

Nikolay Kirilov Stoitsov

Configuration, Control and Monitoring of a FlexRay Cluster via TCP/IP

Bachelor thesis based on the study regulations
for the Bachelor of Engineering degree programme
Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the Hamburg University of Applied Sciences

Supervising examiner : Prof. Dr. Lutz Leutelt
Second Examiner : Prof. Dr. Ulrich Sauvagerd

Day of delivery 21. December 2017

Nikolay Kirilov Stoitsov

Title of Bachelor Thesis

Configuration, Control and Monitoring of a FlexRay Cluster via TCP/IP

Keywords

Automotive network, FlexRay, Ethernet, TCP/IP, lwip, MicroZed, PC client, server node, software application

Abstract

This bachelor thesis describes the development and implementation design of a software application, used to perform initial configuration, control and monitoring tasks of a FlexRay node. The communication is realized by hard-wired cables and a network switch and is based on the TCP/IP protocol.

Nikolay Kirilov Stoitsov

Thema der Bachelorthesis

Konfigurierung, Kontrolle und Überwachung von einem Cluster via TCP/IP

Stichworte

Automobil Netzwerk, FlexRay, Ethernet, TCP/IP, lwip, MicroZed, PC Klient, Server-Knoten, Softwareanwendung

Kurzzusammenfassung

Diese Bachelorarbeit beschreibt die Entwicklung und Implementierung einer Softwareanwendung, die für die Erstkonfiguration, Steuerung und Überwachung eines FlexRay-Knotens verwendet wird. Die Kommunikation erfolgt über festverdrahtete Kabel und einen Netzwerk-Switch und basiert auf dem TCP/IP-Protokoll.

Table of contents

List of Figures	5
List of Tables	5
1. Introduction	6
2. Fundamentals	8
2.1 Introduction to Bus Systems	8
2.2 OSI Model.....	9
2.3 FlexRay Protocol.....	10
2.3.1 Communication Controller	10
2.3.2 FlexRay Communication	14
2.3.3 E-Ray	19
2.4 Communication Protocols.....	28
2.4.1 Ethernet Protocol.....	28
2.4.2 TCP/IP Protocol Suite.....	30
2.4.3 LwIP Protocol/Raw API	32
2.5 Hardware Platform MicroZed Board	34
3. Requirements Analysis	36
3.1 General Requirements.....	36
3.2 Data Objects.....	37
3.3 Software Features and Structure	38
3.4 Evaluation	39
4. Concept	41
4.1 Configuration Data Objects	41
4.1.1 Files Content	41
4.1.2 Files Structure	44
4.2 Communication Data Objects	50
4.3 Software Design.....	53
4.3.1 Graphical User Interface	53
4.3.2 Operation Control Flow	54
4.3.3 Programming Model	56
5. Implementation	58
5.1 Client Application.....	58
5.2 Server Application	62
6. Evaluation	63
7. Conclusion	65

Bibliography.....66
Appendix A68
Appendix B73
Application User Manual87

List of Figures

Fig 1.1. Example of services provided in nowadays vehicle	6
Fig 1.2. TCP/IP connection between a PC client and a FlexRay cluster via a network switch.....	7
Fig 2.1. System bus [1]	8
Fig 2.2. Seven layer OSI model [4]	9
Fig 2.3. Logical interfaces in a FlexRay node [7](p. 26)	11
Fig 2.4. Conceptual architecture of the Controlling Host Interface [7] (Figure 9-1).....	11
Fig 2.5. Overall state diagram of a FlexRay communication controller [7] (p. 37)	12
Fig 2.6. Timing hierarchy within the communication cycle ([7] p. 100).....	14
Fig 2.7. (a) Communication cycle with no transmission in the dynamic segment; (b) Communication cycle with several transmissions in the dynamic segment [10] (Figure 5.5)	15
Fig 2.8. FlexRay frame format [7](p. 90)	17
Fig 2.9. E-Ray block diagram [8](p. 15).....	20
Fig 2.10. Configuration example of message buffers in the Message RAM [8](p. 141).....	21
Fig 2.11. Access to Transient Buffer RAMs [8] (Figure 14).....	22
Fig 2.12. Host access to Message RAM [8](Figure 9).....	23
Fig 2.13. Assignment of message buffers [8] (Figure 1)	24
Fig 2.14. Possible FIFO states [8] (Figure 8).....	25
Fig 2.15. Transmit process via the Input Buffer [8] (Figure 10)	26
Fig 2.16. Receive process via the Output Buffer [8] (Figure 12)	27
Fig 2.17. The major Ethernet layers defined by IEEE [12] (p.13).....	28
Fig 2.18. DIX frame vs. IEEE 802.3 frame [12] (p. 41)	29
Fig 2.19. OSI Model vs. TCP/IP Model [6](p. 129)	31
Fig 2.20. MicroZed block diagram [22] (Figure 1).....	34
Fig 2.21. Boot Mode Jumper Settings with Cascaded JTAG Chain [22] (Figure 8)	35
Fig 2.22. 10/100/1000 Ethernet interface [22] (Figure 5).....	35
Fig 4.1. Possible transition routes to <i>config</i> state.....	54
Fig 4.2. Configuration control flow (a) in the hands of the user; (b) embedded in the code.....	55
Fig 4.3. Model-View-Controller design pattern.....	56
Fig 5.1. Class diagram for the client application, based on the MVC model	58
Fig 5.2. Activity diagram of the configuration process	60
Fig 5.3. Activity diagram of the monitoring process	61
Fig 7.1 Monitoring of data packets via WireShark.....	64
Fig A.1. Protocol operation control context [7](p. 32)	69
Fig A.2. Header section of a message buffer in the Header Partition of the message RAM [8]	71
Fig A.3. Data partition in the message RAM [8] (p. 145)	73

List of Tables

Table 2-1 CHI commands summary ([7] p. 33).....	13
Table 2-2 Definition of cycle set [8] (Table 9)	18
Table 5-1 Number of configuration JSON files - advantages and disadvantages	44
Table A-1 Parameter prefixes [7] (p. 18).....	68

1. Introduction

Automobiles have long become much more than just a mean of transporting people. Each vehicle nowadays represents a system in its own. Regardless of the brand or country of manufacture, each car is equipped with a variety of Electronic Component Units (ECU), which communicate constantly as long as the engine is on and some of them continue even after it goes off. The role of those ECUs are to support the driving control by providing security and defence mechanisms in critical situations, as well as insuring comfort for the driver and his fellows (Fig 1.1).

Nowadays, the average modern automobiles are supported by systems that are reliable for:

- Breaks - antilock braking system (ABS), auto braking system, power brake booster
- Electronic Stability Program (ESP)
- Parking – cameras, distance sensors
- Lights – light, rotary sensors
- Dashboard, navigation systems, infotainment, etc.
- Airbags, windows, mirrors, seats, heat control, etc.

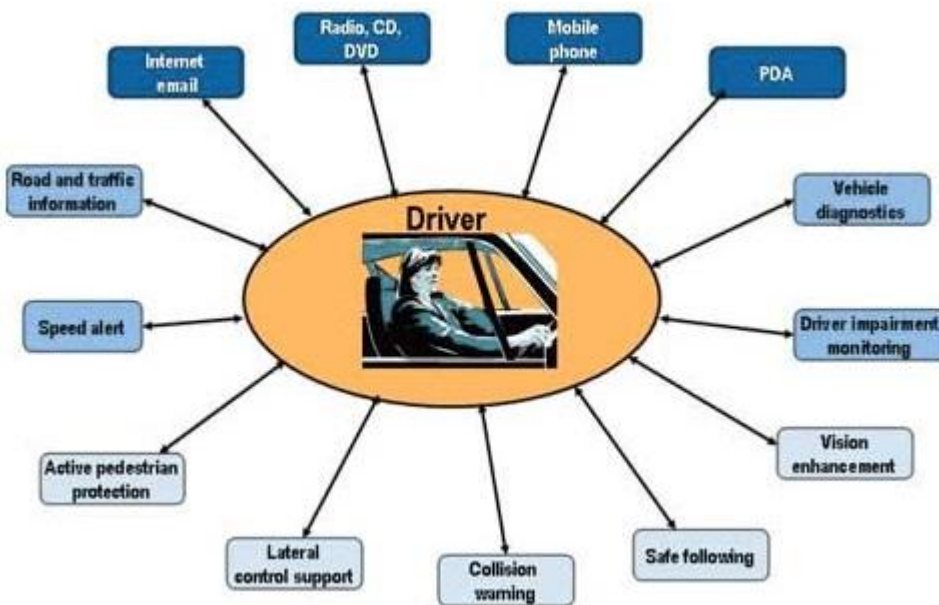


Fig 1.1 Example of services provided in nowadays vehicle

The growth of demand for more extras provided in the car requires the invention of more and more complex system protocols, able to support high-rate but in the same time reliable communication between increasing number of ECUs over a shared communication medium. Parallel with that rises the need of implementation of external stand-alone software applications that provide monitoring and controlling functionalities and testing the capabilities of the system. As a part of the Urban Mobility X-by-Wire(less) project, introduced by the HAW Hamburg [1], the current project is focused on the development of such software application that represents a platform for establishing a connection and providing a bidirectional communication between a computer and a microcontroller device. The microcontroller is assumed to be a part of a system, exchanging data according to the FlexRay communication protocol [2] standards.

FlexRay is one of the latest communication standards used in automotive applications. It is popular with its increased bandwidth (compared with its predecessors) and is used in applications that require a real-time communication, combining both event and time triggered Media Access Control (MAC) mechanisms for better use case adaptability. FlexRay communication provides high fault-tolerance implemented in dual channelling (the two channels are referred as Channel A and Channel B). Every ECU, part of a FlexRay system, is referred as a node and can be connected to only one or both channels, depending on its purpose requirements. All nodes (up to 64) connected to the same wire (bus) form a FlexRay cluster.

The current project aims to design and implement a software application that enables sending and receiving data to and from FlexRay nodes, based on the TCP/IP communication standard. The hardware connection between the computer and the nodes of a FlexRay cluster is realized with the help of cables and a network switch [3] (Fig 1.2). The switch is responsible for the correct addressing of data packets to the connected nodes without further hardware interventions. To enable wider range of ECU types with different architecture and manufacturer, the structure of the communication data objects has to be determined and standardized according to the project requirements.

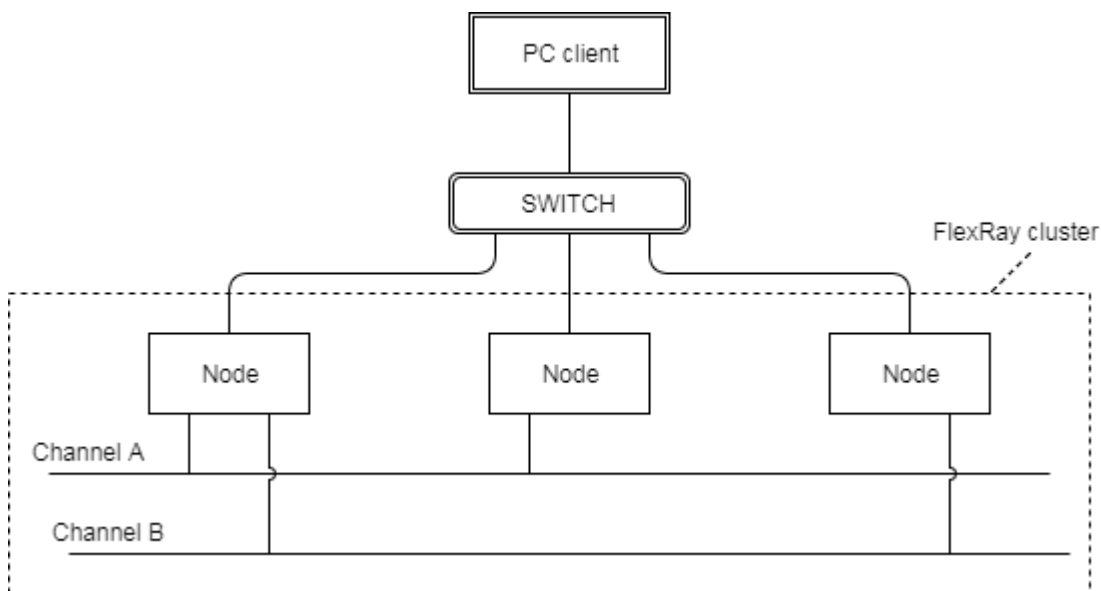


Fig 1.2 TCP/IP connection between a PC client and a FlexRay cluster via a network switch

The motivation for starting this project is provoked by the necessity of having a single software application that besides performing initial configuration, control and monitoring tasks represents an evaluation tool for the microcontroller device capabilities and can be used as a benchmark for testing the correct operation of an automotive network.

2. Fundamentals

This chapter represents a technical overview of the fundamental, for this project, topics. The discussions are based on a various scientific and public sources that can be found online or in the library.

2.1 Introduction to Bus Systems

In computing science a system bus is defined as a pathway that is used for data transfer between the Central Processing Unit (CPU), the main memory and other peripheral components [4]. It is usually composed of cables and connecting units and is defined regarding the hardware system specifications and data exchange standards, established by the communication protocol that is being used in the system. There are two types of system bus implementation – parallel and serial. The parallel bus (Fig 2.1 (a)) is advantageous in terms of speed but it requires more hardware as for every group of data there is a separate line. By the serial bus implementation (Fig 2.1 (b)) there is only one line connecting the components but the data transmission time is increased.

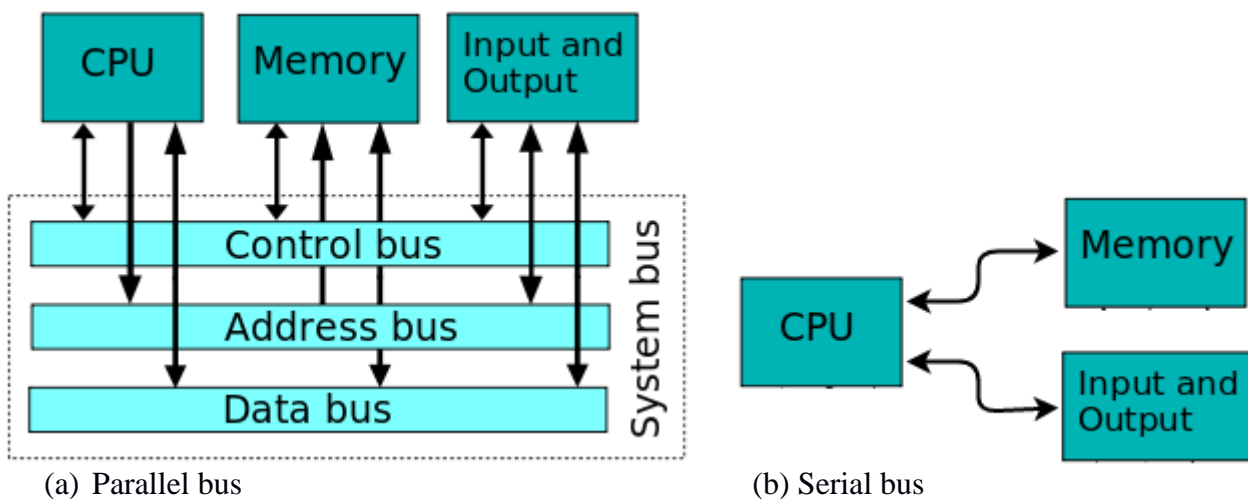


Fig 2.1 System bus [4]

The bus system represents a (usually) standardized interface that links the hardware and software interfaces of an electronic control unit (ECU) and provides mechanisms to establish and control communication between the internal components of a system or between different systems. It refers to the mechanical, electrical, functional and logical aspects of data transfer that includes communication over different mediums, linked in various network topologies. Nowadays bus systems are used in almost every industry field and tempt to develop higher and more reliable data rates for faster and secure communication.

The use of bus systems for the automotive industry dates back to the early 80s when automobiles were first equipped with ECUs [5]. Since then the development of automotive electronics has rapidly increased, leading to the necessity of introducing different communication bus systems to serve various specific technical and economical requirements. Some of the most popular communication standards for the automotive technologies are: Controller Area Network (CAN), Local Interconnect Network (LIN), Media Oriented System Transport (MOST) and FlexRay.

Bus systems can combine one or more network topologies and provide various interaction structures, like: client-server, master-slave, producer-consumer, multi-master, demand-based, time triggered and so on. Communication in a Client-Server communication system is

characterized by a strict differentiation between the service requester (Client) and the service provider (Server) [5].

Typically, bus systems are related to the physical and data link aspects of a communication process. Sometimes, depending on implementation, it can involve some application specific aspects. That means that it can be looked as an independent three-layered structure [6] that is best explained by the standardized 7-layered OSI model.

2.2 OSI Model

The Open Systems Interconnection (OSI) model is a networking standard, defined by the International Standards Organization (ISO) in 1984 [7], [8], [9]. It is designed to represent the networking framework as a hierarchical structure, separated in seven layers, depending on their relation with different software and hardware aspects of the communication process. The lower layers, from 1 to 4, are responsible to physically move the data around (Fig 2.2). They are also called data-flow layers. The upper layers, from 5 to 7, are related to the applications processing the data. Every data segment that is received or transmitted over the network passes through all 7 layers in opposite directions. When transmitted, data goes from the 7th layer down to the 1st, where it is sent as physical impulses over the network medium. When received, those impulses are processed from the 1st up to the 7th layer, where they are represented in the desired by the application or end-user format.

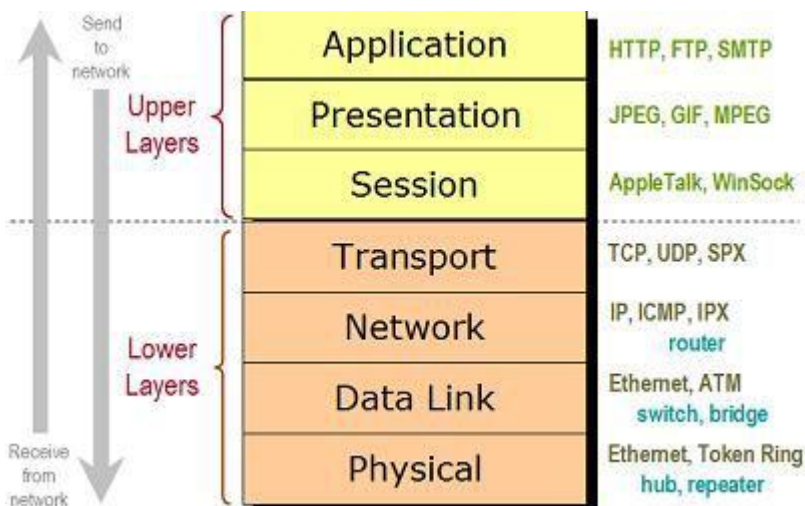


Fig 2.2 Seven layer OSI model [7]

Every layer from 2 to 6 upgrades the preceding one and is a base for the next layer. To get a better understanding of the seven layer OSI networking model, each of its layers is discussed in details:

1. **Physical** layer – this layer refers to the hardware medium that provides the electrical and mechanical interface, such as wires, connectors, hubs, repeaters, etc. Its basic functions are related to handling the electrical voltage impulses, light or radio signals that represent the data bits on a physical level.
2. **Data Link** layer – the second OSI layer is responsible for linking the data from the physical layer into block units (packets, frames) and to perform flow and error control over the transmission links. It can be divided into two sub-layers – Media Access Control (MAC) and Logical Link Control (LLC).

3. **Network** layer – this layer establishes the route between transmitter and receiver. It is responsible for routing and forwarding the data, addressing (IP), error handling, congestion control and packet sequencing.
4. **Transport** layer – as the name supposes, this layer is responsible for the correct transportation of data over the network. It performs error checking and recovery and in cases of transmission error may request retransmission of packets. To this layer belong the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP).
5. **Session** layer – the lowest application layer controls the start and end of transmission and provides mechanisms for managing the process of data exchange on the network. The three session categories are: simplex, half-duplex and full-duplex.
6. **Presentation** layer – at this layer data is encoded/decoded from raw data to the desired by the application format and vice versa. It is also called the syntax layer.
7. **Application** layer – the top OSI layer provides network services for the application software. The network services provide and request data to and from the Presentation layer. Typical Application layer examples are HTTP, FTP, Telnet, etc.

2.3 FlexRay Protocol

The FlexRay protocol is an automotive standard defined by the FlexRay consortium in 2005 that combines the event-driven paradigm of the Controller Area Network (CAN) bus and the time-driven design of the Time-Triggered Protocol (TTP) in one protocol [10]. It is intended for applications with high requirements regarding determinism, reliability, synchronisation and bandwidth. The FlexRay system consists of at least two interconnected electronic control units (ECU) running the FlexRay protocol. Every ECU in a FlexRay system is referred as a **node**. Up to 64 nodes, connected with one or two lines form a **cluster**. The network structure in a cluster can be based on bus topology, star topology or a mixture of both.

FlexRay provides services related to the lowest two layers of the OSI model, which is a subject of the following two sub-sections, starting with the Physical Layer 1 (2.3.1 Communication Controller) and continuing with the Data Link Layer 2 (2.3.2 FlexRay Communication). In the third sub-section of this chapter (2.3.3 E-Ray) is introduced the developed by Bosch E-Ray [11], as a version of an IP module running the FlexRay protocol.

2.3.1 Communication Controller

Every FlexRay device (node) has a protocol engine component that implements the FlexRay protocol and an engine control unit, where the application software is running [10]. The latter is referred as a **Host** and it provides control and configuration data to the protocol engine, referred as **Communication Controller** (CC). The CC responds with status conditions and the received on the bus data (Fig 2.3). The Host also controls the operating modes of the bus driver and reads status and error conditions.

