CHALMERS

A software architecture for embedded telematics devices on Linux

Master of Science Thesis in Secure and Dependable Computer Systems

HUGO HOLGERSSON

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

A software architecture for embedded
telematics devices on Linux

Examiner: Johan Karlsson, johan@chalmers.se
Professor of Dependable and Robust Real-Time Systems
Department of Computer Science and Engineering

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

Abstract

This master thesis proposes a Linux-based software architecture for modern telematics devices. Our device is installed in trucks and has the purpose of forwarding collected GPS data to an online data center in a robust way. Telematics units are are highly concurrent systems where a crucial part of the software architecture is the communication schema: Within the system, sensor data need to be independently gathered and processed. Relevant information is then presented to the truck driver or forwarded to an online data center.

We investigate D-Bus, an IPC system that is mostly used in *desktop* Linux systems, as a potential replacement for traditonal, in-kernel message passing. When moving to more elaborated embedded hardware platforms, such higher level programming paradigms and protocols become more attractive than pure low level programming. This thesis highlights the possibility of incorporating such techniques also in *embedded* Linux systems.

We demonstrate an event-driven system architecture, implemented using the Qt C++ framework. D-Bus is used for the system's interprocess communication, in contrast to TCP sockets that were used in the previous product. We make low level sensor data available via D-Bus by two layers of abstraction: a Linux I2C chip driver and Linux daemon using D-Bus.

Contents

Acronyms

ABI	Application Binary Interface
ADK	Application Development Kit
API	Application Programming Interface
CAN	Controller Area Network
D-Bus	FreeDesktop's "Desktop Bus" for IPC
ELF	Executable and Linkable Format
GCC	GNU Compiler Collection
GDB	GNU Project Debugger
GPS	Global Positioning System
GUI	Graphical User Interface
I2C	2-wire Inter-IC bus
IC	Integrated Circuit
IDE	Integrated Development Environment
IVI	In-vehicle Infotainment
JTAG	Joint Test Action Group (JTAG) standard test access port
PDK	Product Development Kit
PDU	Protocol Data Unit
POSIX	Portable Operating System Interface
Qt	A cross-platform application C++ framework
ROM	Read-Only Memory
SDK	Software Development Kit
SoC	System on Chip
UML	Unified Modeling Language
XIP	Execute in place
XML	Extensible Mark-up Language

Chapter 1

Introduction

With Internet access everywhere new possibilities and markets has emerged in the field of wireless surveillance. Any type of vehicles - whether it is a personal car, an ambulance, a bus in public transport or a normal truck - can nowadays be connected to the Internet thanks to the nation wide cellular phone networks.

A steady Internet connection enables fleet operators to monitor their vehicles' positions in real time. By automatically reporting data, time is also saved for the truck driver, who does no longer need to report his locations and work hours manually. A such system, designed to report information on geographically sparse units, is commonly refereed to as a *telematics* system. Telematics systems designed for fleet management can globally be seen as a combination of three subsystem that co-operates over the Internet:

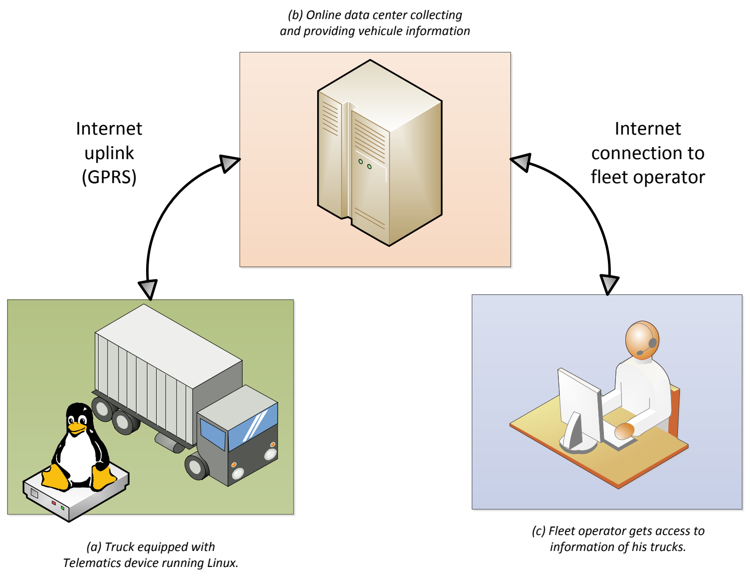
(b) Online data center collecting and providing vehicule information
Internet uplink (GPRS)
Internet connection to fleet operator
(a) Truck equipped with Telematics device running Linux.
(c) Fleet operator gets access to information of his trucks.

Figure 1.1: A typical telematics system for vehicle surveillance.

Each one of the surveilled vehicles is equipped with an embedded computer system (a). The embedded system runs on a customized hardware board that could either be factory-fitted or integrated into the vehicle as an add-on. In any case, these configurations does often include a GPS receiver and a mobile data modem that enables the system to connect to the Internet. Through the modem connection, each of these embedded systems are uplinked to the central server infrastructure (b) to which data concerning the vehicle is uploaded. The back-office server side organizes the data and groups the information according the requirements set out by the fleet that the vehicle belongs to.

These servers do also have an interface towards the fleet's administration department (c) by which fleet operators can benefit from the collected information presented by desktop/phone software applications. Notably, each individual vehicle's geographical location and ongoing mission can be visualized on maps - an excellent tool to simplify fleet administration and logistics.

This report focuses on the design aspects of the onboard (a) computer system, residing inside the vehicles. Often, these systems do simultaneously support some kind of user interaction, along side the functionality of telematics, forming an *In-Vehicle-Infotainment* system.

Due to the implied safety concerns involved in designing systems used in and connected to vehicle, developing IVIs is more difficult than developing personal devices, such as mobile phones; without violating the drivers' safety, these devices do need to use the CAN bus in order to read the out values on for example velocity, charge and temperature.

When designing an embedded software system, it is nowadays common to rely on, already freely available open-source components. Linux, a free general purpose operating system, has become more and more common in the embedded segment the last ten years. FreeElectrons, an embedded Linux consultant company, motivates this success by the ability to re-use components that has been proved to work well, without losing control over the software. Since Linux is free, it can be duplicated to any number of devices, without additional licensing costs which make Linux a potential solution upon which IVIs could be designed.

R. Streif, director of embedded solutions for The Linux Foundation argues that for any Internet-connected consumer device, 95% of its functionality will be similar to other devices of the same type. By relaying in open source technology for this part, more effort and cost can be put on the fine-tuning the product, notably its user experience.

In order to enrich user experience, an IVI system is expected to be extendable with new user applications. A bare Linux system does not have these abilities built-in which was the reason why for example Google Android was designed. Android adds several *middleware services*, on top of Linux that enables further user customization.

We have in this thesis investigated how to extend Linux in a similar way by adding two supporting middleware services that helps developers to (1) benefit from the connected hardware found in the vehicle, as well as (2) design applications that can rely on the online servers. In order for application to benefit from these services, a way of *inter process communication* is needed. This report demonstrates how the Linux/FreeDesktop D-Bus can be used in an embedded system as a way to let middleware services export functionality to the on-top applications.

This report is organized as follows: chapter 2 describes the requirements set out by the company, chapter 3 gives an overview of the proposed solution, chapter 4 details the middleware that gives access to the server infrastructure, chapter 5 describes the foundations behind the middleware that gives hardware abstractions. The final section draw conclusions. Three appendices are given with more information on D-Bus and the development of embedded Linux systems.

Chapter 2

Specification and requirements

Apart from keeping the existing, basic telematics functionality (described in the first chapter) the company gave a few explicit requirements concerning the next product. This chapter describes the meanings of these requirements as well as the main principles of the former product to which we need to remain compatible.

2.1 Requirements

When drawing the next generation product, the following requirements were taken into consideration. The new product needs to:

(a) Comply to the given telematics specification.

(b) Be compatible with the current server infrastructure.

(c) Maintain existing connectivity to vehicle peripherals.

(d) Run the current GUI software presented to vechicule driver.

Requirement (b) implies that the external (proprietary) communication protocol needs to be kept. By keeping the old binary protocol, the new embedded system will be backward-compatible with the existing sever infrastructure. When designing the new product, focus has thus rather been put on improving the embedded system's internal communication but in a way that is transparent to external communication.

Telematics specification The purpose of the embedded Linux system, also shown in figure **??**, is (a) is to gather data from the vehicle's sensors and report this information to a central server on the Internet.

Linux daemons (system *services*) upload data to the online data center (b) whose database is storing information about all registered vehicles. At regular time intervals, each service will upload data, concerning for example the vehicle's current position and internal temperatures.

From the customers point of view, gathered data concerning to their fleet is made available either by a web interface or by a Windows-based software (c).

In-vehicle system A graphical user interface, similar to a common GPS, is presented on a screen next to the truck driver. The current GUI program was formerly run on an external device; a device with an independent operating system. In the upcoming product a dumb frame buffer screen, directly connected to the main Linux system will be used (this avoids the need of a second system with the sole purpose of providing the GUI).

The GUI is designed around the Qt framework and is intended to be kept also in the future system versions. In a software requirement perspective, this implies that the Qt framework also needs to be presented on the future in-vehicle Linux system.

2.2 Implementation of previous product

The previous product was studied as an example implementation of an embedded telematics device. Since the new product needs to be compatible with the existing server infrastructure a good understanding of the former product proved to be indispensable.

2.2.1 Server connection

External communication is the communication directed/origin from outside the embedded device. Notably, to and from the external TCP server and from the surrounding sensors. Each device manages all external communication through a TCP-based application named *MultiEvent*.

