FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

An Inquiry into Gateways

for Embedded Systems and Consumer Devices

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Martin Lampacher
Matrikelnummer 0725908

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Christian El-Salloum

Wien, 16.10.2012

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

An Inquiry into Gateways

for Embedded Systems and Consumer Devices

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Martin Lampacher
Registration Number 0725908

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner
Assistance: Univ.Ass. Dipl.-Ing. Dr.techn. Christian El-Salloum

Vienna, 16.10.2012

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Martin Lampacher
Lerchenfelder Guertel 17/11-12, 1160 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Danksagung

Das größte *Dankeschön!* gilt vor allem meinen Eltern, welche mir meine Ausbildung und damit auch dieses Studium ermöglicht und mich fortwährend unterstützt haben. Ein besonderes *Danke!* an dieser Stelle auch an Christian El Salloum für die hervorragende Betreuung und die sowohl aufbauende als auch konstruktive Kritik, an Leo Mayerhofer für die unzähligen Stunden und Geduld im Labor und an Peter Puschner für die Betreuung der Diplomarbeit. Nicht zu vergessen sind auch Fabian Vent und Johannes Maschler, welche in den zweifelhaften Genuss des Probelesens und Diplomarbeits-relevanter Diskussionen gekommen sind. Danke!

Abstract

Embedded systems are already an integral part of our lives: Mostly hidden away from plain sight, they facilitate our everyday tasks. The tremendous triumph of consumer electronics such as smartphones and tablets has opened up huge possibilities for embedded systems and brings us one step closer to the ambitious concept of the so called *Internet of Things*, where every object can be uniquely identified and plays an active role in informational and social processes. *Gateways* are a vital component towards this *Internet of Things*, as they are the key ingredient needed for connecting systems, and yet there does not exist a common notion of what a gateway actually is, i.e., what the main characteristics of every gateway are. Consumer electronics are a promising approach to function as gateways between embedded systems and the outside world, that is, they can enable the embedded system to go online and are at the same time an easy-to-use interface for the average user.

This thesis discusses the terminology *"Gateways"* to provide a common notion that is valid not only in computer science, but also in other areas of application. Furthermore it investigates the connection of current consumer devices, such as smartphones and tablets, to embedded systems. A case study of a gateway between the diagnostic port of a car and consumer devices illustrates the presented concepts.

Kurzfassung

Eingebettete Systeme sind ein wesentlicher Bestandteil unseres Alltags: Meist im Verborgenen und ohne jegliche Interaktion vereinfachen sie unseren Tagesablauf. Der enorme Erfog der Unterhaltungselektronik, wie z.B. Smartphones und Tablets, hat eine Vielzahl an Möglichkeiten für eingebettete Systeme eröffnet und ist ein wichtiger Schritt in Richtung des hochstrebenden Ziels des sogenannten *"Internets der Dinge"*, in welchem auch Objekte eindeutig identifiziert werden können und an sozialen und informationstechnischen Prozessen aktiv teilnehmen. *Gateways* sind ein wichtiger Baustein in Richtung dieses *"Internets der Dinge"*, da sie die entscheidende Komponente für die Verbindung von Systemen darstellen. Trotz der Wichtigkeit solcher Gateways gibt es noch immer keine einheitliche Terminologie was ein "Gateway" genau ausmacht und was dessen primäre Charakteristiken sind. Tablets und Smartphones sind ein vielversprechender Ansatz um als Gateway zwischen eingebetteten Systemen und der der Außenwelt zu fungieren: Sie bieten die Möglichkeit einer Internetanbindung und sind gleichzeitig ein einfach nutzbares Interface für Normalbenutzer.

Mit dieser Diplomarbeit wird die Terminologie von "Gateways" diskutiert und eine einheitliche, allgemein gültige Definition des Begriffes eingeführt. Des weiteren enthält sie eine Untersuchung über die Verbindungsmöglichkeiten von eingebetteten Systemen zu derzeitigen Tablets und Smartphones. Im Zuge einer Fallstudie eines Gateways zwischen der Diagnoseschnittstelle eines Autos und Smartphones werden die zuvor diskutierten Konzepte praktisch umgesetzt und illustriert.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

The ever increasing degree of computerization that has found it's way into our everyday lives brings along a tremendous amount of information and possibilities for interaction. While just half a decade ago it was rather complicated to bring this information to the masses, consumer devices have opened up a huge possibility for common users to interact with the systems that quietly accompany them in their daily routine.

On the other side, the constant technological progress allows today's embedded systems to become smaller and smaller and to still provide sufficient means for communication. While the number of physical interfaces, such as buttons and displays, decreases, the development in wireless technologies enables new ways to connect interface such systems.

1.1 Motivation

One major challenge that engineers are facing today is to integrate embedded systems into the internet and to make them accessible to consumers. Sensors and actors, for instance, are among the most interesting systems for consumers. One way to make embedded systems open to common users are consumer devices such as smartphones and tablets. They offer a convenient user interface which is familiar to many users, as well as many possibilities for interaction. On the other side, consumer devices can also act as *gateways* for embedded systems to the internet and allow cloud integration or other fields of application.

In the automotive industry, the integration of consumer electronics is currently a hot topic: Manufacturers try to integrate smartphones to allow for third party applications, such as for navigation, monitoring or entertainment. The possibilities are manifold and there's always room for new ideas. Until now, however, there does not yet exist a uniform way to connect to devices from different manufacturers.

1.2 Goals of the Thesis

The thesis has three major goals: The first one is to undertake an investigation into the terminology of a "gateway" and to provide a common notion for the term which is valid not only in computer science. Subsequently an inquiry into interfacing consumer electronics reveals the difficulties and possibilities that arise when connecting embedded systems to consumer electronics. Finally, a case study is carried out in order to demonstrate how the previous concepts can be applied in practice, and to emphasize the key role of computer engineers between embedded systems design and software engineering. The hardware platform that is developed in the course of this case study allows for future experiments and investigations.

1.3 Structure

The structure at the same time reflects the *modus operandi* pursued throughout the development of this thesis: Chapter 2 gives an insight into the basic concepts of security and cryptography that is needed to be able to follow the document. It also presents some state-of-the-art implementations of automotive- and consumer electronic gateways. Subsequently Chapter 3 outlines the very basic research about what *"Gateways"* actually are, which characteristics are inherent in every such gateway and what impact it has on system design. It is not the intention of this inquiry to give a formal definition of a gateway, but to give a basic discussion and common notion. This inquiry serves as a strong fundament for the executed case-study.

The Chapters 4 and 5 cover one side of the gateway that was implemented in the course of this thesis: Chapter 4 provides a general overview of embedded systems in the automotive sector, while Chapter 5 goes more into detail about *on-board-diagnostics* and the diagnostic port of the vehicle, the actual interface that is used by the gateway that was implemented as case study. What implications and possibilities arise when consumer devices are connected to embedded systems is discussed in Chapter 6. It involves general observations, as well as a discussion on the actual interfaces that are used by the majority of today's consumer electronic devices.

Chapter 7 describes the development cycle of the case study: Beginning with a general system model and a requirements analysis a possible *product* is designed, which should allow to execute diagnostic services via the diagnostic port of a car by using a consumer device, such as a smartphone. An investigation for a possible development platform together with the decision for a custom design is followed by the actual implementation and concluded by the evaluation of functionality and the fulfillment of the posed requirements. This case study essentially reflects a whole product design cycle. Chapter 8 concludes this thesis.

CHAPTER 2

Background and Terminology

Regarding the *Inquiry into Gateways*, i.e., the discussion about the terminology of *"gateways"*, Chapter 3 will provide the necessary background information, as it does no good to discuss existing definitions without bringing them directly into the context of the inquiry. The scope of this chapter is to provide a general overview of the concepts needed to make sense about the discussions in the following chapters and to present some notable projects regarding gateways to consumer devices and the automotive on-board-diagnostic port.

2.1 State of the Art

Developing *applications* for consumer electronics is currently a hot topic, and so are gateways to consumer electronics, or in other words, connecting (embedded) systems to consumer devices. The case study presented in Chapter 7 is a gateway between the on-board-diagnostics (OBD) port of a car and consumer electronic devices: The diagnostic port is used by the manufacturers, e.g., for flashing or parametrization, and at the same time provides diagnostic information, such as the current vehicle speed or emission-related test results (refer to Chapter 5 for further details). The term *"consumer electronics"* refers to devices such as tablets and smartphones. There exist a plethora of projects out there, starting from simple hobbyist applications to projects supported by companies such as the Ford Motor Company. The following are among the most interesting projects or products encountered during the research:

OpenXC [74] is a platform that is intended to enable third-party developers to build hardware and/or software that interacts with the car's internals. The Ford Motor Company is one of the founding members of OpenXC. Figure 2.1 depicts the CAN translator hardware, which bases on the Arduino [51] prototyping platform. The hardware is connected to the diagnostic port of the vehicle and currently allows *Android* devices to access a number of

different signals via USB or Bluetooth, where Bluetooth is currently still in the experimental phase.

Figure 2.1: OpenXC CAN translator [74].

TEXA [81] is a manufacturer of vehicle diagnostic solutions. It provides a variety of instruments that can be used by professional mechanics or private customers. The fact worth to be mentioned at this place is that TEXA produces its own viewers, like the TEXA Axone in Figure 2.2, to display the information retrieved by the diagnostic instruments. Common devices such as iPads or Android tablets, on the other side, would already exist and would provide about the same capabilities as those viewer devices.

Figure 2.2: TEXA Axone viewer for TEXA instruments [80].

Another product by TEXA is the OBD log [80], which is depicted in Figure 7.4 (page 70). It is a simple plug that is connected to the diagnostic port and continuously captures data

that can be read out by using a computer connected via USB. This data is intended to provide more information for car workshops or to enable car enthusiasts to access and analyze parameters such as the vehicle speed or consumption.

Although it is not intended for communication with consumer devices, the TEXA OBD log has served as an expiring example in terms of *functionality vs. design*: The diagnostic port is in general not easily accessible, such that minimal user interaction should be required. The OBD log is just plugged into the port and then works autonomously until the data is read out. No cable clutter or mounting is necessary to set up the device.

SmartThings [77] On August 23, 2012, just at the final phase of the case study that was executed as part of this thesis, the *SmartThings* project started to ask for backers on the *Kickstarter* funding platform [66]. It advertises itself to be *"a platform that easily connects everyday things to the Internet."* In fact, the project does look extremely promising and provides exciting possibilities. The initial goal of $\$250,000$ has been easily exceeded: The funding ended on September 22, 2012, and resulted in $\$1,209,423$ pledged, which is 483% of the money they asked for. Figure 2.3 depicts the basic concept..

Figure 2.3: SmartThings platform [77].

The platform allows common *"Things"*, reaching from sensors (thermostats, vibration sensors, accelerometers, pressure sensors) to dog-tags, to be extended with wireless communication such that it can connect to the internet and be easily accessed by consumer devices such as iPhones or Android devices. Through a variety of Apps those *"SmartThings"* can be controlled and observed. One advertised application is the *"Make it look like"* app, which uses the hub depicted in Figure 2.3 to control lights such that it looks like somebody's home.

All in all, the project appears to be a very promising step towards the *Internet of Things*. One thing worth to mention is that the hardware seems to base on similar components as the ones used in the course of the case study: The promotion video shows hints of Microchip [72] products on the board, amongst others a WiFi module. The development board of the case study also bases on Microchip products (cf. Section 7.5.2) and as will be discussed in Section 7.6.4, the case study could be easily extended to other applications than on-board-diagnostics, just as the *"SmartThings"* project.

2.2 Security

The development of embedded systems is subject to three key aspects, wich are *functionality*, *safety* and *security*: Every system has a purpose, a task that it needs to fulfill, its *functionality*. The execution of this function must not pose any risks to the system itself and especially not to humans, i.e., it must be *safe*. *Security*, on the other side, *"is concerned with protection against the manipulation of [IT] systems by humans"* [18]. Clearly, safety and security are closely related, as a manipulation of the system may lead to safety risks. While functionality and safety have always been a major concern in embedded systems design, the demand for security has significally grown in the last few years. Chapter 4, for instance, will briefly discuss the emerging need for security in the automotive sector. [22] provides an excellent overview over security concepts for embedded systems. It introduces the following definition for security:

> *"The term 'information security' means protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction in order to provide three core principles: confidentiality, integrity and availability."* [22]

Those core principles have to be guaranteed for the so called "asset" of the system, which is *"the value of its service to the user"*. Regarding communication, an asset could basically be the information or data that is exchanged with another system. The concepts of confidentiality, integrity and availability[1] cope with the following:

- *Confidentiality* is about *secrecy*: The assets should only be available to *authorized* parties.
- *Availability* ensures that the assets are accessible for all *authorized* parties during the specified periods. A *denial-of-service* happens if legitimate users are prevented from accessing those assets, i.e., the assets are *unavailable*.
- *Integrity* is ensured if it can be guaranteed that the assets cannot be maliciously altered, i.e., *"the modification [...] of an asset requires authorization"* [22].

[1] Regarding the definitions and terminology given especially throughout this Chapter it is virtually impossible not to coincide with one that has not already been used by another author. The used wording has been chosen in all conscience and is covered by the referenced literature.

If all three properties are fulfilled by a system, it is considered secure. In practice, however, design flaws or software bugs often introduce *vulnerabilities* that can be exploited by an attacker. The three major sources for vulnerabilities, also know as the *Trinity of Trouble*, are the following:

- *Complexity.* Typically, the larger a system is, the more interacting components it includes. Clearly, more interaction also increases the complexity of the system, which grows larger and with it the probability for bugs.
- *Extensibility.* For software systems, updates and fixes of existing bugs can introduce new vulnerabilities. For embedded systems, updates and extensions are especially critical as they must also include the hardware. One simple example is allowing firmware updates, which requires the microcontroller to be re-programmable. Excluding this possibility would also discourage attackers from re-programming the device. Today, however, updates and extensions already are an integral property of systems.
- *Connectivity.* An increased number of possibilities to connect to a system clearly increases the attack surface. This can be especially problematic for remote or wireless applications, as they can eliminate the need for physical access.

Additionally, in the special context of *embedded systems*, [22] also adds the following source for vulnerabilities:

- *Operation in an untrusted environment*. For some systems it is of utmost importance that security is also maintained if the system is operated out of specifications. A payment system using smart-cards, for instance, must make sure that the smart-card cannot be altered, e.g., by charging it through a custom, unauthorized device.

2.3 Cryptography

Cryptography (which is also introduced by [22]) is the current approach to provide data security, and is e.g. used by security protocols such as WEP/WPA or WPA2. It involves the following security services [3]:

- *Confidentiality* is the same as previously discussed for security. Through data encryption, only those with a valid key for decryption can actually retrieve the data.
- *Data Integrity* also remains the same for cryptography and is in general achieved through a combination of hashing and encryption or through special message authentication codes.
- *Data Origin Authentication* “guarantees the origin of data” [3] and implies *data integrity*. It is not yet *entity* authentication.

- *Non-repudiation* is a very strong property that allows to verify the sender of data without the consent of the sender, i.e., the sender is not able to *repudiate* the fact that he has sent the data. *Non-repudiation* implies *data origin authentication* and thus also implies *data integrity*.

The basic concept of cryptography is to alter or *encrypt* raw data by using a *key* such that it cannot be retrieved without *de-crypting* it by using another key, which can be the same as the one used for encryption. Thus, cryptography provides the means for a secure and confidential data exchange between two parties. There is, however, one thing to notice:

> *"As a matter of general principle it is not possible to establish an authenticated session key without existing secure channels already being available."* [3]

[3] goes more into detail about this fact and provides a comprehensive overview of key establishment and authentication protocols. In summary this has as a consequence that the the key(s) needed for cryptography must be either already known by the participating entities (e.g. as *pre-shared-key*, or established through a secure physical connection), or that there are certified public keys available. Once such keys are available, the properties presented before can be provided by two possible types of cryptography algorithms:

- *Symmetric Cryptography:* Symmetric key cryptographic algorithms use the same key for en- and decryption and can be implemented in a very efficient manner. [14] provides a short introduction into the topic of symmetric cryptography. The general problem of symmetric key algorithms is that the key must be established through a secure channel or assumed to be already available.

- *Asymmetric Cryptography:* Asymmetric key cryptographic algorithms, or *public-key* algorithms, use different keys for en- and decryption. A *private-key* which is only known by the sending entity is used to encrypt the data. The *public-key* is publicly available to every party and can be used to de-crypt the data. Notice that this public key fulfills the requirement of a *existing secure channel* mentioned previously.

  The problem with asymmetric key cryptography is that it is generally very arithmetic intensive and thus not always suited for lengthy communication. An approach to this problem is to use asymmetric key cryptography to establish a symmetric key which can then be used for encryption. [13] provides a good overview over the fundamentals of asymmetric cryptography.

Thus, symmetric cryptography could e.g. be used to encrypt communication channels, while the key has been established by using asymmetric cryptography. Another very important application of asymmetric cryptography that should be mentioned here are *digital signatures*, which rely on the fact that public key cryptography allows for the previously mentioned *data origin*

authentication, i.e., the receiving party can verify that the data has come from the right sender. This can be used, e.g., to guarantee that a software update actually comes from the manufacturer.

The fact that cryptography, especially asymmetric cryptography, can be quite computationally intensive poses special challenges to embedded systems. [25] (and [22] as well) provides an overview over cryptography in embedded systems, including benchmark results. All in all, cryptography provides the necessary means for guaranteeing security.

There is, however, one thing left to notice: Cryptography only works under the *black-box* assumption, i.e., that it is executed on top of inviolable and perfect systems. In practice, however, this is not the case, as in general it cannot be assumed that the attacker has no physical access to the device and that algorithms execute in zero-time or at least independent of the actual keys. Without going into great detail, there exist feasible attacks to cryptographic implementations based on *tampering* (cf. vulnerability: *Operation in an untrusted environment*), that is, breaking the black-box assumption of the system and thereby the cryptographic boundary. [22] goes more into detail about general attacks, while [19] and [15] address tamper resistance in the context of embedded systems. Basically, there are three measures against tampering:

- *Tamper evidence* is used to be able to determine if a module has been tampered with, e.g., by using strong security seals.
- *Tamper resistance* means that the module is able to resist or revert attacks without any response.
- *Tamper response* is a reactive measure to tampering and results in zeroisation, i.e., the deleting of all relevant information and keys.

Tamper resistance and tamper responsive measures are used to effectively protect the data by preventing any access to it, while tamper evidence does not provide any kind of protection for the internals:

> *"Note [...] that tamper evidence cannot prevent the disclosure of internal confidential information as well as cryptographic keys. Applications that rely on secrecy of cryptographic keys will never rely on tamper-evident measures if the cryptographic module is not permanently protected."* [15]

This last note concludes the discussion of background information. Any other concepts or terminology is explained or referred to in the context of the appropriate chapter.

CHAPTER 3

An Inquiry into Gateways

Todays information systems more often than not function in a distributed manner, operating over a variety of different communication channels with a further diversity in physical implementations. Gateways play a very important role in such systems, as they are the key ingredient for the required communication between the different parts of the system or even between whole systems. This is especially true for embedded systems, where diversity is not only a requirement but also a main quality of the system.

During the development process, the functionality and requirements to gateways are very often considered as secondary in the sense that their function is considered as a rather trivial part of the overall design. This has lead to an inconsistency in the terminology *"Gateway"*, especially considering what functionality (apart from bridging different parts of a system) is considered to be a major quality of a so called gateway. With the rise of security requirements to every kind of system this needs to change, as gateways in general have a huge impact on security assumptions and overall system's security.

3.1 Towards a consistent notion of "Gateways"

The lack of a consistent notion of what a so called "Gateway" really is can easily lead to misunderstandings during systems design. The current state is that the term "Gateway" is directly associated with its functionality or field of application, such as the following definitions that can be found on *Wikipedia*[1]:

> A gateway is a point of entry or exit.
>
> *Gateway (elecommunications)*, a network node equipped for interfacing with another network that uses different communication protocols.

[1] `http://en.wikipedia.org/wiki/Gateway`; last accessed: September 29, 2012

Gateway (computer program), a link between two computer programs allowing them to share information and bypass certain protocols on a host computer.

The problem with those definitions is, that while the first one is obviously a very generic description that can be applied to every kind of a so called gateway, there is no common computer terminology for gateways. Both, telecommunication gateways and computer program gateways have the same exact functionality and still the definition is not interchangeable. A gateway as a *"point of entry or exit"* may at least be considered a valid first step towards a more useful definition. The following definition is what may be found in a dictionary[2]:

gateway (noun); | ˈgātˌwā |

an opening that can be closed by a gate:

we turned into a gateway leading to a small cottage.

- a frame or arch built around or over a gate: a *big house with a wrought-iron gateway.*
- a means of access or entry to a place: *Mombasa, the gateway to East Africa.*
- a means of achieving a state or condition: *curiosity is the gateway to learning.*
- `Computing:` a device used to connect two different networks, esp. a connection to the Internet.

An opening that can be closed by a gate. Clearly, this statement is way to general for computer terminology and thus has to be refined to some extend. Nevertheless, it allows to derive all of the main characteristics of any gateway that can be found in computer science and that will be discussed further on:

- An *"opening"* means that there are (at least) two entities on either side of the gate.
- The *"gate"* which is posed in between the two entities can be opened *selectively*, i.e., it may or may not be open some of the time.
- A gate is not necessarily *completely* closed or open, allowing only a certain *amount of flow* in between the two entities.

3.1.1 Gateways from First Principles

The first property is at the same time the most fundamental one: basically every kind of communication or interaction between two entities, be it physical or not, happens through a gateway. A vital question is: what kind of entities do we talk about? Clearly one could argue that a communication using a well defined protocol over a common channel between two entities, e.g. two

[2]This definition was taken from the *Dictionary* application of OS X v10.7.1.

controllers over a Serial Peripheral Interface (SPI), does not need a gateway. However, from a philosophical point of view, we could argue that then the translation from the channel (signals on a bus) to one entity (the software of the controller) still must happen - through a gateway. This is equivalent to define the channel as the second entity.

It is out of question that such a discussion is beyond of what could be useful when defining what a *gateway* is in computer science. The "problem" here is a collision with the concept of an *interface* as well as the question of what the basic building blocks are. [6] defines *interfaces* as following:

> "Interface*: A point of interaction between a system*[3] *and its environment. By the environment of a system we mean everything other than the system. At the physical level, for instance, an interface can exist as a single line (a serial port) or as a set of lines (a parallel interface)."* [6]

The common and simultaneously main characteristic of a gateway and an interface is the one of being posed in between two entities and thereby enabling some kind of interaction or communication. We can easily overcome this ambiguity by using or respectively defining the difference between a *point (of interaction)* and an *entity*: An entity is a unit of it's own, with a defined functionality, something complex. Thus an entity can simply not be a *point of interaction*. In general, what we want is a *gateway to be an entity with two interfaces*. Most of the time the term *gateway* is only used if there is a physical or protocol mismatch between the two interfaces but we do not want to restrict ourselves to this. To continue our quest for a definition we may adopt the one for interfaces from [6] to our needs:

Definition 1. *An Interface is a point of interaction between an entity and its environment. The interaction may follow a certain protocol. By the environment of the entity we mean everything other than the entity itself. Furthermore, an interface may or may not be physical.*

The only difference is that we do not talk about *systems* but only of entities and that we mention that the interface is not necessarily physical and may comply to a certain protocol. Before going on with the definition of a gateway it is worth to pick up the above similarity between interfaces and gateways to discuss the fundamental structure of a gateway. It is not very surprising that we can restrict ourselves to *unidirectional* gateways, as every bi-directional gateway can be built out of two uni-directional gateways[4].

Observation 1. *Every kind of interaction can be considered uni-directional.*

Similarly we can restrict the number of entities that make use of the "opening" provided by a gateway to two, by merging or forking the required interactions to more entities. Another point

[3] In [6] a *system* is defined as "an entity that is capable of interacting with its environment and is sensitive to the progression of time."

[4] notice that even Wikipedia defined a gateway as a point of entry *or* exit.

of view would be to just consolidate an arbitrary number of entities on either side of the opening and define it to be a single entity.

Observation 2. *Every kind of interaction can be restricted to two entities.*

This leaves us the following definition that we will use to discuss further characteristics and properties of gateways:

Definition* 1. *An* elementary *Gateway is an entity that acts as an (controllable) interface between two entities A and B, where the interaction between A and B can be considered to be strictly unidirectional, i.e., from A to B. The gateway itself consists of two (unidirectional) interfaces: one to each entity A and B.*

The notion that is adopted here is similar to the one used in [9] for the concept of interfaces: An *elementary* gateway is restricted to unidirectional communication, just as *elementary interfaces* in [9]. In order for this concept to be applicable for more complex applications, we introduce the notion of *composite* gateways analogous to *composite* interfaces in [9] in the sense that it allows for bi-directional communication:

Definition 2. *A* composite *Gateway consists of a composition of* elementary *Gateways which allows the connection of multiple entities and bi-directional interaction.*

Notice that the above definitions and observations are not restricted to computer science: One could talk about a garden that is separated from its environment, e.g. a city, by a wall. Access to the garden is granted by a gate. At this point the philosophical problem to differ between an interface and a gateway may become clear: From the perspective of the garden the gate could be seen as an interface, while for the people it is clearly a gateway. The restriction to an unidirectional gateway is no problem at all: Two gates, each of which has only one handle, can be mounted into the wall such that each gate can only be used to enter *or* to leave the garden.

Now that the main functionality of a gateway has been defined and circumscribed to a simple, unidirectional interaction we can focus on the remaining two major characteristics.

3.1.2 Closing the Gate or leaving it ajar

According to the dictionary definition, the opening *can be closed by a gate*. We have already mentioned that this closing has some impact on how the interaction between the two entities A and B can happen.

As for security reasons we have to assume that we live in a world full of malevolent people[5], let us assume that fully *closing* the gate is actually achieved by *locking*, as no-one in such a world simply would walk past our garden if instead of a lock we would just hang up a sign

[5] otherwise the whole concept of *security* would be pointless.

stating “please do not enter if gate is closed”. Furthermore, our gate is of a special kind that can be locked to certain angles, thus reducing the width of the opening. Those two properties of the gate provide us with two powerful tools:

- *Access Control:* By distributing the appropriate key only to privileged people we can *restrict* and *control* who is able to enter our garden.
- *Flow Control*: If we leave the door ajar then it may become impossible for two persons to simultaneously enter the garden, thus enabling us to *control the stream of people* that can enter the garden.

Both mechanisms have the same purpose: they constrain interactions from A to B. Notice that while a certain amount of flow control is inherent to every physical system, access control is completely optional. On the extremes, both mechanisms are identical: A fully closed gate can either be one where we want zero flow *or* one that is closed and no-one has a key for. The same argument holds for a door that is wide opened.

Furthermore, theoretically the mechanisms can coexist as well as be present without one another: If we threw a party in our garden where everyone is allowed to enter, we just want to restrict the number of people in our garden. This can be achieved with flow control. A private party can be organized purely through the mechanism of access control: only the closest friends and family are allowed to enter (with a key or by being let in by the doorman). Implementing both mechanisms could be useful if we didn’t want to disgruntle our friends by not inviting them and thus stating that we do not have the space for everyone.

3.2 About Firewalls and Bottlenecks.

As a result of Definition* 1 we can restrict ourselves to an unidirectional interaction from entity A to entity B. For the following investigations it may be convenient to further specify how this interaction looks like. In general - or at least in computer science - any kind of interaction can be reduced to *communication*. There are two models for communication in distributed systems[6]:

- *message passing*, i.e., two entities communicate through the exchange of messages, or
- *shared memory*, i.e., two entities communicate through a shared, modifiable object.

Message passing can be completely reduced to shared memory communication. The other way around, i.e., the reduction of shared memory to message passing, is in a general sense not possible as *“write before read”* cannot be modeled by message passing in some formal models[7].

[6] Gateways do only make sense in distributed systems. Again, it heavily depends on what the entities of a system are and thus on the point of view of the designer.

[7] E.g., in a model where every message sent has to arrive eventually. Thus the message would have to be handled. Shared memory would allow for such messages “never to arrive” if it is simply overwritten.

If, however, we restrict ourselves to *unidirectional* communication, then the problem of *"write before read"* disappears, thus:

Observation 3. *Any kind of unidirectional interaction from the entity A to entity B can be reduced to message passing, i.e., the basic purpose of any (primitive) gateway can be reduced selective forwarding of messages.*

Clearly, this notation is not restricted to computer science: The people entering and leaving the garden can be seen as messages too, there is no difference whatsoever. It is now much easier to discuss the mechanisms mentioned before.

3.2.1 Access Control

There do already exist some (consistent) definitions of the term *access control*:

> *"Access control is a service restricting access to resources to privileged entities."* [24]
>
> *"Access control: An embedded system's operations and data should be protected from* unauthorized *access."* [22].

While those definitions are adequate for a higher-level view of systems, they are not suitable for our inquiry into the functionality of gateways. Especially the concepts of authorization and privileges shall be introduced *after* defining what we understand under *access control*. We are in the favorable position that we can restrict ourselves to messages from A to B and thus can define *access control* in the following way:

Definition 3. Access control *is the filtering of messages basing upon information about or contained in messages, e.g. the sender or message-id.*

This mechanism is often associated with the term *firewall*. [22] makes the following statement about firewalls and gateways:

> *"A firewall enforces the access controls. From a system designer's point of view, a firewall is not a single device or a group of devices, but the implementation of a security policy. As a perimeter firewall a firewall acts as a special gateway extending a normal gateway for communication control features."* [22]

While it is desirable not to restrict the term *firewall* to a single device, the argument that control features are an extension to gateways does not fit our definition. The inconsistency here is the separation of the filtering capability from the inherent properties of a gateway. Such a separation cannot happen when talking about security policies: The main purpose of such a firewall (or security policy) is to *prevent* certain messages *from actually arriving* at an entity. According to our definition, an interface is not an entity and can thus not provide filtering functionalities.

Thus the "filter itself" must be an entity. It must have two interfaces: One from the source to the filter, and another one from the filter to the target entity. This coincides *exactly* with what we have defined to be a gateway and thus it makes perfect sense to claim that this is not an extension of the functionality of gateways, but an inherent property. It is clear that this is a very detailed analysis of the above definition of a firewall and that the author may not have wanted to be that specific, but for this inquiry it is really necessary to be exact about what is an inherent property of *any* gateway and what is considered an *extension* to gateway functionality. We will further investigate this classification of primary and secondary attributes when talking about *decoupling and authentication*. Now that we can be certain that *access control* is an inherent property of gateways we can proceed to analyze what such a mechanism could provide:

- access control is a *necessary condition to implement security policies* - or according to the excerpt from [22] it *actually is* the implementation of a security policy.
- *integrity checks* can be regarded as a form of access control, i.e., all messages with a wrong CRC checksum are discarded.
- some very basic aspects of *fault tolerance* can be included into the development of access controls, i.e., faulty (but not malicious) messages are not forwarded.
- in a broader sense, access control can enable *functional degradation*, e.g., if messages contain information that is only provided if the user has paid for it.

The necessity of access control for security policies as well as integrity checks as a form of access control are fundamental security-relevant properties of gateways. The approach to fault tolerance emphasizes that access control is not necessarily a security feature, but also relevant for safety. The last feature, functional degradation, may appear to be a long shot, as the functional degradation may happen at the entity that is the source of the messages. Under certain circumstances, however, such a functionality may be required from a gateway. Consider a GPS-enabled truck. The GPS data is delivered by a telematics unit and distributed to various entities that are e.g. used for fleet-management. Clearly, the GPS data could and is used for navigation purposes too, but as an *optional feature* that can be enabled by the manufacturer. Thus, the last point is only true for systems where functionality and messages are strongly correlated or where information and content *are* the functionality.

3.2.2 Flow Control

Every implemented, non-theoretical system has a limited amount of resources. This means that every entity has only restricted computing power and thus every action or communication is limited by the entity's performance. While being a fundamental restraint of any system, it may simultaneously be desirable to restrict the amount of communication to a certain maximum, e.g.

to be able to estimate the workload of an entity. Notice that this maximum is not necessarily constant, i.e., it may change over time. This mechanism is known as *flow control*.

Definition 4. Flow control *is the filtering of messages basing upon resources.*

Clearly, a "bottleneck" can arise if the sender is faster than the receiver. While both mechanisms, flow control and access control, are filtering mechanisms, the major difference between them is the criteria of filtering: access control relies purely on the message itself. Flow control does not inspect the content of messages but only checks of the defined resources are currently available or not. Thus for flow control it is typically not of importance if e.g. a message is invalid, but only whether or not the current setting for the *bandwidth* supports the transmission of another message. In a system where the receiver is the provider of a service, flow control is responsible for one major property:

- Flow control is a *necessary, but not sufficient* property to guarantee the *availability* of the receiver.

During the design process of an entity a workload assessment is necessary to make any conclusions about the availability of the entity. Especially if the sending entity is highly complex, flow control may be necessary to even render such a calculation possible. The information flow that the receiving side has to tackle then depends on the gateway as it abstracts away from the sending entity. This is especially helpful when trying to find an appropriate fault hypothesis.

3.2.3 Stateful and stateless Gateways

During the design process of a gateway there is often the question whether or not to implement it in a straight-forward stateless way or to consider a stateful implementation. Clearly, a stateless gateway is much easier to implement and thus the hardware requirements for such a component are minimal. There is, however, one thing to consider: While *access control* can be implemented in a stateless manner, there is no way to provide *flow control* without any kind of state information. The least information necessary to provide some kind of flow control would be a *notion of time*. This opens up the possibility to implement a time-driven or *time triggered* flow control, i.e., to enable communication only on specific time slots. We define this control mechanism to be a form of flow control, as it does not depend on the message content but only on the moment of reception.

The next step would be to add a message counter, such that the gateway is able to determine the bandwidth, i.e., the number of messages transmitted per second. Those are the two most primitive possibilities for a stateful gateway. Clearly, the introduction of a state opens up a wast space of possible functionalities, such as a dynamic adaption of access and flow control, but this is not the scope of this inquiry.

3.2.4 Access- and flow control interplay

As already mentioned when introducing the concept of the two mechanisms, access control and flow control can work independently of one another, cooperate, or not be present at all. Thus there are four possible combinations of how those mechanisms can be applied:

A gateway without access- and flow control.
: Such an implementation can be regarded as a *pass-through* or direct translating gateway. With no access- or flow control at all, there is no possibility for neither error- nor fault containment. In computer engineering such gateways are often desirable at higher levels. E.g. in the Time Triggered Architecture (TTA) [10] basic access- and flow control are provided by the architecture itself: The higher-level gateway can rely on the fact that it gets only relevant messages at specific time slots and can thus concentrate on forwarding those messages to e.g. a different network.

Access control without flow control.
: In many applications, pure access control can be of use, e.g., a gateway that filters out all the undesired traffic, such as faulty messages or traffic that is not intended for the receiving side. The message flow of such an implementation is purely restricted by the maximal bandwidth that is supported by the gateway, i.e., flow control is completely delegated to the sending entity. This is a common scenario in engineered systems, e.g. where the sending entity is a network of components and the receiving side needs access to a certain part of the traffic. The traffic in such a network in the failure free case is fully known and thus no explicit flow control from a gateway may be necessary. In case of failure, however, such a gateway cannot provide fault containment of the sending entity by itself (e.g. a *"babbling idiot"* failure).

  Another application of such a gateway may be the connection to an *untrusted* entity, i.e., where the sending entity A is untrusted. The fundamental "problem" here is, that the gateway itself must have some knowledge about the messages. Thus access control implies some kind of redundancy: The knowledge about the traffic is saved both, in the gateway and in the receiving entity. Any inconsistency in this knowledge can lead to problems.

Flow control without access control.
: Pure flow control can be necessary if the sending entity is faster than the receiving entity. Clearly, in a fully engineered system, where the throughput is known, such a gateway is not necessary *iff there are no failures*. In case of failure, however, the malfunction of the sender can propagate to the receiver, as it will no longer be able to handle the traffic. Notice that a gateway with flow control does not necessarily solve the problem, as it can fail too. If, however, the receiving side would be able to handle all the traffic provided by the sender, then such an implementation would just result in a bottleneck.

Interplay: Access- and flow control.

In cooperation, access- and flow control are powerful primitives to control the traffic from A to B. Notice that one is not restricted to first apply access control and then flow control or vice versa: The two mechanisms can be applied in a layered approach (e.g.):

First there is a basic message filter that filters out all undesired traffic and erroneous messages (access control). As a next step, the gateway can determine whether or not the remaining traffic does exceed the defined bandwidth (flow control). If so, another filter could determine which part of the remaining messages should be forwarded and which should be discarded (access control). The described scenario is a possible *priority system*, which cannot be implemented by neither pure access control nor pure flow control.