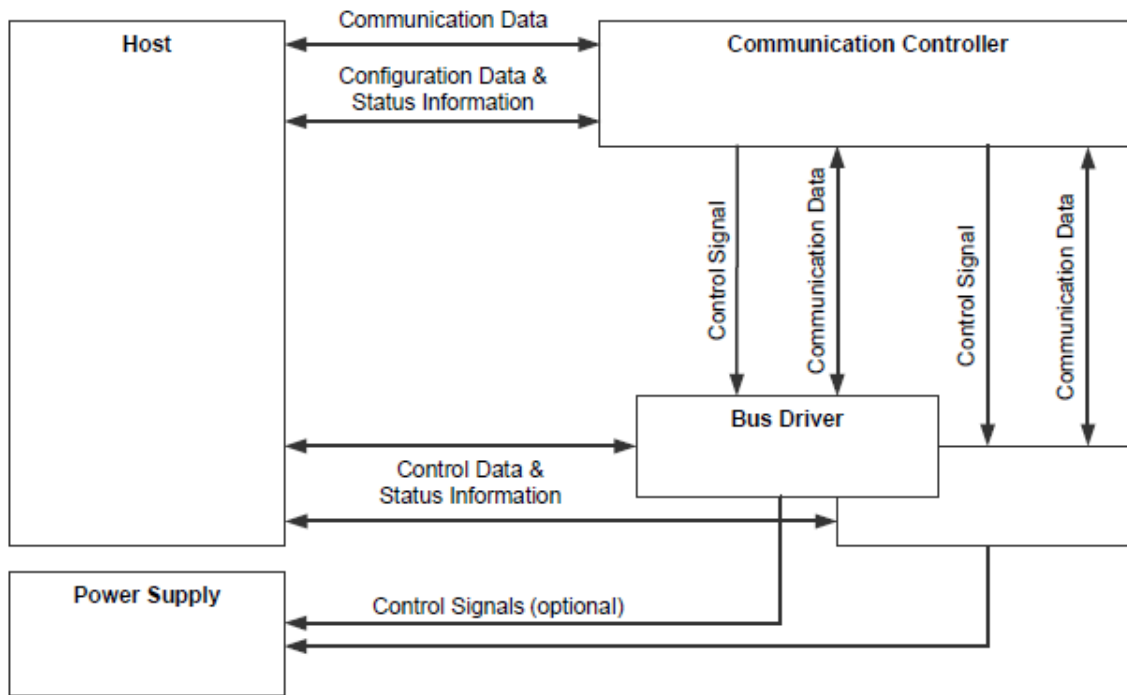


Fig 2.3 Logical interfaces in a FlexRay node ([10] p. 26)

The Host is separated from the FlexRay protocol engine via the **Controlling Host Interface (CHI)** [10]. The CHI provides means for the Host to operate the **Protocol Operation Control (POC)** in a structured manner, transparently to the operation of the protocol. It is product specific and its implementation depends on the node's architecture. In general, the CHI provides two major interface blocks – protocol data interface (PDI) and message data interface (MDI). The PDI manages the configuration, control and status data of the protocol, while the MDI manages the message buffers and the configuration, control and status data related to them (Fig 2.4).

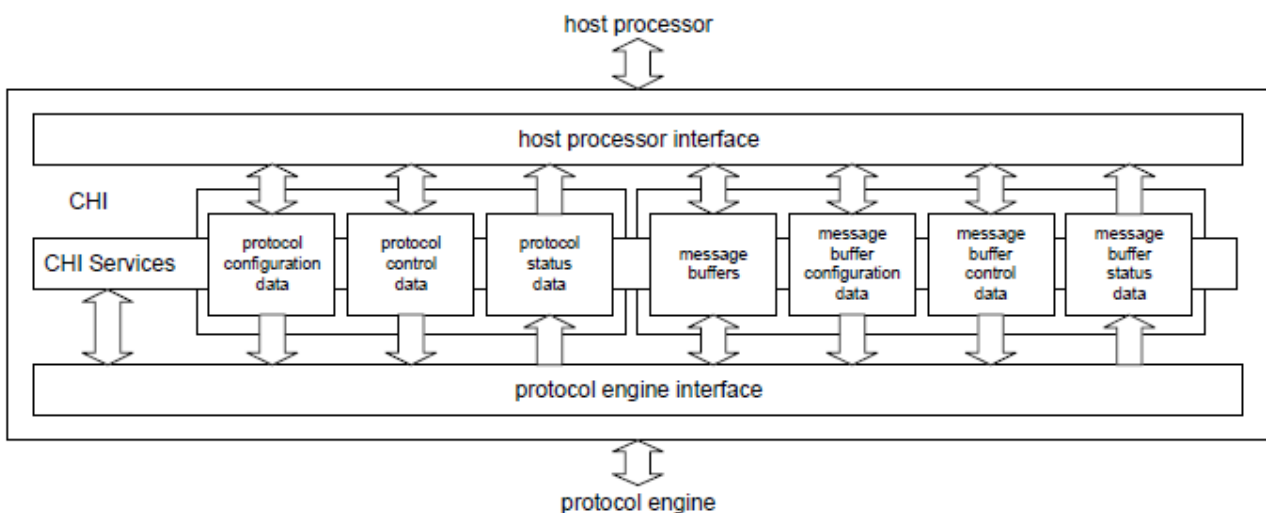


Fig 2.4 Conceptual architecture of the Controlling Host Interface ([10] Figure 9-1)

The purpose of the POC is to apply Host commands, invoke and react to model changes of the core protocol mechanisms¹ and provide the Host with protocol status information in a synchronized manner. For a more detailed view of the relationship between CHI, POC and the core protocol mechanisms refer to Fig A.1 in Appendix A.

The CC has to be in an established POC state to enable the execution of the POC processes [10]. The CC enters *POC Operational*² power state once sufficient power supply is present or after reset. This state determines the operation of the POC. Its structure is based on the finite state machines (FSM) definition [12], where state transitions are internally or externally provoked by a certain control condition and entering a state is enabled only from other state that is logically connected to it. A high-level internal structure of the *Operational* state with all possible state transitions is shown on Fig 2.5.

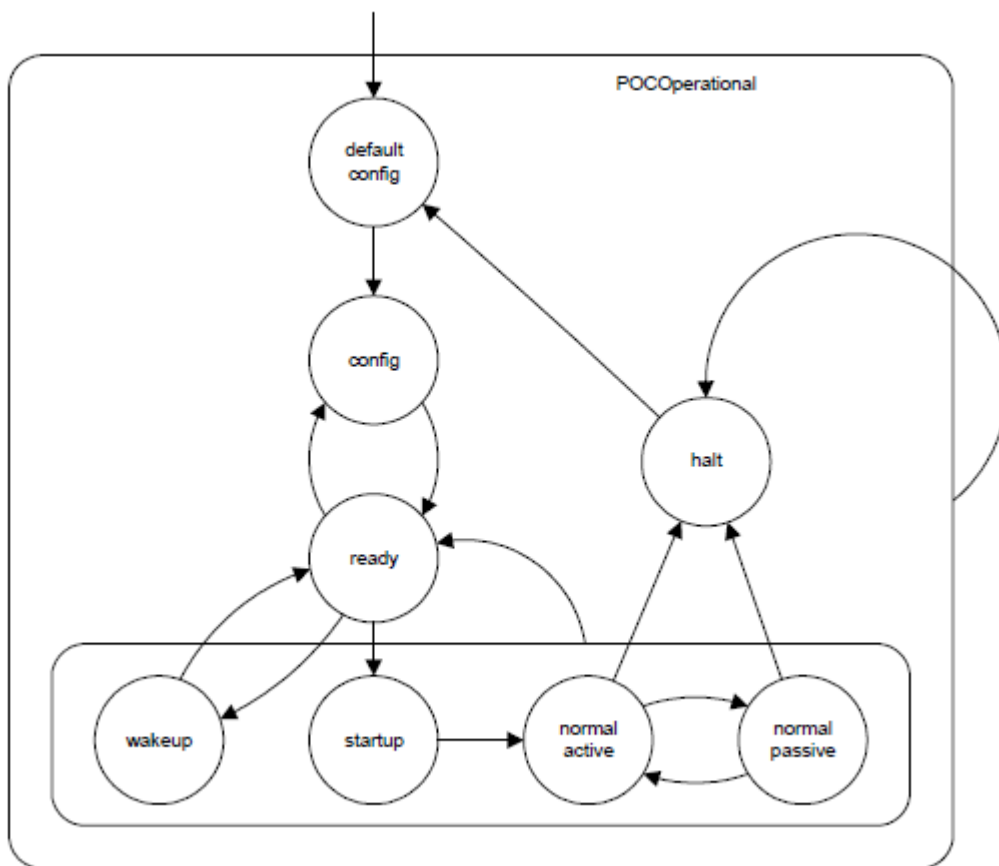


Fig 2.5 Overall state diagram of a FlexRay communication controller ([10] p. 37)

State transitions are typically caused by commands sent to the CC from the Host via the CHI or they are a consequence of completing a task or error condition, occurred by the protocol engine, product-specific built-in self test or sanity checks. The Host is allowed to apply a CHI command at any time but some commands are enabled only when the CC is in a certain state. Applying a command when the controller is in a non legal for that command state has no effect on the operation of the CC, however, the command vector in the specified register is reset and an error flag is raised.

¹ The primary FlexRay protocol principles are realized in four core mechanisms – 1) coding and decoding; 2) media access protocol; 3) frame and symbol processing; 4) clock synchronisation [10]

² In FlexRay signature standards state names are always preceded by *POC* abbreviation. For sake of simplicity it will be skipped in this text. POC states are written in *green italic* and CHI commands in CAPITALS.

³ For distinguishing between different types of parameters signature refer to Table A-1 in Appendix A

The Host may write configuration data to the CC registers only when the CC is in *config* state and its registers are locked for access. Transition to *config* state is only possible when the CC is in *default config* or *ready* state. The *default config* state is entered immediately on power up or after hardware reset. In this case the configuration and message buffers data is automatically cleared. However, when *default config* state is entered from *halt* state, clearing the message RAM has to be explicitly requested by the Host via applying the CLEAR_RAMs command. Both *default config* and *config* states are entered after applying command CONFIG when the CC is in legal for that transition state. Applying a command triggers a reaction in the POC immediately or at the end of the current cycle. All relevant FlexRay CHI commands and their allowed invocation states are shown in Table 2-1.

CHI command	Where processed (POC States)	When
ALL_SLOTS	<i>POC:normal active, POC:normal passive</i>	End of cycle
ALLOW_COLDSTART	All except <i>POC:default config, POC:config, POC:halt</i>	Immediate
CLEAR_RAMs	<i>POC:default config, POC:config</i>	Immediate
CONFIG	<i>POC:default config, POC:ready</i>	Immediate
DEFAULT_CONFIG	<i>POC:halt</i>	Immediate
FREEZE	All	Immediate
HALT	<i>POC:normal active, POC:normal passive</i>	End of cycle
READY	All except <i>POC:default config, POC:config, POC:ready, POC:halt</i>	Immediate
RUN	<i>POC:ready</i>	Immediate
WAKEUP	<i>POC:ready</i>	Immediate

Table 2-1 CHI commands summary ([10] p. 33)

2.3.2 FlexRay Communication

The nodes in a FlexRay cluster are connected with one or two channels, referred as *channel A* and *channel B*, each supporting data rates of up to 10 Mb/s [10], [13], [11]. The controller can send data on both channels simultaneously and independent on each other. A node can receive messages only via the channel(s) it is connected to. Received data is stored in the node message RAM, which structure is based on message buffers, dedicated to transmission (transmit buffers) or reception (receive buffers) process. Part of the receive buffers can be configured in a cyclic First-In-First-Out (FIFO) buffer structure.

Communication between nodes in a FlexRay cluster is based on the Time Division Multiple Access (TDMA) scheme, organized in *communication cycles* that are periodically executed. This scheme guarantees a collision-free communication as there is no competition over the channel access. Each communication cycle is divided in time slots. All nodes in a cluster are assigned time slots in which they are allowed to send data frames.

Communication Cycle

The media access control (MAC) mechanism in FlexRay is based on recurring communication cycles, that combine the time and event triggered data exchange between nodes [10], [13]. Each communication cycle consists of a static segment and a network idle time (NIT). Optionally, it can contain a dynamic segment and/or a symbol window. The structure of a FlexRay communication cycle is shown on Fig 2.6.

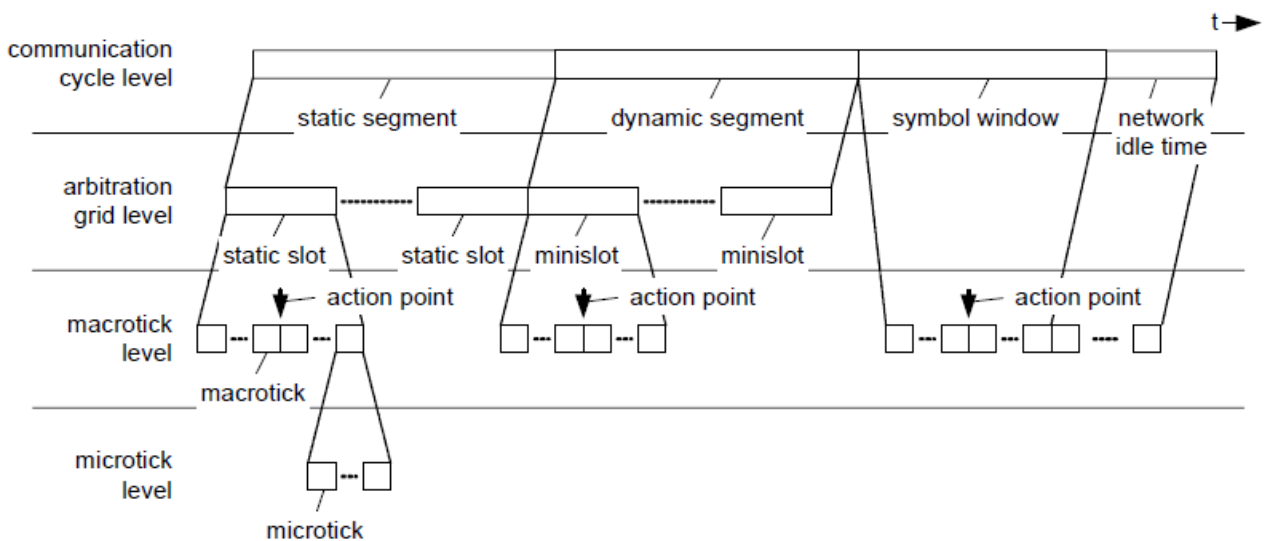


Fig 2.6 Timing hierarchy within the communication cycle ([10] p. 100)

The static segment is based on a static TDMA scheme and is always present in a communication cycle. This scheme is realized in static time slots, which number and length is fixed and is the same for both channels in a cluster (Fig 2.6). The length of the static slots is derived from a predefined number of global clock ticks, referred as *macroticks*. The number of macroticks in the static slot is chosen to be big enough to ensure that the frame and any potential safety margins fit under worst-case assumptions. Each macrotick consists of an integer number of *microticks* that are derived from a local for the node clock (e.g. Host CPU), which number and duration can differ in every node. The macrotick boundary that determines the start of a frame transmission is referred as *action point*.

Contrary to static segments, dynamic segments are present only if the controller has data to send there. The data sent in the dynamic segments is priority and event driven and can be of varying lengths. The structure of the dynamic segment is based on a dynamic TDMA *minislot* scheme (Fig 2.6). Similarly to the static slot the minislot is constituted of a predefined amount of macroticks. The number of minislots and their constituent macroticks is identical in every node of the cluster. The number of macroticks in a minislot is, however, smaller than in the static slot so that if no frame transmission takes place in the dynamic segment less bandwidth is wasted (Fig 2.7 (a)). If data is transmitted in the dynamic segment, the size of the minislot is expanded to fit the transmitted frame (Fig 2.7 (b)). In this case the minislot is referred as a *dynamic slot* and it consists of an integer number of minislots. The numbering sequence of the minislots is kept but their total number per dynamic segment is reduced. A frame is transmitted only if the remaining minislots are enough to fit it. That means that if a frame is assigned a large minislot number, it might have to wait for another communication cycle in order to be transmitted. Therefore, frames priority is dependable on the assigned minislot number.

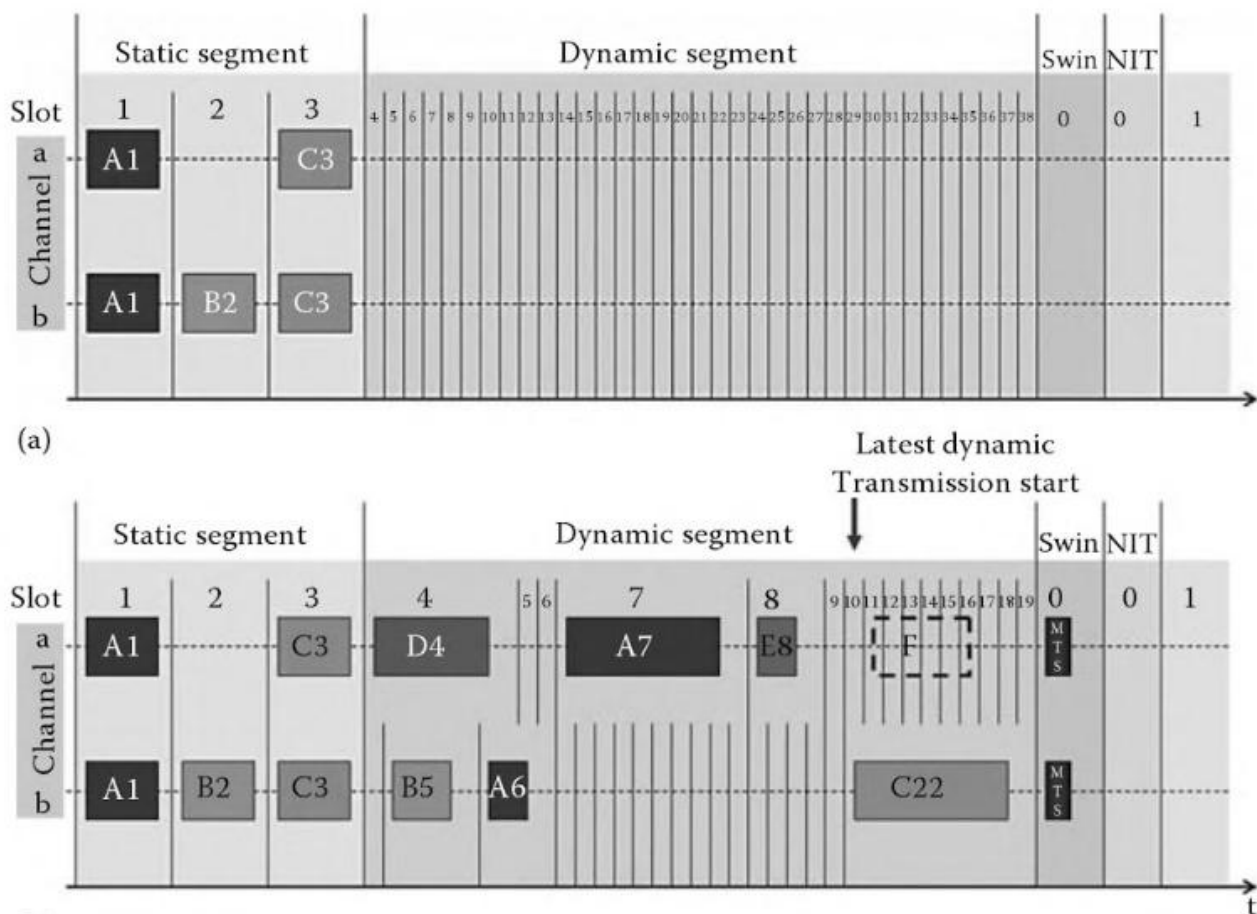


Fig 2.7 (a) Communication cycle with no transmission in the dynamic segment; (b) Communication cycle with several transmissions in the dynamic segment ([13] Figure 5.5)

The presence of a symbol window in a communication cycle is optional. Within the symbol window only one symbol can be sent and no arbitration between different senders is provided. Its size consists of an integer amount of macroticks that is the same for all nodes in a cluster.

The NIT serves for clock correction and synchronization, as well as for implementation specific communication cycle related tasks. It contains the remaining macroticks that are not assigned to the other three components of the communication cycle.

Configuration Timing Constraints for Communication Cycles

This sub-section is intended to reveal the duration limitations for each of the communication cycle segments by introducing the related FlexRay constants and parameters – their minimum and maximum values, according to the FlexRay protocol specifications v2.1 [10] and the hardware restrictions for each node (speed of processors, oscillators, etc).

Communication is based on recurring communication cycles, which number is configured in the protocol constant³ *cCycleCountMax* and ranges between 0 and 63. The number of macroticks per communication cycle is between 10 and 16000 and is stored in the *gMacroPerCycle* parameter. The *pMicroPerCycle* parameter holds the number of microticks per communication cycle and its value is calculated for every node, depending on the duration of the microtick.

The arbitration scheme for transmitting FlexRay frames is realized via unique assignment of frame IDs to the nodes in a cluster for each channel. The frame ID determines in which slot and respectively in which segment a frame shall be sent. Frame IDs range from 1 to the *cSlotIDMax* protocol constant, which value can be up to 2047.