MultiEvent handles the communication in both directions, that is, both the information coming into the device, such as administration requests, and information that is streamed out from the boxes, like the GPS positions. MultiEvent could have been called *gateway*, *firewall* or simply *middleware*, as it incorporates all these functionalities:

MultiEvent multiplexes incoming messages towards the appropriate service. Having only one TCP port exposed to Internet for the incoming connections, the software roughly acts as a filtering firewall: it investigates incoming packages and dispatches them to the appropriate service. In the other direction, when MultiEvent receives sensor informations from the sensor daemons, it will act as a gateway and thus help the services to transfer information gathered in a format recognizable by the online data center.

External communication is expensive since all traffic is debited by the Internet provider. In order to avoid uploading upon each sensor emission, which would imply more transfer overhead and thus more cost, MultiEvent buffers sensor data that are sent in intervals of 10 seconds. This keeps the external communication of outbound packages buffered by MultiEvent in a controlled pace. The buffer also serves a backup where data are kept as long as no Internet connection is available.

2.2.2 Access to vehicle peripherals

Each sensor and peripherals is controlled by a Linux driver. The driver lets an user-space service, similar to a Linux daemon, retrieve the sensor's value at regular intervals.

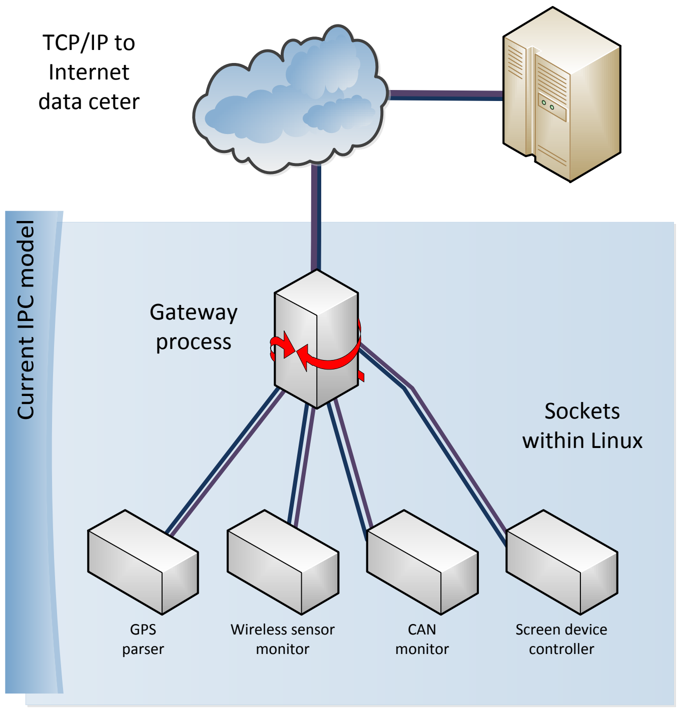

Figure 2.1: All communication, both external and internal, from and within the current embedded Linux system, are done via TCP connections.

There is for example one service dedicated to monitor the vehicles geographical position, another one is monitoring the temperatures.

The internal, interprocess communication among MultiEvent and the underlaying services is managed by a set of internal TCP-type connections. Each service communicates with MultiEvent on two, steadily open connection sockets called "data" and "messaging". When a service starts, it will register itself both on MultiEvent's data socket and MultiEvent's message socket. The created TCP connections are marked as purple/blue lines between the service and MultiEvent in figure **??**.

The data socket is used by the service when it is time to report sensor information. The traffic on this channel is thus unidirectional and asynchronous; it is always sent by the service and always received by MultiEvent for further transmission externally.

Each service listens to the messaging socket in order to discover incoming administration requests. If the request demands some status information, the service will answer on this socket. Consequently, this socket transfers data in both ways: it is used to send administrative commands and possibly also to send urgent information, such as AirBag information, that needs to be treated in real time, without being kept in a buffer.

Chapter 3

Software architecture

This chapter describes the proposed software architecture and its principal components. The new software architecture is centralized around the FreeDesktop D-Bus. All services will henceforth use the D-Bus to export functionality and perform IPC.

3.1 Architectural overview

The FreeDesktop D-Bus is used for all IPC communication in the new system. In contrast to sockets, D-Bus messages are dispatched by an user-space bus daemon. The D-Bus daemon is not a standard component of the Linux system, but is found in practically all desktop Linux distributions. A more detailed description of D-Bus and a comparison to TCP sockets can be found in Appendix A.

D-Bus handles the encapsulation and the binary data transmission between processes. As seen in the conceptual overview, the D-Bus daemon is an user-space software bus (a dispatcher) that is able to connect any program in Linux user space [1].

[1]Even though not demonstrated in this implementation, we could for example imagine a situation where we want to allow the CoProcessor service to reach the Internet data center via the Gateway service.

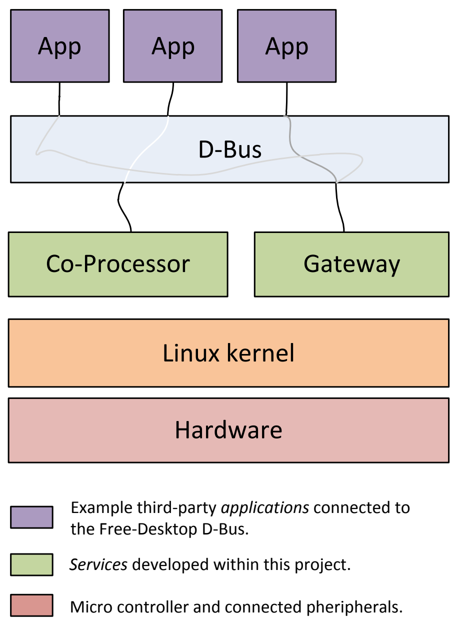
App
App
App
D-Bus
Co-Processor
Gateway
Linux kernel
Hardware
Example third-party *applications* connected to the Free-Desktop D-Bus.
Services developed within this project.
Micro controller and connected pheripherals.

Figure 3.1: Alignment of architecture on top of the Linux kernel's abstraction stack.

3.2 System services

Using D-Bus as a central software hub makes it is easy to export functionality from any *service*. Two example services, representing two crucial features in a telematics system, has been implemented to demonstrate the new architecture. Figure **??** shows how these middleware services are added in user space, on top of the Linux kernel: first, a redesigned middleware *Gateway* to let applications access the online data center. Second, a service, *CoProcessor*, providing an abstraction of a helping microcontroller. *CoProcessor* was developed on top of Linux's I2C driver framework.

Internet gateway D-Bus is used in the proposed software architecture by a gateway/middleware service that dispatches and receives data to/from Linux applications that need connectivity to the Internet server. The illustration shows how the D-Bus forms an essential component, bridging the uplinked middleware *service* with the system's *applications*. The D-Bus daemon is depicted as bus even though it is pure software concept.

The former product, previously described in section 2.2, uses one single TCP connection for both inbound and outbound messages and data. This imposes us to keep the inbound (firewall) and the outbound (gateway) part of the external communication combined into one bi-directional middleware process. The protocol of external communication needs to be kept as it is, according to requirements.

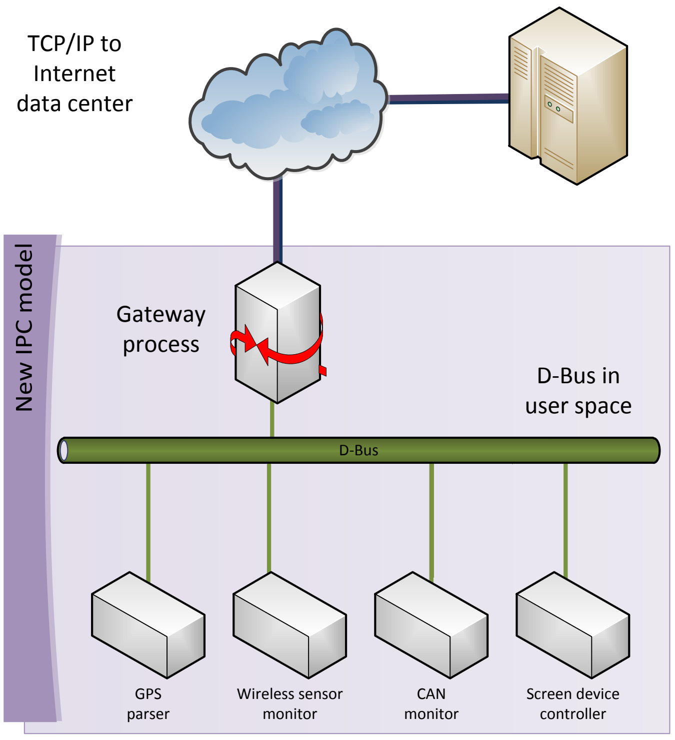

Figure 3.2: D-Bus is used for the IPC between services and applications. TCP is still used to talk to with the online data center. Compare this to the old architecture where sockets also were used internally for IPC.

Hardware abstractions In addition to the new middleware, another service was implemented to forward the functionality provided by a hardware co-processor (described in chapter 5).

Instead of letting each application access the Linux driver directly (by the character device file `/dev/stm32`) the *CoProcessor* service was designed for this sole task; it is meant to be the only program that directly uses the STM32 Linux driver.

We found the advantages of this design to be:

- Hardware access can be denied/allowed for single applications (if the D-Bus daemon is configured correctly).
- Applications use one common IPC system and will not be "polluted" with low-level driver calls. All applications rely on API's that are accessible in the same manner.
- The driver implementation can be simpler. Being sure that only one program uses the driver, the driver can be implemented without taking care of thread/synchronization problems[2].