3.2.5 A final definition

Over the last few sections, a preliminary definition (cf. *Definition* 1*) has been introduced, on behalf of which the properties of *access-* and *flow-control* have been identified as inherent in every gateway. This leaves the following, final, definition of an elementary gateway:

Definition 5. *An* elementary *Gateway is an entity that acts as a controllable interface between two entities A and B, where the interaction between A and B can be considered to be strictly unidirectional, i.e., from A to B. The gateway itself consists of two unidirectional interfaces: one to each entity A and B. The inherent properties of every gateway are* access- *and* flow control*, which embody the* controlling aspect *of interfacing.*

This definition is not restricted to computer science or other kinds of engineering, but is universally usable. The definition of a *composite* gateway does not change, i.e., Definition 2 remains valid for *composite* gateways.

3.3 On the Impact of Gateways on System's Design

Having identified the main qualities of gateways it is advantageous to reflect upon the possible impact of gateways on the overall system design. For this short discussion it is no longer necessary to restrict one-selves to *elementary* gateways, but rather to consider complex, *composite* gateways too. The first observation that can be made may appear to be quite obvious, but is nevertheless mentionable:

Observation 4. *Gateways do generally broaden the attack surface of a system and thus are subject to security investigations.*

This observation follows immediately from the aspect of *Connectivity* in the *Trinity of Trouble* introduced in Section 2.2: Increasing the interfaces to a system does clearly increase the

number of possible attacks. This security issue, however, is only indirectly associated with gateways, as the gateway is only introduced in the overall system because there is a *need* to connect two parts of a system. This connection can happen either directly, i.e., by *expanding* a system with entities that have the same properties, or through a gateway, e.g., if there is a *property mismatch*[8] and thus communication needs translation. In terms of security, an attack via a gateway should be rendered as infeasible as possible. The gateway itself has to fulfill the security requirements of both connecting parties. The following observation comes hand in hand with security requirements:

Observation 5. *If two systems* A *and* B *are interconnected, then a single comprised entity can endanger both systems. Thus, the gateway has to provide appropriate security/safety measures.*

The important fact here is that this observation is not only security relevant, but has also a high impact on the general safety requirements and is independent of safety issues that arise from security problems: Connecting systems (through gateways) does also imply that the fault hypotheses have to be adopted, as a failing entity, e.g., a *babbling idiot*, in one system does also influence the entities of the other one. On the other side, one has to notice that the gateway itself may fail too. Thus, just as it is the case for security requirements, a gateway has to fulfill the fault hypotheses and containment principles of *all the systems it connects to*. This leads us directly to the next observation:

Observation 6. *The impact of a gateway on the overall security and safety of the system directly depends on the underlying architecture.*

When talking about the impact of gateways on system's design, then the observations made do highly depend on the point-of-view of the observer. In general, one has to move to the application-level when considering gateways, i.e., talking about gateways on the architectural level does only make limited sense. E.g., the *Time-Triggered-Architecture* [10] does already make heavy use of gateways on the architecture level in order to inherently provide protection against faults: Every node in the TTA communicates through a *Communication Controller*, which is basically a gateway between the application and the communication channel. Installing an application level gateway in such an architecture does not have the same impact on security or safety as it would have in another one, which does, e.g., not provide basic safety mechanisms at the architecture level. The last observation in this discussion concerns once more about information security:

Observation 7. *A gateway is no warrant for* confidentiality *and* integrity *of information.*

[8] [6] defines a *property mismatch* as *"A disagreement among connected interfaces in one or more of their properties."*

In fact, a gateway is itself a potential security risk and can act as a *man-in-the-middle* (please refer to [22] for details about *man-in-the-middle* attacks) or falsify the content of the information that it is supposed to forward. On the other side, an attacker can *tunnel* information through a gateway, such that it cannot guarantee integrity.

Clearly, those observations are not a comprehensive list of the impact that gateways have on system's design, but it touches the most fundamental issues and findings when talking about gateways. A comprehensive analysis would be desirable, but is out of the scope of this thesis.

3.3.1 Decoupling Gateways

There's one last, important argmuent to discuss regarding the design of the gateway itself: Decoupling. When two entities, which can also be systems themselves, are connected, this connection introduces some amount of control flow between the two entities. Basically, the degree of control ranges from

- *fully dependant*, if the information is forwarded instantly, to
- *decoupled*, if the information is collected from either entity.

This can be basically broken down to *push/pull* style communication: If information is exchanged between two entities A and B, then this information may either be *pushed* from the sender to the receiver, i.e., the receiver is essentially forced to take the data, or *pulled* by the receiver, i.e., the receiver tells the sender that he needs new information. For a gateway this means that

fully dependent control flow implies that the receiving end always gets the information as soon as the sending side delivers new one. This would correspond to direct translation or forwarding of information. The control flow in this case goes from the sender to the receiver, as the sender is able to exert pressure on the receiver. This can lead to problems if, e.g., the receiving side is not able to process all of the information before new one arrives.

complete decoupling of the entities at its interfaces forms the other extreme. This way, either side can provide or collect the information whenever it is able to. It is worth to mention that for the exchange of pure state-information, the gateway may act as a real-time database: The sending side updates the information as soon and as fast as it is able to, while the receiving side retrieves the data when it is able to.

For general information exchange instead of such a real-time database one may implement some kind of queueing system appropriate for the application. Notice that in this case the gateway does not necessarily have to provide overflow handling or such mechanisms, as in a system design cycle the information exchange is typically engineered and thus known up to a certain degree, i.e., the communication does not have to be considered to be arbitrary.

From a system's design perspective, decoupling of the entities can be extremely helpful if, for instance, both participating entities are very complex and thus should only partially be dependent upon each other. Both entities could have their own fault hypotheses and fault containment happens at the entities' boundaries. Clearly, an overal assessment is still necessary, but to a certain degree both entities could be developed independent from one another and would have minimal impact on one another. In general, such a decoupling is preferable over "pass through" implementations that introduce complex dependencies. For some systems, however, such dependencies are at the same time required. In such a case it can be advantageous to gradually increase the amount of coupling between the entities down from a fully decoupled implementation, until those requirements are met.

Thus, all in all, decoupling happens through queueing, where a queue size of one entry is the minimum required for information exchange. Decoupling allows to decrease the dependencies between the entities to a minimum and is therefore desirable, as it reduces the overall complexity of the system.

3.4 Round-up

Basically, gateways can be found everywhere and have always played an integral part in systems design, but the term has always been accompanied by ambiguities. This section has digged deep into the general understanding of gateways and has identified *access-* and *flow-control* as two properties inherent in every gateway. The implications of the lack or presence of each of those properties has been discussed. The definition given for the term *"gateway"* is not a formal one, but allows to address not only gateways in computer science, but also in every other area of application.

Having such a definition at hand, it is much easier to integrate a gateway into a system, as it is no longer necessary to identify what should *not* be provided by the gateway, i.e., what is actually part of the system or application. This is especially true for gateways that connect two distinct systems, such as embedded systems with consumer devices: Both systems are completely independent of one another, but nevertheless it is advantageous to connect those systems - through a gateway. In such a situation, it is important to use the gateway to define the exact borders of the systems and thus to reduce the complexity of the overall design.

CHAPTER 4

Embedded Systems in Automotive

The automotive industry is one of the biggest supporters of embedded systems: Production usually involves high quantities, allowing the use of highly customized systems, e.g., Application Specific Integrated Circuits (ASICs), and top-edge components. On the other hand, a high number of components poses strong requirements on the *costs* of an individual component, as even a small difference in the expenses have a tremendous impact on the overall costs of production.

Two of the currently hottest topics in automotive are *security* and *consumer electronics*: Until now *functionality and safety* have received the biggest attention in the development of automotive systems. *Security* has mostly been considered secondary. Over the last few years, however, the increasing amount of research about communication in- and also between vehicles, new business models and the connection of consumer electronics to cars have urged for more security in vehicles. Regarding consumer electronics, there is still no uniform way to connect and communicate with devices such as smartphones. Projects such as the OpenXC platform presented in Section 2.1 are a first step towards integrating the possibilities provided by consumer devices into the vehicle.

4.1 Overview

The first cars did not involve much electronic components: The power-train was purely mechanical, the motor used a carburetor instead of fuel-injection. Electronics was very simple back then and only used for, e.g., headlights. Starting with an electronic control unit for fuel-injection like the one used for the Volkswagen Type 3 models in 1969, the vehicle has been more and more computerized, allowing for complex systems such as Antilock Braking System (ABS) or Electronic Stability Control (ESC), which today are standard for most new cars.

Today, a modern car can have up to 80 electronic control units [2] and the impact of electronics and software on the overall manufacturing costs is huge: from 30% [2] to 50% [2, 24]. The

connection of all those components results in a network of bus systems. [26] provides an excellent and detailed insight into the bus systems, protocols and applications used in the automotive industry. Figure 4.1[1] depicts a simplification of the basic network architecture of a car:

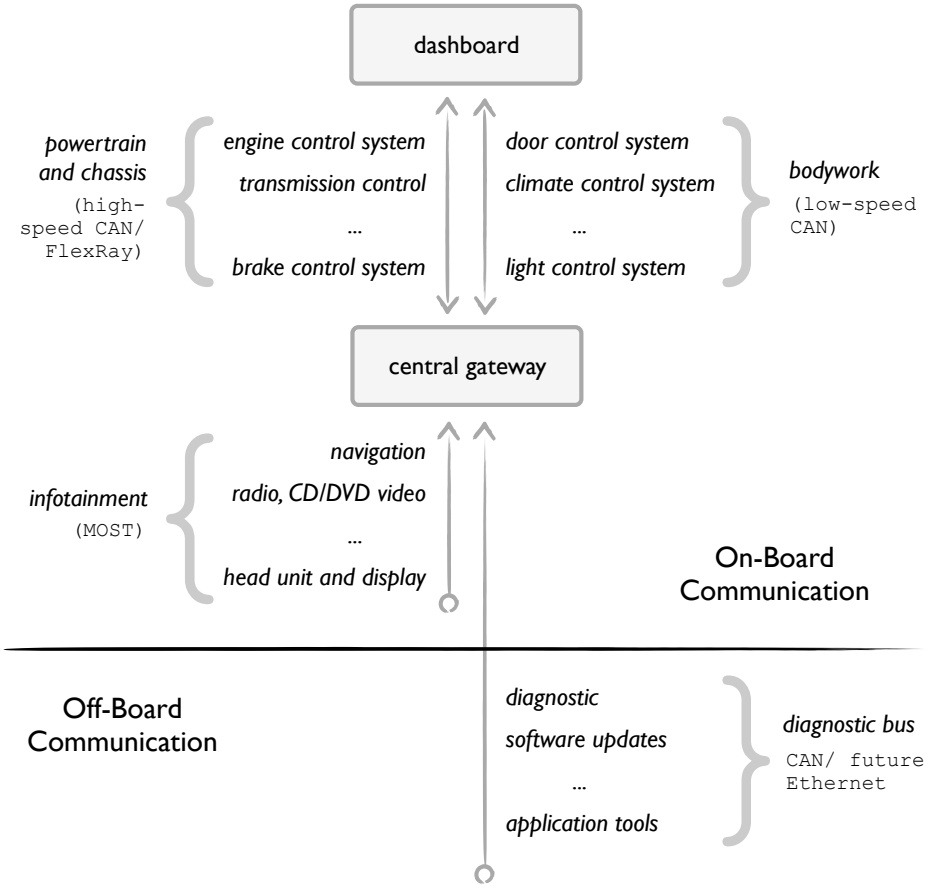


Figure 4.1: Typical network architecture of current vehicles

There are a variety of bus systems used in a car, depending on the application and the needed throughput, as well as physical and safety relevant requirements. Clearly, not every car uses the same bus systems, and there already exist new systems that cannot yet be found in every vehicle, but the most common bus systems are the following:

- *Controller Area Network (CAN)* is currently one of the most used bus systems in automotive, for both low- and high-speed communication (typically 25...125 and 125...1000

[1] This is a simplified version of Figure 1.1 in [26].

Kbit/s) using a message-based protocol. To tackle new requirements posed by modern control systems CAN has been refined to TTCAN.

- *FlexRay* is a bus system that has been developed in the context of *X-By-Wire* systems, such as *Brake-By-Wire*. It is more reliable and faster than CAN, but also more expensive. It is intended to eventually replace the CAN bus especially for safety critical systems and has been standardized as ISO 10681, *Communication on FlexRay*.

- *Local Interconnect Network (LIN)* is an inexpensive, one-wire, serial field-bus system used for sensor and actor networks in vehicle, for instance from the *door control system* to the actual sensors.

- *Media Oriented Systems Transport (MOST)*. In contrast to the previous bus systems, which are used by control systems, *MOST* is a special bus system tailored to the need of multimedia and telematics applications where the main requirement is to allow the transmission of data with high data rates while protecting it from the interferences in a vehicle.

Clearly, the above list is by no means comprehensive: Modern vehicles also include GPS, UMTS and/or GPRS systems, as well as Bluetooth or Wireless LAN and special bus systems required by systems such as the airbag. Furthermore, new technologies such as driver assistance or Human-Machine Interfaces (HMI) urge for even faster bus systems, such as the Gigabit link presented in [12], or *Ethernet* for diagnostic services (cf. Section 5.4).

A remark on embedded system architectures.

A distributed system such as the one in a vehicle offers great possibilities e.g. in terms of reliability, where components should fail independently. On the other side an increasing number of components also increases the complexity of a system (cf. *Trinity of Trouble*, Section 2.2) and poses special challenges in terms of interconnects: In the automotive environment, connector problems are responsible for more than 30% of electrical failures [16]. Reducing the amount of control units by *integrating* different functions into a single control unit could not only decrease the complexity of the system, but would also cut costs and weight, as it would reduce both, the number of components and wires. [16] investigates this approach for multicore System-on-Chips (SoCs) and describes how individual functionalities can be mapped to IP cores instead of individual control units.

This move to a more generic architecture is a general trend and a highly relevant research area: Embedded systems have evolved to complex systems which no longer have to provide only a certain functionality, but do also have to fulfill safety and security requirements, which are very similar for many areas where embedded solutions are needed. Thus, it would be preferable to have a generic architecture for complex embedded systems. The *ACROSS research project* [49] is one approach to tackle this challenge on top of the *GENeric Embedded SYStem (GENESYS) Platform* [62]:

"ACROSS is a research project that aims to develop and implement an ARTEMIS cross-domain reference architecture for embedded systems based on the architecture blueprint developed in the European FP7 project GENESYS." [49]

Clearly, cross-domain platforms have the inherent advantage that research from all the participating domains have influence on the further development. Furthermore, a generic platform such as the one proposed by the GENESYS project, allows developers to concentrate more on the implementation without having to re-consider all the security and safety concepts, as they are provided by the platform. New insights, e.g., in the areas of security and safety will result in new versions of the platform, and existing implementations can directly be ported to this new version while taking advantage of those new properties. In short, generic platforms are definitely the way to go for future complex embedded systems.

4.2 Security in Automotive

In automotive IT systems, security is still only an emerging area. Past implementations used security only for niche applications such as the electronic immobilizer [20] or keyless entry systems, but the general architecture provided only little or no protection at all against malicious manipulation. As apparent from Figure 4.1, however, there is a great need for security not only regarding anti-theft protection such as immobilizers: The amount of communication in vehicles is huge, and the functionality of components and thus also the overall safety of the system rely on the data that is exchanged. Malicious manipulation of this data can not only have economic consequences, but pose extreme risks on the passengers and the environment.

The papers [4, 11] present an experimental analysis of the security in current vehicles on behalf of a representative, moderate priced 2009 model sedan. Basically, [11] is about a full scale analysis of security, which includes physical access to the vehicle before and/or during the attack. With the follow-up publication [4], however, the authors take a step back and analyze the attack surfaces of a car and check if the results obtained in [11] can be repeated, e.g., with short- or long-range attacks including no physical access to the vehicle at all. The results are *alarming*.

Experimental analyses of automotive security [4, 11]

The main exploit that is used throughout the experiments in [4, 11] is to compromise the telematics unit of the vehicle. The reason for this is that this unit has access to both, the high-speed and low-speed CAN buses in the vehicle (cf. Figure 4.1). The low-speed CAN bus is amongst others accessible through the diagnostic port (OBD-II), which is a standardized port used by virtually all vehicle manufacturers (refer to Section 5.1 for further details). Compromising the telematics unit allowed to let it act as a *bridge* between the easy accessible low-speed CAN bus and the high-speed CAN bus used by control modules such as the electronic brake control module. Notice that *bridging* is exactly one of properties of gateways as discussed in Section 3.3 which can

have an enormous impact on the overall security of the system. Once this telematics unit was compromised, it was basically a matter of reverse-engineering of the messages needed to be sent in order to control the braking system of the vehicle, with an alarming result:

> *"We were able to release the brakes and actually* prevent *our driver from braking; no amount of pressure on the brake pedal was able to activate the brakes."* [11]

To be more precise, the authors were not only able to activate or release all the brakes of the car, but they even detected possibilities to lock *individual brakes* and set of brakes. The experiment involves a plethora of other attack scenarios that do not necessarily rely on a compromised telematics unit. Many of them can be executed fully automated, while the vehicle is running, and most important of all: they do not allow for a manual override, i.e., the driver cannot prevent the attack. Such an attack does not necessarily have to involve the braking system, but can also be executed by using systems that are normally considered as not security-relevant. Consider the following scenario: A vehicle could be compromised such that if it is driven in the middle of the night and exceeds, for instance, 90 km/h, it suddenly starts shooting the windshield fluid continuously, disables all auxiliary lights, permanently activates the honk, turns on the windshield wipers and falsifies the speedometer reading. Thus, even rather simple attacks can have a tremendous impact on the safety.

In general, the attacks are possible because of insufficient or non existant security measures. Although some components may include some kind of security measures such as keys, the authors were able to disable or circumvent those precautions rather easily. This is especially true if attackers can actually isolate the components (cf. Section 2.2, *Operation in an untrusted environment*) from the system to analyze it:

> *"If an attacker can physically remove a component from the car, she can [...] reduce the time needed to crack a component's key to roughly three and a half days."* [11]

Notice that this upper bound on the time needed to crack a component is independent of the actual number of components, as they can be cracked in parallel. While the attacks executed throughout [11] required prior physical access to the vehicle, physical access in general is not unrealistic: A serious attacker could have a car of the same make and model to experiment, just as the authors of those publications. But not only insufficient security measures have been identified as possible vulnerabilities: Derivations from standards also open possibilities for attacks:

> *"The standard states that ECUs should reject the 'disable CAN communications' command when it is unsafe to accept and act on it, such as when a car is moving. However, we experimentally verified that this is not actually the case in our car: We were able to disable communications to and from [...] ECUs even [...] while driving on the closed road course."* [11]

As already mentioned, the authors of [11] were not yet satisfied with the results obtained in [11], as they required prior physical access to the vehicle. With [4] they have analyzed possible *remote* attacks. In short: They were still able to compromise the vehicle, even at a distance of over $1,000$ miles (via the cellular network). The experiment includes the following possibilities:

- A *modified audio CD* that is inserted into the media player and allows to send arbitrary CAN messages, while being perfectly played by a PC.
- An attack via *Bluetooth* which is, e.g., used for hands-free applications
- The *Tire Pressure Monitoring System (TPMS)*, which broadcasts tire pressure readings received wirelessly on the CAN bus.
- The *Cellular* connectivity of the remote telematics system, which is typically used for safety (crash reporting), anti-theft (remote track and disable) or diagnostics (to enable an early alert for issues).
- A worm which can be wirelessly installed on *reprogramming/diagnostic tools* used in workshops and compromises any vehicle it connects to. Furthermore this worm is able to spread to other devices in range in a fully automated manner.

Thus in summary they were able to completely compromise the vehicle's system *without requiring direct physical access*. Most attacks could even be fully automated. This does not only pose a safety risk, but also allows to completely bypass the existing anti-theft measures such as the electronic immobilizer [20]. Moreover, the attacker can easily track the vehicle and even use the in-cabin microphone to record data.

Security as an emerging area in automotive

The previous experiments have clearly shown the urge for security in automotive systems which does not only address theft, but also other aspects such as in-vehicle communication. Current automotive systems have not been developed with *security* in mind, and it is a well known fact that adding security features afterwards is typically doomed to fail. In fact, current research in the automotive sector is strongly characterized by security concepts. [2] and [18, 24] provide a good overview over the state of the art and future challenges of automotive security, [23] analyzes the security provided by current bus systems. The following developments are amongst the main hurriers for increased security in automotive systems [18]:

- *Secure Software Updates* and an increasing number of re-programmable ECUs
- *Wireless Communication* with the environment (Car-to-Car and Car-to-Infrastructure, [26] gives a short insight into the topic)

- *New business models*, e.g. for time-limited flash images or pay-per-use infotainment, are under investigation
- *Legislative demands* such as tamper resistant tachographs
- *Privacy concerns* which arise due to an increased collection of data such as driving behavior.
- and of course for *securing the safe execution* of e.g. *X-By-Wire* or existing functionalities.

The authors of [2] also investigate possible approaches for future security architectures, i.e., whether a centralized-, distributed-, or semi-centralized approach would be preferable. They also discuss the Trusted Platform Modules (TPMs) as a secure hardware computing base. Notice that this is also the point where general architectures such as proposed by the GENESYS project come into play. In order to tackle the security problem in automotive, [24] adds the following properties to the security objectives *Availability, Integrity* and *Confidentiality* (as presented in Section 2.2):

- Providing *Hardware and software integrity* implies to render unauthorized modifications to the vehicle's hardware and software infeasible or make them at least detectable by the vehicle.
- *Uniqueness* is guaranteed if unauthorized cloning of a hardware components is infeasible or at least detectable as non-authentic.

Thus, amongst "common" security measures to address, e.g., data security, the integrity of the whole vehicle should be guaranteed. This comes from the fact that upon a large-scale theft countermeasures such as electronic immobilizers are often circumvented by replacing complete parts of the system. *Hardware integrity* would prevent such measures. But regardless of how many security measures there are, it is only possible to prove the presence of flaws, not their absence. Loopholes can even invalidate ambitious and complex security concepts such as hardware integrity. One prominent example is the mafia fraud to bypass keyless entry systems or electronic immobilizers as presented in [20].

Thinking different: The mafia fraud

Keyless entry systems and electronic immobilizers often use wireless transponders to execute authentication protocols, i.e., to enable access to the vehicle. Typically, the range of aforementioned transponders is limited from a few meters to centimeters. This limitation is where the *mafia fraud* attacks: In a common scenario, the vehicle is parked next to the house of the car owner and the keys are typically inside of the house. By using a relaying device, an attacker can establish a radio link between the transponder and the security system installed in the vehicle,

which eliminates the physical distance and allows the security protocol to work as specified and thereby granting access to the vehicle. Thus, although the attacker does not even have physical access to the transponder, he is still able to use it even without the knowledge of the car owner.

Clearly, such an attack could be avoided by e.g. allowing to turn off the transponder or requiring to push a button in order to execute the security protocol, but it is a nice example to demonstrate that even sophisticated security concepts can often be easily bypassed. All in all, security is still a very young area of research in the automotive industry which has yet to establish itself. Generic concepts are often insufficient, as shown by the mafia fraud. Basically, security is needed to improve reliability and to support new business models and should discourage attackers from technical manipulation. Regarding large-scale frauds, security cannot be guaranteed if the attacker is in the possession of valid keys, which can much more easily be obtained through social engineering rather than by cracking the vehicle.

4.3 A remark on consumer electronics in vehicles

Currently there is no uniform interface to connect consumer electronics of different brands to the vehicle while still being able to use a majority of the functionality provided by the device. A variety of accessories provide the possibility to use at least some functions, such as *hands-free*, with practically every device, but specific functions such as accessing the address book of a smartphone is still complicated.

The possibilities, however, are not restricted to telephony services or messaging: Functionalities such as navigation, media and heads-up-displays (HUDs) provide a plethora of possibilities to interact using consumer devices. Already today the maps applications of consumer electronics are much easier to use than the clunky interfaces of special navigators, such that e.g. scrolling in a map of a navigation device is very often cumbersome: Just try to scroll or pan on a device like a TomTom or a navigation system which is integrated into the car, and then switch to a maps application on a smartphone or tablet. In all probability the experience, the usability of the consumer device is on a completely other level. This comes not only from the fact that the touch-screens used in cars or navigation systems are mostly way inferior to the one's of tablets or smartphones, but is also the case because consumer devices are much more sophisticated. For the development in the automotive sector, user interaction through touch-screens or other interfaces is mostly considered secondary.

It only makes sense to directly apply the technologies and progress of consumer devices directly instead of delivering a minor user experience to the customer by implementing a custom application that is somehow integrated into the dashboard. Such applications are a perfect fit for consumer devices. Thus the point is that a car manufacturer should provide a uniform way to connect to consumer devices regardless of brand, as it offers great possibilities for human-machine interaction and should be kept in mind for future automotive applications.

CHAPTER 5

On-Board Diagnostics (OBD) in a nutshell

The early vehicle control systems communicated among themselves (*on-board*) and with the outside world (*off-board*, such as diagnosis devices in repair shops) using analog signals or simple digital communication: The very first *on-board* diagnosis functionality was the Malfunction Indicator Lamp (MIL), a simple lamp on the dashboard which told the driver that there was something wrong with the vehicle - without being specific *what* exactly was wrong with it.

Figure 5.1: The Malfunction Indicator Lamp (MIL) on the dashboard of a VW Tiguan

The huge progress in computer technology has also opened new ways for the automotive industry: One of the first digitalized systems was the powertrain, which is now controlled by a powertrain control module (PCM) and among other things controls the air/fuel mixture, ignition

timing and idle speed. Later, more and more electronic systems such as ESP, the electronic immobilizer, airbags and multimedia systems were introduced - interconnected through various bus systems such as CAN, LIN, FlexRay or MOST - such that nowadays there is a plethora of electronic control units (ECUs) in every car.

In order to make the on-board diagnostics information, i.e., the information that can be obtained from those control units, useable for repair, it somehow had to be communicated to the outside world. Furthermore, the increasing amount of electronic control units (ECUs) made it necessary to introduce an interface that allowed easy flashing of components, as it was no longer feasible or even possible to update every electronic unit individually. Therefore, nearly every manufacturer introduced a so called *diagnostic interface or port*. This diagnostic interface is often also referred to as the *on board diagnostics (OBD) interface* The information that is nowadays provided through *on-board* diagnostics is often sufficient for repairs. The traditional *off-board* diagnosis information, e.g., measuring currents, checking for slack joints or other actions performed by specialized technicians, is now mostly available through on-board diagnosis such that it can be said that *on-board* and *off-board* diagnostics are blending.

In the beginning of the digitalization, the physical interface, protocols and application of on-board diagnosis was manufacturer specific. Repair shops had to use different devices depending on the manufacturer of the car in order to get repair information or to update the car's software.

5.1 Towards a standardized diagnostic port

The main hurriers for a standardized, uniform diagnostics port were on one side the need to get emission-related data to check for compliance to the provisions of law (on board diagnostics - OBD) and on the other side the increasing cooperation of manufacturers as a part of the globalization process which pushes manufacturers to cooperate in the development of new components or even whole cars. Concerning emission related data, the general idea was that cars not only had to pass emission tests upon admission, but also guarantee the compliance to the emission regulations during lifetime, e.g. through emission control systems whose results could be read out at annual revisions. This first step was made by the California Air Resources Board (CARB) to address its smog problem in Los Angeles. It required that starting with 1991 all new vehicles sold have basic emission-related diagnostics capabilities and systems [54]. The term *on-board diagnostics (OBD)* itself was introduced in retrospect after the introduction of the OBD-II standard and from then on referred to *emission related diagnostics* instead of overall on-board diagnosis. The following is a list that should give a rough overview over actual OBD standards and notations:

OBD-I was the original attempt of the CARB to force minimum requirements from manufacturers for annual emission testing. The problem with this standard was that the information exchange itself has not been standardized thus leading to technical difficulties and different implementations.

OBD-1.5 emerged from a partial implementation of the OBD-II standard by General Motors, although GM themselves did never use the term OBD 1.5.

OBD-II is the first comprehensive standard regarding on-board diagnostics (OBD). Not only does it provide the necessary information for implementation (such as the actual connector), but it is also consistent on the application level between Europe and America.

EOBD is the European equivalent to OBD-II (European On-Board Diagnostics) in terms of technical implementation and specification. All newly introduced car models starting with the year 2000 do have to comply to this standard.

EOBD2 is - not as expected - an extension to the European standard, but is used by some manufacturers to address an extended implementation of the OBD-II standard (Enhanced On-Board Diagnostics II).

JOBD is a Japanese version of the OBD-II standard.

The main problem of the standardization process is that it cannot always keep up with the technology, which often leads to overlapping standards. Furthermore, the need of backward-compatibility to older vehicles manifests itself in the actual protocols and standards used.

5.1.1 Emission-Related Diagnostics and OBD, a terminological mismatch

As explained before, the term *on-board diagnostics* in general refers to the automotive system's self-diagnosis capability - without any restriction on *what exactly* is diagnosed. With the introduction of legislated emission related tests, however, the term *OBD* has been associated with emission-related diagnosis: The so called *on-board diagnostic* standards do only require the implementation of emission-related diagnostics. The standards do not define any other functionality such flashing and other (on-board) diagnostic information. For convenience, however, most manufacturers have made the required OBD port the only one available in the vehicle and use it e.g. for programming and the diagnosis of all control systems (c.f. Figure 5.3 on Page 40, unmarked pins are manufacturer specific).

Because of this ambiguity for the terminology in the use of the term *on-board diagnosis* in the rest of the text we will exclusively use the term *diagnostic port* to refer to the physical port of the car while adopting the usage of *on-board diagnostics* for *emission-related diagnostics*.

5.2 Standards overview

Throughout the development of OBD-II, a variety of protocols and thus standards have been introduced and revised. The list on page 37 gives an comprehensive overview about the european standards and protocols involved in the process of getting to the OBD-II standards bundle.

Although all those protocols are intended for the same purpose, inaccurate or non-existent specifications often lead to only slight differences in the details of the implementations and thus to different variants, which can actually be incompatible [26].

ISO 14230: K-Line Key Word Protocol 2000 (KWP 2000)

The first standard that has been adopted by all manufacturers for the diagnostic interface was the "K-Line Key Word Protocol 2000 (KWP 2000)" (ISO 14230) which has been introduced in 1999. It includes the whole protocol stack for serial communication over a K-Line bus system (refer to [26] for more information about K-Line bus systems). The main contribution of this standard was the definition of the communication model and *diagnostic services* between the tester, a device that is connected to the diagnostic port to retrieve data, and the electronic control units (ECUs), i.e. the vehicle. The syntax and semantics of the diagnostic data, however, is not part of the standard. Furthermore, the standard explicitly excludes any definitions about emission related systems (OBD). Only later, in 2000, the fourth part of the standard introduced the *requirements for emission-related systems*.

ISO/DIS 15765: KWP on Controller Area Network (CAN)

Although never actually adopted - because it was replaced by UDS on CAN (ISO 15765-3) before being officially released, see next bullet - the *KWP on CAN* standard is currently one of the most widely used implementations of diagnostic protocols. It encompasses the widely unaltered porting of the Key Word Protocol to the Controller Area Network (CAN), i.e., the content regarding diagnostic services is substantially the same as in the previous ISO 14230 K-Line KWP 2000 and has thus been referenced in this standard. With [83] all vehicles sold in the US after 2008 are required to implement CAN as one of their signaling protocols.

ISO 14229: Unified Diagnostic Services (UDS)

This standard was originally developed under the title of *Diagnostic Systems - Diagnostic Services Specification*, which was then replaced by *Unified Diagnostic Services (UDS)*. The goal was to provide a uniform application layer for on-board diagnostics, which was independent of the underlying bus system in order to be able to make integration of e.g. FlexRay for OBD easier. UDS is regarded as a further stage in the development of diagnostic protocols.

Concerning the basic set-up such as message format and services, UDS is essentially identical to the KWP 2000. The major differences are that the diagnostic services have been re-grouped and new ones have been introduced that could not have been used with a K-Line bus system. Those and other small changes to the KWP 2000 make UDS functionally but not implementation-wise compatible to the KWP 2000.

ISO 15765-3: UDS on CAN

With the introduction of UDS the development of *KWP on CAN* has been stopped and an actual implementation of the UDS on the CAN bus system has been introduced: *UDS on CAN*. With this update to UDS, the transport layer had to be revised too (ISO 15765-2), as e.g. UDS needs messages of more than 8 bytes of payload. OBD related content is specified in part four of ISO 15765.

ISO 15765-1 to -4 together with ISO 14229-1 form the standards-bundle for *UDS on CAN*, which is intended to replace KWP 2000 on CAN. The implementation of UDS on CAN, however, is very complex, wherefore manufacturers usually only implement a subset of the standard.

ISO 15031: Emission-related diagnostics

In contrast to the previous protocols like K-Line KWP or UDS on CAN, which basically describe the communication protocol, this standard defines the diagnostic services and how they are mapped to the underlying communication system (such as UDS on CAN, i.e., CAN is used as underlying bus systems with a transport protocol that supports flow control and variable frame sizes).

This standard defines the actual services that are required on behalf of the law for emission-related diagnostics and is currently known as *On-Board-Diagnostics (OBD)*. There are seven parts which define the diagnostic connector, the communication with the tester, the required services and the interpretation of the data. There exist equivalent standards for all but the first part (General Information) in the USA (c.f. Table 5.1), published by the Society of Automotive Engineers (SAE). Possible underlying bus systems for this standard are CAN (which starting with 2008 is required by all vehicles in the USA), K-Line and J1850 (Pulse Width Modulation PWM or Variable Pulse Width Modulation VPWM).

The *OBD protocol* does only require data about ignition, fuel injection, oxygen sensors (lambda probe) and the catalytic converter. The data has to be accessible through an error memory, requestable readings and as results of observation tests. Other capabilities, like flashing/programming or additional readings, are *not* part of the OBD standard but nevertheless required by most manufacturers and delegated to the diagnostic port (as it is the only port available in the vehicle).

In summary, all those diagnostic protocols use the same basic concepts and implement the same functionality with respect to certain fields of application (such as OBD). K-Line KWP was the first protocol that was used for on-board diagnostic. It was ported to CAN (as KWP on CAN) and later replaced by UDS. The ISO 15031 standards bundle defines the services, e.g., the readout of trouble codes or current powertrain data such as the vehicle speed, that have to be supported by a vehicle that complies to the OBD standards bundle. Those services are executed

on the underlying bus systems, which can e.g. be the KWP on a K-Line bus system or the UDS on the CAN bus.

Figure 5.2 depicts the progress in the standardization for diagnostic protocols (a comprehensive list of the european standards can be found on page 37). The first thing to notice is that the standardization process is way too slow-paced to keep up with the rapid development of vehicles: First mentioned in 2001, the ISO 15031 standard (emission-related diagnostics) introduced part 5 - which defines emission-related diagnostic services - in 2006 and thus five years later. Four of the seven parts had been withdrawn until 2011, such that the current development time for this standard is ten years - an eternity in the fast-paced automotive sector.

One manifestation of this problem is the choice of the CAN transport protocol: The K-Line KWP 2000 protocol that had been used for on-board diagnostics did only define messages with a payload of at most 8 bytes, which fits into every CAN message and thus porting of KWP was possible and has been considered (KWP on CAN, although rejected later). For manufacturers, however, single frames were not sufficient as for example flashing requires a more complex transport protocol that supports frames with a larger payload. Therefore they had to use other transport protocols for communication via CAN, such that now there are three commonly used protocols:

- ISO-TP as later defined by ISO 15765-2,-4 (Diagnostic on CAN)
- CAN TP 1.6 or TP 2.0 which is e.g. used by Volkswagen/Audi
- SAE J1939/21 for CAN which is mostly used by utility vehicles.

As a consequence, Volkswagen/Audi has to provide OBD data using the ISO-TP while using e.g. CAN TP 2.0 for internal communication.

The second important observation is the adoption of the CAN bus system for diagnosis that happened in parallel with the OBD standardization. Thus, nowadays most manufacturers provide diagnostic data on the CAN bus of the diagnostic interface (Diagnostics on CAN, DoCAN, represented by ISO 15765-4 and ISO 14229-1). What is unapparent from this figure and list is the cooperation between the USA and Europe towards a global standard (the cooperation between ISO and SAE). To put it briefly, for ISO 15031-2 to -7 (part 1 contains only general information) there exist nearly coextensive SAE standards such that emission-related diagnostic is pretty much a global standard.

