To prepare the device and install the MeeGo Linux distribution the MeeGo wiki has some notes for installation and configuration which should be followed. (14) The steps are basically that first the memory card needs to be prepared to accept the correct file system; it needs a boot partition in FAT32 format as well as a partition for the distribution in EXT3 format. There are some scripts at the wiki page that is needed after the microSD card has been formatted. These scripts are the ones that take care of building the Kernel and its modules. It handles the assembly of the root file system as well installation of the boot loader, kernel and root file system into the microSD card. To be able to compile the Kernel into the GumStix a cross compiler is needed. GCC has one that is easy obtained and works well. This thesis is utilizing, as earlier mentioned, the MCP2515 CAN-controller that already has support in the Kernel in form of drivers. Although the drivers exist they still need some configuration to be fully implemented. To achieve this, the Kernel configuration for the target platform needs to be altered. In this case since the target platform is a GumStix Overo Fire, the “board_overo.c” file takes care of all the configurations. This is the file that needs altering. In the same directory there are files for all the target platforms that currently have support. In this file the “mcp251x.h” needs to be included to gain access to the driver. This driver also needs to be initialized to tell the Kernel where it can locate this driver after the build. In this thesis the choice of using one of the SPI ports was made so an initialization that connects the MCP2515 driver to the SPI stack, namely in this case the SPI1.1 port. The setup for the MCP2515 that needs to be added to the “board_overo.c” file can be found in Appendix 7. The initialization added to the init function for the SPI can be found below:

```
#if defined(CONFIG_CAN_MCP251X) || \
        defined(CONFIG_CAN_MCP251X_MODULE)
        {
                .modalias      = "mcp2515",
                .bus_num       = 1,
                .chip_select   = 1,
                .max_speed_hz  = 10*1000*1000,
                .controller_data = &overo_mcp251x_mcspi_config,
                .platform_data = &overo_mcp251x_config,
                .irq           =
OMAP_GPIO_IRQ(OVERO_GPIO_CAN0_INT),
                .mode          = SPI_MODE_0,
        },
#endif
```

One other configuration that was needed, for convenience sake, was the real time clock, this to enable the system to keep track of the date and time. To enable these following rows needs to be added to the device init function in “board_overo.c”:

```
#ifdef CONFIG_RTC_DRV_TWL4030
       &overo_twl4030rtc_device,
#endif
```

This will call the device constructor that should be added to the file “board_overo.c”:

```
static struct platform_device overo_twl4030rtc_device = {
       .name           = "twl4030_rtc",
       .id             = -1,
};
```

After all configurations are made the Kernel can now be build. When the build is complete without errors (warnings are a big probability, but not necessary a threat) the Kernel can be installed into the microSD card using the previously downloaded script. When the Kernel and boot loader is installed in the microSD card and all steps from the wiki page have been followed the MeeGo OS should start up when booted on the GumStix. Even though the system starts it will still need some configurations in form of screen resolutions and keyboard languages and so on. During the execution of this thesis the MeeGo window manager was not fully developed therefore to enable the development and use of the program a switch from MeeGo’s window manager to “twm-session” was made. When booted this will instead of the MeeGo GUI just show a basic user interface complete with a terminal, using this the execution of a specific application was made easy. The easiest way to integrate to the GumStix is by using SSH, this way an ordinary PC can be used for all configuration development. A good way to check if the driver initialization is working as intended is by checking the “dmesg” queue and see if the device in question is probed. In this case the command “dmesg | grep mcp2515” will get the result “mcp2515: probed”, which means that the driver was initialized successful. If the probing was successful then the platform is ready for development of CAN communication, otherwise something with the initialization of the driver is incorrect. How to develop to the socketCAN can be found in the documentation in the Kernel or found online (15). The recommended way to interface to the CAN-bus using socketCAN is to set up a socket in a similar way as if the communication would be TCP/IP instead. By using this socket a bit stream will be received that can be matched against a data structure.

```
struct can_frame {
    canid_t can_id; /* 32 bit CAN_ID + EFF/RTR/ERR flags */
    __u8 can_dlc; /* data length code: 0 .. 8 */
    __u8 data[8] __attribute__((aligned(8)));
};
```

This represent a basic CAN frame containing the CAN frame’s ID, its data length code and the data array, maximum of 8 byte of data. This structure can easily be used to identify different ID’s for further data manipulations as for instance typecasting to different objects depending on the ID and DLC.

After a socket has been created and bounded to the system the receiving part of the program is just to create a CAN frame data structure which will be sent as an argument to the read function together with the socket and the size of the data structure. This so that the read function will know where to read from, where to store the results and how many bytes it should read. The read function will return the number of bytes read from the stream and if the return value is zero or less than the size of the CAN frame data structure then the read was incomplete or not correctly configured.

When all the above steps are done the driver and Kernel settings are finished, now the only thing left is to write the middleware and software that will interact with the screen and user. The middleware in this case was just used as a layer between translating the CAN-frames to objects of choice for easier parsing, so that the software does not have to do the parsing against numerical ID but just after object types.

Connecting the middleware to a GUI (Graphical user interface) would then just be to import the library and connect the values read from the CAN-bus to different graphical widgets to update their respective values. The graphical widgets developed with the QT framework will update and redraw when there is time for it; meaning if the value comes faster than the GUI has time to update the update function will take the latest value given and discard earlier ones to minimize memory leaks. To avoid polling every time the GUI should update a threading solution is a recommended one, one thread blocks on the read instruction and pass on the data via message passing to the thread handling the graphical part. This mostly since the QT framework have excellent support for message passing and mutual exclusion functions.

As the MeeGo Linux distribution is a work in progress some details about the installation and compilation of the Kernel have been left out since in probably will change in coming releases.

These steps was the ones that was done in the execution of this thesis project but an alternative way to solve this probably exists but the chosen solution can be seen in figure 4.

A full class diagram can be found in the Appendix 4. A list of all essential hardware can be found in Appendix 3.

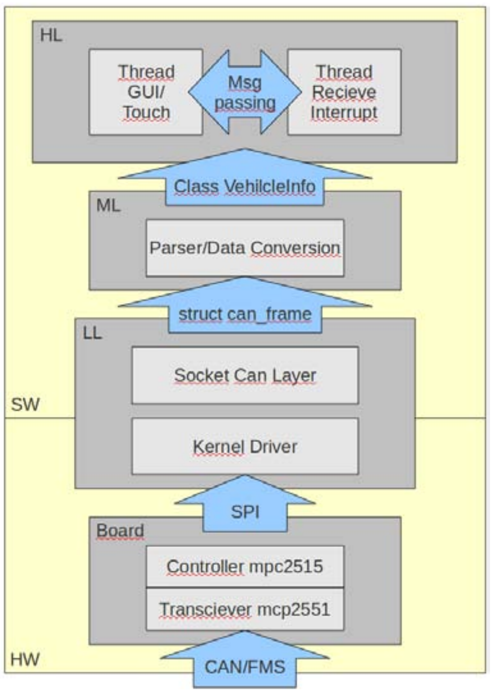


Figure 4 Solution structure

5. RESULT

At the end of the thesis a demonstrator that was fully functional was developed. It has the capabilities to receive CAN-frames from a CAN-bus that follows the FMS (Fleet management system) standard (16). A selection of a few frames was selected to test the implementation. The information received by the demonstrator is shown on the display. The underlying idea for the GUI development was that it should be easy to use without unnecessary information. If the device would be integrated into a vehicle the driver should be able to use the device without risk for accidents. The development of the software was done using C++ and the QT framework. After some development the QT frameworks limitation was shown regarding creation of own widgets and design. Due to this all graphical design had to be done by using the code-based approach, meaning that adding widgets to layouts and setting the size and position of these with numerical values. This took a bit of trial and error until a satisfying result was achieved. But the upside is that full control and insight in the program was given.