[2]This saves initial implementation time but is not recommended as a long-term solution.

3.3 Exported XML interface

One of the main principles behind the software architecture is to let all services export their all functionality via the FreeDesktop D-Bus. The recommended way to define a D-Bus interface is by XML files. [?] These XML files can later on by used to automatically create skeleton classes to use in the implementation.

Services export their API via D-Bus through a mechanism called *proxy objects*. The XML file describes the interface of these objects. Both the exporting service and the (client) applications need to agree on the interface in order to communicate over D-Bus: clients will only be able to use the exported functions, no more, no less. Appropriately, the service is obliged to export all methods described by the XML.

Internet gateway All applications that need to access the Internet server have to pass through *Gateway*. The following XML markup describes the D-Bus interface of *Gateway*. The interface defines the exact functionality that is offered: clients can use the functions getMsg(), uploadData(), uploadMsg() and listen for the msgArrived-signal.

If a message arrives to the channel corresponding to the one of a particular client, its message can be retrieved using the getMsg() function. The upload-functions are called when an application has data to transit online, for example, sensor data or messages about the vehicle's condition. All the messages being passed are bundled in Qt's own type of byte arrays, QByteArray.

Code 1 XML defines the interface to the Gateway service.

```
<node>
  <interface name="com.oktalogic.gateway">
    <signal name="msgArrived">
        <arg name="channel" type="y" direction="out"/>
    </signal>

    <!-- ay means QByteArray -->
    <method name="getMsg">
      <arg name="channel" type="y" direction="in" />
      <arg name="msgData" type="ay" direction="out" />
    </method>

    <method name="uploadData">
      <arg name="msgData" type="ay" direction="in" />
    </method>
    <method name="uploadMsg">
      <arg name="msgData" type="ay" direction="in" />
    </method>

  </interface>
</node>
```

Hardware abstractions Applications in need of a specific hardware feature from the co-processor can use the interface it exports via D-Bus. The service makes an abstraction of the raw commands that are sent to Linux kernel. Having an extra abstraction let us keep the driver simple; the Linux kernel driver will only need to forward the commands (strings of bytes) that are generated by the *CoProcessor* service.

Consequently, when new commands are supported by the co-processor's firmware, *CoProcessor* and its XML interface needs to be updated but not the actual driver [3]. The following XML markup describes the D-Bus interface of the *CoProcessor* service.

Code 2 XML defines the interface to the CoProcessor service.

```
<node>
  <interface name="com.oktalogic.coprocessor">

    <!-- ay means QByteArray -->
    <method name="firmwareVersion">
          <arg name="versionString" type="ay" direction="out" />
    </method>

    <method name="feedWatchdog">
    </method>

  </interface>
</node>
```

Our prototype version demonstrates two, very simple, example functions of the co-processor. It was out of the scope of this project to define the exact functionality and relation between the two microcontrollers.

[3]The Linux driver framework provides us with an interface to the physical I2C system but will not care about what kind of data is being transmitted. See chapter 5.

3.4 The Qt framework

The Qt framework was an explicit requirement of the graphical user interface, presented to the vehicle's driver. Having the framework available on target also makes other user space applications able to benefit from the library. Previously, all programs were purely based on operating system or POSIX primitives. The two proposed services, *Gateway* and *CoProcessor* do both rely on Qt's framework. Qt is designed with two essential paradigms in mind:

- Object oriented programming
- Event-driven programming

Qt includes D-Bus language *bindings* [4] that make it possible to use the D-Bus objects as native C++/Qt-objects. The XML files, describing the D-Bus objects, are read by the `qdbusxml2cpp` program of Qt in order to generate two skeleton classes:

Adaptor Class used by a middleware service to receive incoming D-Bus calls according to the defined interface.

Proxy Class used by any application that wants to use the object exported by the service. The proxy class translates native C++ method invocations into D-Bus calls, directed to the concerned service.

Chapter 4, "Internet gateway design", describes in more detail how these paradigms are implemented practically in a real Qt-application using D-Bus.

[4]Qt's bindings depend on libdbus. It is possible to write programs directly relying on libdbus, but it is highly discouraged by both the Qt and the D-Bus developer communities. One developer stated that "about 2000 lines of code" would be needed if not using a library binding (such as Qt or Glib).

Chapter 4

Internet gateway design

We propose a new object oriented design of the middleware between the server infrastructure and the in-vehicle Linux system. The new middleware fulfills the same functional requirements as the former implementation but has been completely rewritten in C++.

The middleware, simply called *Gateway*, relies on the event-driven paradigm to avoid unnecessary use of threads. Its interface to connected Linux applications has been simplified by using Qt's D-Bus bindings.

4.1 Software modules

The middleware's C++ classes can be categorized into three different modules; `External connection`, `D-Bus` and `Storage` that co-operate as demonstrated in the below UML diagram. The `Storage` module, drawn in blue, works in two directions; is used from both the server-side, representing the online data center, and the D-Bus-side, representing the local applications run in Linux.

`Storage` is an essential part of the middleware: it buffers both incoming and outgoing messages from/to the server before they can be transmitted towards the Internet. While waiting for transmission, all message types are buffered by the this module. Messages to and from Linux applications travel through the D-Bus. Messages to and from Internet use the proprietary protocol, found in this application, on top of TCP/IP.

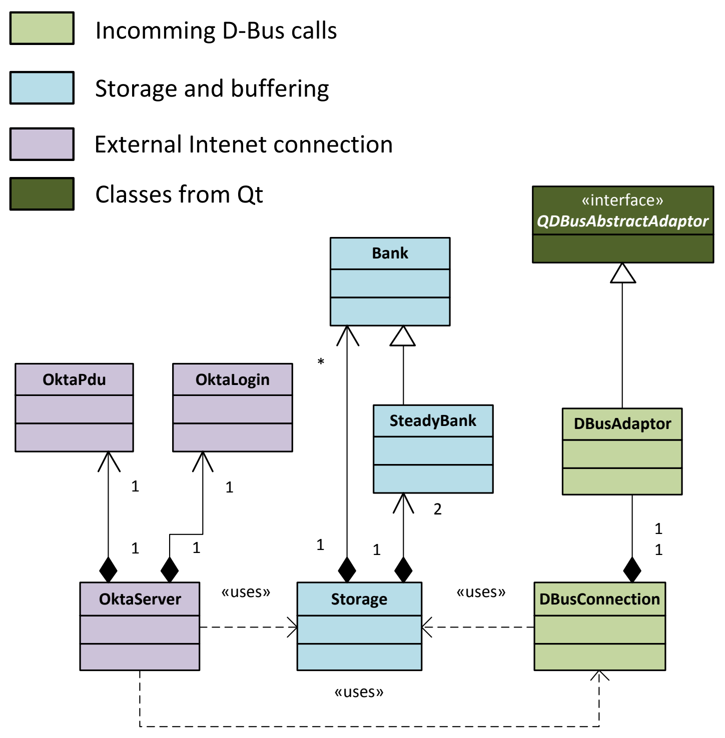

Figure 4.1: UML diagram depicting all classes of the Gateway application. The schema is simplified, only the classes and their relations are shown. All methods and properties are left out.

4.1.1 D-Bus

Theses classes are handling IPC connectivity using Qt's D-Bus bindings. Qt's D-Bus bindings are used to dispatch incoming messages and outbound data/messages from connected Linux applications that need to talk to the Internet server. The D-Bus module is thus used to handle the communication to and from the Linux applications connected to *Gateway* via `D-Bus`. Both transfer directions are treated asynchronously. In order to do this, the classes depend on `Storage` where data are located between calls.

Examples

1. When services want to upload data or messages, functions to receive the outbound data are exported. The data are put into `Storage` for further transmission and treatment by the `External connection`.
2. When a message arrives from the Internet server to an application behind the gateway: in this case, a broadcast is sent over the D-Bus, similar to how it is

done in a "hubbed" Ethernet network. The concerned application, whose identifier matches the receiver field of the incoming package, will then be able to retrieve the message that has been buffered in `Storage` since the moment it was parsed out from the TCP socket.

DBusAdaptor The adaptor defines what methods and properties that will be exported to D-Bus clients. Other programs can transparently call these functions via so called "proxy objects", as if they were native functions.

The `qdbusxml2cpp` tool was used to generate the skeleton of the adaptor-class. The purpose of the adaptor is simply to export native C++ functions and make them available via the D-Bus. The actual implementation of the functions are stored separately, in DBusConnection.

DBusConnection This class implements all functions exported by the D-Bus adaptor. The reason why the class is called DBusConnection because it is also responsibility for setting up the connection to the D-Bus system deamon, running in the background of user space in the Linux system. When connecting to the deamon the name "com.oktalogic.gateway" is given our application, according to D-Bus naming standards.

4.1.2 Storage

This module includes classes that handle *Gateway*'s internal bufferization of inbound and outbound messages. The messages are stored while the services on D-Bus or the Internet data center is not ready to treat the information.

Bank Bank handles a queue of byte streams. These byte streams are temporarily stored within the application. Thanks to the queue, data can be received, buffered and treated continuously.

SteadyBank SteadyBank extends Bank with the feature to backup the structure's data to a file stored in disk/flash memory.

If nobody is able to treat the saved data for the moment, we suppose that somebody will do so in the future. Meanwhile data is saved.

In some rare cases, this future point in time arrives after a system reboot which is why a backup, a serialized "mirror" of the Banks data structure is kept on flash disk.

