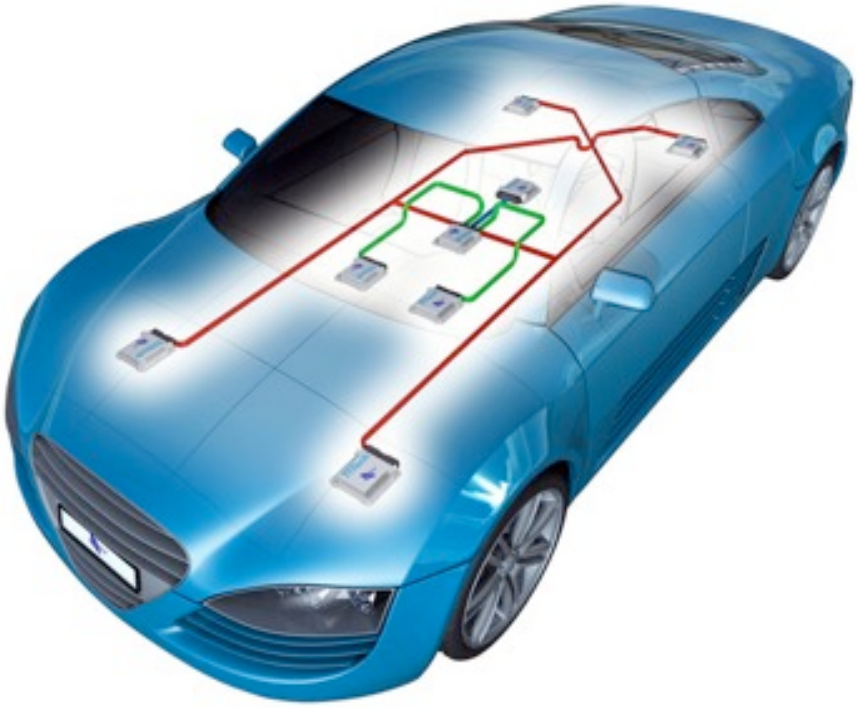
CHALMERS



AUTOSAR Communication Stack Implementation With FlexRay

Master of Science Thesis in the Programme Networks and Distributed Systems

JOHAN ELGERED
JESPER JANSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Gothenburg, Sweden, March 2012

AUTOSAR Communication Stack Implementation With FlexRay

JOHAN ELGERED
JESPER JANSSON

Examiner: ROLF SNEDSBÖL

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

[Cover:
Illustration of a FlexRay car platform, developed by TTTech Automotive and AEV.
http://www.audiblog.nl/?p=11008.]

Department of Computer Science and Engineering
Göteborg, Sweden, March 2012

Abstract

The demand for a high level of fault-tolerance and high bandwidth communication has increased, as a result of a growing number of ECU's (Electronic Control Unit) inside vehicles. FlexRay was developed to meet the continuously growing demands, offering a time-triggered functionality and a higher bandwidth than existing protocols. To facilitate ECU software development, a group of automobile manufacturers, suppliers and tool developers created the standardized automobile software, AUTOSAR (Automotive Open System Architecture). Merging FlexRay and AUTOSAR together creates a new challenge for the automobile industry. By combining a faster and more reliable protocol with a standardized automotive E/E (Electrics/Electronics) architecture, manufacturers hope to meet future demands. The goals with this project were fulfilled with wide margin. In order to obtain a working AUTOSAR communication stack with FlexRay, the FlexRay Interface module was developed according to the AUTOSAR specifications. The new FlexRay Interface module together with the available FlexRay Driver module was integrated into the Arctic Studio environment. This proved that modules of different AUTOSAR versions are compliant with each other and that modules from different vendors can be combined. Also, tests with a developed demonstrator indicated that the FlexRay communication stack is fully functional regarding PDU (Protocol Data Unit) handling. The demonstrator showed that PDU transmissions and receptions were successful, i.e. no messages were lost, and that all time properties were fulfilled. The configuration of the different AUTOSAR modules were made with the Arctic Core configurator in the Arctic Studio, and a new FlexRay configurator adapted to the existing FlexRay modules.

Acknowledgements

During our master thesis we have been in contact with a lot of people, and we want to give special attention to those who have been crucial for our project.

Rolf Snedsböl has been our supervisor and examiner at Chalmers. He has helped us with issues about the report and how to structure it.

Daniel Linné has been our supervisor at QRTECH. He has helped us with practical issues and we have discussed a lot of ideas and suggestions with him.

QRTECH AB and the staff at QRTECH have provided us with the necessary equipment and knowledge to be able to complete this project.

Ecore has provided the essential software and support.

Table of Contents

List of Abbreviations

API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basic SoftWare
CAN	Controller Area Network
CC	Communication Controller
CORBA	Common Object Request Broker Architecture
CRC	Cyclic Redundancy Check
ECU	Electronic Control Unit
GDB	GNU DeBugger
I-PDU	Interaction Layer Protocol Data Unit
ISR	Interrupt Service Routine
L-PDU	Link layer Protocol Data Unit
LIN	Local Interconnect Network
MCAL	MicroController Abstraction Layer
N-PDU	Network layer Protocol Data Unit
NIT	Network Idle Time
ODEEP	Open Dependable Electrical Electronic Platform
OS	Operating System
PDU	Protocol Data Unit
PEEDI	Powerful Embedded Ethernet Debug Interface
POC	Protocol Operation Control
RTE	Run-Time Environment
SDU	Service Data Unit
SWC	SoftWare Component
TCP/IP	Transmission Control Protocol/Internet Protocol
TDMA	Time Division Multiple Access
TTCAN	Time-Triggered CAN
USB	Universal Serial Bus
XML	eXtensible Markup Language

Chapter 1

Introduction

Network communication is an important part in modern vehicle industry. Components inside vehicles today are almost in all cases related to electronics. This has lead to an increasing number of ECU's (Electronic Control Unit). In order for the different ECU's inside a vehicle to communicate with each other, an interconnection through some sort of standardized network is needed. The rapid development in the electronics industry has lead to progress within the area of vehicle network technology [1].

As a result of an increasing number of electrical components inside vehicles, the demands on the communication have increased as well [2]. It is of great importance to maintain a high fault-tolerant level which challenges the capability of current standard protocols such as CAN (Controller Area Network) [3], LIN (Local Interconnect Network) [4] and TTCAN (Time-Triggered CAN) [5]. The automotive network communication protocol FlexRay was developed to meet the continuously growing demands, offering a time-triggered functionality and a higher bandwidth than existing protocols.

Different vendors use different hardware and software components to create an automotive communication network. This has caused difficulties in developing reliable software and integrating the different components. To facilitate software development, a group of automobile manufacturers, suppliers and tool developers created standardized automotive software, AUTOSAR (Automotive Open System Architecture) [6]. This standard aims at making the software more scalable, more reusable and easier to maintain, thus reducing the cost for software development.

FlexRay and AUTOSAR together create a new challenge for the automobile industry. Manufacturers hope to meet future demands when it comes to both product development as well as the use of the products, by combining a faster and more reliable protocol with a standardized automotive E/E (Electrics/Electronics) architecture.

1.1 Background

QRTECH AB [7] is a company which has its focus on development combined with consulting services within the field of electronics and software products. In order for QRTECH to be at the front edge, regarding new automotive technology, they have started the work of implementing parts of the AUTOSAR software stack combined with the FlexRay protocol. One of their products used for development within this area is the ODEEP (Open Dependable Electrical Electronical Platform) platform [8]. One of the goals with the ODEEP platform is to reach a fully implemented and compatible AUTOSAR platform. QRTECH will meet future demands from the automotive industry, with a solution involving both the AUTOSAR platform and FlexRay.

1.2 Problem description

The purpose of this master thesis is to obtain the AUTOSAR communication stack with FlexRay compliance. The final goal is to successfully run a demonstrator, which primary function is to send FlexRay frames in a FlexRay cluster consisting of three different nodes. This is achieved by completing the following tasks:

- Integrating the existing AUTOSAR FlexRay modules from QRTECH with available AUTOSAR modules in Arctic Studio
- Configure all AUTOSAR modules using the QRTECH ODEEP FlexRay configurator and the BSW (Basic SoftWare) Builder configurator in Arctic Studio
- Implement the AUTOSAR FlexRay Interface module release 4.0
- Adapt surrounding modules to be compliant with FlexRay Interface caused by version differences
- Implement a demonstrator for FlexRay communication between three different FlexRay nodes

1.3 Previous work within the AUTOSAR software stack for FlexRay

The ODEEP hardware used in this project is developed at QRTECH. There have been several previous master thesis performed, using this hardware for different applications. Some of these master theses have developed software adapted for ODEEP. Thus, the tools and software needed for the hardware was already in place before the start of this project. Much of the knowledge needed to use these tools was provided via written manuals and guides, but also through the employees at QRTECH.

Software components like start up routines, basic drivers and hardware tools were already developed and without them this project would not have been possible within the given time

frame. The main purpose for the ODEEP hardware is to be used for development and testing in the automotive industry. Therefore, some work has been done to make the ODEEP hardware compatible with AUTOSAR. Several modules have been implemented but the most important modules for this thesis were FlexRay Driver and FlexRay Interface. The modules are implemented with an older 2.1 release of AUTOSAR.

Chapter 2

Theory

The theory defining the base for AUTOSAR and FlexRay is described in this chapter. Firstly, an overview of the concept of AUTOSAR is presented in terms of different software layers. This is followed by more detailed information about the so called AUTOSAR modules that are treated in this project and their relation to FlexRay. Finally, the theory of the FlexRay protocol and its communication features are introduced.

2.1 AUTOSAR

AUTOSAR is a software architecture in the automotive industry, which has been globally developed by companies within the automotive and electrical branch. AUTOSAR is standardized and open, to facilitate software development and maintenance of applications, independently from the existing hardware [6]. It is especially suitable for ECU's that have the following properties:

- Strong interaction with hardware, such as sensors and actuators
- Connection to vehicle networks, for example CAN, LIN and FlexRay
- Microcontrollers (16 or 32 bit) with limited resources of computing power and memory
- Real-time systems.
- Program execution from internal or external flash memory.

It is not compatible with applications such as graphics library, CORBA (Common Object Request Broker Architecture), Bluetooth, USB (Universal Serial Bus), and TCP/IP (Transmission Control Protocol/Internet Protocol) [9].

To achieve hardware independency when using AUTOSAR, it distinguishes between hardware independent software and hardware dependent software by creating different layers. With this software structure AUTOSAR fulfills particularly four technical properties. These are modularity, scalability, transferability, and re-usability. Modularity divides software according to the requirements for specific components and their tasks. This makes it possible to adapt the

software to individual requirements. The scalability of functions makes it possible to use the same software for different platforms, which prevents redundant code for similar functional purposes. Transferability means optimizing the availability of resources in the electrical architecture of the vehicle. Re-usability of functions helps in improving the reliability of the system. A prerequisite for this to work in practice is that there have to be connections between the different software modules. This is an important aspect of AUTOSAR's functionality.

2.1.1 Layered software architecture

As previously mentioned, the AUTOSAR software stack is designed with different layers to separate software that is hardware dependent from software which is hardware independent. As shown in Figure 2.1, the highest abstraction level consists of three different software layers: Application Layer, RTE (Run-Time Environment), and Basic Software.

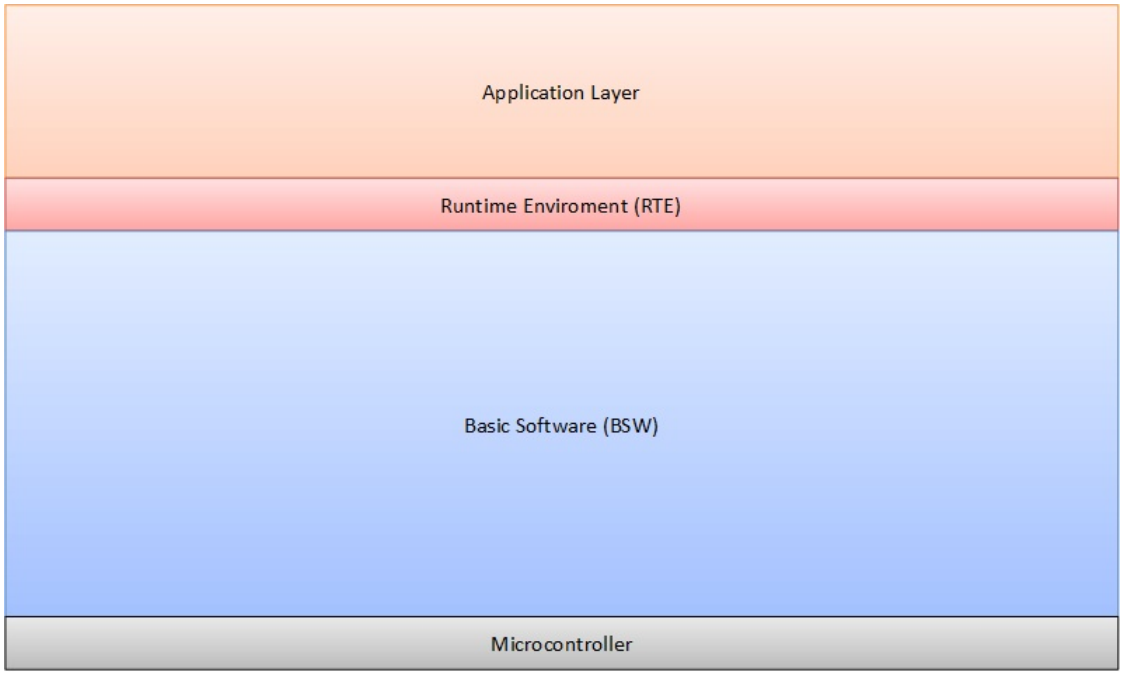


Figure 2.1: The three different software layers on the highest abstraction level: Application, Runtime Environment, and Basic Software [9].

Application layer

A basic design concept of the AUTOSAR software stack is the separation between application and infrastructure. In AUTOSAR, an application is composed of several connected SWC's (SoftWare Components). These should be structured according to AUTOSAR's definition and follow the constraints for the SWC's. Each SWC corresponds to a part of the functionality of the application. There exists no limitation for how large an SWC may be, thus, it may correspond to a function of a lesser extent or the complete functionality of an ECU. An

important property of the SWC is that it should be atomic. This implies that only one instance of a certain SWC may exist in a vehicle, thus, an SWC is assigned to only one ECU.

With this design, SWC's are independent of the type of microcontroller the ECU is using, the type of ECU on which the SWC is executed and the location of other SWC's that interacts with the current SWC. In order for an SWC to be able to execute and communicate with the AUTOSAR Basic Software, a run-time environment is needed.

RTE

All communication that occurs with an SWC and services and/or between an SWC with another SWC at the Application Layer is routed through the RTE. This holds for inter-ECU communication (communication between SWC's mapped on the same ECU) as well as intra-ECU communication (using e.g. FlexRay, CAN, LIN, etc.). The main task of the RTE is to make SWC's independent from the ECU on which they are mapped. As SWC's are dependent on the type of application, the RTE has to be adapted for the specific ECU in which it serves. This is done by ECU-specific generation and configuration and leads to the fact that RTEs differentiate between different ECU's.