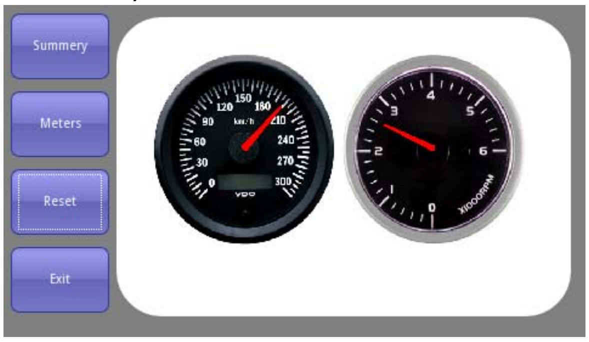
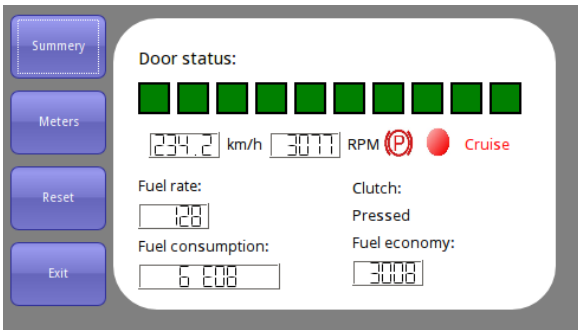


Figure 5 Demonstrator screenshot

See figure 5 for screenshot. As shown in the picture all messages regarding velocity and RPM is parsed and updated so the dials can show an appropriate value for these. A second widget tab was developed, it shows the velocity and RPM but as a numerical value instead with some labels for showing if the breaks, parking break, clutch and cruise control are active or not which can be seen in Figure 6.



Door status:

🟩 🟩 🟩 🟩 🟩 🟩 🟩 🟩 🟩 🟩

234.2 km/h 3077 RPM (P) ● Cruise

Fuel rate: 128

Clutch: Pressed

Fuel consumption: 6 E08

Fuel economy: 3008

Figure 6 Demonstrator screenshot

In the latter part of the development some problem with the communication came up, probable due to faulty cabling. This was temporarily resolved to add a third button to the GUI, a reset button, which sends a reset command to the CAN-controller so it starts receiving again. This button was only used during the development phase to overcome the current hardware problem and later removed. As the project progressed many ideas for implementation came up but just a few where chosen for implementation. Although due to the modularization of the software it is easy to add more custom widgets, more on this in the section Future work. During the development a CAN-to-USB device (17) was used to allow sending customized messages to see if the receiving end got the right package. To this CAN-to-USB device a Perl script was developed to continually send commands for the demonstrator to receive. Here a series of commands were sent to show the extent of the demonstrator's ability to both receive and show information. Where the periods of the CAN-frames ranged from 20ms to 1000ms the demonstrator was able to receive a command with at least 10ms apart, this without needing to use to extensive system resources. With periods of 10ms a zero package loss was achieved out of 1M packaged sent. This displays the demonstrator's ability to handle the small but fast amount of data without any problems.

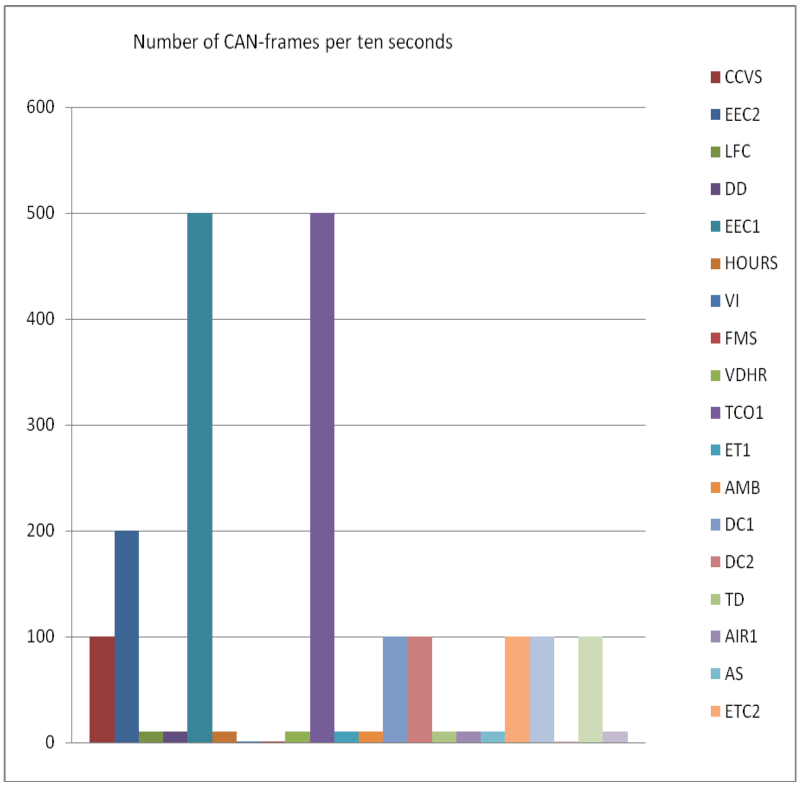


Figure 7 Frame intensity graph

Figure 7 shows a graph over the intensity of CAN-frames sent from a FMS Gateway signifying for example engine speed, fuel consumptions and so on. These frames are sent with different timings where some are sent often and some less often. VI for instance, is sent every tenth second, while others are sent very often. These timings are taken from the FMS specification and the number of packages gives a mean time of 190,3 packages per second, which is about 5ms per package. It is possible that for a full load some packages might be lost. To discover possible limits of the system more extensive testing should be done on the hardware and software.

To verify the implemented functions of the demonstrator a FMS gateway was borrowed from QRTECH. It has a database with values which are broadcasted with real timing intervals according to specification. This gateway will simulate a real CAN-bus under load and send CAN-frames with correct intervals, good for see if the hardware can handle real bus load with all messages according to the FMS specification.

When the final phase of the project started to approach we decided to develop a plug in card to the expansion board to move all our components from the breadboard to this card. This was done to remove the possibility of faulty cablings and contact surfaces. The difference can be seen in figure 8 and figure 9.

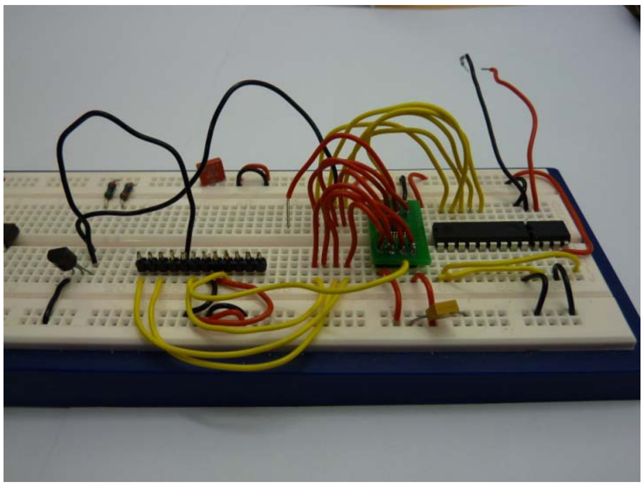


Figure 8 Breadboard

The plug-in extension board removed the problem with the resetting CAN-controller which leads to removing of the reset button in the program. This leads to the conclusion that it was the breadboard that was at fault for resetting the controller.

One of the problems with using an OS like Linux combined with your own hardware that is interrupt driven is that the control over the interrupt vectors is handled by the Kernel. When an interrupt is sent from the peripheral hardware it is really hard to control how and when the Kernel actually will be servicing the interrupt.

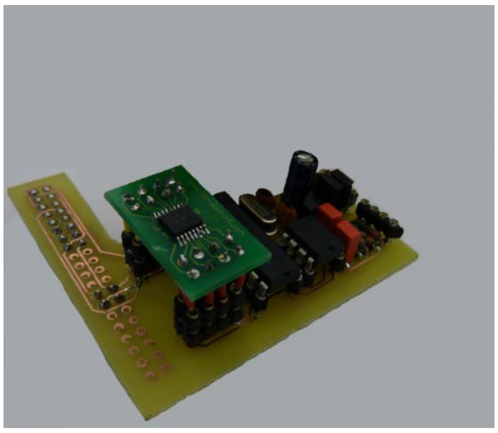


Figure 9 Circuit board

The software did work as intended but as mentioned above, the sending of a CAN-frame was not implemented, this due to time limitations as well as a request from QRTECH. If a device is allowed to send out to a real CAN-bus in a vehicle it has to undergo extensive testing and licensing to assure that no faulty packages are sent. Faulty packages might disrupt the other nodes.