A backup on disk is also needed in case of Internet connection break down or any another service interruption that causes a restart of the *Gateway* middleware.

If the system needs to be restarted in an abrupt way, while still having stored data (to upload), the data will once again be available after reboot by reading the backup file. The file is easily located since its file name is kept constant.

Until data is extracted (and uploaded), we let the bank, stored both in memory and on disk, grow in size.

BankManager BankManager is a central class that controls all bufferization: it holds the banks of both inbound and outbound byte streams towards the online server infrastructure.

Two SteadyBanks are used for outbound streams (one to store data steams, another one to store messages streams). The outgoing byte streams are already marked with the channel number of the sending Linux application. This number is found among the stream's first bytes of data, the "stream header" [1].

Incoming messages are not saved on the disk because the system relies on the data center that is supposed to re-send messages in case of loss. For these messages, one temporary Bank is given to *each connected application.* The number of connected applications is variable (hence the star-cardinality shown in the UML diagram).

The manager is able to form TCP packages, ready to upload, made out of the buffered outbound data. When the timer hits a new periodic upload of data, the BankManager instance will be called to extract outbound data from the two SteadyBanks and make them available for upload.

4.1.3 External connection

The classes of `External connection` are used to initiate, authenticate and keep the connection to the online data center. The communication is carried out in a non-blocking manner, in line with the event-driven approach taken.

OktaServer OktaServer is the manager of external TCP connection to Internet server. This class guards and monitors the Internet TCP connection. The Internet server sends commands that this class will save in a (temporary) storage. It will periodically upload the data it finds in the outbound part of this storage.

OktaLogin OktaLogin is responsible of launching the TCP handshake and providing username/password to authenticate to server. OktaLogin is launched in the program's initial phase when few other events need to be handled.

This class, in contrary to all others in the complete program, uses some blocking functions. We could off-course also have done a separate state machine, representing the steps of the login process, sticking fully to the asynchronous paradigm of signals/slots. However, this would have forced us to write a quite elaborate state machine for something that is in the end very simple with blocking primitives. In this situation, we tried to be pragmatic in order to gain development time for features that give the product more value.

OktaPdu OktaPdu is a container class that manages incoming data (messages).

When messages arrives on server's socket, this class will let the socket buffer the message until the complete PDU is available. The protocol's header-field called "size" determines the size of the data unit.

[1]Details on the actual data of the protocol is not given due to confidentiality reasons.

While parsing and receiving bytes of an incoming package, this class will keep a notation of the state. The implementation uses the enum+switch/case approach for simplicity. Our state mainly serves the purpose of a marker of location in the current PDU's stream of bytes.

The objective programming pattern "the state pattern" could have been used by putting "each state's behavior in an own class". [?] This would have led to more flexibility some would argue, but to the cost of much more classes. For the moment, we want the proof-of-concept implementation to be kept simple. By having all states and transitions handled in the same class, the code becomes easier to overview. Besides, the application's behavior does not change very much depending on current state (which would otherwise have the reason of further abstraction).

4.2 The event-driven paradigm

Thanks to the use of event-driven programming the communication towards the server could henceforth be carried out in an asynchronous, non-blocking manner. For example, if parts of an incoming command from the server are missing, *Gateway* will let the event loop process other events while "waiting" for the remaining data on the TCP socket. This way, the program will neither explicitly block nor pull in its wait on new data. This allows us to serve events in a more dynamic way: the event loop of Qt will trigger the call back function designated to treat new data once it arrives. Another advantage of using event-driven programming and the fact that events are served momentarily is the that we gain automatic protection of shared resources, such as the server's TCP socket.

However, during implementation, we found out that blocking primitives sometimes actually are easier for the programmer. When using blocking primitives we do not need to manage, a sometimes rather complex state machine that describes the communication. [?] This is why we decided to let the initial authentication code remain blocking (until an error time out occurs). Once logged in on the server, event handlers are (re-)connected to treat the following long term communication asynchronously.

Events within Gateway Events in Qt are sent through "signals" and received and handled in methods called "slots". Behind the scenes, a Qt application implements this mechanism using one central UNIX select() system call. [?] However, for the programmer's point of view, each class appears to do its own event management locally, making the code very straight-forward.

The gateway's three software modules do all depend on one or several events. The following table lists the events that can occur within the each of *Gateway*'s three modules:

- External connection
  - Incoming command message
- Storage
  - Periodic upload of stored messages/data
- D-Bus
  - Outbound data transmission
  - Outbound message transmission
  - Request of stored incoming command message

Motivation In event-driven programming, the only thread running is the thread of the event loop, or the server thread. [?] All events are put in the main event queue and are treated, one by one, by calling its designated event handler. The event handlers, Qt's "slots", are called from this server thread.

This forms a huge contrast in comparison to the legacy design where 2x+1 threads were used [2]. The old product created a new dedicated thread with the sole purpose of monitoring one single TCP connection. As we avoid these threads we will improve the overall performance of the system. Managing a large number of concurrently active threads imposed larger latency, context switching, cache misses and release jitter. [?]

Studies also show that there is typically a maximum number of threads that a given system can support, and beyond which performance degradation occurs. [?]. When keeping a fixed number of threads, we will not risk to reach this limit when the number of clients to the gateway increases. Additionally, as the server thread's stack is unwind after the execution of an event handler we do not risk to have several growing stacks that occupy memory.

Finally, in single-threaded applications the effort of implementation is simplified: we do not need to rely on mechanisms for mutual exclusions and, notably, bother about their possible synchronization issues. [?]

4.3 Security concerns

This middleware does now *export* these features to any application that has access to the bus. This setting can be configured in the D-Bus daemons configuration file and should be restricted in productional software.

[2]On for the data socket and one for the message socket for each connected application as well as the normal main.

Chapter 5

Hardware abstractions

The studied hardware configuration uses two microcontrollers. We demonstrate how to use the I2C bus for the communication between the main processor, running Linux, and the supporting co-processor. A Linux I2C chip driver was thus developed to control the STM32 co-processor over the I2C bus. The STM32 microcontroller itself was programmed with a custom software that is able to receive I2C requests using interrupts.

5.1 Board and peripheral layout

To future hardware board, that will run the new Linux system is equiped with to microcontrollers, seen as two individual *System on Chips*. Both chips are based around (different) ARM cores with additional different peripherals[1]. Our low level software implementation enables the physical I2C circuitries, marked as blue in figure **??**, so the two microcontrollers can communicate on the I2C bus.

The bus is local - it does only exist within the embedded system. Hence, its communication does not interfere with the, safety critical information passed on the vehicle's CAN bus.

5.2 I2C bus principles

The I2C bus was studied as a way to interface external devices. I2C is a simple synchronous serial bus commonly used in embedded systems. For example, chips that deliver sensor data are often equipped with an I2C interface.

The main difference between a "traditional" serial interface and I2C is that I2C is a bus. Just as the USB or PCI it allows us to connect multiple devices on the same physical wires. In this way, the routed wires of the board can be reduced and less pins of the microcontroller will be occupied.

[1] ARM licenses their architectures to several hardware manufacturers whose hardware designers decide what functionality to integrate within the dice.

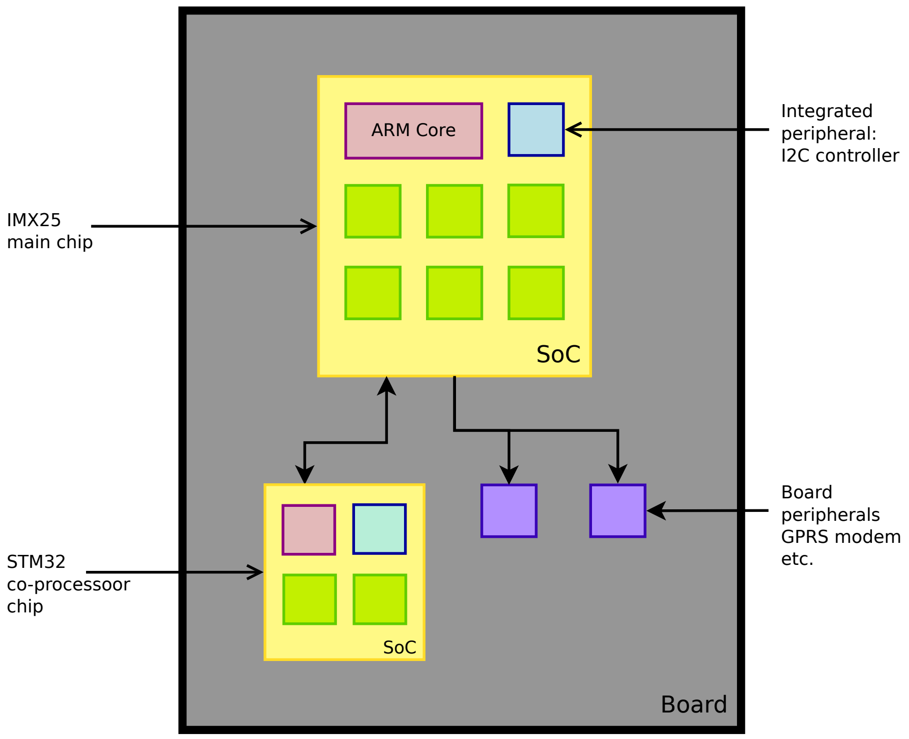
ARM Core
Integrated peripheral: I2C controller
IMX25 main chip
SoC
Board peripherals GPRS modem etc.
STM32 co-processoor chip
SoC
Board

Figure 5.1: Schematic the hardware board and the routed chips.

