Aufgabenstellung

zur

Abschlussarbeit
im Studiengang Master Automotive Software Engineering

für

Herrn Shivraj Gajanan Bathe
geb. am 13. Februar 1988 in Indien

zum Thema

Secure Communication in a Multi-OS Environment

Betreuer/ Prüfer: Prof. Dr. Wolfram Hardt
Ausgabedatum: 06.08.2015
Abgabedatum: 13.01.2016
Tag der Abgabe:

Unterschrift: __________________________
Prof. Dr. F. Hamker
Vorsitzender des Prüfungsausschusses
Secure Communication in a Multi OS Environment

Master Thesis

for
the fulfillment of the academic degree
M.Sc. in Automotive Software Engineering

Faculty of Computer Science

Department of Computer Engineering

Submitted by: Shivraj Bathe, Date of birth: 13.02.1988

Supervisor: Prof. Dr. W. Hardt
Dip. Inf. Bodo Bauer

January, 2016
Abstract

Current trend in automotive industry is moving towards adopting the multicore microcontrollers in Electronic Control Units (ECUs). Multicore microcontrollers give an opportunity to run a number of separated and dedicated operating systems on a single ECU. When two heterogeneous operating systems run in parallel on a multicore environment, the inter OS communication between these operating systems become the key factor in the overall performance. The inter OS communication based on shared memory is studied in this thesis work. In a setup where two operating systems namely EB Autocore OS which is based on AUTomotive Open System Architecture standard and Android are considered. Android being the gateway to the internet and due to its open nature and the increased connectivity features of a connected car, many attack surfaces are introduced to the system. As safety and security go hand in hand, the security aspects of the communication channel are taken into account. A portable prototype for multi OS communication based on shared memory communication with security considerations is developed as a plugin for EB tresos Studio.

Keywords: Multi OS Communication, Shared memory communication, Secure channel, AUTOSAR, Android, Linux
Acknowledgements

I would like to take this opportunity to thank Bodo Bauer, my supervisor at Elektrobit, for giving me the opportunity to work on this topic. I am grateful to Prof. Wolfram Hardt, my supervisor at TU Chemnitz, for accepting the concept of this topic and motivating me by providing guidance which helped me throughout my thesis work.

I also want to thank Martin Böhner for his inputs and views regarding security aspect of the master thesis. Sebastian Rietzsch and Daniel Haap have helped me with the practical issues and I have discussed a lot of ideas and suggestions with them. Daniel Haap has provided me with the necessary equipment and knowledge to be able to complete this project. I want to thank him specially for helping me setup the development environment in the beginning of the project. Sebastian Rietzsch and Simon Dürr have helped me in structuring my report and provided important reviews. I hereby thank Elektrobit Automotive GmbH for its open and relaxing work environment with all the essential software and hardware support.
# Contents

1 Introduction 1
   1.1 Motivation ............................................. 1
   1.2 Thesis Objective ....................................... 3
   1.3 Outline of the Report .................................. 4

2 State of the art 5

3 Fundamentals 9
   3.1 AUTOSAR .................................................. 9
      3.1.1 Basics ................................................. 10
      3.1.2 Multicore Configuration .............................. 11
      3.1.3 EB Tresos Studio ..................................... 13
   3.2 Android ................................................... 14
      3.2.1 Android Architecture ............................... 14
      3.2.2 Device Driver development ........................ 15
   3.3 Security ................................................... 17

4 Concept 21
   4.1 Multi OS Communication Concept ........................ 22
      4.1.1 Producer consumer model ......................... 24
      4.1.2 Message structure .................................. 25
      4.1.3 Synchronization ..................................... 26
   4.2 Security ................................................... 27
      4.2.1 Use Case and Threat Scenario ..................... 27
      4.2.2 Solution 1: Cryptography ......................... 29
      4.2.3 Solution 2: Bandwidth and message content monitoring ..... 32

5 Implementation 34
   5.1 Development Hardware ................................. 34
## List of Figures

1.1 Problem Definition .......................................................... 3

3.1 AUTOSAR Layered Software Architecture [14] .......................... 10
3.2 AUTOSAR Multicore Configuration ...................................... 12
3.3 Android architecture on a mobile stack [25] .......................... 14
3.4 Symmetric Cryptography [26] .............................................. 17
3.5 Asymmetric Cryptography [26] ........................................... 19
3.6 Diffie Hellman Key Exchange [27] ....................................... 20

4.1 MultiOsCom Setup .............................................................. 22
4.2 Producer consumer model .................................................. 24
4.3 Ring Buffer ................................................................. 25
4.4 Message buffer ............................................................. 26
4.5 Use case - Over the air update ......................................... 27
4.6 Key exchange protocol ...................................................... 30
4.7 Bandwidth and message content monitoring .......................... 32

5.1 Freescale SABRE for Automotive Infotainment Based on the i.MX 6 Series ...................................................... 35
5.2 Shared interface class diagram ............................................ 36
5.3 Interaction diagram .......................................................... 39
5.4 Behavior diagram: send function ....................................... 41
5.5 Behavior diagram: receive function ................................... 43
5.6 Plugin Structure .............................................................. 45
5.7 Configuration to code ....................................................... 46
5.8 MultiOsCom configuration GUI .......................................... 48
5.9 MultiOsCom message configuration GUI ............................ 49
5.10 MultiOsCom signal configuration GUI ................................. 49
5.11 XML code for checking cross dependency ............................ 50
5.12 Datamodel dependencies and validation ............................. 51
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Test setup</td>
<td>55</td>
</tr>
<tr>
<td>6.2</td>
<td>CAN interface for USB</td>
<td>55</td>
</tr>
<tr>
<td>6.3</td>
<td>MultiOsCom Linux test application</td>
<td>57</td>
</tr>
<tr>
<td>6.4</td>
<td>PCAN View GUI</td>
<td>58</td>
</tr>
<tr>
<td>6.5</td>
<td>Effect of misaligned pointers</td>
<td>59</td>
</tr>
<tr>
<td>6.6</td>
<td>Data transfer performance</td>
<td>60</td>
</tr>
<tr>
<td>6.7</td>
<td>Performance results with AES (128 bit key)</td>
<td>61</td>
</tr>
<tr>
<td>7.1</td>
<td>Future research</td>
<td>64</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Shared interface configuration parameters ........................................ 37
5.2 Usage of structure variables .............................................................. 38
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>File operations data structure</td>
<td>15</td>
</tr>
<tr>
<td>5.1</td>
<td>Buffer struct definition</td>
<td>37</td>
</tr>
<tr>
<td>5.2</td>
<td>Callback function</td>
<td>44</td>
</tr>
<tr>
<td>5.3</td>
<td>MultiOsCom Signals XDM Data model</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>Code generator Input</td>
<td>52</td>
</tr>
<tr>
<td>5.5</td>
<td>Code generator output</td>
<td>52</td>
</tr>
<tr>
<td>6.1</td>
<td>Test application for Linux device driver</td>
<td>56</td>
</tr>
<tr>
<td>6.2</td>
<td>Test application for AUTOSAR</td>
<td>57</td>
</tr>
</tbody>
</table>
Introduction

This chapter gives the introduction to the domain of the thesis which is mainly the development of a software solution for communication between two operating systems on a multicore microcontroller. The need as well as the motivation for work in this field is highlighted in subsequent sections. The structure of the entire report is also mentioned.

1.1 Motivation

Electronic systems are becoming smaller in size and faster in processing speed. These innovations in electronics and information technologies are adapted in automobile industry for better functionality and user experiences. For example the closed loop control systems which once were implemented on electrical circuits, are now completely implemented on microcontrollers. The ever increasing demand for better functionality, safety and security has increased the complexity in such systems. Nowadays embedded software frameworks are employed to handle these complexities. For the automotive industry the AUTomotive Open System Architecture (AUTOSAR) standard (briefly described in section 3.1) defines the specifications for such a system.

An operating system is developed for performing multiple tasks and handle different types of applications, but there can be certain limitations on applications due to the fact that the OS was developed for a particular domain. For example a general purpose operating system may not provide the real time constraints of real-time operating system and a real-time operating system may not provide all features and complexities of a non-real-time operating system. With advances in multicore microcontrollers there is now a possibility to run different operating systems on a single microcontroller such that a
combination of similar or different operating system can be used to develop the desired functionality. Multicore microcontrollers provide more computational power and benefit of independent instances of applications, but the development of such systems is difficult as compared to single core applications. Furthermore, if applications which use data from different cores running heterogeneous operating systems, the inter-process communication between the applications becomes a key factor in the performance. Automotive manufacturers along with chip manufacturers and software giants like Google are moving towards integrating a popular mobile operating system called Android in modern cars [1]. Android has interactive Human Machine Interface (HMI), Internet connectivity and Media applications. This operating system can run along with AUTOSAR on a multicore microcontroller such that it can access the car’s various sensors and actuators. This situation can give car driver a better access to cars function for example internet based monitoring of cars mileage or a personal android app developed by user for air conditioning inside car. For such cases an efficient communication mechanism between Android and AUTOSAR is required. As safety goes hand in hand with security, such a secure channel needs to be safeguarded from hackers.