Standards introduced for on-board diagnostics (grayed out standards have been withdrawn).

- *ISO 14230*; Road Vehicles - Diagnostic Systems - Key Word Protocol 2000

  1999: Part 1; Physical Layer
  1999: Part 2; Data-Link Layer
  1999: Part 3; Application Layer
  2000: Part 4; Requirements for emission-related systems

- *ISO 15765*; Diagnostics on Controller Area Networks (DoCAN)

  2004-2011: Part 1; General Information
  2004-2011: Part 2; Network Layer Services (KWP on CAN)
  2005-2011: Part 4; Requirements for emission-related systems

  2011: Part 1; General Information and use-case definition
  2011: Part 2; Transport protocol and network layer services
  2004: Part 3; Implementation of Unified Diagnostic Services (UDS on CAN)
  2011: Part 4; Requirements for emission-related systems

- *ISO 14229 [1998-2006]*; Diagnostic Systems - Diagnostic Services Specification

- *ISO 14229 [2006]*; Unified Diagnostic Services (UDS)

  2006: Part 1; Specification and Requirements

- *ISO 15031*; Road Vehicles - Communication between vehicle and external equipment for emission-related diagnostics

  2001-2010: Part 1; General Information
  2004-2010: Part 2; Terms, definitions, abbreviations and acronyms
  2006-2011: Part 5; Emission-related diagnostic
  2005-2010: Part 6; Diagnostic Trouble Code definitions

  2010: Part 1; General Information and use-case definition
  2010: Part 2; Guidance on terms, definitions, abbreviations and acronyms
  2004: Part 3; Diagnostic connector and related electrical circuits, specification and use
  2005: Part 4; External test equipment
  2011: Part 5; Emission-related diagnostic services
  2010: Part 6; Diagnostic Trouble Code definitions
  2001: Part 7; Data-link security

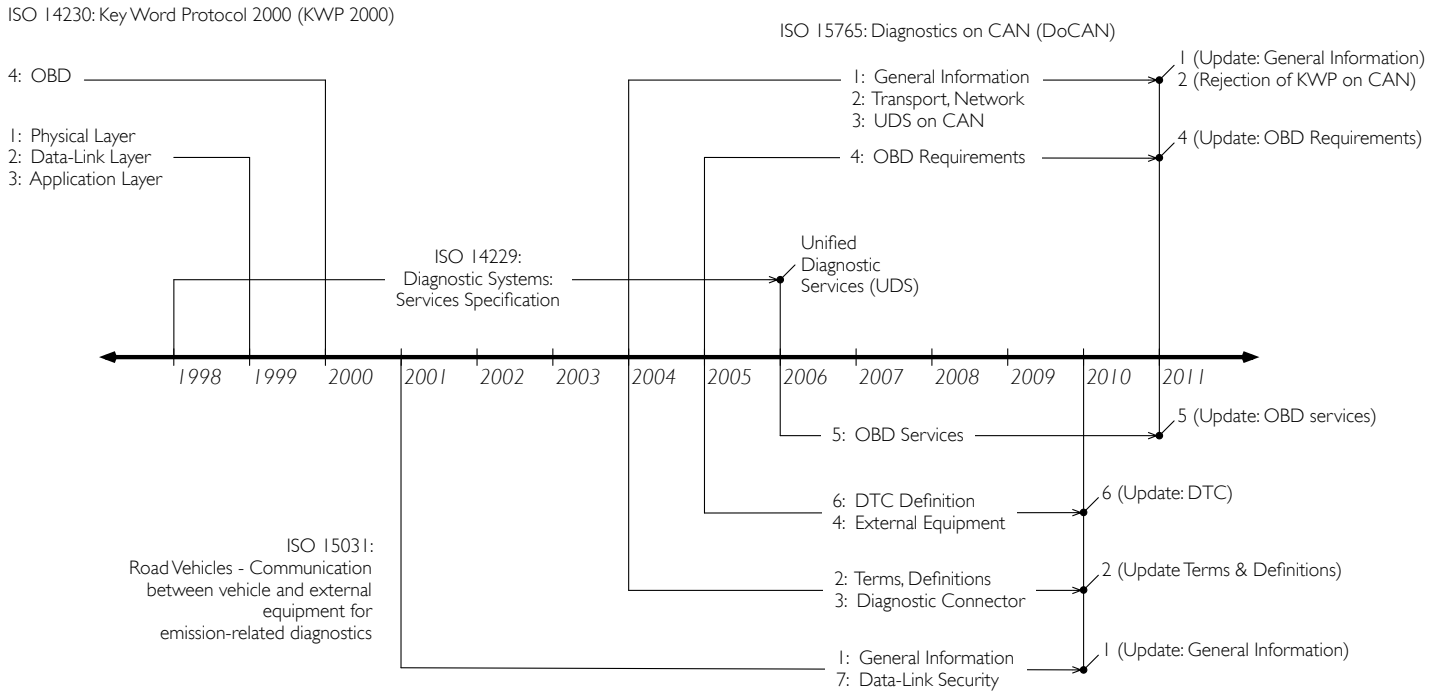


Figure 5.2: Protocol timeline for the OBD-II bundle standards.

Timeline of the protocols presented on page 37. The descriptions are shortened and correspond to the current standard with given protocol part-number. Withdrawn protocols are marked with dots and the part-number that has been withdrawn together with a short description of what has been done. Data acquired from *ISO* [65].

5.3 On-Board Diagnostics

What is nowadays refered to as *On-Board Diagnostics* is a bundle of standards that has emerged from the need for emission-related tests required for vehicles to comply to the provisions of law of Europe (EOBD) and the USA (OBD-II). Table 5.1 shows the standards that are included in the OBD bundle for Europe (ISO) and the USA (SAE).

Content	ISO	SAE
General information	ISO 15031-1	-
Terms, definitions and abbreviations	ISO 15031-2	J1930
Diagnostic connector	ISO 15031-3	J1962
External test equipment (tester)	ISO 15031-4	J1978
Diagnostic services	ISO 15031-5	J1979
Diagnostic Trouble Codes (DTCs)	ISO 15031-6	J2012
Data Link security	ISO 15031-7	J2186
Underlying bus systems	ISO 9141-2 K/L-Line CARB ISO 14230-2 K-Line KWP 2000 ISO 15765-2/-4 CAN SAE J1850 PWM/VPWM	

Table 5.1: Standards included in the OBD diagnostic protocol [26]

Thus for Europe, the protocols from ISO 15031 define all but the underlying bus system required for OBD. The mapping of the services to the communication protocol (i.e., the mapping of the actual data that forms the request or response to the messages used by the communication protocol) is defined in ISO 15031-5. Section 7.6.3 goes more into detail about the actual implementation using the CAN bus.

5.3.1 The diagnostic port

The actual port used for diagnostics is called *OBD-port* because it is specified in the ISO 15031-3 standard. It is important to keep in mind, however, that this port is *not* dedicated to OBD, i.e., to emission-related diagnosis. Manufacturers can still use unassigned pins or execute other protocols. The afore mentioned diagnostic protocols coexist on this port, such that both K-Line KWP 2000 and UDS on CAN could be supported by the vehicle. Figure 5.3 depicts the diagnostic connector as defined in ISO 15031-3, the unassigned pins are manufacturer specific. The location of the diagnostic connector is not fixed but *"shall be out of the occupant's (front and rear seat) normal line of sight, but easily visible to a crouching technician"* and *"shall not require a tool for the removal of an instrument panel cover, connector cover, or any barriers"*

[ISO 15031-3]. Typical locations are on the driver's side, underneath the dashboard in the area under the steering column or mounted on the center console.

As already mentioned, starting with 2008 all OBD certified vehicles sold in the US are required to use the ISO 15765-2,-4 communication protocol [83], which is *Diagnostics on Controller Area Networks (DoCAN)*, i.e., OBD is possible through the pins 6 and 14, which are the high and low lines of the CAN bus system.

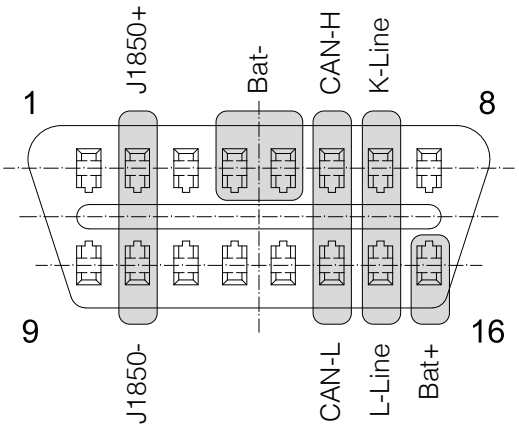


Figure 5.3: OBD diagnostic connector according to ISO 15031-3

Figure 5.4: The diagnostic connector of a VW Tiguan

5.3.2 Diagnostic Services

The diagnostic services are defined by ISO 15031-5 for every of the aforementioned bus systems. The differences between the implementation on the different bus systems are only related to the transport protocol, e.g., K-Line uses other timings than CAN and is restricted to single-frames and thus of messages of constant size, while the ISO-TP (ISO 15765-2/-4) protocol for the CAN bus system allows messages of up to 4096 bytes. In terms of semantics, the services are equivalent. Basically OBD-II/EOBD data is limited to

- problems related to the Malfunction Indicator Lamp (MIL)[1] and
- emission diagnostics.

Thus other services like parametrization, control module diagnostics or flashing are still manufacturer specific and can not always be accessed through OBD. Much of the powertrain data, such as the vehicle speed, engine RPM (revolutions per minute), ambient temperature, are emission-related and can thus be obtained through OBD. Two examples of data that is *not* regarded as emission-related and thus not accessible through OBD are the total distance travelled or the current trip distance. The distance travelled while the MIL was active, on the other hand, can be obtained through OBD. ISO 15031-5 defines the following diagnostic services (the grayed out service `0x0A` was only introduced in 2011):

Service `0x01`; Request current powertrain diagnostic data

Service `0x02`; Request powertrain freeze frame data

Service `0x03`; Request emission-related diagnostic trouble codes (DTCs)

Service `0x04`; Clear/reset emission-related diagnostic information

Service `0x05`; Request oxygen sensor monitoring test results

Service `0x06`; Request on-board monitoring test-results for specific monitored systems

Service `0x07`; Request emission-related diagnostic trouble codes detected during current or last completed driving cycle

Service `0x08`; Request control of on-board system, test or component

Service `0x09`; Request vehicle information

Service `0x0A`; Request emission-related diagnostic trouble codes with permanent status

[26, Table 5.3.2, Page 174] introduces a categorization of the services into error memory, test of emission-related components and requests of information from control units:

[1] Notice: The actual status of the MIL is *not* directly related to the MIL status that can be obtained through OBD. E.g., the MIL is on for a few seconds when the ignition is turned on while the actual MIL status is off.

Error Memory

Service `0x03`; Request emission-related diagnostic trouble codes

Every time a monitor detects an emission-related fault a so called Diagnostic Trouble Code (DTC) is stored (which in turn causes the MIL to light up). The definition of the DTCs (ISO 15031-6) has been chosen such that manufacturers can use the same format for their own diagnostics too. Every trouble code can be assigned to either the *body (B)*, *chassis (C)*, *network (U)* or *powertrain (P)* of the vehicle and has either its own subfault strategy or can be assigned to one of the four basic categories *circuit/open*, *range/performance*, *circuit low (shorted to ground)* or *shorted to battery positive*. Examples for DTCs are:

- `P0070`: Ambient Air Temperature Sensor Circuit Low
- `U0158`: Lost Communication With Head Up Display
- `B0050`: Passenger Seatbelt Sensor (Sub-fault)
- `C0081`: ABS Malfunction Indicator (Sub-fault)

Service `0x0A`; Request emission-related diagnostic trouble codes with permanent status

DTCs obtained through Service `0x03` can be cleared, e.g., after a successful repair (see Service `0x04`). The purpose of permanent DTCs is to *"prevent vehicles from passing an in-use inspection by simply disconnecting the battery or clearing the DTCs with a scan tool prior to the inspection"* [ISO 15031-5]. Permanent DTCs are cleared by the OBD system itself if it has determined that the malfunction that caused the DTC is no longer present, by clearing the fault information in the ECU or if the ECU containing the DTC is re-programmed.

Service `0x07`; Request emission-related diagnostic trouble codes detected during current or last completed driving cycle

This service allows to retrieve the DTCs that have been saved during the last driving cycle, which can be useful for technicians performing a test drive. Those DTCs are not necessarily also "confirmed", i.e., they may not yet be considered as a fault by the monitors and are ignored by the services `0x03` and `0x0A`. We will not go into detail about the so called "pending" status here. It is enough to mention that a malfunction is de-bounced before being considered as "confirmed".

Service `0x02`; Request powertrain freeze frame data

For every DTC that is saved there exists a so called *freeze frame* which captures the engine conditions, such as vehicle speed, intake air temperature or engine RPM at the time the DTC is set. By duplicating these conditions a technician can, e.g., verify a repair.

Service `0x04`; Clear/reset emission-related diagnostic information

This service allows to clear the saved DTCs (without permanent status) as well as

the corresponding freeze frames and other emission related data such as the distance travelled while MIL is activated or system monitoring test results.

Test of emission-related components

Service `0x05`; Request oxygen sensor monitoring test results (see below)

Service `0x06`; Request on-board monitoring test-results for specific monitored systems
ISO 15031-5 defines various tests that can be performed on emission-related sensors or systems. Service `0x05` is exclusively used for oxygen sensor tests. The same results can be obtained through Service `0x06`, which allows to retrieve test results of other components too.

Service `0x08`; Request control of on-board system, test or component
This service allows to control the in- and outputs of components, systems or to perform certain tests. It is specifically used to test the tank ventilation.

Request of information from control units

Service `0x01`; Request current powertrain diagnostic data
Current emission-related data can be retrieved through this service: ISO 15031-5 defines various Parameter IDs (PIDs) that can be requested from the vehicle. Through those PIDs a test equipment can also determine which PIDs are actually supported by the vehicle and subsequently request the actual value.

- `PID 0x00`: Supported PIDs `0x01` to `0x20`
- `PID 0x20`: Supported PIDs `0x21` to `0x40`
- `PID 0x01`: MIL status, number of emission-related DTCs, supported tests
- `PID 0x0D`: Vehicle Speed Sensor
- `PID 0x1F`: Time Since Engine Start
- `PID 0x21`: Distance Travelled While MIL Is Activated

Service `0x09`; Request vehicle information

According to ISO 15031-5 *"the purpose of this service is to [...] request vehicle-specific vehicle information such as Vehicle Identification Number (VIN) and Calibration IDs."*

In summary, OBD defines an error management system, allows to retrieve real-time data from the vehicle, perform various tests and the readout of vehicle information such as the Vehicle Identification Number (VIN).

5.4 Concluding remarks and future

This chapter may have made apparent that there is still much work to do in the standardization of the communication of on-board information to off-board devices. In summary most vehicles nowadays use CAN as the bus system for diagnostics. The ISO 15765 and ISO 15031 standards are used for on-board-diagnostics all over the world.

The next step towards a world-wide uniform OBD standard is already in progress: The World Wide Harmonized On-Board-Diagnostic (WWH-OBD) is currently being developed. Together with UDS Diagnostic Services (ISO 14229-1) the WWH-OBD should be a uniform diagnostic interface for both, on-board diagnostics and general diagnostic services (such as required for maintenance, flashing etc.). One major priority for this uniform diagnostic interface is that it shall be completely independent from the underlying communication system and depend only on the UDS Session Layer Services (ISO 14229-2). Figure 5.5 depicts the structure of the protocol stack for WWH-OBD:

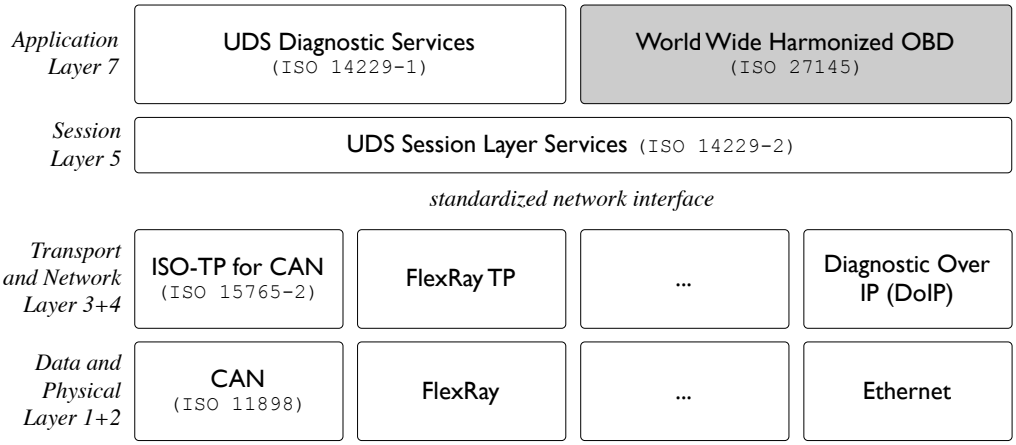


Figure 5.5: Future structure of the diagnostic protocol stack (Figure 5.4.2 [26])

What is also apparent from Figure 5.5 is that future diagnostic protocols will use TCP/IP over ethernet or wireless technologies for communication. The two main motivators for a switch to TCP/IP is on one side the increasing bandwidth and on the other side economic reasons. For further information about the WWH-OBD standard, [78] is a presentation of the Society of Automotive Engineers (SAE) and gives a good overview over the changes, while the official proposal of the United Nations (UN) [82] goes more into detail.

CHAPTER 6

Interfacing Consumer Electronics

Since the introduction of Apple's first generation iPhones in 2007, the smartphone and tablet market has figuratively exploded such that today there exist a plethora of *consumer devices* of different brands. Android and Apple devices make up for the majority of the market share and both provide excellent developer tools and "App Stores" to motivate developers from all over the world to contribute by writing their own Apps - paid or for free.

Even now, five years later, new generation products such as the iPhone 5 are still sold millionfold within the first 24 hours of introduction. Also, the sheer flood of third party applications does not seem to come to an end. In fact, we're still only scratching the surface of the possibilities provided by consumer electronics. The *SmartThings* platform presented in Section 2.1 is only one example of a whole area of application that has yet to emerge.

6.1 General Observations

Regarding (deeply) embedded systems, this huge development in consumer electronics has awaken our desire to connect ourselves with everything that surrounds us and is a natural step towards the so called *Internet of Things*. Such systems very often run autonomously, hidden from plain sight, and there's hardly any user interface to those systems, especially not for common users. Being able to access such systems, however, would often be very convenient. Consumer electronics could possibly fill this gap of a lacking interface and could thereby enable interesting possibilities for embedded systems designers not only regarding human interaction. Connecting embedded systems such as sensors and actors to, e.g., smartphones or tablets, however, also brings along other implications that should be regarded during the system design phase. The major observations made when considering to connect systems to consumer electronics are the topic of this section. The first observation concerns the consumer device as a *possible uniform interfacing device*:

Observation 8. Usability*: Consumer electronics such as tablets and smartphones offer a convenient and easy interface to users, e.g., through uniform interface elements.*

Older people, or people with only little or no experience at all in dealing with computers or other information technology, are often overwhelmed even by the simplest interfaces such as thermostat regulators or common remote controls. What is often considered as "intuitive" or easy to use by developers just isn't for them. In some cases the reluctance to handling something new is just too high such that some might even refuse to use new technologies. Smartphones and tablets open up an opportunity here: When confronted with a *single* device used in everyday life[1], the learning curve for new technologies is much lower. Access to new technologies that happens through *Apps* on consumer devices can take full advantage of familiar user interface elements and thus tremendoulsy increase the usability of the product. It is much simpler than having to use a variety of devices to control different products. The next observation concerns implications about the usability when consumer electronics are used for user interfacing:

Observation 9. *For consumer devices as "user interfaces", the* usability *of the application running on the device is of utmost importance. The user experience provided by the consumer device should be preserved.*

The point is that an application that is not well-conceived can completely quash the usability gain provided by the consumer devices as user interface. If consumer electronics are used as user interfaces, it is very important that built-in security and usability features of the consumer device are extensively used, such that the original user experience is preserved. The next important observation may seem trivial, but is nonetheless important to mention:

Observation 10. *Connecting to consumer electronics can* severally amplify the attack surface *of the connecting device.*

The point why this observation may seem trivial is that any kind of augmentation of the interfaces of a device clearly increases the number of possible attacks (cf. Connectivity in the *Trinity of Trouble*, Section 2.2). One reason why it is mentioend anyways, is that not only the increase of interfaces is of concern. Smartphones and tablets are extremely easy to use and *developing applications* for such platforms is easy too. This ease of access to develop software for such devices opens up a possibility for abuse to a great number of people that would otherwise have never have had the possibility to compromise the system. It is thus also more probable that the connected system is compromised through the consumer device by the sheer fact that there are more people that *simply can* do it. While this may appear to be a long

[1]Clearly, one cannot assume that everyone uses a smartphone, but it is way easier to introduce someone to a smartphone then to other single-purpose devices, especially because this is a one-time learning experience: One does not have to learn the same thing twice if it is extensively used. The only thing that changes for smartphones are the *Apps*, which should stick to uniform interface elements.

shot for some applications, it is definitely something that has to be considered for highly security and safety relevant applications. Clearly, the fact that the consumer device may introduce new interfaces to the system is also relevant, especially when considering wireless communication. The consumer device can act as a gateway to the embedded system and its wireless interface can completely eliminate the need for physical access to the system in order to compromise it. The next observation is basically a consequence of this discussion and is theoretically not restricted to interfacing with consumer devices, but is of general concern:

Observation 11. *Interfacing a system with consumer electronics should by no means provide an easier interface to compromise the system.*

Consider the following scenario: A common vehicle shall be enabled to communicate with consumer devices to enable third party applications, e.g., to track consumption or fleet management. The data that is sent to the consumer device is retrieved from the vehicle's internal electronic control units, which may reside on different buses. Nowadays, the OBD port already provides some services that could be used for such applications, but retrieving information through this port requires a detailed knowledge of the bus systems and various standards and protocols. Thus, without the possibilities of a consumer device, any kind of attack would require this knowledge. The point is not to make the communication with the consumer device at least as complicated as the task to directly retrieve data from the electronical control units, but to ensure that this communication is safe and secure, e.g., that it is not possible to flood the vehicle's internal buses via a consumer device. Decoupling interfaces (cf. Section 3.3.1) is only one of the mechanisms that can contribute to fullfil this requirement, while a feed-through implementation, e.g., a direct translation of wireless requests to CAN frames, may simplify attacks to the underlying systems. Notice that this has already been discussed in Section 3.3: The consumer device in this case acts as a gateway to the system. The security measures should be appropriate such that compromising the system via the gateway should not be easier than directly compromising it. While security and usability clearly play an important rule when regarding the possibility to connect a device to consumer electronics, there is one thing to notice above everything else:

Observation 12. Availability *is of utmost importance to users. If a device cannot or does not do what it is supposed to do, then the reason for failure is only of marginal importance.*

Availabilty is something that has to provided by the overall system, i.e., it involves both, the consumer electronics and the system it connects to: If a consumer device is used to interface the system, then it still can't do any good if the system is not available. On the other side, the application running on the consumer device should be well-matched to the system, which could e.g. become temporarily unavailable. The main objective of the consumer application is to remain *responsive* as effectively as possible and to *guide* the user even if the system is unavailable.

6.2 On available interfaces

Todays consumer electronics offer a variety of possibilities to be addressed from a system. Clearly different devices include different features, depending on the size of the device as well as on the manufacturer. When considering to connect a system to consumer devices it is of great importance that it is supported by the the majority of those devices - regardless of the brand. The following interfaces are provided by the majority of current smartphones and tablets:

- *Wired* interfaces: Serial port
- *Wireless* interfaces: Bluetooth, Wireless LAN, cellular (LTE, UMTS, GSM/GPRS)[2].

Clearly every interface itself has its pros and cons but more often than not the actual system that it should connect to has the biggest impact on what is considered a downside and what an advantage. Before continuing to discuss about what interfaces may be suited for what kind of application, one has to first look at manufacturer specific restrictions.

6.2.1 Choosing target platform(s)

One very important question when building custom hardware and interfacing with consumer electronics is to think about what device or set of devices the hardware should be able to communicate with. Clearly, restricting the communication to a single device of a specific brand is much simpler than trying to communicate with many devices at once. This is not necessarily a design decision, but it could be that it does not make any sense to communicate with more than one device, e.g., when building a robot that can be remotely controlled by a smartphone or tablet. Regarding the brand of current consumer electronics, the two biggest platforms currently on the market are:

- *Apple*, with the iPod, iPad and iPhone
- *Android*, used by many different manufacturers such as Samsung or HTC.

Choosing just one specific kind of device or one platform can make life much easier: Most platforms offer extremely powerful and convenient frameworks that allow interaction with the device. Sometimes it is no longer necessary to implement communication protocols or the like and to just concentrate on the application itself, as the rest is provided by the framework. Communicating with multiple devices of different brands can basically be accomplished by

- either separately implementing the communication for the corresponding platforms (in hardware and/or software)

[2]Long Term Evolution (LTE), Universal Mobile Telecommunications System (UMTS), Global System for Mobile Communications (GSM), General Packet Radio Service (GPRS).

- or relying on standardized implementations

The first principle is not really elegant: It involves duplicated components, be it in hardware and/or software. Furthermore, from an empiric point of view, this implies more bugs and thus more effort in maintenance. The second point, especially the required standard implementations of both, hardware *and* software, are clearly preferable but are not necessarily possible: The reason for this are developer programs such as Apple's *Made For iPod* (MFi).

6.2.2 Apple's *Made For iPod (MFi)*

When only developing *software* applications for iPods, iPhones or iPads, there are no restrictions in terms of communications whatsoever by the normal Apple developer program[3]. The provided tools are quite powerful and there exist libraries that make communication between iOS devices easy. When trying to build a custom hardware that can talk to an iOS device, however, things become more complicated. It already starts at the selection of interfaces that are available for communication with such devices: While technically every iOS device has a serial port (cable), Bluetooth and Wi-Fi, for some interfaces it is simply *not possible* to access them without restrictions as hardware interfaces. In short, without approval by Apple itself, one cannot communicate by using neither the serial port nor Bluetooth[4] without *jail-breaking*[5] the product. The magic key to those interfaces is the MFi (Made for iPod) licensing program:

> "MFi Program*: Join the MFi licensing program and get the hardware components, tools, documentation, technical support, and certification logos needed to create AirPlay audio accessories and electronic accessories that connect to iPod, iPhone, and iPad."*
> Source: `https://developer.apple.com/programs/mfi/` *(2012-10-02)*.

Notice that Apple's viewpoint here is that everything that wants to connect to an iOS device is an *accessory*. The way we'd like to see it here, i.e., when considering *consumer devices as user interfaces for embedded systems*, is exactly the other way around: *the consumer electronics is the accessory*. Unfortunately this does not matter at all. The big problem here is that it is extremely hard to figure out what is exactly included in this MFi bundle, i.e., as long as one does not participate to this program, the actual gain remains under disclosure. This is due to the fact that great part of those informations make actually part of the *Non Disclosure Agreement (NDA)* that one has to agree to when participating to the MFi program.

[3] The full development tool-chain is actually free, one only needs to register as a developer; downloading the application to a device or selling it through the Apple App Store, however, is only possible through a subscription.

[4] The so-called *GameKit* allows peer-to-peer Bluetooth, but this does only work between two iOS devices and thus not for communication with a custom product.

[5] The process of *jail-breaking* removes the limitations imposed by Apple on iOS devices through hardware and/or software exploits, but invalidates the warranty and updating capabilities.

The natural question that arises here for every developer of a custom hardware is: Why not participate? The first thing to notice is, that Apple's application website does not make any implications about the *costs* of participating to the MFi program. A quick search through the internet leads to the well known portal of *Stackoverflow*[6]. There, a group of developers was *"looking for experiences on the Apple MFi program registration process"*[7], i.e., they were developing a small electronic device that they would like to connect to an Apple device. The discussion gives the following insights into the MFi program:

- There are no up-front costs to actually join the program, *but*
- you'll need a nearly finished, profound product, with a known target market and an approximate pricing information to be able to participate.
- This product has to pass a series of tests executed by a *third party* to make sure that the product won't interfere with the iOS hardware. Such tests can roughly cost between $\$20,000$ and $\$80,000$
- You need to be a company.
- Every kind of docking interface must be manufactured by *Avnet*[8].

The last question asked in this thread is the very same that every developer of a custom electronic device will eventually come up with: How does the interface really look like? The answer to this very important question is, however, under NDA, i.e., the exact specifications of the interfaces are only available through the MFi program.

To wrap it up: Connecting custom hardware via Bluetooth or a serial port to an iPhone, iPod or iPad is only possible when participating to the MFi program. The reason for those obstacles seems to root in the very philosophy of Apple products, as the following quote from a WWDC (World Wide Developer Conference) keynote emphasizes:

> *"We're only so good as the product we are interacting with or the customers are interacting with at a particular time, so we need to work together."* Brian Tucker; Senior Software Engineering Manager, Mobile Bluetooth Technologies, WWDC 2010 - Session 201 *Developing Applications that work with iPhone Accessories*[9]

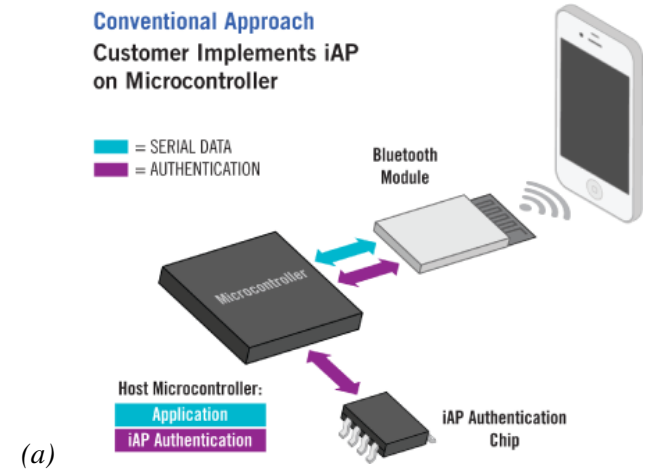
[6] A *"language-independent collaboratively edited question and answer site for programmers"* `http://www.stackoverflow.com`

[7] `http://stackoverflow.com/questions/5734071/looking-for-experiences-on-the-apple-mfi-program-registration-process`; last accessed: September 27, 2012

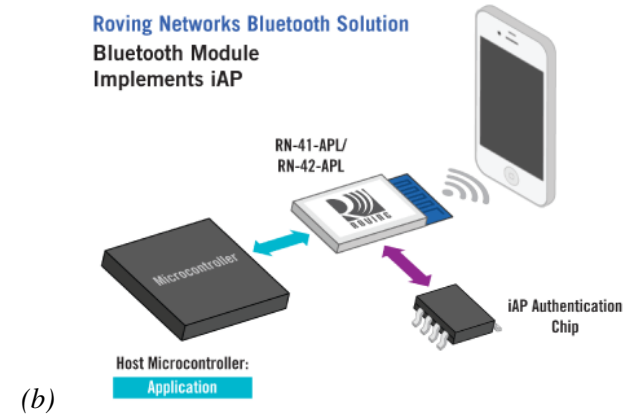
[8] `https://mfi.avnet.com/MFI/`; last accessed: September 27, 2012

[9] Apple WWDC 2010 - Session Videos (registration required): `https://developer.apple.com/videos/wwdc/2010/`; last accessed: September 27, 2012

Thus what Apple does, just as in their App Store, is to pose itself as some kind of quality control. A Bluetooth connection, for instance, requires a special iPod Accessory Protocol (iAP) authentication chip, as depicted in Figure 6.1-*(a)*. Such a chip allows, e.g., to force the use of public-key cryptography to provide authentication and to integrate this whole security procedure into the development framework such that it can be easily used by any developer. This is especially interesting when considering that not all iOS developers are studied programmers or engineers. Thus, security can still be provided while allowing for easy development.



(a)



(b)

Figure 6.1: Bluetooth support for Apple iOS devices [76]

As apparent from Figure 6.1-*(b)*, there already exist simplified solutions for custom hardware to connect to iOS devices, such as the Bluetooth module of *Roving Networks* [76]. The use of such a module, however, does still require MFi participation. Clearly, posing such requirements to developers of custom hardware does not make it easy, e.g., for startups to connect to iOS devices. Thus, at the end of the day, Apple's definition of the MFi program to address

accessories somehow still makes the difference: The custom hardware is considered as an *accessory* to the iOS product, and not as a stand-alone product that just wants to connect to the iOS device.

> *"'Made for iPod/iPhone/iPad' means that an electronic* accessory *has been designed to connect to iPhone, iPad, and iPod models and has been certified by the developer to meet Apple performance standards."*
> Source: `http://support.apple.com/kb/HT1665` *(2012-10-02)*.

6.2.3 The multi-platform challenge

The previous section has made clear that developing for multiple platforms includes big decisions like whether or not to take hurdles such as the MFi program. The actual decision depends on the product, the available time and clearly on the available economic resources. For Bluetooth and serial communication it does not look like Apple is going to accommodate to developers for multiple platforms, as it seems to be an integral part of the company's philosophy. Thus at the moment there are only two options if participating to the MFi is out of question: To drop certain interfaces or to drop certain platforms. The major conclusion about developing for multiple target platforms is the following:

Observation 13. *The real challenge when interfacing consumer electronics is the quest for interfaces that can be used by the majority of the available devices without the needs of specific implementations.*

6.3 Interface discussion

When considering to build a system that should connect to consumer electronics, it is important to analyze *all* the available interfaces, regardless of "restrictions" such as the MFi program. The question whether or not to take those hurdles should be ultimately answered afterwords, i.e., the discussion of the interfaces should happen regardless of the brand and restrictions, those obstacles should only be kept in mind for a final decision.

6.3.1 Wired Communication

The serial port is the only wired interface that is currently available on the majority of smartphones and tablets[10]. In contrast to wireless communication, the *channel* of wired communication is actually physical and thus inherently offers (better) protection against attacks such as eavesdropping. Thus, from a security point of view an actual physical medium is preferable

[10]Basically, the audio jack can be abused as serial communication interface too, but this option is excluded from this discussion.

over wireless communication, as in wireless communication physical access to the device is often no longer necessary. Physical contact, however, does not completely eliminate the need for security: *Authenticity*, for instance, is in general *not* implied.

In terms of data rates, wired communication can be very fast, depending on the actual communication protocol used. Current smartphones basically allow data rates comparable to USB 2.0, which has an effective throughput of up to 35 MB/s[11]. Furthermore, a physical connector can at the same time power the consumer device. On the other side, when talking about interfacing *consumer devices*, wired communication implies communication via the serial port and thus practically restricts the number of clients to a single device. Furthermore, there is a clear trend towards wireless communication to prevent cable clutter.

6.3.2 Wireless Communication

In contrast to wired communication, the major weak spot of wireless networks is the fact that the attacker has an easy time accessing the transport medium. Wireless communication systems are evolving fast and thus it is hard to keep track of vulnerabilities and to give a solid response to the question of *how secure* wireless communication is. Furthermore, there are still many *legacy* devices in use, especially for Bluetooth, which operate using outdated standards. [1] gives a short insight into the key issues of wireless communication, but it is not the scope of this thesis to provide a detailed analysis of the security aspects of communication systems. An interested reader may refer to [5], which provides a more detailed analysis of current network security.

Basically, there are only *two* different *kinds* of wireless communication systems to consider when talking about consumer electronic devices:

- Cellular networks, and
- Radio networks in the 2.4 GHz *Industrial, Scientific and Medical (ISM)* radio band, which for consumer devices are Wi-Fi and Bluetooth

Cellular (LTE, UMTS, GSM/GPRS)

Using cellular networks requires subscribing to a data plan offered by some telecommunications company. The difference in the data rates between the different standards is huge and ranges from 13 kbit/s for GSM, to at most 42 Mbit/s for UMTS, up to 299.6 Mbits/s (peak download rate) for LTE. The security concepts are also quite different, but base on a Subscriber Identity Module (SIM) which stores an international mobile subscriber identity (IMSI) and a unique secret key, which serve for authentication procedures that are executed on the rather complex network infrastructure. [5, Chapter 19, Page 311] gives a more detailed description of the security concept and issues of GPRS and UMTS telecom networks.