A bus is characterized by having many devices connected, branched out from a common wire. When many devices are connected to the same physical wires they will all share the same electronic voltage, thus a structured way of sharing the bus is needed. The I2C protocol defines how this is done by a few simple principles:

Each bus has one appointed master node. The master is the only node that can initiate a data transfer. In the proposed architecture the main processor, running Linux, is appointed master. The master can request to either send or to retrieve data from one of the remaining slave devices on the bus. For example, in order to retrieve one byte of data from a slave device, the master will need to send a data request to the slave. [?]

I2C provides a protocol for sending and receiving bytes, but the interface presumes nothing about the contents of the bytes being transferred. For example, in order to inform the slave about what kind of data we we are looking for, a write-request (data transmission *to* slave), containing a hint to the slave about what to deliver next, can be sent before prior the read-request.

The I2C bus does only need two wires, given that the devices share a common ground - which is the case normally the case in an embedded system. One wire, SDA, for the serial data (and slave address information) and SCL as a periodic clock. The clock signal will be used by the slave in order to understand when to sample a bit value from the

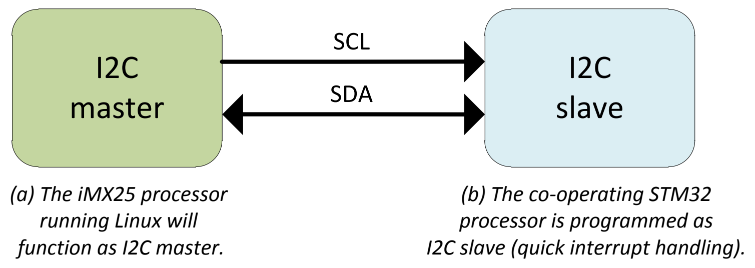
I2C master
SCL
SDA
I2C slave
(a) The iMX25 processor running Linux will function as I2C master.
(b) The co-operating STM32 processor is programmed as I2C slave (quick interrupt handling).

Figure 5.2: Schematic view of the two wires of the I2C bus.

data wire. [?]

The data are thus transferred in serially, according to the speed shown by the data wire. The voltage level of these wires are "pulled-up" to 3,3V when no communication takes place. To send a bit is physically equal to connect the SDA wire to the ground.

5.3 Development setup

When developing hardware drivers one does often needs to deploy other techniques than those used in traditional software development. In our situation, having two hardware boards - one equipped with the main processor running Linux and another with our bare co-processor - the development was carried out, without software simulations on real hardware. For this, an oscilloscope proved to be very useful to visualize and confirm sent/lost bits.

Hardware configuration The development took place by connecting the I2C pins of an IMX25 PDK board to the corresponding pins of a STM32 Discovery board. Since the IMX25 PDK includes several components that are already using the bus, this board did already include the needed pull-up resistors (they have a voltage of 3.3V at idle). Using a voltmeter we concluded that PIN connectors marked I2C SDA and SCL were indeed shared among many board components.

Software development environment Standard printk()-calls were use to debug the Linux kernel driver. The development of the STM32 firmware with I2C was carried out using the IAR integrated development environment on the STM32 discovery board. Since the STM32 Discovery board incorporates a built in JTAG-interface, debugging could be done using break points and investigation of the I2C flags found in the controller's status registers. All registers concerning the I2C controller are mapped into fixed addresses of the STM32's memory map.

ST Microelectronics has provided an example firmware library with pre-written functions to ease access of the microprocessors peripheral components. By reading STM32's reference manual and comments in the library code we got an abstract view of how the I2C controller functions. [?]

5.4 I2C abstractions in Linux

A driver is the abstraction layer between user space programs and some hardware circuitry; as such, it needs to talk with both of them. [?] Both "sides" can be developed in a rather decoupled manner but need to linked into one single object file, that the kernel can load at runtime. Within the Linux kernel, an I2C bus driver pilotes the I2C controller hardware of the IMX25 master device while a character device driver abstracts the STM32 slave device.

5.4.1 Driver usage

One crucial design aspect of Unix is that everything in the system is represented as files. This allows applications to manipulate all these "system objects" with normal file functions such as open(), read(), write(), and close(). All files representing hardware are called device files and are found under the `/dev` directory of the Linux file system. [?] Consequently, also the STM32 chip, our I2C slave, needs to have a file representation. We chose to name the chip's driver interface `/dev/stm32`.

The I2C slave is reached from user space through the "character device" interface. LDD3 classifies a "character driver ... suitable for most simple hardware devices". Indeed, the purpose of our driver is to send small commands and receive answers from the STM32 chip. These commands and answers are sent as strings of character via the character device. [?]

The development steps of a character device are very well described in LDD3. Essentially, a character device driver needs to:

1. Register itself with the kernel (to later be able to create a file found under the `/dev`).
2. Set up pointers to C-functions callbacks, called when a Linux program uses `/dev/stm32`.

The driver decides what data that are actually communicated using I2C. The communicated data can be seen as a layer on top, and can thus can be anything that we decide it to be, as long as the both master and slave agree on the protocol. In our implementation the driver is simply forwarding the same byte strings that it was fed with from user space.

5.4.2 Linux kernel modifications

All I functionality needs to be implemented in the Linux kernel. Luckily, Linux has a general driver architecture that is similar for many buses, among them USB, PCI as well as I2C, that makes it easy to add abstractions for new devices/chips connected to the system's I2C bus.

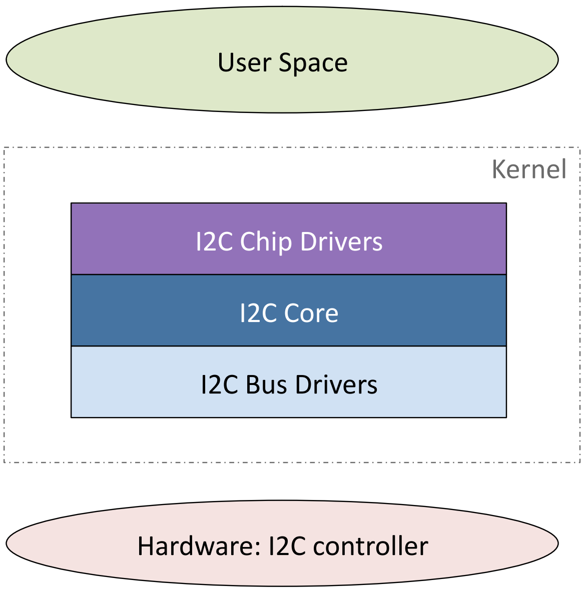

Figure 5.3: Bus drivers and chip drivers are developed separately and then glued together by Linux kernel's I2C core.

Binding driver to device The USB, PCI and I2C subsystems all share the same *binding model* for drivers:

- The bus controller in hardware is controlled and monitored by one *bus driver*. Slave devices, attached to the bus, are represented by *chip drivers*.
- All chip drivers need to register with the I2C core.
- The core calls the chip driver's probe()-function to ask the driver if it is able to control a newly found slave device. [?]

The bus driver is responsible for controlling the low level signaling (addresses, data and clock information) performed by the I2C controller of the microprocessor. Furthermore, several chip drivers are developed, one for each slave device connected to the bus. This is similar to how each device connected to the PCI or USB bus have one dedicated driver, while there is always only one physical bus and bus controller. [?]

In the IMX-series of processes FreeScale has already provided a bus driver for their customized Linux kernel. Using this kernel, we could thus focus on the development implement an I2C chip driver as an abstraction of our STM32 co-processor. When talking about a Linux I2C driver in general, one is actually often referring to the I2C *chip* driver.

Recognizing slave devices An embedded system normally has a fixed number of connected I2C devices. Each slave device is identified with an unique 7 bit address that the master device uses for addressing. [?]

Upon compilation of the Linux kernel for an embedded system it is possible to define a mapping between I2C addresses and name identifiers. In order to for an I2C chip driver to be accepted (linked in at runtime) by the Linux kernel, the drivers name identifier needs to match one of the names defined during kernel compilation.

One might argue that this approach imposes a certain rigidity to the software: once the kernel has been compiled, no names/address-mappings can be modified without a complete recompilation. This can indeed be a problem, but in cases of embedded systems the hardware is usually fixed and we will know what devices (and their addresses) that are to be connected on the product. Nevertheless, this was the approach taken in this design.

To make the kernel aware of the connected STM32 chip, it was given the name `stm32-i2c` and the address `0x0B`, in the kernel source code that describes the hardware configuration, before compiling the kernel. For the IMX class of processors, information about the system's hardware configuration is added to certain files of the Linux kernel source directory `/arch/arm/mach-imx/`. [?]

Code 3 The C structure matches I2C addresses of our hardware board.

```
static struct i2c_board_info mxc_i2c_board_info[] __initdata =
{
        {
            ...
            ...
        }
        {
         .type = "stm32-i2c",
         .addr = 0x0b,
        },
        {
            ...
            ...
        }
};
```

5.5 Implementing an I2C slave firmware

Both the IMX25 and STM32 are equipped with similar I2C controllers in hardware that can be used to get information about ongoing bus activity and state. However, the STM32 is a microcontroller that is much simpler than the IMX25 and will not run standard operating system: the STM32 firmware is supposed to run on "bare metal".

Consequently, when not having an operating system all peripherals need to be configured manually: the STM32 firmware needs to both manage the I2C controller (as Linux's bus driver does) and interpret received data (as Linux's chip driver does).