To solve the problem mentioned above, this thesis work is related to developing an effective and secure inter-OS communication channel between AUTOSAR and Android operating systems running on a multicore Electronic Control Unit ECU.
1.2 Thesis Objective

This thesis work was carried out at Car Infrastructure (CIS) Department, at Elektrobit Automotive GmbH, Erlangen and the problem definitions are aligned with current research projects within the company. The main goals are defined as follows:

- Effective and secure communication in a Multi-OS-Environment within a single large ECU running different operating systems (AUTOSAR and Android)
- Data exchange between different OS instances in a secure manner
- Performance considerations regarding en-/decryption

One such example is a multicore ECU is having 4 cores, from which AUTOSAR OS runs on 2 cores and Android OS runs on the remaining 2 cores. Android OS can provide interactive graphical interface to the user, whereas AUTOSAR is the real time operating system controlling the entire functionality of the car. In this case, there can be a bidirectional communication between AUTOSAR and Android. Therefore, an effective and secure communication mechanism is needed between these operating systems. Considering the above scenario, the aim is to firstly study conceptual analysis of secure communication mechanisms in other domains. Analysis should be done on existing communication mechanism towards security. A threat model should be defined and possible solutions towards security should be outlined. A prototype should be implemented for secure communication mechanism between two operating systems Android and "EB Autocore OS" (an operating system by Elektrobit Automotive GmbH complying AUTOSAR standards). Additional challenge is to integrate the solution in EB tresos Studio.
1.3 Outline of the Report

This chapter introduces to the domain of the topic and the need for improvement in the current state of the art. Chapter 2 gives the current state of the art gathered from the literature review of the current research in this area. Chapter 3 briefly introduces the reader to the basic concepts which are used further in this thesis. The two operating systems Android and AUTOSAR are introduced with respect to multicore implementations and inter process communications provided by these systems. Also, in Chapter 3 a short overview is given about software development tools such as "EB tresos Studio" and Linux device driver development. In Chapter 4, various communication mechanisms shortlisted from a literature review are presented in detail, after an evaluation of these mechanisms a final concept is derived. Also in Chapter 4, firstly a threat scenario is defined and the security aspect is covered by analyzing current possible security mechanisms for defined threat scenarios. Chapter 5 gives complete details of the prototype implementation. The data structures, flowcharts, behavior diagrams of individual functions are discussed in this chapter. Since this thesis work is largely the development of a basic software module and these modules are represented as plugins, the implementation detail of this plugin such as XML Data model, code generator and GUI is also described in this chapter. Results and conclusion is described in Chapter 6, which also covers the test setup. A conclusion is drawn from the results and literature review and possible future work is suggested in this chapter. Finally, Chapter 7 lists out the references used throughout this report and Chapter 8 is the Appendix with code snippets and other details related to this work which can be useful for the reader.
This chapter presents the results of literature review regarding current technologies in multicore domain, inter-processor communication in Android and AUTOSAR operating systems. Also, the current state of the art in secure communication mechanisms using cryptography which could be extended to communication between heterogeneous OS operating systems (e.g. Android and AUTOSAR) sharing a single ECU is discussed.

Implementation of resource demanding systems such as Infotainment needs fast computation power, and traditionally for fast computation, the clock frequency has to be increased. But there is an upper limit to this frequency because the heat produced in the CPU is proportional to the clock frequency. As suggested by [2] this situation can be improved by using multicore CPUs. Moghaddam and Safaei [3] have shown that with parallel and distributed computing, modern CPUs can achieve up to 74% speedup. Most of the general purpose multicore microprocessors have homogeneous cores, i.e. the Instruction Set Architecture and performance of all the cores is same. There are also microcontrollers which exploit the possibility of heterogeneous architectures, for example Cell BE architecture [4], but this thesis work will consider only homogeneous multicore architectures.
Advent of Multicore Electronic Control Units:
With great computing power as in multicore systems comes great responsibility for development of the system. The responsibility is the synchronization between programs running on separate cores, which cause additional overhead with relation to inter-core communication. There are several methods of this synchronization, such as spinlocks, which are used for managing resources between the cores [3]. These multicore CPUs are connected together with various on chip inter-connect like bus interconnect, mesh inter-connect, ring and crossbar [5]. A global address space is present on the microprocessor which allows programmers to effectively write parallel programs. This global memory sharing takes place on CPU interconnects using a shared memory bus, but to avoid processors from flooding the global memory bus, local caches are introduced for local processor operations. Cache coherent memory system is needed for reliable operation when multiple cores with local cache read and write global variables [5].

Inter Core Communication:
Considering the architecture of a microcontroller, a Global Interrupt Controller is responsible for managing and handling local and global interrupts. There are also Inter Core Interrupts which can pave the way for synchronization of communication between the two cores. But in current setup such ICIs are mainly used in OS startups activities, and synchronization between cache and MMU. Currently in automotive industry, onboard communication protocols like SPI or I2C are used when two instances of heterogeneous operating systems are running on a multiprocessor ECU. There is also possibility to use existing field bus protocols like CAN or FlexRay to communicate between different operating systems on same ECU. CAN bus can provide up to 125 kBit/s and FlexRay can provide up to 10 MBit/s bandwidth [6]. This is not a feasible solution since there will be additional configuration and synchronization overhead for this communication will require additional CAN controllers on the hardware. In current hardware setup both the cores are on the same processor, therefore the communication between the cores using fieldbus protocols will not be the fastest or the safest solution. This work by Youseff and others [7] evaluates virtualization and paravirtualization for inter-core communication. But this work is relevant where hardware can support such a hypervisor. However the hardware support for hypervisor comes at a cost and it is not always feasible to implement such a solution.

Shared memory:
The memory shared between the cores can be used as a communication channel. There are mainly two communication models when using shared memory for communication, namely message passing and shared memory. In message passing communication model a FIFO message queue is used for sharing data. Messages are written to and read from
this message queue. Whereas, in shared memory communication model instead of writing it to a queue, memory address of the common variables is shared between concurrent programs. Klaiber and Levy [8] have compared architectural mechanisms of shared memory and message passing communication models. Generally, message passing communication model is implemented on software, on the hardware Inter Core Interrupts or polling can be used to facilitate synchronization. There are architectures which provide hardware support for message passing and inter core communication, for example this work by Poletti and others [9] proposes a scratchpad memory and a portable implementation of producer consumer communication model for inter core communications. Direct Memory Access is used to shed the communication load from the processors. Lee and others [10], present message passing mechanisms based on dedicated instructions on MIT M-Machine. Based on simulations, Chandra S. and others [11] studied message passing and shared memory programs and depending on the timing behavior of the program shown that message passing model did not have a clear advantage rather additional synchronization overhead caused it to perform poorly compared to the shared memory model.

Inter core communication in AUTOSAR and Linux:
There are implementations and specifications of inter-core communication for Android and AUTOSAR systems like for example the Remote procedure message protocol (rpmsg) is a Linux IPC mechanism. Rpmsg is a virtualization based messaging bus that allows kernel drivers to communicate with remote processors available on the system [12]. However, this method is not specified and implemented for AUTOSAR therefore it cannot be directly used. Moghaddam and others [3] have presented a technique for Inter OS communication between two operating systems running in parallel on a multicore system, but this implementation is restricted as both the operating systems need to be AUTOSAR. A similar implementation by Killig S. [13] gives a promising approach but it does not completely cover the need for message based communication and security aspects. Therefore this work by Killig S. [13] is taken into consideration but there is still a lot of room for improvement. As the Inter OS Communicator specified by AUTOSAR [14] are useful when all the OS instances are AUTOSAR complaint. In this thesis work MCAPI standards [15] taken into consideration, these standards however do not specify the hardware implementation of the communication rather an overall topology and structure of the messages.

Security:
The public key cryptosystem RSA presented by Rivest R., Shamir L. and Adleman L. [16] describes a method to share secrets using public key cryptography and Miller and others [17] describe the Advanced Encryption Standard (AES) standard. A study on comparison of various data encryption algorithms [18] shows that AES performs better in terms
of speed and key lengths than other available encryption algorithms. RSA provides flexibility in sharing the keys, but it is slower due to many mathematical operations. Another novel method for key exchange is presented by Diffie and Hellman [19] which is considered in the concept design of the security mechanism. A mechanism called SENSS by Zhang and others [20] describes a program for a secure communication on shared communication bus with high bandwidth but a hardware implementation of cryptographic engine is required. Therefore for better portability of the code, other standard security mechanisms are considered. Currently in industry TLS [21] and SSL [22] protocols are used as a standard in various secure communication schemes. Since SSL protocol deals with tunnels and password based authentications adapting it to a shared memory communication would take additional efforts, therefore in this thesis TLS protocol is considered as possible a mechanism for security.

**Summary:**
Considering all the related works mentioned above, this thesis work is different in a sense that it also takes in consideration, not only heterogeneous OS shared memory communication but also the security requirements. Specially in a use case scenario where an attacker is able to freely develop an application on the Android platform and deploy it.
This chapter shall provide a brief introduction to fundamentals related to the concepts and tools used in this thesis work. An overview of AUTOSAR and Android operating systems in regards to the multicore implementation described in section 3.1. In 3.3 fundamentals of security mechanisms are described.

### 3.1 AUTOSAR

The Automotive Open System Architecture (AUTOSAR) standard was introduced to help OEM’s reuse embedded software. AUTOSAR supports software sharing by providing architecture, infrastructure, methodology, and the basic software. AUTOSAR simplifies the combination of software by different providers. The baseline for such a scenario is the nonexclusive usage of AUTOSAR basic software provided by the host ECU. Software can be shared by vehicle platforms of the same OEM. In addition, standardized application interfaces support the exchangeability of software between suppliers. AUTOSAR specifies the interfaces of Software Components (SWCs) and often used applications from all automotive domains regarding their syntax and semantics. [23]

By now there is a broad variety of basic software implementations from different vendors for many hardware platforms on the market, so that the basic software and corresponding configuration tools can be regarded as off-the-shelf products. The development of the application software components with the definition of the internal behavior, coding, and implementation are independent from hardware and can be done separately for each application level software component (SWC). The functionality on the hardware is achieved through layers of lower level Basic software components (BSWs), which need
to be configured and integrated as per the requirements of the application level software components (SWCs).

3.1.1 Basics

- Layered Software Architecture

![AUTOSAR Layered Software Architecture](image)

Figure 3.1: AUTOSAR Layered Software Architecture [14]

The **Microcontroller Abstraction Layer** is the lowest software layer of the Basic Software. It contains internal drivers, which are software modules with direct access to the microcontroller internal peripherals and memory mapped microcontroller external devices. The **ECU Abstraction Layer** interfaces the drivers of the Microcontroller Abstraction Layer. It also contains drivers for external devices. It offers an API for access to peripherals and devices regardless of their location (microcontroller internal/external) and their connection to the microcontroller (port pins, type of interface). **Complex Device Drivers** provide functionality which is not specified by Autosar standards. However if an SWC needs to access a complex device driver it must access it through RTE. In this thesis work MultiOsCom module is implemented as a complex device driver. The **Service Layer** is the highest layer of the Basic Software which also applies for its relevance for the application software: while access to I/O signals is covered by the ECU Abstraction Layer, the Services Layer offers:

- Operating system functionality
- Vehicle network communication /management
- Memory services (NVRAM management)
- Diagnostic Services (UDS, OBD)
- Mode management
The **RTE** is a layer providing communication services to the application software (AUTOSAR Software Components and/or AUTOSAR Sensor/Actuator components). Above the RTE the software architecture style changes from "layered" to "component style". The AUTOSAR Software Components communicate with other components (inter and/or intra ECU) and/or services via the RTE. The **Application Layer** is a layer providing application software (AUTOSAR Software Components and/or AUTOSAR Sensor/Actuator components).