[11] The actual data rate could not be determined, as it was not possible to get any details about e.g., the serial port used by Apple devices. Basically, FDTI chips [61] allow serial communication for custom devices and implement the USB 2.0 standard. Those chips operate at speeds up to 12 Mbits/s.

In general, the security measures provided by cellular networks can be considered moderate or secure. Those networks allow to communicate over extremely high distances and can even include internet access. Clearly, the high range at the same time increases the attack radius. In terms of consumer electronics as user interface for an embedded system, cellular networks may not be the optimal choice: Exchanging data over such networks is costly, which is why they are not very well suited for constant communication. As an additional interface, however, a cellular network can provide huge possibilities.

Bluetooth

Bluetooth is a wireless communication standard that is intended for data exchange over short distances. As already mentioned, it uses the same frequency bands as Wi-Fi. The standard is basically managed by the *Bluetooth Special Interest Group (SIG)* [52] and is repeatedly ratified by the *IEEE Standards Association* [63] as the *IEEE 802.15* standard. The standard has evolved from a rather insecure communication system to a secure communication standard for so called *wireless private area networks (WPANs)*. This category basically includes portable equipment:

> *"Bluetooth technology is a wireless communications system intended to replace the cables connecting many different types of devices, from mobile phones and headsets to heart monitors and medical equipment."*
> Source: `www.bluetooth.com` *(2012-10-02)*

The fact that Wi-Fi and Bluetooth operate on the same frequency bands requires measures to guarantee coexistence. For this Bluetooth uses *frequency hopping*, i.e., it continuously changes the actual band on which it transmits the data. In order to be able to do so, Bluetooth bases on a master-slave structure, where every slave shares the master's clock to hop between the different frequencies. Regarding multiple clients, this master-slave structure results in the following dilemma: Some consumer devices, such as Apple's iPhone, use the same chip for Wi-Fi and Bluetooth communication. In order to guarantee maximum coexistence, it would be preferable if the consumer device would be the master, such that it can specify the hops depending on its Wi-Fi connection. On the other side, allowing multiple clients requires the embedded system to be the Bluetooth master, and there can only be one master.

What is especially interesting for embedded systems is the *pairing/bonding* procedure that comes with the Bluetooth standard: In order to be able to establish a secure connection *without any user interaction required*, the communicating systems must initially be *bonded*. The process of creating this bond is called *pairing*, which for instance can happen by typing in a PIN on both devices. Notice that this is exactly what has been mentioend in Section 2.3, i.e., that an authenticated key and thus a secure connection cannot be established without a shared secret. The current state of the art pairing procedure is *Secure Simple Pairing (SSP)*, which can happen through different processes. The actual procedure used depends on the interface elements provided by the systems. Once the systems have been bonded, subsequent connections do

not require any user interaction at all. This pairing procecure, however, is at the same time the weak-spot of the standard. Most successful attacks are executed during bonding/pairing. The fact why this *pairing mechanism* is so interesting for embedded systems is that it completely eliminates the need for a custom security approach: The Bluetooth SIG already delves deeply into the problem of establishing a secure connection with minimum user interaction.

The actual throughput depends on the version of the used standard. *Wikipedia* provides a good overview over the current standards[12]. Basically, up to Bluetooth v2.1 the maximal data rate is 3 Mbit/s. With Bluetooth v3.0+HS (High-Speed) data rates up to 24 Mbit/s are possible, though not over the Bluetooth link itself but rather on a 802.11 (Wi-Fi) link that has been established using Bluetooth (and is transparent to the user). Bluetooth v4.0 adds a low-energy protocol (BLE, Bluetooth Low Energy) to the v3.0+HS standard, which operates at 1 Mbit/s and is basically only suited for transferring short bursts of data. All in all, Bluetooth is well suited for low cost and low power applications, provides a satisfactory throughput and good security (Advanced Encryption Standard, AES-128bit). One major downside of Bluetooth, however, is that it does very often not provide any legacy compatibility, such as it is the case now for low-power applications (they require v4.0).

Wi-Fi

The term *"Wi-Fi"* generally refers to what is known as *wireless LAN* or one of the *IEEE 802.11* standards of the *IEEE Standards Association*. *"Wi-Fi"* was actually introduced by the so called *Wi-Fi Alliance* [85], which is an organization that certifies products on the basis of the *IEEE 802.11* standards. Notice that *"Wi-Fi"* does not stand for *"Wireless Fidelity"*. Basically, wireless LAN was intended to replace the wired LAN networks in work areas. In terms of security, the Wired Equivalent Protocol (WEP) should have guaranteed the same integrity and confidentiality of data as an equivalent wired communication. In the meantime, WEP has been replaced by the far stronger security protocol of Wi-Fi protected Access (WPA) 2, which bases on the AES as encryption standard. WPA2 security even meets the *U.S. government computer security standards*[13]. In terms of throughput, Wi-Fi can be as fast as 250 Mbps [85].

In comparison with Bluetooth, setting up a wireless network using Wi-Fi typically involves more steps than there are needed to connect systems using Bluetooth. To address this issue, the Wi-Fi Alliance has introduced the *Wi-Fi Protected Setup (WPS)*[14] standard, which should have allowed for an easy establishment of secure wireless networks. The implementation, however, has several serious vulnerabilities and can even be broken by a *brute-force* attack [79]. Thus, an easy setup method has yet to be found for Wi-Fi networks.

[12] `http://en.wikipedia.org/wiki/Bluetooth`; last accessed: October 4, 2012.

[13] `http://www.wi-fi.org/media/press-releases/wi-fi-alliance-introduces-next-generation-wi-fi-security`; last accessed: September 27, 2012.

[14] `http://www.wi-fi.org/knowledge-center/articles/wi-fi-protected-setup`; last accessed: September 27, 2012.

6.3.3 Summary

Wired Communication via the Serial Port

- Bandwith: up to 35 MB/s or 12 Mbits/s for FDTI chips [61]
- Security: custom
- Range: short (serial)
- Pros: physical transport medium, possible power source
- Cons: different connectors for different brands, wiring, single-client

Wireless Communication via Cellular

- Bandwith: up to 299 Mbit/s
- Security: moderate
- Range: highest
- Pros: extremely high range, internet connection, throughput
- Cons: subscription needed, costs, many different standards (compatibility)

Wireless Communication via Bluetooth

- Bandwith: up to 3 Mbit/s or 24 Mbit/s resp.
- Security: sufficient
- Range: short
- Pros: Secure Simple Pairing (easy setup), low power, low costs
- Cons: Pairing as security issue, legacy compatibility, master-slave architecture

Wireless Communication via Wi-Fi

- Bandwith: up to 250 Mbit/s
- Security: high
- Range: medium
- Pros: security standards, throughput, simple implementation
- Cons: setup procedure (WPS issues)

*no wireless communication system can guarantee *Availability*.

6.4 Concluding remarks

The possibilities of a serial port and different wireless interfaces provided by consumer electronics offer a variety of ways to connect to them. In general, serial- and Bluetooth communication can be cumbersome because of developer programs such as Apple's MFi. Such programs make prototyping a hard task if no membership is available and are thus hindering any simple investigation on possible communication schemes.

Wired communication can be very interesting for applications that require continuous interaction with the user, e.g., for navigation. This is not only a security question, but also helps to power the consumer device during operation. Cellular connections should only be considered if the communication happens rather sporadic and proximity to the device is generally not possible. Furthermore, an individual cellular connection for an embedded system should only be introduced when necessary, e.g., if access to a router can not be assumed. If such a connection would be continuously available, such as in home automation systems, an engineer should consider to connect to such a router in order to access the internet. Bluetooth and Wi-Fi have comparable capabilities. The Bluetooth standard comes with simple and yet basically secure pairing procedures, that can come in very handy for embedded applications, as in general such system have only limited hardware interfaces. Wireless LAN on the other side, has the best security protocols, is rather simple to implement and allows for higher data rates. For this inquiry, where the serial port and the Bluetooth connection are out of question because of the MFi program, Wi-Fi is the best option.

What has not yet been discussed is the possibility to not only use the consumer device as user interface, but also as a gateway to the internet. This is something that still needs some investigation, but opens a whole lot of possibilities. In respect to cloud computing, direct internet access would not necessarily be required, as every time a consumer device connects to the system, the systenm can use it to execute services that require internet access. Notice that internet access is not necessarily required while the consumer device is connected to the system: The system can just provide the information to the consumer device, such that the next time when there is an internet connection available, the device can execute all the necessary services or put the data on the web.

This lack of further investigation into consumer devices and generally the integration of embedded sytems into the internet is currently of general concern and there's yet much research to do. The following case study serves as a first step to investigate how this connection with consumer devices can work out.

CHAPTER 7

Case Study: A Gateway between the Diagnostic Port and Consumer Electronics

As part of this inquiry into Gateways an actual case study has been executed. The key priorities are tightly coupled to the concepts, observations and insights given in the previous sections: Gateways are a hot topic in the automotive sector, which warrants the choice of the automobile as one party communicating through the gateway. On the other side, consumer electronics and casual- or ordinary end-users pose a special and very interesting challenge to the developer of such a platform and at the same time guarantee that the result is something tangible, comprehensive and nicely presentable. The development of this case-study reflects the essential process of a product development cycle:

- A requirements analysis for the gateway,
- the design of the communication and operation architecture,
- the choice and/or design of the hardware platform,
- the implementation of the firmware and
- the application software for the target application.

Clearly, the result of this case-study is still far from production ready, but nonetheless every aspect of the development cycle of an actual product can be and was considered during the design such that improvements and extensions to the implementation are feasible without major efforts.

7.1 System Model

Figure 7.1 depicts the general system model. It involves three principals: The consumer (C), the manufacturer (M) of the vehicle and the developer (D) of the gateway hardware. The gateway hardware is not considered as a part of the vehicle's electronic system but is a third party component to the vehicle just like common OBD scan tools.

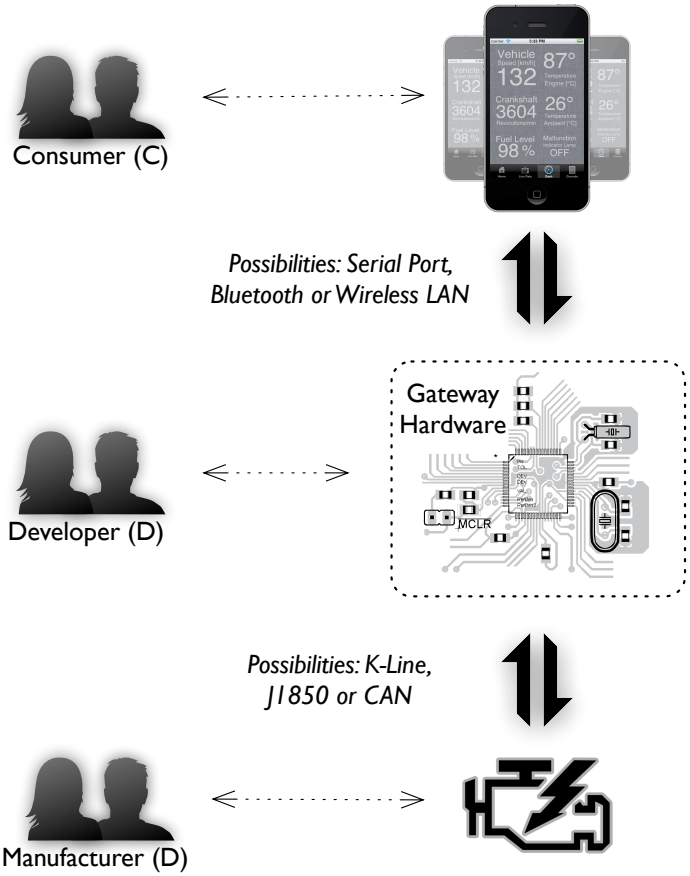


Figure 7.1: Generic system model for gateway and application.

The major component of the case study is the gateway itself, i.e., the development of a hardware platform together with the firmware. The consumer electronics is assumed to be a closed component with reference to the hardware, the software application shall be considered exemplary to emphasize the transition from computer engineering to software engineering. The communication happens on two bidirectional (and thus *composite*) interfaces:

- The diagnostic port forms the interface to the vehicle (cf. Figure 5.3 on Page 40) and provides three possible bus systems for communication (K-Line, J1850 or CAN) as well

as a possible power supply.

- In Section 6 the serial port, Bluetooth and wireless LAN have been identified as the most common interfaces to consumer electronics.

Regarding the involved parties there are virtually no relations to be considered: The consumer, developer and manufacturer operate independent of each other and the interests are only of economic nature. From the manufacturers' point of view there is no difference between a regular scan tool and the custom gateway hardware: If the vehicle complies to the OBD standards, the diagnostic port and services have to fulfill the ISO standards and thus the gateway hardware can communicate with the vehicle. The (regular) consumer is typically the car owner and does therefore have physical access to both, the vehicle and the gateway hardware. The developer is only responsible for the final product.

7.2 Gateway Requirements Analysis

As this happens to be a case study, the initial requirements to the gateway are minimal.

Functionality.

`Req.1:` The final result shall implement a chosen subset ($\neq \emptyset$) of the OBD services.

`Req.2:` The consumer shall be able to execute the specified services using a consumer electronics device.

`Req.3:` The execution of the services shall comply to the on-board diagnostic standards.

Security.

`Req.4:` The gateway hardware must by no means facilitate an attack on the vehicle via the diagnostic port, i.e., it must not amplify the attack surface of the vehicle.

`Req.5:` The data that is exchanged through the gateway shall be protected against eavesdropping using appropriate security measures.

`Req.6:` The gateway shall respect the security requirements that are specified by the on-board diagnostic standards (cf. requirement 3).

Connectivity.

`Req.7:` The chosen interface for communication between the hardware and the consumer electronic shall be supported by the majority of commonly used devices.

`Req.8:` The chosen communication scheme shall be portable to other platforms with reasonable effort; wireless connections shall be preferred.

Safety and Usability.

`Req.9:` The security measures shall have minimal to no impact on the usability of the product, i.e., they shall not require special knowledge.

`Req.10:` The installation and/or removal of the gateway hardware should require minimal effort.

`Req.11:` The removal of the hardware shall be optional and not necessary. This requirement comes from the fact that the diagnostic port is not necessarily easily accessible and thus being required to remove the hardware is tedious.

`Req.12:` The usage of the gateway hardware must not pose any further safety risks, i.e., the gateway operations must not have any impact on safety aspects. Notice that this requirement depends upon the implemented service: the gateway hardware must be able to determine if executing a service in the current vehicle state is appropriate. E.g. a request to disable the headlights while driving under insufficient lighting conditions must not be executed[1].

`Req.13:` The gateway hardware should require minimal maintenance (e.g. battery exchange).

`Req.14:` The usage of the hardware itself should require minimal user interaction, especially regarding physical interaction with the hardware.

`Req.15:` Security measures should by no means lead to a denial of service (e.g. failed authentification attempts).

Constraints.

`Req.16:` A final hardware product should be affordable, i.e., the costs for individual components should be held as low as possible.

`Req.17:` A possible final hardware product shall have minimal space requirements (cf. requirement 10: easy installation) such that the space provided by the diagnostic port is optimally usable.

`Req.18:` The choice of the development tools, hardware components and target consumer electronics have to allow high implementation speed in order to support a reasonable deadline for this master's thesis.

[1]Notice that this functionality is just an illustrating example. Such a function is not defined in the on-board diagnostic standards bundle, but could be executed through the diagnostic port (e.g. through manufacturer specific services. The functional requirements do only specify that at least one OBD service shall be implemented, but does not restrict any further functionality.

7.3 Solution Overview

The key-considerations of the chosen approach are depicted in Figure 7.2. The chosen target platform is Apple's iOS (mobile operating system) or OS X respectively. Both operating systems are in line for the target platform as there are no fundamental differences between developing an iOS application or a fully featured OS X applications. This allows to add (Apple) computers and notebooks to the list of currently supported target platforms.

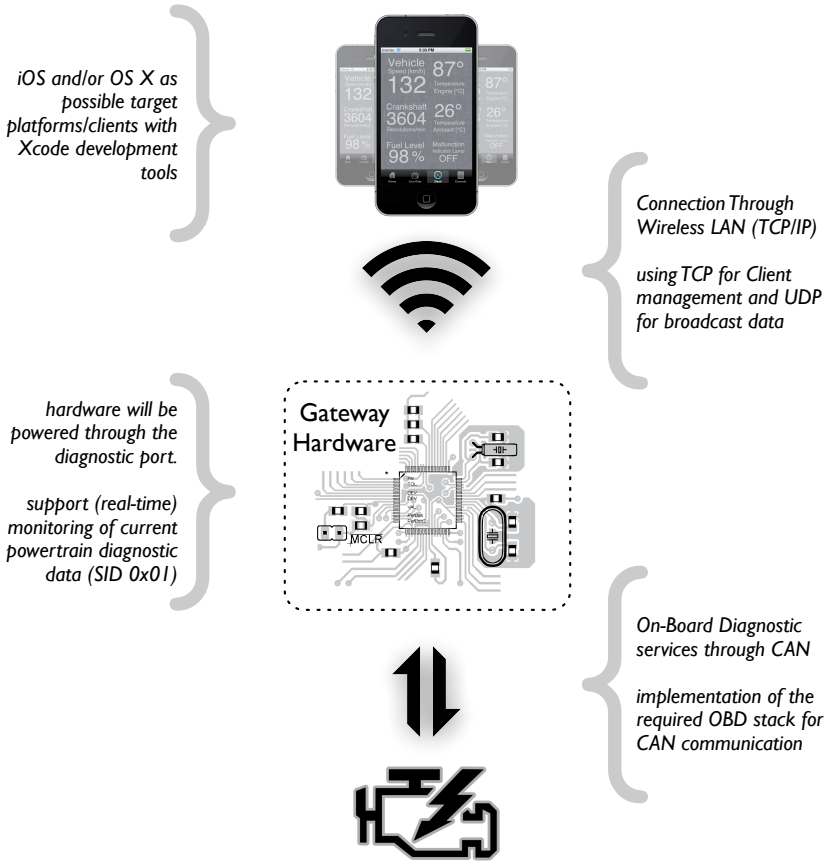


Figure 7.2: Overview of the chosen approach for implementation.

The main motivation for iOS/OS X as target platform is the development framework, which significally limits the implementation overhead once the developer is familiar with it, and the community. The learning curve is sufficiently high, but the tremendous developer community offers countless examples, tutorials and instant feedback which limit the time exposure to a reasonable amount. Regardless of which difficulty or problem that may arise throughout the

development of the application, there is a very high probability that it has already been solved by someone else. There are countless entries with reference to iOS development, e.g., on `www.stackoverflow.com`.

For communication between the gateway hardware and the target platform(s) TCP/IP over wireless LAN has been chosen. Wireless communication is obviously the preferable choice as it eliminates the need for wiring and thus greatly enhances the usability of the product. A wired communication between the hardware and consumer electronics would only be acceptable if it could be seamlessly integrated into the dashboard, but as the hardware is considered a third party product, this approach is not yet viable. Furthermore, the MFi program (cf. Section 6.2.2) complicates the usage of physical interfaces and Bluetooth. But not only physical constraints have lead to the choice of wireless LAN: As elaborated in Section 6 this approach is highly portable: It works on most mobile and computer platforms and is thus practically platform independent. The fact that TCP/IP is the same for every platform strongly simplifies the porting for consumer electronics applications. This fulfills the Requirements `Req.7` and `Req.8` and allowed to postpone the decision about which target consumer electronics to use to a later point in time when more details about the implementation were available.

As the diagnostic port (cf. Figure 5.3 on Page 40) features a power supply it seems only natural to use it. CAN has been chosen as bus system for on-board diagnostic because *"only one protocol is allowed to be used in any one vehicle to access all legislated emission-related functions"* [ISO 15031-4] and future vehicles are required to support CAN for diagnostics. Notice that the restriction to a single bus system means that the hardware will not be ISO compliant, as *"a fully compliant external test equipment shall support all communication protocols [as specified in ISO 15031-4]."* [ISO 15031-4]. Nevertheless, supporting more protocols only increases the implementation time while not contributing to the case study (c.f Requirement `Req.18`).

Throughout the rest of the document the following terminology will be used: Applications connecting via wireless LAN will be referred to as *"Clients"* or *"target application(s)"* and resp. *"target platform(s)"* when talking about the actual hardware. The gateway itself will be referred to as *"gateway hardware"*, *"hardware platform"* or simply *"Gateway"*.

7.3.1 Implemented Functionality

Before jumping to the actual application it may be convenient to analyze what applications could be built on top of the on-board diagnostic services.

Choosing an appropriate function

The on-board diagnostic standards offer services for essentially three possible functionalities as identified in Section 5.3.2 (page 41):

- The readout and erasure of an error memory which saves diagnostic trouble codes (DTCs),

- tests of emission-related components, and
- the request of information about the vehicle and from control units.

Without any detailed knowledge about emission-related components, the execution and evaluation of emission-related tests does not provide any useful information to a normal user, which is why those services were out of question. Services on the error memory, however, would appear quite interesting: With only minor knowledge about vehicles a user could check for saved trouble codes before deciding to bring the vehicle to a repair shop. This would be especially interesting because the Malfunction Indicator Lamp (MIL) is also on for "minor" failures which would not require immediate repair. The implementation of such a functionality, however, is very tedious as the mapping of DTCs to their description involves about 100 pages of tables (the definitions can be found in ISO 15031-6). Therefore, the implementation of this feature was out of question too. One thing to notice regarding DTCs and the MIL is that with the WWH-OBD standard (Section 5.4 on page 44) a *"failure severity indication via the dashboard malfunction warning signal"* [82] is planned which would essentially be the functionality described before. Ruling out diagnostic trouble codes leaves the following OBD services for implementation:

`Service 0x01`: Request Current Powertrain Diagnostic Data,

`Service 0x09`: Request Vehicle Information.

The actual application

Not every part of the services `0x01` and `0x09` is easily comprehensible by a normal user, which is why only a subset of the information that can be obtained through these services is used. The only information used of service `0x09` is the *Vehicle Identification Number (VIN)*. By using this information, a target application (e.g. running on a smartphone) could take different actions if it is connected to an unknown vehicle (and e.g. build a database for each known car).

The main idea was to use service `0x01` to provide a real-time database of current vehicle data: A target application can basically choose between a request to the gateway to

- *add* a monitor for a specific parameter (e.g. the vehicle speed sensor), or to
- *remove* a (previously added) monitor for a specific parameter.

Not all the Parameter IDs (PIDs) specified by ISO 15031-5 are supported by the gateway application. The intention of this restriction is to reflect a list of allowed parameters, i.e., a list defined by the manufacturer who specifies which data can be retrieved from the vehicle. Table 7.1 lists a number of commonly comprehensible PIDs, which have been chosen to be supported by the gateway. Those PIDs are only exemplary, i.e., adding further PIDs to the implementation is not an issue and does not require much effort. Notice that the vehicle does not necessarily support all of those PIDs, which is why the gateway application synchronizes its PID-list with the actually

supported PIDs each time it is connected to a vehicle. If the gateway application supports the requested monitor, it saves the request. The target application is then informed whether or not the monitor could be added. Based on this list of requested monitors, the gateway application builds and executes requests for the vehicle to retrieve the data according to the ISO 15031-5 standard. The (positive) responses from the vehicle are then forwarded to the target application. If the target application no longer needs the data, it issues an un-register request for the monitor which causes the monitor to be removed from the gateway's list.

(a) Live-Data *(b) Virtual Dashboard*

Figure 7.3: Screenshots of the applications implemented on top of Diagnostic Service `0x01`.

Figure 7.3 shows two screenshots of how those monitors are currently used by the target application: The App includes two tabs, one to provide live-data (Figure 7.3-a) and one virtual dashboard (Figure 7.3-b).

Live-Data: The live-data tab is essentially a 1:1 implementation of the monitoring functionality described above: Basing upon a list of pre-defines parameters the user can choose to add or remove a monitor for a specified parameter. Those requests are forwarded to the gateway application which retrieves the data from the vehicle and sends it back to the target application. The values for the parameters are updated with the received data until the monitor is removed by the app (e.g. by switching tab).

Virtual Dashboard: The dashboard uses the same exact functionality as the Live-Data but with a fixed list of parameters, namely the vehicle speed, the engine RPM, fuel level, ambient and engine temperature and the status of the malfunction indicator lamp. It is used to

demonstrate that the real-time database provided by the gateway can be used in other ways too.

PID Number	Description
0x01	Status Of The Malfunction Indicator Lamp (MIL)
0x04	Calculated Load Value
0x05	Engine Coolant Temperature
0x0B	Intake Manifold Absolute Pressure
0x0C	Engine RPM
0x0F	Intake Air Temperature
0x11	Absolute Throttle Position
0x1F	Time Since Engine Start
0x21	Distance Travelled While MIL Is Activated
0x2F	Fuel Level Input
0x31	Distance Since DTCs Cleared
0x33	Barometric Pressure
0x46	Ambient Air Temperature
0x4D	Time Run By The Engine While MIL Is Activated
0x4E	Time Since DTCs Cleared
0x5A	Relative Accelerator Pedal Position

Table 7.1: List of coherent Parameter Identifiers (PIDs) used by the implementation.

7.3.2 Communication model

As the choice of the interface(s) that should be used for communication with the consumer electronics forms a part of the requirements analysis (cf. `Req.7`, `Req.8` of Section 7.2), during the initial design phase it was still uncertain whether or not multiple clients should be supported. The choice of wireless LAN and TCP/IP opened up the possibility to allow multiple clients to connect to the gateway which resulted in the obvious decision to exploit this capability.

Supporting multiple clients typically implies a higher communication overhead, especially in conjunction with providing real-time data: If clients request different monitors, the amount of data that has to be communicated increases linearly with the number of clients, e.g., with two clients, there is twice as much data to be transfered. There is, however, a possibility to limit the overhead if clients chose the same monitors. As apparent from Figure 7.2 the protocols TCP and UDP have been chosen for communication. They are used in the following way:

TCP is used for client management, which in this case means that control information, such as requests for adding and removing monitors as well as the responses from the gateway are sent via TCP. The main hurrier for this choice is the fact that TCP ensures a reliable and ordered transmission of messages and that it is connection oriented.

UDP is a connectionless and unreliable protocol. Its simple nature implies a low communication overhead and is thus well suited for the transmission of real-time data where dropping packets is preferable over waiting for re-transmission. This is a perfect fit for the data provided by the gateway: The real-time data is not critical, i.e., dropping one or the other value is not of importance as it is periodically updated.

In order to increase the throughput for multiple clients (monitoring the same parameters) the real-time data is currently being broadcasted, i.e., everyone listening to the correct port can obtain the data. Another option would be multi-casting, but this would require some management overhead. This way, every client just has to register the appropriate monitors and pick the appropriate data out of the UDP broadcast. The UDP communication is unidirectional from the gateway to the client(s), i.e., the gateway does not respond to any message sent through UDP, the UDP channel is dedicated to forwarding the OBD data to the clients (through broadcasting).

It is mentionable that using broadcasts simplifies the actual OBD application running on the gateway (cf. Section 7.6.1), as the application does not need to keep track of the parameters that a client has registered for monitoring.

The communication of the gateway with the vehicle is fixed through the OBD standards: They define message timeouts and minimal waiting times between two consecutive requests for an external test equipment. Those requirements are to be fulfilled by the gateway. Section 7.6.3 will go more into detail about the OBD communication.

7.4 When Ideas face Constraints: Implementing the Idea

Before moving on to the implementation details it is convenient to pause and reflect on the decisions that lead to the actual solution. The design and development of a new system most of the time isn't a clean step by step process but the result of what parts of an idea can actually be implemented under certain circumstances. Requirement `Req.18` incorporates the circumstances under which this project has come to live: the design process was particularly influenced by the limited time available for a master's thesis together with the fact that the background information had yet to be acquired throughout the thesis and thus throughout the design process.

The main idea for this project was to built an automotive gateway for consumer electronics as a case study for the preceding inquiry into gateways, i.e., to identify the results of the inquiry in an actual implementation. The concept of a gateway is tightly coupled with security considerations. In the automotive sector, security is currently a hot topic, which supports the choice

of this field of application. The actual functionality, which for this gateway are on-board diagnostic services through the diagnostic port (cf. Requirement `Req.1`), was only of secondary importance and should not severely influence the actual implementation, i.e., the fact that the functionality is only limited (in this case to real-time data) must not have major impacts on e.g. security considerations.

Thus the impelling idea was a prospect on an actual product that incorporates all the research done about interfacing consumer electronics and gateways. The development of such a product would involve security and usability experiments as well as a final prototype. At the same time the limited possibilities of a master's thesis had to be considered, i.e., it was not a priori clear to what extent the result of the practical work would actually differ from the Idea, as the main focus was to get the system up and running. Nevertheless, having this ambitious goal of a final product in mind is very important for design decisions, such that the actual outcome is not far off the product, in the sense that only little *engineering work* is needed to get from the actual result to the product. In terms of *implementation work*, however, there might yet be many hours of (mostly unchallenging) work to do.

7.4.1 Towards a Product to keep in Mind

The diagnostic port is an interface that is not intended to be in the field of view of the vehicle's passengers and thus forms a special challenge in terms of physical user interfaces, such as buttons, LEDs or displays, as well as communication interfaces, i.e., how the consumer electronics connect to the device. The preferable communication interfaces have already been elaborated in Section 7.3: They are wireless LAN for the communication with the consumer electronics and CAN for the communication with the vehicle.

Hardware user interfaces

As already mentioned in Section 5.3.1 the locations for the diagnostic port can vary but should be generally easy to access, i.e., not require any specific tools or the removal of a panel cover. To be specific, ISO 15031-3 gives the following definition for the vehicle connector location and access for passenger cars and light duty vehicles:

> *"The connector shall be located in the passenger or driver's compartment in the area bounded by the driver's end of the instrument panel to 300 mm beyond the vehicle centreline, attached to the instrument panel and easy to access from the driver's seat. The preferred location is between the steering column and the vehicle centreline."* [ISO 15031-3]

Thus, accessing the diagnostic port can for the most part be considered as tedious, but it is still within reach from the driver's seat. ISO 15031-3 further specifies that it shall be located *"so as to permit a one-handed/blind insertion of the mating external test equipment connector"*

but it still has to be *"out of the occupant's (front and rear seat) normal line of sight"*. This is especially important when planning to use visual user interfaces, such as LEDs or displays. Basically, there are three possibilities how the final hardware can be used in the vehicle:

Mounting approach: Make the hardware an accessory that can be mounted at some place that allows physical interaction, such as the dashboard. Notice that the hardware still has to be wired to the diagnostic port.

The main reason why this approach is not acceptable is the fact that the diagnostic port may reside under the steering column and thus directly at the feet of the driver.

`Req.10`: urges easy installation and removal. A proper installation of the hardware would require cautious wiring such that the wires do not interfere with the driver.

`Req.12`: Even if properly wired there is a possibility that the driver could get caught up in the cables which interferes with driving. The worst case would be the plug being pulled out causing the hardware to slip under one of the pedals. The sheer possibility of such an event is reason enough for this approach to be declined[2].

Besides the fact that this approach allows visual contact with the hardware, there is another potential advantage that could favor a dislocation of the device: The adapter needed to connect to the diagnostic port could be used to allow for even easier installation and removal of the gateway than the original port. The facts that - strictly speaking - the diagnostic port itself is designed for easy access, and that the removal of the hardware should not even be necessary (cf. Requirement `Req.11`), however, invalidates this advantage.

Secondary interfaces: Another approach would be to provide some form of secondary interface to the hardware, such as an external display (which could be a second piece of hardware communicating wirelessly with the gateway) to allow further interaction.

While this approach may be a valid one, it does not fit well into the concept of a gateway for consumer electronics, as the consumer electronics itself is already an external interface for the gateway: it visualizes the data that is exchanged between the vehicle and the device and allows interaction. Requiring e.g. a secondary display *and* and a device like a smartphone is an ill defined concept.

Plug&Play: Make the hardware *Plug and Play*.

History has not been kind with the *Plug and Play* concept, which bitterly is also often referred to as *Plug and Pray* as actual implementations have regularly failed to master the

[2]Clearly, theoretically every approach could suffer the scenario described in this approach, as the hardware always has to be plugged into the diagnostic port. For this approach, however, the cables offer additional room for failure while for the other approaches the hardware may be properly fixed such that it cannot unplug without interaction while driving, i.e., it may not fall out of the diagnostic port because of vibrations or shocks.

concept. For embedded systems, however, which most of the time operate according to a rigorous specification, the concept of Plug and Play is a perfect fit: Embedded systems do not require much interaction for setup and interaction is mostly associated with demands of the user. The gateway that was implemented as part of this master's thesis, for instance, runs independent of user interaction (cf. Section 7.6.1). The only interaction required by the user is to connect to the wireless network and request monitors for certain parameters[3].

Making the hardware Plug and Play has the tremendous advantage that it essentially eliminates the need for physical hardware interfaces such as displays, which in turn allows for smaller designs. Thus, Plug and Play would be the optimal concept in terms of usability and space requirements (favoring Requirement `Req.17`).

All in all, physical interfaces to the hardware should only be used if absolutely necessary, *Plug and Play* is the optimal concept to be pursued when building embedded systems. This is especially true when interfacing consumer electronics, as they are themselves perceived as an interface by the user. The picture of the *Texa OBD Log* [80], which has already been presented in Section 2.1, has accompanied the project as a paramount example for physical interfaces: It only features a single USB port, which perfectly fits the purpose it has been built for. There's nothing left to do than to plug it into the OBD port for data acquisition and remove it to retrieve the data through a target application. Although the gateway developed as part of this thesis has slightly different requirements, the physical form of the product to keep in mind should be the one depicted in Figure 7.4: A simple plug.

Figure 7.4: The *Texa OBD Log* [80] as inspiring example.

[3]The wireless network which the user has to connect to is known a priori and not communicated through e.g. an LCD display. This knowledge can be part of factory defaults (cf. Section 7.4.3) which can possibly be changed through explicit user interaction, but again, no physical interaction should be required.

The interfaces that are used for the gateway have been identified as wireless LAN for communication between gateway and consumer electronics and CAN for communication between the gateway and the vehicle. This decision bases mainly on the *MFi* program (cf. Section 6.2.2) which complicates the usage of the serial interfaces for communication with consumer electronics. During system design, however, the serial port was not yet completely out of question, i.e., it's usefulness had yet to be evaluated - regardless of the MFi program. One reason why the serial port at this stage of development should not yet be excluded are possible security measures.

7.4.2 Excursion: *The Resurrecting Duckling*

The choice of used interfaces does not only depend on usability questions but also on security measures. The investigation for a proper security policy has lead to a very interesting publication titled *"The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks"* [21]. The paper introduces the *resurrecting duckling* security policy model *"which describes secure transient association of a device with multiple serialized owners"*.The following paragraphs outline the essentials of the metaphor of the resurrecting duckling.

The device under investigation in [21] is an embedded system, underlying constraints such as limited computing power, which discourages the use of asymmetric cryptography [22], and limited available energy, e.g., limited battery power. One key aspect is the fact that the device has *"multiple serialized owners"*, i.e. there is a succession of owners that should have temporary *control* over the embedded system. The main contribution of the paper tackles the authentication problem, i.e., the question to whom a principal can talk to. The security properties of *Confidentiality* and *Integrity* are then analyzed on top of the authorization process[4]. Usual approaches involve authorization mechanisms through a centralized system administrator, an approach that is not feasible in ad-hoc environments. The solution to this problem is as simple as it is well presented by the authors:

> It all bases upon the exchange of a shared secret through a physical channel.

The metaphor used in [21] is, just as the title suggests, a *resurrecting duckling*: Initially, the embedded system - or "duckling", is in a pre-natal state, i.e., it has not yet been associated with any controlling device. This association is established upon birth: *"A duckling emerging from its egg will recognize as its mother the first moving object it sees that makes a sound regardless of what it looks like"*. Similarly, the embedded system should exchange or accept a *shared secret* from the first device it talks to. This key exchange, or *"Imprinting"* [21], can either happen wirelessly or through a physical channel. The key issue for the wireless association procedure is that it still requires physical interaction with the embedded system, such as a button that has to be pressed upon the pairing procedure or an LCD displaying a hash-code, as there is no other way to make sure that the right device has received the key. Thus *"in many applications, there*

[4] *Availability* is discussed separately, as denial-of-service attacks are not necessarily influenced by authentication procedures, e.g, jamming for wireless applications.

will only be one satisfactory solution, and we [21] advocate its use generally as it is effective, cheap and simple: physical contact". On top of this shared secret, protocols for authentication and key establishment can be executed.