5.5.1 Hardware and bus initialization

A common scenario in microprocessors is that several PINs of the microprocessor can be mapped to different hardware resources. The principle is called "input/output muxing" and used to allow a more flexible wire routing on the board where the processor and the other chips and components are placed. One of the first things to do in order to activate a built-in I2C controller, is thus to enable and route the I2C to external pins. The two pins, SDA and SCL, need to be configured as "open drain" [2]. [?]

Before powering on the I2C controller, all parameters concerning the slave needs to be configured by setting appropriate values in the controllers control registers. We found some open source examples of how to configure the STM32 as an I2C master but no complete example of how to configure a STM32 slave was found in our research.

We discovered that the library code contained a bug regarding the placement of 7 bit addresses in the ADDR register: the standard code does not peform the needed bit shift as it assumes that a long address is used. This is an odd assumption since long I2C addresses are almost never used in practice. According to the Linux kernel documentation [?], no device using long addressing has yet been seen on the market. Indeed, when the STM32 is configured to act I2C master, the contents of the `ADDR` register is less important. Even though masters also have an unique address, it will never be used since it is always the master who initiates communication and addresses the devices.

5.5.2 Serving requests by interrupt handlers

An I2C slave needs be ready to serve the I2C master whenever the master asks for it. One way to solve this is to let the slave's software wait actively on changes in the I2C controller's registers, polling. This is not a sustainable solution since our co-processor is normally busy running its specific program code. As it cannot actively "poll" for events reported by the I2C controller, the slave's software needs to *interrupt* the current task in order to quickly respond to demands. The master decides the transfer rate and the slave cannot do anything else than follow those orders, whatever it used to do.

The same principle is also found in all types of USB communication. The USB 2.0 specification states explicitly that an USB device must, at any given time, be able to

[2]Open drain explains that the connected wire can be drained to ground, by whatever that is hooked on to it, that is, any of the connected devices.

serve requests from the USB host (the PC-side). [?]

In the STM32 microprocessor, two interrupt channels from the I2C sub system are connected to the Cortex processor core. The channels are masked by default so we needed to unmask them by setting the bits `ITEVFEN`, for bus events and `ITERREN`, for error events. [?]

Bus events One essential bus event is when the master has called for the slave's address. The hardware controller handles the low level acknowledgement on the wire but the software interrupt is needed in order to prepare the slave's upcoming transmission or retrievement of bytes.

If the status register reports that the master has requested a transmission of bytes the slave will immediately need fill the data register with the first byte. Subsequent bytes are put into the data register once the master has ACK'ed the first transmission.

Otherwise, if the request was to receive bytes from the master, the software will wait a second bus interrupt that occurs once the I2C controller has received a byte. Upon this event, our firmware copies the byte of the data register and goes on working until the event happens again.

Error events An error interrupt is signaled upon "ACK failure". The is event does not necessarily mean a failure; it is also set when the master does not want to receive more data. When the master has received the last byte it omits the acknowledgement which was interpreted as a failure to the slave. This bit needs to be cleared in the error handler before new requests could be treated.

Chapter 6

Discussion and conclusion

We have added two middleware services on top of the Linux kernel to provide a new software architecture for telematics devices. One middleware was developed to communicate with the online data center, the second service exports features from a co-operating microprocessor. Both middlewares have been built from scratch using the Qt framework.

Since middleware services need to export functionality to applications, one of the key components in a software architecture is the IPC system. We chose to use D-Bus because of its good bindings in Qt that helped initial development.

In the former product, TCP sockets were used to export features - each IPC connections required a pair of sockets as communication end-points. By using the language bindings found in the Qt framework, D-Bus calls can be treated like any other Qt event, eliminating the overhead of threads that monitor sockets. In an embedded context with a microprocessor with one single core, no performance advantages can be drawn from using parallel threads. When we no longer need to monitor several sockets the implementation can be made much simpler.

D-Bus has built-in support for exporting functionality from one service to many applications. The proof-of-concept implementation shows how D-Bus can be used as a way to do inter process communication, between these handcrafted middleware services and the users' applications. Using an IPC layer such as D-Bus is probably more secure than using bare TCP sockets since the intermediate daemon only dispatches messages that are addressed to methods actually being exported. A program's external interface can thus not be flooded with nonsense data which otherwise is a risk with TCP streams. Clearly, the functions being called still need to be written to deal with ambiguous data input.

One obvious design disadvantage of the system would be the risk of a failed D-Bus daemon. Relying on the D-Bus daemon indeed adds another possible point of failure to the telematics system, in addition to the Linux kernel itself and the various (middleware) services. Meanwhile, the D-Bus daemon needs to be configured manually to only accept traffic from permitted programs.

Extending Linux by adding middleware services is a concept that is also found in Google Android. An in-vehicle infotainment system with similar functionality could indeed have been built by adding a telematics middlware to Android, as studied by the authors of [?]. Our conclusion is, however, that building a system from scratch, around a standard "vanilla" Linux, could be easier simply because fewer components need to be ported. Building a system from scratch, on top of a vanilla Linux kernel, gives more freedom to the developer, as each and every software component can be chosen freely.

More freedom gives bigger open source communities to turn to for help. This freedom also makes it easier to find software that has already been tested with a similar hardware configuration. In contrary, when building a Linux system from scratch, due to time limitations, often only the most crucial components are actually implemented. If extendability in terms of third party applications is required, it could possibly be easier to port a more complete solution such as MeGoo or Android.

Appendix A

D-Bus in an embedded context

Before settling down for a new Linux based software architecture it was useful to study ways of communication between the internal processes. This chapter presents a study on D-Bus, the de facto standard of interprocess communication (IPC) on desktop Linux setups and compares it traditional sockets.

Both sockets and D-Bus allows communication between software processes, not necessarily written in the same programming language. While not being a manifested requirement this pays the way for more flexible implementation.

While D-Bus still is the IPC standard method in embedded systems built on MeeGo, other distributions has taken other approaches. One such example is Google Android's "Binder". Binder will not be studied further in this papers because it is highly coupled to Android systems and is only supposed to work within that context. D-Bus operates entirely in user space, in contrast to for example Andorid's Binder, which is incorporated into the Linux kernel.

A.1 Comparison to sockets

Sockets are used in the former product's software implementation and is an IPC-method that is present in all modern operating systems.

A socket is defined as an endpoint for communication. When two processes communicate, locally or over a network, they employ a pair of sockets, one for each process. In general, are sockets use in client-server architectures; the server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection. [?]

Sockets can be used, by its POSIX API, defined in the C language, or with the help of an application development framework such as Qt. Qt's network module provides an integration of sockets into Qt's event loop. This way network messages can be treated in an asynchronous manner; messages are treated in the event of their arrival. [?]

The main advantages of using socket are simplicity and efficiency. Being a standard components of the all modern operating systems there is no need to add extra supporting libraries nor services to use them. By relying on kernel space mechanisms for passing the messages, the number of context switches are naturally few.

A.2 Introduction to D-Bus

The Desktop-Bus was originally designed to help communication between user applications in the KDE desktop environment. Compared to TCP, D-Bus carries messages as discrete items - not continuous streams of data. The messaging applications sends acknowledgments by default, a system that is transparent to the application programmer and thus arguably more secure.

In contrast to sockets, D-Bus messages are dispatched by an user-space bus deamon. The D-Bus deamon is not a standard component of the Linux system, but is found in practically all desktop Linux distributions. D-Bus connections are nevertheless stateful and connection-based, just as TCP. [?] Linux kernel's hardware abstraction layer can be configured to send notices about hardware changes to the D-Bus, for example upon insertion of a CD or flash-card, to notify the end users desktop environment.

D-Bus is often used to export and call functions across process boundaries, implementing a way to do RPC. From a programmer's perspective, calling a properly encapsulated remote procedure should be no different to calling any other function, except perhaps taking a little longer time to execute. [?] Remote objects are in RPC papers referred to as stubs- or proxy objects.

A.3 Criticism

Using D-Bus involves adding more dependencies to the system. Furthermore, dispatching D-Bus messages requires significantly more overhead than in IPC methods built into the kernel. As messages are dispatched by an user space application, more memory copies and context switches are involved. The counterargument would be that this additional cost is compensated by more reliable and controlled transfers.

There have been discussions about to move the D-Bus into kernel space to improve responsiveness. This is generally a discouraged idea since it is conflicting against the idea of keeping the kernel as small as possible.

A.4 Requirements

In order to get information on the resource requirements of D-Bus and its Qt bindings, tests were conducted in QEMU simulating a target hardware. The tested Linux environment was built by BuildRoot and compiled for ARM using CodeSourcery's cross-compiler. The idea was to use an image with a configuration equal to the one that later will be deployed on the real hardware.

In order to send a message to D-Bus, the client application would need to use the libdbus library. It is however recommended to use a higher-level binding implemented such as the one of the Qt framework. A language binding implies an extra dependency but is less error-prone to use. This chapter explains how to study the additional cost in disk/memory/cpu usage and footprint of including a D-Bus into an embedded system.

A.4.1 Language bindings

Bindings are used to ease the translation of types used in the chosen programming languages to the internal D-Bus representation. Qt' bindings follow the D-Bus approach of exporting methods in an objective oriented manner.

For example, a C++ object is exported to the bus by implementing a D-Bus interface. The calling application can access this exported interface by using a D-Bus proxy object. The Qt toolkit provides a tool that from an XML definition is able generate native C++ code representing this concept. [?]

The C++ code representing the proxy object files are included by the calling application. The interface is included by the receiving application to map incoming D-Bus calls to a native function.