- **Interface** Application interfaces are defined while defining Software components in Application Layer. There are various standardized interfaces like Sender/Receiver and Client Server. These interface types mainly define the type of the communication and data types, furthermore the direction of the communication is defined by ports there can be two type of ports Provide Port and Required port. Combination of the above interfaces and ports works hand in hand to implement the dataflow between software components.

- **Methodology** AUTOSAR is relatively large scale development work which may be carried between various teams and possibly different vendors and companies. This standard also defines the standardized data exchange formats and methodology for generating the ECU software. The definitions and automatic generation of such data exchange formats is specified in this [14].

### 3.1.2 Multicore Configuration

Starting from AUTOSAR 4.0 the specification of multicore architecture is introduced. Given below are few important developments regarding multicore configuration. Given below are few enabling features for multicore configuration in AUTOSAR.

**Modes:**
A multicore OS can have two modes, Lockstep or parallel processing mode. In lockstep mode two identical OS instances run on separate cores on single ECU, and an independent comparator compares the results of the executions and generates a trap in case of discrepancy. This feature is used in safety applications. In parallel processing mode, as shown in diagram below, one of the OS serves as the master and the others as slave OS kernels. Since the RTE is partitioned, Os calls like ActivateTask() can work directly across cores.

**Multi-core startup /shutdown functions:**
New API functions are introduced to start particular core StartCore() and StartOs().
3.1. AUTOSAR

In the beginning core 0 should is started then before starting the OS on core 0 rest of the cores and OS instances are initialized.

**Loadable Entities:**
An application group (containing timers, counters, alarms and tasks) can be assigned to a particular core [24]. This feature allows developers to use individual cores as and when needed. In this thesis work, AUTOSAR is using core number 2 and 3. Core 3 is used for printing debug messages to console by assigning the debug task to an application and further configuring the parameter OsApplicationCoreAssignment to 3 for this application.

**Spinlocks:**
A mechanism to lock shared resources between cores. It is a busy waiting mechanism. When a task needs a shared resource it requests a spinlock through an atomic function to test and set the spin lock. If the resource is used by other tasks of current or other cores it is returned with spinlock value 1. It waits and polls the spinlock till the lock is on the shared resource released (spinlock value becomes 0). By this mechanism critical sections of the program and private memory areas can be protected during execution of functions. Spinlock works on shared memory locks because the priority ceiling protocol used in single core cannot be extended to multicore processors. [24]
AUTOSAR IOC:
Inter OS Communication is an OS service which provides communication which can be accessed by configured nodes beyond OS Application boundaries. Communication is possible between different CPU cores and memory partitions. In EB Tresos studio configuration can be done OS configuration. Both queued and un-queued communication channels are possible in IOC. But this service is available only when both OS instances are AUTOSAR complaint.

3.1.3 EB Tresos Studio
EB tresos Studio is an Eclipse based tool developed by Elektrobit Automotive for Configuration and generation tool for ECU standard software. This tool follows the AUTOSAR methodology but can be used for more than just configuring BSW modules. The functionality of this tool can be can be extended by free usage of code generator engines data model access and corresponding GUI support. Each basic software module can be configured and code can be generated according to the configuration. Configuration errors or problems of dependencies between various Software Components can be detected before code generation. A number of problems due to configuration errors can be avoided. If the configuration does not have any errors EB tresos Studio automatically generates the c code for the basic software (BSW) Stack. After generating the entire project with .c and .h files, with corresponding makefiles it can be compiled with an external compiler.[6]

AUTOSAR standards may not provide an exact solution to a particular problem in an automotive system, for example adding support of a complex sensor/actuator. Such functionality can be implemented in the Complex Device Drivers (CDD). An additional integration step has to be followed to bridge between customer needs and standards. Integration can also be done when a desired feature is not (yet) available or does not work in the expected way. Possible solutions can be to check release notes for deviation lists and studying AUTOSAR specifications. If desired functionality is not specified or deviates from the requirements it can be implemented by writing a module. This module can provide a .c and .h files for a particular function. Basic software modules are developed as plugins. EB Tresos Studio provides the possibility of extending the functionality by creating new plugins with the help of java extension APIs. Details of plugin are given in implementation chapter.
3.2 Android

Development of a device driver as the MultiOsCom implementation on the Android side was one of the major tasks in this thesis work. This section gives introduction to Android and Linux device driver development. Android, which is developed by Google and Open handset alliance (OHA), is a Linux based mobile operating system which includes its GUI, web browser and end user applications which are user friendly and easy to download from Google play store. Android was primarily developed for touchscreen smartphones but are further improved to work with television, cars and wrist watches providing advanced user interface. It is popular as it provides a low cost, readily built foundation and modifiable operating system which can be customized according to advanced user needs. It gives a freedom to develop user defined device specifications and hardware drivers.

3.2.1 Android Architecture

Android is a customizable operating system which can be tailored according to the user needs due to its open nature and easy access to the Android SDK. Android architecture consists of software stacks which are divided into four horizontals and five sections namely applications, application framework, libraries, android runtime and Linux kernel. Each element of the below architecture is tightly assembled to lay out an integrated optimized development framework and development environment.

![Android Architecture Diagram](image)

Figure 3.3: Android architecture on a mobile stack [25]

- **Linux Kernel:**
As shown in figure 3.3, out of the four layers of Android OS architecture, the bottom layer of the architecture consists of a Linux kernel. It is an abstraction layer between the hardware and the upper software stacks of Android. Linux kernel provides low level core services of memory, process management, network interfaces and device drivers for the hardware. Android uses a version of Linux kernel with special additional features to develop device drivers.

### 3.2.2 Device Driver development

There are basically two types of devices: block devices and character devices. Block devices possess a buffer system for storing the requests. Out of all the requests read, the best request will be answered to first following the next important ones. This technique is applicable for storage devices where read and write of elements is easy. The input and output in such devices is in the form of blocks whose size can vary according to the different devices. Unlike block devices, character devices do not have any buffering method to read the requests. So it responds to the request as quickly as possible by reading the byte data in a character by character stream fashion. Additionally, it does not operate on a fixed block size but uses minimum number of bytes for the communication. Character device have defined functions which read the requests from the device.

**Typical structure of file_operations data structure for character device:**

```c
static struct file_operations fops =
{
    .open = dev_open,
    .read = dev_read,
    .write = dev_write,
    .release = dev_release,
};
```

The file_operations data structure has pointers to the functions defined to handle the requests. The structure of file_operations entries, which is defined in 'linux/fs.h', gives the address of the function to be called for performing the requested operation on the device. There are few operations which are not implemented by the device and hence their definitions should be set to NULL. As the device driver is developed in kernel space, only kernel space library functions can be used. For example the user space function printf() is not available, instead a kernel level function printk() must be used.
3. FUNDAMENTALS

3.2. Android

Reading from device:
Following code specifies the function prototype declaration for device read operation.

static ssize_t dev_read(struct file *filep, char *buffer, size_t len, loff_t *offset);

The inputs to the function are the pointer to the character input (*buffer) and the length of the input (len). Apart from this standardized declaration, the actual implementation of the read operation is left entirely to the developer.

Writing to device:
Similar to device read operation, the device read operation also has a standard function declaration as follows:

static ssize_t dev_write(struct file *filep, const char *buffer, size_t len, loff_t *offset)

Desired functionality can be implemented in the write function. Kernel space programs are mapped to a virtual memory address, by accessing this mapped address, the device write operation can write to RAM memory. The character device is not suitable to implement polling operations. If the read or write operations are cyclic, it should be implemented by either through a user space application or through another kernel space module using hardware timers and interrupts.
3.3 Security

Security can be ensured by means of cryptosystems and firewalls. Following descriptions give an overview of the important crypto algorithms.

**Cryptography:** Cryptography is a field which was introduced to ensure security. Security as explained above aims to protect or prevent security breaches. Cryptographic algorithms form a major part to accomplish this goal. Cryptography aims at achieving security for the system by encrypting the data during transmission of messages hiding its original identity. If a message is passed from one source to another, it is encrypted or converted in a form which is not readable by any third party attacker. When received at the destination, it is converted into its original form by decryption and the message communication is finished without any security breach of the information being transferred. When it comes to encryption, questions regarding generation of security key, cryptographic algorithm and their storage arise. To figure out ways, two important mechanisms are designed namely symmetric encryption and asymmetric encryption, which are described in below subsections.

**Symmetric:**
Symmetric encryption deals with single key encryption method where the same key is used to encrypt the plain text into a cipher text. Encryption algorithm encrypts the plain text data with a secret key. At the receiving end, the same key is used to decrypt the cipher text to recover the plain text. It is also termed as conventional encryption or single key encryption. Of course an encryption and decryption algorithm is used in conjugation with the secret key to complete the security mechanism.

![Symmetric Cryptography](image)

Figure 3.4: Symmetric Cryptography [26]

There are two main requirements for operation of symmetric encryption method:

1. **Keep the encryption algorithm strong enough:** The algorithm is known to
the sender and the opponent. Even if the opponent possesses the plain text and n number of cipher texts, he should not be able to figure out the secret key or decrypt the cipher text himself by breaking the encryption/decryption algorithm.

2. Secure key exchange: The sender and receiver should possess the secret encryption key in a much secured manner not being able to be intruded by anyone. If the key and algorithm is handy, it is very easy to read the communication. Hence utmost care should be taken.