Finally, the transient association with the controlling device implies that the duckling may *die* eventually, i.e., the device changes back to its pre-natal state and thus loose state information such as whom it is associated to. There are essentially five possibilities for this to happen:

1. The duckling/device *dies of old age*, which can happen through a timeout.
2. The devices is *designed to die*, i.e., through some sort of physical reset mechanism.
3. The controlling device may decide to release it and thus *instruct it to die*.
4. A higher authority, such as the manufacturer, shall hold *"the role of Shōgun [...] that can command the device to commit suicide"* [21].
5. The device may "commit seppuku[5]" as a consequence of tamper evidence and thus as one of the measures required to provide integrity.

The first case is especially interesting for devices that are not intended to be used for an extended amount of time, for example if some data provided by the device is only valid for a certain time period. Reset mechanisms can be used to put the device back into a well-known state, e.g, if a password has been lost and the device can no longer be accessed. The third case can be interesting for devices that should only be used together with another party and not function with other devices, e.g., a car key is only associated with one car and should not work with others. If the controlling party is broken, however, it could be of interest to be able to associate the device to a new device, which could be done through a higher authority (case 4). The difference to a physical reset mechanism is the need for an authorized party to execute the reset. The last case, in which the duckling willingly commits suicide, can prevent an attacker from gaining insights into the device such as critical information like private keys.

The decision which ways of death shall be considered for the system as well as the amount of information that shall be available from previous incarnations heavily depends on the actual device, its usage and especially the security requirements.

7.4.3 Including the Lessons learned from *The Resurrecting Duckling*

The idea of a transient association between a controlling device and the embedded system is exactly what would be interesting for a general gateway to the vehicle, i.e., one that would not necessarily be restricted to retrieving OBD data: The owner of the vehicle would associate the gateway with his consumer electronic device, while other devices may only have limited access

[5] Seppuku is a japanese ritual suicide by disembowelment reserved for samurai to die with honor, for instance to die rather than to risk to fall into the hands of their enemies.

to the gateway. Clearly, upon buying a new device such as a new smartphone, the owner should be able to transfer the control to the new device. Physical contact for a key exchange could be established through the serial port.

Evaluating the suitability of Imprinting

While a key exchange through the serial port may be the most secure way to create this association, it remains to investigate if it is appropriate for the current application: In the field of consumer electronics *usability* is of utmost importance. As already determined in Observation 9 of Section 6, the user experience of the consumer electronics should be preserved by all available means. New concepts required for security could clutter the application and ruin the benefits gained by using consumer electronics as interface. In this case, having to "register" e.g. a smartphone that is used with the gateway does just not feel right. The main reason for this is that it conflicts with the *Plug and Play* concept: The product should be *usable out-of-the box* with only minor to no setup required. In this case, the user would have to first associate the consumer electronics to the gateway and would only then be able to use it.

Thus it remains to investigate into other possibilities to establish an association, i.e., connect to the device, in a secure manner. As mentioned in Section 6.3.2, there is still no completely secure and simple procedure to setup a *secure* wireless connection, i.e., setting up a secure wireless connection with only minimal user interaction. *Wi-Fi Protected Setup*, for instance, defines a *Push Button Configuration*, which uses a button on the router to initiate a two minute timeout where clients can join the network without requiring a password. While this is a very elegant solution, there are still security issues related to this setup routine. Furthermore not all devices support Wi-Fi Protected Setup.

Another approach to the problem of initiating a secure wireless connection are *Factory defaults*: Upon first use, factory defaults allow to connect to the device *the same way as in normal use*, i.e., in terms of the connection routine there is no difference between the first connection attempt and successive connections. In terms of *The Resurrecting Duckling*, the manufacturer would take the role of the delivery nurse, preparing the duckling for its first contact with the mother, which herself has been prepared for this situation during pregnancy. Unlike the *Wi-Fi Protected Setup* procedure this approach still requires the user to enter a password, but most of the time the user has to type the password only once anyways, as most devices are capable of saving connection profiles. Having connected to the device, the user may or may not change the factory defaults, e.g., a new wireless password. This choice is at the same time the weak point of factory defaults: users tend *not* to change the default settings of devices, which clearly decreases security. There are two ways to approach this problem:

- Force the user to change the settings upon the first connection.
- Provide more or less unique parameters as factory defaults (whenever possible), such as random passwords and network names.

Both are valid approaches that are used in practice. The first approach, however, can be perceived as slightly bothersome by the user as it requires an additional step between the installation and the usage of the product (cf. Plug and Play). Another downside of this first approach is the simple fact that, e.g., choosing a secure password can be cumbersome, which is why many users tend to rely on relatively simple passwords. Thus the second approach is not only less intrusive, but in the majority of cases even more secure: Random factory defaults can be tailored to specific security needs.

All in all, assuming the existence of a shared secret for a secure wireless communication has been found to be preferable over relying on physical contact for key exchange.

Transience and state information

What remains to discuss from *The Resurrecting Duckling* is to determine how and when the device shall *"die"*. The state information saved by the device is mainly application specific: It depends on the usage of the device whether or not user- or configuration information should be stored. Section 7.6 will go more into detail about how the "life"-cycles of the product are handled, i.e., whether and when or not it should "die". Considering a general purpose gateway using wireless LAN the only consideration concerns with connection profiles. In general, network specific parameters such as password and network name, should be configurable and thus stored by the device. The user of this device has to know those parameters too to be able to connect to the device. The only thing to consider is thus that the user may loose this information and must therefore be able to force the device to commit suicide such that it returns to a known state - the previously discussed *factory defaults*. The most common approach is a simple reset-button, that has to be pressed for a long time.

Summary: The Product to keep in Mind

All in all, the "ambitious goal" that has accompanied the development of the current implementation has the form of the *Texa OBD Log* as depicted in Figure 7.4 with the only differences of a missing reset-button and a bulge for the integrated antenna (cf. Section 7.5). The serial port could be kept for PC application, but not for security reasons or communication with consumer electronics, which happens solely through wireless LAN.

7.5 Design of a Hardware Platform

Finding an appropriate hardware platform was one of the first tasks. The reason for this is the relatively long wait until the hardware is readily available, for instance delivery times. When considering custom hardware, it is even more difficult to meet a schedule: While the time for development and manufacturing can be mostly planned, the time needed to get the hardware to work can range from a few days to weeks, depending on design and manufacturing errors. Thus the first question was whether to develop a custom hardware platform or to rely on existing solutions, such as development boards.

7.5.1 An argument on custom design

The decision did mainly depend on the requirements to the communication interfaces, but as well on the *long-term* costs (cf. Requirement `Req.16`) and existing support such as drivers and development tools (cf. Requirement `Req.18`). What is meant with *long-term* costs are the expenses that would be required for the development of a *final product*: The production costs for custom hardware is high for low quantities, such that as a matter of fact custom hardware could never beat the price of, e.g., development boards or starter kits. The transition from a custom development board to a final product, however, is much easier from a custom hardware than from a standard kit, as all that's left to do is to remove unnecessary hardware components, re-arrange the board and minor changes to the firmware. The usage of a standard board would require a complete hardware design which would then still have to be tested. Adjustments needed for the firmware should keep within reasonable limits if the hardware consists of similar or the same components as used by development board.

Before going into detail about the requirements to the hardware there's one thing to quickly emphasize at this point: The restrictions imposed by the *MFi* program. As already elaborated in Section 6.2.2, Apple's *MFi* makes it virtually impossible to build an evaluation prototype that should communicate with an iPhone, iPad or iPod via Bluetooth or the serial port, as the product has to be practically finished for it to be approved. Therefore, for a development platform, Bluetooth has no longer been considered as a possible communication interface, while the serial port shall be kept for general purpose. In summary, the hardware should include the following basic functionalities:

- Wireless LAN
- CAN according to ISO 11898 (required for diagnostics)
- a serial port

While there are many microcontroller development boards available, the requirements of a CAN interface reduced the number to a manageable amount[6]. Digilent's *Cerebot MX7cK*

[6] A serial port is provided by most of the available platforms and does thus not really narrow down the choice.

Microcontroller Development Board [56] is one of those boards: It features a PIC32 Microcontroller [69] and among others one ethernet and two CAN interfaces. It lacks, however, the required wireless LAN capability, which could be added through a custom upgrade. Another very popular prototyping platform is *Arduino* [51], for which there exist a plethora of so called *shields* that can be stacked on top of the platform to provide further capabilities. Digilent's *chipKIT Max32* [57] is such an Arduino based prototyping platform. The dedicated *chipKIT Network Shield* [58] and *chipKIT Wi-Fi Shield* [59] provide the required CAN and Wi-Fi capabilities. While the expenses for those platforms are considerably less (ranging from about $100 to $150), at the end of the day the decision was the one to stick to the custom approach:

- Several reliable sources have discouraged from using the Arduino platform because of difficulties in customization
- Choosing a standard platform would still have required customization in order to work with the specifications of the diagnostic port: Most development boards need a pretty stable 5V power source, while the diagnostic port provides a nominal voltage of 12V to 14V and is subject to high variations.
- A custom platform would leave the door open for an eventual final product prototype as the required changes are assessable and can be minimized
- The custom development board could be used for future experiments by the department
- The development of a custom hardware platform completes the product development cycle and is thus a perfect fit for a master's thesis.
- Given the chance, one shouldn't hesitate to take the opportunity to design a hardware platform.

7.5.2 Key components

The choice of the key components was influenced by pricing, flexibility and the development tools. The key decision of any hardware project concerns the choice of an appropriate microcontroller or processor. Microchip [72], an American manufacturer of integrated devices (microcontroller, memory and analog semiconductors), offers a freely available, cross-platform development environment called MPLAB X [73]. The most prominent products of Microchip are the PIC microcontrollers [68], for which Microchip, amongst other things, offers a multitude of example applications and ready-to-use libraries. These libraries and collection of examples allow to concentrate on the essence of the implementation, without having to cope much with the implementation of e.g. drivers (cf. Requirement `Req.18`). One such library is the *Microchip TCP/IP Stack* [71], a freely available TCP/IP Stack implementation for PIC microcontrollers. This stack was the main reason for using a PIC microcontroller as the heart of the hardware platform. The following is a list of the key components selected for the development platform:

Microchip PIC32MX795F512H Microcontroller [33]. The main features that have justified the selection of a PIC32MX microcontroller are:

- Clock speed of up to 80 MHz
- 10/100 Ethernet MAC with MII/RMII Interfaces
- 2 x CAN2.0b modules with 1024 buffers
- 512K Flash (plus 12K boot Flash) and 128K RAM (can execute from RAM)
- Hardware RTCC (Real-Time Clock and Calendar with Alarms)
- Watchdog Timer with separate RC oscillator

The PIC32 is a powerful microcontroller capable enough for performing asymmetric cryptography: Using AES for encryption, a PIC32 running at 80MHz can still provide a throughput of up to 764 KBytes/sec for 128-bit AES, 634.4 KBytes/sec for 192-bit AES and 544.4 KBytes/sec for 256-bit AES [67] [28]. 512K of Flash and 128K of RAM provide plenty of space for powerful applications.

The CAN interface is required for todays on-board diagnostics, while the ethernet interface is interesting for future implementations and experiments (cf. Section 5.4). Using two CAN interfaces can be interesting for debugging and testing purposes, e.g., by simulating the CAN frames that would be expected to come from an actual vehicle.

A real-time clock and watchdog timer can be used for safety and security measures. All in all, a 32-bit microcontroller has been selected as it provides powerful features with a price tag that is not substantially higher than the one of smaller devices.

Microchip MRF24WB0MA Wi-Fi Transceiver Module [31]. The choice of this wireless module is closely related to the choice of the microcontroller: The MRF24WB0MA Wi-Fi module is designed for use with, amongst other PIC microcontrollers, the PIC32MX and the Microchip TCP/IP Stack. Its main features are:

- An Integrated PCB Antenna
- Simple four-wire SPI interface
- WEP, WPA-PSK, WPA2-PSK Security
- Data Rates: 1 and 2 Mbps
- IEEE 802.11 compliance, IEEE 802.11b/g/n compatible
- Range: up to 400m

The most interesting feature is the out-of-the-box support for WEP, WPA-PSK and WPA2-PSK security: Those security measures are also inherently supported by the majority of consumer electronic devices. Further details will be discussed in Section 7.6.2. The data

rates of 1 and 2 Mbps may seem little, as in Section 6.3.2 the bandwidth of Wi-Fi has been listed as an advantage of Wi-Fi versus Bluetooth, but is most of the time more than sufficient for embedded applications. For those which require a higher throughput, Microchip is already working on a new, faster module, the MRF24WG0MA [32], which will support up to 54 Mbps (and Wi-Fi Protected Setup). It is pin-compatible with the current module.

FDTI Chip FT232RL USB to serial UART interface [27]. This chip provides an integrated solution for a UART to USB interface, i.e., it handles the entire USB protocol such that there is no need for a USB specific firmware on the microcontroller. Royalty free device drivers are available for Windows, Linux and Mac OS. The supported baud rates range from 300 Baud to 3 MBaud.

Texas Instruments SN65HVD230 3.3V CAN Transceiver with Standby Mode [37]. For the OBD communication there were basically two possibilities: a custom approach using a standard ISO 11898 compatible CAN PHY, such as the SN65HVD230, or a ready-to-use integrated circuit as the ELM Electronics "OBD Interpreter" [60], which already implements the OBD protocol stack.

While using a ELM could severely cut implementation time, it would also limit the possibilities of the development board to on-board diagnostics. Furthermore, the costs for such an IC vary from 19$ up to 32$, which is more than double the costs of a PIC32 microcontroller and thus not acceptable.

Texas Instruments LM2940CT Low Dropout Regulator (5V) [36]. The LM2940CT is the automotive version of the very popular LM7805 voltage regulator (which is now "obsolete", but replaced by the UA7805 [38]). It is specifically tailored to the needs of automotive electronics, i.e., it is able to withstand strong voltage variations and has a dropout voltage of only 0.5V. It is used to provide a stable 5V from the 12V provided by the diagnostic port as effectively as possible.

Microchip MCP1700 Low Dropout Positive Voltage Regulator (3V3) [30]. Most of the previously presented components - in fact all but the FDTI chip - work at a voltage of 3V3. The LM2940CT, however, is only available down to an output voltage of 5V, which is why another voltage regulator is needed. The MCP1700 is available in an extremely small package, called Small Outline Transistor (SOT), is fairly cheap, and has no problem to provide 3V3 as output for a stable 5V input voltage. Notice that the FDTI chip can also be powered by the USB interface itself.

Reducing the development platform to those key components plus perhaps some LEDs and a pushbutton would provide a perfect basis for the design of a final product: The pin assignments, e.g., for the microcontroller, could remain valid, the only thing left to do would be to *remove* components such that only minimal adaptions to the firmware would be necessary to provide a running product.

7.5.3 The development platform

For a development platform, the main focus lies on providing as much flexibility as possible: It should not be completely tailored to the planned application, but also provide the facilities for further experiments and most of all consider possible future applications. Figure 7.5 depicts the first prototype of the development platform designed in the course of this inquiry.

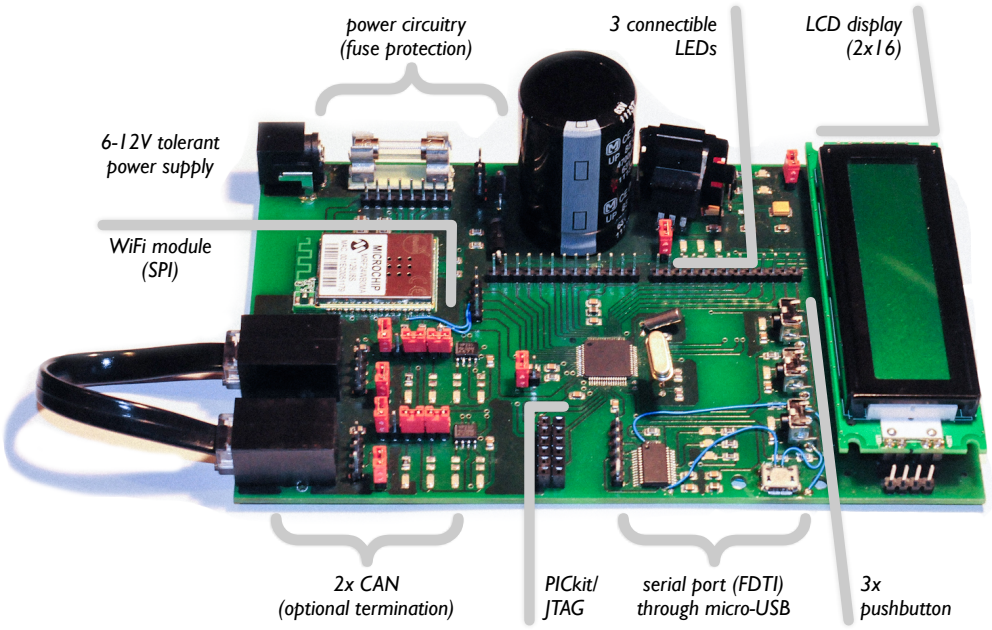


Figure 7.5: The first prototype of the development platform.

The board was designed using the freely available CadSoft EAGLE PCB Design Software [53]. As may be apparent from Figure 7.5, the initial design suffered some teething troubles, such as the classic of forgetting to cross receive/transmit nets. All in all, however, the initial problems kept within reasonable limits and did not slow down the project. The main features of the development board are:

Power circuitry: The development board features a rather sophisticated power circuitry that was proposed in [17] for automotive Arduino projects. One interrupt pin of the microcontroller is pulled high directly (protected by a zenerdiode) by the power-source, while the supply for the rest of the board is backed up by the rather gigantic capacitor. This way, upon a power-off, the microcontroller is supplied for just a little bit more time while knowing that the power will go away very soon. This could be used e.g. to close files.

In order to be able to perform measurements, both, the 5V and 3V3 supply for the board (behind the capacitor) can be cut off through a jumper. The power status for both voltages is indicated through two LEDs.

Optional LEDs: Just right to the capacitor there are three programmable LEDs. Those LEDs are currently used to indicate the status of the wireless LAN connection. Through a jumper right next to the LEDs, the supply voltage can be cut off such that the three pins can be used for other purposes through the pin-header right below the LEDs.

LCD: The 16 character x 2 lines LCD shares the pin-header with the LED ports and just as the LEDs it can be detached from the power source by a jumper such that the I/O pins otherwise occupied by the LCD can be used for other purposes. While a LCD display may not always be suited for debugging, it can be extremely useful for displaying volatile data, such as the current RPM obtained through on-board-diagnostics. Printing such data to a console (e.g. through UART over USB) is not that convenient.

Pushbuttons are extremely useful for debugging purposes. While more buttons would always be preferable, the three buttons provided by the development board turned out to be more than sufficient. They have been heavily used throughout the project and added great value to the board such that developing using this board has really been enjoyable.

It is worth to mention that all three buttons are pulled high and furthermore feature capacitors, such that de-bouncing them in software has not been necessary.

Communication Interfaces: There are basically four communication interfaces available on the development board

- The Wi-Fi module is connected to the microcontroller through a 4-wire SPI, which can also be accessed by a pin header, e.g., if another external had to be added it could use this pin header as SPI and a select-signal from another pin header like the one used by the LCD and the LEDs. There is no jumper for the Wi-Fi module as it has a dedicated hibernate pin that allows to put the module to sleep.
- The microcontroller provides two CAN ports and both are used on the development board. The CAN modules have three jumpers each[7], which are used for cutting off the power supply, enabling/disabling the module[8] and 120 Ohm bus termination.

  As already mentioned, having two CAN ports has proven to be extremely useful for debugging purposes, as the whole communication could be simulated and verified

[7] One jumper has been removed in the final version of the board.

[8] Pin 8 of the SN65HVD230 allows to select between three modes of operation: high-speed, slope control and low-power. Pin 8 is driven by the microcontroller, i.e., if the jumper is set and the pin is driven low, the IC enters high-speed mode. Using a resistor on this pin header allows for different rise and fall slopes. Applying logic high to pin 8 makes the controller enter listen-only mode. Refer to [37] for further details.

without external tools such as a CANalyzer, which is a very powerful but expensive tool for analyzing CAN traffic [84].

- The serial port is formed by the FDTI chip and a micro-USB port. The FDTI chip is powered by the USB port, which allows to perform power-cycles on the board without loosing the connection. During this project, the port has been used for debugging and logging purposes.
- The PICkit/JTAG port is used for programming and debugging: Depending on the actual tools, one might prefer to use a JTAG programmer/debugger instead of the tools provided by Microchip. In the course of this project, the PICkit 3 programmer/debugger [34] has been used.

All in all, the development board has proven to be very enjoyable to use. It provides sufficiently many means for debugging and direct interaction. During this project, the pushbuttons have been used to interfere with the execution without having to use a debugger (debugging using the PICkit is rather slow), the LEDs have been used to display the Wi-Fi connection status and the LCD to display status information such as the current IP-address of the hardware, OBD status (disconnected/error/running, cf. Section 7.6) or fast-changing data such as the retrieved parameters (e.g. engine RPM).

The serial port has mainly been used for debugging. The fact that it can operate at high speeds was extremely helpful, as it allowed to log CAN and wireless LAN traffic without having to implement an interrupt driven output function, i.e., the data could be logged using simple blocking functions, which would not be possible for low baud rates because it would block the TCP/IP stack operation.

A short remark on considered extensions and applications

The pin assignments for the 64-pin PIC32 on the development has been chosen with great care to provide as much functionality as possible. Figure 7.6 is a snippet from the development board's schematic and gives a rough overview of the pin assignments. Basically, the only "hard-wired" pins, i.e., the pins that are not usable in another way, are the CAN I/O, the Wi-Fi I/O pins "enable" and "hibernate", the interrupts for power-off and the Wi-Fi module, the pushbuttons (whose pins are input-only) and the UART RX/TX pins to the FDTI chip[9]. The LCD, LED, Wi-Fi SPI and the remaining pins (see below) are all accessible through pin headers by removing the appropriate jumpers. The pin headers can be found just below the big capacitor (cf. Figure 7.5). There are two large 13-pin headers side-by-side:

- The left pin header is designated to allow a possible *Reduced Media Independent Interface (RMII) [75] Ethernet* application. It features all the pins necessary to connect an RMII 10/00 Ethernet Transceiver, such as the LAN8720 by SMSC [35].

[9] Practically, even the RX/TX pins could be used through the pin header next to the FDTI chip when no USB cable is plugged into the micro-USB port.

- The right pin header consists of the pins for the LCD and the three LEDs. Both modules can be disconnected through jumpers (or in the case of the LCD completely removed from the board) such that the pins can be used for other purposes.

Table 7.2 gives an overview over the functionalities available through the pins of both pin headers (aside from the general-purpose I/O functionality of the pin) and whether or not Ethernet/LCD/LED usage is still possible when using the feature. Furthermore it lists the pins required from the Ethernet header (E) or the LCD/LED header (L). By just disconnecting the LEDs, for instance, one could use the pins for I^2C communication, as interrupts or capture inputs, while ethernet and LCD usage are still possible.

Usage	RMII Ethernet	LCD	LEDs	Affected Pins
UART4 (RX/TX only)	√	-	-	L8, L9
UART1 (RX/TX only)	√	-	√	L2, L3
UART1 (incl. RTS/CTS)	-	-	-	L2, L3, L9, E11
Analog Pin AN15	-	√	√	E2
Analog Pin AN5	-	√	√	E1
Change Notification Inputs CN13-16	√	-	√	L4, L5, L6, L7
Change Notification Input CN12	-	√	√	E2
Change Notification Input CN7	-	√	√	E1
Interrupts INT1-3	√	√	-	L8, L9, L10
Capture Inputs IC1-3	√	√	-	L8, L9, L10
Inter-Integrated Circuit I^2C3	√	-	√	L9, L10
Inter-Integrated Circuit I^2C1	√	√	-	L2, L3
Serial Peripheral Interface SPI3 (w/o SS)	-	√	-	E11, L2, L3

Table 7.2: Alternate applications for Ethernet pin header (E) and LCD/LED pin header (L).

As mentioned, the left pin header is explicitly intended for RMII Ethernet. The intention for this is to allow the development board to be used for future versions of on-board diagnostics too, which will most likely happen through ethernet (cf. Section 5.4). The main motivation behind the remaining functionalities were the following two resp. three features that would be interesting for a future automotive gateway (cf. Section 8):

SD and MMC cards. The current hardware can only save parameters by using the program flash, which cannot be written too often. Using the information given in [70] the hardware could be easily extended to read and write to SD or MMC cards. This would allow offline data-exchange between the hardware and a target, such as a computer.

The approach described in [70] does only require a SPI and three general purpose I/O's. The SPI could be shared with the Wi-Fi module, as it is accessible through a separate pin header, while the three I/O ports could be taken from either of the two large pin headers (e.g. by just disconnecting the LEDs, both, ethernet and LCD would still be operational).

GSM/GPRS/GPS. GSM/GPRS and GPS are very interesting for further experiments on the "Internet of Things". The application note [29] describes how to use a LEON200 [39] GSM/GPRS module together with a NEO-6 [40] GPS module through a UART connection. Such interfaces can be used for Machine to Machine communication, which is a very hot topic in automotive, e.g., *Car to Car* or *Car to Infrastructure* communication (refer to [26, Chapter 9, Page 415] for a short overview).

Thus, extending the platform with SD/MMC capabilities together with GSM/GPRS is currently possible. This would provide a very powerful platform for further experiments, featuring internet access, GPS positioning, storage capabilities, wireless LAN, a serial port, CAN interfaces and push buttons.

The main motivation for this was a prospect for a general purpose gateway between automotive and consumer electronics, which is definitely needed for future vehicles as consumer electronics are slowly becoming a part of our lives and thus consistently change the way we live. Section 8 will go more into detail about this.

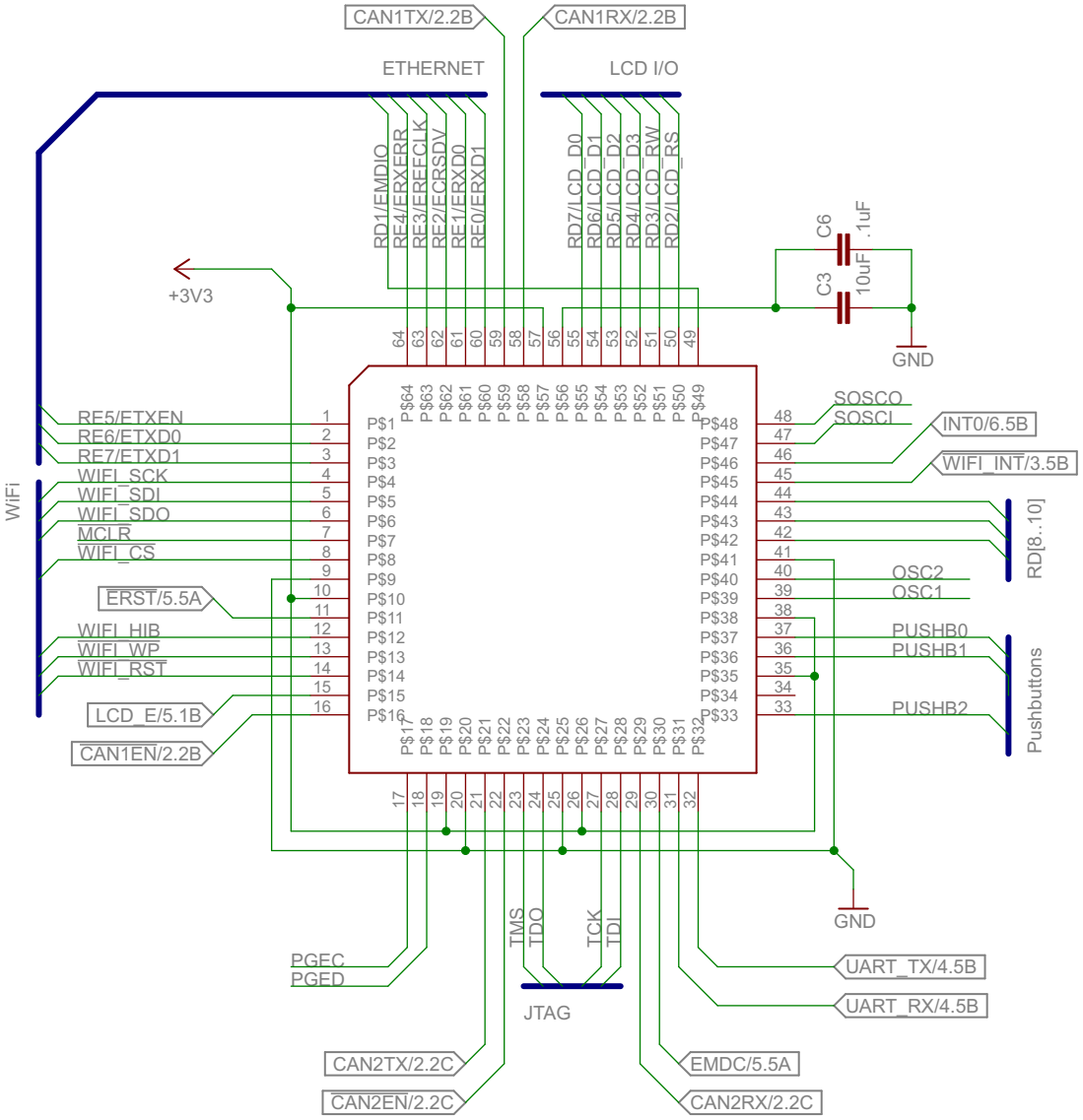


Figure 7.6: PIC32 pin assignment (schematic snippet).

The *Reduced Media Independent Interface (RMII) Ethernet* interface consists mostly of the pins in the upper left corner, but also includes $\overline{\texttt{ERST}}$ (reset) on pin #11 and $\overline{\texttt{EMDC}}$ (RMII signal) on pin #30. `INT0` is used as power-off interrupt by the power circuitry, `SOSCO/SOSCI` and `OSC1/OSC2` are the pins for the primary and secondary oscillator, `RD[8..10]` is where the LEDs are connected to. `PGEC/PGED` are the pins used for debugging/programming by the PICkit3. Pin #34 is the only pin that is not connected: it can only be used USB applications as bus power monitor.

7.6 Gateway Implementation

Section 7.3 has already given a sneak peek at what the implementation looks like: The gateway implements a subset of the on-board-diagnostic services, which in this case are parts of the services `0x01` and `0x09`. The application on top of those services allows to

- retrieve the Vehicle Identification Number (VIN) and
- to add or remove monitors for certain parameters.

This section will go more into detail about the implementation of the gateway, i.e., the firmware that is currently running on the development board. First, the basic architecture is discussed, then the applied security measures are explained and finally a more detailed discussion of the communication interfaces, which are the OBD application for gateway/vehicle and the TCP/UDP Framework for gateway/client communication, is given.

7.6.1 Basic architecture

Implementing the monitoring services, which will also be referred to as *Live-Data* in the remaining document, requires bi-directional communication on both interfaces: *Clients* request to add a parameter to the Live-Data list and the gateway has to provide/forward the data accordingly. The data is obtained from the vehicle through the OBD services, which basically follow a simple request/response protocol (cf. Section 7.6.3 for a detailed discussion).

Leaving the monitoring-approach aside for a moment, the information flow would essentially be the following: The client wants to know the value for a certain parameter and thus requests it from the gateway. The gateway forwards this request to the vehicle which responds with the corresponding data. This data is then sent back to the client by the gateway. The actual hardware interfaces have the following properties:

Client/Gateway. The communication between the Client(s) and the gateway happens through wireless LAN. As mentioned in Section 7.5.2, the hardware supports data rates of up to 2 Mbps.

Gateway/Vehicle. The chosen interface for the on-board-diagnostics is the CAN bus, which according to ISO 11898 supports bit rates of up to 1 Mbps for network lengths below 40m. The Diagnostic on CAN standard, however, defines the bit rates for ISO 15765-4 compliant communication to be either 250 or 500 kBit/s.

Thus, both interfaces operate at different speeds and the gateway is responsible to resolve this mismatch, i.e., it has to make sure that the communication is ported from the faster interface to the slower interface while the latter must not be overstressed. The other way around, i.e., from the slow interface to the fast interface, is typically not an issue.

Decoupling Interfaces

This boundary problem is a common problem in digital design, where two parties have to communicate at different clock speeds. It is typically solved using dual-clocked or rate-matching first in first out (FIFO) queues. As discussed in Section 3.3.1 those approaches make it possible to completely de-couple the interfaces, such that both communicating parties can still operate at the defined speeds but inherently regard the speed of the other party. The approach pursued in the current implementation is not a 100% decoupling of the interfaces, but takes into account all the requirements presented in Section 7.2 and allows a simpler implementation. Figure 7.7 depicts an overview of the gateway architecture. The two main ingredients are the Communication Framework and the OBD Application:

Communication Framework. The management of client connections is handled by the Communication Framework: It makes use of a TCP and UDP server, which are built on top of the Microchip TCP/IP stack.

Those connections are stream-oriented, i.e., it may be that data send to or received from a client is split into multiple send/receive events. In order to be able to provide an application interface for clients, the Framework supports *framing* of transmissions, i.e., the communication happens through *frames*, which are marked with Start-Of-Frame, End-Of-Frame and Escape characters, such that clients can send well defined requests and receive well defined response messages. Section 7.6.4 gives further details about the Communication Framework. Clearly, sending and receiving such frames must also be supported on the Client side, which is elaborated in Section 7.7.

Another part of the Framework's responsibility is to forward the requests to the OBD Application. It is important to mention that while the Framework and thus the gateway supports multiple clients, the actual OBD Application has currently no notion of "clients".

OBD Application. Requests to the vehicle and the corresponding response handling is done by the OBD Application. It is responsible to establish a connection with the vehicle via CAN and send OBD service requests. For this purpose a custom OBD Stack has been implemented: On-board-diagnostics do not require a full implementation of the CAN transport protocol as specified by ISO 15765-2; the requirements for OBD are defined by ISO 15765-4. Thus, this stack only implements the functionality that is actually required for on-board-diagnostics. A more detailed discussion of the implemented OBD communication is given in Section 7.6.3.

As apparent from Figure 7.7, the Communication Framework and the OBD Application are separated by a PID list, i.e., leaving the positive-response arrow aside for a moment, there is no direct interaction between the Framework and the OBD Application.