Basic software

Basic Software is a standardized software layer and is further divided into different layers as shown in Figure 2.2. These layers are: Services, ECU Abstraction, and Microcontroller Abstraction.

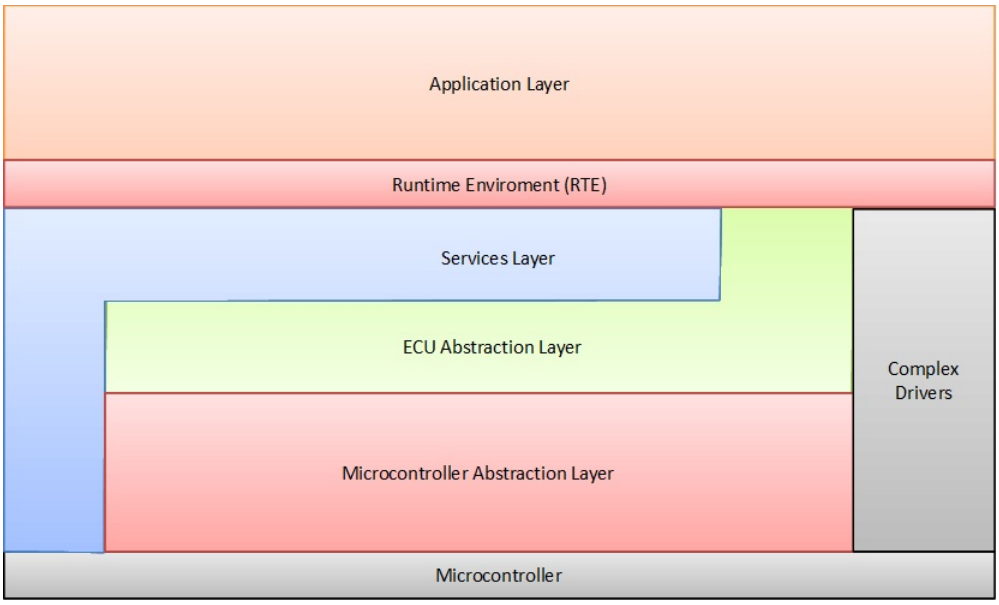


Figure 2.2: The layers of Basic Software: Services, ECU Abstraction, Microcontroller Abstraction and Complex Drivers [9].

The MCAL (MicroController Abstraction Layer) contains internal drivers and has the function of making the higher software layers independent of the microcontroller. The purpose of the ECU Abstraction Layer is to make higher software layers independent of ECU hardware layout. More specifically, it provides an API (Application Programming Interface) for devices and their connection to the microcontroller. The Service Layer is localized just underneath the RTE. It provides basic services for applications and Basic Software modules such as operating system functionality, diagnostic protocols, memory management, and communication services.

AUTOSAR communication stack

The layers of the Basic Software are further divided into modules, forming functional groups. One of these functional groups is the communication stack. The communication stack constitutes a part of the Basic Software reaching from the bottom connection with the microcontroller, to the top connection with the RTE and SWC's, see Figure 2.3.

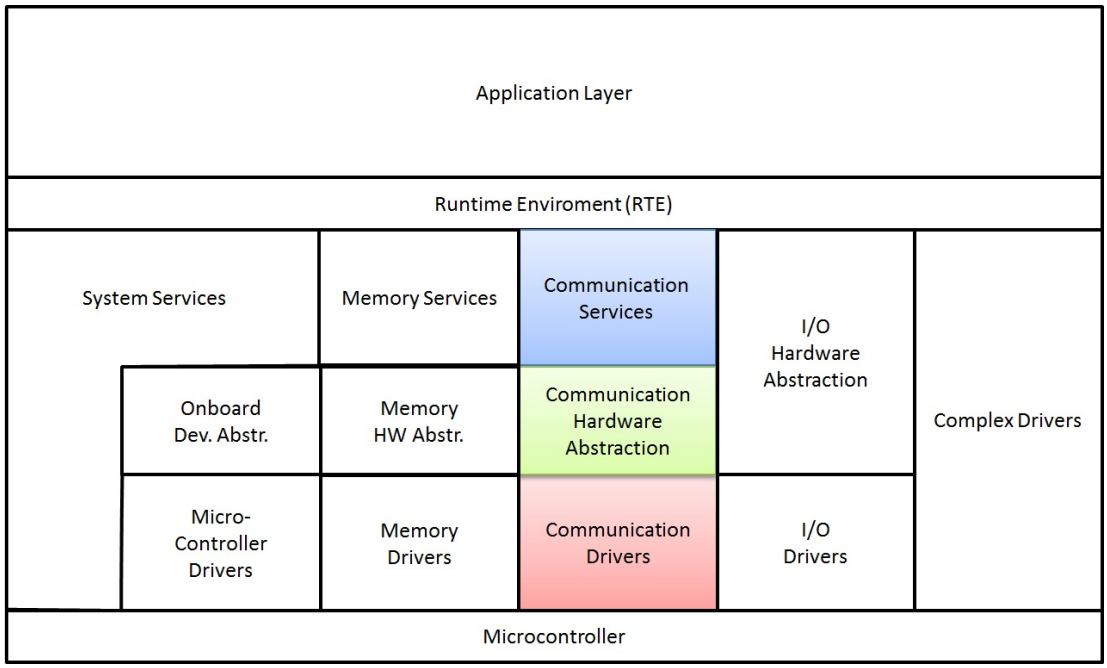


Figure 2.3: The AUTOSAR communication stack; Communication Drivers, Communication Hardware Abstraction, and Communication Services [9].

The communication stack is implemented according to the specifications of the communication protocol used, e.g. FlexRay, CAN, LIN, etc. Figure 2.4 shows the communication stack for FlexRay. One portion of the communication stack is the communication services, which consists of three of the modules important for this project, namely FlexRay State Manager, PDU Router, and AUTOSAR COM. These are also shown in Figure 2.4. Beneath the communication services lies the communication hardware abstraction layer. This is a protocol specific layer, consisting of the modules FlexRay Interface, and Driver for FlexRay Transceiver.

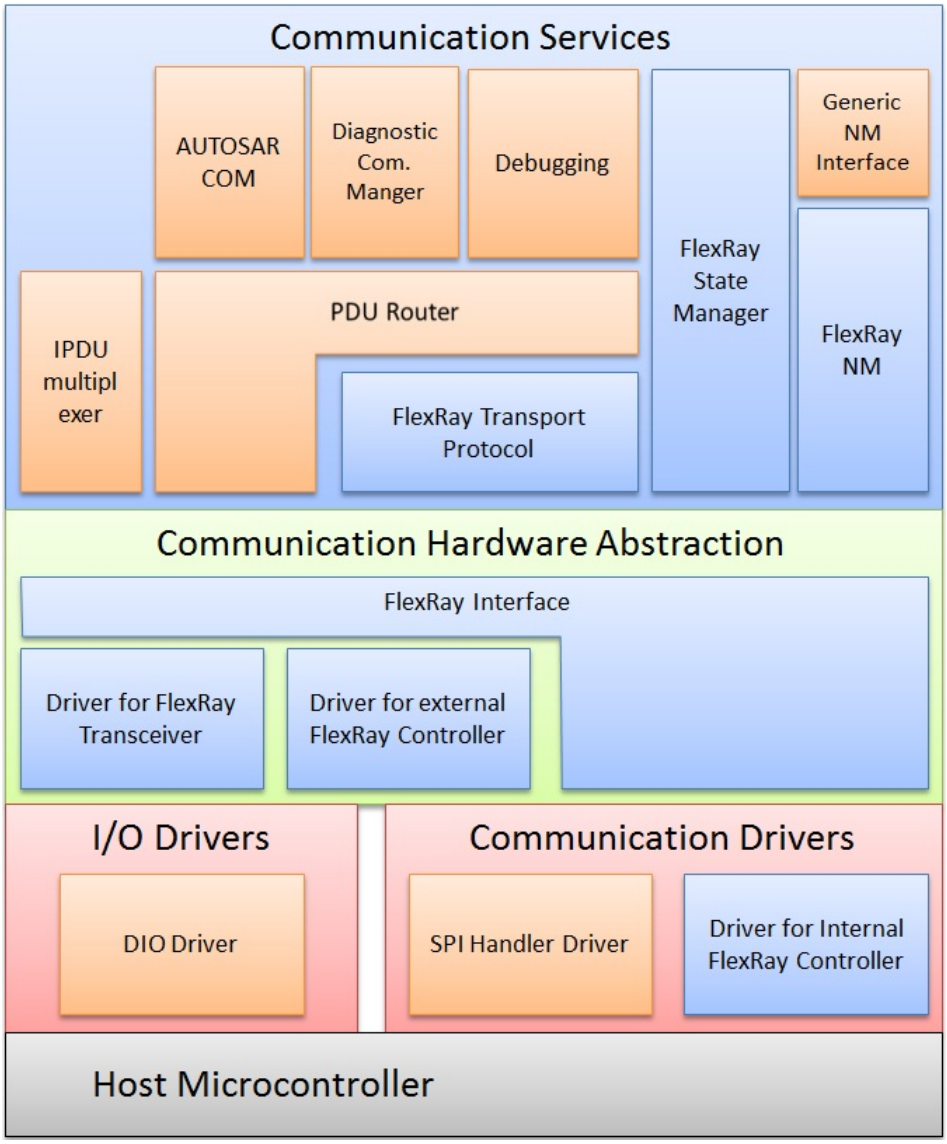


Figure 2.4: The AUTOSAR communication stack for FlexRay [9].

A message between the various modules is named differently depending on which module it is sent from. A PDU (Protocol Data Unit) refers to a data unit that is defined in the protocol for a certain module. It contains data of that module, SDU (Service Data Unit), with protocol information added. Thus, an SDU as seen from one module, is a PDU from a module localized above in the communication stack. The definitions of the different PDU's are seen in Table 2.1.

FlexRay Driver

The FlexRay Driver abstracts FlexRay CC's (Communication Controller), which is compliant to both the latest and earlier FlexRay specifications. All properties of a specific CC are made

Table 2.1: The definitions of the different PDU's [9].

PDU Definition	Description
I-PDU	PDU of an upper layer module, e.g COM, DCM etc.
L-PDU	PDU of the FlexRay Interface module
N-PDU	PDU of the FlexRay Transport Layer. It contains address information, protocol information and data (N-SDU)

available through the FlexRay Driver module. Since different CC's offers different hardware implementation features, a single FlexRay Driver module supports only one separate type of a FlexRay CC [10].

FlexRay Interface

The purpose of the FlexRay Interface module is to provide a generally specified interface to the communication system for the upper layer modules, e.g. PDU Router and COM. However, this is limited to the transmission of data. The configuration of the Interface module depends on the communication bus. Since the configuration relies on specific features of the communication system.

The FlexRay Interface does not access the hardware directly. It uses one or several FlexRay Driver modules to get access to the FlexRay CC(s). The same applies for the FlexRay Interface when accessing the FlexRay Transceiver(s), by using one or several FlexRay Transceiver Driver module(s) [11].

FlexRay Transport Protocol

This module is localized just above the FlexRay Interface and underneath the PDU Router. The object with the FlexRay Transport Protocol is to segment and perform reassembly of PDU's that are too large to fit in one L-PDU [12]. However, it is not required to use the FlexRay Transport Protocol in order to have a functioning AUTOSAR communication stack. This holds as long as the size of a message fits in a FlexRay frame, which can be up to 127 words or 254 bytes. The FlexRay Transport Protocol is not used within the scope of this thesis.

PDU Router

The function of the PDU router is to statically route I-PDU's based on the I-PDU identifier. Thus, there is no occurrence of dynamic routing during run-time. Since the PDU router is generally specified [13], it serves in the same way irrespective of which communication protocol that is used, except for the name of the call-back functions used to the surrounding modules. The PDU router uses the modules COM, FlexRay Interface and FlexRay Transport Protocol for providing its services.

AUTOSAR COM

Just as the PDU Router, the COM module has the same design independent of which communication protocol that is used [14]. Its main function is to provide an interface to the RTE, packing and unpacking AUTOSAR signals into I-PDU's.

FlexRay State Manager

The function of the FlexRay State Manager is mainly to provide an abstraction layer to the AUTOSAR Communication Manager module [15]. Since the Communication Manager controls the states of all communication channels bounded to the ECU, and the fact that the module must be compliant with all type of AUTOSAR communication protocols, an abstraction layer is needed. Together, the FlexRay State Manager and the FlexRay Communication Manager handles the process of starting up and shutting down the communication of a FlexRay cluster. In turn, the FlexRay State Manager does not access the FlexRay hardware directly, but rather through the FlexRay Interface module.

2.2 FlexRay

FlexRay is a recent protocol for vehicle network communication. It was developed by the FlexRay Consortium from 2000 until 2010. The FlexRay Consortium does no longer exist, but the specifications of the latest release (version 3.0.1) are still available for download [16]. Core companies included in the FlexRay Consortium were BMW, Bosch, Daimler, Freescale, General Motors, NXP Semiconductors and Volkswagen. Because of increased demands within the vehicle areas: fuel efficiency, comfort and safety, more calculations and communication are needed. For a network system to be able to efficiently handle these demands, a faster and more reliable network protocol is required. FlexRay was developed to meet todays and future requirements.

2.2.1 Comparison with existing protocols

The CAN protocol is one of the standard protocols in the vehicle industry [3]. It was officially released in 1986 and was introduced to the market the year after. Today, CAN is the dominating communication protocol in vehicles, and is also used to manage operating rooms in hospitals, to control devices such as lights, tables, x-ray machines and patient beds. Other areas of use are laboratory equipment, sports cameras, automatic doors and coffee machines [17]. There are several advantages of the CAN protocol which have contributed to its success. CAN is inexpensive in the way that it provides the condition for ECU's to rely on only one common interface to communicate with other devices in the system. Each device in the system is considered to be smart. Since each device has a CAN controller, all devices see the transmitted messages and can decide for themselves if a specific message is relevant or if it should be disregarded. Messages are sent on the common bus with different priorities. Thus, a message with higher priority will be sent first while a message with lower priority will be postponed [17].

Another widely used standardized protocol is the LIN protocol. It was first released in 1999 by the LIN-Consortium, which expired in 2010. LIN is a single-wired serial communications protocol and is organized as one master and several slaves (up to 16 slaves). Nodes acting as slaves in a LIN network determines the master node, and do not bother about the remaining network configuration. The messages from the master device are broadcasted out on the network and there is no mechanism for collision detection. Advantages with the LIN protocol are that it is easy to use and it is cheaper than other protocols, for example CAN. LIN is not seen as a full replacement for CAN, but rather a good complementary to CAN where cost is prioritized over bandwidth [4].

The TTCAN protocol was first developed by Bosch in 2003 [5], with the aim to add synchronization between nodes to the already existing CAN protocol. Thus, its main purpose is to reduce latency jitter, which may be present in a CAN network even for nodes with high priorities. This can occur if another message is already in process for transmission or if some other message has higher priority also tries to get access to the bus. TTCAN serves as an additional layer to CAN and brings the ability of combining both time-triggered and event-triggered communication. An advantage is that it can easily be modified to be used on already existing CAN systems. A considerable drawback is that it lacks in reliability [18].