In D-Bus, a method call on a proxy object is transformed into two asynchronous message; one containing the forwarded call and another message containing the returned data of the method. D-Bus converts the call and its answer into intermediate RPC messages.

A.4.2 File size footprint

Since the programs using the bus rely on languages bindings provided as shared libraries, the executables' file sizes are only a few kilo bytes each. However, the extra libraries added to the system would require some additional disk space.

When measuring the cost of adding D-Bus to our target system one would need to count the cost of adding the D-Bus daemon as well as the supporting libraries and language bindings needed for application using the bus. The executable file size represents the space occupied by the ELF-binaries of the deamon and the libraries on the flash.

A.4.3 Memory usage

When investigating the amount of memory a process occupies, one normally relies on the UNIX ps or top tools. Both of these tools get the information about the systems processes from the `/proc`-file system. In an embedded environment one can thus directly rely on Linux `/proc` filesystem, notably the `/proc/<pid>/status`-file. According to the Linux Kernel documentation, the "virtual set size" (VSZ) is the size of all pages addressed by the process whereas the "resident set size" (RSS), is the size of all frames, the actual use of physical memory. [?]

By investing the client/server in `/proc/<pid>/maps`, one realizes that the server and client use exactly same libraries. Besides libdbus and libQtDbus, many of the loaded libraries are standard components in the Linux system.

When several processes share dynamic libraries, when a new service is added, thanks to virtual memory, only its new code and data (ie. the writable parts of the program and its libraries), will eventually be allocated memory. The paging is done progressively because Linux uses copy-on-write.

This makes VSZ and RSS less representative as they count the pages/frames of the shared libraries several times. We are interested in the memory footprint of a complete D-Bus setup, including the daemon as well as the libraries shared by the applications using the bus. To get accurate information about the number of frames actually occupied in memory we would need a way to summarize the frames (marked as RSS) for both processes but only counting shared code and data from library ones.

Appendix B

Embedded Linux components

A traditional embedded Linux system contains a set of standard components. To start with, the Linux kernel it needs to be loaded into memory by a boot loader, customized for the built/chosen hardware. Once the kernel is loaded to memory and the system's hardware has been detected, user programs can be launched. [?] This type of relations, between the different software components found in a Linux system, are described in this chapter.

B.0.4 Boot loader

The bootloader is the first program[1] that runs once on the target board has been powered up. It carries out the low-level hardware initialization of processor and memory but also the serial and Ethernet interfaces towards the host computer.[?] Several different bootloaders targeted for embedded Linux exists, among them, the widely used "U-boot".

B.0.5 Kernel

The kernel is responsible for maintaining all the abstractions of the operating system, including virtual memory, processes and other hardware abstractions. The kernel needs to be configured to support the filesystem used by the embedded device. [?]

Within the kernel layer, Linux is composed of five major subsystems: the process scheduler (sched), the memory manager (mm), the virtual file system (vfs), the network interface (net), and the inter-process communication subsytem (ipc). Conceptually each subsystem is an independent component of the Linux kernel even though there is some interdependence among the five subsystems in the actual implementation. [?]

[1]Actually, the very first code to be executed is found in a read-only memory called the boot strap. The boot strap locates the boot loader program in an external memory according to a configuration set by hardware pins on the board.

B.0.6 Libraries and development frameworks

While the Linux kernel indeed offers a wide API, the abstraction level is much lower than what is expected by the application developers. For example, to ease development of visually appealing user applications confirming to upcoming standards of infotainment systems further layers of abstraction are added. A supporting software layer does often help building graphical interfaces but can also incorporate other functional domains to form a complete a software development *framework*. Such frameworks make more assumptions and are thus easier to use. [?]

The Qt Framework Qt is one example of a commonly used framework. It was originally developed by Trolltech but has been managed by Nokia since 2008. Applications built with the help of frameworks are supposed to be easier to develop and maintain as an extensive API allows more abstractions and simplifications. According to Nokia, the Qt framework could reduce development costs, improve maintenance across platforms and shorten time to market by 75%. [?]

Still, Intel has a great interest in the framework as it plays a central role in MeeGo, their Linux distribution and software platform. In MeeGo, the D-Bus is for example used to let applications access information about the system's network connection. [?]

B.0.7 System daemons and services

A common Linux system has a set of programs called *daemons* running invincibly in the background. Each daemon, or service, has one unique purpose that helps the system to function.

Kernel daemons are a special kind of daemons that reside inside the kernel's part of the memory. For instance, one kernel daemon performs the work of swapping out memory pages that has not been used for a while. These tasks are typically found in the output of the `ps` command with brackets surrounding their names and are reported to consume zero memory virtual memory. Furthermore, these kernel daemons' output cannot be found in the `/proc` filesystem.

Generally, all programs launched in the background upon system boot are considered daemons, whether it is a HTTP-server or any other process that is set up to run continuously.

B.0.8 User applications

All programs that directly interact with the end user, the user applications, are commonly refereed to as application level software. These programs are, with the help of the underlaying system libraries or frameworks, written for exactly this purpose.

With clear, available API and a market infrastructure (such as Apple's "AppStore" or Google Android's "Market"), third-party *apps* can be developed to directly target the IVI system.

Appendix C

Methods of development

This chapter highlights the embedded Linux traditional development setup. We present some freely available tools to build and debug open source code that is designed to run on embedded microcontrollers.

C.1 Cross-compiling

Software development of embedded Linux systems is performed at a *host* work computer. Since the processor architecture of the host and the target differs, a cross-compiler is used: the produced machine code will not be of the same kind as the one of the host.

Developing embedded Linux typically involves cross compiling, notably when developing applications to run on ARM or another non x86-based hardware platform. The cross-compiled system is then debugged either on a physical target, similar to the final hardware, or in a hardware simulator run on the PC.

Cross-compilers and the tools needed in a cross development environments are freely available on the Internet. This saves the developer from starting the project by compiling the toolchain from source - a tedious task that involves much configuration. [?] One example of a free toolchain is CodeSourcery's ARM Lite++ that also contains a matching GNU C library. This toolchain was for all ARM-based development in this project.

A cross-compiler is compiled against a certain C library, which is why the target's C library must be chosen and at same time as building or selecting, the tool chain [1]. [?] Changing one of the two at a later point is not recommended. A swap of toolchain and thus the chosen C library could imply a recompilation of all the systems' software components [2]. Furthermore, all binaries, both kernel and user space, in the system must

[1]Actually, GCC only needs the C library when it compiles the C++ language. Since the C library needs to be cross compiled itself we need a preliminary "bootstrap" cross compiler to do this. Hence, a first compiler us built with support for C only, and a full compiler is built once the C library is available. [?]

[2]In a desktop Linux setup, ldconfig is used to keep symbolic links updated. For example, /lib/ld-linux.so.2 points to /lib/i386-linux.gnu/ld-2.13.so in Ubuntu. But it is still considered dangerous to upgrade glibc.

be compiled with the same ABI. The ABI is a convention that covers, for example how arguments are passed to functions on assembly level. Nowadays, everybody uses EABI. [?] All of these compatibility problems are taken care of when using a prebuilt tool chain.

C.2 Debugging Linux applications

The ways of debugging an embedded system differs according to phase in development and type of application. If no hardware is available at all, development and tries can often be done on a Linux host: either by simulating the target in an emulator or by running the application natively. Peripherals that will later be found in the real system will need to be simulated. When, for example developing Android or Iphone applications, the SDK integrates a simulator to let the development phase commence before having a real device available.

Since our gateway application runs within Linux, and depends on the portable Qt framework, could develop and test it in native Linux, before deploying in onto the real hardware board. To run the program on the target, the native compiler is replaced by a cross compiler. Indeed, as we depend on the Qt library, also Qt has to be compiled using this cross compiler toolchain.

C.3 Debugging "bare metal" code

A different approach to debugging needs to be taken when developing "bare metal" software, that is, code that will run without a supporting operating system. Examples of such software are boot loaders or "bare" firmware code. When customizing a boot loader, trails often need to be done in an iterative manner on real target hardware. Hardware simulation is not possible since the designed board normally differ from the boards emulated in QEMU.

In these situations a JTAG device can be used to read memory and register data, halt execution and retrieve other information about the connected hardware. Many proprietary JTAG devices are bundled with development environments but they do all tend to be quite expensive [3]. Furthermore, Linux and other open-source projects often depend on the GNU toolchain and could thus be troublesome to compile with the toolchains found in commercial kits and environments.

OpenOCD is an academic project that was designed to enable debugging of on-chip code built with the GNU toolchain, using standard JTAG devices. OpenOCD is an user space program that runs on the development workstation. It acts as a wrapper, that takes instructions from either telnet or GDB and translates these instructions to the JTAG device (connected by USB). We used Segger's J-LINKs, one JTAG that is compatible with our ARM cores and is supported by the OpenOCD project [4].

[3]Wolfgang Denk, U-Boot's founder, endorses the comercial BTI2000 JTAG device that incorporates a "gdbserver" which removes the need of OpenOCD.

[4]Segger actually also their own "J-Link GDB Server" for this purpose, but the tool is only available on Windows. It communicates over TCP and should thus be independent of toolchain and GDB version.

In order for GDB to be able to follow execution of code it needs information about the where written source code is put in RAM of target. When a "bare metal" software is compiled, two files usually are generated for this purpose:

1. A binary file (pure machine code) that is flashed to target board.
2. A complementary ELF-file that contains debug information.