Each channel keeps track of the time slots (static and minislots) in its own variable *vSlotCounter* that starts from 1 in the beginning of every communication cycle and is incremented by 1 with every new slot. This slot counter is increased simultaneously for both channels in the static segment of the communication cycle, while in the dynamic segment both counters are incremented independently, according to the arbitration scheme used there.

The number of static slots, composing the static segment, is stored in the *gNumberOfStaticSlots* parameter. It ranges between 2 and the value assigned to the *cStaticSlotIDMax* protocol constant, which maximum value is 1023. Static slot duration ranges between 4 and 661 macroticks and is stored in the *gdStaticSlot* parameter. The action point offset within the static slots is assigned to the *gdActionPointOffset* parameter and ranges between 1 and 63 macroticks.

The presence of a dynamic segment in the communication cycle is optional and therefore the number of minislots can be 0 and up to 7986. It is stored in the *gNumberOfMinislots* parameter. The *gdMinislot* parameter keeps the duration of the minislot that ranges between 2 and 62 macroticks. The number of macroticks, constituting the offset of the action point within a minislot is stored in the *gdMinislotActionPointOffset* parameter and has a value between 1 and 31. The number of the last minislot in which a transmission in the dynamic segment can be started is configured in *pLatestTx* node parameter.

The symbol window duration can be up to 142 macroticks and is stored in the *gdSymbolWindow* parameter. Its value can be 0 if no symbol window is required. The action point offset is stored in the same parameter as for the static segment. The number of macroticks for the NIT is between 2 and 805 and is stored in the *gdNIT* parameter.

³ For distinguishing between different types of parameters signature refer to Table A-1 in Appendix A

Frame format

A FlexRay frame consists of three segments: header, payload and trailer segment (Fig 2.8) [10], [13]. Header and trailer segments have a fixed length of 5 and 3 bytes respectively, while the payload segment length varies from 0 to maximum of 254 bytes. The frame is transmitted in the introduced segment order, starting always with the most significant bit (MSB) and followed by the subsequent bits.

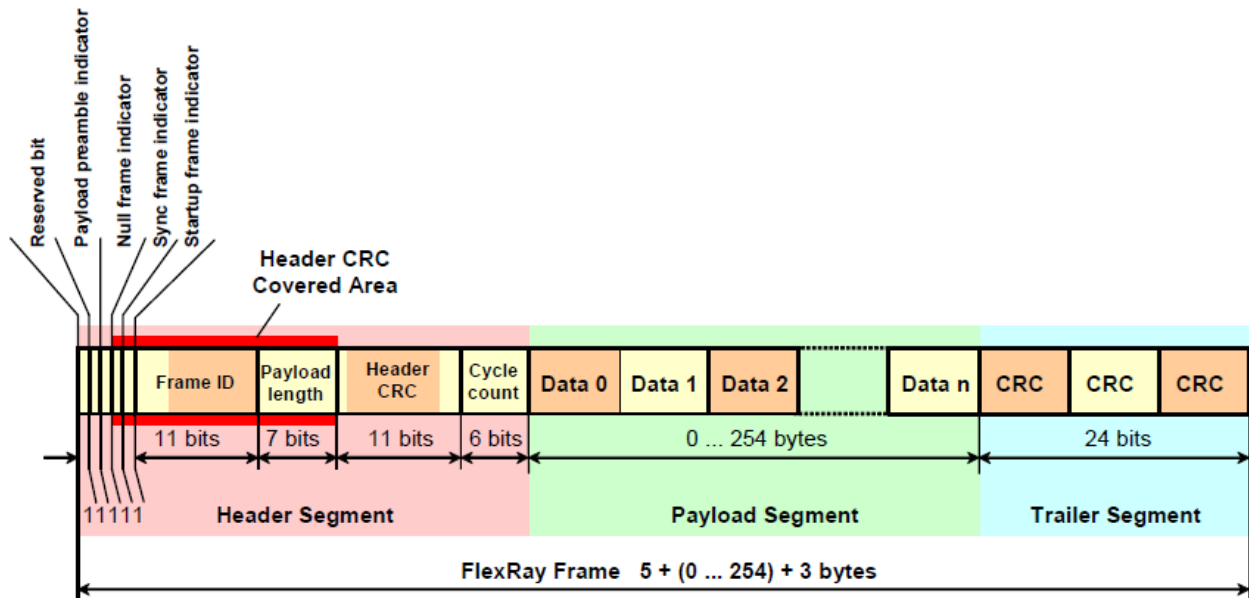


Fig 2.8 FlexRay frame format ([10] p. 90)

The frame is transmitted, starting from the header segment. The header MSB is reserved and has no relevance. It is followed by the *payload preamble indicator* (PPI) that indicates the presence of a network management (NM) vector (when frame is transmitted in the static segment) or message ID (when frame is transmitted in the dynamic segment) in the beginning of the payload. The *null frame indicator* bit signals whether the payload segment contains relevant data⁴. The *sync frame indicator* bit is used for clock synchronisation of all nodes in a cluster when set to 1. Accordingly, the *startup frame indicator* bit determines whether the frame is a startup frame. It is usually set to 1 only when the sync frame indicator bit is also 1. In the following 11 bits is coded the *frame ID*, which determines the slot in which the frame shall be transmitted. It must be a unique number between 1 and 2047 for every channel in the cluster. The *payload length* is coded in the following 7 bits, representing the number of 2-byte data words, i.e. the payload length number equals the actual payload length in bytes divided by 2. The sync frame indicator, startup frame indicator, frame ID and payload length are protected by an 11 bits *cyclic redundancy check* (CRC) code. For transmitted frames, the CRC code is not calculated by the transmitting CC but it is provided by means of configuration. The header CRC code is calculated by the CC only for received frames. The last 6 bits of the header section specify the value of the *cycle counter* at the time of frame transmission.

The payload segment of a FlexRay frame contains the actual message payload, starting from the first data byte (MSB to LSB order) and followed by the subsequent data bytes. The product specific host interface maps the position of data bytes in the buffer with their position in the payload

⁴ A *null frame* is a frame with null frame indicator bit set to '0' and a payload length field set also to '0' [10]

segment of the frame. As the payload length field in the header segment holds the number of 2-byte data words, the number of data bytes in the payload segment is always even. To achieve that, sometimes padding is applied.

The last FlexRay frame segment is the trailer segment. It consists of 24 bits CRC code that covers all bits of the frame, including the header CRC and payload padded byte. Both channels use the same generator polynomial for the computation of the CRC but different initialization vectors – 0xFEDCBA for channel A and 0xABCDEF for channel B.

Message filtering

Message filtering is based on *slot number*, *channel* and optionally on one or more cycle numbers (*cycle set*) [11]. The filtering configuration data is part of the header section of the message buffers, which are discussed in RAM Configuration section in Appendix A.

Every message buffer is assigned a slot number and one or two channels in which it is allowed to transmit or receive data. The slot number is encoded in the 11-bit message buffer frame ID field and needs to be greater than 0. The frame ID is compared against the slot counter variable of the corresponding channel(s). Every static slot belongs to only one node, regardless of whether cycle number filtering is provided or not.

Channel filtering is provided via 2-bit field – one bit for each channel. It serves as filtering mechanism for receive buffers and control mechanism for transmit buffers. Only for the static segment it is allowed both channels to be assigned. If both channels are assigned for the dynamic segment, then no frames are transmitted/received on any channel. It is equivalent as if no channels were assigned for the dynamic segment.

The cycle set filtering is encoded in the 7-bit cycle count field of the message buffer header and is used to distinguish between different message buffers, belonging to the same node, that are assigned the same channel(s) and frame ID. The filter is passed every time a cycle number matches an element, belonging to the assigned cycle set (Table 2-2).

Cycle Code	Matching Cycle Counter Values		
0b000000x	all Cycles		
0b000001c	every second Cycle	at (Cycle Count)mod2	= c
0b00001cc	every fourth Cycle	at (Cycle Count)mod4	= cc
0b0001ccc	every eighth Cycle	at (Cycle Count)mod8	= ccc
0b001cccc	every sixteenth Cycle	at (Cycle Count)mod16	= cccc
0b01ccccc	every thirty-second Cycle	at (Cycle Count)mod32	= ccccc
0b1cccccc	every sixty-fourth Cycle	at (Cycle Count)mod64	= ccccccc

Table 2-2 Definition of cycle set ([11] Table 9)

In order for a message buffer to be allowed to transmit or receive data, all filters must match. If more than one buffer is assigned the same frame ID, cycle number and channel, the message buffer with lowest message buffer number is chosen. The receive FIFO can be configured for further delimitation of the received messages via the FIFO Rejection Filter (FRF). Apart from filtering, based on frame ID, channel and cycle count, the FRF can be configured to reject/accept all messages in the static segment and/or null frames. In the FIFO Rejection Filter Mask (FRFM) are pointed those bits of the frame ID that are marked as “don’t care” for the FRF.

Transmit process

A message can be transmitted in the static segment of a communication cycle on channel A, channel B or both channels simultaneously [10], [11]. If few messages are pending, the one with frame ID corresponding to the next time slot is sent next. When transmitted in the dynamic segment, a message can be sent on channel A or channel B only, thus allowing concurrent messages to be sent simultaneously on both channels. The message with highest priority (lowest frame ID) is sent next only if its length fits in the remaining minislots of the current cycle. Transmission takes place in every time slot of the static segment. If there is no assigned buffer with matching filter criteria for a given static slot or that buffer does not have its transmission request (TXR) flag set (i.e. TXR = '0'), a null frame is transmitted. Null frames are not transmitted in dynamic segments.

Receive process

Analogously to the transmit process, messages can be received on one or both channels, when transmitted in the static segment, or on only one channel, when transmitted in the dynamic segment of a communication cycle [10], [11]. If a received frame passes all filter criteria, it is saved in the designated receive buffer apart from its frame CRC field. In this case a flag is raised to inform the communication controller that the new message is ready to be processed. If a frame is not processed by the time a new frame, designated for the same message buffer arrives, that frame is overwritten and lost. A message lost (MLST) status flag is raised in such cases. If no frame, null frame or corrupted frame is received and has passed the filter criteria, it is not saved, however the buffer status data is updated. In cases where the payload data length of the received frame is larger than the length, configured for that buffer, the received payload is truncated to the configured length.

2.3.3 E-Ray

[11] E-Ray is an electronic device that has an IP module installed and performs communication according to the FlexRay protocol specification v2.1 [10]. It supports the provided by FlexRay data rates of up to 10 MB/s on each channel. E-Ray registers can be directly accessed by an external Host via the controlling host interface (CHI) to directly perform configuration, control and monitoring tasks. An additional bus driver hardware is required for connection to the physical layer.

The E-Ray module is provided with 8kB configurable message RAM consisting of maximum 128 message buffers that can be configured to hold up to 254 bytes of data⁵ (Fig 2.9) [11]. Message buffers can be configured as static or static + dynamic buffers, dedicated to transmit or receive process. Part of the receive buffers can be organized in a cyclic First-In-First-Out (FIFO) structure. Message RAM is accessed by the Host via the provided Input Buffer (IBF), for write access, and Output Buffer (OBF) for read access. Access to the Physical Layer is under the control of Channel Protocol Controller (PTR A/B) and supported by transient buffers (TBF A/B) for intermediate message storage. All functions regarding handling of messages are implemented in the Message Handler. E-Ray is supported by an 8/16/32-bit generic CPU interface that enables compatibility with a wide range of customer-specific Host CPUs.

⁵ 128 message buffers with max of 48 bytes data section or up to 30 message buffers with 254 bytes data section [11] (p. 14)

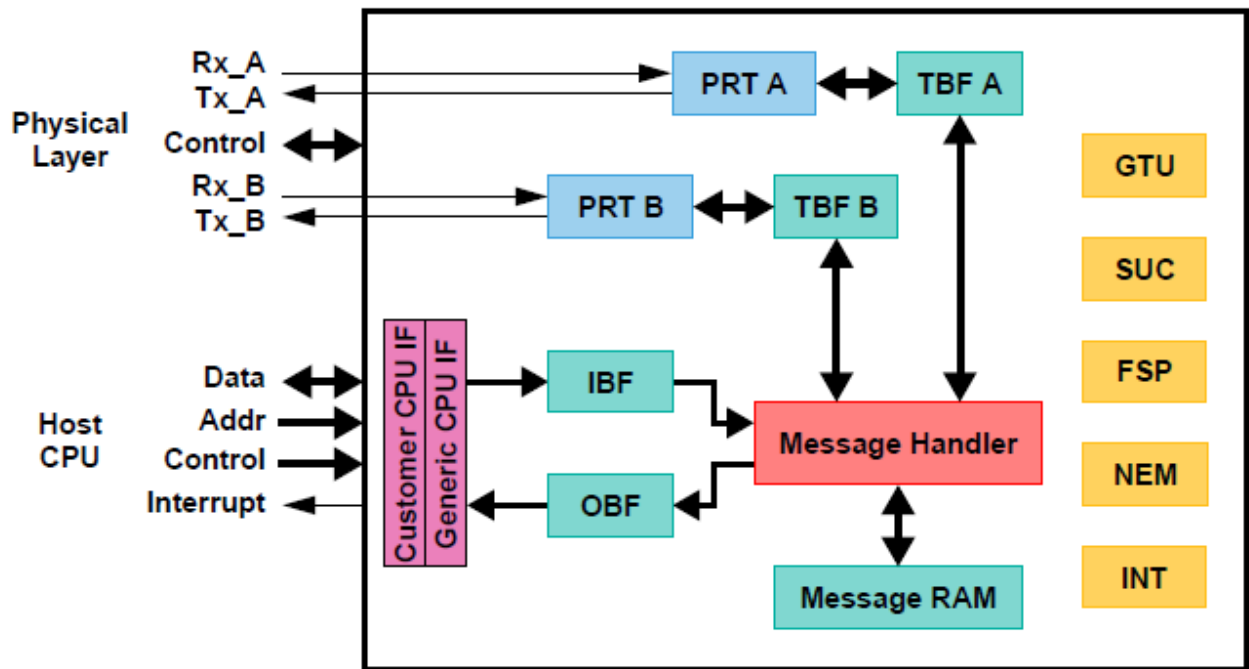


Fig 2.9 E-Ray block diagram ([11] Figure 1)

The FlexRay *channel protocol controllers* (PRT A and PRT B) represent interfaces for controlling the transmission and reception of FlexRay frames and symbols between the internal device memory and the connected channel(s) [11]. They consist of shift registers and FlexRay protocol FSM and perform functionalities for control of the bit timing, generation of the frame CRC and check of the received header and frame CRC codes. The FlexRay channel protocol controllers have interfaces to:

- Physical Layer (bus driver)
- Transient Buffer RAM (TBF A/B) – stores the data section of two complete messages.
- Message Handler – controls data transfer between IBF/OBF and message RAM and between TBF A/B and message RAM.
- Global Time Unit (GTU) – performs generation of micro and macroticks; fault tolerant clock synchronization and support of external clock correction; timing control of static and dynamic segments; cycle counter.
- System Universal Control (SUC) – provides control over: configuration, wakeup, startup, normal operation, passive operation and monitor mode.
- Frame and Symbol Processing (FSP) - checks the correct timing of frames and symbols; tests the syntactical and semantical correctness of received frames; sets the slot status flags.
- Network Management (NEM) – handles the network management vector.
- Interrupt Control (INT) – provides error and status interrupt flags; enables/disables interrupt sources and module interrupt lines; assigns interrupt sources to one of the two module interrupt lines; manages the two interrupt timers; captures the stop watch time.

The message RAM in E-Ray is structured in 2048 words of 33 bits⁶, making a total of 67 584 bits memory space [11]. It provides support for 8 and 16-bit accesses and is able to store up to 128 message buffers, depending on the configured payload length, as the data bytes in the FlexRay frame can vary between 0 and 254. In order to achieve a better flexibility, the message RAM is divided in Header and Data partitions and has the structure shown in Fig 2.10.

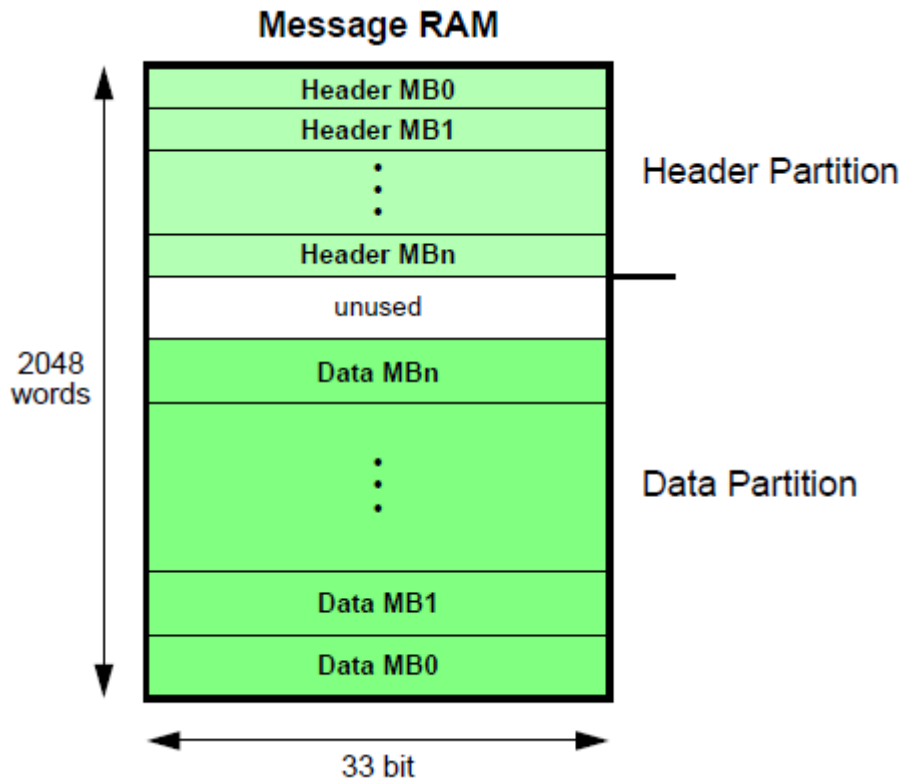


Fig 2.10 Configuration example of message buffers in the Message RAM ([11] Figure 15)

The *Header Partition* holds the header sections of the configured message buffers. The header section of each message buffer consists of four 32+1 byte words starting with the first word in the message RAM for message buffer 0. The *Data Partition* starts after the last word, occupied by the Header Partition. Its minimal allowed position is: $(\text{the number of last configured buffer} + 1) * 4$. In the Data Partition is stored the data section of each message buffer. For a detailed view of the header and data sections, refer to Message RAM section in Appendix A.

The E-Ray *Message Handler* is an interface that provides functionalities for controlling the data transfers between the Host and the message RAM (via the Input and Output Buffers) and between the PRTs and the message RAM (via the Transient buffers). Those functionalities include the acceptance filtering of received messages, the maintaining of the transmission schedule, as well as the providing of message status information.

⁶ 32 bit word plus 1 parity bit

Channel protocol controller access to message RAM

The FlexRay channel protocol controllers (PRT A/B) are connected to the *transient buffer* RAMs (TBF A/B) and to the physical layer via the bus driver [11]. The TBF is used as an intermediate storage, able to store two complete FlexRay messages. It is built-up as a double buffer – one assigned to the corresponding PRT and the other accessible by the Message Handler (Fig 2.11).

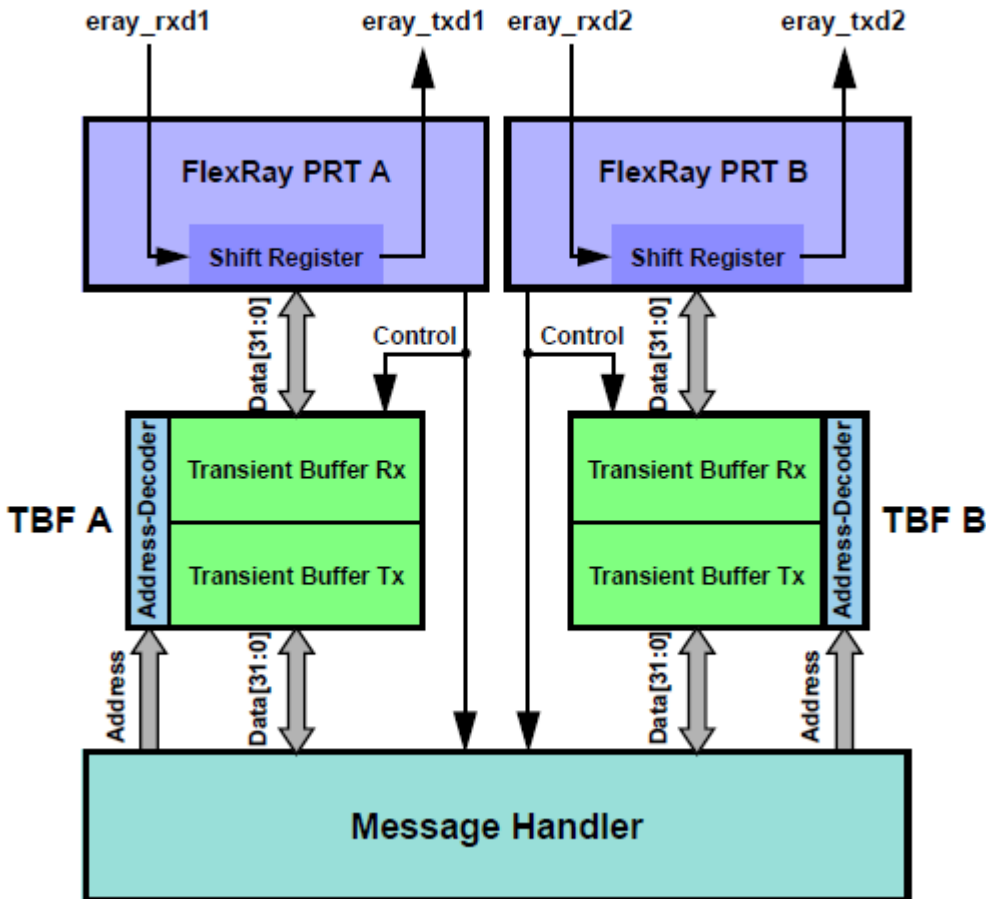


Fig 2.11 Access to Transient Buffer RAMs ([11] Figure 14)

Receiving/transmitting a FlexRay message to/from the message RAM is possible via the TBFs. The Message Handler writes the message to be transmitted to the TBF Tx of the corresponding channel (Fig 2.11). The PRT writes the received on the channel message to the corresponding TBF Rx. During transmission of the message stored in the TBF Tx the Message Handler transfers the last received message from TBF Rx to the message RAM (if it passes the acceptance filtering).