The aims with FlexRay are to provide communication that achieves high data rates up to 10 Mbit/s. It provides both time-triggered and event-triggered communication in comparison to CAN, which is an event-triggered based protocol. Also, it aims to provide a more fault-tolerant and redundant architecture [19]. Figure 2.5 illustrates the comparison between FlexRay, CAN, TTCAN and LIN in their relation to cost and bandwidth. FlexRay came in use for the first time in 2006 in BMW's X5 car, with FlexRay version 1.1. Several more car models have been equipped with FlexRay during the last couple of years, such as BMW 5 and 7-series, Audi A8, Bentley Mulsanne and Rolls-Royce Ghost [20].

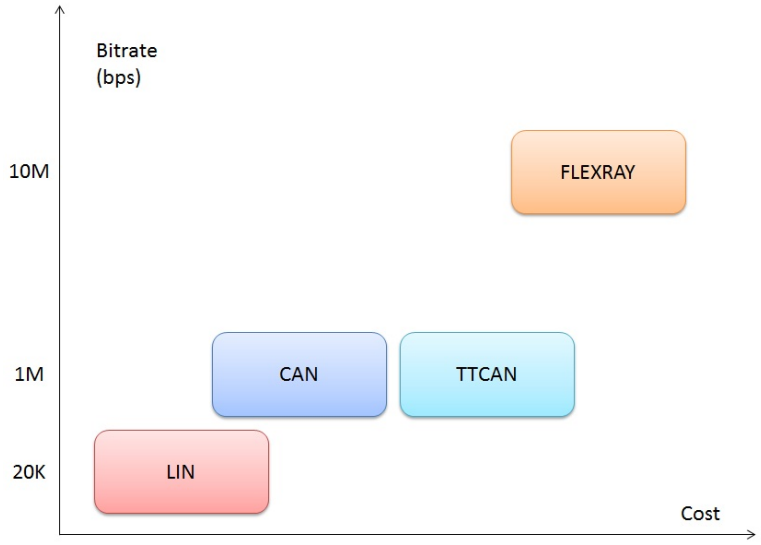


Figure 2.5: The difference between LIN, CAN, TTCAN and FlexRay in their relation to cost and bandwidth [21].

2.2.2 FlexRay communication

FlexRay provides both time-triggered and event-triggered communication in comparison to CAN, which is an event-triggered based protocol where nodes are allowed to make bus access depending on the priority-level for the event. With FlexRay, the communication between nodes must be precisely defined. This is achieved by allocating a specific time slot for a message within a communication cycle according to the TDMA (Time Division Multiple Access) method. A communication cycle is divided into a static segment, a dynamic segment, a symbol window, and a NIT (Network Idle Time) as shown in Figure 2.6.

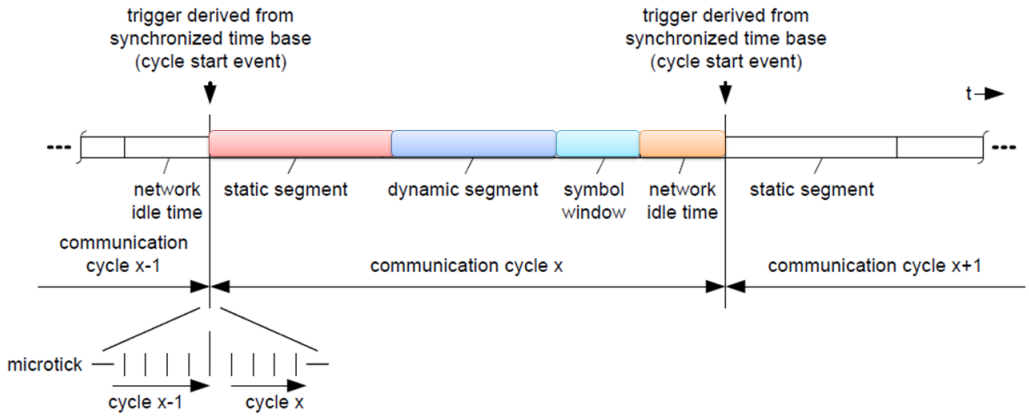


Figure 2.6: The different segments of a FlexRay communication cycle [22].

Static segment

The static segment consists of statically configured time slots of equal length, thus each static slot consists of the same number of macroticks. A macrotick is a FlexRay timing unit, for further details see Section *Timing Hierarchy*. The number of macroticks per time slot is defined by a global variable for all participating nodes in the network. Time slots can be of different types, either a key slot or a non-key slot. A key slot is used by a node to transmit sync and startup frames. One or more key slots can exist in the static segment. Time slots that are non-key slots are used for transmission of frames on either one channel or both.

Each node uses two slot counters, one for each channel, to facilitate the scheduling of transmissions. At the start of a communication cycle the slot counters are set to the value one, and is incremented at the end boundary of each time slot. The number of static time slots are the same for all nodes in the network. These precisely defined timing characteristics of the static segment allows time-triggered communication between nodes. Since each frame transmission is defined to occur during an exact point of time, all participating nodes involved in that transmission have knowledge about when the transmission and reception of a frame will take place.

Dynamic segment

The dynamic segment allows event-triggered communication between FlexRay nodes. A communication slot in the dynamic segment is called a minislot. In comparison to the equal length of transmitted frames in the static segment, a transmitted frame in the dynamic segment may be of different length. However, minislots in the dynamic segment contains an identical number of macroticks defined by a global variable for the network, just as for the number of minislots. There exist no sync or startup frames in the dynamic segment.

Another feature within the dynamic segment is in the process of scheduling transmissions. A node uses one slot counter for each channel. They may, however, be incremented independently in compliance with the dynamic procedure of transmitting frames of different lengths. If no transmission takes place, the dynamic slot transmission phase consists of one minislot. If there is an ongoing transmission, then several minislots may compose a dynamic slot. The length of a dynamic slot depends on the transmitted frame size. The communication behavior in the dynamic segment can be described as asynchronous in which some frames are prioritized over others. This means that a node can use full bandwidth for a transmission, but it also means that frames with lower priority may not be transmitted within the current communication cycle.

Symbol window

The symbol window can be used for transmission of different symbols, for example a start up or a wake up symbol. A wake up symbol is used as a power management tool to wake up a node which is currently in sleep mode. The symbol window is not required in a FlexRay communication cycle.

Network idle time

The NIT is a phase that is used for calculation of clock divergence and clock correction between the nodes in the network. Also, other tasks are performed during this phase such as error handling and updating counters.

Clock synchronization

Since FlexRay is a time-triggered communication protocol all nodes in the network must have the same view of the time. This is needed because all nodes need to know when to send and when to receive data on the bus. This common view of time is called global time. As all nodes have their own clocks with different clock skew and offset, they have their own interpretation of the global time. This is why FlexRay needs a clock synchronization algorithm. The following text will introduce and explain how FlexRay handles time and clock synchronization [22].

Timing Hierarchy

FlexRay handles time in three different levels; the communication level, the macrotick level and the microtick level as illustrated in Figure 2.7. Microtick is the smallest timing unit, and the length of a microtick is defined as x number of clock ticks on the CC's oscillator. The value of x is different on each node since they have different rates on their oscillators.

The next level in the timing hierarchy is the macrotick, which has a length of y number of microticks. This structure is used by the clock synchronization mechanism to be able to make the time go faster or slower on the local node, this is described further in the next part. The macrotick is the smallest unit that the nodes try to synchronize with.

With a cluster that has perfectly synchronized clocks, the nodes have exactly the same number of macroticks within the third timing level, the communication cycle. A communication cycle is built up by z number of macroticks, and if the clock synchronization is correct, all nodes in a cluster shall have the same cycle number at any given time.

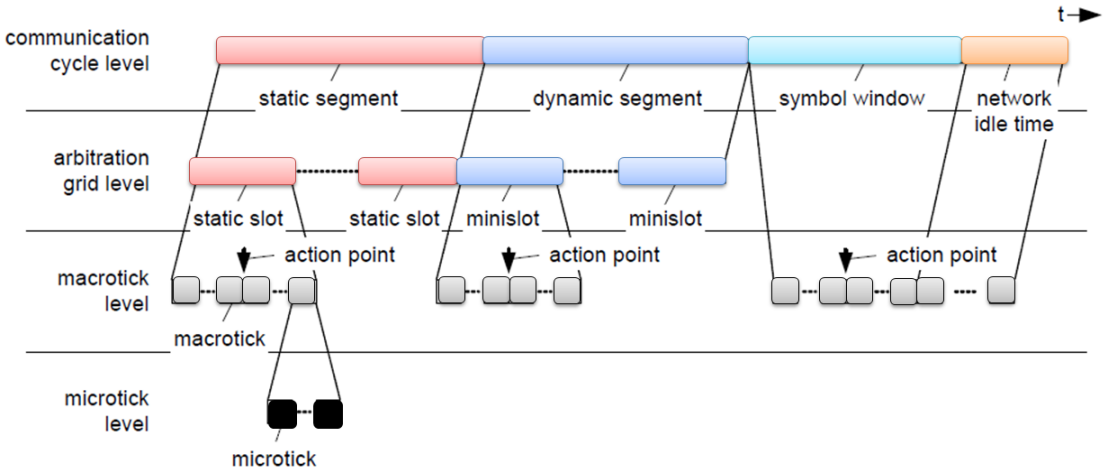


Figure 2.7: The timing hierarchy in FlexRay communication [22].

Clock calculation and correction

The global time is not a time taken from a real clock, the global time is represented by visible events sent on the FlexRay bus. It is with these events the nodes can get a virtual interpretation of the global time. However, with this method there must be some requirements fulfilled to make it work. The majority of all local clocks need to act in a correct way. A correct local clock is a clock that does not deviate too much from all other clocks. The clock of one node shall not drift more than 0.15% macroticks from the global time. This means that the difference between the slowest and the fastest clock cannot be more than 0.3% macroticks.

The event that is used for the time measurements is called sync frames. These sync frames are sent from several nodes that are selected as sync nodes. When a node receive a sync frame it compares the time of the arrival with the time of the expected arrival time. The

deviation of these times is stored for every cycle. The clock is corrected in two ways, rate and offset correction. Offset correction is done by scheduling the next execution earlier or later compared to what otherwise would have been the case. This is accomplished by adding or removing macroticks from the network idle time. To make the local clocks on the nodes "tick" in the same rate, microticks are added or removed within a macrotick. By doing this the node can correct its own clock without affecting the number of macroticks within a cycle, but still make the macroticks longer or shorter according to their own local time.

2.2.3 Frame format

The FlexRay frame is composed of three segments, the Header segment, the Payload segment and the Trailer segment, see Figure 2.8

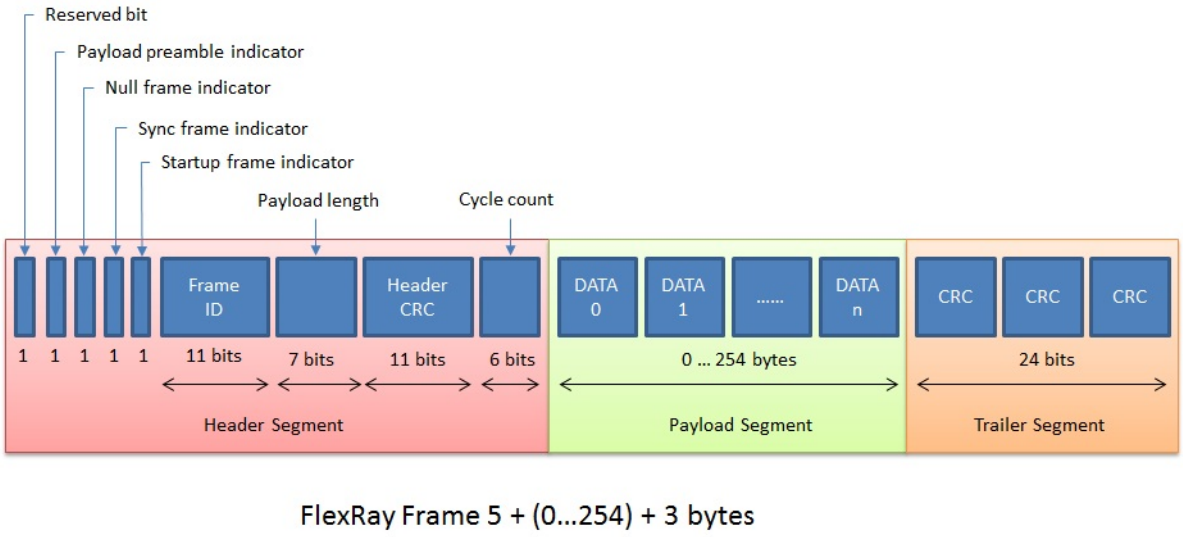


Figure 2.8: The segments and fields of a FlexRay frame.

The frame is transmitted on the network with the Header segment first, followed by the Payload segment and last the Trailer segment.

Header segment

The header segment is composed of nine different fields, taking up five bytes of the frame altogether. The first five fields in the header segment, consisting of one bit each, are the reserved bit, the payload preamble indicator, the null frame indicator, the sync frame indicator and the startup frame indicator.

The frame ID indicates the slot in which the frame should be transmitted and ranges from 0 to 2047, where a value of 0 indicates an invalid frame ID. A frame ID is used only once on each channel in a communication cycle.

The payload length defines the size of the payload segment. Since only seven bits are used for this field it has the payload size value divided by two, since the payload may consist of as much as 254 bytes.

The header CRC (Cyclic Redundancy Check) contains error-detecting code that uses the fields: the sync frame indicator, the startup frame indicator, the frame ID, and the payload length to compute the CRC.

The Cycle count contains the value of the cycle counter as seen by the transmitting node.

Payload segment

The Payload segment consists of 254 bytes of data. It is organized as 0 to 127 two-byte words, as indicated by the Payload length field, and therefore contains an even number of bytes. The first byte is identified as "Data0", the second "Data1", etc., increasing the number for each individual byte.

Trailer segment

The Trailer segment consists of one 24 bit CRC field. The CRC is computed over all the fields within the header segment and the payload segment of the frame.

Chapter 3

Development methods

This chapter describes the hardware and software tools used in this project. Among these tools are the ODEEP platform with the on-chip FlexRay controller, flash and debugging devices and the Eclipse platform with a debugger plug-in. The tools were accessible from the very beginning of the project which facilitated the execution of the initial tasks in the working process. All software used was executed on Windows XP and Windows 7 OS (Operating System).

3.1 ODEEP

ODEEP is the hardware platform used for testing and development. This platform is the only hardware targeted in this thesis and was provided by QRTECH AB. The platform has many features like CAN, LIN, Ethernet, I/O, and Micro SD-Card interface. The most important feature for this work is the two FlexRay interfaces that the ODEEP provides [8]. The ODEEP platform has a processor that is named MPC5567 and is produced by Freescale [23].