Symmetric algorithm keeps only its algorithm as public and its key as a secret which makes it feasible for its usage. This makes it relatively inexpensive as the storage and privacy effort is much less and thus makes the algorithm less complex to process. DES (Data Encryption Standard) is the most widely used symmetric cypher which is now replaced by AES (Advanced Encryption Standard) [17]. AES is widely used by US government for protection of top secret information. It is implemented in hardware and software throughout the world for encryption of highly sensitive data. AES has three block ciphers: AES-128, AES-192 and AES-256 corresponding to its respective key lengths 128 bits, 192 bits and 256 bits. Its uses a 128 bit block of plain text which is depicted as a 4 x 4 square matrix of bytes. The process contains N rounds which depend on the key length: 128-bit keys have 10 rounds, 192-bit keys have 12 rounds and 14 rounds for 256-bit keys. Each round consists of combination of substitution, transposition and permutation techniques to manipulate the plain text and finally convert into cipher text. Major drawback of symmetric encryption is the protection of the secret key which results in encrypting the secret key and storing it. This will lead in an unending chain of dependencies to ensure secrecy of the key. Once the key is found, communication can easily be breached. Cryptography not only depends on the number of keys used but also depends on the type of methods used to transform plain text to cipher text. Traditionally, two methods are used, namely substitution (where each element in plain text is mapped to another element) and transposition (where all elements are rearranged) ensuring no loss of data. Encryption of the plain text data is also an important dimension of cryptography. One of the two techniques: block cipher or stream cipher are used to encrypt the plain text data. Block cipher takes a chunk or block of data of equal size and encrypts them. Whereas stream cipher technique encrypts continuous data stream one bit or one byte at a time.
Asymmetric encryption (also called as public key encryption) method is the one which involves two different secret keys during encryption and decryption respectively. Encryption algorithm uses a public key to encrypt or scramble the data. However, decryption algorithm uses a private key to decrypt or unscramble the original data. The public key is known to all users while encryption but the private key has to be generated by each participant. The public and private key pair makes the intruder difficult to disrupt the process. When compared to symmetric encryption, asymmetric encryption has increased calculations making it difficult to use for short data communications but proves secure for large and safety critical systems. Asymmetric encryption algorithms are based on mathematical functions rather than substitution and permutation. Applications of this cryptography include encryption/decryption, digital signature and key exchange. Public key cryptography is primarily based on Fermat’s theorem and Euler’s theorem. Out of many algorithms, RSA (Rivest-Shamir-Adleman) [16] is the widely used and accepted general purpose algorithm which fulfills all the requirements of public key cryptography. RSA algorithm works on the block cipher principle where the plain text and cipher text are integers (between 0 to n-1). Plain text are encrypted in blocks whose size is equal to or less then \( \log_2(n) + 1 \). Encryption and decryption are performed by following equations:

\[
C \text{ (cipher text block)} = M_e \mod n \\
M \text{ (plain text block)} = C_d \mod n = M_{ed} \mod n,
\]

where \( n \) is known to sender and receiver both.

Here, Public key \( PU = e, n \)

Private key \( PR = d, n \)
Key Exchange methods:

Key exchange protocols are a must to execute cryptography to enable data communication. These protocols enable secure exchange of secret keys to help encryption of messages. RSA and Diffie Hellman are two known candidate key exchange methods widely used. RSA has been already described in the above asymmetric encryption sub section. Diffie Hellman [19] key exchange method overview can be seen in the figure below:

![Diffie Hellman Key Exchange](image)

Figure 3.6: Diffie Hellman Key Exchange [27]

Where, X is private value, Y is public value q and $\alpha$ are global public values and K is the key.

User A sends the public value to the User B and gets in return public key from User B. Based on these values, private key K can be calculated using the global public values and decryption algorithm.

Summary:

This thesis work deals with a variety of software development areas, ranging from AUTOSAR basic software development to Linux device driver development with an overview of security. This chapter introduced the reader to fundamental and lays a path for developing the concept of this thesis.
The results of literature survey in Section 2 suggest the need of a communication mechanism between two heterogeneous operating systems sharing a single ECU with security. This chapter describes the concept which shall fulfill following requirements: i. Inter OS communication between AUTOSAR and Linux using shared memory. ii. Secure exchange of data. iii. Ability to configure the module by defining number of messages, data size and address of shared memory buffer.

Highlights of this chapter are:

**Design Overview**: following a top down approach, Section 4.1 gives the overall setup of the desired communication channel.

**Communication mechanism**: Section 4.1.1 gives the implementation details of communication channel on shared memory

**Security**: Section 4.2 shall present the security aspects. The approach is to first define a use case and consider various threat scenario for this use case. For these threat considerations a security mechanism is presented.

The the final prototype resulting from this concept will be referred to as **MultiOsCom** (Multi OS Communication) module in the next sections.
4.1 Multi OS Communication Concept

The figure 4.1 shows a setup where Linux and AUTOSAR OS run on cores 0/1 and 2/3 respectively. The configuration of such a multicore system is already explained in the Fundamentals chapter of this thesis.

A ring buffer is implemented on the shared memory between two OS instances. This feature implements a non-blocking communication between sender and receiver. Non-blocking type of communication is used here because blocking communication will need additional interlocks and will slow down the communication. In this case, the latest entry in the buffer is considered the most recent. Therefore, a free running ring buffer is used. When the rates of read and write operations between sender and receiver does not cope with each other’s speed, some data might be lost in this communication method. As the shared RAM memory is limited in size, as suggested by Lee P. and others [28] a ring buffer by far is an efficient solution towards a lock free communication. The responsibility of matching the rate of read and write operations is left to the user and it will depend on how often user invokes send and receive functions.
As described in the Chapter 4, a program can run on Linux platform in user space or kernel space. The difference between these two modes is the division of system memory. Kernel space modules have greater access to memory and system hardware than user space applications. Since Android OS uses Linux as its base for accessing hardware, the MultiOsCom module is developed as a device driver in the Linux kernel space. There are various types of device drivers in Linux kernel space for example char drivers, block drivers, network drivers, etc. On the AUTOSAR side however there is no support for complex drivers like block drivers. Therefore the MultiOsCom module is developed as a character device.

A character device is an interface which provides the basic operations like open, close, read and write. Since MultiOsCom is an interface between two operating systems and apart from initialization, send and receive all other internals are hidden from user applications. To access this char device, a test application is created in the user space. Based on the user input from console, this test application implements basic function calls of read and write operations to the MultiOsCom character device. On the AUTOSAR side, MultiOsCom is developed as a Complex Device Driver (CDD) and is invoked by an OS Task. This OS task can be accessed by the user application for sending data.

The data shared is packed in messages which have unique ID number, and each message has its own set of signals. The length of the individual signals can be defined at the time of configuration. The security mechanism is implemented inside the MultiOsCom modules and users need not configure them. This implementation is configured by a GUI in EB tresos Studio and depending on the configuration .c/.h files are automatically generated.
4.1.1 Producer consumer model

![Producer consumer model diagram]

Figure 4.2: Producer consumer model

Producer consumer model is a mechanism for communicating between two nodes using shared resource. In such a case producer produces a character on the memory and consumer detects and reads it. In this case, there is a set of producer and consumer for each OS instance.

As seen in the figure 4.1.1, there are 2 producers and 2 consumers. Each OS instance has a separate ring buffer for write operation. Read operation is performed on the shared buffer of opposite OS instance. The ring buffer is a free running buffer i.e. there is no blocking if the consumer reads the message. The buffer size is bounded with maximum length of N bytes. This length and start location is configurable at the time of code generation. There are 2 constraints to this buffer:

1. Initially write operation should start from position 0 and continue till N-1th consecutive write operations. On reaching N-1th position, for next write operation it should start again from initial position on 0th buffer position.

2. Consumer should not read buffer if there is no data available.
4. Concept

4.1. Multi OS Communication Concept

Figure 4.3: Ring Buffer

This concept seems easy and effective but in practical there are a few problems related to producer consumer model. The problems relevant to this thesis are memory interleaving and/or buffer overflow. Let us look into these problems briefly:

- **Problem 1: Memory Interleaving:** If both producer and consumer try to access shared memory buffer at once, the sync between read and write operations can be lost, leading to false sharing of data. Therefore, at a particular time, only producer or consumer should access the shared memory region achieving mutual exclusion.

- **Problem 2: Buffer overflow:** If the producer produces more than what the consumer can consume, this can create too much data for the buffer to store. But in this implementation a free running ring buffer is used. In this ring buffer, buffer overflow will occur every time then buffer is full. However it is intended by design, whenever the buffer reaches the end, the transmit index is reset to zero, which causes the next message to be overwritten on previous message. The responsibility of synchronizing read and write is dependent on polling rate. This polling rate has to be configured by the user.

4.1.2 Message structure

The message structure has fixed header starting with a predefined and fixed "magic number" to declare that the sender/receiver is initialized at this location. Transmit and receive counters indicate count of messages sent or received by this buffer. Signal start address gives the address of the first message in the buffer. Depending on the Tx counter and maximum length of a message, the sender and the receive buffer calculates the address of the next message. The message buffer is contagious asMemcpy operations are used to copy blocks memory. MSG_ID_1 has SIG1, SIG2, SIG3, SIG4 of 1 byte each, whereas MSG_ID_2 and MSG_ID_3 are smaller than MSG_ID_1. Still the address space is allocated according to max
message length. It is evident that some memory is not used at all. But it is necessary for memory alignment which is described as below:

**Memory alignment:** Microcontrollers can read addresses which are multiples of their word size. Word size is defined by architecture. Generally, a 32 bit processor will have 4 byte word size and 64 bit processor will have 8 byte word size. That means it can read memory addresses which are multiple of 4 bytes word size (for example 0xFF04, 0xFF08, 0xFF0C, etc.) If the address is not in this order, it can still be read. But depending on the microcontroller architecture it might need multiple read, rotate and shift instructions to read the exact address [29]. This can cause unexpected results in MultiOsCom receive function. To avoid such misalignment, padding bits can be added in the structure. In the MultiOsCom module, this problem is avoided by ensuring the pointer calculation results in a number which is a multiple of the word size. As explained in the diagram above, the calculation of next message address depends on the following code:

1. `sizeof(MultiOsCom_Buffer)+sizeof(Message_Signal)`
2. `+(Max_message_length)*Tx_counter`

The `sizeof(MultiOsCom_Buffer)` and `sizeof(Message_Signal)` is fixed. Therefore, by selecting the variable ‘max_message_length’ properly, it can be ensured that the next address is a multiple of the word size.