Host access to message RAM

The Host accesses the message buffers in the message RAM via intermediate buffers and the help of the Message Handler [11]. Read access is done via the Output Buffer (OBF) and write access via the Input Buffer (IBF). Similarly to the Transient Buffer (TBF) the IBF and OBF have double buffer structure – one half accessed by the Host (IBF Host/OBF Host) and the other half accessed by the Message Handler (IBF Shadow/OBF Shadow). The Host writes the number of the target buffer in the message RAM to the corresponding intermediate buffer, together with other configuration data (header and/or message data), and the Message Handler proceeds its request by providing the requested access (Fig 2.12).

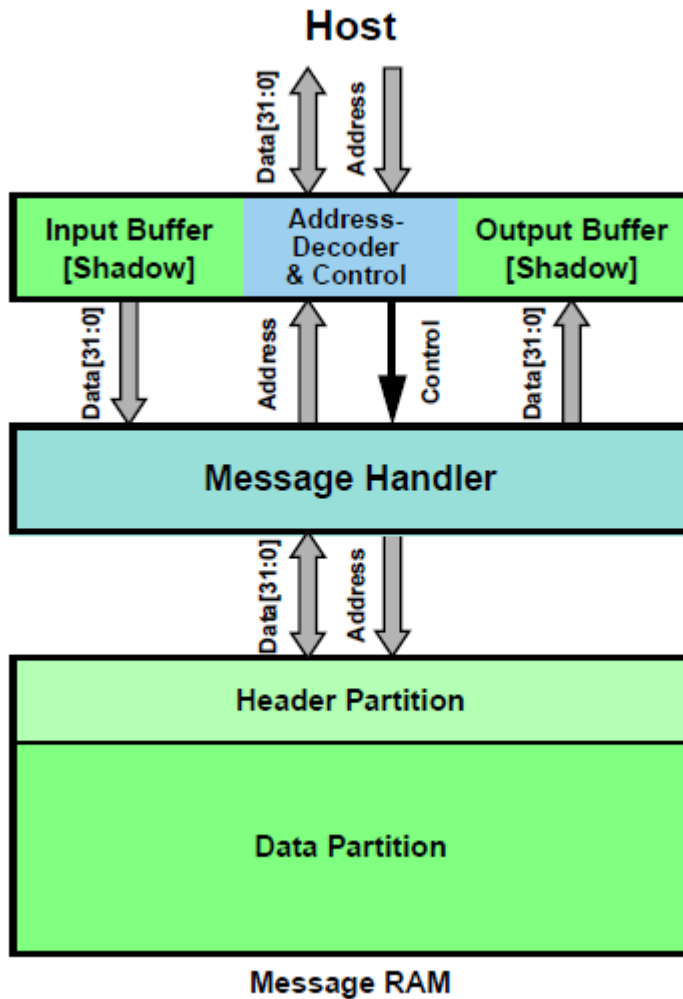


Fig 2.12 Host access to Message RAM ([11] Figure 9)

RAM Configuration

Message RAM can be configured into three groups of message buffers via the Message RAM Configuration (MRC) register, only when the CC is in *config* or *default_config* state (Fig 2.13) [11]. The first group is dedicated to messages sent in the static segment of a communication cycle. Specifying the number of the First Dynamic Buffer (FDB) determines the number of static message buffers. As the maximum allowed number of message buffers in the message RAM is 128, any number assigned to FDB greater or equal to 128 will assign all message buffers to the static segment. Following the same logic, the First FIFO Buffer (FFB) holds the number of the first FIFO message buffer. The last message buffer number is configured in the Last Configured Buffer (LCB) field of the MRC register.

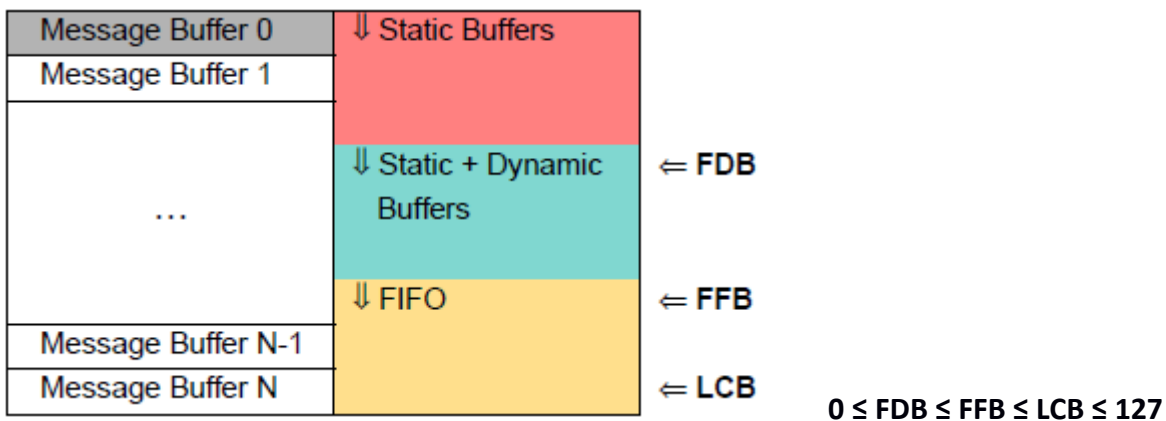


Fig 2.13 Assignment of message buffers ([11] Table 1)

A message buffer can be configured as receive or transmit buffer by configuring the CFG bit in its header section. Every transmit buffer can be configured to operate in *single-shot* or *continuous* mode by setting the appropriate value to the transmission mode (TXM) flag, also part of its header section. Part of the receive message buffers can be configured as a cyclic FIFO buffer. Every message, passing the FIFO Rejection Filter (FRF) is stored, starting with the first and proceeding with every next message buffer assigned to the FIFO. When the last buffer, belonging to the FIFO is reached, the receive process starts again with the first buffer and so on.

There are two index registers associated with the FIFO. The PUT Index Register (PIDX) points to the next message buffer to be used for receiving a frame. It is incremented every time a new message passes the FIFO filters and is saved there. The GET Index Register (GIDX) points to the next message to be processed and incremented after read access. The GIDX shall never reach the PIDX when the FIFO is in non-empty state, otherwise a FIFO overrun is observed – the message stored on that buffer index is overwritten and both indexes are incremented. In this case an error flag is raised and an interrupt is generated.

The possible FIFO states are shown on Fig 2.14.

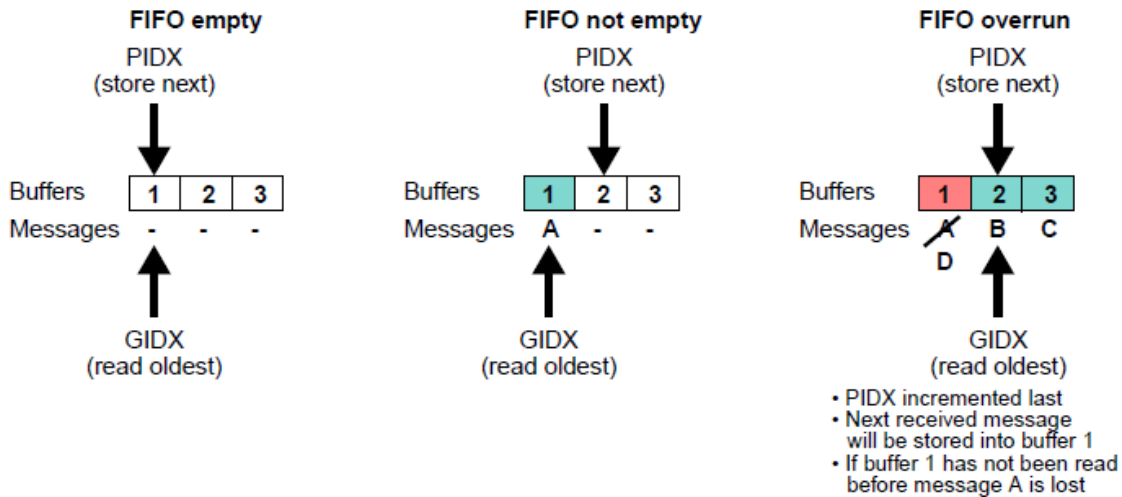


Fig 2.14 Possible FIFO states ([11] Figure 8)

An important constraint is that all buffers, belonging to the FIFO must be configured with the same payload and data section length. The programmer shall take care of correct input as the CC does not check for erroneous configuration ([11] p. 72).

Depending on its configuration, the first message buffer (with index 0) of the message RAM can hold the startup frame, sync frame or designated single slot frame. This ensures that each buffer can transmit only one startup or sync frame per communication cycle. The configuration of the 0th message buffer is possible only during configuration time when the CC is in *default_config* or *config* state but not during runtime. If enabled by the programmer in the MRC register, reconfiguration of all other message buffers is possible during runtime via the Input Buffer (IBF). However, it shall be noted that due to the data partition, reconfiguration of the payload length may lead to memory corruption and erroneous outcome.

Host Write Access via Input Buffer

The Host can request write access to one message buffer at a time via the IBF. If reconfiguration is desired, the Host writes the configuration data to the three header registers (WRHS1...3) of the IBF [11]. The actual message is written to the IBF data registers (WRDSn with n = 1...64). The option whether to update only the header, only the data, or both sections of the targeted message buffer in message RAM is specified via the Input Buffer Command Mask (IBCM) register. Lastly, the target buffer number is written to the Input Buffer Command Request (IBCR) register that triggers the swap of the IBF Host and IBF Shadow (Fig 2.15).

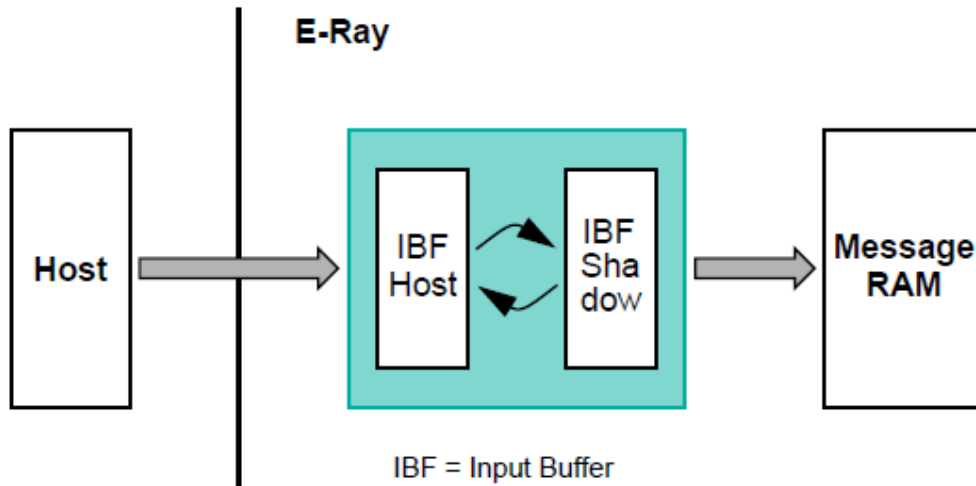


Fig 2.15 Transmit process via the Input Buffer [11] (Figure 10)

Once the IBF Host and IBF Shadow sections are swapped, the Message Handler starts the transfer of the header and/or⁷ data section to the targeted buffer in the message RAM [11]. While the transfer takes place, an Input Buffer Busy Shadow (IBSYS) flag is raised, however, in the meanwhile the Host may write the data for the next message buffer to the IBF. After the transfer is finished, the IBSYS flag is reset and the Host may request the next transfer by writing the target buffer number in the IBCR register. If the Host writes to the IBCR register while the IBSYS flag is still on, an Input Buffer Busy Host (IBSYH) flag is raised and the last transfer request is pending until the current transfer is finished. A further attempt for the Host to write to IBCR register, while both flags are raised, has no effect but an error flag is raised.

The procedure to configure/update the n-th message buffer is as follows: ([11] p. 127, 136)

1. Wait until IBSYH flag is reset
2. Write data section to WRDSn
3. Write header section to WRHS1...3
4. Write command mask configuration to IBCM
5. Write target message buffer to IBCR to demand message transfer
6. Check whether the message buffer has been transmitted by checking the respective TXR bit (TXR = '0') in the TRXQ1/2/3/4 registers (single-shot mode only).

If the designated transmit message buffer is configured to operate in single-shot mode, after the transmission is completed, the CC clears the respective transmission request (TXR) flag and the Host may update the buffer. In continuous mode, this flag is not cleared by the CC and message is transmitted every time it matches the filter criteria. The TXR flag is reset by the Host when executing step 5 from the update procedure described above.

⁷ Depending on the configuration of the IBCM the message Handler updates only header, only data or both sections of the target buffer in the message RAM [11]

Host Read Access via Output Buffer

The Host can request contents and status of a message buffer in the message RAM via writing the message buffer number in the Output Buffer Command Request (OBCR) register [11]. The desired contents (header, data or both) are specified via the Output Buffer Command Mask (OBCM) register. The Host triggers the transfer of the buffer contents from message RAM to the OBF Shadow by writing '1' to the REQ bit, also part of the OBCR register. Writing '1' to the VIEW bit of the OBCR register swaps the contents of OBF Host and OBF Shadow and the Host can read the message buffer data via the OBF header registers (RDHS1...3) and OBF data registers (RDDSn, n=1...64) (Fig 2.16). The message buffer status is accessible via the message buffer status (MBS) register.

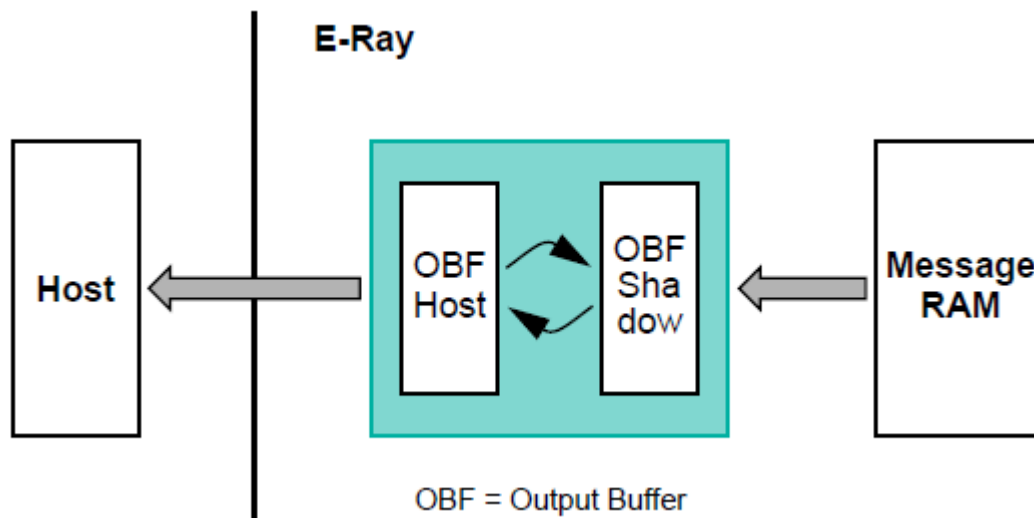


Fig 2.16 Receive process via the Output Buffer [11] (Figure 12)

During the transfer of message buffer contents from the message RAM to the OBF Shadow, the OBF Busy Shadow (OBSYS) flag is set. In the meantime, the Host may access the contents of the OBF Host or write the next message buffer number to OBCR register but the REQ and VIEW bits cannot be set until the OBSYS flag is reset. If REQ and VIEW bits are set with the same write access, while OBSYS is '0', the OBF Host and OBF Shadow are swapped first and then the new transfer of message buffer data from message RAM to OBF Shadow is triggered.

The procedure of requesting one or more message buffers' content is as follows: ([11] p. 138-139)

1. Wait until OBSYS is reset
2. Write command mask configuration to OBCM
3. Write target message buffer number to OBCR and '1' to OBCR.REQ bit
4. Wait until OBSYS is reset
5. If no further message buffer is requested – proceed to 9
Otherwise write command mask configuration to OBCM for the next message buffer
6. Toggle OBF Host and OBF Shadow and start transfer of next message buffer contents to OBF Shadow simultaneously by writing the buffer number to OBCR and '1' to OBCR.REQ and OBCR.VIEW bits
7. Read out the previous message buffer via RDHS1...3, RDDSn and MBS registers
8. Wait until OBSYS is reset
9. Demand access to last message buffer contents by writing '1' to OBCR.VIEW bit
10. Read out last transferred message buffer via RDHS1...3, RDDSn and MBS registers

2.4 Communication Protocols

In the following subchapters are described the hardware and software aspects of the communication protocols used for data exchange between the PC client and the FlexRay server node, according to the 7-layered OSI model. For this project this is the Ethernet protocol and the network protocols that run over it – the TCP/IP protocol suite and its lightweight version for embedded systems the lwIP.

2.4.1 Ethernet Protocol

Ethernet is a network interface that provides high speed connectivity, widely used in local (LAN) or metropolitan (MAN) area networks [14]. It is standardised by the Institute of Electro and Electronics Engineers (IEEE) and known as the IEEE 802.3 standard. Commonly used communication mediums are twisted pair and fiber optic links (together with repeaters, hubs, switches, etc.), providing data rates from 10 Mb/s to 100 Gb/s.

According to the 7-layer OSI model, Ethernet relates to the bottom two layers – Physical and Data Link [15]. According to the IEEE 802.3 standard [16] these two OSI layers are additionally divided into sub-layers to present a finer structure (Fig 2.17). At physical layer, the IEEE sub-layers depend on whether 10, 100 or 1000 Mb/s Ethernet standard is used. The MAC and LLC layers are above the layers that define the physical and media specifications and do not depend on them. The LLC is not bounded to the Ethernet standard but is intended to serve all LAN systems. Therefore, the LLC layer is not formally part of the IEEE 802.3 system specifications.

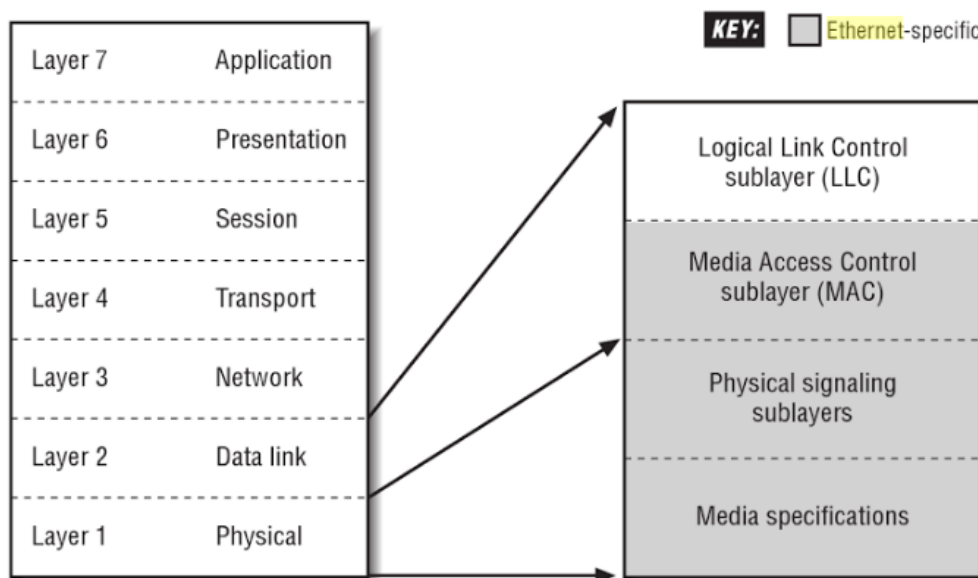


Fig 2.17 The major Ethernet layers defined by IEEE ([15] p.13)

The physical Ethernet specifications are separated into two basic groups of hardware components – signalling and media components [15]. The signalling components represent an Ethernet hardware interface that is used to send and receive signals over the physical medium. The media hardware components are used to build the physical medium where signals are transferred – twisted-pair or fiber optic cables, transceivers, repeaters, etc.

Ethernet can operate in half-duplex and full-duplex mode. In half-duplex mode the media access is based on the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) media access control (MAC) protocol. This protocol represents a set of instructions that arbitrate the access to the shared channel. In full-duplex mode the CSMA/CD is not necessary and therefore it is switched off.

Data is transmitted over the Ethernet in frames ([17] p. 155-169). The **frame** is a standardized sequence of bits that besides the actual payload data, it carries additional information – source and destination address, payload length, checksum, etc. On Fig 2.18 is shown the frame composition of DIX⁸ and IEEE 802.3 frame standards. Although very similar, these standards have some minor differences that make them incompatible with one another, unless hardware that supports both frame formats is used. Nowadays, IEEE 802.3 is the established standard for the majority of vendors.