3.1.1 FlexRay configurator

To get FlexRay to work it is required to be able to set many parameters. This tool provides a graphical interface to be able to configure all parameters in an easy way. The tool generates data structures for the AUTOSAR BSW modules the FlexRay Driver and the FlexRay Interface. The data structures are then used when the code is compiled to give the nodes the configured parameters [24].

3.2 Development environment

3.2.1 Software environment

The Arctic Studio open source development platform [25] with the Eclipse IDE for C/C++ Developers plug-in were used for code development. The Arctic Studio is based on the Eclipse platform. Therefore plug-ins for Eclipse are compatible with the Arctic Studio as well. The C/C++ Developers plug-in includes additional functionality to the platform such as an editor with syntax highlighting and code completion for C/C++, a debugger, a search engine, and a makefile generator. This environment is available to the public by the Eclipse Foundation via their homepage [26]. The hardware debugger, called GDB (GNU DeBugger) debugger, is included in this plug-in and was used to trace the execution of different configurations on the ODEEP platform. The GDB Debugger allows a user to modify the value of program variables and call functions independently of the predefined execution of the program.

3.2.2 Hardware environment

This subsection explains which hardware components that were used and how they work. After reading this section it should be rather easy to be able the setup the system that was used in this master thesis. Components used were:

- ODEEP QR5567 (hardware platform)
- FlexRay Channels (two twisted pair cables)
- USB Qorivva MPC55xx/56xx Multlink (named multilink) [27]
- PEEDI (Powerful Embedded Ethernet Debug Interface) [28]
- USB cables
- Ethernet cable
- PC

As mentioned before, the ODEEP was used as a platform, and it is mostly the environment surrounding ODEEP that is interesting. Between the ODEEP platforms there are two FlexRay channels, each channel is a twisted pair cable. Here all communication between the ODEEP platforms was done. Of course, it is possible to extend the bus and add more nodes. The USB cable between the nodes and the PC's was used to control the nodes. It is also here the nodes send information and printouts. With the configuration explained so far it is possible to have FlexRay communication and be able to control the nodes. To be able to load new software on the nodes two different flash programmers have been used. Two flash programmers were used because it was only possible to debug with the PEEDI flasher. Both of them are connected to the nodes with a JTAG interface but they have different connections to the PC's. The PEEDI is connected through an Ethernet cable and can be accessed via a Telnet session. With this connection every PC on the local network can flash the connected node. The Multilink flasher was connected to a PC via a regular USB and the program eSys Flasher [23] was used to flash

the nodes. Figure 3.1 describes graphically how the environment was setup.

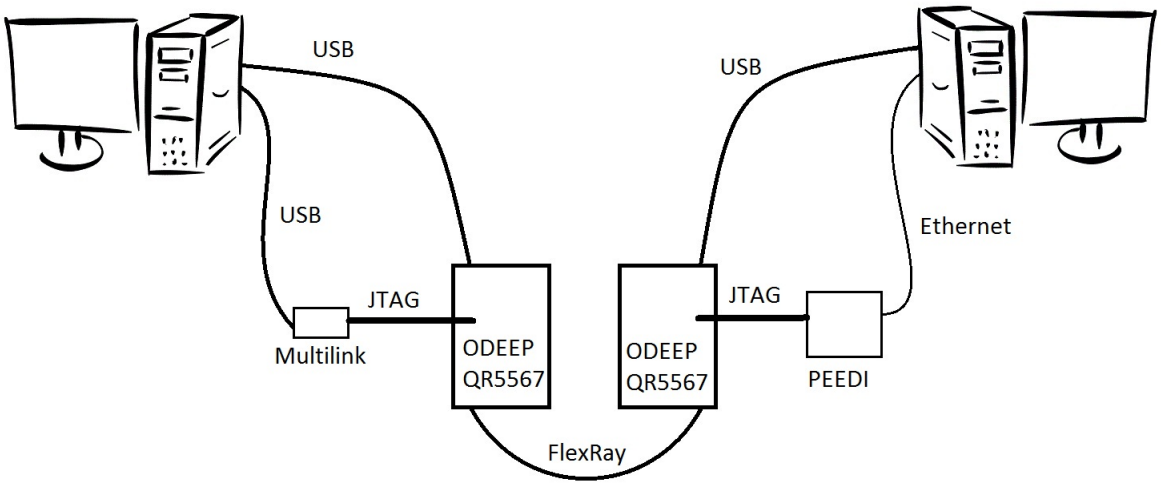


Figure 3.1: The hardware environment setup.

Chapter 4

Implementation

This chapter describes the implementation of the different AUTOSAR modules. It covers relevant data structures, functions, and included components in the Arctic Studio tools.

4.1 FlexRay Interface

The implementation of this module was based on the AUTOSAR FlexRay Interface specification [11]. As previously mentioned, the FlexRay Interface module communicates with the FlexRay controller(s) indirectly through the FlexRay Driver module. It provides the following features for the upper layer AUTOSAR BSW modules [11]:

- initialization
- data transmission (sending and reception)
- start/halt/abort communication
- FlexRay specific functions (e.g. to send a wake-up pattern)
- set operation mode
- get status information
- various timer functions

4.1.1 Interaction with other modules

The FlexRay Interface achieves abstraction of the CC(s) and Driver(s) to the upper layer modules, by providing an abstract and unique index for each type of resource. This holds for all resources independent of what type of resource it is, where the resource is localized and how it is accessed. For example, the FlexRay Interface may provide a function pointer to an equivalent Driver's API service. Such a function pointer is localized in a static configuration table, called container. So, whenever an upper layer BSW module passes an abstract index

to the FlexRay Interface, it retrieves the function pointer from a container. It then calls the corresponding lower layer BSW module's API service via the function pointer and forwards the translated index in the API call. This mechanism is further described in the subsection 4.1.2 about controller initialization.

FlexRay Transceiver Driver

The different states of a FlexRay Transceiver are controlled by the FlexRay Interface via the FlexRay Transceiver Driver [29]. This is done by calling the function `FrTrcv_SetTransceiverMode`. The function call is initially done by FlexRay State Manager. A FlexRay Transceiver module of AUTOSAR version 2 was available at the start of this project. It is compatible with a newer FlexRay Interface module. Thus, no new implementation of the FlexRay Transceiver Driver was needed.

FlexRay State Manager

As mentioned before, the FlexRay State Manager handles the startup and shutdown of the communication of a FlexRay cluster. This is done through the resources which FlexRay Interface offers to the upper layer modules. Since the FlexRay State Manager did not exist in AUTOSAR version 2.0, an implementation of the basic parts of FlexRay State Manager was done during this project.

The most fundamental structures in the FlexRay State Manager are the state machines. There is one state machine for each FlexRay cluster. The different states are mapped to the POC (Protocol Operation Control) states for a specific CC. Examples of some of the states are `FRSM_READY`, `FRSM_WAKEUP`, `FRSM_STARTUP` and `FRSM_ONLINE`. For further explanations of the state machine see specification [15].

4.1.2 Main function and initialization

The main function of the FlexRay Interface, `FrIf_MainFunction_<ClstIdx>`, is called once for each loop of a certain task. The time interval between the Main Function calls depends on the length of the FlexRay cycle. Because the FlexRay cycle length may differentiate in its configuration between different FlexRay clusters, the Main Function calling period may be of different length between two FlexRay clusters. The task, responsible of calling the Main Function, is defined in the BSW Scheduler module. There must exist one Main Function for each FlexRay cluster. Since the use of several FlexRay clusters is outside the scope of this thesis, only one Main Function was implemented. The purpose of the Main Function is to monitor and control the FlexRay Job List Execution Function, which is further described in the upcoming subsection 4.1.4.

Initialization of the FlexRay Interface is launched by the AUTOSAR ECU State Manager by calling the function `FrIf_Init`. The State Manager passes a pointer to the address of the static configuration structure of the FlexRay Interface module. `FrIf_Init` makes sure that the configuration is made available to all functions within the FlexRay Interface module. It

also initializes the local memory space used for storing PDU data, PDU properties and state variables for the FlexRay Interface State Machine.

Controller initialization is done through the function `FrIf_ControllerInit`. This function is a typical example of how FlexRay Interface wraps the corresponding FlexRay Driver API function. The mechanism performs the following two steps:

1. Translation of the static configuration of the FlexRay CC index, to a FlexRay Driver specific CC index.
2. Based on the retrieved CC index, find the appropriate FlexRay Driver and call the corresponding FlexRay Driver API function.

The mechanisms for all FlexRay Driver wrapping functions are similar to this example. The difference between them lies in what resource data type is being retrieved. For a better understanding of how this is performed, the code for `FrIf_ControllerInit` is presented in Listing 4.1.

```
1  /* Pointer to main configuration, set by FrIf_Init */
2  static FrIf_ConfigType const *FrIf_ConfigCachePtr;
3
4  Std_ReturnType FrIf_ControllerInit(uint8 FrIf_CtrlIdx) {
5
6      FrIf_IdxConfigType const *MyIdxPtr;
7
8      /* 1) Translate to Driver Idx using FrIf Idx */
9      MyIdxPtr = &(FrIf_ConfigCachePtr->FrIf_IdxConfigPtr[FrIf_CtrlIdx]);
10
11     /* 2) Call corresponding Driver Init-function, passing correct config struct
           */
12     Fr_ControllerInit(MyIdxPtr->FrCtrlIdx, MyIdxPtr->FrLowLevelConfSetIdx,
           MyIdxPtr->FrBufConfSetIdx);
13
14 }
```

Listing 4.1: Part of the code for the function `FrIf_ControllerInit`

4.1.3 FlexRay Job List

The Job List data structure is simply a list containing the predefined FlexRay communication jobs. The different jobs are sorted after their execution start time. Each job is scheduled according to which FlexRay cycle and the macrotick offset within a FlexRay cycle, in which the job should start its execution. A job may have one or several Communication Operations. Table 4.1 shows the different Communication Operations available in the AUTOSAR release 4.0.

FrIf_JobListExec_<ClstIdx>

This function is rather to be seen as an ISR (Interrupt Service Routine), linked to the FlexRay CC. This holds if the CC does not guarantee an asynchronous buffer access. If the CC can

Table 4.1: The different Communication Operations.

Communication Operation	**Description**
DECOUPLED_TRANSMISSION	Decoupled Transmission
PREPARE_LPDU	Prepare message buffer of CC
RECEIVE_AND_INDICATE	Immediate reception
RECEIVE_AND_STORE	Decoupled reception
RX_INDICATION	Reception indication
TX_CONFIRMATION	Transmission confirmation

guarantee asynchronous buffer access, the Job List Execution Function can be executed in a regular OS task.

Just as the Main Function, there must be one Job List Execution Function for each FlexRay cluster. The main task for the Job List Execution Function is to execute the configured Jobs, defined in the FlexRay cluster's Job List, at specific points in time. Thus, the execution of jobs must be synchronized with the global time of the FlexRay network.

4.1.4 Data transmission

Three modules are active and interact with each other during a data transmission. These are the FlexRay Interface, the upper layer BSW module, which is PDU Router in this case, and the lower layer BSW module, the FlexRay Driver.

`FrIf_Transmit`

Whenever the upper layer BSW module wants to request a transmission of a specific PDU, it calls the `FrIf_Transmit` function. The function call passes the unique PDU ID together with a pointer to the data of the PDU. `FrIf_Transmit` then accomplishes one of two things depending on the configuration of the PDU. The PDU can be configured for either decoupled transmission or immediate transmission.

If a decoupled transmission is desired, the PDU is not yet passed to the underlying FlexRay Driver. Instead, the FlexRay Interface remembers the PDU's transmission request by incrementing a counter. This means that the upper layer BSW module may call `FrIf_Transmit` whenever desired, asynchronously of the FlexRay communication system. It also implies that the upper layer BSW module must buffer the PDU data and be able to copy the data to the right location when a transmission interrupt has been obtained.

When the PDU has been configured for immediate transmission, the FlexRay Interface simply passes the PDU to the FlexRay Driver module at once. However, one requirement must be fulfilled in order to use an immediate transmission. The upper layer modules must be synchronized to the FlexRay communication schedule, so that when a transmission is desired, the data will be available during the intended time slot. For such a setup, the overhead of decoupled transmission in FlexRay Interface is not needed.

Transmit with decoupled buffer access

FlexRay Interface supports decoupled transmission by queuing transmit requests from the upper layer BSW module. This is done when `FrIf_Transmit` has been called. The remainder of the communication operation execution is done by the Job List Execution Function.

Since a FlexRay frame may hold several PDU's, the Job List Execution Function firstly iterates over these to check whether it is queued and ready for transmission. If that is the case, a call to the upper layer's `<UL>_TriggerTransmit` function is done. The PDU ID is sent in the call together with a pointer to a temporary buffer within FlexRay Interface for collecting the L-SDU. When appropriate counters and flags have been set, the FlexRay Driver function `FR_TransmitLPdu` is called for the current L-PDU. If this function would return an indication that the transmission failed, counter values must be restored to their previous value. Otherwise the transmission is seen as successful, from FlexRay Interface's point of view, and the Communication Operation is finished. The flow chart for decoupled transmission is shown in Figure A.2 in Appendix A.

Transmit with immediate buffer access

Immediate transmission is performed within the FrIf_Transmit function, which is called by the upper layer BSW module. This means that the PDU is directly passed to a buffer handled by FlexRay Driver through the call to `FR_TransmitLPdu`. The flow chart for immediate transmission is shown in Figure A.1 in Appendix A.

Transmit confirmation

The Communication Operation Transmit Confirmation is performed by the Job List Execution Function. Initially it calls the FlexRay Driver API function `Fr_CheckTxLPduStatus` to check if the transmission for a PDU was carried out. When the check is done, it finds the appropriate PDU in the FlexRay frame. If transmission confirmation for this PDU hasn't been registered, it calls the upper layer BSW module's confirmation function and marks the PDU as confirmed. The flow chart for transmit confirmation is shown in Figure A.3 in Appendix A.

4.1.5 Data reception

The FlexRay Interface communicates directly with the upper layer module, the PDU Router, and the lower layer module, FlexRay Driver, during data reception. There are two types of reception methods, "Receive and Indicate" and "Receive and Store". Both of these methods are treated below.

Three parameters are derived, for both reception methods, in order to be able to receive the correct data from the correct FlexRay Driver. These are the FlexRay controller index, so that the Driver knows which controller is receiving the data, the L-PDU index, to identify the PDU and a pointer to a temporary buffer for storing the PDU data.

Receive and indicate

This mechanism is constructed to indicate to the upper layer module that the FlexRay Driver has received data. This is done in the following way: for a specific FlexRay frame, a loop is started to iterate over its containing PDU's. For each PDU, a check is done whether it is updated or not, in other words if any new data has been received. If no new data are available, then the next PDU in the frame is handled. Otherwise, a call is made to the upper layer module function <UL>_RxIndication, with the PDU ID and a pointer to the PDU data. In this way the PDU is fetched upwards in the software stack. The Receive and Indicate communication operation is illustrated in Figure A.4 in Appendix A.