6. DISCUSSION

As this thesis was classified as a more or less exploratory, QRTECH did not know if it was possible to do or how to get this done, there was a lot of trial and error. This was both good and bad in some sense. The bad part would be that when something came up that was hard to solve we had no one to ask for help leaving us more or less for ourselves. The good part about executing a exploratory thesis is the amount of knowledge amassed when trying out to see what will happened changing different things in the kernel configuration for instance. All in all it was a good experience.

6.1. SOFTWARE

As the development progressed, a general idea of what the high level software should do resulted in a couple of smaller software projects. The smaller projects were needed to solve some minor issues and developed independent of each other. Later they were integrated into one bigger project. The smaller project that was merged was:

- GUI programming: Using the IDE and manipulating the different widgets available as well as creating our own.
- Threading: Since the program uses threading this had to be investigated to be able to use it in a secure way.
- Message passing: To communicate between the threads we chose to use message passing, mostly since the QT framework has a good support for this.
- System calls and blocking: Since one of the threads will be calling the underlying layer and block until a message is available this had to be checked to see any unwanted behavior.

As these projects were completed we had a very good foundation of knowledge for creating what would be our demonstration program. See figure 5 for screenshot. One of the problems during the software development phase was caused by the need of our own widgets. This enabled us to encapsulate the different parts into one but it also disabled the possibility to use the graphical interface IDE from QTCreator. So the layouts had to be created using code style instead of the build in drag-and-drop system.

All in all, the QT framework and it's IDE, QTCreator, is fairly good but it needs some more work before it's really mature to be used to it's full capabilities.

After using the QT framework for some time we had a discussion if it would be better to use GKT+ instead. GTK+ is another graphical framework which is used extensively in the Ubuntu Linux distribution. But we decided upon keeping QT since it already has a compiler for ARM architecture as well as optimization for this.

There were also some discussions on how the information would be parsed and transported between the different software layers in the system. Should the CAN-frames be parsed and sorted in the lowest possible layer or should this be done in the highest possible layer? We concluded that it would be more computational effective to do it as close to the driver as possible to avoid unnecessary copying and reading of memory addresses. The other option would have been for the lower layers to pass on all the CAN-frames with a valid format and remove faulty ones. This would have left the sorting to the higher levels. The sorting part would be developed as a XML format where the CAN-ID was matched and all information about the number of data-bits and name of the frame was stored in an XML structure. This would enable further extension of this program to accept new CAN-frames with just knowledge of the FMS and some basic XML (Extended markup language). But this idea became unfeasible as it would not be possible to represent this new CAN-frame graphically in a good way without recompiling the GUI. Also it was not part of the thesis description but could be regarded as future work. The solution that was used instead was a more object oriented one where each CAN-frame would have a match in a collection of classes inherited from a base class, see class diagram in Appendix 4 for more detail. Each of these classes describes exactly how that specific object has its data sorted. In the case of extending this structure to accept a new CAN-frame it is just a question of extending the base class and implementing the respective methods.

The basic idea for the software - hardware interaction when receiving frames can be seen in the system sequence diagram in figure 10. The general idea is that the gateway thread will read from the caninterface which is the program code closest to the driver. The caninterface in turn will block if there is no data available. If the frame is known it will send it to the graphical user interface thread that will update correct widgets with this new data.

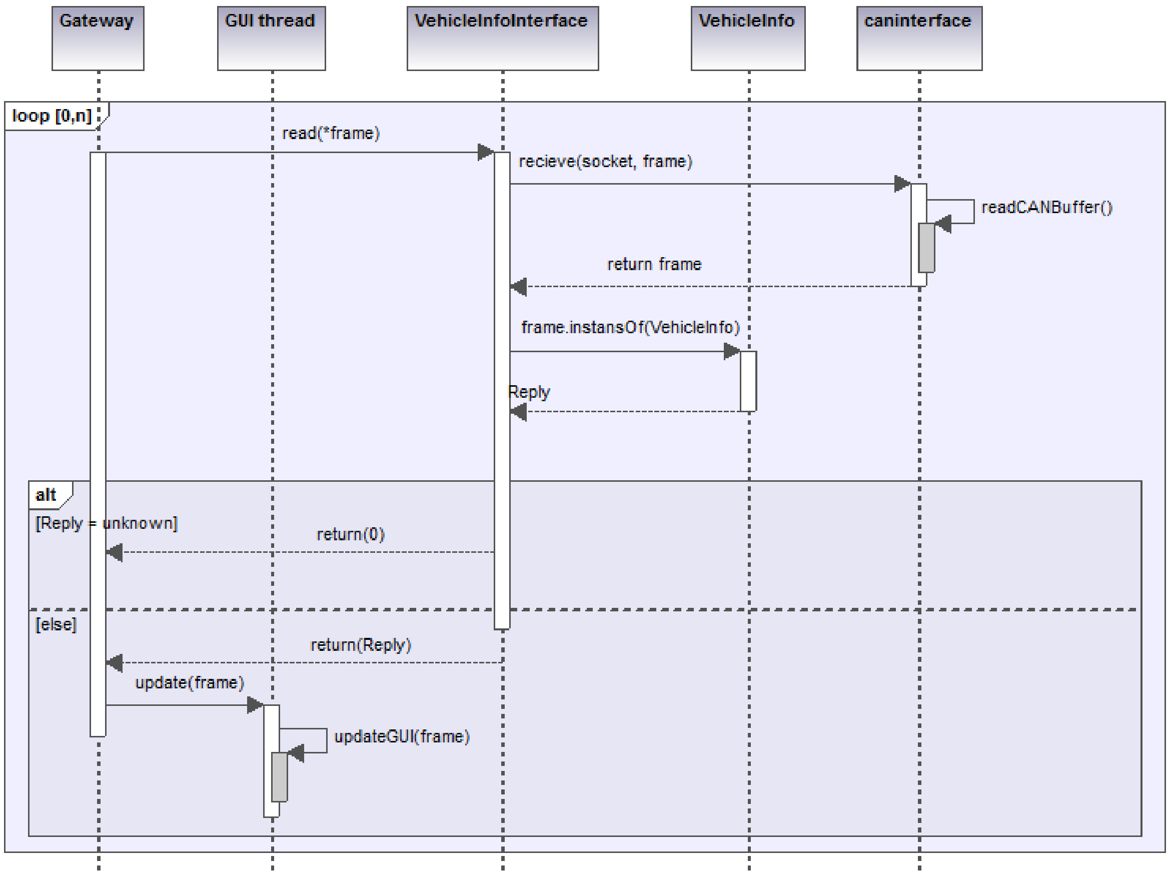


Figure 10 System sequence diagram

In the figure above, Figure 10, all communication between libraries and threads are done using pointers to different VechicleInfo objects. An early solution for this was to send entire objects but this was later on changed to use pointers instead, both to save memory and decrease execution time. The pointer solution is better in the aspects of memory conservation, execution speed and code structure. If the copy solution would be used a lot more execution time would be spent, time that might lead to missed CAN-frames.

As mentioned earlier there was some problem with the default user interface that belongs to MeeGo, the window manager did not work as intended at the current release. It was poorly optimized resulting in long response times and the configuration of menu items did not work as intended. This resulted in that only the standard programs were able to be started and even these worked poorly. That is why it was decided to change into the twm window manager, there had been enough time spent on trying to configure the windows with no result. The window manager was nothing that we had any control over and since the intended software would use the whole screen it did not matter that much which window manager that was used.

6.2. HARDWARE

To easier debug the hardware, i.e. what was sent and received, we had the opportunity to borrow one Logic unit (18) together with the device software. With this we could plot the different signals to determine if a signal propagated properly from one end to another. The logic unit and an oscilloscope gave us a fairly good idea what was happening on the breadboard.

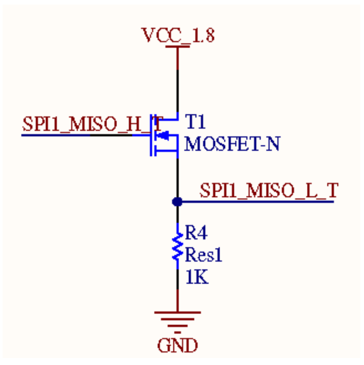


Figure 11 Transistor solution

The CAN-controller of choice is driven by 5V but the GumStix I/O ports are only on 1.8V so a level converter had to be integrated (19). This converter will act as a bridge between the CAN-controller and the I/O ports, but for some unknown reason the level converter can drive all the ports except the MISO (Master input slave output) port. This port will get a logical zero all the time. To verify that it was not a faulty circuit a second one was used but with the same result, tested different channels as well but the MISO still failed. So a solution involving a transistor was used instead for the MISO channel as can be seen in Figure 11.