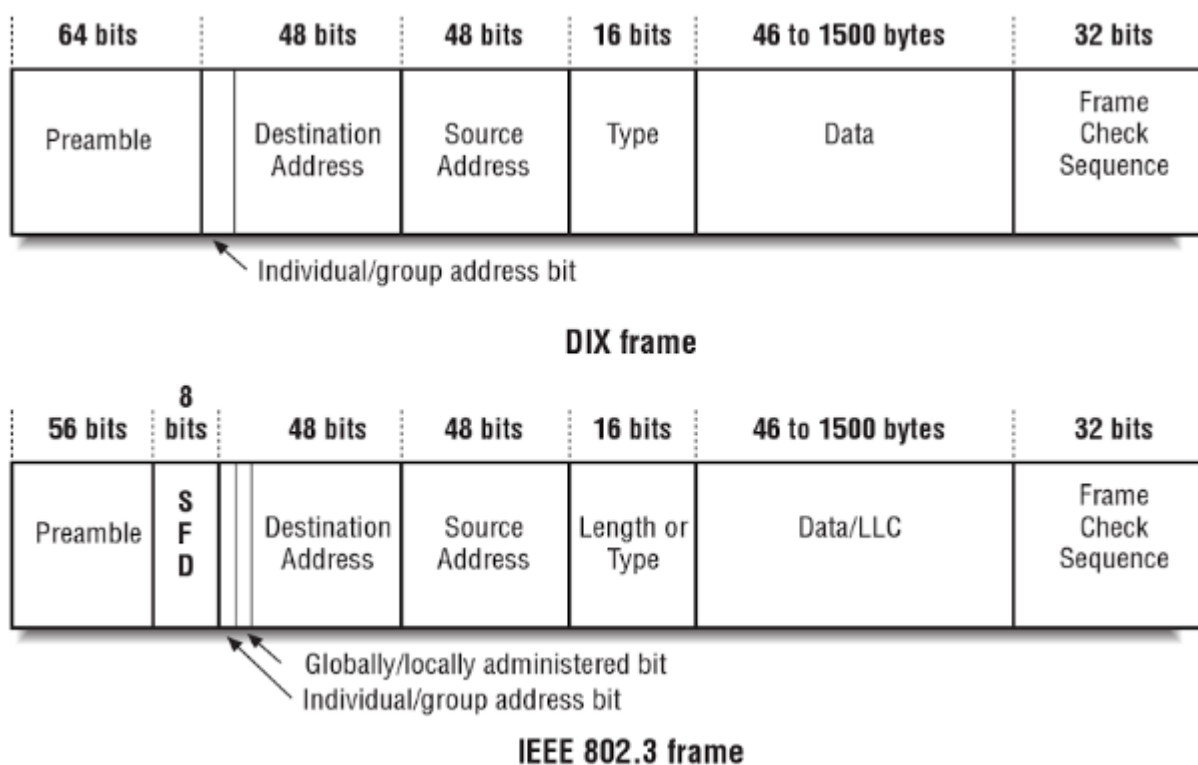


Fig 2.18 DIX frame vs. IEEE 802.3 frame ([15] p. 41)

The similarity in both Ethernet frame specifications allows their structure to be investigated in parallel, highlighting their differences:

- The 64 most significant bits (MSB) are used for channel awakening and clock synchronization and so to prevent some potential data loss⁹. In the IEEE 802.3 variant, the last 8 bits are called Start of Frame Delimiter (SFD) and are used to signal the start of the frame transmission.
- The 48 bit field that follows holds the destination address. In both frame formats, the MSB of this field reveals whether the frame is designated to one (*individual, physical, MAC,*

⁸ DIX is abbreviation of DEC-Intel-Xerox consortium that first standardises the Ethernet frame. [15](p.5-7)

⁹ The Preamble bits play a role in 10 Mb Ethernet only. In Fast and Gigabit Ethernet the Preamble bits have constant values because due to other technologies used for coding the signal, they are not really needed. Nevertheless, being part of the Ethernet frame standard, they are still being sent. [15](p.41-42)

unicast address) or multiple (*group, multicast* address) receivers. The IEEE notation gives significance and to the second MSB of that field to distinguish between *locally* and *globally* administered addresses. DIX addresses are always globally administered.

- The next bit field also consists of 48 bits where the Ethernet MAC (EMAC) address of the transmitting source is coded. This address has no significance for the EMAC protocol but it is provided for the higher-layer protocols.
- The payload length and/or the type of the high-level network protocol that is used (e.g. TCP/IP) are coded in the next 16 bits. In DIX (and initially in IEEE) standard, this field is used only as high-level protocol identifier. In IEEE 802.3 version, however, this field was assigned a dual interpretation, depending on the value that it holds. The logic is as follows – if the coded decimal value is less or equal to 1500 (which is the maximum transfer unit (MTU) of data in bytes), then this value represents the length of the actual message, contained in the following Data field; otherwise, if that decimal value is greater than the MTU, it is interpreted as the specified in DIX standard high-level protocol type identifier.
- The actual payload is stored in the Data field and must be between 46 to 1500 bytes. If the minimum amount of payload data is less than 46 bytes, padding is used to increase the size. In the IEEE 802.3 standard, a Logical Link Control (LLC) protocol may also reside in the Data field to provide control information or be used as a high-level protocol identifier in case the previous field is used as a Length field.
- The least significant 32 bits of the frame holds the cyclic redundancy checksum (CRC) which tests the data for errors that might have occurred during transmission.

The high-level protocol information that is embedded in the Ethernet frame is what actually establishes the successful communication between connected nodes. In order to achieve higher speeds, the purpose of Ethernet is limited to simply forwarding the frames to their destination, based on the “best effort” design. That means that if retransmission request time-out occurs, the frame is discarded. Ethernet frames are standardized but the information that they carry can differ, i.e. Ethernet does not depend on the higher-level protocols. Therefore, Ethernet does not provide any guarantee for successful data delivery. The correct delivery acknowledgement and order are guaranteed by protocols of higher levels. One of these protocols is the TCP/IP protocol suite that is the topic of the next chapter.

2.4.2 TCP/IP Protocol Suite

Transmission Control Protocol (TCP) and Internet Protocol (IP) are networking protocols which together form one of the most common networking protocol suites that is used in the end-to-end communication, known as the TCP/IP [9], [18]. It is standardized in a multi-layer stack, similarly to the OSI 7 layers model (2.2 OSI Model). TCP/IP precedes the OSI model with around a decade but their structures are identical. That is why it is common to explain the one with the other. The two models differ in the number of their layers – 7 in OSI model and 4 in TCP/IP model. Their functional definition and hierarchical sequence, however, are very similar (Fig 2.19).

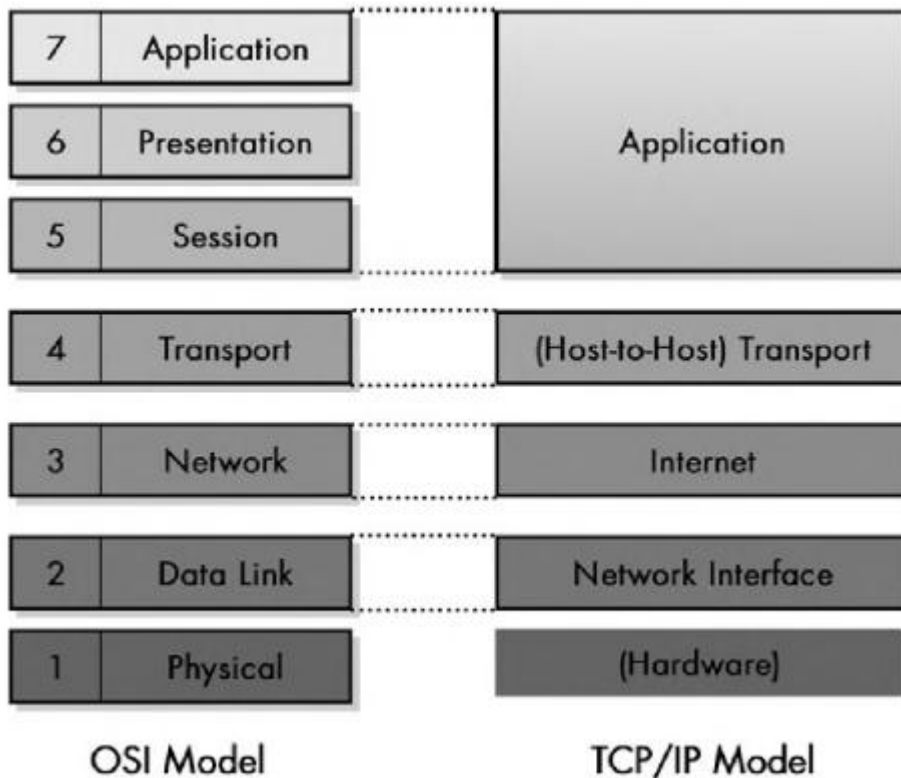


Fig 2.19 OSI Model vs. TCP/IP Model ([9] p. 129)

- **Network Interface** is the bottom TCP/IP model layer. According to some sources it merges the first two OSI layers [18](p. 7-8), while to others, the Physical OSI layer is excluded from the TCP/IP model [9](p.128-129).
- **Internet** layer refers to the network operations like: addressing, routing, data packaging, etc. The IP protocol resides in this layer.
- **Transport** layer corresponds to the OSI Transport layer and part of the Session layer. The key protocols for this layer are the TCP and UDP (User Datagram Protocol).
- **Application** layer is the top layer in TCP/IP model. It combines layers 5, 6 and 7 from the OSI model, due to their similar nature.

IP is a connectionless protocol that simply provides addressing and routing methods to deliver transmitted messages to their destination [9], [18]. It does not provide reliability, flow control, or error checks. The basic unit of data in an IP network is called *datagram*. There are two IP versions – IPv4 and IPv6.

TCP is connection oriented, bi-directional protocol, that guarantees packet deliveries in the correct order [18], [19]. It provides error checking and recovery mechanisms and requests retransmission of erroneous or lost data packets. TCP handles congestion and flow control and provides handshaking sequences¹⁰ for establishing a connection. It is used in applications that require high transmission reliability. Some of the most common application protocols that use TCP are HTTP, FTP, Telnet, SMTP, etc.

UDP is simpler, connectionless, one-directional protocol, where data packets are broadcasted by the transmitting source but there is no guarantee whether these packets are successfully received [18], [19]. Unlike TCP, UDP keeps no track on the data packets and in case of errors, erroneous

¹⁰ TCP requires three packets to set up a socket connection, before any user data can be sent. [19]

data packets are simply discarded. It is more lightweight and faster than TCP¹¹. UDP is used in cases where transmission and processing speed is of higher importance for the application than guaranteeing correctness of each data packet, like video and audio streaming, games and so on. Common protocols that use UDP are DNS, DSCP, TFTP and so on.

DHCP (Dynamic Host Configuration Protocol) is a protocol based on the UDP protocol¹² that is used to provide configuration data to an IP host [18], [20]. It assigns the host with a unique IP address and provides other network information depending on the allocation mechanism that is used. It supports three mechanisms of IP allocation – automatic, dynamic and manual. Dynamic allocation is the only one that supports automatic reuse of an IP address that is no more in use by the host.

The key concept of the TCP/IP protocol suite is in the Client-Server communication notation ([9] p.126). Clients and servers can be synonyms in the context of a TCP/IP connection and this can sometimes be confusing. In order to distinguish their roles for this project, the client is the one that initiates the connection and sends the provoked by the user requests to the server. The server responds to these client requests by providing the requested service and data.

2.4.3 LwIP Protocol/Raw API

Lightweight Internet Protocol (LwIP) is a lightweight version of the TCP/IP protocol suite, designed by Adam Dunkels at the Swedish Institute of Computer Science [21]. It is intended to provide a full scale TCP with less resource requirements. It requires only a few tens of kilobytes of free RAM and around 40 kB of ROM code to run which makes it the best solution for embedded applications with limited resources.

LwIP provides three Application Program's Interfaces (API) to enable program communication with the TCP/IP code:

- Socket API
- Netconn API
- Raw (native) API

The first two are considered high-level APIs, while the Raw (also called Native) API is a low-level API as it does not require an operating system to run [22]. The Raw API is designed as a set of callback functions that are triggered on certain events, like: new data available, data ready to be sent, data transmitted, errors, connection loss and so on. These callback functions have to be properly registered at start-up (via *tcpip_init_callback()*) or at run time (via *tcpip_callback()*). They can only be called from the main thread (*tcpip_thread*) and are not protected from concurrent accesses in multithreading environment [23].

The LwIP raw API provides access to protocols of all TCP/IP layers but the one of interest for this project is the TCP. The TCP network design is based on the raw API event-driven callback mechanism. Before making a call to any of the TCP functions, the lwip has to be initialized (via *lwip_init()*). A TCP connection is identified by a Protocol Control Block (PCB).

¹¹ TCP header size is 20 bytes; UDP header size is 8 bytes. [19]

¹² More particularly DHCP is based on the Bootstrap Protocol (BOOTP) that is based on the UDP protocol. [20]

To setup an active PCB connection the following steps are required:

1. Call *tcp_new()* to create a PCB.
2. Optionally call *tcp_arg()* to associate an application-specific value with the PCB.
3. Optionally call *tcp_bind()* to specify the local IP address and port.
4. Call *tcp_connect()*.

To send data over a TCP connection:

1. Call *tcp_sent()* to specify a callback function for acknowledgements.
2. Call *tcp_sndbuf()* to find the maximum amount of data that can be sent.
3. Call *tcp_write()* to enqueue the data.
4. Call *tcp_output()* to force the data to be sent.

Receiving data over the TCP connection is callback based:

1. Call *tcp_recv()* to set the callback function that will process the received data.
2. Call *tcp_recved()* when data has been processed and the application is ready to receive more.

The maximum size of the receiving window is configured via parameter `TCP_WND` in *lwipopts.h* header file.

LwIP is based on polling [23]. When connection is idle, LwIP polls it by calling a callback function every predefined period of time. The polling time interval and the callback function are set via function *tcp_poll()*. The interval is specified in number of TCP coarse grained timer shots, which typically occurs twice a second.

Connection is closed by calling function *tcp_close()* which will also deallocate the PCB. If there is insufficient memory for performing closing process, call to *tcp_abort()* never fails. It aborts the connection by sending a reset segment to the remote host and deallocates the PCB.

2.5 Hardware Platform MicroZed Board

MicroZed is a low-cost evaluation board from the Xilinx Zed board family [24], [25]. It provides features that allow it to be used as a stand-alone evaluation board or extended as an embeddable system-on-module (SOM). It is equipped with Zynq XC7Z010-1CLG400C or Zynq XC7Z020-1CLG400C AP SoC, both part of the Xilinx Zynq®-7000 All Programmable SoC family (Fig 2.20). These products are supplied with ARM Cortex-A9 CPU, which provides an on-chip and external memory interfaces and a variety of peripheral connectivity interfaces.

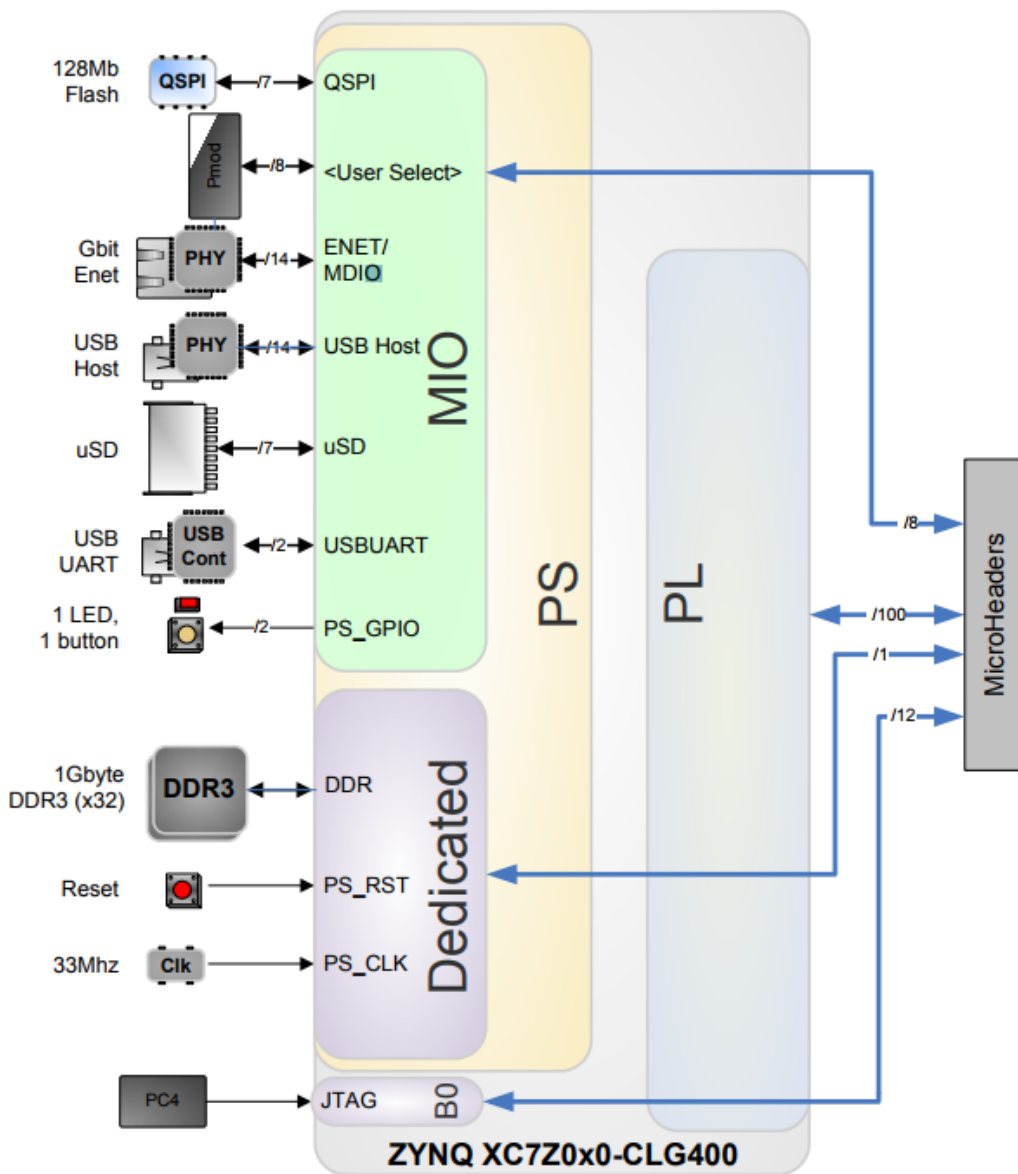


Fig 2.20 MicroZed block diagram ([25] Figure 1)

The Xilinx MicroZed board, used for this project as an example of a FlexRay node, is equipped with Zynq XC7Z020-1CLG400 AP SoC. The most essential for the project features that are provided are:

- 1GB DDR3 RAM
- JTAG connectivity
- 10/100/1000 Ethernet PHY

The Dynamic Memory Interface provides a 1 GB of address space that uses a single rank configuration of 8-bit, 16-bit or 32-bit DRAM memories. It includes a dynamic memory controller and static memory interface modules. The DDR memory controller is multi-ported which allows the processing system (PS) and the programmable logic (PL) to have shared access to a common memory.

The Zynq-7000 AP SoC devices support three different boot modes – JTAG, QSPI and SD card. The desired mode is configurable via the boot mode jumpers (Fig 2.21). When the boot mode is configured on JTAG, an external JTAG cable is needed. MicroZed is designed with a Platform Cable JTAG connector – 2x7, 2mm, shrouded, polarized header. It is compatible with Xilinx Platform Cables and Digilent JTAG HS1 or HS2 Programming Cables.

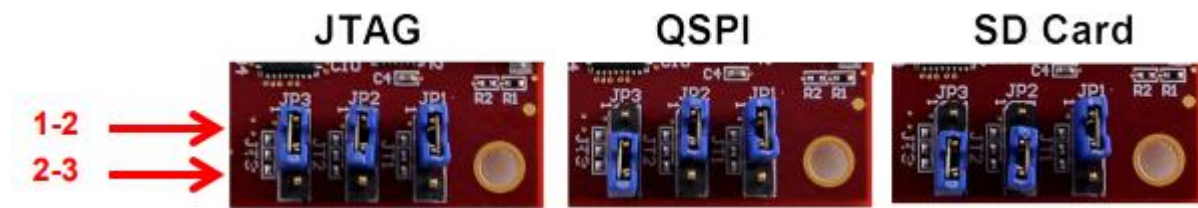


Fig 2.21 Boot Mode Jumper Settings with Cascaded JTAG Chain ([25] Figure 8)

The MicroZed is equipped with Marvell 88E1512 PHY for 10/100/1000 Ethernet network connection that operates at 1.8V. On Fig 2.22 is shown a high-level block diagram of the Ethernet module. The RJ45 connector is shared with the USB-Host interface and 2 LEDs for traffic and valid link state indication are provided.