The debug information includes the symbol table and information about where the compiled C functions' (machine) instructions are put into RAM of target. This mapping is needed by GDB to give an interactive view of which lines in of the source that is currently executed by the target. The ELF-file's debug information is used by GDB to match the program counter's value, given "live" by the JTAG, with an exact location in the C source code.

With OpenOCD runing as *gdbserver*, the brigde between GDB and the JTAG hardware, we could successfully perform traditional breakpoint-debugging on the most low-level code of the microcontroller. Notably, OpenOCD proved to be very useful while tuning U-Boot, the Linux boot loader used in this project. In principle, the same methods can also be used to debug the Linux kernel, or any piece of code that runs in an environment where the "on target" gdbserver is not availble. OpenOCD was also used successfully to flash a sample firmware for the STM32 microcontroller.

C.3.1 Debugging U-Boot

Enabling debug information In order to do efficient debugging of U-Boot on real hardware a few compiler settings need to be set. First, add the -g option to config.mk, otherwise GDB reports "no debugging symbols found". Also we temporarily remove compiler optimizations as they could make the step-command behaving unpredictable.

Code 4 Correct settings in U-Boot's config.mk before compilation.

```
DBGFLAGS= -g -DDEBUG
OPTFLAGS= -Os -fno-schedule-insns -fno-schedule-insns2
```

Handling re-location of code The first code the microcontroller executes is called the *bootstrap* and is found in a fixed ROM memory. The bootstrap is never touched and is made by the microprocesor vendor. It is responsible of locating the first piece software (in our case U-Boot) and start running it. According to physical PIN-configurations, the bootstrap code will either go look for a program in NOR flash (to do "execution in place", XIP) or from another source, a NAND flash for example. If the program is stored in a memory that does not support XIP, it is first copied to RAM.

Once the bootstrap has located U-Boot it passes control the U-Boot software. The main purpose of U-Boot is to locate the Linux kernel in flash and pass it control. Working with the flash memory is easier if U-Boot runs from RAM. This is the reason why U-Boot relocates itself to RAM. All code, functions as well as variables will be given new addresses, a change that the GDB debugger needs to be informed of. GBD needs to be configured differently if code to be debugged is run from flash or has been relocated to RAM as seen in Code **??** and Code **??**.

Code 5 .gdbinit example. Before relocation, execution takes place directly from flash so the addresses found in the ELF file can be used by GDB.

```
# Connect to OpenOCD
target remote localhost:3333

# Inform GDB about hardware.
set remote hardware-breakpoint-limit 2
set remote hardware-watchpoint-limit 0

# Run until breakpoint at subroutine "reset".
thbreak reset
continue
```

When debugging code from ROM, only *hardware* breakpoints can be used (software interrupts requires write access to insert `BKPT` instructions). The ARM9-based microprocessors do only have two hardware breakpoints [**?**]. This limitation forces us to manually disable a breakpoint once being hit. GDB's `thbreak` command is used to set temporary hardware breakpoints that are automatically disabled and thus being freed to break at another address.

Code 6 After relocation, GDB needs to be informed about the code's new placement in RAM. The relocation address can be read out from U-Boot's internal data structure using GDB commands.

```
...
(gdb) thbreak board.c:418
(gdb) print/x addr
$3 = 0x83fbb000

(gdb) symbol-file
 (y or n) y
(gdb) add-symbol-file u-boot 0x83fbb000
 (y or n) y
Reading symbols from u-boot...done.
(gdb) thbreak board_init_r
```

After relocation the `.text` and `.data`-segments have been copied from flash to RAM. GDB can be informed of the code's new location by first resetting the symbol table with `symbol-table` and then re-read it using the new base address found by `print/x addr`.

Configuring OpenOCD OpenOCD needs to be configured for given target processor. The code bellow shows our configuration.

Code 7 OpenOCD configuration for our ARM9-based microprocessor.

```
# JTAG speed
adapter_khz 1000

# Use combined on interfaces or targets
# that can’t set TRST/SRST separately
reset_config trst_and_srst

$_TARGETNAME configure -event gdb-attach {
puts "Reset on GDB connected..." ; reset halt
}

# Only HW breakpoints can debug ROM
gdb_breakpoint_override hard
arm7_9 fast_memory_access enable
```

We reset and halt the processor once GDB connects to let GDB set breakpoints on the very beginning of the execution.

C.4 Build tools

As an alternative to put together and build the embedded system to from scratch, both commercial and freely available embedded Linux distributions exists. According to [?], these distributions "are becoming increasingly sophisticated and, in many cases, adequate for certain embedded project requirements."

Even though it normally is considered a tedious task, a complete embedded Linux system could off-course be built from scratch. This involves, downloading, tweaking and cross-compiling of each needed package by hand. The FreeElectrons lecture [?] describe how this is done. What makes this approach even more time consuming is the fact that for each application or library to be built, all support libraries on which it depends need to be cross-compiled prior to it. Handling the this ordering is automaticly taken care of by a build tool like Buildroot. Another task that also is automatized by all common build tools is creation of target file-system image, often abbreviated as *RootFS*, once all software components are in compiled. Some systems do also handle the kernel configuration and compilation.

The following paragraphs describe the most well-known of the freely available build tools environments that were considered when settling down for a new build system.

BuildRoot Buildroot was used in this project and is a free build tool that is built upon Make and the Makefile format. It can build its own tool chain or use an existing one, such as the one from CodeSourcery, to create ready-to-use target file system image. The "ncurses"-based configuration menus are also linked the standard Linux kernel configuration.

Yocto The Yocto Project, announced in late 2010 by Intel is an attempt by hardware vendors to unit with embedded Linux developers in creating "custom" Linux distributions. In similarity to BuildRoot, Yocto can generate a complete RootFS image but we believe that, in order to do so, it requires more configuration and is by far not as stright forward as in Buildroot.

MeeGo MeeGo is conceived to ease the development of embedded devices relying in a graphically oriented applications such as smart phones, smart TVs but also in-vehicle infotainment systems. Instead of being a build system, MeeGo also provides complete precompiled binary distributions provided for certain common hardware targets.

C.4.1 IDE integration

Some of the embedded Linux distributions do sometime include pre-configured patches to development environments such as Eclipse CT. When cross-compiling through Eclipse we need to inform Eclipse about what tools to use, otherwise a normal compilation will be performed. This can be done either manually or with the help of plug-in that makes the cross tool chain available through the IDE's "Project Wizards".

No such plug-in is today bundled with more minimal build systems such as Buildroot. In these cases we need to tell Eclipse to use our cross tools ourselves manually, in the Eclipse-project's properties dialogs, or using a personal built plug-in. [?]

Yocto provides an Eclipse-plugin aimed to automatize tasks that are usually associated with development and debugging. However, the plug-in is highly associated to the Yocto ADT, Application Development Toolkit, forcing the developer to use also the entire Yocto build system. One disadvantage with this approach is that in a sense, the developer will thus be locked to chosen build system.

Bibliography

[1] Nokia. Qtdbus quick tutorial. `http://www.developer.nokia.com/Community/Wiki/QtDbus_quick_tutorial`, December 2011.

[2] Elisabeth Freeman, Eric Freeman, Bert Bates, Kathy Sierra, and Elisabeth Robson. *Head First Design Patterns*. O'Reilly Media, 2004.

[3] Thiemo Voigt Adam Dunkels, Oliver Schimidt and Muneeb Ali. Protothreads: Simplifying Event Driven Programming of Memory-Constrained Embedded System. 2006.

[4] FreeElectrons. Various powerpoint presentations. `http://free-electrons.com/docs/`, December 2011.

[5] Andy Wellings and MinSeong Kim. Asynchronous Event Handling and Safety Critical Java. *JTRES 10*, pages 53–62, August 2010.

[6] Jean-Marc Irazabal and Steve Blozis. *AN10216-01, The I2C Manual*, March 2003.

[7] STMicroelectronics. *STM32F100xx advanced ARM-based 32-bit MCUs (Reference Manual RM0041)*, July 2011.

[8] Alessandro Rubini Jonathan Corbet and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, 2005.

[9] Kernel developers. *Linux kernel source code documentation on I2C*, 2011.

[10] Jan Axelson. *USB Mass Storage: Designing and Programming Devices and Embedded Hosts*. Lakeview Research LLC, 2006.

[11] M. Torchiano G. Macario and M. Violante. An In-Vehicle Infotainment Software Architecture Based on Google Android . *IEEE International Symposium on Industrial Embedded Systems, 2009. SIES '09*, pages 257–260, 2009.

[12] Peter B. Galvin Abraham Silberschatz and Greg Gagne. *Operating System Concepts, Seventh Edition*. John Wiley & Sons, 2004.

[13] Andreas Aardal Hanssen. Qt's Network Module. DevDays2007, PowerPoint presentation, 2007.

[14] FreeDesktop.org. D-bus specification. `http://dbus.freedesktop.org/doc/dbus-specification.html`, December 2011.

[15] Christopher Hallinan. *Embedded Linux Primer: A Practical Real-World Approach.* Prentice Hall, 2006.

[16] Nokia. Qt in use. `http://qt.nokia.com/qt-in-use/`, December 2011.

[17] MeeGo Wiki. D-bus/connman. `http://wiki.meego.com/D-Bus/ConnMan`, December 2011.

[18] Gilad Ben-Yossef Karim Yaghmour, Jon Masters and Philippe Gerum. *Building Embedded Linux Systems.* O'Reilly Media, 2008.

[19] ARM Ltd. Debug infrastructure on arm-based systems. `http://www.redacom.ch/software/arm_keil/docs_infopage/documents/keil_beginners/debug_and_trace.pdf`, December 2007.