One of the biggest weakness in the hardware setup is the CAN-controller which has only two buffers whiles the ones used more often in the industry have somewhere between 16 and 64 buffers. So it is theoretically possible for some package loss under full bus load due to full buffers, but that situation has not been simulated.

During the second half of the development phase some packages losses were discovered and some packages were too corrupt that the CAN-controller interpreted it as a reset command. This problem was probably due to connection glitch on the breadboard. The problem appeared from time to time during the development but was resolved when all components was migrated to our own extension board as suspected, see Figure 9.

6.2.1. MOTHERBOARD

All development and integration was done on the Overo Extension board as well as a breadboard for complementary components. QRTECH decided that an in-house motherboard was to be developed. The motherboard was intended to minimize the board size as well as removing unnecessary components and add the required functionalities. The development and design of this motherboard was outside the scope of this thesis work but we had the opportunity to review the hardware requirement specification to see if something was amiss.

Due to unforeseen factors this motherboard was not finished within the scope of the thesis and hence we were not able to test if the stability improved or not. Since the motherboard would not be delivered on time we developed our own extension board for the chestnut43 board to add the CAN functionality. A full circuit schema can be found in Appendix 2 as well as the layout for the board.

7. CONCLUSION

Here we will summarize some conclusions that we reached during this thesis execution, both regarding the software and hardware as well as what kind of future work this may lead to.

7.1. SOFTWARE

Regarding the software development we have first and foremost learnt a lot in terms of object oriented programming in a larger scale as in terms of lower level programming for Linux. This is a lot harder than one may think. The software became more generic than we originally thought, mostly because we had some time left at the end of the implementation phase. This makes it easier to extend the program in the future for further development. Sadly the development IDE was not one of the best on the market so some extra time on finding plug-ins (for instance to Eclipse (20) to use the QT framework) would have saved us time in the beginning. Also, most middleware communication had to be tested on the hardware. The second half of the development of the program was done through SSH using an ordinary terminal text editor.

7.2. HARDWARE

During the final phases of the development most problems were due to hardware problems, faulty wiring or contact surfaces that did not work.

For the trouble shooting of the hardware the Logic unit was priceless. It was used extensively throughout the hardware and development phase (18). The unit consists of eight small connectors for the hardware and one USB connector to the computer, as can be seen in Figure 12. With the associated program the unit can measure logical ones and zeros at each of these eight ports. This will in the program give eight graphs that will show the signals and their timing. The program itself is configurable so it can interpret the signal of a specific channel as, for instance, a CAN-frame.

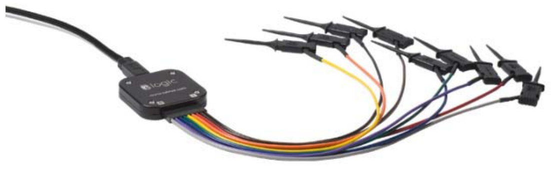


Figure 12 Logic unit

From the graphs that the Logic unit plots it is typically quite easy to find where the hardware is at fault, for example if a signal does not end up where it is supposed.

Examples as how the graphs looks like with some explanation can be seen in Figure 13.

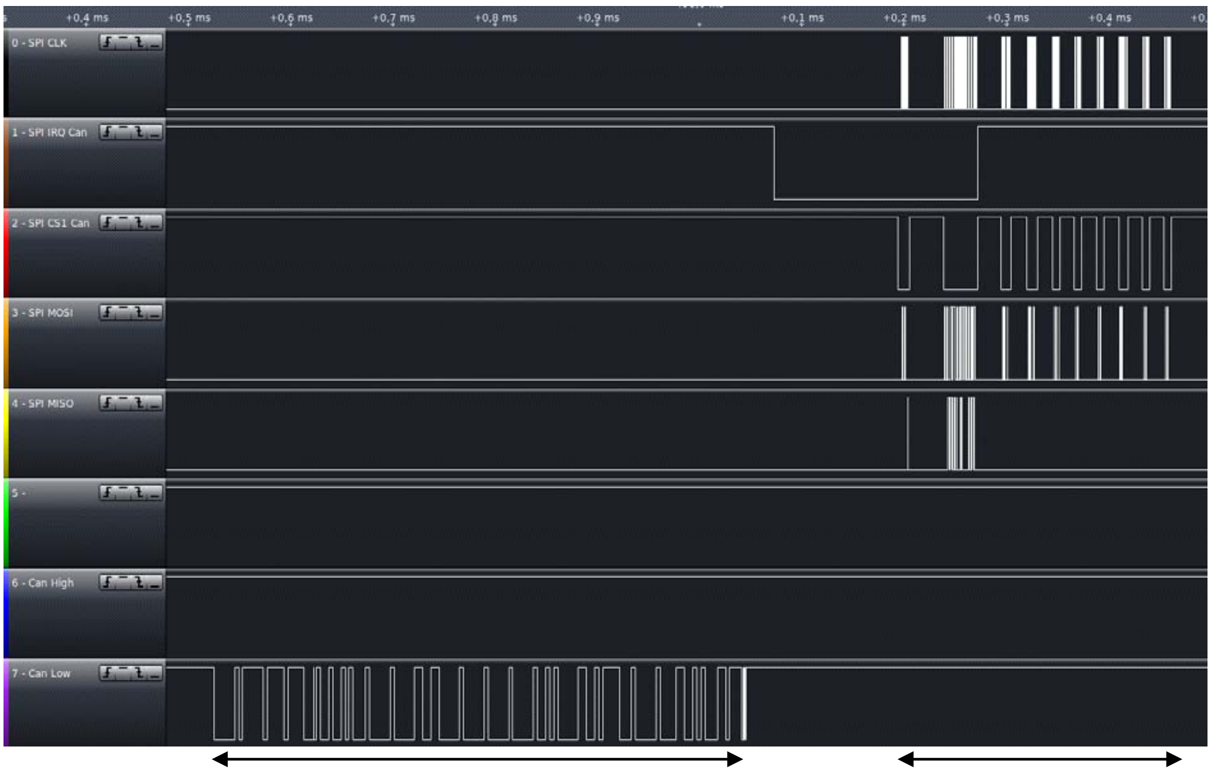


Figure 13 Plot over one CAN transmission

As Figure 13 shows that the CAN data in the CAN format before the CAN-controller takes about 0.5 ms to send with a 250kb/s bit rate, while it after the CAN-controller in the SPI (Serial peripheral interface bus) format only takes about 0.05ms in 10 MHz clock frequency; the second logical zero on the SPI CS1 channel (the red one). After the data have been received, and the IRQ has been set to logical one, the driver resets the status registers and interrupts flags on the CAN-controller.

As shown in Figure 14 each peak in the graph furthest down is a CAN-frame. In the figure each peak in the MOSI (Master output slave input) and MISO graphs are the same frames but on the other side of the CAN-controller. These peaks are smaller because the SPI bus has the capability to handle the data at a much faster rate than the CAN bus. These messages were generated from a CAN gateway that simulates the full load from a vehicle according to the FMS standard. By counting all incoming messages and interpreting them we can see that the demonstrator has a zero percent loss of packages.

As shown in Figure 14 where the demonstrator is under full load from a FMS gateway there is still plenty of time between the parsing of the different CAN-frames.