In contrast to the aforementioned FIFO, the PID list is static, i.e., of fixed size. The elements of the list are pre-defined, e.g., a list of parameters defined by the manufacturer that are allowed

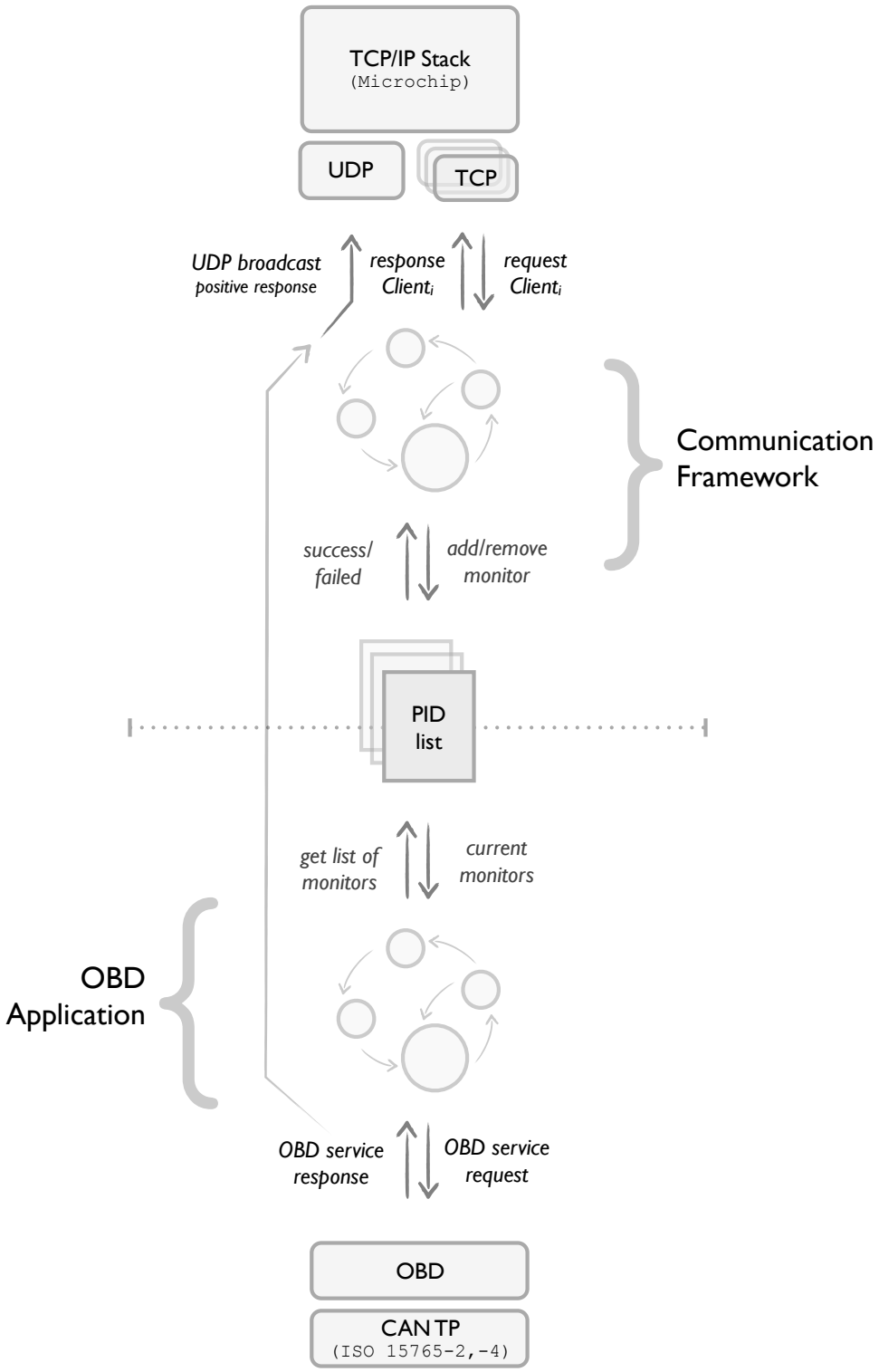


Figure 7.7: Visualization of the Gateway firmware architecture.

to be retrieved. The list of currently implemented parameters has already been introduced in Section 7.3, Table 7.1 (page 66). The list is used in the following way:

- A client connects to the gateway via TCP and requests the monitor of a specific PID by sending a pre-defined *frame*. The Connection Framework checks whether or not the parameter can be monitored, that is, whether it is actually in the list of allowed PIDs *and* supported by the vehicle. If it is supported, the Framework increases the *reference count* of the PID in the list: This way the PID list is independent of the actual client that has requested the monitor, as it only saves how many monitoring-references there are for every PID. The Framework then responds to the client with a frame that indicates whether or not the PID could be registered for monitoring.

- The OBD Application, on the other hand, has two responsibilities regarding the PID list: When first connecting to the vehicle, for every allowed PID it has to request whether or not the PID is actually supported by the vehicle, i.e., if there is any electronic control unit (ECU) that actually provides a value for the PID (cf. Section 7.6.3 for further details). This support information is saved in the PID list such that monitoring requests are rejected for unsupported PIDs.

  The other responsibility of the OBD Application is to actually perform the OBD service requests for currently monitored PIDs: It periodically checks for monitored PIDs and builds service requests, which can consist up to 6 PIDs per request. Thus the application cycles through all the monitored PIDs, grabs the appropriate number of PIDs, builds and issues the request and waits for the response(s) or timeout. The response(s) or timeout are handled according to the on-board diagnostic standards, only then the next request can be issued.

Thus, while the Communication Framework can handle requests from the clients at the maximum supported speed, the OBD Application can independently handle the OBD communication according to the OBD standards. In this sense, one can speak of a *complete decoupling* of the interfaces, as all the information has to be passed through the PID list, which functions as real-time database.

There is, however, one more important thing to say about this architecture: The above explanation does not mention how the actual value of the PIDs is passed to the Framework and thus to the clients, it only describes how monitors are related to actual OBD requests. Basically, there are two possible ways how state information, such as parameter values, can be accessed:

- Either the client executes periodical requests for the value, i.e., *polling*

- Or the client is informed about changes to the value.

The first approach, i.e., *polling*, would correspond to a truly decoupled implementation: The parameter value would have to be stored into some type of memory, e.g. the PID list, which

is read by the Framework and written by the OBD Application. The current implementation, however, follows a different approach: Just as depicted in Figure 7.7, the OBD Application directly forwards positive responses of PID requests (which contain the values for the requested PIDs) to the Communication Framework. The Communication Framework then broadcasts this information via UDP (cf. Section 7.3.2). This approach essentially includes two decisions: Instead of polling, the clients are informed about value changes. Furthermore, broadcasting eliminates the need to send the values to each client (cf. Section 7.3.2). There are several reasons why this approach has been chosen:

- The main reason is that broadcasting allows to reduce the communication overhead in the case when clients request monitors for the same parameters (cf. Section 7.3.2).
- In contrast to polling, the communication overhead is again significantly reduced: Every client would have to poll for the value of every PID it has registered for monitoring. This way, every client is informed about value changes through a single broadcast.
- The choice for notifications instead of polling is at the same time more elegant: The only thing a client needs to do is register/unregister monitors. There's no need to cycle through all the monitors, as the gateway informs about value changes as soon as they are available.
- It eliminates the need to handle invalid values: Polling for a value that has never been received must result in a special error handling, i.e., one that indicates that the gateway has not yet received any data for the PID.

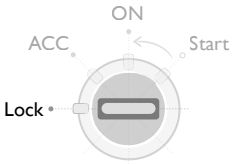
Thus, all in all, it heavily simplifies the communication between clients and the gateway while also simplifying the implementation: Clients do not need to poll for data and the gateway does only have to forward the data received by the vehicle to the clients.

Considering the fact that the communication between clients and the gateway is about four to eight times faster than the communication between vehicle and the gateway, this is a valid approach. Furthermore, the data is broadcast using UDP, thus lost packages will not result in communication overhead.

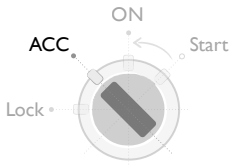
There is one last thing to notice regarding decoupling for generic service requests: Surely, the choice of service `0x01` to demonstrate decoupling is a very intuitive one, as it allows the use of a static list for decoupling. There is, however, no major difference for implementing generic service requests using decoupling: Just like in digital design, one could either use a FIFO to queue service requests, or simply not allow multiple requests per clients. Thus a client may issue a service request to the gateway, which is acknowledged by the Framework. As soon as the OBD Application is ready, the request is executed and the client is notified about the results. If a client issues another request while the last one has not been indicated as complete, the Framework may simply issue with a *"busy"* response and thus refuse the service.

Hardware states and communication availability

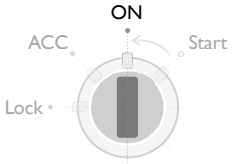
Powering the hardware through the diagnostic port has as a consequence that the power-cycles directly depend on the state of the ignition lock of the vehicle[10]. The ignition lock, however, does not only determine the state of the power supply of the hardware, but also the state of the vehicle's ECUs and thus of the CAN bus that is used for diagnostics. Thus it may be that the hardware is powered on while the CAN communication is not *yet* available. The following shall give a better understanding of the implications introduced by the ignition key on behalf of a common 3-stage ignition lock:



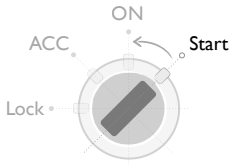
The initial position of the key is *Lock*, which enables the steering-wheel lock and the engine immobiliser. At this stage the diagnostic port does neither deliver any current nor is any communication possible through CAN.



By inserting the key and turning it clock-wise until it clicks into place for the first time, the ignition lock is put into the *ACC-umulator* state, which enables the power supply of the electronics powered by the automotive battery, such as the radio. The diagnostic connector, however, does not yet provide neither current nor CAN communication.



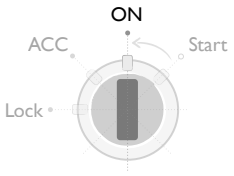
Only after the key has been turned until the next click, current is also delivered to the remaining electronic systems, such as the ignition or the powertrain control module. At this stage the diagnostic connector provides power and communication with the control units is possible (as now they are themselves powered on) - assuming that OBD through CAN is supported by the vehicle.
Requesting current powertrain data (Service `0x01`) while being at *ON*, however, will not provide much information as the engine is not yet running.



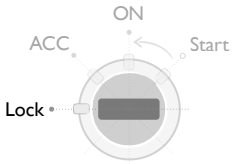
Turning the key further clock-wise will fire up the engine *(Start)*. The ignition lock is typically equipped with a spring mechanism, such that it will automatically go back to *ON* after the key is released.

[10] W.l.o.g. a common 3-stage ignition lock may be assumed, i.e., an ignition lock that uses a traditional car key. Start/Stop buttons, as used in current BMW cars, could not be tested as none were available, but are nonetheless covered by the described sequence.

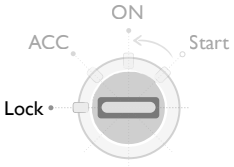
Thus, if the hardware is connected right from the start, power and communication will be possible right away. But what happens if the ignition is turned off again?



Ignition is currently *ON*, the engine is running and the hardware is powered on. The powertrain provides current diagnostic data that can be forwarded to the clients.



Turning the key counter clock-wise will stop the engine and CAN communication *but the diagnostic port does still provide current.* This is very convenient as the gateway will still be possible to communicate with the clients.
Notice that turning the key back to *ON* would re-enable CAN communication.



As the engine was previously running, only after the key is removed from the ignition lock (which is typically accompanied with a *clock*-sound) the diagnostic port will cut off the power supply causing the hardware to shut down.

Considering the fact that the hardware could be plugged in at any time, e.g., after the engine has already been fired up, there are basically three possible combinations of communication/power state to consider:

1. Most of the time, the hardware should be powered on and CAN communication should be possible. This is always the case if the ignition lock is *ON*, regardless of whether or not the engine is running.

2. If CAN communication is not supported by the vehicle, or if the hardware is plugged in after the engine is powered off (cf. upper sequence before removing the key), then the hardware is supplied by the diagnostic port while not being able to communicate with the vehicle through CAN.

3. Following the sequence described above, i.e., plugging in the hardware before starting the engine, then firing the engine and turning it back off, leaves the hardware powered on but does no longer allow communication through CAN. The hardware, however, was *previously able to talk to the vehicle*. The key difference between this case and the previous one is, that the hardware knows that CAN communication is possible, just not at the moment.

The resulting hardware states are depicted in Figure 7.8. It is important to notice, that the hardware does not require any interaction at all from the user to switch between the states. It is simply plugged into the diagnostic port (cf. Plug and Play, Section 7.4.1). The OBD

Application takes care of determining the communication state and provides this information to the clients by notifying the Communication Framework about changes to the hardware state. The Communication Framework then passes this information to the clients, which should react appropriately, e.g., inform the user about the changes. The following is a short description for the state-machine depicted in Figure 7.8:

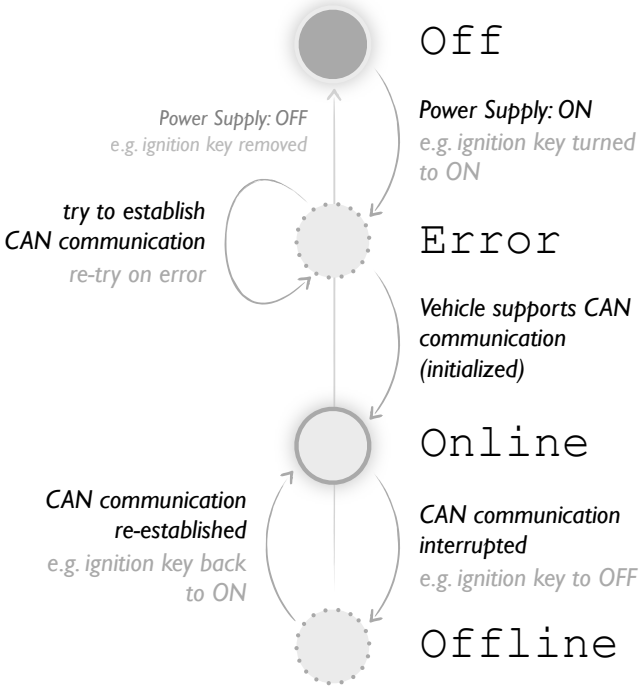


Figure 7.8: Visualization of the Gateway hardware states.

As soon as the hardware gets powered on, the OBD Application invokes an initialization sequence to determine whether or not communication through CAN is currently possible. If there's no reply from the vehicle, the application remains in the `Error` state and retries to execute the initialization sequence periodically. As soon as it succeeds, it changes to `Online`, where service requests can be executed. Notice that the engine is not necessarily running while the application is in the `Online` state. As described before, it can happen that the gateway looses the CAN connection while still begin powered on, thus it may change to `Offline`. The fundamental difference between `Offline` and `Error` is that vehicle specific information, such as supported parameters, do not need to be requested anymore. Furthermore, it provides a more fine-grained state information, i.e., the user can be sure that his vehicle is supported by the gateway and that he only needs to turn the ignition back to *ON* for the application to provide data (switch back to `Online`).

What's left to discuss is the *"sudden" shut-down*, which can happen from basically any state as hinted by the non-stop straight arrow to `Off` in Figure 7.8: As soon as the key is removed, the hardware is no longer supplied with current and thus there's no possibility whatsoever for further interaction with the clients. At the same time, however, *communication is no longer needed*: If the engine is shut down and the electronic control units cannot communicate with the gateway, then the gateway itself cannot provide any more useful information to the clients and may as well be powered off. The only thing that may be bothersome for the user is that the Wi-Fi connection is dropped without interaction, i.e., the user cannot decide when the communication is over through his device. There are three major reasons why this is nonetheless preferable over e.g. a battery extension that would allow the hardware to remain active:

1. First of all, by turning off the ignition *and removing the key* the "main" user, i.e., the vehicle owner, has already put an end to the communication with the vehicle and thus indirectly disabled all the functionality the gateway hardware could provide. The only natural action for the user would be to turn off the communication between gateway and consumer electronics directly afterwards. Instead of keeping the hardware alive, the only thing to consider here would be to handle the disconnect *on the client side* as unobstrusively as possible and to keep the application responsive, e.g., by providing offline data (cf. Section 7.7).

2. Secondly, using the diagnostic port power cycles directly couples the gateway with the visual display of the car's dashboard: If there's nothing to be seen on the dashboard, then there's nothing to be done with the gateway. This way, the dashboard indirectly serves as a status display of the gateway. Furthermore, a typical user would already be familiar with the power cycles of the car and would, e.g., know that the ignition has to be set to *ON* for the air-conditioning to work. Coupling the gateway hardware with these power-cycles seamlessly integrates it along with the other electronic systems.

3. Thirdly, the gateway has been designed to allow multiple users. Therefore, if the gateway should not shut down before a user explicitly disconnects from the hardware this would imply that it would have to keep track of how many users are currently connected before being allowed to power-off.

   While this may be preferable from an application's point of view, the vehicle owner can never be sure that the hardware is indeed turned off without removing it or getting some visual feedback, which on the other hand is hard to get because the hardware is assumed to be out of the passenger's line of sight. Clearly, one could again define a "super-user" upon whose disconnect the hardware shuts down, but at least for this application this would be bad design.

In short: Any connection to the client(s) will be dropped upon a power-off, but this is not a problem as the gateway cannot provide any functionality whatsoever. *It is the responsibility*

of the application running on the consumer electronics to remain responsive to user interaction regardless of the hardware state, i.e., the gateway's functionality should not be influenced by consumer electronics applications (cf. Section 7.7). The gateway only provides on-board-diagnostic data and that's that. The current implementation follows this approach, i.e., the iOS application remains responsive even while the hardware is offline and the hardware is directly powered by the diagnostic port.

Life-cycles and state information.

In Section 7.6.1 it has been mentioned that *parameter values are not stored* by the gateway but forwarded when positive responses from the vehicle are received; just according to the motto "I'll tell you if I get news about the parameters you've requested". Together with the previous discussion about hardware- and power states this raises the question what information should be kept for how long, e.g., whether or not the gateway should save client information or vehicle information. This question has already been touched in Section 7.4.3: How and why shall the gateway *"die"*? The general proposal was that connection profiles should be saved and be re-settable to factory defaults by a reset button.

The discussion about connection profiles requires a short explanation regarding Microchip's TCP/IP Stack. In short: Changing connection profiles is currently only possible by uploading an updated firmware. The reason for this is that basically connection profiles are set up using C-header files, i.e., the network structure (AdHoc/Infrastructure), IP and security settings are all statically defined. The stack includes a library called *"Zeroconf"* which would basically allow run-time configuration, but the documentation of Microchip libraries is very sparse or non-existant, such that most information has to be retrieved from example applications. The hours that would be needed for the implementation of this feature were one reason not to do it. Every feature included in this project means doubled effort: It needs to be implemented on the gateway and on the client side. Furthermore, it would not greatly contribute to this inquiry: It is not a problem to execute experiments using different firmwares. Thus, connection profiles are subject to future implementations providing a richer feature-set. This project forms the basis for such implementations.

Future implementations shall use a re-set button to be able to load default connection profiles. The current implementation is "prepared" for such a feature as it already includes a reset button whose action, however, is empty. Regarding other possible information, such as saved parameter lists, client settings or known vehicles, the decision for the current implementation is the following:

> The gateway hardware does not save any information whatsoever, i.e., it "dies" with every power cycle.

This approach is as simple as it is elegant: Including state information does only increase the complexity of the gateway and would only provide features for *specific* client applications:

Not every target application that wants to access on-board-diagnostic information through this gateway would necessarily want to save information. If they would really decide that they need to save something, they can do it themselves. Consumer electronics are way more powerful than the gateway hardware, and everything that can be delegated to the consumer electronics should be delegated to it.

> The gateway application's only concern is about the *functionality transfer* between clients and vehicle, i.e., in this case it should make the vehicle's on-board-diagnostic services available to the user of the consumer electronics device. Saving information makes no part of on-board-diagnostic services and should therefore be implemented by the target application.

7.6.2 Security analysis

Until now, security and usability have always been a matter of trade-offs, and it looks like this is not going to change for a while. Standards like Wi-Fi Protected Setup are definitely the right move to make security easy to use and are a promising approach to diminish the needs for custom protocols, especially in conjunction with consumer electronics.

This is exactly what was intended with this case study: When working with consumer electronics, it is very important to balance the pros and cons of custom protocols. As already emphasized with Observation 8 (page 46), consumer electronics are a very promising user interface for embedded systems. Users are already familiar with the basic interface concepts and it is the responsibility of the developer to take advantage of this situation. Complex security measures could destroy the user experience of device, which is why developers should take the maximum advantage out of the means that are provided by the device, such as known security protocols. Before actually identifying the security measures, it may be convenient to recall the security-relevant requirements given in Section 7.2:

`Req.4`: The gateway hardware must by no means facilitate an attack on the vehicle via the diagnostic port, i.e., it must not amplify the attack surface of the vehicle.

`Req.5`: The data that is exchanged through the gateway shall be protected against eavesdropping using appropriate security measures.

`Req.6`: The gateway shall respect the security requirements that are specified by the on-board diagnostic standards.

Requirement `Req.4` is basically a countermeasure to Observation 10, which emphasizes that consumer electronics (or gateways in a broader sense) can amplify the attack surface of the systems they are connected to. But what does this imply for the diagnostic connector?

Existing security measures for the diagnostic connector

The current implementation uses the CAN bus provided by the diagnostic port. CAN does not bring a great deal of security measures. In effect, CAN communication is quite insecure. The papers [7] and particularly [8] provide a good insight into this topic. The following is a *simplified* round-up of current basic security problems in CAN networks:

- CAN frames do not identify the sender, they are based on IDs. Everyone can send a message with any ID, thus no *authenticity* can be guaranteed
- *Availability* is an issue in CAN networks too: An attacker could simply flood the network with the message with the highest priority.
- The CRC sum included in CAN frames is not sufficient to provide *Integrity* of the message content.
- *Confidentiality* is also an issue in CAN networks: CAN frames are broadcast and every receiving party can decide on its own whether or not to use (or even forward) the content.

In short, without special means like protocols or encryption, CAN doesn't do any good at providing security. The CAN bus used for on-board-diagnostic does not implement any such protocols. Also, Section 4.2 has given a short insight into general security measures currently used in automotive, with basically the same result: There are only few to no security measures in the automotive sector, which is why Requirement `Req.6` is always fulfilled.

In the context of Requirement `Req.4` an amplification of the attack surface would therefore only be the fact that the consumer electronics introduces more ways or interfaces to interact with the CAN bus. In this specific implementation, the gateway communicates with the target device via wireless LAN, eliminating the need for physical contact with the diagnostic port. Thus, the Wi-Fi interface of the gateway has to provide security measures that restrict the access to authorized parties.

Security measures supported by consumer electronics and the gateway hardware

The current gateway hardware uses the MRF24WB0MA Wi-Fi transceiver, as presented in Section 7.5.2. This module supports WEP/WPA-PSK/WPA2-PSK security right out of the box. The applied security measure depends on the network type:

- Ad-hoc networks do only allow WEP security, which is not regarded as secure.
- Infrastructure networks, i.e., networks with a router, support WPA2 security. If no authentication server is available, the network key is pre-shared. This is the only option available for the MRF24WB0MA.

In general, infrastructure networks should be preferred. Future vehicles will most likely include wireless LAN capabilities and thus an infrastructure should be available. As already discussed, the current project only supports static connection profiles which, however, can be simply changed by replacing a header file and re-compiling the firmware. There exists a header file for ad-hoc networks and one for infrastructure networks. A future implementation should allow to dynamically change those settings and provide the facility to change the wireless key.

WPA2 is broadly considered secure. As every other protocol with a static key (which can be changed though, but must be changed by all users), the security strongly depends on the choice of the key. Strong keys include special characters, numbers as well as lower and uppercase characters. As mentioned in Section 7.4.3, a strong key should be provided as factory default, which should be changeable using the consumer electronics. For the key change, it would be preferable to randomly generate a new key which then is displayed to the user instead of requiring the user to choose a (secure) key.

Both network types are typically supported by consumer electronics, including inherent support for the security protocols. The use of the MRF24WB0MA transceiver allows to kill two birds with one stone: To connect to the gateway, the user has to setup the connection anyways, which includes possible security settings. Thus, the steps for security setup and connection setup are combined into a single step that is completely handled by the standard user interface of the consumer device.

Current security concept

Figure 7.9 depicts the pursued security concept. It's underlying the following basic assumptions:

Physical Contact: The lack of security measures provided at the diagnostic port makes it reasonable not to consider attacks through physical contact, as the attacker could just as well disconnect the gateway and perform the attack on the diagnostic port. This assumption is still in accordance with Requirement `Req.4`, as it does not *amplify* the attack surface.

Tamper Resistance: The previous assumption that physical contact does not have to be considered implies that tamper resistance is not needed.

Availability: Wireless LAN cannot provide any form of protection against availability attacks. In fact, *jamming* can prevent any form wireless communication. Clearly, there exist counter-measures such as frequency hopping. Those mechanisms, however, still cannot provide complete protection against availability attacks, which is why such attacks are not considered in this analysis.

Attacks: In general, attack scenarios must be considered for both interfaces of the gateway. In this case, however, it is reasonable to assume that the vehicle does not pose a threat to the gateway and thus consumer electronic device. Furthermore, with physical attacks being ruled out, only attacks via wireless LAN have to be considered.

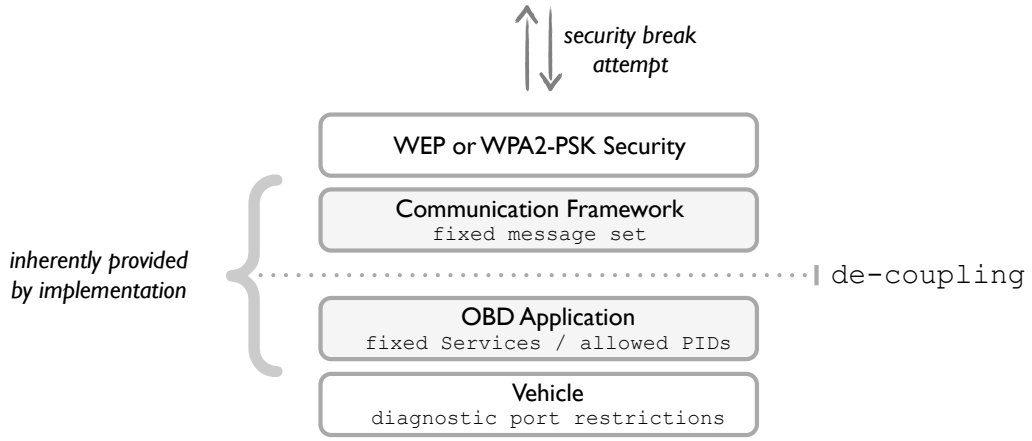


Figure 7.9: Security levels of the Gateway implementation.

The first step required to infiltrate the gateway is to break the Wi-Fi security, which is either WEP for ad-hoc networks or WPA2-PSK for infrastructure networks. Those security measures provide authenticity, confidentiality and integrity of the system, as it cannot be physically compromised. One cannot shove aside the possibility of a technical break of this level, but can basically assume that the procedure requires very high amount of time and - if passwords are changed frequently - uninterrupted access to the device. The fact that the up-times of the device are rather short and that most of this up-time the system is *moving* complicates the act of breaking it and would require, e.g., to bug the vehicle (a drive-by attack can be considered infeasible). Nevertheless, once broken, the attacker can subsequently connect to the device until the password is changed.

What the attacker cannot do by only breaking wireless security is to directly compromise the system by saving information: the gateway is perceived as completely stateless and would forget all of the information with the next power cycle (which will happen soon). Thus the invader would have to repeat the attack on every power cycle.

The next levels of security base directly on the design decisions taken for the application and the vehicle-inherent resistances: First of all, the Communication Framework has a fixed set of messages that it can handle, every other message is simply ignored. Secondly, the control flow from the target application and thus from the attacker to the actual OBD interface is simply not present: The interfaces are de-coupled (only the other way around back-pressure is possible), i.e., the attacker cannot force an arbitrary behavior of the gateway module by only breaking the Communication Framework (e.g. through loopholes). If both, the Communication Framework and the OBD Application (and thus the whole application) are infiltrated, the vehicle itself can provide some sort of resistances, e.g., by de-coupling the OBD CAN bus from the vehicle's internal bus systems.

In short: An attacker would have to break through wireless security and *completely* circumvent the gateway's firmware to be able to compromise the vehicle through the CAN bus of the OBD connector. Simple target applications that are allowed to talk to the gateway have no control over the vehicle whatsoever, they can only *retrieve* information through the gateway. For this project, the provided security measures are more than sufficient and also promote security protocols that are inherent to the consumer electronics. Furthermore, future hardware such as the MRF24WG0MA [32] can be integrated into the project without major changes.

7.6.3 The OBD protocol stack

Having explained the basic concepts and architecture pursued in this project, it remains to give a rough overview over how OBD service requests and responses actually look like, how they are executed and how they are translated into CAN frames of 8 Bytes size. The basic structure of the OBD protocol stack has already been introduced by Figure 7.7 (page 87) of Section 7.6.1, Figure 7.10 reveals more details about the stack. It will serve as reference for the following description, which is accompanied by an exemplary request. The following should give a short heads-up about the involved layers:

OBD Application Layer: The actual OBD functionality of the gateway is defined through the OBD Application, which implements a subset of the service `0x01` (request current powertrain diagnostic data) to provide a monitoring service for vehicle parameters; and retrieves the Vehicle Identification Number (VIN, part of service `0x09`). Responses to OBD requests are expected to arrive within a certain time-out.

OBD Layer: Requests issued by the OBD Application consist only of information about the service and its parameters which should be executed. The CAN TP, however, needs more information to issue the request, such as addressing information. The OBD layer adds this information and forwards the request to the CAN TP layer.

Regarding service responses, the OBD layer makes sure that the OBD Application does not time-out requests while there is still communication happening on the lower layers, i.e., it basically reloads the timer used by the ODB Application to time-out requests until it is clear that no further response is pending.

CAN TP Layer: The CAN Transport Layer is responsible for fragmentation and de-fragmentation of requests and responses that exceed the CAN frame size of 8 Bytes (so called *multi-frames*). Furthermore, it maps the address information provided by the OBD Layer onto actual CAN Identifiers, be it Standard IDs or Extended IDs.

CAN Layer: The lowest layer is basically a non-blocking CAN driver, using the Microchip libraries for the CAN transceiver. It allows to initialize, re-initialize or disable the CAN

module, to send CAN frames and to retrieve received CAN frames in a de-coupled manner, i.e., the reception of a CAN frame is indicated via a flag such that another layer may collect the frame as soon as it is its turn to execute - instead of being forced to process it upon the reception, which may result in a chain of overlapping calls whose execution could take too long.

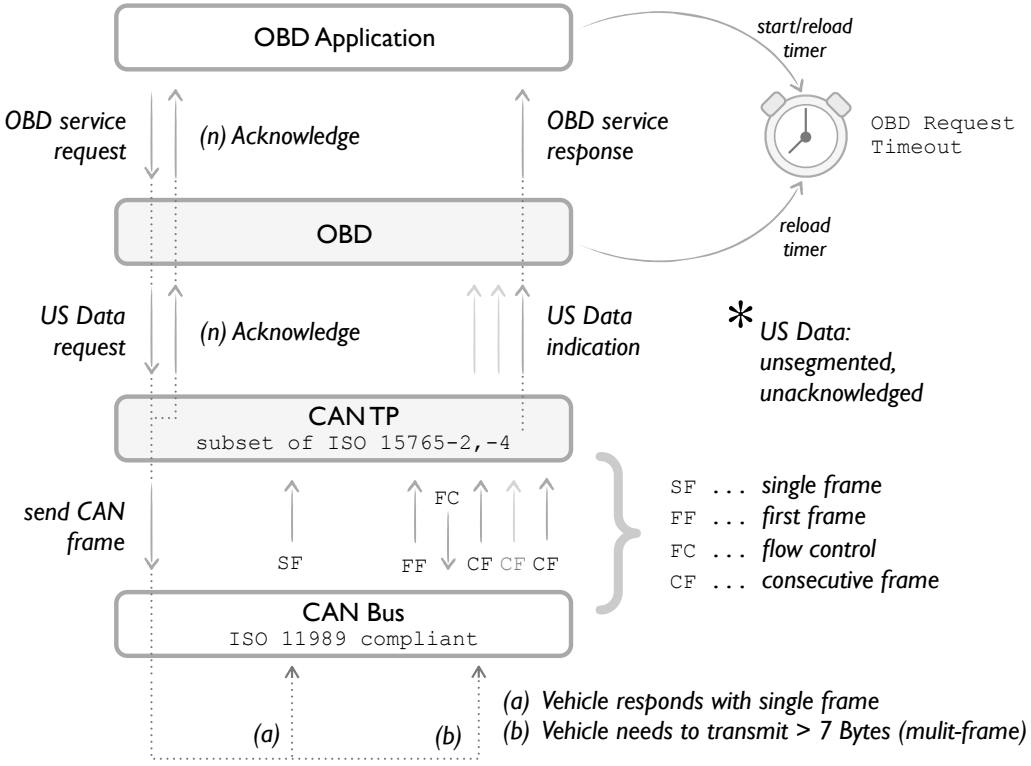


Figure 7.10: Detailed depiction of the OBD Stack implementation.

OBD Application

Before going more into detail about the stack architecture, it is convenient to first take a look at the actual OBD Application. Figure 7.11 gives an abstract model of the state-machine used for the application.

Obviously, this state-machine looks a lot like the hardware states depicted in Figure 7.8 (page 92): The application starts in the `Start` state. The first thing to do is to check the CAN communication. This is done by invoking the OBD initialization sequence provided by the OBD layer, which is defined in ISO 15765-4. The initialization sequence checks whether or not

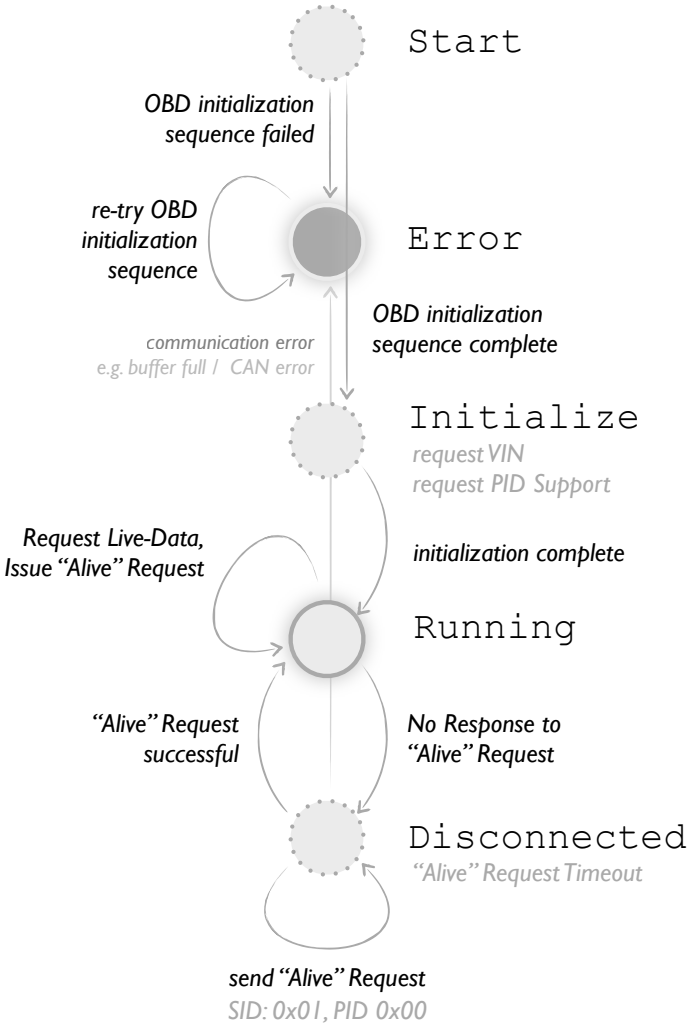


Figure 7.11: Visualization of the OBD Application states.

there is an error or a response from the vehicle when using a combination of the baud rates 250 and 500k together with SID/EID addressing methods. If there is no response, the application changes to the `Error` state[11]. It does, however, not remain there silently but keeps executing this initialization sequence as there is still a chance that communication will be possible (cf. Section 7.6.1).

Once the initialization sequence has been executed successfully, the application knows which baud rate and identifier is used by the vehicle and continues with the software initialization

[11] As hinted in Figure 7.11, the `Error` state is reachable from all the other states too, e.g. upon CAN communication errors.

(`Initialize`), which consist of the retrieval of the Vehicle Identification Number (VIN) and the synchronization of the PID support for the parameters that can be monitored. Only when it is done with this sequence it changes to `Running`.

What is simply labeled as `Running` in Figure 7.11 includes the whole procedure of fetching current monitors and executing the corresponding OBD requests, together with an "Alive" check: The application periodically sends out an OBD request with `SID=0x01` and `PID=0x00`. This request is defined as *"universal initialization/keep-alive/ping message for all emission-related OBD ECUs"* by ISO 15031-5 and must be supported by the vehicle[12]. If such an "Alive" request times out, the application switches to `Disconnected` but continues to issue those requests in order to be able to switch back to `Running`.

It bears repeating to mention that the whole OBD Application runs without user interaction. The gateway is simply plugged in and runs regardless of how many users are connected via Wi-Fi. The only difference is that live-data requests are actually executed if one or more users request monitoring of certain parameters.

Service `0x01`; Requesting Current Powertrain Diagnostic Data

The parameters that should be monitored by the OBD Application are requested through the OBD service `0x01`. Thus, in order to request the value of parameters like the vehicle speed, the OBD Application has to build an OBD service request. Table 7.3 shows how such a request message for CAN communication (ISO 15765-4) has to look like according to ISO 15031-5[13]:

Data Byte	Parameter Name
`#1`	Request Service Identifier (SID)
`#2`	service-specific data byte #1
`#3`	service-specific data byte #2
`#4`	service-specific data byte #3
`#5`	service-specific data byte #4
`#6`	service-specific data byte #5
`#7`	service-specific data byte #6

Table 7.3: General request message format for ISO 15765-4 according to ISO 15031-5

Without going further into detail about the CAN Transport Protocol at this point it should be mentioned that this request message fits into a single CAN frame. As every OBD request message is restricted to this size, the Transport Layer does not have to provide the capability

[12] The same message is used by the initialization procedure to determine whether or not the vehicle is ISO 15765-4 compliant.