### 4.1.3 Synchronization

In a multi-threaded program, atomic operations like lock(), release() and wait() are used for synchronization between producer and consumers. With these interlocks, it can be ensured that only one thread modifies the buffer at one single time. But these interlocks bring about few problems such as deadlock and starvation. [30] To avoid these problems, separate message buffers are used for both OS instances and the synchronization is done by polling of transmit and receive counters on respective message buffers.
4.2 Security

Since human lives can be dependent on its proper functionality, an automotive system is considered as a safety critical system. As safety and security go hand in hand, this section takes into consideration the security aspects of MultiOsCom module. These are related to automotive embedded information security.

4.2.1 Use Case and Threat Scenario

To develop a threat scenario, firstly a possible use case is defined as follows: A software update currently is done over CAN bus or Ethernet by connecting the system to another device which connects to the server of car manufacturer. A software update is then downloaded and flashed on to the ECU. There are some configurations and manual wiring connections need to be done for this process. The car needs to be brought to a workshop for this update. To automate this process, following use case is proposed.

The figure 4.5 the use case of MultiOsCom module for over the air update of an ECU. It shows a multicore ECU with Android OS on core 0/1 and AUTOSAR on core 2/3. Android OS is connected to the internet via a wireless network. Original Equipment Manufacturers (OEM’s) publish updates for ECU on a centralized server. Periodically, an Android application scans this update server and when
the update is available for a particular ECU it notifies the user. If the user wishes to update the ECU software, the ECU software is first downloaded on Android OS. Using MultiOsCom module, this update is then transferred to AUTOSAR. In this situation AUTOSAR is considered a more secure side of the system and Linux is considered a less secure side because of the following reason: As described in the Chapter 3, Android OS uses Linux kernel for its primitive functions like memory management, network management etc. Since Linux is a free and open source operating system, users can customize it by creating personal applications. As a publicly reviewed operating system it is largely considered a secure system, and it is one of the well-accepted operating systems for web servers around the globe. However, few factors point to the security risk like its sheer size and complexity. Having millions of lines of code as open source, there is a likelihood of some mistakes in the code. Attackers are known to find out these mistakes and exploit them. The most relevant aspect why Linux is considered insecure in this scenario is because it is the connectivity gateway to the outside world via internet, due to this connectivity it can become a more vulnerable target as compared to AUTOSAR. The other OS counterpart (AUTOSAR) has its own implementation of a real time operating system which is closed and does not have a direct Wi-Fi or USB connection to the outside world.

**Threat considerations:** If the update downloaded from OEM server is modified by an attacker, it can cause system failure or ambiguity in runtime. Since AUTOSAR OS instance has access to vehicles sensors/actuators and also the connection to other communication buses, an attack on this system can cause severe consequences, depending on the role of the ECU in the system. For example, consider an Infotainment ECU with both Linux and AUTOSAR systems running on same CPU but on different cores. In such situation if the Linux operating system is compromised and Linux root user level access is gained by the attacker, following attack possibilities are possible:

1. Attacker may flood the AUTOSAR side by sharing garbage data at high speeds on the multi OS communication mechanism, hampering the functioning of the infotainment system.

2. Attacker may interrupt multi OS communication by inserting false values on shared memory buffer, if the car is already on the road this attack can distract the driver causing a safety threat.

3. Attacker can interfere with the ECU firmware update process by interfering with the shared memory buffer during such an update. However the firmware updates are usually cryptographically signed by car manufacturers and are verified internally by ECUs. If the attacker is able to surpass the
ECU firmware update signature verification process, malicious code then can be written in the ECU. Since AUTOSAR controls the external bus systems like CAN. With maliciously updated firmware, arbitrary CAN messages can be sent to other ECUs on the same network. Access to the CAN bus will open up Pandora’s Box for the attacker. For example, with compromised CAN messages, attacker can send wrong speed signals to Automatic Cruise Control (ACC) resulting in unwanted acceleration in car speed or a total break condition.

**Summary of possible attacks:**

1. **Man in the middle**: Attacker can create an application to read shared memory buffer.

2. **Impersonation**: Attacker can write application which behaves similar to a genuine application and take control of the communication

3. **Denial of service (DoS)**: Attacker can create an application to flood the AUTOSAR shared memory buffer with data at high speeds keeping AUTOSAR busy or even crash the system.

### 4.2.2 Solution 1: Cryptography

As described in the Section 3.3, for ensuring security in cryptographic systems there are mainly two mechanisms, namely Symmetric and Asymmetric. In Symmetric cryptosystems, AES is widely used in industry because of speed, relatively easy implementation and efficient encryption as compared to other symmetric mechanisms. But symmetric cryptosystems need to share keys before the actual secure communication take place. In the case of MultiOsCom module, source code of one of the module for Linux will be provided to the application developers. Therefore the symmetric keys will be exposed. On the other hand, RSA being a public key cryptosystem is used as a method to exchange keys for further use in symmetric cryptosystem. Thus the speed and efficiency of symmetric cryptography along with key sharing methods of RSA are combined in TLS (Transport Layer Security) or SSL (Secure Sockets Layer) protocol. Generally TLS/SSL protocol is used in connection oriented reliable protocols like TCP (Transmission Control Protocol) for encapsulating application specific protocols HTTP (Hypertext Transfer Protocol). It is wise to use a publicly viewed and standardized protocol for key exchange. Therefore existing TLS/SSL protocol can also be used in MultiOsCom module for establishing a secure channel.
A procedure to establish a secure channel similar to TLS/SSL is shown in figure 4.6. In this method communication channel is the shared memory buffer. RSA is used initially for exchanging keys and then AES is used for encrypting/decrypting actual communication data. The keys will not be stored anywhere in the code and will not be shown anywhere on Tresos GUI as well. Therefore the problem
of supplying keys separately can be eliminated. New keys will be provided on each reset. It will make it hard for the attacker to guess the keys as it will generate different cipher text on every restart for similar plain text.

Attack situations:

1. **Man in the middle**: This will be an actual threat only when the attacker steals the private keys. Therefore at the time of initialization, the Private RSA keys should be kept in protected memory areas. The symmetric keys generated on runtime should also be kept in protected memory areas of both the OS instances. But there is no protected memory area on the shared memory. Also there is no secure hardware unit present on current development hardware.

2. **Impersonation**: While publishing the Public keys, an attacker can replace the key with own key and impersonate as the legitimate OS instance and gain symmetric keys. Therefore there should be some way of authentication for the public key. One solution is to store a signature of the generated key along with the actual key. A hardware identity like MAC address can be used as seed for generating the signature. But the hardware identity is also not tamper proof and can be changed by the users. Therefore impersonation attack is also possible for attacker.

3. **Probabilistic derivation of keys**: If the keys are generated by random number, an efficient random number generation algorithm is a must.

4. **Denial of Service (DoS)**: In such type of attack the attacker tries to shut down or disturb the services provided by the ECU by sending huge amount of data to the ECU. For example, if the data receive function of the ECU initially calculates hash signatures of CRC, flooding the ECU with too much data can result in keeping the ECU too busy to perform other operations or crash the ECU altogether.

The solution in figure 4.6, does not provide protection against denial of service is not ensured. Since all the users of android will be able to call MultiOsCom module, even with this security mechanism attacker can create own secure channel. This channel being authenticated by the above mechanism can be further used to flush too much data into the AUTOSAR side. The cryptographic channel establishment using a TLS/SSL type of protocol is not effective in protecting the AUTOSAR side from attacks as mentioned above. Following section presents another solution for this threat scenario.
4.2.3 Solution 2: Bandwidth and message content monitoring

The method shown in figure 4.7 is based on monitoring the communication channel to detect intrusions. Both the OSs have separate memory areas and there is no direct dependency between them after booting process.

As described in the Section 4.1.1, the communication is based on shared memory and the synchronization between the two operating systems is based on polling of Rx and Tx counters of respective shared memory buffers. Using timestamps, the maximum bandwidth of the channel can be limited to prevent against Denial of Service attacks. This limitation will reduce the rate of data transfer. Considering the threat scenario where attacker floods the AUTOSAR shared memory buffer with garbage values and ECU keeps calculating CRC values can result in ambiguity during run time. It is proposed that speed of both receive and send function should be limited. This will be helpful when there are multiple messages of different frequency on the shared memory channel. The messages sent by high frequency channels will flood the shared memory buffer and messages sent with lower frequency will be missed by the receiver because the buffer is a ring buffer and communication mode is non-blocking.

The minimum time interval between two messages can be set in the EB Tresos
configuration GUI. It is assumed that the user will approximately know the frequency of the data transfer and possibly the type or range of the data in the messages during configuration. For example data transfer rate of messages defined for an Android application which locks or unlocks the doors will be low, as compared to messages which carry temperature sensor data for climate control application. Thus the limitation on bandwidth is useful and realistic method for preventing denial of service attacks.

Bandwidth limitation also gives a possibility to detect intrusions by recording number of times an application tries to send data faster than the specified limit. Using this concept as a basis, a firewall can be developed to drop messages which

- Operating system functionality
- Do not match the datatype specified
- Do not match CRC specified along with the message
- Do not fall in the specified range during configuration
- Do not match the pattern defined during configuration

Summary:

Classical resource sharing model called the "producer consumer model" is presented as communication model for exchanging data between Linux and AUTOSAR operating systems. General problems associated with such models are discussed.

The Solution 2 described in Section 4.2.3 is at risk of man in the middle attack and impersonation. But even in case of attack, the functioning safety critical system AUTOSAR will not be hampered due to denial of service attack. Solution 2 is chosen for further implementation, details of which are in the next chapter.
This chapter gives implementation specific details for MultiOsCom module. A variety of diagrams are used to capture different aspects of functions, interfaces and interactions. A detailed design of integration of the module with EB Tresos studio is provided with GUI and code generator.