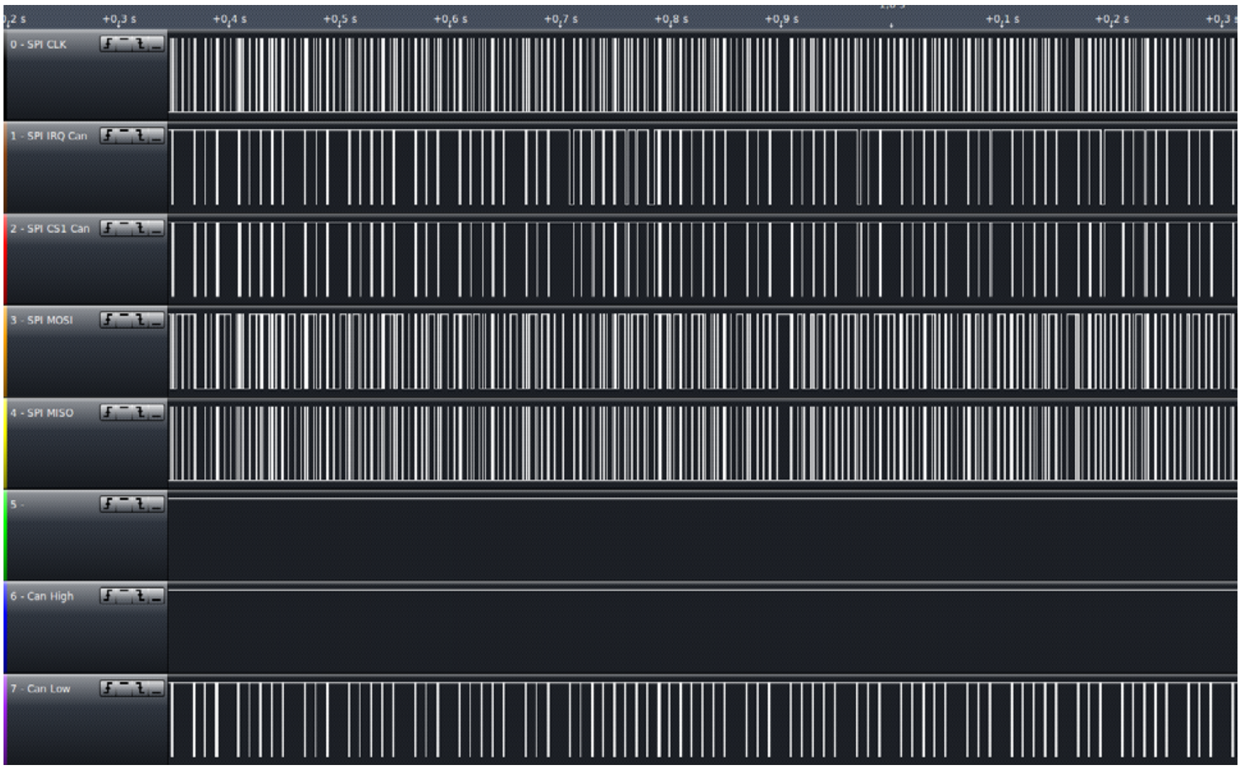


Figure 14 Screenshot from Logic unit under full FMS load

7.3. FUTURE WORK

As the areas of IVIs are huge any future work on this product could end up in basically anything. For instance, the analog velocity meter in the car with a digital screen, or the fuel consumption could be shown in real time to give the driver some insight in how economic his/her driving is. For entertainment purpose it could be possible to integrate it with screens in the backseat for playing video games, surfing on the Internet or just watching movies. An example would be when there are children in the back seat watching a movie, but they may fall asleep, the driver could then pause the movie, from the driver seat, until they wake up and want to continue watching.

This is just the tip of an iceberg for the possibilities of a good IVI system.

On a more relevant note for this demonstrator, the capabilities of adding and removing widgets in real time would enable it to accept new CAN-frames without having to recompile the project. This would require a lot of work and was therefore decided to not be within the scope of this thesis.

If the focus of this project would change to use another CAN standard than the FMS the only adaptation needed would be to change the implemented class in the middleware to receive a different bit pattern.

When this thesis was closing in on being finalized QRTECH decided that they wanted to continue on this development and thus created another master thesis that will use this one as a basis. This new thesis will focus more on the actual demonstration program now that the kernel and drivers are in place.

8. SOURCES

1. GumStix. [Online] [Cited: 09 01, 2010.] www.gumstix.com.

2. MeeGo. [Online] [Cited: 08 04, 2010.] www.meego.com.

3. GumStix Expansion board Chestnut43. [Online] gumstix, 2010. [Cited: 02 18, 2011.] http://www.gumstix.com/store/catalog/product_info.php?products_id=237.

4. Maemo. [Online] [Cited: 11 20, 2010.] http://maemo.nokia.com.

5. *GENIVI alliance.* [Online] 2010. [Cited: 10 20, 2010.] http://www.genivi.org.

6. Wikipedia. [Online] [Cited: 02 04, 2011.] http://en.wikipedia.org/wiki/Controller_area_network.

7. CAN history. [Online] [Cited: 02 17, 2011.] http://www.can-cia.de/index.php?id=161.

8. **Kerl Henrik Johansson, Martin Törngren, Lars Nielsen.** *Vehicle application of Controller Area Network.* Stockholm : s.n., 2005.

9. CAN specification. [Online] Bosch, 1991. [Cited: 01 05, 2011.] http://esd.cs.ucr.edu/webres/can20.pdf.

10. **Pazul, Keith.** Controller Area Network Basics. [Online] [Cited: 02 15, 2011.] http://www.cl.cam.ac.uk/research/srg/HAN/Lambda/webdocs/an713.pdf.

11. Wikipedia. [Online] [Cited: 02 20, 2011.] http://en.wikipedia.org/wiki/V-Model_(software_development).

12. QT framework. [Online] Nokia , 2011. [Cited: 02 17, 2011.] http://qt.nokia.com/products/.

13. Stand-Alone CAN Controller With SPI Interface. [Online] 2010. http://ww1.microchip.com/downloads/en/DeviceDoc/21801F.pdf.

14. MeeGo on Beagleboard. [Online] [Cited: 01 20, 2011.] http://wiki.meego.com/ARM/Meego_on_Beagleboard_from_scratch.

15. Socket CAN. [Online] [Cited: 02 17, 2011.] http://www.kernel.org/doc/Documentation/networking/can.txt.

16. FMS-standard. [Online] [Cited: 02 17, 2011.] http://www.fms-standard.com/index.htm.

17. Home automation project. [Online] [Cited: 01 16, 2011.] http://projekt.auml.se/homeautomation:hardware:avr:corecard.

18. Saleae - Logic unit. [Online] http://www.saleae.com/logic/.

19. TXB0106 level converter. [Online] 9 2008. [Cited: 02 18, 2011.] http://focus.ti.com/lit/ds/symlink/txb0106.pdf.

20. Eclipse. [Online] The Eclipse Foundation, 2011. http://www.eclipse.org/.

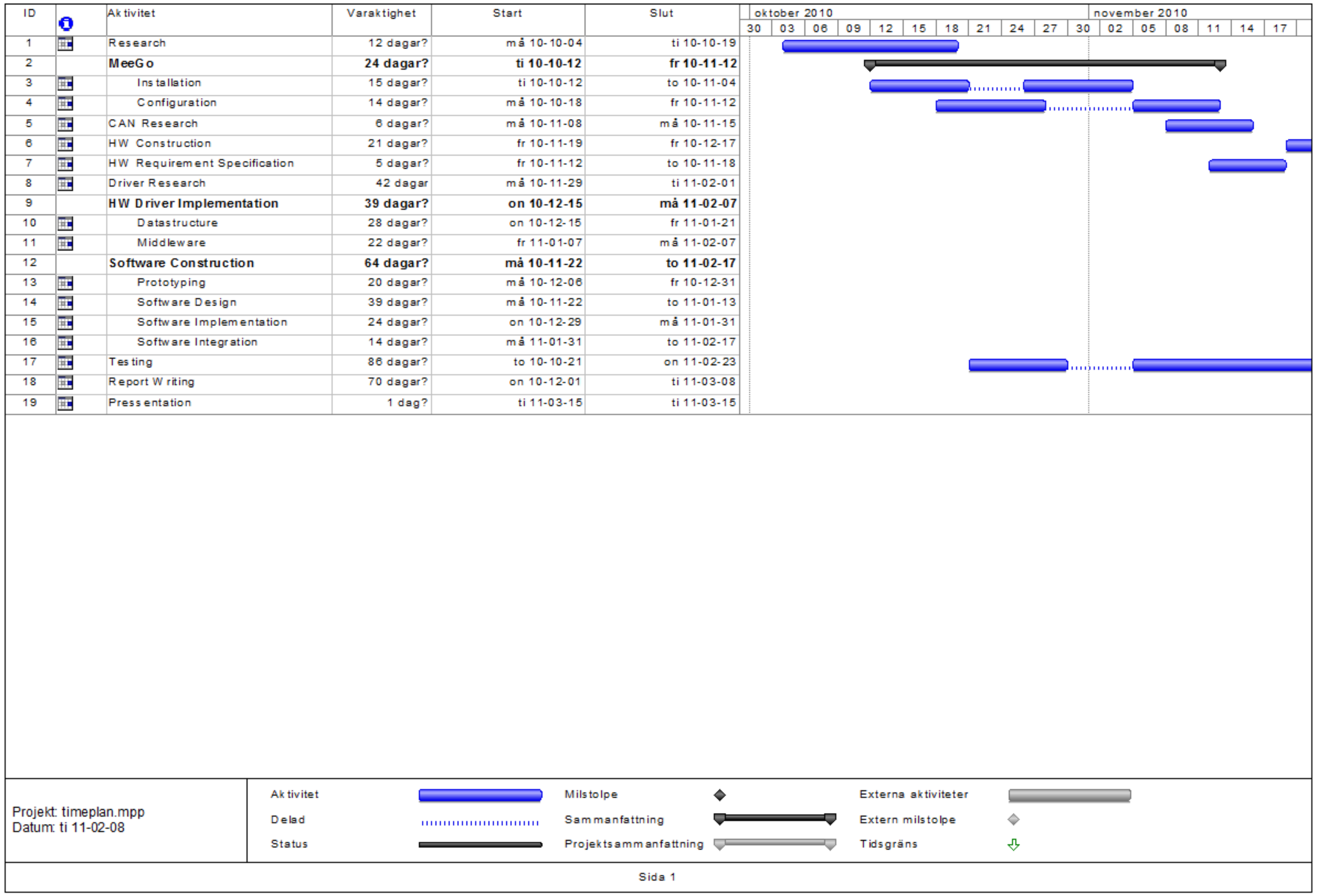
21. GumStix Overo Fire COM. [Online] gumstix, 2010. [Cited: 02 18, 2011.] http://www.gumstix.com/store/catalog/product_info.php?products_id=227.