Receive and store

This reception mechanism is executed in almost the same manner as for Receive and Indicate. The difference, is that instead of calling the upper layer, it directly stores the PDU data in a FlexRay Interface specific buffer and marks it as up-to-date. This is further illustrated in Figure A.5 in Appendix A.

Provide Rx indication

The communication operation Provide Rx Indication is a complement to the Receive and Store mechanism, see Figure A.6 in Appendix A. It simply calls the upper layer module function, <UL>_RxIndication, for a specific PDU contained in a FlexRay Interface PDU buffer. After the indication to the upper layer module is performed, the buffer is marked as outdated.

Summing up data reception for FlexRay Interface, either the communication operation Receive and Indicate can be scheduled, or the Receive and Store together with Provide Rx Indication. The advantage of using the latter, is that the indication to the upper layer module can be scheduled to be performed an arbitrary time after the reception is taking place. This is useful if for example the ECU has short of time after a receive. Then the Rx indication operation could be scheduled when the ECU is idle and have no other critical communication operations.

4.2 Adaptation of considered modules

According to AUTOSAR it should be possible to change or replace modules to be able to reuse parts [6]. This is one of the key features in AUTOSAR and in a perfect "AUTOSAR world" this chapter should not be needed. Nevertheless, two kinds of problems emerge when integrating the FlexRay modules into the Arctic Studio environment. One of them is that the modules in Arctic Studio is not complete, they have no support for FlexRay. The second problem is that the modules are of different AUTOSAR release versions. These two problems make the AUTOSAR environment to a non-perfect "AUTOSAR world" and this is why this chapter is necessary. The two following subsections are about how this was solved.

4.2.1 The connection between Arctic Studio modules and FlexRay modules

The only module of the Arctic Studio environment that is connected to the FlexRay modules is the PDU Router. This is the point where the problem emerges between modules from the different environments. The reason for this problem is due to the lack of support of FlexRay in the PDU Router module. What the PDU Router needed was the functionality for the FlexRay Interface module to be able to connect to it. This is done through three different callback functions. Since the PDU Router already had all other functionality implemented it was only these three callback functions and some routing cases needed to be added. With these functions implemented PDU's can travel all the way through the COM, the PDU Router, the FlexRay Interface, the FlexRay Driver, and finally on to the bus and vice versa.

4.2.2 Problems with modules of different releases of AUTOSAR

To have modules of different release versions is possible, but some changes might be needed to get the system up and running. A good feature with AUTOSAR is the modularity property. This means that it is only needed to control the modules that communicate with each other and have different versions. For example, if the FlexRay Interface has version 2.0 and the PDU Router and the COM has 3.1 it is only the connection between the PDU Router and the FlexRay Interface that needs to be controlled. This is because the COM communicates with FlexRay Interface through PDU Router. Of course, if there are new features in newer versions of FlexRay Interface these cannot be used by the other modules. Also if there have been some major changes between the functionality in modules between versions that affects the whole stack they may not work with older versions. This was however not the case in this thesis. The two scenarios that were handled in this thesis are described in the sections below.

Version 3.1 vs version 2.0

In the beginning of the thesis the modules used were implemented with different AUTOSAR releases as references. This resulted in two groups of modules. The modules from Arctic Studio are of release 3.1 version and the FlexRay modules are of release 2.0 version. However, the property that the two groups only communicates between each other through the PDU Router and the FlexRay Interface, and the fact that the communication only uses a limited number of functions, made it easy to control what information the groups expected from each other. For the groups to be able to communicate some "glue code" was needed because the two groups expected data of different types. Figure 4.1 depicts how the modules are related and what releases they were based on at the beginning of this thesis.

Version 3.1 vs version 4.0 vs version 2.0

After implementing the FlexRay Interface another scenario emerged. The Arctic Core modules were still written as AUTOSAR 3.1 but the FlexRay Interface was upgraded to AUTOSAR 4.0 and the FlexRay driver was still AUTOSAR 2.0. The "glue code" between the PDU Router

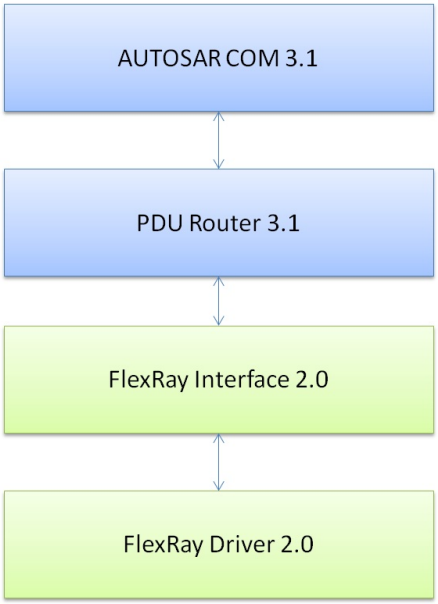


Figure 4.1: The AUTOSAR FlexRay communication stack at the beginning of the thesis

and FlexRay Interface that was mentioned in previous section could be removed. Since the connection between the PDU Router 3.1 and the FlexRay Interface was compatible. However, there was instead a problem between the FlexRay Interface and the FlexRay Driver. This problem was of the same type as in the previous case between the PDU Router and the FlexRay Interface and could easily be solved with some "glue code". Figure 4.2 depicts the AUTOSAR FlexRay stack with the FlexRay Interface updated to release 4.0.

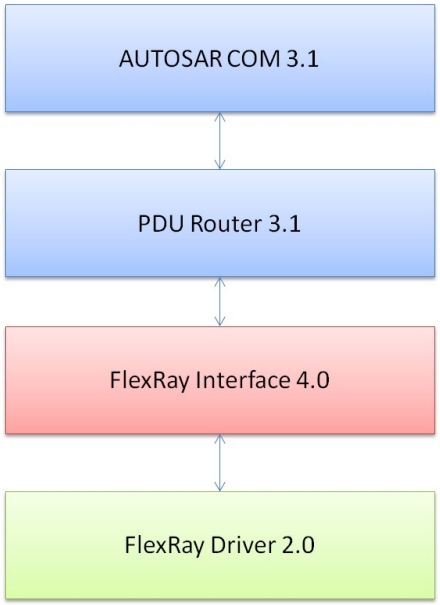


Figure 4.2: The AUTOSAR Flexray communication stack with updated FlexRay Interface

4.3 Code generator

The QRTECH ODEEP FlexRay Configurator only has support for generating the configuration files for the FlexRay Interface version 2.0. In order to configure version 4.0 of the FlexRay Interface in an efficient way, a code generator was implemented.

The code generator was implemented in Java [30]. It reads an XML (eXtensible Markup Language) file, where the different XML elements are composed of parameters for a specific FlexRay node. An example of some FlexRay Interface parameters defined in an XML file is presented in Listing B.1 in Appendix B. For each XML element containing child elements, a Java class was created constituting the adherent parameters as class variables. Once all parameter values was assigned to the appropriate class variables, a configuration file was created with C syntax and constructed to be compliant with the FlexRay Interface module. A part of a C configuration file is also presented in Listing B.2 in Appendix B.

Alongside with the development of the code generator, QRTECH developed an upgraded version of the QRTECH ODEEP FlexRay Configurator. Since the configuration file in C of the FlexRay Interface version 4.0 had to be constructed in a different way, the structure of the XML configuration file also had to be changed. With the new configurator and code generator working together, configuration of the FlexRay Interface version 4.0 is made in a fast and efficient manner.

Chapter 5

Configuring the development environment

This chapter provides descriptions of how the FlexRay communication was configured with the FlexRay configurator, and how the contributory AUTOSAR modules in Arctic Core were set up.

5.1 Configuration in FlexRay configurator

As mentioned before, the QRTECH ODEEP FlexRay Configurator was used to configure parameters and generate code for the FlexRay Driver and the FlexRay Interface. The configurator consists of different parts, e.g. for setting up global parameters there is a window called "Cluster Parameters". There are also windows for configuring each node separately. The different parts of the configurator are illustrated in Figure 5.1.

5.1.1 Global parameters

In the global parameter window, there are options for setting up the properties for a communication cycle and the clock synchronization. Some of the changeable properties for a communication cycle are the length of the static segment, the dynamic segment, the the symbol window and the network idle time. Also, the duration of a macrotick and the maximum payload length in both the static and dynamic slots are examples of changeable properties.

5.1.2 Node parameters

For each node it is possible to configure general node settings, properties of the message buffer (define receive and transmit buffers), and define the PDU's and the different jobs that will be executed.

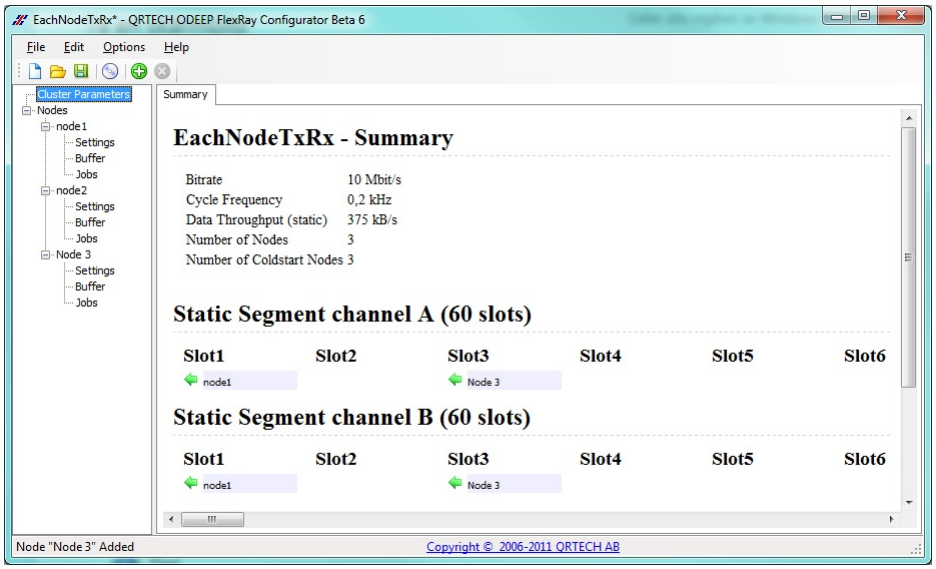


Figure 5.1: The different parts in the main window of FlexRay configurator.

A node may be defined as a synchronization node, to broadcast synchronization settings to all other nodes in the network. It is also possible to mark the node as a startup node, having the responsibility to wake up the other nodes by sending a wakeup signal.

In the job configuration, transmit and receive PDU's are defined. Significant parameters are for example the ID of the PDU, the size and which FlexRay frame it is supposed to be in. This is also where the jobs are scheduled, by stating how many macroticks after startup they should be initiated. A schedule for three different jobs are shown in Figure 5.2. Moreover, each job has one or more operation types defined. For instance, it may consist of a transmit operation, a transmit confirmation operation or a receive operation. The functioning of jobs, PDU's and other parameters are further described in the previous Section 4.1.

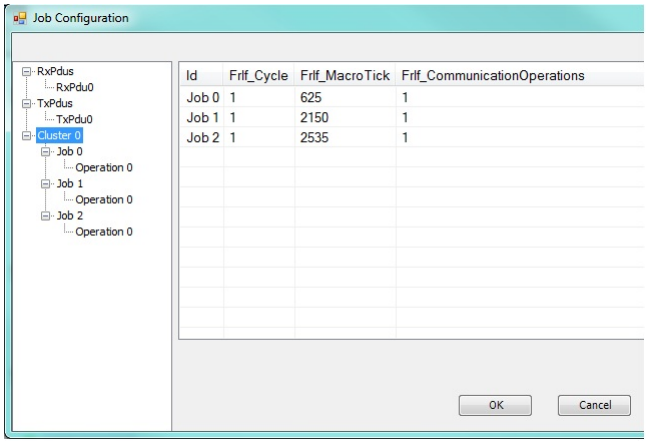


Figure 5.2: The schedule of three defined jobs for a specific node.

When all parameters are set as desired for the FlexRay cluster, the FlexRay Configurator is

ready to generate code for the configuration files that are used by the FlexRay Interface and the FlexRay Driver.

5.2 Configuration in Arctic Core

Since the FlexRay Configurator only generates configurations to the FlexRay Driver and the FlexRay Interface, some other tool is needed for the remaining AUTOSAR modules. For this, the Arctic Studio environment was used. Arctic Studio supports all other necessary modules needed to achieve a working FlexRay communication stack. This section describes what was used in the Arctic Studio and in some sense how the configuration was done.

When using the Arctic Studio, it is possible to use a repository with the source code to all AUTOSAR modules that are supported by the Arctic Studio. This repository is called the Arctic Core and the current version of this is AUTOSAR 3.1 [25]. With the Arctic Core it is possible to create a complete AUTOSAR environment. However, only the source code to the modules is not enough to create the AUTOSAR environment. These modules needs to be configured just like the FlexRay modules needed to be with the FlexRay Configurator. For this, Arctic Studio provides a plugin that is called BSW Builder. In this plugin all modules belonging to the Arctic Core repository can be configured. In the BSW Builder the user can choose which modules are needed for the current ECU. When the modules have been chosen, all of them can be configured with the specific properties for that module. Figure 5.3 shows the BSW Builder and the window to add modules. It is of course possible to use the source code without the BSW Builder, but then all parameters need to be configured manually for each module, which will take a long time.

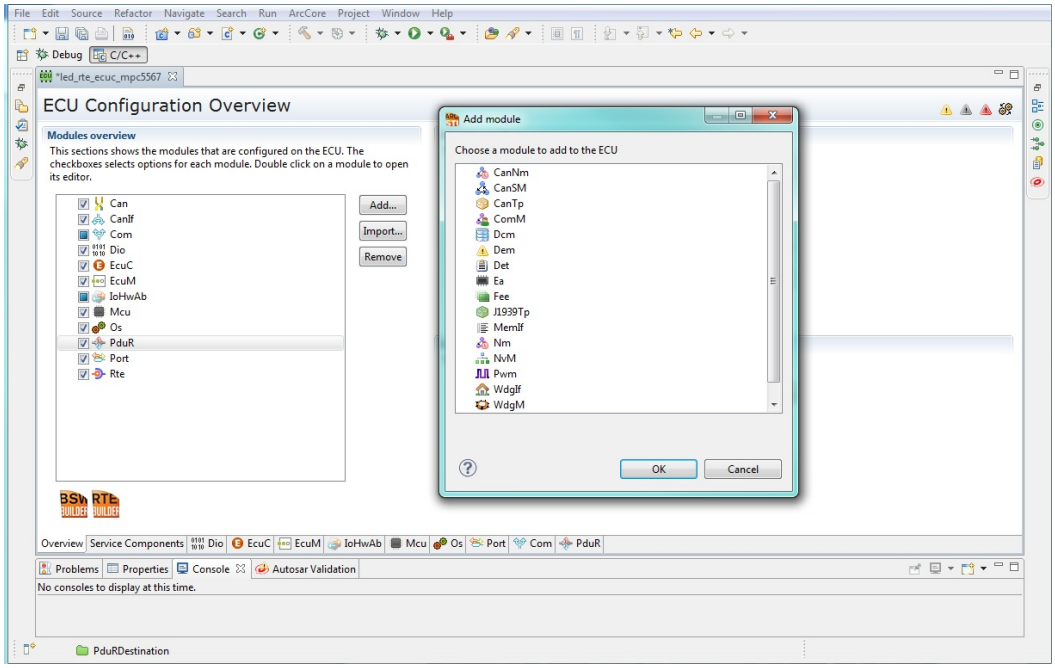


Figure 5.3: The Arctic Studio environment together with the BSW Builder plugin.