5.1 Development Hardware

SABRE for Automotive Infotainment Based on the i.MX 6 Series

- ARM Cortex A9 Quad core processor
- 4 GB DDR3 RAM
- SD Card support
- JTAG, Ethernet and RS232 and CAN Bus interface
- LCD/Touch Panel

As seen in the figure 5.1, the DIP switches provided on the board give the ability to configure this board to boot from various devices. The boot sequence was configured to boot from the SD card. On the SD card a boot partition containing a script "boot.scr" executed the Uboot Bootloader [31] sequence. The binary file for entire EB Autocore OS was loaded in the SD Card which was copied to RAM area of the microcontroller on system reset. Ubuntu virtual machine with arm-linux-gnueabihf (C compiler for armhf architecture) was used for compiling and
debugging Linux side of the MultiOsCom code. AUTOSAR code was compiled on a Windows platform using Wind River Diab Compiler 5.9.3.0 Compiler.

### 5.2 Shared Interface

A fixed protocol is required for communication between Linux and AUTOSAR. In MultiOsCom module, it is implemented by defining a common shared interface between both operating systems. This shared interface consists of static message structures along with dynamically generated message structures from code generator. These message structures are linked to the configuration parameters as defined in the EB Tresos Studio. The static struct definitions are stored in MultiOsCom.h and dynamically generated configuration parameters are stored in MultiOsCom_Cfg.h. The data types are platform dependent and their implementation can vary between operating systems. In the case of MultiOsCom, module specific datatypes are defined. These datatypes should be mapped to the platform dependent data types at the time of integration. The data type definitions are stored in MultiOsCom_Types.h Table 5.1 provides a list of configuration parameters defined in MultiOsCom_Cfg.h file:
5. IMPLEMENTATION

5.2. Shared Interface

Figure 5.2: Shared interface class diagram
### 5.3 Shared Memory Buffer Structure

#### Listing 5.1: Buffer struct definition

```c
static struct file_operations fops =

typedef struct
{
    MultiOsCom_uint32 magic_number;
    MultiOsCom_uint32 rx_counter;
    MultiOsCom_uint32 tx_counter;
    MultiOsCom_uint08 *Signal_startaddr;
} MultiOsCom_Buffer;

typedef struct
{
    MultiOsCom_uint32 id;
    MultiOsCom_uint32 sequence_number;
    MultiOsCom_uint32 length;
} MultiOsCom_Signal;

typedef struct
{
    MultiOsCom_uint08 SIG1[1];
    MultiOsCom_uint08 SIG2[2];
    MultiOsCom_uint08 SIG3[3];
    MultiOsCom_uint08 SIG4[1];
}
```

---

### Table 5.1: Shared Interface Configuration Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTIOSCOM_CFG_MAXMSIGNALLEN</td>
<td>Defined as the maximum length by summing up length of individual signals in each message structure</td>
</tr>
<tr>
<td>MULTIOSCOM_CFG_ASR_MAGICNUMBER</td>
<td>Unique number to identify if MultiOsCom AUTOSAR is initialized</td>
</tr>
<tr>
<td>MULTIOSCOM_CFG_LINUX_MAGICNUMBER</td>
<td>Unique number to identify if MultiOsCom Linux is initialized</td>
</tr>
<tr>
<td>MULTIOSCOM_CFG_ASR_TXBSTARTADDRESS</td>
<td>AUTOSAR shared memory start address</td>
</tr>
<tr>
<td>MULTIOSCOM_CFG_LINUX_TXBSTARTADDRESS</td>
<td>Linux shared memory start address</td>
</tr>
<tr>
<td>MULTIOSCOM_CFG_ASR_TXBUFLENBYTE</td>
<td>AUTOSAR shared memory buffer length</td>
</tr>
<tr>
<td>MULTIOSCOM_CFG_LINUX_TXBUFLENBYTE</td>
<td>Linux shared memory buffer length</td>
</tr>
<tr>
<td>MULTIOSCOM_CFG_[&lt;Message Name&gt;]_ID</td>
<td>User defined Message ID</td>
</tr>
<tr>
<td>MULTIOSCOM_CFG_[&lt;Message Name&gt;]_CBK</td>
<td>User defined name of callback functions, invoked by the receive function of individual messages.</td>
</tr>
</tbody>
</table>
5. IMPLEMENTATION

5.3. Shared memory buffer structure

The above code snippet in Listing 5.1 shows the structure of the shared memory buffer consisting of two fixed structs defined as MultiOsCom_Buffer, MultiOsCom_Signal and series of structs MultiOsCom_MSG_MSG_0 to MultiOsCom_MSG_MSG_<N>, which are automatically generated as per user defined messages. It is following the message structure defined in the concept chapter 4. The length of the overall ring buffer then becomes:

\[
\text{sizeof(MultiOsCom_Buffer) + sizeof(MultiOsCom_Signal) + sizeof(MultiOsCom_MSG_MSG_0)} + \ldots + \text{sizeof(MultiOsCom_MSG_MSG_<N>)}
\]

Table 5.2 describes the usage of individual variables of the structure:

<table>
<thead>
<tr>
<th>Structure</th>
<th>Variable</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiOsCom_Buffer: This is a</td>
<td>Magic_number</td>
<td>A defined number to identify if the SenderReceiver is initialized successfully.</td>
</tr>
<tr>
<td>common message structure</td>
<td>Rx_counter</td>
<td>Count of received message packets.</td>
</tr>
<tr>
<td>between AUTOSAR and Linux</td>
<td>Tx_counter</td>
<td>Count of sent message packets.</td>
</tr>
<tr>
<td>defining the shared memory</td>
<td>Signal_startaddr</td>
<td>Memory address of the first possible message address.</td>
</tr>
<tr>
<td>buffer header.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MultiOsCom_Signal: This is a</td>
<td>id</td>
<td>A fixed id of every message type.</td>
</tr>
<tr>
<td>common message structure</td>
<td>Sequence_number</td>
<td>Sequence number of every message sent, auto incremented by the MultiOsCom send function.</td>
</tr>
<tr>
<td>between AUTOSAR and Linux</td>
<td>Length</td>
<td>Length of the message as defined by its individual size of message structure definition given below.</td>
</tr>
<tr>
<td>defining the shared memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>buffer. Used as a header before</td>
<td></td>
<td></td>
</tr>
<tr>
<td>every message.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MultiOsCom_MSG_MSG_0 to</td>
<td>SIG[1] ..</td>
<td>Signal name and length are generated at the configuration by user.</td>
</tr>
<tr>
<td>MultiOsCom_MSG_MSG_&lt;N&gt;: Structures defining individual MultiOsCom message structures</td>
<td>SIG[n]</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Usage of structure variables
5.4 Prototype implementation details

In this section, following function calls are described in detail:

- **MultiOsCom_Init()** - For initialization of the shared memory interface and Transmit and receive counters.

- **MultiOsCom_Send()** - For sending over the signal to other operating system.

- **MultiOsCom_receive()** - For receiving the signal from other operating system.

- **Callback Functions** - Are triggered for reception of particular messages.

**MultiOsCom_Init()**

![Interaction diagram](image)

Both shared memory buffers are first initialized after OS booting by setting the magic numbers and transmit and receive counters. The address of the shared memory buffer is taken from the configuration header file as defined in the EB Tresos Studio. The Send function is called by the user application whenever it needs to transfer data between the two operating systems. But the receive function is called cyclically by polling. Upon successful receipt of a new message, the corresponding call back function with pointer to new message is invoked.
As the send and receive functions are to be used as kernel level functions, they are designed as re-entrant functions. It is done with the help of various interlocks. For example: If the magic numbers don’t match, no further execution is carried out. In case where the buffer is full, the Tx counters are simply reset. As the buffer is a free running, the data will be overwritten even if it is not read by the receiver. In this process there is a chance of missing data packets but it is intentional by design so as to keep the most updated values on the buffer. These interlocks are necessary for an error free communication, details of which are provided through behavior diagrams shown in the next sub-section.
MultiOsCom_Send() 

The behavior diagram shown in figure 5.4 describes the flow and the activities performed by MultiOsCom_Send function. This function has an additional input called Message ID and a pointer to the message which is not visible in the above diagram. The first step is to check if the transmit buffer is properly initialized.
This is done by checking the magic number. The second step is to check the time interval between the last sent message and current time. A message is sent only if the time interval is greater than the minimum time interval specified in EB Tresos, otherwise the message is dropped. Depending on the number of transmitted messages, the next possible message address is calculated by following code:

```c
MultiOsCom_dest_Signal_ptr = MULTIOSCOM_CFG_ASR_TXBSTARTADDRES
+(void*)sizeof(MultiOsCom_Buffer)
+(void*)(Size_of_signal*(MultiOsCom_TxBufferPtr->tx_counter))
```

Depending on the Message ID, a message packet is prepared by packing individual signals in a message structure and sequence number. If the CRC (cyclic redundancy check) flag is set in Tresos Studio configuration, then the CRC is calculated in Send function and written along with the message structure. This prepared packet is written to destination address and the Tx counter of the sender buffer is incremented for synchronization. As the buffer is circular, the Tx counter is reset when it reaches to the maximum possible count defined in configuration file.