22. MCP2551 CAN transceiver. [Online] 2003. http://ww1.microchip.com/downloads/en/devicedoc/21667d.pdf.

23. LG 4.3 inch touch screen. [Online] gumstix, 2010. [Cited: 02 18, 2011.] http://gumstix.com/store/catalog/product_info.php?products_id=244.

9. APPENDIX

9.1. TIME PLAN

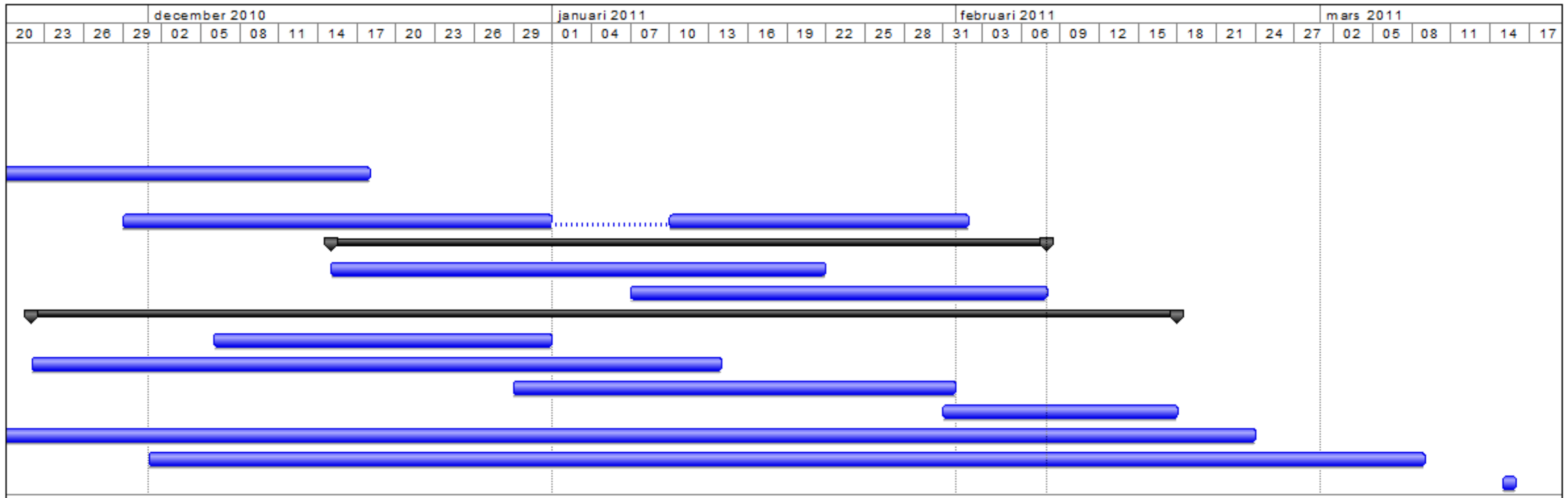


ID		Aktivitet	Varaktighet	Start	Slut
1	▦	Research	12 dagar?	må 10-10-04	ti 10-10-19
2		**MeeGo**	**24 dagar?**	**ti 10-10-12**	**fr 10-11-12**
3	▦	Installation	15 dagar?	ti 10-10-12	to 10-11-04
4	▦	Configuration	14 dagar?	må 10-10-18	fr 10-11-12
5	▦	CAN Research	6 dagar?	må 10-11-08	må 10-11-15
6	▦	HW Construction	21 dagar?	fr 10-11-19	fr 10-12-17
7	▦	HW Requirement Specification	5 dagar?	fr 10-11-12	to 10-11-18
8	▦	Driver Research	42 dagar	må 10-11-29	ti 11-02-01
9		**HW Driver Implementation**	**39 dagar?**	**on 10-12-15**	**må 11-02-07**
10	▦	Datastructure	28 dagar?	on 10-12-15	fr 11-01-21
11	▦	Middleware	22 dagar?	fr 11-01-07	må 11-02-07
12		**Software Construction**	**64 dagar?**	**må 10-11-22**	**to 11-02-17**
13	▦	Prototyping	20 dagar?	må 10-12-06	fr 10-12-31
14	▦	Software Design	39 dagar?	må 10-11-22	to 11-01-13
15	▦	Software Implementation	24 dagar?	on 10-12-29	må 11-01-31
16	▦	Software Integration	14 dagar?	må 11-01-31	to 11-02-17
17	▦	Testing	86 dagar?	to 10-10-21	on 11-02-23
18	▦	Report Writing	70 dagar?	on 10-12-01	ti 11-03-08
19	▦	Pressentation	1 dag?	ti 11-03-15	ti 11-03-15

Projekt: timeplan.mpp
Datum: ti 11-02-08

Aktivitet	▬	Milstolpe	◆	Externa aktiviteter	▬
Delad	┈┈┈	Sammanfattning	▬	Extern milstolpe	◆
Status	▬	Projektsammanfattning	▬	Tidsgräns	⇩