5.2.1 Available modules

The modules Com and PduR are necessary to be able to have a FlexRay communication working according to the AUTOSAR platform. There are other modules included in the FlexRay communication stack as well, but these two are the ones necessary to get PDU's to travel from the Com module to another ECU. The parameters configured in these two modules are mostly about what to do with the PDU's. In the Com module, each PDU is configured with PDU specific settings. Examples of PDU settings are if they are of transmit or receive type, what should happen if received and default values on transmission PDU's. The Com module also configures signals and which PDU the signals belong to. The signals initial value, size and endianess are also examples of configurable parameters in the Com module. With the knowledge of the settings done in the Com module, the PduR module can be configured. Since the PduR routes the PDU's, this module needs to be aware of them. The PduR module routes the PDU's according to the routing table that is configured. Each PDU that is configured in Com needs to have a routing path configured in PduR. A typical configure for a PDU that is supposed to be sent from the ECU has the source module configured to the Com and destination to the FlexRay Interface.

Other modules that are needed to get a working platform are Dio, EcuC, EcuM, IoHwAb, Mcu, Os, Port, and Rte. Some of them are more important then others and those are further described below. The other ones that are not treated here have no effect on the communication, or use some default settings.

Modules that are important for this work are the Mcu and the Port module. These two modules need to be configured in a proper way to get the communication to work, but they are not part of the FlexRay communication stack. The Mcu module configures the clock settings for the processor and examples of settings are values for the PLL (Phase-locked loop) registers. These settings are needed to control the clock speed on the processor. In the Port module the ports needed are configured. In our case we needed to configure the ports for FlexRay, RAM and some I/O ports. These modules do not affect the FlexRay stack directly, though they are needed to configure the hardware in a proper way to be able to accomplish FlexRay communication.

When all necessary parameters are configured, the BSW Builder generates the configuration files just like the FlexRay Configurator did. With the configuration files and the source files from the Arctic Core, the software is ready to be used.

Chapter 6

Results

In order to test and verify the functionality of the implemented modules, a demonstrator was made during this project. To be able to verify the implementation, the demonstrator should at least be able to transmit and receive PDU's. This chapter describes how the demonstrator was designed, how it performs the testing and the outcome.

The demonstrator use three ODEEP QR5567 platforms to perform the tests. From here, the three platforms are called Node 1-3. All three nodes are connected with each other via a FlexRay bus. Each node has a unique configuration in both the Arctic Studio environment and in the FlexRay Configurator. The purpose with the test is that Node two and Node three sends a message each to Node one. Node one controls the message and since Node one can predict the correct value of the messages it signals a failure if any of the two messages are wrong. Then, Node one replies to Node two and Node three with the same messages, and Node two and Node three controls if the messages are the same as they sent earlier. By doing this test it is possible to control that data that is sent and received is handled in a correct way.

Another test that is carried out is to control that no messages are lost. This is done by incrementing a counter in each new FlexRay cycle. If the counter has been incremented more than one step since the last received message there has been a message loss, and an error is signaled.

The demonstrator is configured in the following way:

- Node two sends a message in slot 10 to node one
- Node one performs previous described tests
- Node three sends a message in slot 12 to node one
- Node one performs previous described tests
- If all test are successful, Node one replies to Node two and Node three in slot 25
- Finally, Node two and Node three controls the response from Node one and the procedure is repeated

This is further illustrated in Figure 6.1.

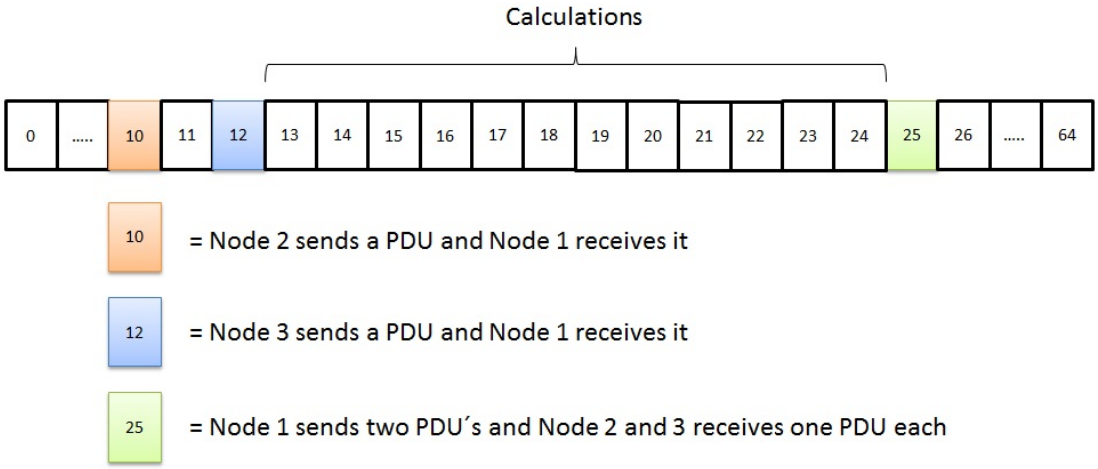


Figure 6.1: The configuration of the demonstrator

The Arctic Studio configuration is same on all three nodes, regarding general parameters for the ODEEP QR5567 platform. AUTOSAR modules configured in the same way are the DIO, the ECUm, the IOHwAb, the MCU, the OS, the PORT, and the RTE. The other modules that are used, but configured in different ways, are the EcuC, the Com and the PduR. They are configured in different ways due to differences between the PDU's. The configuration on Node two and Node three are the same, both of them sends one PDU and receives one. Node one has a slightly more complicated configuration. Node one sends two PDU's and receives two PDU's.

The FlexRay configuration has a lot of parameters configured, but the most important are the ones regarding transmission and reception. It is also these parameters that are different between the nodes:

- Node one receives in slot 10 and 12 and transmits in slot 25
- Node two transmits in slot 10 and receives in slot 25
- Node three transmits in slot 12 and receives in slot 25

One thing that should be noted here is that Node one only transmits in slot 25, but it has two PDU's to send. This is possible due to the feature in FlexRay Interface to be able to handle several PDU's in one FlexRay frame.

Once the cluster and all nodes are configured in the previous explained way, the demonstrator can be started. On startup, Node two and Node three are configured to always transmit. But the PDU is however marked with an updated bit only if new data has been assigned to the PDU. Assigning data to the PDU is performed by function calls in the Com module. Thus, the test starts in the Com module on Node two and Node three by assigning new data to the PDU's. If Node one receives a frame with the PDU update bit set, the Com module on Node one will be called and the data is checked. If Node one should receive a PDU with no update bit set, the Com module will not be called. In that case, the test will fail because then the

counter mentioned earlier has been incremented without a PDU received. Once the test has started, Node one should receive PDU's every time, or else a PDU is lost.

The data that is sent "travel" from the Com module to the PduR to the FrIf and then to the Fr Driver via the bus and then received on Node one. Here the data travels the opposite way, from the Fr Driver to the FrIf to the PduR and finally to the Com. When Node one replies, the data travels the same way but in the opposite direction.

By doing these tests the demonstrator shows that the configuration and implementation is working in a correct manner. The following features are tested by the demonstrator:

- The transmission of a PDU from the top of the AUTOSAR communication stack, i.e. the Com module, is done in a correct way
- The reception of a PDU is done in a correct way
- No PDU's are lost

The demonstrator has been running for at least 12 hours with messages sent every 300 millisecond with no reported errors. The demonstrator only tests the correct behavior while using three nodes and with the configuration used in this example. Figure 6.2 depicts the main modules used for the AUTOSAR communication stack obtained in this project, and which configurator that is used on different parts of the stack.

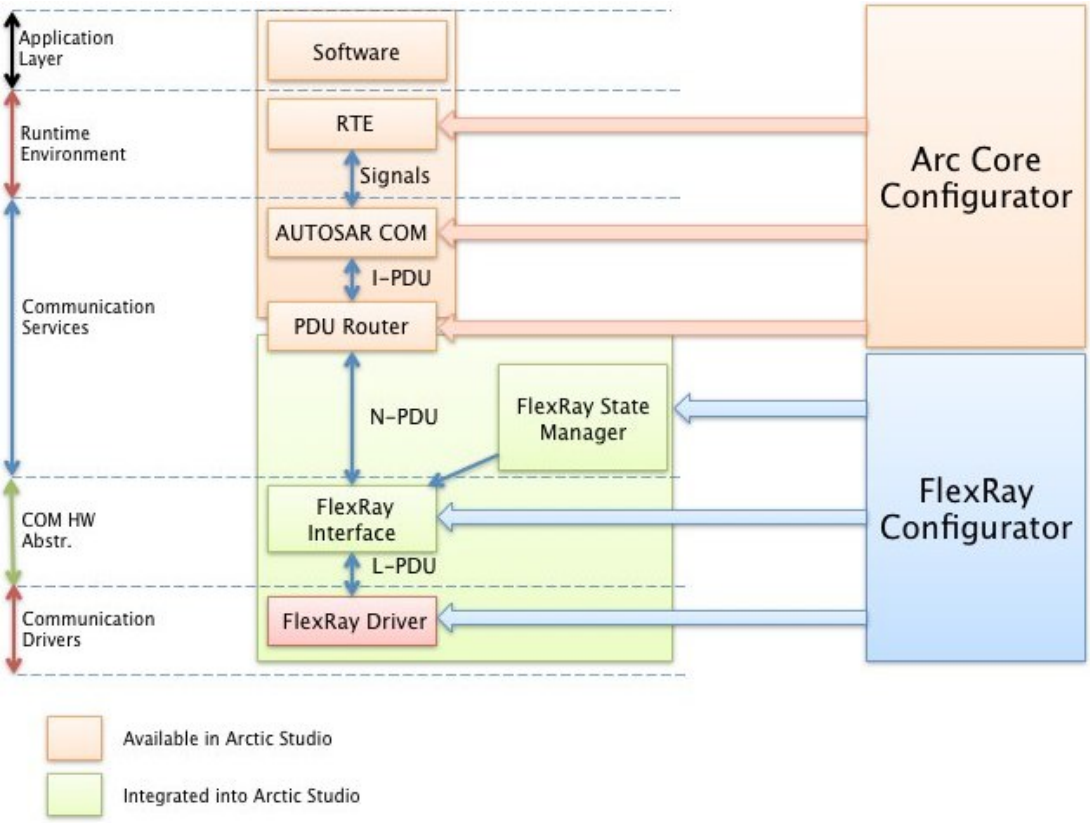


Figure 6.2: The main AUTOSAR modules used for transporting a PDU.

Chapter 7

Discussion and conclusion

This chapter presents the final conclusions of this project. It reveals the choices made and problems met during the working process. Discussions are made regarding the outcome and future work is suggested.

7.1 Discussion

The process of integrating available FlexRay modules in the Arctic Studio included certain modifications of the software available in the Arctic Studio, since it had no FlexRay support at the start of the project. For example, specific FlexRay interrupts had to be introduced, a completely new port configuration for FlexRay was made, and "glue code" had to be used to allow interaction between the different AUTOSAR modules of different AUTOSAR versions. This proves that AUTOSAR modules of older versions are, to some extent, compliant with the newer AUTOSAR version 4.0. The overall functionality of the modules is however limited to the features existing in the older versions.

One issue, which was faced during the implementation, was how AUTOSAR had constructed the specification for the FlexRay Interface. When all data structures and variables had been declared and the functions were to be defined, difficulties in reaching certain resources appeared. In order to accomplish certain tasks within some of the functions, new parameters had to be defined in addition to the ones defined in the AUTOSAR specification. According to [31], AUTOSAR specifications are not complete in some cases, which enables room for interpretation. This fact leads to another issue within this project regarding the interaction between different configuration tools.

Since new parameters were introduced in the implementation, the updated FlexRay Configurator needed support for these. A restriction which appears with this is the fact that the FlexRay Configurator will only be utilized correctly together with the FlexRay Interface module implemented in this project. This is why several vendors offer their own configurators with their AUTOSAR implementations [32], [33], [34].

Integrating FlexRay in the Arctic Studio would bring a scenario where the Arctic Studio

modules would be configured in one program, Arctic Core, and the FlexRay modules in another program, QRTECH ODEEP FlexRay Configurator. The configuration process would then be time consuming and not very user friendly. Therefore, QRTECH began the development of an upgraded version of the QRTECH ODEEP FlexRay Configurator during this project. The aim with the new FlexRay Configurator is that it will be coordinated with Arctic Core in the Arctic Studio. A problem that will arise with this is that Arctic Core uses single node configuration, while FlexRay Configurator uses multiple node configuration or cluster configuration. The overall solution of the configuration process lies however outside the scope of this project and will be treated further in the future.

One major difference between FlexRay Interface version 2.0 and version 4.0 was that in version 2.0 a state machine was included. As from late 2007 [15], the state machine was removed to be handled in a separate module, the FlexRay State Manager. Thus, to obtain a complete and genuine FlexRay Interface 4.0 module, basic parts of the FlexRay State Manager were implemented, including an upgraded state machine. However, implementing the whole FlexRay State Manager module would require a significant amount of time, because it interacts with several other AUTOSAR modules, e.g. ComM, BswM, SchM and of course FlexRay Interface [15]. The work of implementing remaining FlexRay communication stack modules is outside the content of this project.

An alternative path for this project was to implement the AUTOSAR module Complex Driver, instead of the FlexRay communication stack. As shown in Figure 2.2, the Complex Driver can be used to communicate from the microcontroller all the way to the RTE layer. A disadvantage of using the Complex Driver is that the implementation will miss out on the modularity property, meaning that if support for another microcontroller is needed, a completely new Complex Driver must be implemented. An advantage of using the Complex Driver is that it can be optimized, offering less run-time overhead than the AUTOSAR communication stack. According to [9], the main purpose with Complex Driver is to offer the possibility to integrate drivers for devices which are not specified within AUTOSAR or that have high timing restrictions.

Using the terminal as an interface for the demonstrator caused certain difficulties. Occasionally, data indicating a successful transmission was not printed out to the user. The reason for this was because printing data as terminal output was too time consuming in relation to the transmission speed of FlexRay. Thus, the method of printing data to the user had to be limited.

Another problem, regarding FlexRay timing, was how frequent transmissions/receptions could be scheduled within a communication cycle. Testing showed that it was possible to schedule eight time slots between each transmission/reception for a specific configuration without failure. This is approximately only a third of the possible speed of 10 Mbit/s of which FlexRay can reach. However, the significant difference is most probably because the ODEEP QR5567 platform does not manage to execute the transmissions in time. A future method would be to execute the tests on a platform with higher performance.