**MultiOsCom_receive()**

The figure 5.5 depicts the behavior diagram of the receive function. In the MultiOsCom_receive function, firstly the time interval between the last sent message and current time is checked. A message is read only if the time interval is greater than the minimum time interval specified in EB Tresos, otherwise the message is dropped. It is checked that the receive buffer is properly initialized by checking the magic number. Both the configurations are considered matched if the corresponding magic numbers match. The Rx and Tx counters are compared for checking if that is a new message on the buffer with following code:

```c
((MultiOsCom_RxBufferPtr->tx_counter)
- (MultiOsCom_RxBufferPtr->rx_counter) != 0UL)
```

If the result of above operation is non zero and not negative, then the address of the received message is calculated by following code:

```c
MultiOsCom_Receive_Signal_ptr = (void *)MultiOsCom_RxBufferPtr
+(void *)sizeof(MultiOsCom_Buffer)
+(void*)(((sizeof(MultiOsCom_Signal) + MULTIOSCOM_CFG_MAXMSGNALLEN) * (MultiOsCom_RxBufferPtr->rx_counter)));```
If the CRC flag is set in the EB Tresos Configuration, the CRC of the received message is calculated in the receive function and checked against CRC sent along with the message. If the CRC check results a match, then depending on the ID...
of the received message, a validation is carried out with \texttt{MultiOsCom\_validate} function. On successful validation, the corresponding callback function is called. The Rx counter of the sender is increased and check for a new message is again performed with Rx and Tx counter comparison. The activities of checking for new message and invoking callback loops until all the received messages are read successfully.

\textbf{MultiOsCom\_Validate()}

A regular expression are used to check configuration of the MultiOsCom module. Also the requirements for each signal is specified in EB tresos Studio. These requirements are crosschecked in validate function, if the received message does not fulfill the requirements, it is dropped. The possible checks done by Validate function are listed below:

1. Check if the integer is within the specified range
2. Check the Data type
3. Check if alphanumeric or special characters
4. Check if the message received fits a struct

It is assumed that the user already knows the nature of the data at the time of the configuration. Therefore, this validation function is entirely dependent on user configuration. If required, this function can also be skipped by setting a flag to 'false' in the GUI configuration.

\textbf{Callback Functions:}

Callback functions can be passed as an argument to another function and which can be invoked after occurrence of an event.

In MultiOsCom module, these functions are defined for individual message \texttt{MultiOsCom\_Cbk.c}. On receipt of a message, its individual callback function is called and passed a pointer to the message. The code below shows the callback function on receipt of message MSG\_0:

\texttt{
\begin{verbatim}
extern boolean CALLBACK1(MultiOsCom\_uint08 *MultiOsCom\_MSG\_MSG\_0)
{
    DEBUG_INFO("AUTOSAR:MSG\_0_ptr->SIG1=%s\n",pointer_to_message->SIG0);
    DEBUG_INFO("AUTOSAR:MSG\_0_ptr->SIG1=%s\n",pointer_to_message->SIG1);
    DEBUG_INFO("AUTOSAR:MSG\_0_ptr->SIG1=%s\n",pointer_to_message->SIG3);
    DEBUG_INFO("AUTOSAR:MSG\_0_ptr->SIG1=%s\n",pointer_to_message->SIG4);
    return TRUE;
}
\end{verbatim}
}

\texttt{Listing 5.2: Callback function}
In the code shown above individual signals of this message are printed to console. Many other operations can be written in these functions. For example: the unpacked signals SIG0 to SIG4 can be sent over to other ECU over CAN bus. This implementation is completely left to the user.

5.5 MultiOsCom Tresos Plugin

The MultiOsCom module is developed as a plugin in EB Tresos Studio. EB Tresos studio provides java extension points to create and integrate eclipse based plugins with itself. The figure 5.6 is the overall structure of a plugin: Plugin.xml &

![Plugin Structure](image)

**PlugIn-Archive meta-information (MANIFEST.MF):**

The file META-INF/MANIFEST.MF contains meta-information describing the Plugin. The meta information is used to associate the PlugIn-Archive with a PlugIn. Plugin.xml contains information like plugin name, vendor, version link to java extensions for code generator and path to configuration files.

**Plug-in folder structure has following folders**

- generate : code templates for the generator
- include : header files
- make : makefiles for the plug-in
- src : *.c files
- config : EB description file *.xdm
- .c and .h Code templates:

The final output expected from the plugin is the .c and .h files and corresponding makefiles. The C code developed for MultiOsCom module is used as basis for
code generation. The code templates are created from developed code and segregated into static and dynamic code sections. These code sections are marked by syntax which the code generator parses and generates the final output to be compiled by the compiler. Code generation process is explained in detail in next sections.

![Figure 5.7: Configuration to code](image)

The figure 5.7 shown above describes the process of generating code from EB Tresos GUI. The GUI configuration stores the settings in an xml file which is used by the code generator to generate .c/.h files for both Linux and AUTOSAR.

### 5.5.1 GUI and Configurator

The extension points provided by EB Tresos give the possibility to link the XDM (XML Documentation Markup) data model to GUI. The Dialog Engine stores the changes made by the user directly into the corresponding Configuration Data tree and provide GUI elements to enter configuration parameters for the module the Plugin handles.
XDM Datamodel:

MultiOsCom plugin is based on a data model in the XDM format. Data Description can be distributed over multiple XDM files (Hardware independent and hardware dependent part). For example: Definition of MultiOsCom Signals data model as shown in the listing 5.3:

Listing 5.3: MultiOsCom Signals XDM Data model

```xml
<datamodel>
  <view name="MultiOsCom" Version="1.0">
    <v_tab name="Configuration">
      <v_list name="MultiOsComSignals">
        <v:ctr name="MULTI_OS_COM_MSG_NAME" type="IDENTIFIABLE">
          <a:a name="LABEL" value="Signal Name"/>
        </v:ctr>
        <v:var name="MULTI_OS_COM_SIG_ID" type="INTEGER">
          <a:a name="LABEL" value="Signal ID"/>
        </v:var>
        <v:var name="MULTI_OS_COM_SIG_START_POS" type="INTEGER">
          <a:a name="LABEL" value="Signal Byte Position"/>
        </v:var>
        <v:var name="MULTI_OS_COM_SIG_LEN" type="INTEGER">
          <a:a name="LABEL" value="Signal Length(Byte)"/>
        </v:var>
        <v:var name="MULTI_OS_COM_SIG_RANGE" type="STRING">
          <a:a name="LABEL" value="Value Range Min-Max"/>
        </v:var>
      </v:list>
    </v_tab>
  </view>
</datamodel>
```

The hierarchy of the data is defined as xml nodes. As seen in the xdm code listing 5.3, the MultiOsComSignals list container is placed under a tab called configuration. Parameters are defined inside the list container using `<v:var name="parameter_name" type="INTEGER">`. Similar XDM data model is created for entire plugin structure. This results in GUI configurators as described in the next sections.
MultiOsCom configuration GUI:

![MultiOsCom configuration GUI](image)

Figure 5.8: MultiOsCom configuration GUI

The above figure 5.8 shows the graphical user interface for the Configuration of the MultiOsCom plugin. It is generated from the XDM data model in the config folder. As defined in the concept chapter 4, the first two parameters are the start addresses of the shared memory buffer for AUTOSAR and Linux and the corresponding lengths of the buffers. The parameter ‘Min time interval between two Messages (Microseconds)’ configures the bandwidth. Setting this parameter to ‘0’ will remove the limitation on the bandwidth. The ENABLE_DBG_SUPPORT flag controls the debug messages on the serial console. It is used during development and testing of the module.
5. IMPLEMENTATION

5.5. MultiOsCom Tresos Plugin

**MultiOsCom Message Configuration GUI**

![MultiOsCom Message Configuration GUI](image)

Figure 5.9: MultiOsCom message configuration GUI

The figure 5.9 shown above depicts the configuration window for MultiOsCom messages. User can create multiple messages with unique message ID and callback function name. Callback functions are filled with runnable code by users after configuration and code generation.

**MultiOsCom Signal Configuration GUI**

![MultiOsCom Signal Configuration GUI](image)

Figure 5.10: MultiOsCom signal configuration GUI

Every message contains a list of signals which can be of varying length. This GUI provides configuration settings for MultiOsCom signals. The signals have
a unique ID, length and its position in the message package. The "value range" field is used for entering regular expression for individual signals. If the firewall function is enabled, as described in section 5.4, MultiOsCom_Validate function checks this signal with corresponding regular expression.

Cross Dependencies and Parameter Validation The AUTOSAR code for MultiOsCom module uses few other basic software modules like CRC. There is a dependency between these two modules. The dependencies between variables, described via XPath, allow marking specific configurations as valid or invalid. The GUI, in reaction to changes in the configuration, validates the dependencies and grays out, changes title, tooltips, etc. The attribute values can be either statically defined or calculated via XPath.

```
<datamodel>
  <view name="MyPlugin" version="1.0">
    <attribute name="INVALID" type="XPath">
      <test test="/OtherPlugin/MASK > 0 and /otherDriver/MASK < 100 " false="Invalid node ID"/>
    </attribute>
  </view>
</datamodel>
```

Figure 5.11: XML code for checking cross dependency

The code shown in 5.11 is an example of parameter validation and it will result in a dependency check as shown in the figure 5.12
5. IMPLEMENTATION

5.5. MultiOsCom Tresos Plugin

Figure 5.12: Datamodel dependencies and validation

**Makefiles:**
Since the module codes are dynamically generated and there can be different versions of a code, there is a huge amount of configuration parameters. Compiler should know which files should or should not be compiled. A makefile provides a list of files/folders which should be included in compilation process. For MultiOsCom module, a makefile is defined in plugin to include the entire generated file in the compilation process.

5.5.2 Code Generation:
The generation process (handled by the Generators) is responsible for the following generation tasks:

- Verifying the consistency of the settings made for its Plugin
- Generate operating system or module sources
- Generating documentation documenting the made settings
- Generate code templates and makfiles

For MultiOsCom module, a template code with XPath expression is used to access the configuration data. A Generator implements the code generation process. The Generator not only has the job to generate module code but also is in charge of verifying the configuration. The generation API is so flexible that virtually any number of generation/verify task may be supplied by the Generator. In this way, it should be possible to encapsulate all target platform dependent code in the Generator. In the listing 5.4 shown below is one of the code snippets for
MultiOsCom_Cfg.h file generator.

**Code Generator Input:**

Listing 5.4: Code generator Input

```plaintext
#ifndef MultiOsCom_Cfg_h
#define MultiOsCom_Cfg_h

#define MULTIOSCOM_CFG_MAXMSGNALLEN 7u
#define MULTIOSCOM_CFG_ASR_TXBSTARTADDRES 0x54000000
#define MULTIOSCOM_CFG_MSG_0_ID 1
#define MULTIOSCOM_CFG_MSG_1_ID 2
#define MULTIOSCOM_CFG_MSG_3_ID 3
#define MULTIOSCOM_CFG_MSG_4_ID 4
#endif /* MultiOsCom_Cfg_h */
```

The maximum length of the message buffer is calculated by looping through all the message definitions and further adding the length of individual signals. The

**Code Generator Output:**

Listing 5.5: Code generator output