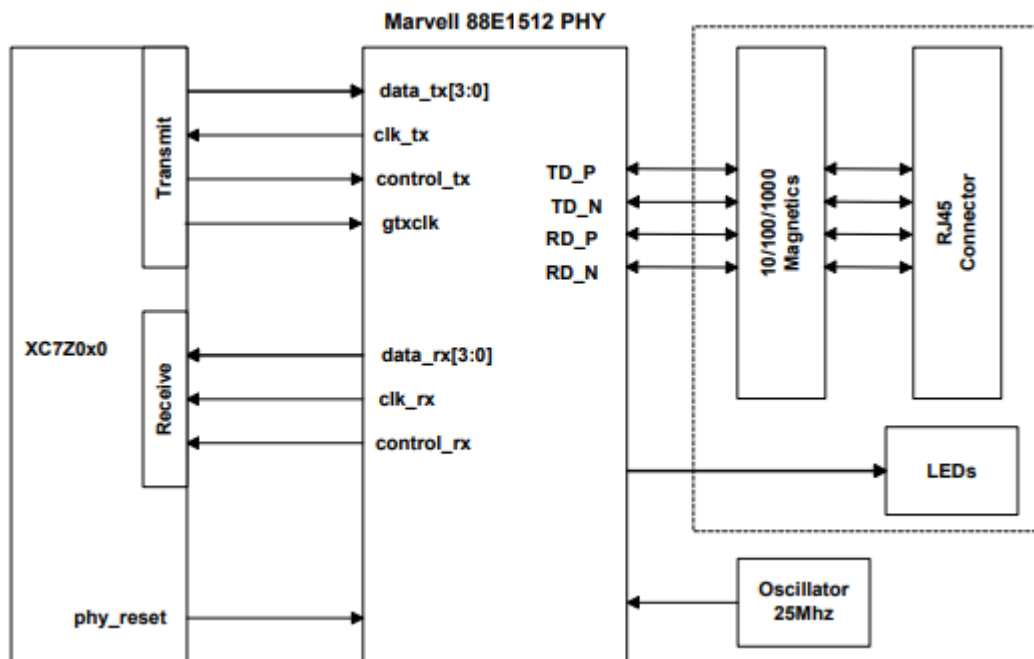


Fig 2.22 10/100/1000 Ethernet interface ([25] Figure 5)

3. Requirements Analysis

The following requirements analysis is separated in four main topics. The first one refers to the requirements, set with the project assignment, based on previous researches in this field. The rest are related to the problems that arise within the different use cases, regarding the structure and contents of the data objects and the software architecture, as well as testing and evaluation of the software features. The conceptual decisions and solutions of these problems are discussed in Chapter 4.

3.1 General Requirements

This project assignment is an extension of a similar task that was accomplished by previous thesis in the context of Urban Mobility X-by-Wire(less) project [26]. That allows some of the base requirements that were analyzed in the previous project to be taken as granted:

- The service interface for configuration, control and message data transfer between the PC and the nodes in a FlexRay cluster shall be implemented via Ethernet standard (2.4.1 Ethernet Protocol), Cat6 Ethernet cables and an unmanaged Ethernet switch
- For ISO/OSI network and transport layers (2.2 OSI Model) the TCP/IP protocol (2.4.2 TCP/IP Protocol Suite) shall be used
- The TCP/IP stack shall be implemented by its lightweight version for embedded systems (LwIP) (2.4.3 LwIP Protocol/Raw API)
- Ethernet communication shall support 1Gb/s data rate
- Configuration, Control and Message data objects shall be structured based on the JavaScript Object Notation (JSON) format [27]
- Programming language for the PC client application shall be Java; for the microcontroller server application shall be C

For a more detailed discussion of previous requirements refer to [26].

As another predefined requirement for this project, the Xilinx MicroZed board shall be used as a prototype FlexRay node, in contrast to the Texas Instruments Hercules TMS570 board, used for the previous project assignment [26]. MicroZed has a Gigabit Ethernet interface installed that allows interfacing to a FlexRay controller (not integrated) and gives the possibility of integrating new communication modules in its on-board FPGA fabric. In addition, there are known integrations of the lwip stack, which allow receiving and transmitting data from and to the microcontroller in the defined by the project requirements way.

After the task requirements that have been set with the project assignment are revealed, the next step is to analyze those aspects of the current project implementation that open a field for discussion. The first aspect that should be considered is the internal JSON structure of the data objects, their purposes and contents. Having this done, leads to the next aspect – how is that JSON data going to be processed and sent over the TCP/IP network. That, from another hand, raises the

question of how are the configuration, control and monitoring processes going to be implemented. As we can see, each answer leads to a new potential question. Once solutions for all these questions are found, summing them up should give the premises for foundation of a user interface that best fulfils the established project requirements.

3.2 Data Objects

Data objects are text objects that contain information needed for applying a correct operation of the configuration, monitoring and control processes of a FlexRay cluster. There are two main groups of data objects. The first one – **configuration data objects** – refers to data related to the FlexRay protocol constraints and the device specific hardware architecture. This data is needed for the correct configuration of the controller node and shall be provided when required by the application. Therefore, it shall be stored in a text file in the system. Such file shall exist for each distinct node and shall be maintained by a programmer. The second group of data objects is **communication data objects**. This group refers to text objects used for data transfer between the PC client and the participants in the FlexRay cluster. Communication data objects shall be generated by the application during runtime and their content shall depend on the current request. It shall contain as less overhead as possible, in order to maintain higher communication speeds.

The structure of each group of data objects shall be based on the JSON format notation [27]. The reasons for choosing file format JSON in the previous [26] and current project is that it contains a small amount of overhead, is easy to read by both humans and machines, maintains a hierarchical structure and has a wide support of libraries. Those features of the JSON format shall be used when defining a suitable internal JSON structure of each of the data objects. A good internal JSON structure shall be expressed by a well organized JSON hierarchy, established with respect to maintainability by humans, low memory and processing requirements, data reuse and adaptability. Relation of data to the different FlexRay protocol and hardware aspects, as well as the different use cases involved, shall also be taken into account.

The heterogeneous character of the FlexRay configuration data enables to distinguish separate data categories, according to their relation to different FlexRay protocol (e.g. constants, global (cluster) parameters, local (node) parameters, variables, etc) and hardware aspects (e.g. register-address map). This property shall be used when estimating the most appropriate internal JSON structure of the configuration file. In order to ease the process of reading, editing and parsing that information, separating the configuration data into multiple files should also be considered.

Configuration is a process where a device is brought into a default or start-up state and is typically executed after a reset or at power up [28]. It is, in general, based on writing predefined data to specific memory address registers of the configured device. This configuration process follows a defined flow of actions and sometimes requires time delays between consecutive register write accesses. A mean of providing this information to the node's Controller Host Interface (2.3.1 Communication Controller) shall be determined.

Hexadecimal representation of values shall be supported by the configuration files. Optionally, support for other number format representations should also be implemented. Although the current project is intended to relate only to systems running the FlexRay protocol, the configuration file structure shall be specified to enable the integration of other protocols in the future with minimum amount of effort.

3.3 Software Features and Structure

The software implementation is divided into two parts. It consists of one client application, programmed in Java, and one server application, programmed in C (according to the general requirements). The workload of both applications shall be shared in a way that the resource power of the PC client is maintained. Therefore, the main decision taking and logic algorithms shall be executed on the PC. The client application shall compose and transmit data objects, containing only relevant for the requested access data in the desired request order. The purpose of the server application should be limited to parsing the received data objects, processing the read/write access to registers¹³ in the defined by the client order and composing and sending response data objects back to the client. Eventually, these two applications shall work with each other in a synchronized manner under the control of the user.

The client application shall be based on a graphical user interface (GUI) that shall be intuitive in use to enable less experienced users to work with it. That implies that a compromise between usability and complexity should be agreed. The GUI shall be designed to accept user requests and display usable information, where all the program logic is executed in the background. Its purpose shall include some basic debugging functionalities to inform the user for error conditions by error and warning messages.

The application shall enable the user to browse through the system memory for a configuration file. A parsing algorithm shall be implemented that parses the JSON data from the specified file in the application memory. As JSON format has popularity and support among programmers, the use of standardized, well-approved and widely used libraries shall be considered. Suitable parsing algorithms shall be developed for both server and client applications.

The user shall be able to connect to a desired node by addressing its assigned IP address and port number. For this project, simultaneous connection to FlexRay nodes shall be limited to one node at a time. Once a TCP/IP connection between the PC client and the FlexRay server node is established, the user shall be able to perform the three fundamental processes – initial configuration, control and monitoring of the message buffers of a FlexRay node.

The limited memory resource on the FlexRay node (e.g. buffer of TCP/IP stack and FlexRay input/output buffers) as wells as the limited processing speed for parsing and access to the (mock-up) FlexRay module shall be taken into account when designing the application algorithms. An appropriate TCP/IP stack, specially designed for resource limited embedded systems, is already set

¹³ Due to the lack of a FlexRay module for the MicroZed board, the register space is mimicked. See Section 3.4

by the general project requirements. However, it shall be investigated whether it is supported by the MicroZed board, which is the example FlexRay node for this project. The application design shall be aimed to ensure no buffer overflows and correct execution of the requested processes and shall give status feedbacks for success, failure, warnings and so on.

Requesting node status and message buffers content is the essence of the monitoring process. That involves heavy data traffic between the server and the client. Therefore, the implementation of the software algorithms shall be targeted towards the speed of data processing. Once requested, the monitoring process shall be maintained until further user request or connection loss. The requested FlexRay message buffer content shall be displayed in a GUI text field(s). As this information is of high interest for the user, the software should offer the opportunity to save this data on a text log file. Optionally, other status related information should be displayed for user reference.

As a part of the control process, the user shall be able to send CHI commands (2.3.1 Communication Controller) and retrieve current node states. Additionally, requesting a read or write access to a selected node register shall also be supported. For this purpose, all node registers shall be presented in a list.

As the current project is intended to be continued in the future, good programming practices shall be followed. The code shall be structured in functional blocks, so that it should be possible to update each block independently on the others. In order to ease the process of reading, method and variable names should be self-describing and accompanied by comments.

3.4 Evaluation

For proper evaluation of the software features, at least two microcontrollers (representing FlexRay server nodes) having Ethernet and FlexRay modules installed shall be available. For each distinct microcontroller the FlexRay register-address macro shall be present. The PC client shall have Telnet enabled and Java version 1.7 or 1.8 installed. For installing and running the server code, the Vivado 2014.4 IDE, including Software Development Kit (SDK) tool, shall be installed and the board definition files for the MicroZed board shall be included¹⁴. An Ethernet switch and Cat6 Ethernet cables shall be used for the connection between the PC client and the FlexRay server nodes.

For this project there is only one MicroZed board available and no FlexRay module nor FlexRay register-address header file are present. Therefore, the evaluation of the project shall be based on investigation of the correct communication between the PC client and the server node. The server register space shall be mimicked by a global array of 32-bit integer values and the client R/W access request shall be performed on that array.

¹⁴ The board definition files for the MicroZed board for Vivado IDE can be downloaded from [35]. The installation procedure is described in *Install Avnet Board Definition Files in Vivado 2014.2 v1.3.pdf* file included in the download package.

In most cases the evaluation of the correct operation of the developed software application shall be performed manually by the user. Having the available information from the configuration JSON data objects (which design is still to be established), the user shall predict the outcome and compare it to the result of the application algorithm for a given use case. That includes the following use cases:

- Parsing and composing of JSON data objects by both client and server applications
- Composition and decomposition of register values based on the available information – FlexRay parameter/status values, bit ranges and offsets
- Encoding/decoding of transmit/receive FlexRay messages
- R/W access to the global 'register space' array

The evaluation of the TCP/IP communication shall be investigated with the help of suitable software (e.g. WireShark). That includes investigation of the sizes of the microcontroller's receive message buffer and the transmitted JSON objects. For that purpose the implementation of a small test local server application shall be considered to mimic a second FlexRay node and provide additional testing opportunities.

4. Concept

This chapter describes the options that were taken into account in attempt to find the solutions that best correspond to the project requirements established in Chapter 3 Requirements Analysis. That includes a detailed discussion over the considered approaches, their advantages and disadvantages.

4.1 Configuration Data Objects

According to the general requirements the format of the configuration data objects is JSON. A JSON structure consists of objects which bodies are defined by braces [27]. The first open brace and its corresponding closing brace determine the *root* JSON object. The root object represents the *parent object* of all nested JSON objects, referred as *child objects*. The internal JSON object structure is based on key-value pairs, referred as *tokens*, which are separated by comma. The key must be a unique in the scope of its parent object string, while the value can be one of the following formats: string, number, boolean, null, array or another object.

The determined requirements regarding the internal JSON structure of the configuration data object aim to ease the process of maintaining that object by humans, while in the same time a significant amount of heterogeneous data to be contained there. Hence, a compromise between complexity of the object's internal JSON structure and ease of processing it needs to be made.

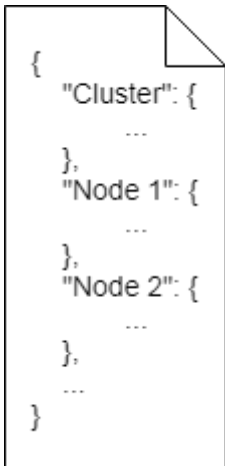
4.1.1 Files Content

The first subject of discussion is whether all the data to be present in one file or to be spread across multiple files. FlexRay parameters are divided into two groups: **cluster specific** and **node specific**¹⁵. Cluster specific are those parameters that have the same value in all nodes belonging to that cluster. They are also referred as global parameters. An example for global parameters are the number of macroticks in a communication cycle, duration of a static slot, duration of a minislot, header CRC, etc. (2.3 FlexRay Protocol). Node specific parameters are local for the node and can have different values in each node (e.g. number of samples per microtick, number of microticks per macrotick, connected channel, etc.). Both groups are part of the FlexRay protocol specification. Register names and their corresponding addresses are **device specific**, i.e. they depend on the device architecture but not on the FlexRay protocol. Separating the configuration data into three distinguishable groups grants the opportunity to implement a more appropriate structure of the JSON configuration file.

First, let us consider having all cluster configuration information in one root JSON object (Example 5-1). The global cluster data, as well as the local and device specific data for each node, are enclosed in a separate child JSON object. The advantage is that there is only one file in the system for each cluster that is shared between the nodes, belonging to that cluster, and the amount of redundant information is small. However, the overall size of data in this file is big and requires a

¹⁵ For more information about the different types of FlexRay parameters and variables refer to Table A-1 in Appendix A.

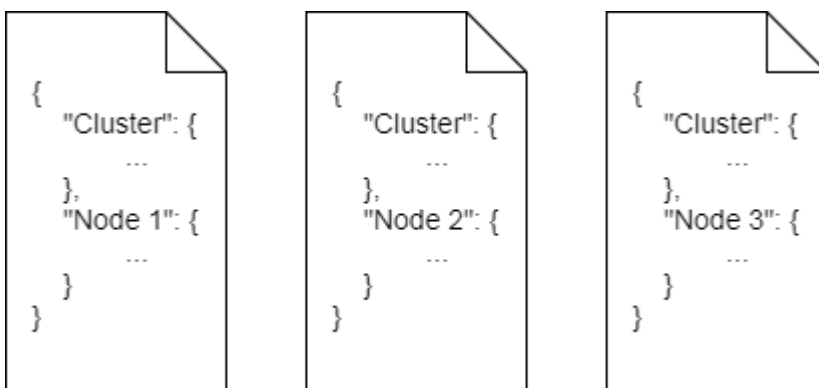
complex internal structure of each node JSON object. This increases the complexity of parsing algorithms and decreases the readability and possibility of data reuse. On the other hand, having all data in a single file is disadvantageous in terms of security – damaging one file leads to loss of big amount of information.



```
{
  "Cluster": {
    ...
  },
  "Node 1": {
    ...
  },
  "Node 2": {
    ...
  },
  ...
}
```

Example 5-1. Example JSON structure of configuration file

Another consideration is to have a separate configuration file for each node (Example 5-2). The advantage is that the amount of data in each file is significantly reduced, which improves readability and the possibility of data reuse. The process of parsing the JSON data is also improved as only the needed file is loaded to the application memory. However, the bottlenecks of the parser algorithm design come from the internal structure of the node JSON objects. Due to the heterogeneous type of data contained there (protocol and device specific), its internal JSON structure remains complex which does not bring much of improvement in the design of the parsing algorithm, compared with the previous case. On the other hand, redundant information is present as the JSON cluster object is the same in each configuration file related to that cluster. From that follows that changing a single cluster parameter value requires update of every configuration file referring to a node of that cluster.



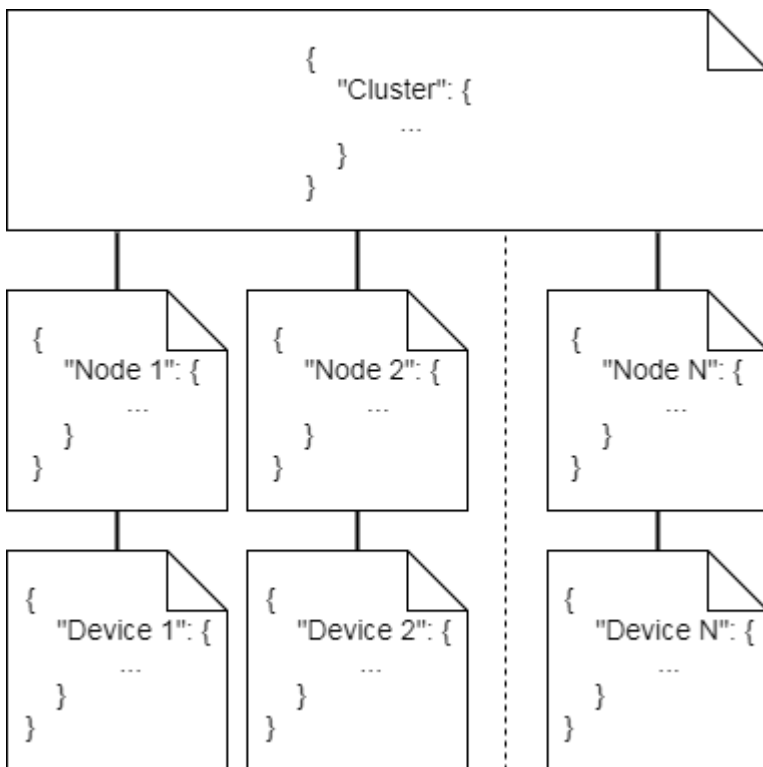
```
{
  "Cluster": {
    ...
  },
  "Node 1": {
    ...
  }
}

{
  "Cluster": {
    ...
  },
  "Node 2": {
    ...
  }
}

{
  "Cluster": {
    ...
  },
  "Node 3": {
    ...
  }
}
```

Example 5-2. Example JSON structure of configuration files

A third consideration is to separate the configuration data for each node in three JSON files, based on the three parameter groups that were distinguished – cluster, node and device specific (Example 5-3). One file contains only cluster (global) parameters and is shared between all nodes belonging to the same cluster. For each node there is one file, containing local parameters and status variables and another file containing the device specific information. The disadvantage with this approach is that the overall number of files per cluster is increased. However, the separation of data regarding different FlexRay protocol and hardware aspects significantly decreases the complexity of the inner JSON structure and amount of data in each file. That leads to simplification of the parser algorithms and increases the ability of data reuse and file maintenance. There is minimum amount of redundant data and each file can be modified by the programmer independently on the others¹⁶ and be reused for another FlexRay application.



Example 5-3. Example JSON structure of configuration files

¹⁶ In fact there are certain dependencies, for example when a new node is added to a cluster its ID has to be added in the Cluster file or in cases where a certain node parameter value is directly related to a cluster parameter value or vice versa, e.g. $gdBit = cSamplesPerBit * gdSampleClockPeriod$ [µs] (for details see [10] (Appendix A and B))

In Table 5-1 are summarized the advantages and disadvantages of each of the discussed cases:

Case	Number of configuration JSON files	Advantages	Disadvantages
1	1 file per cluster	<ul style="list-style-type: none"> • Only one file to work with • Less number of files in system • Small amount of redundant data 	<ul style="list-style-type: none"> • Big amount of data concentrated in one file • Complex file structure • Complex parser algorithm • Hard to read/modify by human • Low possibility of reuse
2	1 file per node	<u>Compared to Case 1:</u> <ul style="list-style-type: none"> • Less amount of data per file • Less data loaded in application memory • Faster execution of parsing process • Better readability • Increased possibility of data reuse 	<u>Compared to Case 1:</u> <ul style="list-style-type: none"> • More number of files in system • Internal JSON object structure complexity remains • Parser algorithm complexity remains • Data redundancy • Hard to update
3	3 files per node	<u>Compared to Case 1 and Case 2:</u> <ul style="list-style-type: none"> • Less amount of data per file • Better internal JSON structure • Simpler parser algorithm • Small amount of redundant data • Easier to read/update/modify • Reusability 	<u>Compared to Case 1 and Case 2:</u> <ul style="list-style-type: none"> • Increased number of files in the system

Table 5-1. Number of configuration JSON files - advantages and disadvantages

After summing up the advantages and disadvantages for each of the three cases, shown in Table 5-1, it is clear that Case 3 responds best to the project requirements established in Chapter 3.

4.1.2 Files Structure

The previous section was engaged with the conceptual decision of separating the data, required for the configuration, control and monitoring processes, across three files. The current section is related to the different approaches and decisions that were considered regarding the internal JSON structure of each of these files. That includes discussion on the structural regulations and constraints that has to be taken into account when creating the files.

The fact that JSON format supports nesting of JSON objects [27] gives more options when defining a suitable internal structure for each file, as a response to the set project requirements. A good internal JSON structure is expressed in a well defined hierarchical structure, depending on the level of relation of data to different protocol or hardware aspects. The three data categories specified – cluster, node and device specific (4.1.1 Files) – determine the content of each of the three files, referred in general as **Cluster**, **Node** and **Device** files.