7.2 Conclusion

This project has resulted in a working AUTOSAR communication stack for FlexRay. This was achieved by integrating existing FlexRay modules and an implemented FlexRay Interface version 4.0 according to available specifications. These AUTOSAR modules together with the modules available in Arctic Studio, has made a complete AUTOSAR FlexRay Stack regarding the most important feature, to send and receive PDU's. The results shows that modules of different AUTOSAR versions can be put together to form a functional AUTOSAR BSW. It also proves that modules developed by different vendors can be combined. However, integrating modules from different vendors requires full understanding about which role each module has in the BSW.

The configurations of the different AUTOSAR modules were made with a new FlexRay configurator and Arctic Core configurator in Arctic Studio. Constructing a new configurator for the new module requires extensive knowledge about the modules internal functions. This is needed because the AUTOSAR specifications enables room for own interpretations. Also, a demonstrator has been implemented to visualize and test communication between three different FlexRay nodes, occurring within one communication cycle. The outcome of the demonstrator shows that the communication stack functions in a correct manner.

A further extension to this project would be to implement the remaining AUTOSAR FlexRay modules, such as FlexRay Transport Protocol, FlexRay Network Manager and the remaining parts of FlexRay State Manager.

One extension, regarding the configuration of the AUTOSAR modules, could be to put the Arctic Studio together with the FlexRay Configurator. Merging these two configurators would form a powerful tool for the automobile industry in the future.

References

[1] Leen G, Heffernan D. Expanding Automotive Electronic Systems. Computer. 2002;35(1):88–93.

[2] Arvidsson M, Gjorloff C. AUTOSAR Demonstrator for FlexRay. Chalmers University of Technology; 2007.

[3] CAN in Automation [homepage on the Internet]. Nuremberg: CAN in Automation; c2001-2011 [updated 2011 Oct; cited 2011 Oct 27]. Available from: `http://www.can-cia.org/`.

[4] Local Interconnect Network [homepage on the Internet]. Munich: Altran; 2011 [updated 2011 May; cited 2011 Oct 27]. Available from: `http://www.lin-subbus.de`.

[5] TTCAN [homepage on the Internet]. Stuttgart: Bosch; 2011 [updated 2011 Oct; cited 2011 Nov 23]. Available from: `http://www.semiconductors.bosch.de`.

[6] AUTOSAR [homepage on the Internet]. Munich: AUTOSAR; c2003-2011 [updated 2011 Oct; cited 2011 Oct 27]. Available from: `http://www.autosar.org`.

[7] QRTECH [homepage on the Internet]. Gothenburg: QRTECH AB; 2011 [updated 2011; cited 2011 Dec 3]. Available from: `http://www.qrtech.se/`.

[8] ODEEP [homepage on the Internet]. Gothenburg: QRTECH AB; 2011 [updated 2011; cited 2011 Nov 14]. Available from: `http://www.odeep.se/`.

[9] AUTOSAR-Administration. AUTOSAR Layered Software Architecture [pdf]. Munich: AUTOSAR; 2008.

[10] AUTOSAR. Specification of FlexRay Driver, version 2.5.0, release 4.0 [pdf]. Munich: AUTOSAR; 2011.

[11] AUTOSAR. Specification of FlexRay Interface, version 3.1.0, release 4.0 [pdf]. Munich: AUTOSAR; 2010.

[12] AUTOSAR. Specification of FlexRay Transport Layer, version 3.1.0, release 4.0 [pdf]. Munich: AUTOSAR; 2010.

[13] AUTOSAR. Specification of PDU Router, version 3.1.0, release 4.0 [pdf]. Munich: AUTOSAR; 2010.

[14] AUTOSAR. Specification of communication, version 3.2.0, release 4.0 [pdf]. Munich: AUTOSAR; 2010.

[15] AUTOSAR. Specification of FlexRay State Manager, version 2.2.0, release 4.0 [pdf]. Munich: AUTOSAR; 2011.

[16] FlexRay [homepage on the Internet]. Munich: Altran; 2011 [updated 2011 May; cited 2011 Oct 27]. Available from: http://www.flexray.com.

[17] Controller Area Network Overview - Developer Zone [homepage on the Internet]. Austin: National Instruments Corporation; 2011 [updated 2011 May; cited 2011 Oct 27]. Available from: http://zone.ni.com/devzone/cda/tut/p/id/2732.

[18] Fernström M, Ungerdahl D. Master's Thesis. TTCAN Reference Application - An investigation of time-triggered network performance. 2006;.

[19] FlexRay [homepage on the Internet]. Kawasaki: Fujitsu; c1995-2011 [updated 2011; cited 2011 Oct 27]. Available from: http://www.fujitsu.com/global/services/microelectronics/technical/flexray/m.

[20] Wikipedia [homepage on the Internet]. San Francisco: Wikipedia Foundation, Inc.; 2011 [updated 2011 Oct 24; cited 2011 Oct 27]. Available from: http://www.wikipedia.org/wiki/FlexRay.

[21] Keskin U. In-vehicle Communication Networks - A Literature Review. 2009;.

[22] FlexRay-consortium. FlexRay Communications System Protocol Specification Version 3.0.1 [pdf]. Munich: Altran; 2010.

[23] Freescale [homepage on the Internet]. Austin: Freescale Semiconductor, Inc.; c2004-2011 [updated 2011 Nov; cited 2011 Nov 14]. Available from: http://www.freescale.com/.

[24] QRTECH-AB. QRTECH ODEEP FlexRay Configurator – User Manual v1.3 [pdf]. Gothenburg: QRTECH AB; 2011.

[25] Arc Core [homepage on the Internet]. Askim: ArcCore AB; 2011 [updated 2011 Nov 9; cited 2012 Jan 25]. Available from: http://arccore.com/.

[26] Eclipse [homepage on the Internet]. Ottawa: Eclipse Foundation; 2011 [updated 2011 Nov 9; cited 2011 Nov 14]. Available from: http://www.eclipse.org.

[27] PE Micro [homepage on the Internet]. Watertown: PE Microcomputer Systems; 2011 [updated 2011 Jul 16; cited 2011 Nov 14]. Available from: www.pemicro.com.

[28] PEEDI [homepage on the Internet]. Vienna: Ronetix; c2005-2011 [updated 2011 Nov 9; cited 2011 Nov 14]. Available from: http://www.ronetix.at/peedi.html.

[29] AUTOSAR. Specification of FlexRay Transceiver Driver, version 1.5.0, release 4.0 [pdf]. Munich: AUTOSAR; 2011.

[30] Oracle Java [homepage on the Internet]. Redwood Shores: Oracle; 2011 [updated 2012 Feb 29; cited 2012 Mar 3]. Available from: http://www.oracle.com/technetwork/java/index.html.

[31] Diekhoff D. AUTOSAR Tooling in practice [pdf]. Erlangen: Elektrobit Automotive GmbH; 2010.

[32] Automotive Solutions [homepage on the Internet]. München: Mentor Graphics; 2012 [updated 2012 Mar 1; cited 2012 Mar 6]. Available from: http://www.mentor.com/products/vnd/autosar-products/.

[33] AUTOSAR Basis Software [homepage on the Internet]. Vienna: TTTech; 2011 [updated 2011 Nov 30; cited 2012 Mar 6]. Available from: http://www.tttech.com/products/automotive/safe-runtime-software/autosar-basis-software/.

[34] MICROSAR - Your Basic Software for AUTOSAR [homepage on the Internet]. Stuttgart: Vector Informatik GmbH; 2012 [updated 2012 Feb 29; cited 2012 Mar 6]. Available from: http://www.vector.com/vi_microsar_en.html.

Appendix A

Flow charts

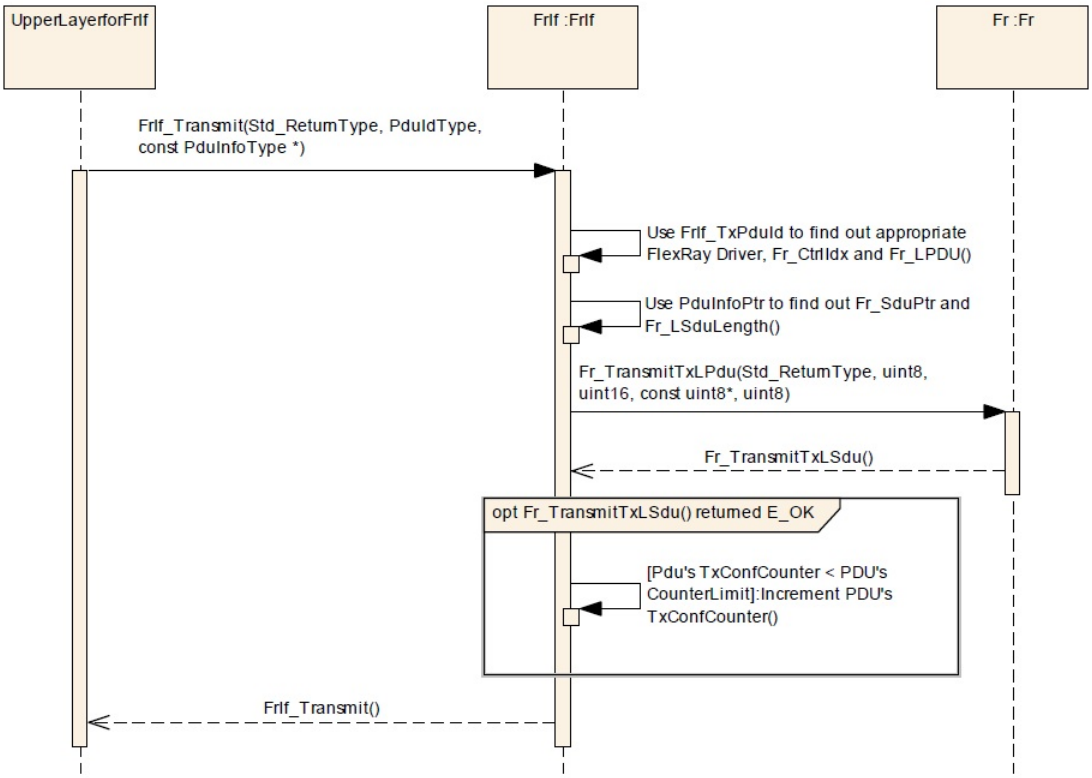


Figure A.1: Transmit with immediate buffer access [11].

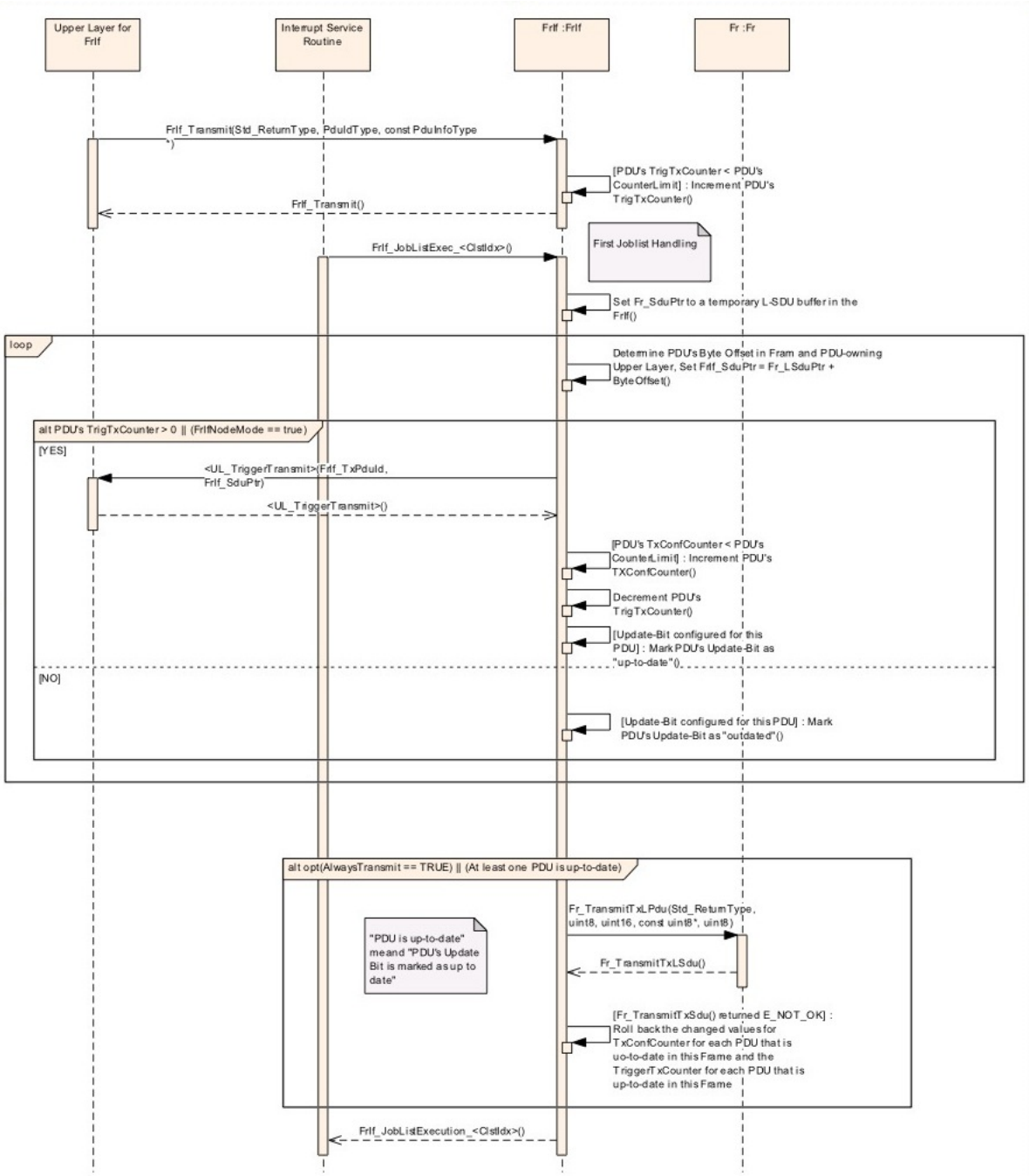


Figure A.2: Transmit with decoupled buffer access [11].