[13] Notice that the byte numbering in the ISO 15031-5 document starts with *one* and not with *zero*. For consistency reasons it is better to stick to this notation.

to send multi-frames. Thus, as hinted in Figure 7.10, every service request issued by the OBD Application layer will be translated into a single CAN frame. Service `0x01` shall serve as the example that will accompany this round-up of the OBD stack. For service `0x01`, the request message has to look like the one illustrated with Table 7.4:

Data Byte	Parameter Name	Hex Value
`#1`	Request Current Powertrain Diagnostic Data request SID	`0x01`
`#2`	PID#1	`xx`
`#3`	PID#2	`xx`
`#4`	PID#3	`xx`
`#5`	PID#4	`xx`
`#6`	PID#5	`xx`
`#7`	PID#6	`xx`

Table 7.4: Request message format for service `0x01` according to ISO 15031-5

Thus, the application may only request at most 6 parameter at a time, the PIDs #2 to #6 are optional. As an example, the OBD Application might want to request the number of emission-related DTCs, the MIL status, the engine RPM and the engine coolant temperature. Table 7.5 shows how the request message would look like:

Data Byte	Parameter Name	Hex Value
`#1`	Request Current Powertrain Diagnostic Data request SID	`0x01`
`#2`	PID: Number of emission-related DTCs and MIL status	`0x01`
`#3`	PID: Engine RPM	`0x0C`
`#4`	PID: Engine Coolant Temperature	`0x05`

Table 7.5: Examplatory request message for service `0x01`

Service `0x01`; Current Powertrain Diagnostic Data Response

For the moment, it is convenient to stay at the OBD Application level and to look at the response that is expected by the application. Table 7.6 depicts the general format of a response message to the service `0x01`. The first thing to notice is the length. This is where the CAN Transport Layer comes into play: The message length may exceed the maximum 8 Bytes that fit into a standard CAN frame. It is the responsibility of the CAN TP layer to stitch multiple CAN frames messages together such that at the end of the day the application will get a single response message. Secondly, not every parameter has the same amount of data-bytes: The bytes B to D

are optional. And finally, not every requested PID is necessarily supported by the responding ECU: The response message only includes the PIDs it actually has values for.

Data Byte	Parameter Name	Hex Value
#1	Request Current Powertrain Diagnostic Data response SID	0x41
	data record of first supported PID = [	
#2	PID #1	xx
#3	data A	xx
#4	data B (optional)	xx
#5	data C (optional)	xx
#6	data D (optional)]	xx
⋮	⋮	⋮
	data record of *m*-th supported PID = [	
#n-4	PID #*m*	xx
#n-3	data A	xx
#n-2	data B (optional)	xx
#n-1	data C (optional)	xx
#n	data D (optional)]	xx

Table 7.6: Response message format for service 0x01 according to ISO 15031-5

Table 7.7 shows a possible response for the request defined in Table 7.5: The ECU has a positive response to service 0x01, that is 0x40|(SID=0x01)=0x41 as the value of the first byte. The number of emission related DTCs and the status of the MIL are both contained in the first byte of the PID 0x01: The bits 0-6 define the number of DTCs while the status of the MIL is represented by bit 7 of byte A. The other information contained in PID 0x01 indicates which emission related tests are supported by the vehicle as well as their completion status.

For PID 0x0C ISO 15031-5 defines two bytes of payload which together represent the current engine RPM. The PID 0x05 on the other side does only provide one byte of payload. Annex B of ISO 15031-5 gives a comprehensive list about supported PIDs, the information contained in the payload and scaling information.

Service 0x09; Retrieving the Vehicle Identification Number

Other services, like service 0x09, are basically executed the same way as service 0x01: The request message is still the same as defined by Table 7.3 and somewhat similar to the request message for service 0x01 (Table 7.4), with the only difference that the "PIDs" are replaced by so called "InfoTypes" and that the Service IDentifier is now 0x09 instead of 0x01. Even

Data Byte	Parameter Name	Hex Value
#1	Request Current Powertrain Diagnostic Data response SID	0x41
#2	PID: Number of emission-related DTCs and MIL status	0x01
#3	MIL: ON; Number of emission-related DTCs: 3	0x83
#4	Misfire-, Fuel System-, Comprehensive monitoring	0x33
#5	Catalyst-, Heated catalyst- ... monitoring supported	0x0C
#6	Catalyst-, Heated catalyst- ... monitoring test (not) complete	0x0C
#7	PID: Engine RPM	0x0C
#8	Data byte A: 667 rpm	0x0A
#9	Data byte B: 667 rpm	0x6B
#10	PID: Engine Coolant Temperature	0x05
#11	Data byte A	0x6E

Table 7.7: Examplatory response message for the request defined by Table 7.5

the response message is basically the same - again "PIDs" are replaced by "InfoTypes" and the response SID is 0x49. In general, the ISO 15031-5 standard can be a bit misleading and is a bit cumbersome to read, as it is filled with tables that sometimes only differ by single values.

Addressing modes

The first stop on it's way down to the actual CAN bus, the service request is passed to the OBD layer. As mentioned before, the OBD layer is, amongst other things, responsible for adding addressing information to the OBD request message. Basically, the CAN TP defined by ISO 15765-2 would allow two different addressing schemes: *remote diagnostics* and *diagnostics*. Remote diagnostic would allow to extend the address range, but OBD only uses the *"diagnostic"* message type[14]. Messages of the *"diagnostic"* type are addressed using the following parameters:

Target Address (TA): a 1 byte address defining the receiver

Source Address (SA): a 1 byte address defining the sender

Target Address Type (TAtype): a single bit that allows to choose between

[14]ISO 15765-2 defines a Message Type MType, which can be *diagnostic* or *remote diagnostic*. For OBD, however, the Message Type is defined to be *diagnostic*. For further information please refer to ISO 15765-2 and ISO 15765-4.

- *physical addresses*, which are unique for any participating entity, e.g., a fixed address for the powertrain control unit, or
- *functional addresses*, which allow to address multiple entities, e.g., all ECUs that should respond to emission-related diagnostic requests.

Thus, using physical addresses it would be possible to send requests directly to a specific ECU, while using a functional address may be useful if it is not known what ECUs should respond to the service, e.g., for OBD it is not initially clear, which ECUs should respond to a service request, thus functional addressing is used. Table 7.8 shows the possible combinations defined for OBD in ISO 15765-4:

	Target Address (TA)	Source Address (SA)
Funcional request	Legislated OBD system = `0x33`	External test equipment = `0xF1`
Physical response	External test equipment = `0xF1`	Legislated OBD ECU = `0xXX`
Physical request	Legislated OBD ECU = `0xXX`	External test equipment = `0xF1`

Table 7.8: Diagnostic addresses used for OBD according to ISO 15765-4

As apparent from Table 7.8, functional OBD requests have a fixed source- and target address. Basically, every OBD request can be addressed using functional addresses. This is also what's done in the implementation: Every request issued by the OBD Application is provided with the functional addresses `TA=0x33` and `SA=0xF1` by the OBD layer. Functional *responses* are not defined by ISO 15765-4, as every ECU responds using physical addressing. The `0xXX` in Table 7.8 is a placeholder for the ECU number, which can range from 1 to 8. Clearly, once it is known that an ECU responds to a diagnostic service, it could also be addressed using a physical request.

What's left to do to actually being able to continue sending the message, is to map the address information consisting of the target address, source address and target address type to an actual CAN identifier. This is done in the CAN TP layer. ISO 15765-2 defines three different mapping schemes, denoted as *normal fixed addressing*, *extended addressing* and *mixed addressing*. All of those have to consider 11- and 29 bit CAN identifiers (SID/EID) and whether or not the target address type is functional or physical, resulting in a plethora of possible mappings. Fortunately, ISO 15765-4 defines the

- *normal addressing format* for 11 bit CAN identifiers (SID) and
- *normal fixed addressing format* for 29 bit CAN identifiers (EID)

as the only options to be used for OBD. Without going into great detail, 11 bit CAN identifiers have a fixed mapping, i.e., ISO 15765-4 defines a list of 17 CAN identifiers that are to be used for the corresponding TA/SA/TAtype combinations, while for 29 bit CAN identifiers the source address byte, target address byte and the bit indicating functional/physical addressing are mapped directly into the CAN identifier.

Thus, the 5 bytes defined by Table 7.5 have now been passed down to the CAN TP layer and got a corresponding CAN identifier. At this point, the CAN TP issues an acknowledge to the OBD layer, which is forwarded to the OBD Application layer, which starts a time-out by which a positive response to the request is expected. All that's left to do is to put the request onto the CAN bus, which brings us to the topic of message handling in the CAN Transport Protocol according to ISO 15765-2.

CAN TP message handling

In order to be able to send and receive messages with length greater than 8 bytes, ISO 15765-2 defines four message types that are used by the transport protocol. Figure 7.12 depicts the structure of the messages:

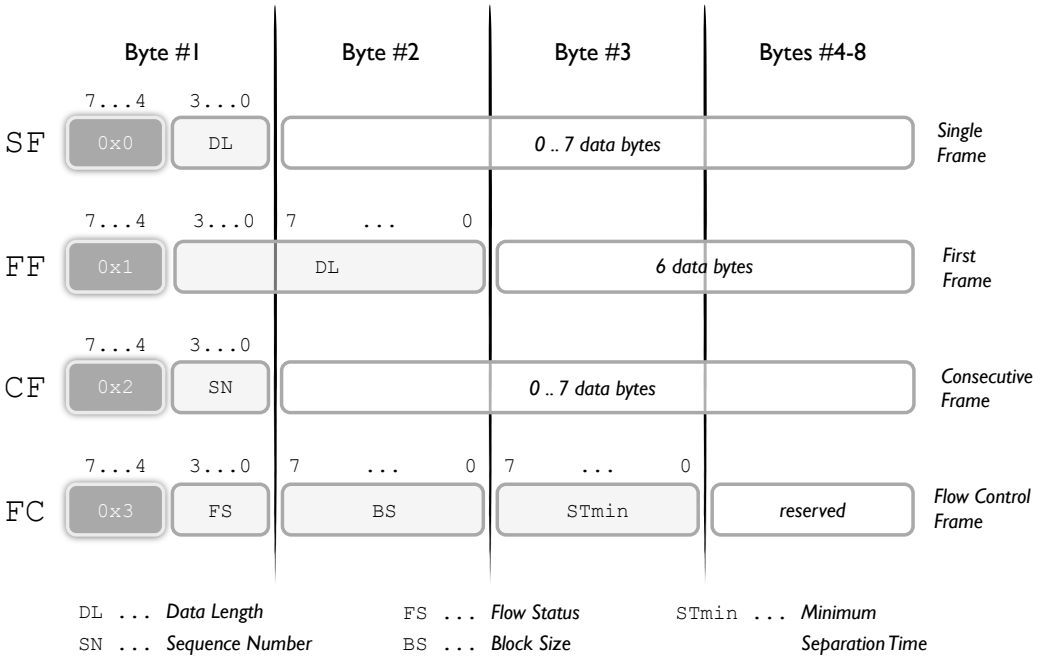


Figure 7.12: Message types defined for the CAN TP in ISO 15765-2

The message type is identified by the four most-significant bits of byte #1 of the CAN frame. The first type, *Single Frames*, are used to transmit/receive messages with a payload of at most

seven bytes. The actual number of valid data-bytes contained in the message is saved in the *data length* `DL` field, which consists of the four least-significant bits of byte #1. Thus, the request example defined by Table 7.5 would be mapped to a *Single Frame* with `DL=0x5`. As all OBD requests according to ISO 15031-5 consist of at most seven bytes, every OBD request is actually a *Single Frame*. Thus, the current implementation does not support the sending of so called *multi-frames*. It has, however, to be able to receive them. Sending more than 8 bytes is enabled through the remaining three message types. Figure 7.13 depicts the sending of a so called *multi-frame*:

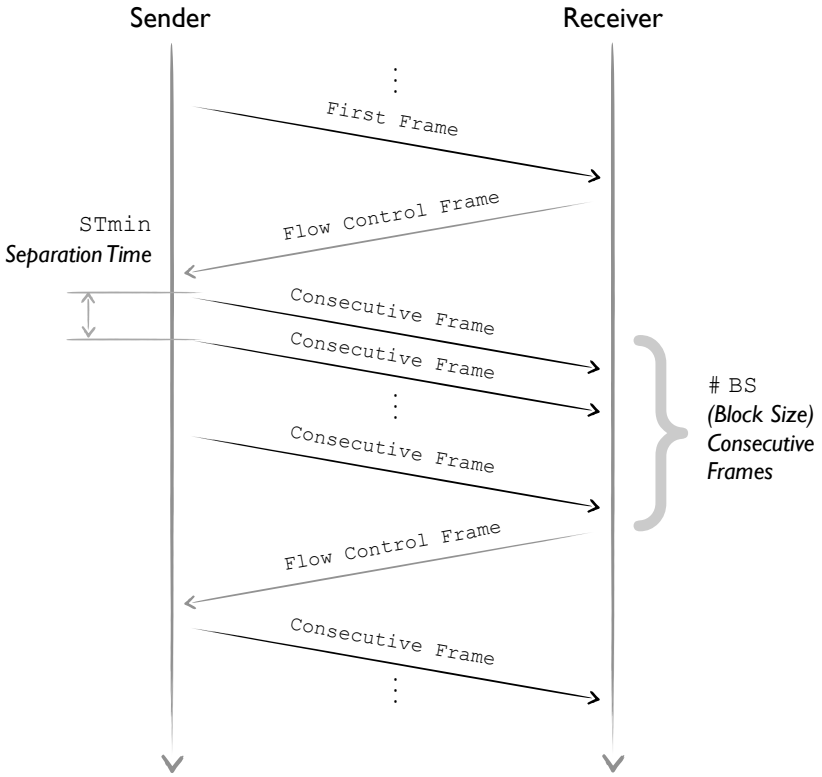


Figure 7.13: Basic flow control mechanism for multi-frames

First of all, the sender transmits a *First Frame*, which contains the first 6 bytes of the message payload. Notice that every multi-frame *must* consist of at least 8 bytes of payload, as otherwise a *Single Frame* would have been sufficient. The *First Frame* furthermore contains a *data length* `DL` field, which indicates the total number of data bytes the receiver has to expect. The first six bytes of payload transmitted through the *First Frame* are included in this number. Thus, the receiver can calculate how many messages it will take to send the data and also how many data bytes will be contained in the last frame. After sending the *First Frame*, the sender waits for a

Flow Control Frame from the receiver.

By sending a *Flow Control Frame*, the receiver would have the possibility to control the amount of *Consecutive Frames* he receives until he issues the next *Flow Control Frame* (through the *block size* `BS` field), how long the sender has to wait between sending two messages of a block (through the *separation time* `STmin` field), and tell the sender whether he should stop or continue to send or notify him about an overflow (through the *flow status* `FS` field). ISO 15765-4, however, fixes the values for `BS`, `STmin` and `FS` to zero, the consequence being that only a single *Flow Control* has to be sent by the receiver (the external test equipment) which indicates that the sender may send all the *Consecutive Frames* at once with no separation time needed.

The *Consecutive Frames* only consist (at most) seven data bytes and a *sequence number* `SN` which is increased with every further message. *Consecutive Frames* have to be received in the proper order, else the resulting multi-frame should be ignored by the external test equipment.

Thus, all in all the CAN Transport Layer takes care of CAN specific matters, such as mapping the address information to actual CAN IDs and the transmission and reception of *variable length messages* rather than single CAN frames. For receiving multi-frames this implies that the CAN TP is also responsible to send the *Flow Control Frame*, i.e., the OBD Application should not have to bother with flow control decisions.

Reception of vehicle responses

The last few pages have lead all the way down to the CAN bus and provided all the information that is necessary to understand the basic principles of sending data to the vehicle and receiving a multi-frame. Thus, the request defined by Table 7.5 has reached the vehicle and the vehicle has sent back the information in the form of a mulit-frame as defined by Table 7.7. Table 7.9 summarizes the whole request/response message flow for the previously mentioned example.

Step (a). The service request is passed down to the CAN TP, which packs it into a *single frame* and sends it to the vehicle

Step (b). The vehicle sends back a *first frame*, informing the CAN TP that it should expect 11 data bytes. As the *first frame* itself already contains 6 bytes of data, the CAN TP at this point already knows that only one *consecutive frame* will follow and that it will contain only 5 of 7 available data bytes.

Step (c). In order to tell the vehicle to continue sending, the gateway sends back a *flow control frame*. `FS=0` indicates that the vehicle shall *continue to send*, `BS=0` means that the transmission is not divided into blocks, i.e., the vehicle shall send all the remaining frames, and finally by setting `STmin` to zero the gateway informs the vehicle that it may send the frames as soon as they are ready with no waiting time whatsoever.

Step (d). Having received the *flow control frame*, the vehicle sends all the remaining frames, which in this case is a single *consecutive frame*.

(a) Gateway → Vehicle; Current Powertrain Diagnostic Data *Request*

Byte	Content	Hex Value
#1	Frame type: *Single Frame* 0x00; *Data length* DL=4	–
#2	Request Current Powertrain Diagnostic Data request SID	0x01
#3	PID: Number of emission-related DTCs and MIL status	0x01
#4	PID: Engine RPM	0x0C
#5	PID: Engine Coolant Temperature	0x05

(b) Vehicle → Gateway; *Response:* "this is going to be a multi-frame of size 11"

Byte	Content	Hex Value
#1,#2	Frame type: *First Frame* 0x01; *Data length* DL=11	–
#3	Request Current Powertrain Diagnostic Data response SID	0x41
#4	PID: Number of emission-related DTCs and MIL status	0x01
#5	MIL: ON; Number of emission-related DTCs: 3	0x83
#6	Misfire-, Fuel System-, Comprehensive monitoring	0x33
#7	Catalyst-, Heated catalyst- ... monitoring supported	0x0C
#8	Catalyst-, Heated catalyst- ... monitoring test (not) complete	0x0C

(c) Gateway → Vehicle; *Flow Control:* "ok (FS=0), continue to send *all* remaining frames"

Byte	Content	Hex Value
#1	Frame type: *Flow Control Frame* 0x03; *Flow Status* FS=0	–
#2	*Block Size* BS=0	0x00
#3	*Minimum Separation Time* STmin=0	0x00

(d) Vehicle → Gateway; *Response*

Byte	Content	Hex Value
#1	Frame type: *Consecutive Frame* 0x02; *Sequence Number* SN=1	–
#2	PID: Engine RPM	0x0C
#3	Data byte A: 667 rpm	0x0A
#4	Data byte B: 667 rpm	0x6B
#5	PID: Engine Coolant Temperature	0x05
#6	Data byte A	0x6E

Table 7.9: Current Powertrain Diagnostic Request/Response message example.

What has not yet been discussed is how and why the transmission is actually timed out by the OBD Application. When an external test equipment, or in this case the gateway, is first connected to a vehicle, it does not yet know which services and parameters the vehicle actually supports. Therefore, upon issuing a request such as the above example, the gateway does not know whether or not any ECU might respond to the request. Without going into great detail, it is worth noticing that for every possible request, the external test equipment (or gateway) can determine which ECUs can respond to it. But initially, this information is not known.

For that reason, ISO 15031-5 defines application timing parameters, which amongst others involve the so called `P2CAN` timeout. This timeout defines the time by which an ECU that has a valid response to the issued request must respond to it, i.e., either the ECU responds within the `P2CAN` timeout, or it does not respond at all. As more than just one ECU can respond to such a request, the timeout is reloaded after every response such that every ECU has the chance to respond to the request. While this is a clean approach for single frame responses, as soon as multi-frames are involved the timeout procedure is no longer sufficient: Single-frame responses are received as a whole, thus reloading the timeout upon a reception of a single frame is sufficient. For multi-frames, the timeout is only reloaded after the reception of the *first frame*. ISO 15031-5 defines the appropriate handling for every possible case, but basically, the application timing can be broken down to the following:

- The application must always wait for mulit-frames to complete, i.e., even if a `P2CAN` timeout occurs, the application still has to wait until all the pending multi-frame has been received as a whole.
- If a timeout occurred while at least one multi-frame was still pending, the application may immediately issue the next request upon the completion of all multi-frames
- If all pending multi-frames have completed before the `P2CAN` timeout, then
  - the application must wait for the `P2CAN` timeout if it *does not know* which ECUs respond to the request, or
  - the application may immediately issue the next request if it has received a response from every ECU *that responds to the request*, i.e., it knows the exact number of responses for the request.

The fact that the application does not have to wait for a timeout if it knows that it has already received all the possible responses has as a consequence that a clean separation between the OBD layer and the application layer is not possible: Either the OBD Application would have to cope with first/single/consecutive-frame receptions, or both, the OBD layer and the OBD Application have access to the timeout. For this project, the latter approach has been chosen, i.e., the OBD Application does only *start* and *stop* the `P2CAN` timeout and receives responses as a whole from the OBD layer, regardless of whether or not the response had to be transmitted as

multi- or single frame. It issues the next request if it either has received all responses and thus does not have to wait for the timeout, or if a timeout occurs *and no multi-frames are pending*. The OBD layer is responsible for reloading the `P2CAN` timeout. In order to be able to do so, it gets an *indication* from the CAN TP layer if:

- A *single frame* has been received,
- a *first frame* has been received or
- the reception of a *multi-frame* is complete or has failed.

Upon the reception of a single- or first frame, the OBD layer reloads the timer as there might be yet another ECU that wants to respond to the request and has not yet had the possibility to do so. Regarding first- and multi frames, the OBD layer keeps track of how many multi frames are currently pending, as the OBD Application must not issue the next request before all pending multi frames have been received or failed. Thus, the only thing left to do in the OBD Application layer is to wait for the timeout to happen and all pending multi frames to be finished before issuing the next request.

7.6.4 A remark on the Communication Framework

While the OBD Stack is completely application specific, the Communication Framework is not: The framework could be used together with any other application too. As apparent from Figure 7.7 (page 87) and discussed in Section 7.3.2, the communication model allows for UDP broadcasts (send only) and TCP clients (send and receive). In the current implementation, the UDP broadcast is used to send positive responses to OBD service `0x01`, while client communication, i.e., monitoring management, is done via TCP. The communication for both, TCP and UDP, bases on *frames*, i.e., the data that is sent is provided with a *Start-of-Frame* and *End-of-Frame* character. Clearly, the payload has to be escaped using *Escape Characters*, e.g., if it contains a *Start-of-Frame* character as payload. The only requirement to the clients is that they too have to support this simple framing procedure.

The rest, however, is completely generic and can be changed in the twinkling of an eye: The clients communicate with the framework through a set of messages that is known by both, the gateway and the client. Upon the reception of a certain message, the framework performs the corresponding action and responds to the client. By simply exchanging the messages and calling the appropriate functions, the Communication Framework can be used by any application, e.g., using the free I/O ports of the development board.

This has been proven extremely useful during the development phase: Changing debugging projects was like shelling peas, adding and removing simple functions to the gateway could be done without any impact on the actual OBD Application, by simply isolating the message set of the OBD Application from the new functionality. The simulator presented in Section 7.8.1 is just one example of how easy it is to extend the capabilities once the clients and the gateway are able to communicate using the Communication Framework.

7.7 Closing the Design Cycle: The Clients

As already mentioned in Section 7.3, the target applications - or clients - implemented in the course of this project base on the iOS / OS X platform. While one reason that lead to this choice was clearly personal interest, the main motivators were the vast developer community and the freely available toolchain [50]. Furthermore, a preceding investigation concerning possible target platforms has lead to the `CocoaAsyncSocket` [55] libraries, which promise easy and ready-to-use abstractions for communication through sockets, for both iOS and OS X - a promise they have been able to keep. Finally, the programming language used for iOS and OS X development is *Objective-C*, which, at the end of the day, is still *C*.

It doesn't pay to go deep into the implementation details of the clients in this thesis. In this Section, the iOS target application is briefly discussed. Another client, the Simulator (which is a OS X application), will be touched in Section 7.8.1.

7.7.1 The challenge: from `(UINT8 *)` to CoreData

The gateway firmware is a straight *C* application, where everything is handled using basic datatypes such as unsigned integers or pointer to some location in memory. The content of the frames sent through the *Communication Framework*, for instance, is defined by an array of simple 8 bit unsigned integers, which is basically managed through a pointer to the first element, i.e., `(UINT8 *)`. When moving to the client application, however, such primitive datatypes are seldom used for anything but loop variables.

This is where the term *"Objective"* of the *Objective-C* programming language comes into play: Standard *C* is an imperative programming language, *Objective-C* is *object-oriented*. Thus, instead of primitive datatypes, there are objects for everything: Arrays, strings, dates, numbers and even *null* are all objects now. The same holds for the frames sent by the gateway: In a manner of speaking, the gateway sends them away as simple arrays, but the target application receives them as objects. As soon as there are objects, it is no longer pure computer engineering, it is *software engineering*. Clearly, this transition from computer- to software engineering could happen as a clean cut at the interfaces, i.e., the frames are sent and received by both parties and that's that. But this is not desirable: In a real-world scenario people work on projects in teams, and it would most probably be the case that there exists a qualified person for the iOS development, e.g., a software engineer. Typically, the computer engineer doesn't like objects while the software engineer doesn't like to work with primitive datatypes. Thus, it would be desirable to *split the difference* appropriately, such that the software engineer can make use of the complex concepts available in software engineering, while the computer engineer can make sure that his system is used according to his ideas. This can only be achieved by *collaboration*.

Transferring the Communication Framework to iOS

In this project, the first task towards the iOS application was to throw up the iOS counterpart of the Communication Framework. At the iOS side, the framework is practically responsible for the same tasks as it is for the gateway hardware: TCP and UDP communication management, receiving/sending of whole frames instead of stream-oriented communication (reading from *Start-of-Frame* to *End-of-Frame*) and (un)escaping of the message payload.

The first task, TCP and UDP communication management, can be handled by using the `CocoaAsyncSocket` libraries: The software engineer already knows how this is done and especially how this can be integrated into the interface, such as a view that allows to change connection settings, or error handling if no connection can be established. Computer engineers typically do not have to cope with human interaction as embedded systems most of the time run autonomously. For the extraction of frames out of a stream, the `CocoaAsyncSocket` already provides a method that allows to *read from data-to-data*. Thus, by reading from *End-of-Frame* to *End-of-Frame* full frames can be extracted out of the stream. What remains is the un/escaping of the frame payload. Figure 7.14 sketches the situation and what approach is used to resolve it.

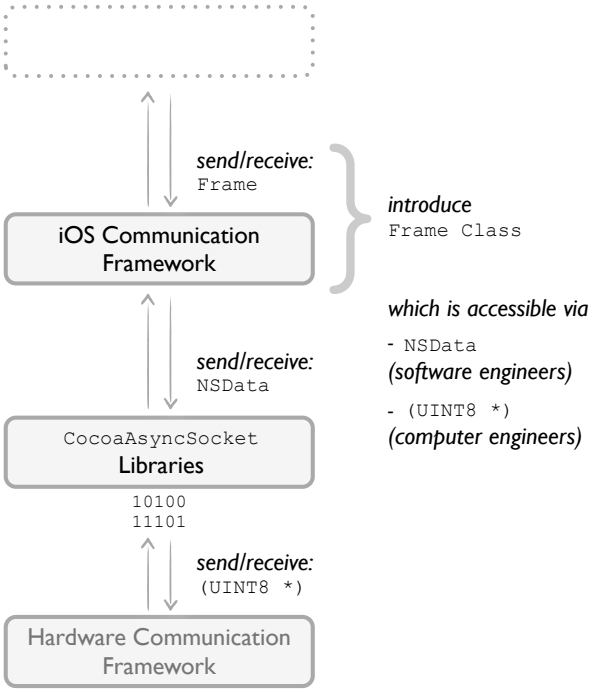


Figure 7.14: Transferring the Communication Framework to iOS.

The computer engineer has all the functions for un/escaping and frame handling already ready at hand - *but as C functions*. The frames have been defined using primitive datatypes and

this is what the gateway perceives. As soon as the data arrives at the iOS device, however, the `CocoaAsyncSocket` libraries provides them as `NSData` *objects*. Thus, either the methods for un/escaping have to be newly implemented, or the software engineer helps out in translating the *objects* to primitive datatypes such that the original functions can be adopted. This is where the fact that iOS uses *Objective-C* as programming language comes in very handy: Standard *C* functions can practically be just copied&pasted into any *Objective-C* project. What's done in this implementation is that a *Frame* class has been created that allows to access the payload by both, using primitive C-arrays or objects. This allows to directly apply the un/escaping methods onto the payload (those methods have been implemented as class methods). In order to send a frame, all that's left to do is to pass the payload *as an object* to the `CocoaAsyncSocket` libraries.

Splitting the difference: From primitive datatypes to data management

Clearly, un/escaping of a message payload could also be done in a purely object-oriented manner, such that the *Frame* class would not need to allow for a primitive access to the payload. The reason why there is still a possibility to access the data using a primitive C array, is because the computer engineer's work is not yet done: Consider the response message from Table 7.7 (page 105): This response is broadcast using UDP, thus on the iOS side those elven bytes are received just as they are, together with a single heading byte indicating that this message contains OBD data. The computer engineer already knows the whole communication scheme and the message contents - he has designed it. The software engineer, on the other side, would still have to dig into the ISO standards and the communication protocol to be able to use this data.

A desirable solution would therefore split the difference: The software engineer has the necessary insight into the possibilities provided by the iOS development framework, while the computer engineer knows how the gateway has to be handled. In this project, the computer engineer knows everything about how the data has to be extracted and what hardware states are to be considered, while the software engineers knows how the data management has to work. Figure 7.15 depicts the approach taken in this implementation. The basic concepts used for the implementation are the following:

CoreData [64] is an extremely powerful framework for handling data in the wider sense: It is able to handle everything from normal files, databases up to cloud-based data management. The difference between a database and CoreData is, that CoreData actually manages *objects*, not only data.

A good example is the usage of CoreData to populate table views, such as the one used for the live-data display (cf. Figure 7.18-b): The actual items of the list are linked to *objects*, not just retrieved from a database. The major difference is, that every other part of the application can manipulate this object and the table view will be automatically updated. Thus, CoreData provides access to *"living objects"*, not *"dead data"*. Clearly, the data

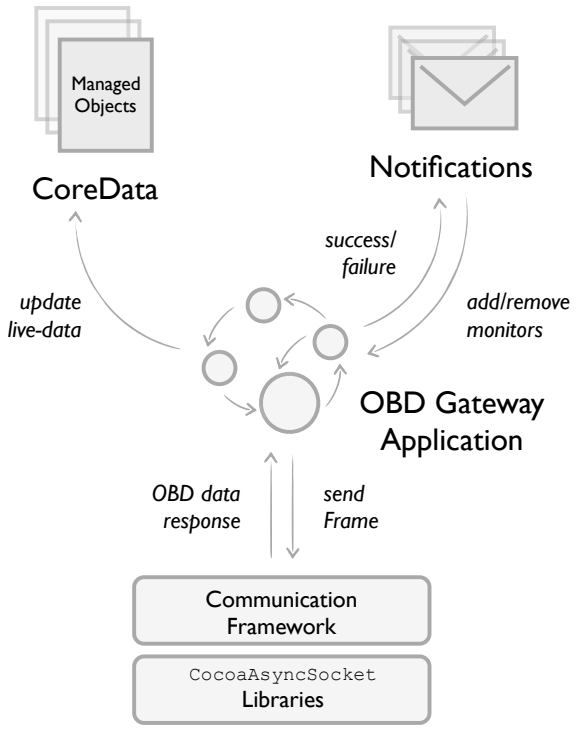


Figure 7.15: Sketch: from primitive datatypes to data management.

contained in those objects can be easily saved to files or databases. From a software engineering point of view it would therefore be preferable if all the OBD data could be obtained through the CoreData framework.

Notifications can be used to pass informations between classes, in this case between the view controllers of the different application tabs. This is a much more elegant solution than one class shared by all the other instances or a so called *singleton*[15].

Thus what's done is the following: The computer engineer "embeds" a standard *C* state-machine into the iOS project where he handles the communication with his hardware. The Communication Framework is used to receive and transmit *Frames* that can be accessed in a familiar manner, using standard C-arrays. The received data is extracted and passed to the CoreData framework, while notifications are used to inform the software about hardware changes and to add or remove monitors. Clearly, the software engineer is involved in this process, especially when connecting the state-machine to the iOS primitives. What remains is to build the iOS application on top of these familiar concepts of CoreData and Notifications.

[15] A singleton is basically the only instance of a class, i.e., there is no other instance of this class in the project, that can be globally accessed.

`Home`
allows to connect/ disconnect to the gateway and displays the hardware status of the gateway

`Live-Data`
list of currently monitored parameters, data is updated in real-time

`Dashboard`
a custom view using monitoring services

`Console`
logs basic events such as communication events or the adding/removal of monitors

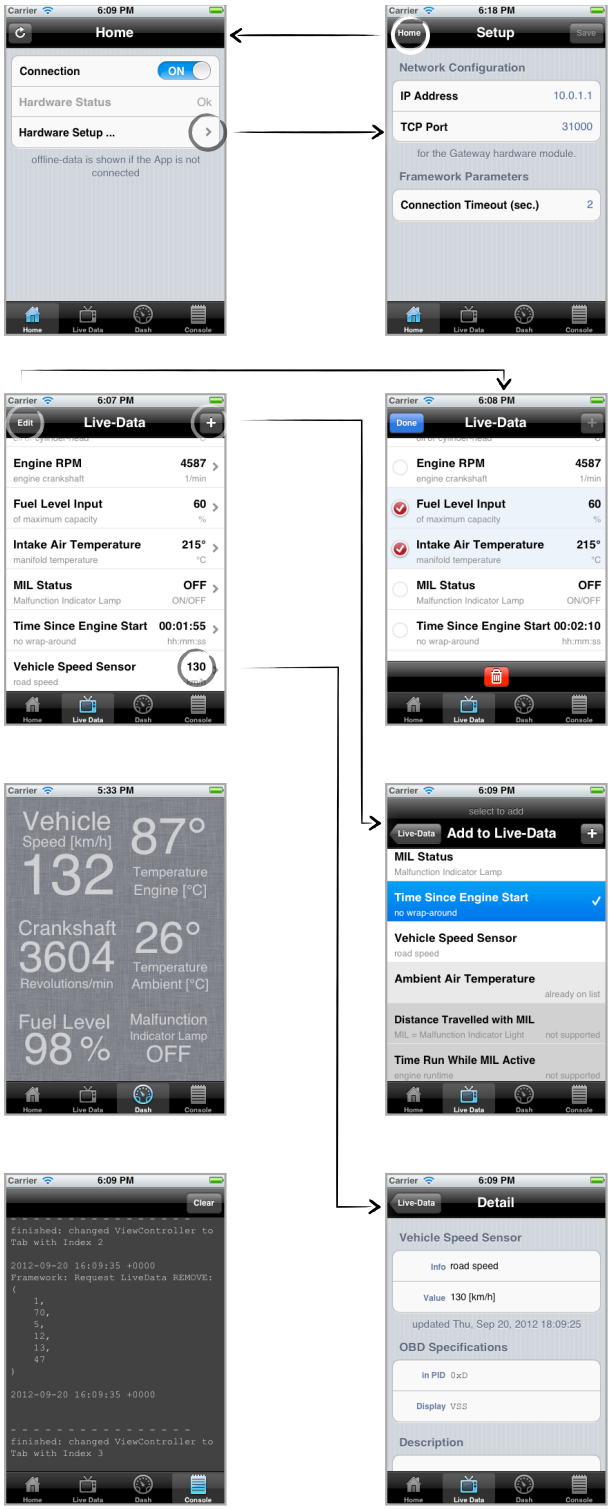


Figure 7.16: iOS application overview.

7.7.2 iOS application

Figure 7.16 shows the complete structure of the implemented iOS application. It is a tab-based application, i.e., by tapping on either of the symbols in the bar at the bottom of the screen, a different view can be selected. The screenshots on the right side in Figure 7.16 depict the views that can be accessed through certain actions, e.g., by clicking on the *"+"* button in the `Live-Data` tab, the view changes to a list-view where parameters can be selected to be added to the live-data list. Before describing each of the tabs separately, it is convenient to make a short remark about the *CoreData* and *Notification* frameworks mentioned just previously:

Basically, each of the screenshots depicted in Figure 7.16 is controlled by a so called *View-Controller*. Independent of those ViewControllers, the actual OBD application is working in the background, as depicted in Figure 7.15. It quietly updates the values obtained by monitoring and sends out notifications if, e.g., the hardware status changes. Thus all such a ViewController has to do to participate, is to subscribe to the notifications, make use of CoreData, and send notifications to add/remove monitors to the OBD application in the background. The concept may become clearer when looking at the actual application:

Tab 0: The home screen. The first tab is the one displayed when the App is first started. It is basically only used to connect to the hardware, i.e., to establish the connection over Wi-Fi. The actual hardware status of the gateway is displayed by the second item. As already discussed in Section 7.6.1 and depicted in Figure 7.8 (page 92), the hardware states are

- *Error*, if the gateway could not yet communicate with the vehicle,
- *Ok*, or `Online`, if on-board-diagnostic are currently running, and
- *Disconnected*, or `Offline`, if the connection to the vehicle has been lost.