```plaintext
#ifndef MultiOsCom_Cfg_h
#define MultiOsCom_Cfg_h
#define MULTIOSCOM_CFG_MAXMSGNALLEN 7u
#define MULTIOSCOM_CFG_ASR_TXBSTARTADDRES 0x54000000
#define MULTIOSCOM_CFG_MSG_0_ID 1
#define MULTIOSCOM_CFG_MSG_1_ID 2
#define MULTIOSCOM_CFG_MSG_3_ID 3
#define MULTIOSCOM_CFG_MSG_4_ID 4
#endif /* MultiOsCom_Cfg_h */
```
maximum length obtained from this loop is stored in parameter "MULTIOSCOM
Cfg_MAXMSGNALLEN". Other parameters such as MULTIOSCOM_CFG
ASR_TXBSTARTADDRESS can be directly accessed from GUI using XPath query.
By this method, MultiOsCom .c and .h code for both Linux and AUTOSAR sides
is generated by a template based code generator.

Summary:
This chapter described the implementation details of key functions like Mul-
tiOsCom_Send() 5.4 and MultiOsCom_receive() 5.4. The security aspects of the
communication are also incorporated in these functions. These functions are then
consolidated in a EB tresos Plugin. As described in section 5.5 this plugin helps in
configuring the overall system and generates code accordingly. The performance
of the generated code along with test setup is described in following chapter.
This chapter gives details about the test setup and measurements for performance. The evaluation criteria for the results is measurement of data transfer rate. The data transfer rate should be greater than other fieldbus protocols like CAN. Furthermore as described in section 4.2.3, messages should be dropped if sent at rates higher than specified in EB Tresos studio, and should also be dropped if they do not fit the validation criteria.
6.1 Test Setup

Figure 6.1: Test setup

Figure 6.1 shows the test setup for MultiOsCom module. The existing hardware shown in Section 5.1 has a RS232 and CAN interface. With the help of EB tresos Studio, a CAN node is established on the AUTOSAR OS.

Figure 6.2: CAN interface for USB

PCAN was used as an external CAN node for sending and receiving CAN messages from Windows PC. Serial console connection is mainly used by Linux OS debugging. Since both the OS are running on the same microcontroller, both have
6. RESULTS

6.2 Linux test application

A test application was written in Linux user space to interact with the MultiOsCom char device. A code snippet for writing to the device is shown in listing 6.2:

```
#define DEVICE "/dev/MultiOSCom-dev"

int debug = 1, fd = 0;

int write_device() {
    int write_length = 0;
    ssize_t ret;
    char *data = (char *)malloc(20480 * sizeof(char));
    printf("please enter the data to write into device\n");
    scanf("%[^\n]", data);
    write_length = strlen(data);
    if(debug)printf("the length of data written = %d\n", write_length);
    ret = write(fd, data, write_length);
    if(ret == -1)
        printf("writing failed\n");
    else
        printf("writing success\n");
        if(debug)fflush(stdout);
    free(data);
    return 0;
}
```

Figure 6.3 shows the trace log of test application on Linux serial console. The messages starting with "MultiOsCom:" are debug traces of the MultiOsCom character device. Timestamps were used for calculating the time interval of the send and receive functions. Messages with varying length were sent and the time intervals were measured. For deployment in the use case, there has to be a background application to invoke the poll buffer cyclically.
6.3 AUTOSAR Test application

A runnable triggered by CAN message receive event is configured. For testing, the send and receive functions of MultiOsCom were implemented in the runnable shown in listing 6.2.

Listing 6.2: Test application for AUTOSAR

```
FUNC(void, COM_APPL_CODE) CanMsgKomb1_01(void)
{
    uint16 speed = 0u;
    uint16 rpm = 0u;
    uint8 buf[5];
    /* Get can messages */
    Com_ReceiveSignal(ComConf_ComSignal_SGGeschwindigkeit_512R, &speed);
    Com_ReceiveSignal(ComConf_ComSignal_SGRPM_512R, &rpm);
    DEBUG_INFO("\nAUTOSAR: CAN Msg Received\n");
    if (rpm == 0x1111)
    {
        /* Send over to Linux*/
        MultiOsCom_Send(1,"helloLinux");
    }
}
```
As seen in the figure 6.4, the Message with ID 110h is the cyclic counter in AUTOSAR. With Message ID 200h it is possible to call MultiOsCom send or receive functions. For deployment in the use case, there has to be a cyclic task to poll the read buffer.
6.4 Results

The synchronization of send and receive between the OS is implemented by polling. Therefore, following measurements are taken without accounting the synchronization overhead. These are basically time interval taken by MultiOsCom module for reading and writing data on the shared memory. For measurement of data transfer performance, a hardware counter was configured to measure execution cycles during send and receive functions. The execution cycles of respective send and receive functions were later added together for corresponding messages sizes. The bandwidth of the communication is calculated against clock speed of the processor.

**Misaligned memory:**

![Execution Cycles at 792MHz (Misaligned memory pointers)](image)

Figure 6.5: Effect of misaligned pointers

The figure 6.5 shows the effect of misaligned memory pointers. In the initial stages of this thesis, the maximum bandwidth achievable bandwidth was between mere 20 KBps and 1 MBps, which was very poor as compared to memory to memory data transfer rates of the processor. This problem was solved by fixing the code for pointer arithmetic to prevent the misalignment. The workaround in pointer arithmetic was to ensure, that the calculated memory address should always be multiple of word size of the processor.
Performance results:

![Execution Cycles at 792MHz](chart.png)

Figure 6.6: Data transfer performance

The result of improved pointer arithmetic is seen figure 6.6. The execution time for sending signals over shared memory is not completely linear with data size. The reason is the data transfer mechanism. Memcpy function is used for memory to memory data transfer. Depending on the size of the data to be transferred, the compiler tries to choose the best suitable data transfer scheme for Memcpy function. Such optimizations are the reason for the nonlinear behavior for data size 2000 bytes and 6000 bytes in the graph shown in figure 6.6.
Performance with AES Encryption:

This measurement is carried out for considering the effect of encryption and decryption on the communication bandwidth. As described in figure 4.6 initially RSA algorithm is used to exchange keys for subsequent symmetric encryption. Therefore the performance of symmetric algorithm will be affect during actual communication. In this case, the existing crypto library of Linux kernel was used for AES algorithm with 128 bit key size. The latencies of for the functions of encrypt, send, receive and finally decrypt were added together to consolidate the figure 6.7.

![AES Encryption with 128 Bit key at 792MHz](image)

Figure 6.7: Performance results with AES (128 bit key)

The figure 6.7 shows overall communication bandwidth with AES algorithm, however synchronization overhead is not accounted for. The latencies of send and receive functions with encryption are measured. The effect of computation overhead of encryption on the overall bandwidth is evident in figure 6.7.

The performance in figure 6.6 is an idle case, in practice the transfer rates are forcefully limited by design. As mentioned in section 4.2.1, this is necessary for protecting the system against attacks involving denial of service, therefore messages are dropped if transferred at speeds higher than the specified speed in EB tresos Studio configuration. In the MultiOsCom_Validate function messages are dropped whose pattern does not match as specified in configuration for individual signal.
Conclusion

Depending on the results shown in the previous section and my experiences during concept establishment and implementation, following are some concluding remarks for this thesis work.

Data exchange between two operating systems is established via shared memory in a multicore ECU. The implementation is portable as it uses only shared memory as a medium and no other special hardware. Although the inter core interrupts can be used for notification of a new message but such interrupts are costly in terms of overall system performance, therefore polling was chosen as mechanism for synchronization. Since the communication channel is the memory to memory data transfer, the performance of Memcpy function significantly affects the overall communication performance, especially when the size of the message is large. Furthermore, misaligned pointers can also cause unexpected and drastic attrition in bandwidth.

Security aspects of the communication channel were studied during this thesis. In my concept for security, I have argued that limiting the minimum time between two messages helps in improving the security for the defined threat scenario. The reason for such an approach is mainly because of the difficulty in maintaining secrecy during key exchange and storage in cryptography, especially when the entire shared memory area is accessible to all the user programs. In situations where high bandwidth is required, for example transferring a file containing an update, a message with large size (for example 500KB) should be chosen such that even if
7. Conclusion

The minimum time between two messages is limited the overall bandwidth will still be high. The sanity of the received message is checked through checking the CRC sent along and pattern of the message. If an attempt is made to send messages higher than the specified rate, messages are dropped and notification to higher layers can be sent through callback functions. Though this solution is not completely safe from all type of attacks, but my belief is that security is not only about providing strong cryptography but also to provide mechanisms to prevent the safety critical system from crashing.

Lastly the implemented prototype was integrated in EB tresos Studio as a plugin. This plugin provides ease of configuration, automated integration with existing code and supports configuration checks before code generation.
Future Scope:

**Integration with AUTOSAR communication stack:** Current implementation of MultiOsCom module needs a brief integration step for use in complex automotive system. Additional OS tasks need to be defined in order to poll the receive buffer.

![Diagram showing AUTOSAR communication stack](image)

Figure 7.1: Future research

Figure 7.1 shows a possible future outlook for the MultiOsCom module. It can be further developed and integrated in AUTOSAR communication stack. Furthermore, for providing a strong cryptographic support, there can be more research done on ways to store and exchange cryptographic keys in a multi OS environment with better secrecy on a shared memory microcontroller.
Bibliography


Bibliography


Selbstständigkeitserklärung*

Ich erkläre gegenüber der Technischen Universität Chemnitz, dass ich die vorliegende Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.


Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keinem anderen Prüfer als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

Datum: 12.01.2016

Unterschrift: .................................................................

* Statement of Authorship

I hereby certify to the Technische Universität Chemnitz that this thesis is all my own work and uses no external material other than that acknowledged in the text.

This work contains no plagiarism and all sentences or passages directly quoted from other people's work or including content derived from such work have been specifically credited to the authors and sources.

This paper has neither been submitted in the same or a similar form to any other examiner nor for the award of any other degree, nor has it previously been published.