As next, the structural definition of each of these files is discussed. That includes a top layer view of the constituent JSON objects and discussion on some structural constraints. For more detailed examples of the Cluster, Node and Device JSON files refer to Appendix B.

Cluster file

The global FlexRay parameters can be subdivided into two groups – *system constants* and *cluster configuration parameters* ([10] Appendix A and B). The system constants refer to those FlexRay protocol related parameters that must have the same value in every cluster of a FlexRay system. Names of FlexRay constants start with the ‘c’ prefix (for full list of prefixes refer to Table A-1 in Appendix A). Global cluster parameters are identified by the prefix ‘g’ and their values must be identical in the scope of the cluster. The feature of the FlexRay data to be subdivided into more specific groups allows us to introduce a hierarchical structure that better corresponds to the project requirements. It is implemented in separating the data into sub-JSON objects (child objects), which naming relates to the data they contain (Example 5-4).

```
{
  "general": {
    "id": 5,
    "protocol": "FR",
    "node_ids": [1,...,n],
  },
  "constants": {
    "name": value,
    ...
  },
  "parameters": {
    "name": value,
    ...
  }
}
```

Apart from FlexRay related data, some additional information, intended for the user and the operation of the parsing application, is embedded inside each file. It is referred as *general information* and the JSON object where it is placed is called “general” (Example 5-4). The structure of the “general” object¹⁷ in each of the three file types is identical but its content differs. An example for such type of data, contained in Cluster file, is the cluster ID¹⁸ (assigned by the user for differentiation), a list of node IDs part of that cluster (discussed later) and name of the communication protocol (based on the requirement for future integration of other than FlexRay protocols).

Example 5-4. Cluster file structure

An advantage of grouping the data is that at the end it is presented simply as chunks of key-value pairs, which significantly simplifies the parser algorithm. The key of each JSON object token represents a name, which can be defined according to the FlexRay protocol specification v2.1 notation [10] or by the programmer. When the token name is defined or modified by the programmer it has to be accordingly updated in the software application. The JSON format guarantees no order of tokens [27] so their order in the Cluster JSON file is not relevant.

One of the set requirements regarding the structure and contents of JSON data objects is the hexadecimal representation of constant/parameter/variable values. In information engineering the hexadecimal format is preferred as it contains information for the position and value of each

¹⁷ In the text the naming of JSON objects is based on the name of the ‘key’ that corresponds to the object value.

¹⁸ Each of the three file types is assigned an ID used for file differentiation. It is assigned by the programmer and is used by the application to link the three files corresponding to a node. The IDs shall be unique so that for every node there is only one possible combination of files.

bit. Therefore, hexadecimal format is supported by the application. However, as JSON notation does not support hexadecimal numbers, the value must be presented as a string, i.e. surrounded by quotes. For user's comfort decimal representation is also allowed by the parser algorithm. Therefore, the user is free to choose the number format representation of constant/parameter values.

Node file

The FlexRay protocol data that may differ in each node of a cluster is stored in the Node JSON file. Unlike the Cluster file, where the data is separated according to its relation to the FlexRay protocol, in the Node file data is structured, depending on the purpose and relation of data to different register sections (Example 5-5). The reason is the better relation to the Device file, where data is organized according to register grouping (see next section).

```
{
  "general": {
    "id": 3,
    "IP": "192.168.0.10",
    "device_id": 123
  },
  "control": {
    "cc_control": {},
    "mb_control": {}
  },
  "status": {
    "cc_status": {},
    "mb_status": {}
  },
  "message_buffers": {
    "first_mb_number": {},
    "second_mb_number": {},
    ...
  },
  "command": {
    "command 1": value,
    "command 2": value,
    ...
  },
  "lock": [first value, ..., nth value]
}
```

Example 5-5. Node file structure

In the "general" JSON object is placed the general information about the node. That includes an ID that is used for differentiation of the Node files. It is assigned by the programmer who decides whether it shall be unique in the scope of the current cluster or in the whole FlexRay system (if more than one cluster is present). Every node is assigned an IP address that is unique in the scope of the FlexRay network. The last token of the "general" object is the ID assigned to the corresponding Device file (discussed later).

The "control" JSON object holds the part of the FlexRay protocol data, which is related to the configuration of the control registers for the Communication Controller (CC) (2.3.1 Communication Controller) and message buffers in message RAM (2.3.3 E-Ray). In the same sense, the "status" object relates to the status variables for the CC and the message RAM buffers. Every message buffer is configured via the Input Buffer. The configuration data for each buffer is placed inside the "message_buffers" JSON object.

The *italics* notation of token keys means that in the file those keys are replaced with an appropriate naming¹⁹. In this case the "first_mb_number" is simply "0", the "second_mb_number" "1" and so on. As token keys must be strings, they are always surrounded by quotes. FlexRay commands are placed inside the "command" JSON object. The command names are standard for the FlexRay protocol but their corresponding values might differ among the variety of node types.

¹⁹ This is valid for all following examples from this section.

The last JSON token of the Node file does not have a JSON object as a corresponding value but an array containing the values required for unlocking the registers after the configuration process has finished ([11] p. 24). These values are placed in an array for the reason that their order must match a defined sequence of write accesses to the Lock register.

The structure of the “control” and “message_buffers” child objects, as well as the “command” object, is based on key-value tokens. The key is the name of a FlexRay parameter/command and its corresponding value can be presented as a hexadecimal string or decimal integer. Unlike them, the “status” child JSON objects are structured differently. The values of status variables are represented as JSON objects, containing the possible status conditions related to that variable (Listing 4.1). As the value determines the status, the key-value pair is based on “value”-“status” notation, where regardless whether the value is represented in hexadecimal or decimal format, it must be surrounded by quotes as it corresponds to the key of the token.

```
“mb_status”: {  
    “vSS!ValidFrameA”: {  
        “0x00”: “No valid frame received on channel A”,  
        “1”: “Valid frame received on channel A”  
    },  
    ...  
}
```

Listing 4.1. JSON structure of a status variable object

Device file

The purpose of the Device file is to provide a link between the FlexRay protocol data, contained in the Cluster and Node files, and the hardware dependant register space. In response to the requirement that most of the algorithm logic shall be done on the richer in resources PC client, the Device file has to provide all the information needed by the application for composing data objects that contain minimum amount of overhead and are in ‘ready-to-use’ by the FlexRay server application form (still to be determined). That includes:

- Register-address map
- Position of each FlexRay parameter/variable inside the register
- Sequence of register accesses

The node’s memory space is hardware specific and the link between the register names and their corresponding addresses has to be included in the Device file. However, this map already exists in one of the device header files. Therefore, this information is automatically extracted by the application. This increases the algorithm complexity but the only additional data that is included in the Device file is the name of that header file.

The position of each FlexRay parameter/variable inside its corresponding register is defined via specifying the allowed for that parameter/variable bit range and the offset of the base register address.

```

{
  "general": {
    "id": 123,
    "name": "MicroZed",
    "type": "ZYNC",
    "version": "XC7Z020",
    "header": "header_file.h"
  },
  "control": {
    "cc_control": {},
    "mb_control": {}
  },
  "status": {
    "cc_status": {},
    "mb_status": {}
  },
  "input_buffer": {
    "header": {},
    "command_mask": {},
    "command_request": {},
    "busy_control": {}
    "data": ["register1", ..., "registerN"]
  },
  "output_buffer": {
    "header": {},
    "command_mask": {},
    "command_request": {},
    "busy_control": {}
    "data": ["register1", ..., "registerN"]
  }
  "command": {},
  "state": {},
  "lock": {},
  "new_data": ["register1", ..., "registerN"],
  "mb_status_changed": ["register1", ..., "registerN"]
}

```

As JSON format represents an unordered set of tokens it does not guarantee any sequential order. Therefore, for defining a certain order of sequential register accesses two approaches are considered. The first approach is to add indexing inside the JSON objects. The disadvantage is that this adds an additional payload and complexity in the parser algorithm and the designing of the Device file as changing one index is followed by changes in the subsequent indexes, too. The second approach is to use a notation that guarantees sequential order. In JSON format the only element that guarantees order is the array. This adds only a minor amount of additional payload (two brackets) but the main advantage is that editing the array does not affect the rest of its elements. This array notation is required only for JSON objects related to more than one register.

Example 5-6. Device file structure

The registers in the Device file are grouped according to their relation to different FlexRay protocol aspects (Example 5-6). The only exceptions is (as by the Cluster and Node files) the “general” object, which holds the device specific information and the name of the header file that defines the macros for the register addresses of the FlexRay module. The purpose of the assigned by the programmer ID is for file differentiation and provides a link to the corresponding Node file (18). For a full view of an example Device file, refer to Appendix B.

Those JSON objects that are related to only one register have the notation shown in Listing 4.2. That is the case with: “command”, “state”, “lock” and all except “header” object from “input_buffer” and “output_buffer” objects (Example 5-6).

```

"object": {
  "Register": {}
}

```

Listing 4.2. JSON structure of an object related to one register

JSON objects, containing more than one register, shall follow an array signature that implements the sequential register access order. That array holds the JSON objects ordered in their desired order of access (Listing 4.4).

```
"object": [{  
    "Register 1": {}  
  },  
  {  
    "Register 2": {}  
  },  
  ...  
  {  
    "Register N": {}  
  }  
}]
```

Listing 4.4. Array structure of a JSON object related to multiple registers

This array notation is valid only for JSON objects²⁰ containing more than one register object (Example 5-6):

- “cc_control” and “mb_control” objects part of “control” object;
- “cc_status” and “mb_status” objects part of “status” object;
- “header” object of “input_buffer” and “output_buffer” objects.

The relation between the register and all parameters/variables that are part of that register, together with their bit range and offset, is hierarchically structured as shown in Listing 4.3. The register name is a parent object of its related parameters. The order of parameters inside that object has no relevance but in their corresponding array values the bit range comes first and then the offset. The only accepted number format for the ‘range’ and ‘offset’ is decimal.

```
"Register": {  
    "parameter 2": [range, offset],  
    "parameter 3": [range, offset],  
    "parameter 1": [range, offset],  
    ...  
}
```

Listing 4.3. JSON structure of a register object

The “data” array holds the data registers for the Input and Output Buffers (2.3.3 E-Ray). The “new_data” array refers to the registers dedicated to inform the Host (2.3.1 Communication Controller) that new data is available for processing. The “mb_status_changed” array that there is a change in the message buffer status.

²⁰ As the value, corresponding to the key is an array but not an object, it is not fully correct to call them objects. Nevertheless, for consistency they are still referred as JSON objects as the array notation is added as a response to a project requirement.

All FlexRay parameters and variables that are present in the Cluster and Node files should be matched to their corresponding registers in the Device file. Their naming must be the same, taking into account case sensitivity. There are some variables, however, that are present in the Device file but do not exist in the other two files. The reason is that these variables do not have any predefined status condition related to them but the value itself is of interest for the user – for example time duration, slot counter, cycle counter, etc. In Device file, these variable names are preceded by an asterisk (e.g. "*vRemainingColdstartAttempts"). This notation is used by the application algorithm to distinguish between the different type of variables, as well as for user reference when maintaining the file. In cases where the variable name has no relevance for the application, its name can be custom or just an asterisk used as a placeholder ("*").

4.2 Communication Data Objects

Communication data objects are used for transmitting data between the PC client and the participants in the FlexRay network. According to the general project requirements, their design shall be based on the JSON format notation (3.1 General Requirements). They are composed by both client and server applications during runtime and depending on the communication direction with respect to the request invoker (client) are differentiated **request** (transmit) **data objects** and **response** (receive) **data objects**. The request data objects can be further differentiated according to the requested by the client register access to **write request objects** and **read request objects**.

In the current chapter are discussed the different approaches considered, regarding the structure and contents of the R/W request and response data objects in attempt to maximally fulfil the set project requirements.

One of the established requirements, regarding the structure and contents of the communication data objects, aims to maintain higher communication speeds by providing the data with minimum amount of overhead. That implies that the JSON data objects shall contain only relevant for the desired register access data – where to read/write (register address) and what to write (register value) (Listing 4.5). As the read request objects do not have the 'what' part, the structural constraints are presented only according to the write request objects. Once a suitable structural standard, that best responds to the project requirements, is defined for the write request objects, a discussion on the design of the read request and response data objects will be made.

```
{
    "address 1": "value 1",
    "address 2": "value 2",
    ...
}
```

Listing 4.5 JSON write request object containing only address-value pairs

All the information needed from the client application to compose the shown in Listing 4.5 request JSON object is contained in the Cluster, Node and Device files (4.1.1 Configuration Data Objects). The register address is defined by the macros in the register header file, while the register value is

automatically calculated by the client application by superposing all related to that register parameter values on their defined positions inside the register.

The exposed approach regarding the design of the communication data objects (Listing 4.5), tempts to have a minimum amount of overhead. There is, however, an issue with this approach. According to one of the project requirements, any major scripting and decision taking procedures shall be avoided on the FlexRay server application. Therefore, the client shall pass the data to the server in a 'ready-to-use' form (which in this case it does) and in the desired sequential access order. And here comes the problem – this JSON structure does not guarantee that the tokens will be extracted in the same order by the receiver's parser.

In order to design a JSON structure that guarantees order, a similar to the construction of some of the Device file JSON objects approach is taken. An array structure is embedded inside the root JSON object (Listing 4.6). Each of the tokens is surrounded by braces and represents an independent JSON object (as arrays cannot hold JSON tokens as values). The 'key' of the root token can be any custom defined name (in Listing 4.6 is `"*"`) that serves as a placeholder. That approach adds some additional overhead but guarantees that the JSON tokens will be extracted in their sequential array order.

```
{
  "*":[
    {"address 1" : "value 1"},
    {"address 2" : "value 2"},
    ...
  ]
}
```

Listing 4.6 JSON write request object with embedded array structure

When implementing the parsing algorithm for the server application with the help of an external JSON library (discussed in Section 5.2), some disadvantages regarding this structure were realized. The main disadvantage is that traversing the tokens by 'key' is supported by the library and much easier to implement. In this case those 'keys' are addresses and cannot be known in advance. Therefore, the final version of the structure of a communication JSON data object is the one shown in Listing 4.7. Addresses and their corresponding values are placed at the same index in the two arrays, ordered according to their desired access sequence. The JSON 'key' names – 'A' and 'V' (stand for 'Addresses' and 'Values') – are intuitively chosen and are known by both client and server applications. This way the up-mentioned disadvantage is removed.

```
{
  "A": [address1, address2, address3, ...],
  "V": [value1, value2, value3, ...]
}
```

Listing 4.7 Final structure of write request JSON data object

The established JSON structure for the *write request data objects* can be easily integrated to the *read request data objects*. The difference is in the contents of the 'Values' array: if the "V" array contains values – write those values to the addresses from the "A" array (write request); if the "V" array is empty – fill it with the values read from the addresses from the "A" array (read request). Therefore, the *response data object* will have the same structure as the *write request object* (Listing 4.8).

Read request JSON object

```
{
  "A": [address1, address2, address3],
  "V": []
}
```

Write request/Response JSON object

```
{
  "A": [address1, address2, address3],
  "V": [value1, value2, value3]
}
```

Listing 4.8 JSON structure of R/W request and response data objects

If an error has occurred during the process of parsing/composing the request/response JSON object, an *error object* is returned (Listing 4.9). It has the agreed JSON structure but no values in either array.

```
{ "A": [], "V": [] }
```

Listing 4.9 Error JSON object contains no values in its arrays

After a JSON structure for the communication data objects is established, the next step is to discuss the representation format of the register addresses and values. As defined by the project requirements, the number format representation is hexadecimal. In this case both addresses and the values are represented as 'string', as JSON format does not support hexadecimal representation of numbers [27]. The software application takes care for the correct parsing and composing of the hexadecimal strings.

Configuration data may be too big for the receive buffer if sent in one JSON object. That contradicts with the requirement that the algorithm implementation shall ensure no receiver buffer overflows during transmission. On the other hand, in some cases time delays between consecutive write accesses during configuration are required. Therefore, a ***request-response communication design*** is considered. Instead of sending the configuration data over to the node in one go, the client composes and transmits a JSON object containing one register address and one register value and waits for a response from the server before sending the next data object. This procedure is repeated in the background without further user interaction until all configuration data is sent. This method increases the overall configuration time but due to the fast Ethernet speed of 1 Gbit/s it is barely noticeable. However, the advantages are that relatively small JSON objects are transmitted over the network that guarantees no buffer overflows, and time delays between consecutive write accesses are provided.

4.3 Software Design

In this section are discussed the conceptual decisions that were taken, regarding the design of a software application that best responds to the set project requirements. That involves discussion on the features and the outlook of a 'user friendly' graphical user interface (GUI), together with the programming design practices that are implemented in the client and server application.

4.3.1 Graphical User Interface

According to the established project requirements, regarding the design of a graphical user interface (GUI), it shall provide the user with control functionalities and be a helping tool for performing the following services:

- Provide the configuration data contained in the config files to the application
- Connect to a particular node without hardware interventions
- Perform initial node configuration
- Control the node's Communication Controller by sending FlexRay commands
- Monitor node's message buffers
- Display message buffer contents and save it to file

The first considered option is to create a simple GUI based on a command prompt design. The user enters text commands that are translated into computer commands and executed in the background by the client application. The advantage with this approach is that it requires less time invested in the implementation of the GUI, as it contains only a few components, most important of which are: one text field for command input, one button to send the command to node and another text field for displaying the received content. However, although being so simple in design, it requires deeper knowledge of the FlexRay protocol as the user has to know the purpose and syntax of every FlexRay command. Also, besides the standard FlexRay commands, this approach requires the implementation of some custom user defined commands for the different use cases, like: connecting to a node, loading configuration data, save data to file and so on. Considering the variety of use cases, the amount and complexity of these commands may arise. Therefore, this approach contradicts with the project requirement that this software shall offer a platform that is easy and simple to work with, not targeted only to a qualified audience. Another inconsistency with the project requirements is that accepting user input is usually accompanied by validating algorithms that adds an additional load over the complexity of the parsing algorithm design.

The second option is the front end of the user interface to be based on a user-friendly button-controlled design that enables the user to perform complex operations with a simple button click. That adds complexity in the GUI design but eases the operation of the software and hence enables a wider range of users to work with it, which corresponds better to the project requirements. In addition to that, the possibilities of user control input are limited to selections of pre-coded command definitions (Lists, checkboxes, buttons), which reduces the needs for implementation of input validation algorithms. That is the chosen for this project GUI design.

4.3.2 Operation Control Flow

Operation control flow represents a defined pathway for sequential execution of individual requests, instructions, statements and so on [29]. All three fundamental, for this project, processes of **initial configuration** of the FlexRay registers of a node, **control** of the node's Communication Controller (CC) and **monitoring** of the FlexRay message buffers, follow a predefined control flow. According to the project requirements and the established design of the GUI (discussed in the previous section), each of these processes shall be executed in the background after user initiation. Therefore, their control flow shall be embedded in the code by the programmer or be guided by the user via button clicks, selections and so on.

The possibility of implementing some scripting procedure in the FlexRay server application algorithm, determining the flow of actions, contradicts with the established project requirements for maintaining the resource power of the PC client. Therefore, the client is the one to determine the control flow of all three processes.

Configuration and Control processes

Configuration is a process that follows a certain control flow that shall be accompanied by time delays between consequent register write accesses [28]. In general, for any FlexRay node, the overall configuration process can be divided in three steps:

1. Pre-configuration process – bringing the node's CC into the **config** state and clearing the message RAMs.
2. Configuration process – consequential execution of write accesses to the FlexRay registers.
3. Post-configuration process – execution of procedure for unlocking the FlexRay registers for access and transition to **ready** state.

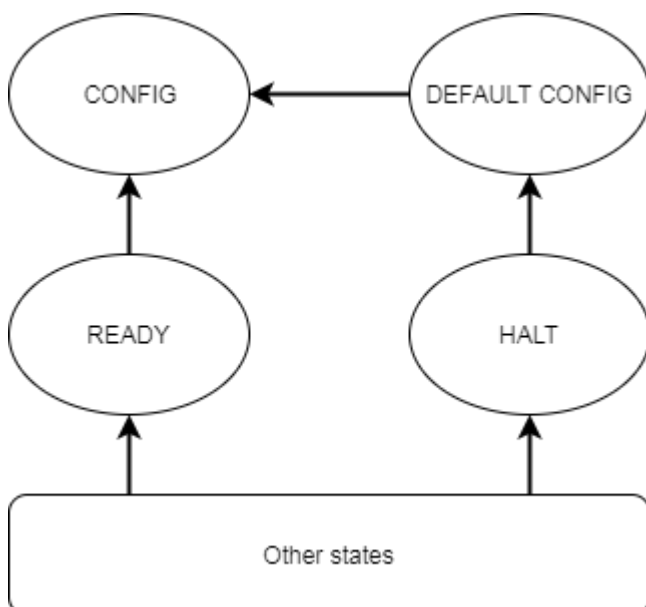


Fig 4.1 Possible transition routes to **config** state

Step 1 is part of the overall control process, which is expressed in control of the node's CC via FlexRay CHI²¹ commands. In general there are two possible routes to take in order to get the node into the **config** state without the need of a hard reset. The first route is via **ready** state and the second route is via **halt** and **default config** states (Fig 4.1). Step 2 represents writing generated by the application or defined by the user values to the respective register addresses, based on the data provided in the node configuration files. Step 3 is based on sequential write accesses to a dedicated lock register and transition to **ready** state.