Sida 1

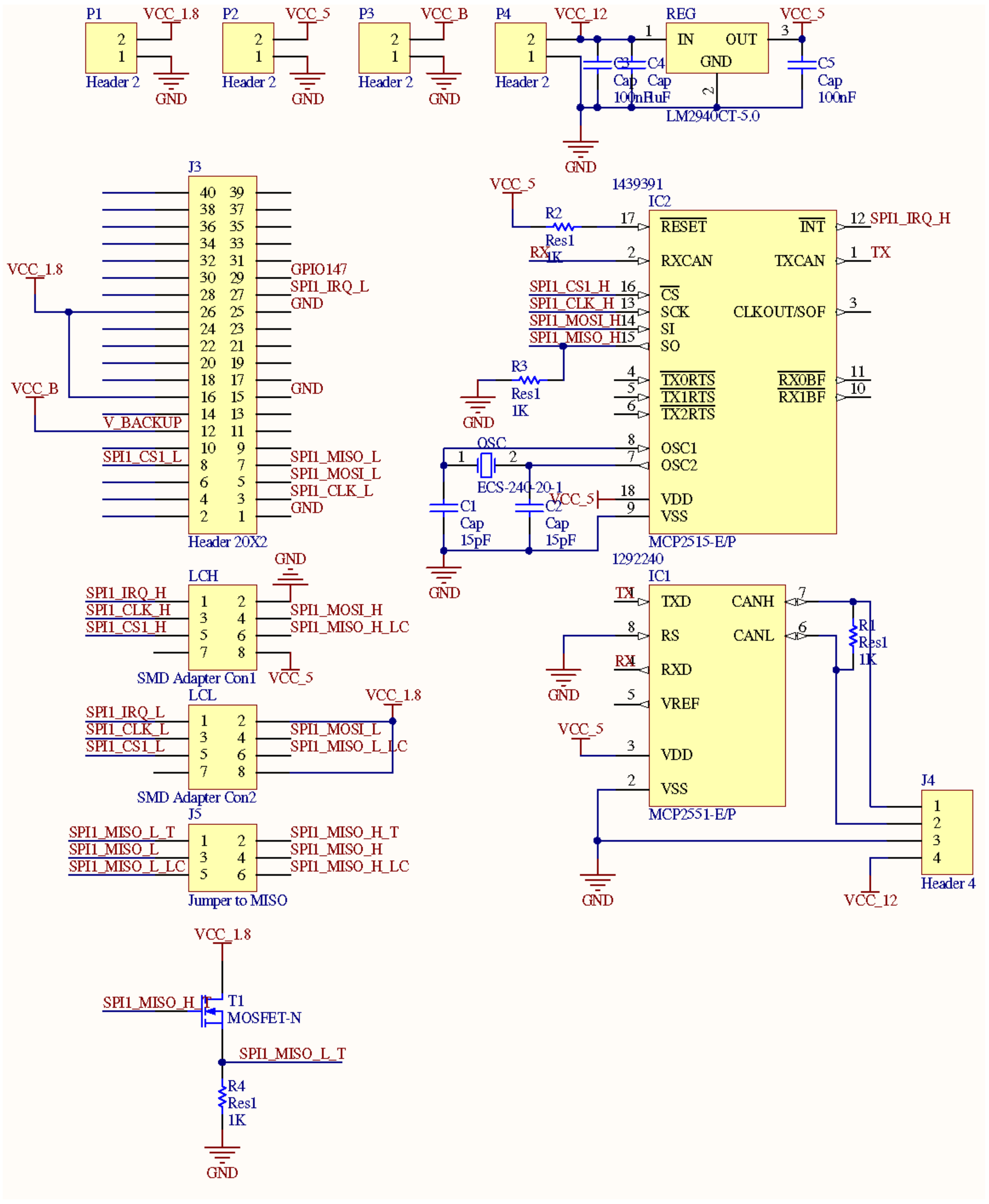


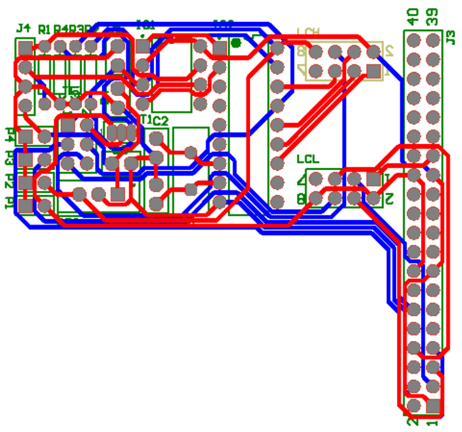
Projekt: timeplan.mpp
Datum: ti 11-02-08

Aktivitet	▬	Milstolpe	◆	Externa aktiviteter	▬
Delad	┈	Sammanfattning	▬	Extern milstolpe	◇
Status	▬	Projektsammanfattning	▬	Tidsgräns	⇩

Sida 2

9.2. CIRCUIT BOARD

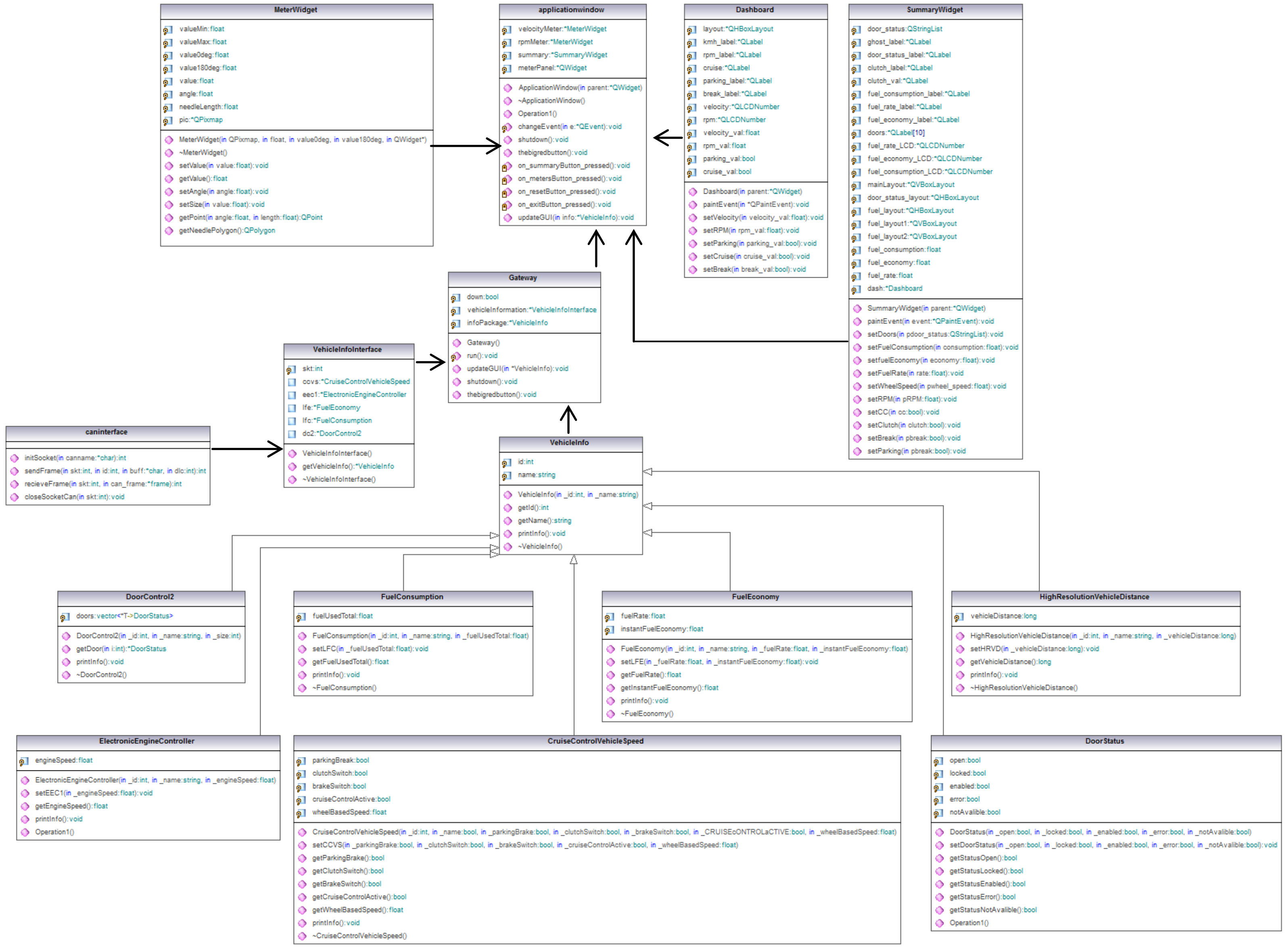




9.3. LIST OF HARDWARE

- GumStix Overo Fire COM (21)
- Overo expansion board cheastnut43 (3)
- Saleae Logic (18)
- MCP2515 – CAN controller (13)
- MCP2551 – CAN transceiver (22)
- LG 4.3 inch LCD touch screen (23)
- TXB0106 – Level converter (19)