Notice that `Off` is not included, as it is already indicated by the connection status: The App won't be able to connect to the hardware if it is powered off. The hardware- and connection states are obtained through *notifications*. CoreData is not used by this tab as there's no data to be displayed.

By tapping on the *Hardware Setup* item, the user gets to a view that allows to change the IP address and TCP port where the App should try to connect to.

Tab 1: Live-Data. The `Live-Data` tab is essentially a user interface to the monitoring functionality of the gateway: It allows to add and remove parameters from a list of currently monitored items. The items displayed in this list are all actual *objects* managed by CoreData - the same objects that are updated by the OBD Application running in the background. Thus, as soon as a frame containing OBD data has been processed, the OBD Application updates the corresponding objects which automatically causes the value in the live-data list to be updated as well.

Through *Notifications*, the application can issue requests to the OBD application running in the background to add or remove such monitors. E.g., as soon as the user switches to the `Live-Data` tab, the ViewController of this tab issues a request to the background application to add monitors for all the items in the list. If the user switches to another tab, e.g., the `Home` tab, it issues a request to remove them.

The buttons *"Edit"* and *"+"* at the top bar of the list allow to add or remove items, just as depicted by the screenshots indicated by the arrows. Parameters that are already on the list cannot be added a second time and are colored with a light gray. Parameters that are not supported by the vehicle are colored with a darker gray. Clearly, upon adding or removing parameters, new notifications have to be sent out.

The last view reachable by the live-data list is a detail-view, which shows additional information for the selected parameter, such as the last time the parameter has been updated, the PID-ID as defined by ISO 15031-5 together with the identifier that shall be used by external test equipments, and a short description.

Tab 2: A virtual Dashboard. The second tab is used as an example for what else those monitoring services could be used, which in this case is a `Virtual Dashboard`. The functionality is essentially the same as for the `Live-Data` tab, only with a fixed list of parameters: Upon switching to the tab, the ViewController requests to add this fixed list of monitors, upon exiting the controller requests a removal. The displayed data is again linked to the objects managed by CoreData and is thus updated as soon as new values are sent by the gateway.

Tab 3: Console. The third tab is a `Console` which allows to log the actions taken throughout the exection. Those actions include the requests to add and remove monitors, as well as connection-related information.

Staying responsive.

The last thing to discuss about the iOS application is the responsiveness of the application: In Section 7.6.1 it has been argued that the gateway should be allowed to shut-down as soon as the diagnostic port does no longer provide any current. This can happen quite often and could be perceived as bothersome by the user if not handled right by the iOS application.

(a) Alert *(b) Live-Data* *(c) Virtual Dashboard*

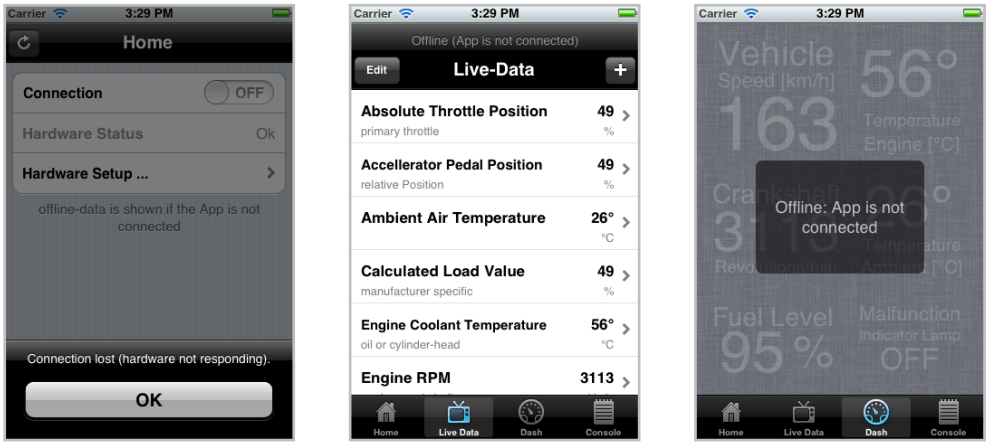


Figure 7.17: Screenshots of the iOS application while the gateway is offline.

The approach followed in this implementation is to basically allow the application to be used regardless of whether or not the hardware is connected. The only situation where currently[16] an alert is shown, is if the connection to the hardware is lost. Figure 7.17 depcits the situation for the first three tabs (the console is only used for logging): The alert shown upon a disconnect is visible as an overlay over the tab the user is currently at, which in the case of Figure 7.17-a is the `Home` tab.

If the gateway is not connected, the `Live-Data` shows a little prompt in the top-bar (cf. Figure 7.17-b). This little prompt embodies the only difference in the behavior of the tab to when the gateway is connected and running: The user can still scroll through the list, add and remove items or even check out the detail view. There's no reason not to allow this interaction. In fact, the user could assemble a list of interesting parameters while still begin offline. As soon as the App is connected to the gateway, the prompt is hidden and the parameters on this list are monitored. For the `Dashboard`, the situation is slightly different, as it does not allow any interaction whatsoever. In order to make it clear that the tab can currently not display any valid data, an overlay is shown which informs the user that the App is currently not connected to the

[16] It is not yet clear if this alert should be kept, i.e., whether or not it should be removed. This depends on feedback from actual users. Another option would be to allow an optional activation of the alert in the Settings page.

vehicle. The view, however, is not completely covered by this overlay, i.e., it does still allow to get an idea about the functionality of this tab.

Regarding the OBD hardware states discussed in Section 7.6.1 and depicted in Figure 7.8 (page 92), the App behaves the in the same way, just without an alert. Figure 7.18 illustrates the behavior if the gateway cannot communicate with the car, but has been able to before (it is *disconnected*). The `Home` tab simply displays the according hardware state as the second item (cf. Figure 7.17-a). The `Live-Tab` shows a prompt, just like when the App is not connected. The only difference is the message, which now indicates that the *"OBD hardware is offline"*, including the hint that this may be the case because the ignition may be off. The same message is shown in the overlay of the `Dashboard` tab.

(a) Home *(b) Live-Data* *(c) Virtual Dashboard*

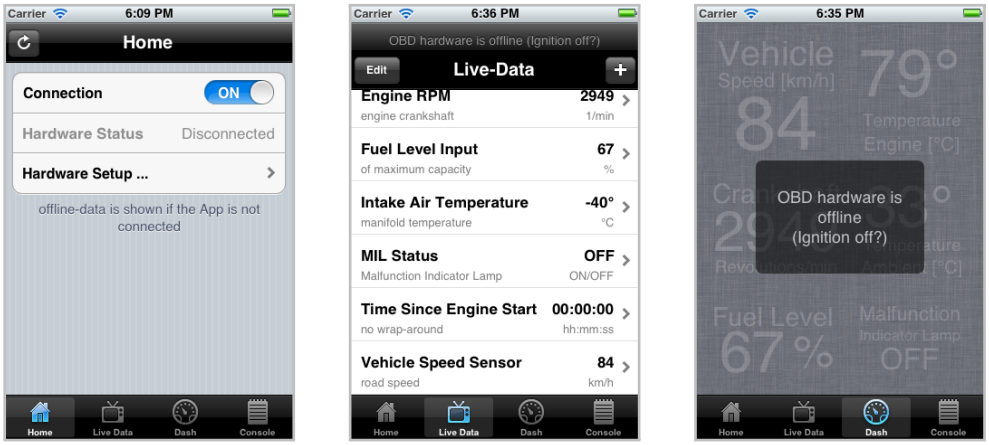


Figure 7.18: Screenshots of gateway hardware-state handling by the iOS application.

Round-up

Thus, all in all the iOS application remains responsive regardless of hardware- or connection states. State changes are handled as unobtrusive as possible, requiring minimal or no user interaction at all. With this iOS application, the otherwise rather technical application of OBD services has been made easily accessible to an every-day user. The App uses as many standard interface elements as possible: Only the `Dashboard` is an untypical sight for iOS users, but still uses only labels and a standard background.

7.8 Evaluation

To conclude this case study, it is necessary to verify that the initial requirements as listed in Section 7.2 have been met and that the system as a whole is functionally valid.

7.8.1 Functional validation: The Simulator

Throughout the development the hardware was only seldom connected to an actual vehicle: Most of the development has been done using very specific test projects, e.g., to test the OBD initialization procedure, the reception of multi-frames or the wireless communication. Only to verify that the functionality actually works in practice, the hardware has been connected to the diagnostic port. Especially at the end of the development phase a much more complex test environment was necessary, which should include the possibility to specifically change the diagnostic data that is simulated. The result is depicted in Figure 7.19.

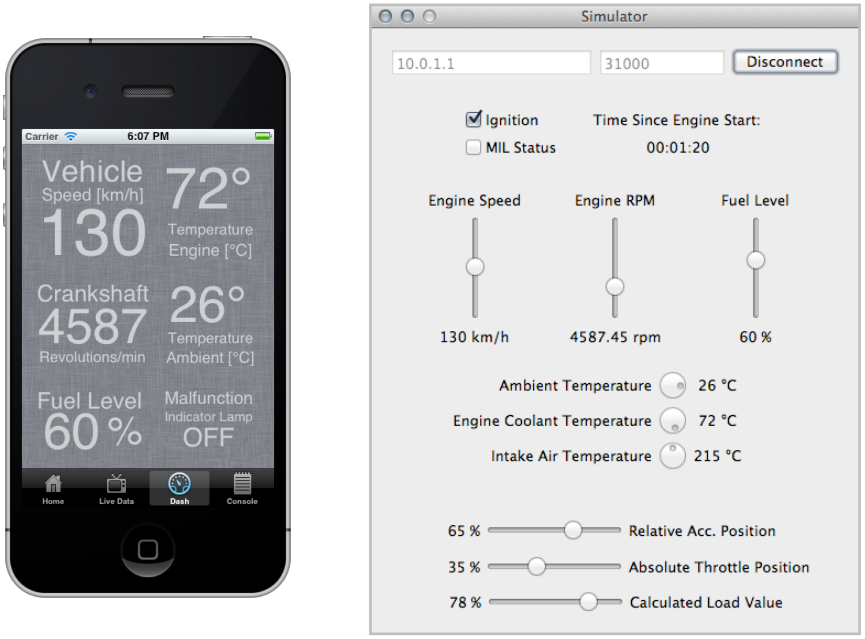


Figure 7.19: Simulator screenshot side-by-side with the iOS App running on the iOS Simulator.

The "Simulator" is essentially a system that consists of an application that runs on the gateway hardware and the corresponding OS X application which functions as a user interface to this program. It basically simulates the behavior expected by the vehicle using the second CAN interface of the hardware, i.e., the simulator sends and receives actual CAN frames according to the ISO standards. Figure 7.5 (page 79) depicts the situation: both CAN interfaces are interconnected. Thus, for the OBD application running on the gateway there is no difference between communicating with an actual car or with a simulator application running in parallel.

The fact that the whole iOS Communication Framework can be used 1:1 by an OS X application allowed the Simulator to be up and running in a matter of hours. The functionality was already provided by various test projects, it only had to be put together to an actual application. Removing the Simulator from the gateway firmware is simply done by removing a single `#define` statement. As apparent from Figure 7.19, the OS X Simulator allows to set the values for certain parameters as well as simulating the ignition lock (ignition on and off). The parameters are communicated to the Simulator application running on the gateway hardware, which saves the changes such that subsequent requests via the other CAN interfaces receive the new value as response. Notice that the values are updated continuously, i.e., clicking and dragging a slider or control knob effects in multiple value changes, which can also be perceived through the iOS application if the parameter is monitored.

7.8.2 Requirements verification

What remains to be done is to check whether or not all the requirements posed in Section 7.2 have actually been fulfilled by the implementation:

Functionality.

`Req.1:` The final result shall implement a chosen subset ($\neq \emptyset$) of the OBD services.
The gateway implements a subset of the services `0x01` and `0x09`.

`Req.2:` The consumer shall be able to execute the specified services using a consumer electronics device.
Consumer electronics can connect via wireless LAN (TCP/UDP), an iOS application has been implemented which allows to execute monitoring services

`Req.3:` The execution of the services shall comply to the on-board diagnostic standards.
The services have been implemented according to the ISO standards.

Security.

`Req.4:` The gateway hardware must by no means facilitate an attack on the vehicle via the diagnostic port, i.e., it must not amplify the attack surface of the vehicle.
The wireless interface is currently the only interface to the gateway and is protected by appropriate security protocols (WEP/WPA2-PSK).

`Req.5:` The data that is exchanged through the gateway shall be protected against eavesdropping using appropriate security measures.
Cf. requirement `Req.4`: data encryption is provided by WEP/WPA2-PSK.

`Req.6:` The gateway shall respect the security requirements that are specified by the on-board diagnostic standards (cf. requirement 3).
The application does not violate any security requirements, as the OBD bundle currently does not provide any for the services `0x01` and `0x09`.

Connectivity.

`Req.7:` The chosen interface for communication between the hardware and the consumer electronic shall be supported by the majority of commonly used devices.
Wi-Fi is supported by virtually all of the devices.

`Req.8:` The chosen communication scheme shall be portable to other platforms with reasonable effort; wireless connections shall be preferred.
TCP/UDP communication is a common standard and the used communication scheme is fairly simple to implement.

Safety and Usability.

`Req.9:` The security measures shall have minimal to no impact on the usability of the product, i.e., they shall not require special knowledge.
WEP/WPA-PSK are commonly used protocols and requires minimal setup.

`Req.10:` The installation and/or removal of the gateway hardware should require minimal effort.
The hardware is Plug&Play.

`Req.11:` The removal of the hardware shall be optional and not necessary. This requirement comes from the fact that the diagnostic port is not necessarily easily accessible and thus being required to remove the hardware is tedious.
The hardware does not have to be removed, it is powered by the diagnostic port and requires no maintenance.

`Req.12:` The usage of the gateway hardware must not pose any further safety risks, i.e., the gateway operations must not have any impact on safety aspects. Notice that this requirement depends upon the implemented service: the gateway hardware must be able to determine if executing a service in the current vehicle state is appropriate. E.g. a request to disable the headlights while driving under insufficient lighting conditions must not be executed.
The implemented services do only retrieve diagnostic data and can be executed at any vehicle state.

`Req.13:` The gateway hardware should require minimal maintenance (e.g. battery exchange).
Cf. discussion of `Req.10` and `Req.11`.

`Req.14:` The usage of the hardware itself should require minimal user interaction, especially regarding physical interaction with the hardware.
The hardware is Plug&Play and may be plugged in all the time. The iOS application connects to the hardware with a single tap.

`Req.15:` Security measures should by no means lead to a denial of service (e.g. failed authentication attempts).
The hardware supports an unlimited number of connection attempts.

Constraints.

`Req.16:` A final hardware product should be affordable, i.e., the costs for individual components should be held as low as possible.
The components needed for a final product can be reduced to the PIC microcontroller, the CAN- and Wi-Fi transceivers, the FDTI chip and the two voltage regulators. All of them are available at an acceptable price.

`Req.17:` A possible final hardware product shall have minimal space requirements (cf. requirement 10: easy installation) such that the space provided by the diagnostic port is optimally usable.
The final product can be expected to be slightly larger than the TEXA OBD log as depicted in Figure 7.4 (page 70).

`Req.18:` The choice of the development tools, hardware components and target consumer electronics have to allow high implementation speed in order to support a reasonable deadline for this master's thesis.
Both, the Microchip and the Xcode frameworks are freely available, offer a plethora of libraries and examples and allowed a fast implementation speed.

Thus, the implementation indeed fulfills all the requirements.

7.8.3 Numbers

Throughout this thesis the inclusion of actual source-code has been intentionally avoided. The major concepts and the implementation details given in this section provide more than enough details to find one's way in the projects or to reconstruct the concepts. To give some idea about the size of the actual projects, that is, the gateway firmware and the iOS client (the Simulator is excluded here), the following Table 7.10 provides an overview over the number of lines-of-code[17] excluding external sources such as the Microchip TCP/IP Stack or the `CocoaAsyncSocket` libraries.

Gateway firmware

Language	files	blank	comment	code
C	13	1174	1680	3537
C/C++ Header	14	354	1144	624
Sum	27	1528	2824	4161

iOS client application

Language	files	blank	comment	code
Objective C	21	1420	1236	3588
C/C++ Header	20	166	271	246
Sum	41	1586	1507	3834

Table 7.10: Code-size breakdown of the case study

In summary this results in

- lines of pure code: $7,995$
- lines of code and comments (excluding blanks): $12,326$
- lines of source code: $15,440$

Projects with a code-size of about $15K$ lines of code can be considered quite complex. For this project, the complexity has been neatly packed into an easy-to-use, always responsive iOS client application that is comprehensible by a common user.

[17] The tool that has been used is CLOC (Count Lines of Code) `http://cloc.sourceforge.net`

CHAPTER 8

Summary and Outlook

This master's thesis has provided a fundamental insight into gateways in conjunction with consumer devices and embedded systems on behalf of a case study of a gateway between the diagnostic port of a car and consumer devices. The implemented gateway uses *Wi-Fi* as interface to the consumer device, which is basically supported by *every* device on the market. Security has been included in the system design process right from the start, such that the result is sufficiently secure and the usability did not suffer. Furthermore, porting the current iOS/OS X target application to other platforms, such as Android, is extremely simple.

Clearly, the resulting *"product"* is not something fundamentally new. It was not the target of the thesis to introduce some new technology, but to *do things the right way*. When comparing it to the OpenXC platform, the solution presented in this thesis has as its major advantages wireless communication and portability: The OpenXC platform does primary rely on wired communication over the serial interface. This restricts current applications to Android devices - and obviously requires wiring. Furthermore, Apple's MFi program makes porting this project to their platform rather cumbersome.

Regarding the current functionality, there is yet room for improvement, e.g., adding further encryption on the broadcasted data or the implementation of a secure software update. The case study of this thesis, however, is not directly intended to be a full-featured product, but rather a first step in the research on the integration of embedded systems into the internet and the interfacing of embedded systems using consumer devices. This goal has evidently been achieved, as the current communication with the consumer device can be easily adopted for generic embedded systems. The development platform that has been designed throughout this case study provides the necessary means for future experiments.

The *SmartThings* platform is definitely a paramount example of what should be considered for future research in this area. The concept is great and offers huge possibilities and it will be something to look forward- and in-to.

APPENDIX A

Acronyms

ACROSS ARTEMIS CROSS-Domain Architecture

AES Advanced Encryption Standard

ARTEMIS Advanced Research and Technology for Embedded Intelligence and Systems

ASIC Application Specific Integrated Circuit

CAN Controller Area Network

CARB California Air Resources Board

CRC Cyclic Redundancy Check

DIS Draft International Standard

DTC Diagnostic Trouble Code

ECU Electronic Control Unit

ESP Electronic Stability Program

FDTI Future Technology Devices International Ltd.

GPRS General Packet Radio Service

GPS Global Positioning System

GSM Global System for Mobile Communications

HMI Human Machine Interface

HUD Head Up Display

I^{2}C Inter Integrated Circuit

iAP iPod Accessory Protocol

IC Integrated Circuit

IEEE Institute of Electrical and Electronics Engineers

IMSI International Mobile Subscriber Identity

ISM Industrial, Scientific and Medical (Radio Band)

ISO International Standards Organization

JTAG Joint Test Action Group

KWP Key Word Protocol

LAN Local Area Network

LIN Local Interconnect Network

LTE Long Term Evolution

MAC Message Authentication Code

MIL Malfunction Indicator Lamp

MMC Multimedia Card

MOST Media Oriented Systems Transport

NDA Non Disclosure Agreement

OBD On Board Diagnostics

PCB Printed Circuit Board

PID Parameter IDentifier

PIN Personal Identification Number

PWM Puls Width Modulation

RMII Reduced Media Independent Interface

RPM Revolutions Per Minute

SAE Society of Automotive Engineers

SD Service Digital (Memory Card)

SID Service IDentifier

SIG Special Interest Group (Bluetooth)

SIM Subscriber Identity Module

SoC System on Chip

SOT Small Outline Transistor

SPI Serial Peripheral Interface

SSP Secure Simple Pairing

TCP Transmission Control Protocol

TPM Trusted Platform Module

TTA Time Triggered Architecture

TTCAN Time Triggered Controller Area Network

UART Universal Asynchronous Receiver Transmitter

UDP User Datagram Protocol

UDS Unified Diagnostic Services

UMTS Universal Mobile Telecommunications System

VIN Vehicle Identification Number

VPWM Variable Puls Width Modulation

WEP Wired Equivalent Privacy

WPA Wi-Fi Protected Access

WPS Wi-Fi Protected Setup

WPAN Wireless Private Area Network

WWDC World Wide Developer Conference

WWH-OBD World Wide Harmonized OBD

Bibliography

Literature

[1] W.A. Arbaugh. Wireless security is different. *Computer*, 36(8):99 – 101, aug. 2003.

[2] A. Bogdanov, D. Carluccio, A. Weimerskirch, T. Wollinger, and escrypt GmbH. Embedded security solutions for automotive applications. In J. Valldorf and W.Gessner (eds.), editors, *Advanced Microsystems for Automotive Applications.*, pages 177–191. Springer Verlag, 2007.

[3] Colin A. Boyd and Anish Mathuria. *Protocols for Authentication and Key Establishment*. Information Security and Cryptography. Springer, 2003.

[4] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, Berkeley, CA, USA, 2011. USENIX Association.

[5] Christos Douligeris and Dimitrios Nikolaou Serpanos. *Network Security: Current Status and Future Directions*. IEEE Press, Piscataway, NJ, USA, onl edition, 2007.

[6] M.-C. Gaudel, V. Issarny, C. Jones, H. Kopetz, E. Marsden, N. Moffat, M. Paulitsch, D. Powell, B. Randell, A. Romanovsky, R. Stroud, and F. Taiani. Final version of the dsos conceptual model (chapter 3). Technical report, December 2002.

[7] A. Groll and C. Ruland. Secure and authentic communication on existing in-vehicle networks. In *Intelligent Vehicles Symposium, 2009 IEEE*, pages 1093 –1097, june 2009.

[8] Tobias Hoppe, Stefan Kiltz, and Jana Dittmann. Security threats to automotive can networks — practical examples and selected short-term countermeasures. In *Proceedings of the 27th international conference on Computer Safety, Reliability, and Security*, SAFECOMP '08, pages 235–248, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] H. Kopetz. Elementary versus composite interfaces in distributed real-time systems. In *Proceedings of the The Fourth International Symposium on Autonomous Decentralized Systems*, ISADS '99, Washington, DC, USA, 1999. IEEE Computer Society.

[10] Hermann Kopetz and Günther Bauer. The time-triggered architecture. In *Proceedings of the IEEE*, pages 112–126. IEEE Computer Society, 2003.

[11] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 447–462, Washington, DC, USA, 2010. IEEE Computer Society.

[12] A. Krepil and Inova Semiconductors GmbH. Automotive 1gbit/s link opens new century in car hmi and driver assistance systems. In J. Valldorf and W.Gessner (eds.), editors, *Advanced Microsystems for Automotive Applications.*, pages 193–199. Springer Verlag, 2007.

[13] Sandeep Kumar and Thomas Wollinger. Fundamentals of asymmetric cryptography. In C. Paar K. Lemke and M. Wolf (eds.), editors, *Embedded Security in Cars: Securing Current and Future Automotive IT Applications.*, pages 145–166. Springer Verlag, 2005.

[14] Sandeep Kumar and Thomas Wollinger. Fundamentals of symmetric cryptography. In C. Paar K. Lemke and M. Wolf (eds.), editors, *Embedded Security in Cars: Securing Current and Future Automotive IT Applications.*, pages 125–144. Springer Verlag, 2005.

[15] Kerstin Lemke. Embedded security: Physical protection against tampering atacks. In C. Paar K. Lemke and M. Wolf (eds.), editors, *Embedded Security in Cars: Securing Current and Future Automotive IT Applications.*, pages 208–217. Springer Verlag, 2005.

[16] R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz. From a federated to an integrated automotive architecture. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(7):956 –965, july 2009.

[17] Jonathan Oxer and Hugh Blemings. *Practical Arduino: Cool Projects for Open Source Hardware*. Apress, Berkely, CA, USA, 2009.

[18] Christoph Paar. Embedded it security in automotive application - an emerging area. In C. Paar K. Lemke and M. Wolf (eds.), editors, *Embedded Security in Cars: Securing Current and Future Automotive IT Applications.*, pages 3–13. Springer Verlag, 2005.

[19] Srivaths Ravi, Anand Raghunathan, and Srimat Chakradhar. Tamper resistance mechanisms for secure, embedded systems. In *Proceedings of the 17th International Conference on VLSI Design*, VLSID '04, pages 605–, Washington, DC, USA, 2004. IEEE Computer Society.

[20] Ahmad-Reza Sadeghi, Christian Stüble, and Kerstin Lemke-Rust. Anti-theft protection: Electronic immobilizers. In C. Paar K. Lemke and M. Wolf (eds.), editors, *Embedded Security in Cars: Securing Current and Future Automotive IT Applications.*, pages 51–67. Springer Verlag, 2005.

[21] Frank Stajano and Ross J. Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. In *Proceedings of the 7th International Workshop on Security Protocols*, pages 172–194, London, UK, UK, 2000. Springer-Verlag.

[22] Armin Wasicek. Embedded security at a glance: Security concepts for embedded systems. Technical Report 182-1/2007/70, Vienna University Of Technology - Department of Computer Engineering, 2007.

[23] Marko Wolf, André Weimerskirch, and Christof Paar. Secure in-vehicle communication. In C. Paar K. Lemke and M. Wolf (eds.), editors, *Embedded Security in Cars: Securing Current and Future Automotive IT Applications.*, pages 96–107. Springer Verlag, 2005.

[24] Marko Wolf, André Weimerskirch, and Thomas J. Wollinger. State of the art: Embedding security in vehicles. *EURASIP J. Emb. Sys.*, 2007, 2007.

[25] Thomas Wollinger, Jorge Guajardo, and Christof Paar. Cryptography in embedded systems: An overview. In *Proceedings of the Embedded World 2003 Exhibition and Conference*, pages 18 – 20, 2003.

[26] Werner Zimmermann and Ralf Schmidgall. *Bussysteme in der Fahrzeugtechnik: Protokolle, Standards und Softwarearchitektur (ATZ/MTZ-Fachbuch) (German Edition)*. Vieweg+Teubner Verlag, 4, akt. und erw. aufl. 2011 edition.

Datasheets and Application Notes

[27] FDTI Chip FT232RL USB to serial UART interface. http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf. Accessed: 2012-09-13.

[28] Microchip AN1044: Data Encryption Routines for PIC24, dsPIC and PIC32 Devices. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en027644. Accessed: 2012-09-13.

[29] Microchip AN1373: Using PIC32 MCUs to Develop GSM/GPRS/GPS Solutions Application Note. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en553690. Accessed: 2012-09-15.

[30] Microchip MCP1700 Low Dropout Positive Voltage Regulator (3V3). http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en010642. Accessed: 2012-09-13.

[31] Microchip MRF24WB0MA Wi-Fi Transceiver Module. http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en548014. Accessed: 2012-09-13.

[32] Microchip MRF24WG0MA Wi-Fi Transceiver Module. http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en559196. Accessed: 2012-09-13.

[33] Microchip PIC32MX795F512H Microcontroller. http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en545655. Accessed: 2012-09-13.

[34] Microchip PICKit3 In-Circuit Debugger. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en538340. Accessed: 2012-09-13.

[35] Smart Mixed-Signal Connectivity (SMCS) LAN8720 Ethernet Transceiver. http://www.smsc.com/index.php?tid=149&pid=59. Accessed: 2012-09-15.

[36] Texas Instruments LM2940CT Low Dropout Regulator (5V). http://www.ti.com/lit/ds/snvs769h/snvs769h.pdf. Accessed: 2012-09-13.

[37] Texas Instruments SN65HVD230 3.3V CAN Transceiver with Standby Mode. http://www.ti.com/product/sn65hvd230. Accessed: 2012-09-13.

[38] Texas Instruments UA7805 Fixed Positive Voltage Regulator (5V). http://www.ti.com/product/ua7805. Accessed: 2012-09-14.

[39] u-blox LEON-G100, G200 GSM/GPRS 2.5G modules. http://www.u-blox.com/en/wireless-modules/gsm-gprs-modules/leon-gsm-module-family.html. Accessed: 2012-09-15.

[40] u-blox NEO-6 series (GPS modules). http://www.u-blox.com/en/gps-modules/pvt-modules/neo-6-family.html. Accessed: 2012-09-15.

Relevant Standards

[41] ISO 15031-1:2010; Road vehicles – Communication between vehicle and external equipment for emissions-related diagnostics – Part 1: General information and use case definition, 2010. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=51828, Accessed: 2012-10-02.

[42] ISO 15031-2:2010; Road vehicles – Communication between vehicle and external equipment for emissions-related diagnostics – Part 2: Guidance on terms, definitions, abbreviations and acronyms, 2010. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=50815, Accessed: 2012-10-02.

[43] ISO 15031-3:2004; Road vehicles – Communication between vehicle and external equipment for emissions-related diagnostics – Part 3: Diagnostic connector and related electrical circuits, specification and use, 2004. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=29021, Accessed: 2012-10-02.

[44] ISO 15031-4:2005; Road vehicles – Communication between vehicle and external equipment for emissions-related diagnostics – Part 4: External test equipment, specification and use, 2004. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=40087, Accessed: 2012-10-02.

[45] ISO 15031-5:2011; Road vehicles – Communication between vehicle and external equipment for emissions-related diagnostics – Part 5: Emissions-related diagnostic services, 2011. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=50816, Accessed: 2012-10-02.

[46] ISO 15031-6:2010; Road vehicles – Communication between vehicle and external equipment for emissions-related diagnostics – Part 6: Diagnostic trouble code definitions, 2010. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=50817, Accessed: 2012-10-02.

[47] ISO 15765-2:2011; Road vehicles – Diagnostic communication over Controller Area Network (DoCAN) – Part 2: Transport protocol and network layer services, 2011. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=46045, Accessed: 2012-10-02.

[48] ISO 15765-4:2011; Road vehicles – Diagnostic communication over Controller Area Network (DoCAN) – Part 4: Requirements for emissions-related systems, 2011. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=46045, Accessed: 2012-10-02.

Miscellaneous

[49] ACROSS Project Homepage. http://www.across-project.eu. Accessed: 2012-09-25.

[50] Apple Developer Homepage. https://developer.apple.com. Accessed: 2012-09-20.

[51] Arduino Open-Source Electronics Prototyping Platform. http://www.arduino.cc. Accessed: 2012-09-13.

[52] Bluetooth Special Interest Group (SIG) Member Website. http://www.bluetooth.org/. Accessed: 2012-09-28.

[53] CadSoft EAGLE PCB Design Software. http://www.cadsoftusa.com/eagle-pcb-design-software/?language=en. Accessed: 2012-09-13.

[54] California Environmental Protection Agency (CARB): On-Board Diagnostic II (OBD II) Systems - Fact Sheet / FAQs. http://www.arb.ca.gov/msprog/obdprog/obdfaq.htm. Accessed: 2012-10-02.

[55] CocoaAsyncSocket, easy-to-use and powerful asynchronous socket libraries for Mac and iOS. https://github.com/robbiehanson/CocoaAsyncSocket. Accessed: 2012-09-20.

[56] Digilent Cerebot MX7cK Microcontroller Development Board. http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,396,986&Prod=CEREBOT-MX7CK. Accessed: 2012-09-13.

[57] Digilent chipKIT Max32 Prototyping Platform. http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,892,894&Prod=CHIPKIT-MAX32. Accessed: 2012-09-13.

[58] Digilent Network Shield for chipKIT Max32 Prototyping Platform. http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,892,942&Prod=CHIPKIT-NETWORK-SHIELD. Accessed: 2012-09-13.

[59] Digilent WiFi Shield for chipKIT Max32 Prototyping Platform. http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,892,1037&Prod=CHIPKIT-WIFI-SHIELD. Accessed: 2012-09-13.

[60] ELM Electronics "OBD Interpreter" Integrated Circuits. http://elmelectronics.com/obdic.html. Accessed: 2012-09-13.

[61] FDTI Chip Homepage. http://www.ftdichip.com. Accessed: 2012-10-04.

[62] GENESYS Project Homepage. http://www.genesys-platform.eu. Accessed: 2012-09-25.

[63] IEEE Standards Association. http://standards.ieee.org/. Accessed: 2012-09-28.

[64] iOS Data Management. https://developer.apple.com/technologies/ios/data-management.html. Accessed: 2012-09-20.

[65] ISO - International Organization for Standardization. http://www.iso.org. Accessed: 2012-10-02.

[66] Kickstarter funding platform for creative projects. http://www.kickstarter.com. Accessed: 2012-09-23.

[67] Microchip Encryption Routines for PIC24, dsPIC, and PIC32. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2680&dDocName=en537998. Accessed: 2012-09-13.

[68] Microchip PIC Microcontrollers. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2551. Accessed: 2012-09-13.

[69] Microchip PIC32 Microcontroller Families. http://ww1.microchip.com/downloads/en/DeviceDoc/39904L.pdf. Accessed: 2012-09-13.

[70] Microchip PICtail Board for SD& MMC. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en537238. Accessed: 2012-09-13.

[71] Microchip TCP/IP Stack for PIC18, PIC24, dsPIC & PIC32. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2680&dDocName=en537041. Accessed: 2012-09-13.

[72] Microchip Technology Inc. http://www.microchip.com. Accessed: 2012-09-13.

[73] MPLAB X Integrated Development Environment (IDE). http://www.microchip.com/pagehandler/en-us/family/mplabx/. Accessed: 2012-09-13.

[74] OpenXC Platform. http://openxcplatform.com. Accessed: 2012-09-20.

[75] Reduced Media Independent Interface (RMII) Specification. http://www.national.com/assets/en/other/rmii_1_2.pdf. Accessed: 2012-09-15.

[76] Roving Networks Apple iOS Support. http://www.rovingnetworks.com/Apple_iOS_Support. Accessed: 2012-09-23.

[77] SmartThings. http://www.smartthings.com. Accessed: 2012-09-20.

[78] Society of Automotive Engineers (SAE); WWH-OBD presentation. http://www.sae.org/events/training/symposia/obd/presentations/2005renaudin.pdf. Accessed: 2012-10-02.

[79] Stefan Viehböck. Brute forcing wi-fi protected setup, when poor design meets poor implementation. http://sviehb.files.wordpress.com/2011/12/viehboeck_wps.pdf/. Accessed: 2012-09-28.

[80] TEXA OBD Log Diagnostic Device. http://www.texa.com/prodotti_dett.asp?sez=dett&id=22. Accessed: 2012-09-15.

[81] TEXA S.p.A. http://www.texa.com. Accessed: 2012-09-23.

[82] United Nations, Proposal for new draft Global Technical Regulation (GTR) - Uniform provisions concerning the technical requirements for on-board-diagnostic systems (OBD) for road vehicles. http://oica.net/wp-content/uploads/060316-wwh-obd-final-rev1-2.pdf. Accessed: 2012-10-02.

[83] U.S. Environmental Protection Agency (EPA): Control of Air Pollution From New Motor Vehicles and New Motor Vehicle Engines; Modification of Federal On-Board Diagnostic Regulations for: Light-Duty Vehicles, Light-Duty Trucks, Medium Duty Passenger Vehicles, Complete Heavy Duty Vehicles and Engines Intended for Use in Heavy Duty Vehicles Weighing 14,000 Pounds GVWR or Less. http://www.epa.gov/fedrgstr/EPA-AIR/2005/December/Day-20/a23669.htm. Accessed: 2012-10-02.

[84] Vector Informatik GmbH, CANalzyer 8.0. http://www.vector.com/vi_canalyzer_en.html. Accessed: 2012-09-15.

[85] Wi-Fi Alliance. http://www.wi-fi.org/. Accessed: 2012-09-28.