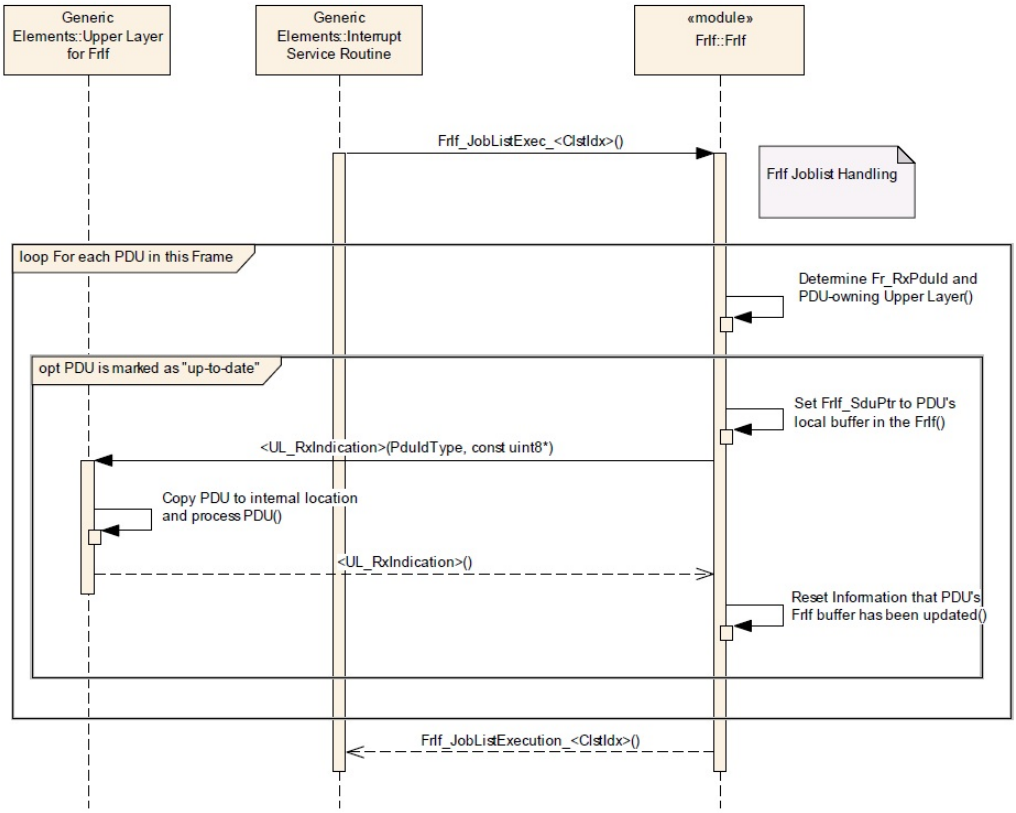


Figure A.3: Transmit confirmation [11].

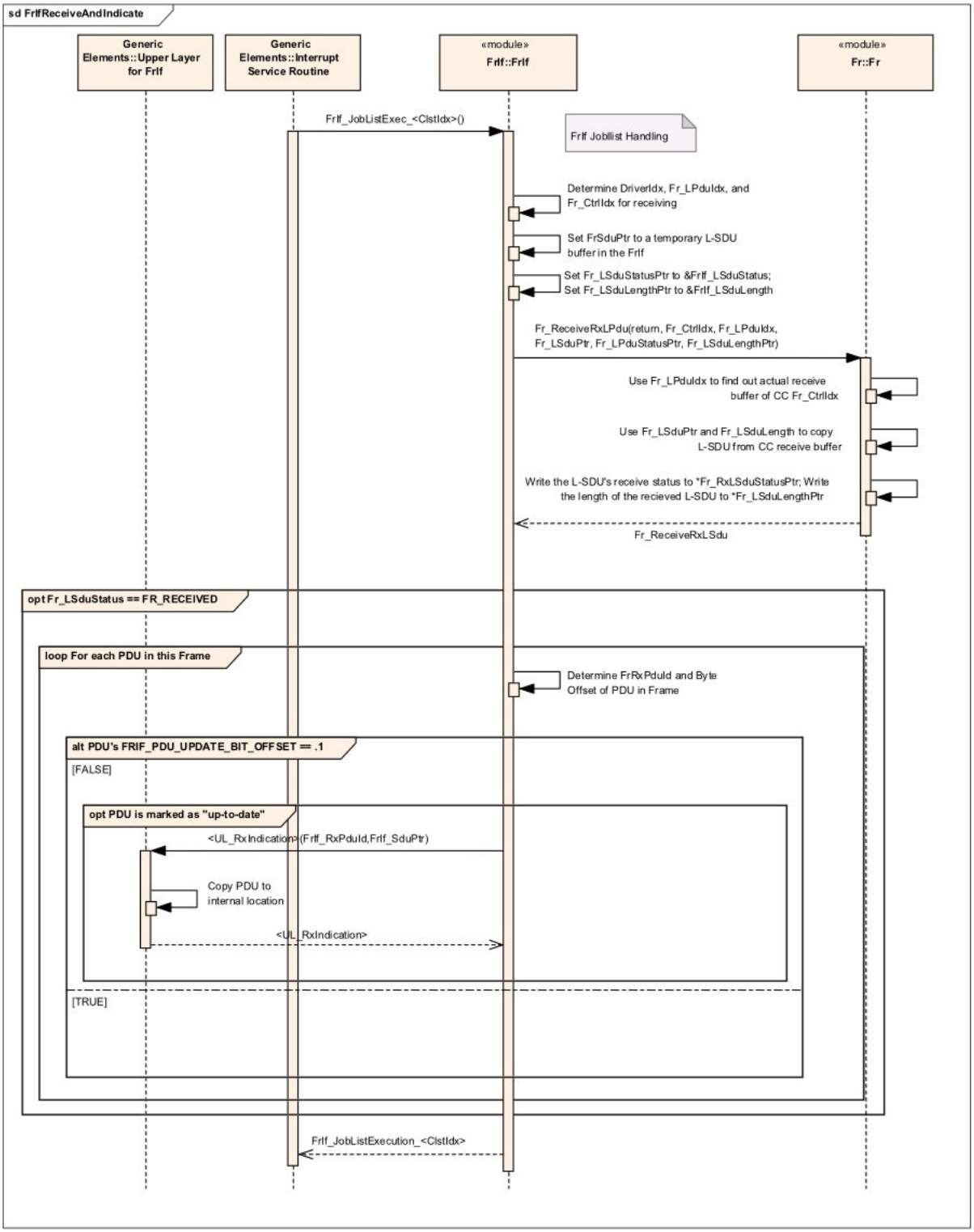


Figure A.4: Receive and Indicate [11].

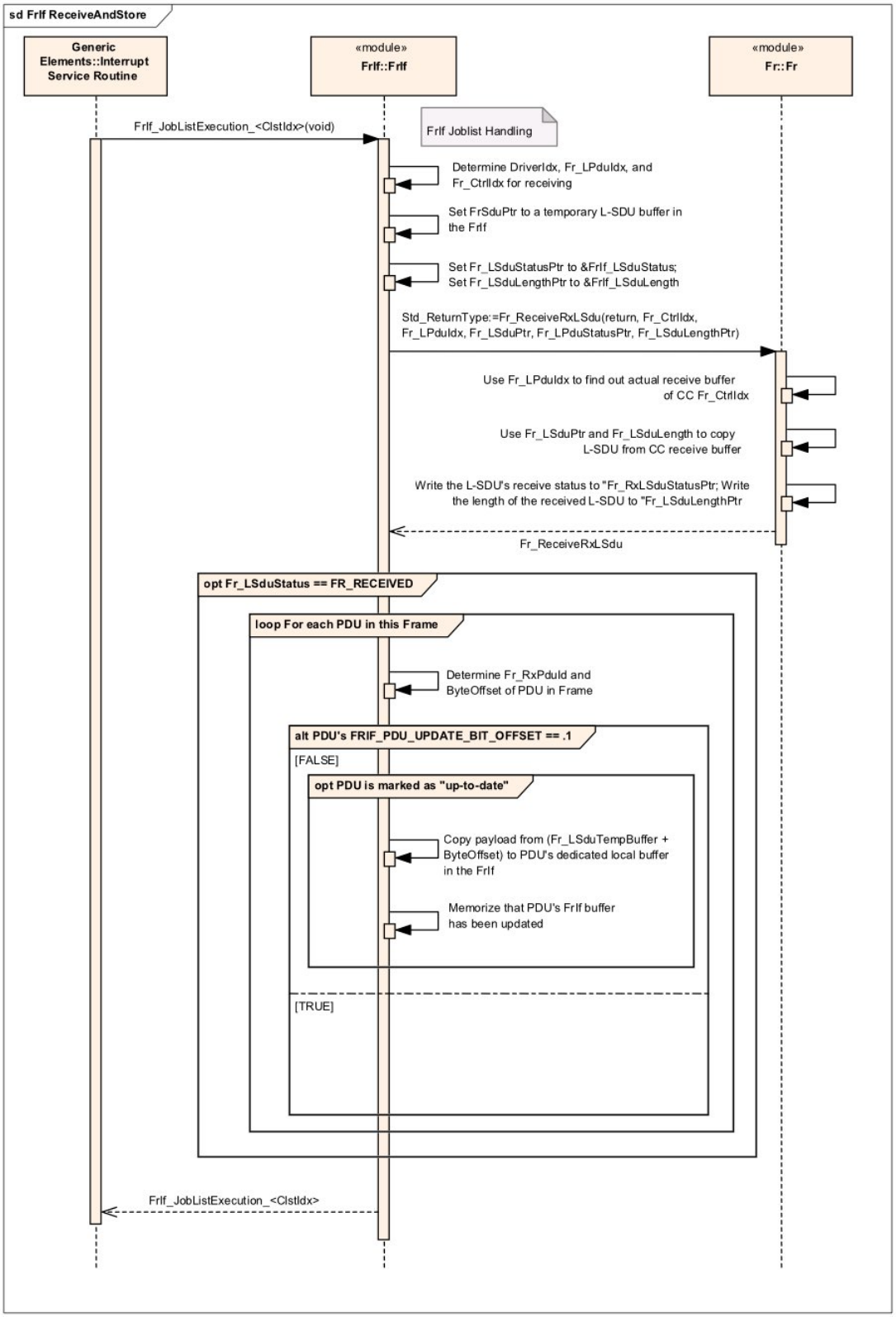


Figure A.5: Receive and store [11].

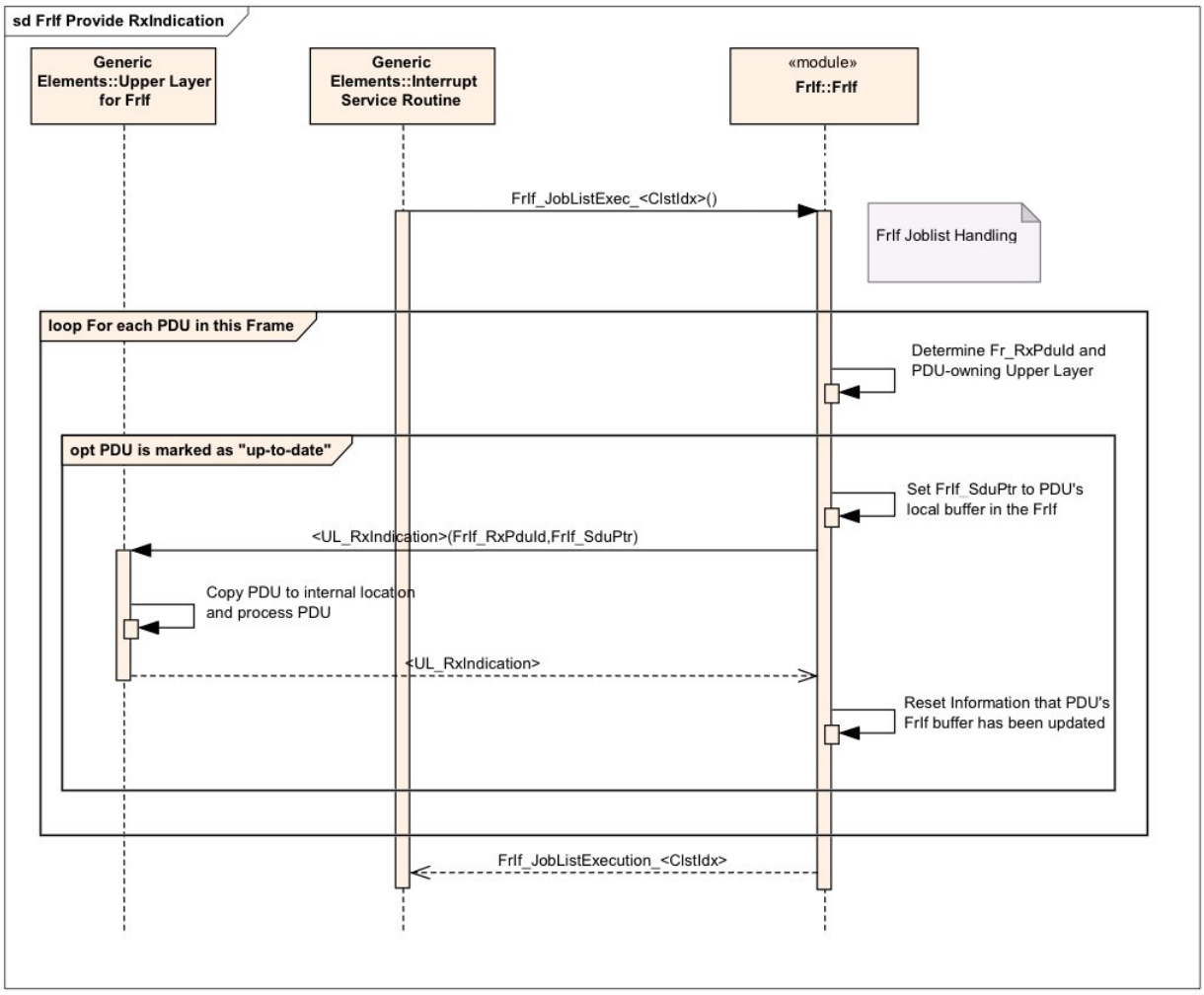


Figure A.6: Provide Rx indication [11].

Appendix B

XML and C code examples in configuration files

```xml
<FrIf_Tx_Pdu_Types>
  <anyType xsi:type="FrIf_Tx_Pdu_Type">
    <Id>TxPdu 0</Id>
    <FrIf_Confirm>true</FrIf_Confirm>
    <FrIf_CounterLimit>10</FrIf_CounterLimit>
    <FrIf_NoneMode>false</FrIf_NoneMode>
    <FrIf_TxConfirmationName>PduR_FrIfTxConfirmation</FrIf_TxConfirmationName>
    <FrIf_TxPduId>0</FrIf_TxPduId>
    <FrIf_UserTriggerTransmitName>PduR_FrIfTriggerTransmit</
        FrIf_UserTriggerTransmitName>
    <FrIf_UserTxUL>PDUR</FrIf_UserTxUL>
    <FrIf_TxPduRef>null</FrIf_TxPduRef>
    <FrIf_Offset>0</FrIf_Offset>
    <FrIf_SlotId>20</FrIf_SlotId>
    <FrIf_TxLength>16</FrIf_TxLength>
  </anyType>
</FrIf_Tx_Pdu_Types>
```

Listing B.1: XML code for a transmission PDU

```c
const FrIf_Tx_Pdu_Type FrIf_TxPduType[] = {
  {
    .FrIf_Confirm = TRUE,
    .FrIf_CounterLimit = 10,
    .FrIf_NoneMode = FALSE,
    .FrIf_TxConfirmationName = PduR_FrIfTxConfirmation,
    .FrIf_TxPduId = 0, // Reference to global pdu definition
    .FrIf_UserTriggerTransmitName = PduR_FrIfTriggerTransmit,
    .FrIf_UserTxUL = PDUR,
    .FrIf_TxPduRef = NULL,
    .FrIf_Offset = 0,
    .FrIf_TxLength = 16
  }
};
```

Listing B.2: C code for a transmission PDU