9.4. CLASS DIAGRAM


MeterWidget
valueMin:float
valueMax:float
value0deg:float
value180deg:float
value:float
angle:float
needleLength:float
pic:*QPixmap
MeterWidget(in QPixmap, in float, in value0deg, in value180deg, in QWidget*)
~MeterWidget()
setValue(in value:float):void
getValue():float
setAngle(in angle:float):void
setSize(in value:float):void
getPoint(in angle:float, in length:float):QPoint
getNeedlePolygon():QPolygon
applicationwindow
velocityMeter:*MeterWidget
rpmMeter:*MeterWidget
summary:*SummaryWidget
meterPanel:*QWidget
ApplicationWindow(in parent:*QWidget)
~ApplicationWindow()
Operation1()
changeEvent(in e:*QEvent):void
shutdown():void
thebigredbutton():void
on_summaryButton_pressed():void
on_metersButton_pressed():void
on_resetButton_pressed():void
on_exitButton_pressed():void
updateGUI(in info:*VehicleInfo):void
Dashboard
layout:*QHBoxLayout
kmh_label:*QLabel
rpm_label:*QLabel
cruise:*QLabel
parking_label:*QLabel
break_label:*QLabel
velocity:*QLCDNumber
rpm:*QLCDNumber
velocity_val:float
rpm_val:float
parking_val:bool
cruise_val:bool
Dashboard(in parent:*QWidget)
paintEvent(in *QPaintEvent):void
setVelocity(in velocity_val:float):void
setRPM(in rpm_val:float):void
setParking(in parking_val:bool):void
setCruise(in cruise_val:bool):void
setBreak(in break_val:bool):void
SummaryWidget
door_status:QStringList
ghost_label:*QLabel
door_status_label:*QLabel
clutch_label:*QLabel
clutch_val:*QLabel
fuel_consumption_label:*QLabel
fuel_rate_label:*QLabel
fuel_economy_label:*QLabel
doors:*QLabel[10]
fuel_rate_LCD:*QLCDNumber
fuel_economy_LCD:*QLCDNumber
fuel_consumption_LCD:*QLCDNumber
mainLayout:*QVBoxLayout
door_status_layout:*QHBoxLayout
fuel_layout:*QHBoxLayout
fuel_layout1:*QVBoxLayout
fuel_layout2:*QVBoxLayout
fuel_consumption:float
fuel_economy:float
fuel_rate:float
dash:*Dashboard
SummaryWidget(in parent:*QWidget)
paintEvent(in event:*QPaintEvent):void
setDoors(in pdoor_status:QStringList):void
setFuelConsumption(in consumption:float):void
setfuelEconomy(in economy:float):void
setFuelRate(in rate:float):void
setWheelSpeed(in pwheel_speed:float):void
setRPM(in pRPM:float):void
setCC(in cc:bool):void
setClutch(in clutch:bool):void
setBreak(in pbreak:bool):void
setParking(in pbreak:bool):void
Gateway
down:bool
vehicleInformation:*VehicleInfoInterface
infoPackage:*VehicleInfo
Gateway()
run():void
updateGUI(in *VehicleInfo):void
shutdown():void
thebigredbutton():void
VehicleInfoInterface
skt:int
ccvs:*CruiseControlVehicleSpeed
eec1:*ElectronicEngineController
lfe:*FuelEconomy
lfc:*FuelConsumption
dc2:*DoorControl2
VehicleInfoInterface()
getVehicleInfo():*VehicleInfo
~VehicleInfoInterface()
caninterface
initSocket(in canname:*char):int
sendFrame(in skt:int, in id:int, in buff:*char, in dlc:int):int
recieveFrame(in skt:int, in can_frame:*frame):int
closeSocketCan(in skt:int):void
VehicleInfo
id:int
name:string
VehicleInfo(in _id:int, in _name:string)
getId():int
getName():string
printInfo():void
~VehicleInfo()
DoorControl2
doors:vector<*T->DoorStatus>
DoorControl2(in _id:int, in _name:string, in _size:int)
getDoor(in i:int):*DoorStatus
printInfo():void
~DoorControl2()
FuelConsumption
fuelUsedTotal:float
FuelConsumption(in _id:int, in _name:string, in _fuelUsedTotal:float)
setLFC(in _fuelUsedTotal:float):void
getFuelUsedTotal():float
printInfo():void
~FuelConsumption()
FuelEconomy
fuelRate:float
instantFuelEconomy:float
FuelEconomy(in _id:int, in _name:string, in _fuelRate:float, in _instantFuelEconomy:float)
setLFE(in _fuelRate:float, in _instantFuelEconomy:float):void
getFuelRate():float
getInstantFuelEconomy():float
printInfo():void
~FuelEconomy()
HighResolutionVehicleDistance
vehicleDistance:long
HighResolutionVehicleDistance(in _id:int, in _name:string, in _vehicleDistance:long)
setHRVD(in _vehicleDistance:long):void
getVehicleDistance():long
printInfo():void
~HighResolutionVehicleDistance()
ElectronicEngineController
engineSpeed:float
ElectronicEngineController(in _id:int, in _name:string, in _engineSpeed:float)
setEEC1(in _engineSpeed:float):void
getEngineSpeed():float
printInfo():void
Operation1()
CruiseControlVehicleSpeed
parkingBreak:bool
clutchSwitch:bool
brakeSwitch:bool
cruiseControlActive:bool
wheelBasedSpeed:float
CruiseControlVehicleSpeed(in _id:int, in _name:bool, in _parkingBrake:bool, in _clutchSwitch:bool, in _brakeSwitch:bool, in _CRUISEcONTROLaCTIVE:bool, in _wheelBasedSpeed:float)
setCCVS(in _parkingBrake:bool, in _clutchSwitch:bool, in _brakeSwitch:bool, in _cruiseControlActive:bool, in _wheelBasedSpeed:float)
getParkingBrake():bool
getClutchSwitch():bool
getBrakeSwitch():bool
getCruiseControlActive():bool
getWheelBasedSpeed():float
printInfo():void
~CruiseControlVehicleSpeed()
DoorStatus
open:bool
locked:bool
enabled:bool
error:bool
notAvalible:bool
DoorStatus(in _open:bool, in _locked:bool, in _enabled:bool, in _error:bool, in _notAvalible:bool)
setDoorStatus(in _open:bool, in _locked:bool, in _enabled:bool, in _error:bool, in _notAvalible:bool):void
getStatusOpen():bool
getStatusLocked():bool
getStatusEnabled():bool
getStatusError():bool
getStatusNotAvalible():bool
Operation1()

9.5. MCP2515 SETUP

```
#if defined(CONFIG_CAN_MCP251X) || \
   defined(CONFIG_CAN_MCP251X_MODULE)

#include <linux/can/platform/mcp251x.h>

#define OVERO_GPIO_CAN0_INT     146
#define OVERO_GPIO_SPI1_CS1     175
#define INSTRUCTION_RESET       0xC0

static int mcp251x_setup(struct spi_device *spi)
{
       printk(KERN_INFO "Running: mcp251x_setup, Reseting...");
       u8 spi_tx = INSTRUCTION_RESET;
       gpio_set_value(OVERO_GPIO_SPI1_CS1, 1);
       msleep(1);
       gpio_set_value(OVERO_GPIO_SPI1_CS1, 0);
       msleep(1);
       if(spi_write(spi, &spi_tx, 1) < 0)

              printk(KERN_INFO "mcp251x Reset: Error");
       else
              printk(KERN_INFO "mcp251x Reset: Successful");

       msleep(1);
       gpio_set_value(OVERO_GPIO_SPI1_CS1, 1);
       return 0;
}

static struct omap2_mcspi_device_config overo_mcp251x_mcspi_config =
{
      .turbo_mode     = 0,
      .single_channel = 1,    /* 0: slave, 1: master */
};

static struct mcp251x_platform_data overo_mcp251x_config = {
    .oscillator_frequency = 20 * 1000 * 1000,
    .board_specific_setup = &mcp251x_setup,
    .model                 = CAN_MCP251X_MCP2515,
    .power_enable          = NULL,
    .transceiver_enable    = NULL,
};

static void __init overo_mcp251x_init(void)
{
    /* printk(KERN_INFO "overo_mcp251x_init(): requesting
interrupt\n"); */
```

```
    if ((gpio_request(OVERO_GPIO_CAN0_INT, "MCP251X_0_INTERRUPT") ==
0) &&
           (gpio_direction_input(OVERO_GPIO_CAN0_INT) == 0)) {
        gpio_export(OVERO_GPIO_CAN0_INT, 0);
    } else {
         printk(KERN_ERR "overo_mcp251x_init(): could not obtain
gpio for MCP251X_INTERRUPT\n");
        return;
    }
}

#else
static inline void __init overo_mcp251x_init(void) { return; }
#endif

static int __init overo_spi_init(void)
{
#if defined(CONFIG_CAN_MCP251X) || \
        defined(CONFIG_CAN_MCP251X_MODULE)
        overo_mcp251x_init();
#endif


//next initialization
```