²¹ Controller Host Interface (2.3.1 Communication Controller)

There are two options considered, regarding the implementation of the control flow in the software operation. The first option is to grant the control flow in the hands of the user (Fig 4.2 (a)). For that purpose the GUI provides possibility for the user to request write access to registers and control the CC by sending FlexRay CHI commands and acquiring the current CC state. This way it is the user who decides which route (Fig 4.1) to take and access to which register to request next. The other option is the control flow to be embedded in the client application algorithm (Fig 4.2 (b)). That increases the required implementation time and complexity but eases the operation process. As both approaches have their advantages and complement each other, both are implemented in the software.

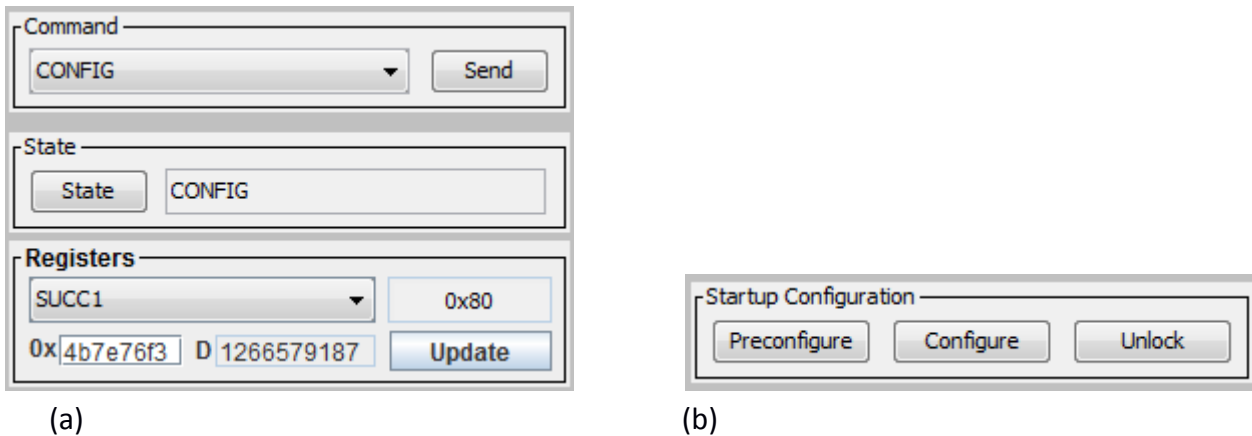


Fig 4.2 Configuration control flow (a) in the hands of the user; (b) embedded in the code

Monitoring process

The monitoring process is expressed in acquiring data from particular message buffers, for which there is a 'new data' flag raised in the dedicated for that purpose 'New Data' registers [11]. As it can be predicted neither by the user, nor by the programmer when new data is received by the FlexRay node, the only option considered, regarding the implementation of the monitoring process, is the control flow to be embedded in the application code. Once a connection between the PC client and the FlexRay node is established, the control of process initiation and abortion, as well as saving of data to files, is provided by the user, while the flow of instructions for acquiring message buffer data is accomplished due to the procedure described in Host Read Access via Output Buffer.

4.3.3 Programming Model

According to the project requirements, due to the lack of a real FlexRay system, this project is intended to be extended and finalized in the future by someone else. Therefore, good programming techniques are required, in order to ease the process of reading and understanding the code. In this section are discussed the different approaches considered, regarding the structure and implementation of the client and server applications.

For the client application the Model-View-Controller (MVC) design pattern is used [30]. MVC is a software engineering pattern where the user interactions are strictly separated from the algorithm logic. The View is a static component, typically a GUI that accepts user commands and passes them to the Controller. The Controller requests data to and from the Model and updates the View. Here is where the algorithm logic and decision taking is done. The Model executes the Controller requests and provides data that could be read from a file, result of calculations, measurements, enumeration and so on.

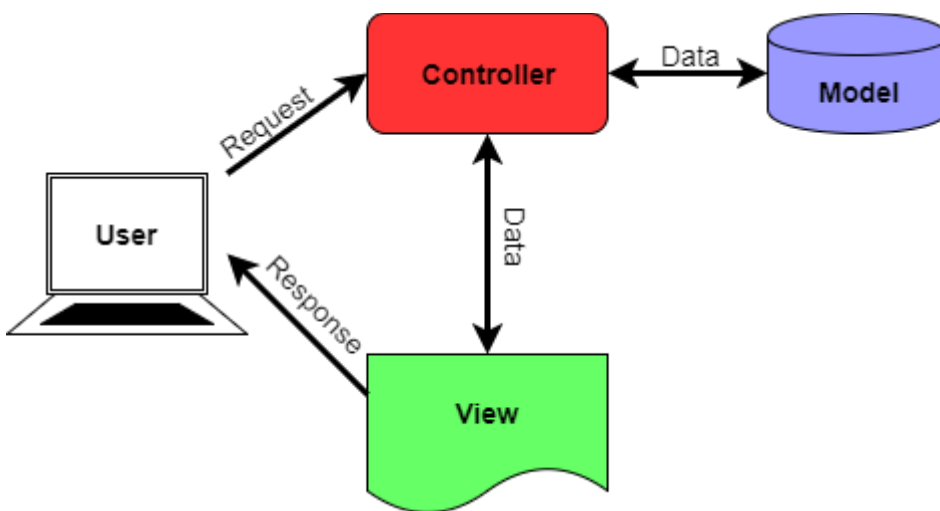


Fig 4.3. Model-View-Controller design pattern

From a programming point of view, separating the program code into three interconnected parts gives a structure that increases the code readability and results in easier modifications of each part without affecting the other two. Each part is independent and therefore can be reused in other applications.

One of the reasons for choosing JSON, as the established data format for communication between the PC client and the FlexRay network participants, is that it has a wide support of tested and open sourced libraries in various programming languages. Referring to an external library that is approved and tested by other programmers, saves us time as it decreases the complexity in the development of the algorithms for processing the JSON data objects.

There are various JSON libraries written in Java language that have proven abilities in processing JSON data and can be freely used. Some of the most popular among the programmers and hence considered as possible candidates are: JSON.simple, GSON, Jackson and JSONP [31]. Each of these libraries has some advantages over the others in different use cases and environments. The criterion, for choosing the best for the client application JSON library, lies in the project requirements. For our project the chosen library is expected to be able to process JSON data of different sizes equally good in terms of speed and convenience. The size of the JSON data objects may vary from just a few bytes to tens and hundreds of kilobytes (for the configuration files). Comparing the four candidate libraries shows that *JSON.simple* performs equally well for different sizes of JSON data [31]. Besides that it is lightweight, flexible and has no dependencies on other external libraries [32]. That makes JSON.simple the most appropriate option for our project.

In addition to the criteria for selecting the most appropriate JSON library, set for the client application, for the server application an important requirement is the lightweight. Here the choice is eased by the fact that such a research was already made in a previous project assignment [26]. It has shown that the library that corresponds best to the project requirements is the 'jsmn' (pronounced like 'jasmine') library.

The jsmn library [33] is written in C language by Serge Zaitsev and is specially designed for resource limited embedded system environments. It is highly portable as it involves no external or non standard C libraries. It uses no memory allocation and contains no token data but holds only the token boundaries and the number of child objects for each JSON object, which allows traversing to the token of interest. It is designed to work even with erroneous data which makes it robust against data losses that can occur during the transmission.

5. Implementation

The implementation of a software application refers to the realization of the software requirements on a programming level and its detailed discussion might include the necessity of a deeper knowledge of the programming language used. This will significantly overload the section with information that is barely or not at all related to the project's topic. Therefore, in this chapter are discussed more general overviews of the overall code structure and design of both client and server applications and the sequential flow of some of the most fundamental for this project use cases – establishing a TCP/IP connection between the PC client and the FlexRay server, configuration of the FlexRay node and monitoring of its message buffers. For more details, regarding the implemented procedures and functions, refer to the comments provided in the programming code.

5.1 Client Application

Following the concept decision that is made, regarding the implementation of the client application, the structure and relation of Java classes is according to the *Model-View-Controller* (MVC) design pattern and is visualized in Fig 5.1.

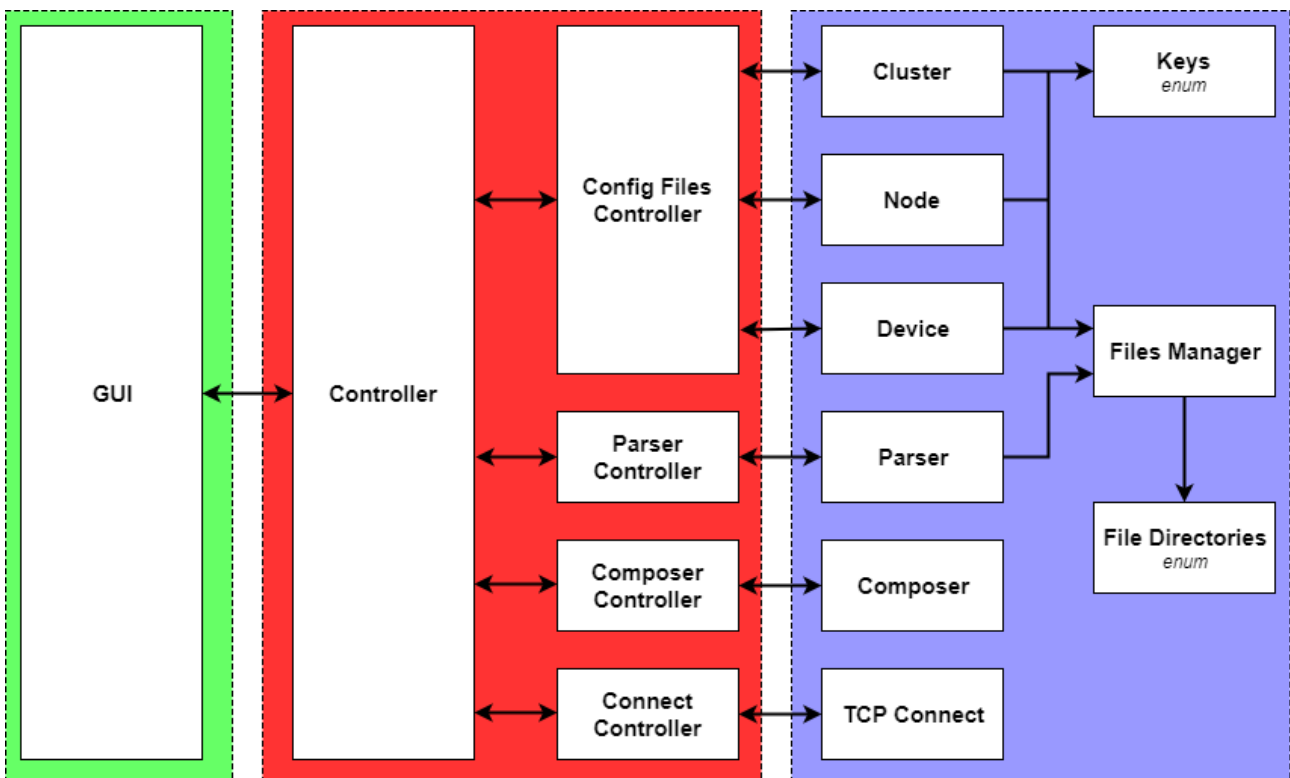


Fig 5.1. Class diagram for the client application, based on the MVC model

The Controller part of the MVC model plays the role of a mediator between the user interactions and the services executed in the background. Its code and complexity arises having in mind the complex nature of the project. Therefore, the Controller functionalities are separated among multiple controller classes. Each of these sub-controllers manages a certain part of the Model under the guidance of a main Controller class. Inside the main Controller class there are two inner


classes – Input Buffer and Output Buffer (not shown on the figure). Their purpose is to further improve code structure, as they hold only functions that are directly related to requesting read and write access to the message buffers of the node's message RAM.

The purpose of the *Model* part is to provide the requested by the *Controller* data and services. The Cluster, Node and Device classes contain functions related to the parsing of the JSON data contained in the three configuration JSON files, established in Section 4.1. The content of each file is loaded in the application memory and broken down into its constituent JSON objects. That necessitates the standardisation of the JSON object 'key' names, as per their definition established in 4.1.2 Files Structure, and providing them to the application via the Keys enumeration. That implementation occupies more memory but increases the speed when traversing for the JSON token values. This approach is a consequence to the requirement that the program implementation shall be targeted to the speed of execution of the parsing algorithms.

The Parser and Composer classes are dedicated to functionalities, related to the parsing and composing of communication JSON data objects (4.2 Communication Data Objects). The TCP Connect class is responsible for establishing a connection between the PC and the desired node. All file manipulations are managed by the Files Manager class. The file directories have to also be provided to the application and are hard coded in the File Directories enumeration.

Use cases

Connection

After selecting the desired node ID (provided in a list of IDs), its IP address and port number, it is bounded to, are automatically displayed in text fields. The user can then request a connection to the FlexRay node by pressing the 'Connect' button provided by the GUI. If connection is successful, the  icon is displayed next to the button. When the connection is terminated the icon disappears as soon as the client attempts to send data.

In order to expand the testing possibilities, a small local server application is implemented. It basically does the following: opens a TCP/IP socket on the local server with IP address 127.0.0.1 on an assigned by the programmer port, listens and accepts messages, modifies them depending on the test case and sends those modified messages back. The purpose of this application is to mimic a second FlexRay node in order to test the functionality of the software to successfully disconnect from one node and connect to another. Another useful purpose is to create some additional testing opportunities by making use of the provided Java functions.

Configuration

In 4.3.2 Operation Control Flow are introduced two possibilities of performing initial configuration of the FlexRay registers of a node. The first one is directed by the user and is realized in requesting write and read accesses to registers specified by the user. In the second option the whole configuration process is embedded and once requested, it is automatically performed by the

application in the background. The user is being informed via status messages whether the operation was successful or not. Fig 5.2 depicts that automated process for initial node configuration.

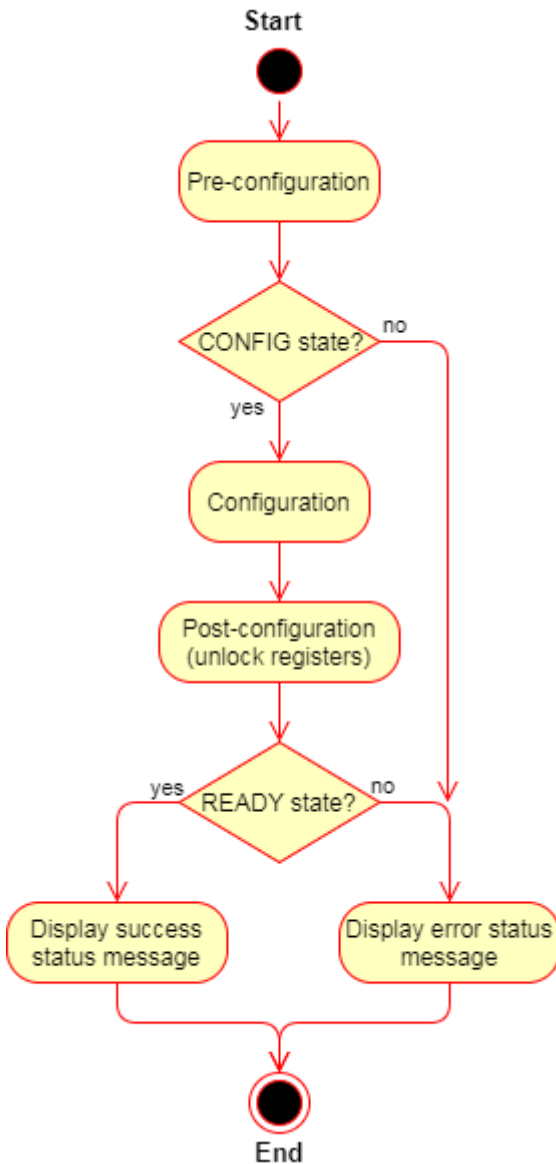
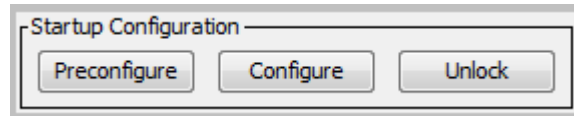


Fig 5.2. Activity diagram of the configuration process

The pre-configuration process is expressed in attempt to establish the node's communication controller (CC) into the *config* state (2.3.1 Communication Controller). If the node is not set into that state an error message is displayed and the process exits. However, if the node has entered *config* state, the client starts sending configuration data in the order specified in the Device JSON file (Device file). When all configuration data has been sent and written into the node's FlexRay registers, the procedure of exiting *config* and entering *ready* state is started. It is expressed in a sequential writing of predefined values to a dedicated Lock register (for details see [11] p. 24). Depending on whether the CC has entered *ready* state the appropriate message is displayed and the process is exited.

Although, on Fig 5.2 the whole configuration process is visualized as one uninterrupted chain, it is



implemented in the GUI behind three buttons – each dedicated to one of the three configuration processes: pre-configuration, configuration and post-configuration. The reason is that if, for example, the program fails to successfully execute the unlocking procedure (post-configuration), the user can retry it without redoing the preceding two processes again.

Monitoring

Another use case that is of high importance for the project is the monitoring of the FlexRay message buffers and requesting read access to their contents in message RAM. As the user cannot predict when or where is there going to be a new message available, the monitoring process is implemented in continuous polling of the New Data registers²². That polling process is realized in an endless while loop, which start and stop conditions are controlled by the user. It is implemented in its own thread so that the other functionalities of the software are not blocked (Fig 5.3).

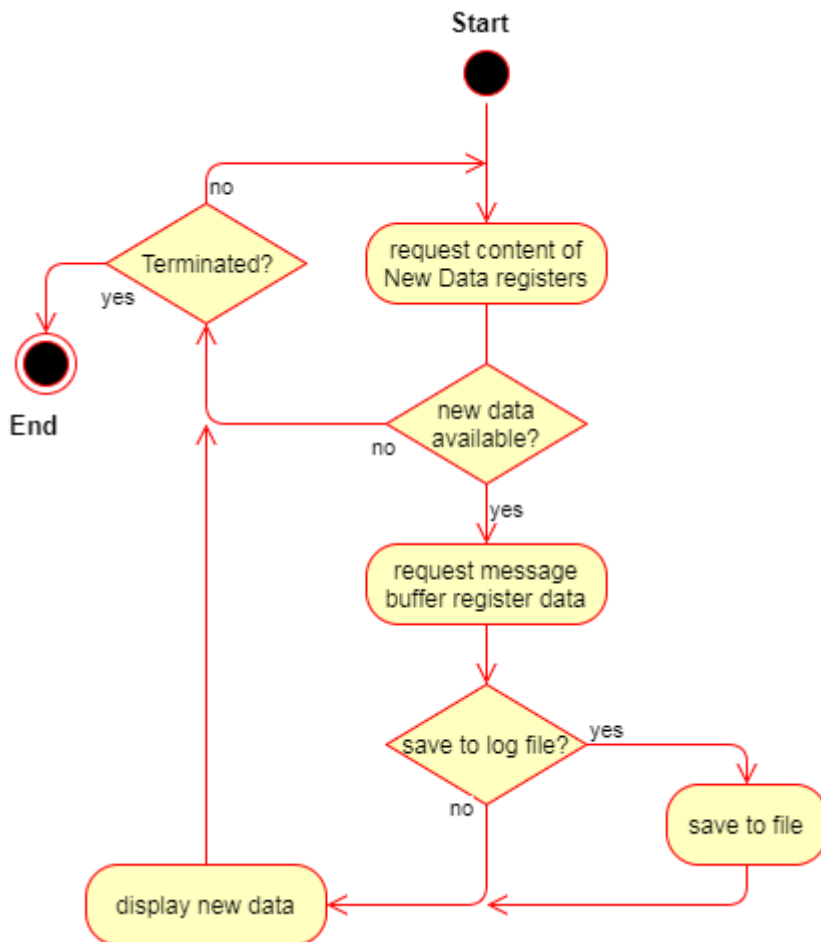


Fig 5.3. Activity diagram of the monitoring process

²² For the 128 message buffers in E-Ray there are four 32-bit New Data registers. The position of each bit in the 4 registers represents the message buffer number and the bit value represents a flag that is set to 1 when there is new data available in the corresponding message buffer. The flag is cleared automatically after data has been requested [11].

After the monitoring process is started by the user, the New Data registers of the node are polled. When a 'new data' flag for a given message buffer is raised, the data is requested and displayed in the GUI and optionally can be saved on a file. The boolean condition of the while loop is implemented in a volatile variable that is inverted via a button.

5.2 Server Application

According to the project requirements the server application is limited to perform the following tasks:

1. Assign an IP address to the node and bind it to a defined port
2. Listens the port and accepts data
3. Parses the received data and depending on the values in the 'V' array (4.2 Communication Data Objects) execute the requested access
4. Compose a response JSON object
5. Send data back to the client

The Xilinx SDK IDE provides a few ready-to-go examples that can be used as a starting point for some project developments. One of these examples is the 'Lwip Echo Server' which purpose, usually, is to test whether a microcontroller supports the lwip stack. It basically does the following – assigns an IP address to the microcontroller device, opens a TCP/IP socket on a pre-assigned port, receives data packets and echoes those packets back to the source. This small C program already implements steps 1, 2 and 5 from the list above, which makes it suitable to be used and extended in order to meet the requirements for our server application.

Extending the 'Lwip Echo Server' example requires additional parser and composer algorithms to be implemented. The parser application parses the received JSON objects and depending on the values in the 'V' array it executes the requested access to the node's FlexRay registers. According to the project requirements, due to the lack of a real FlexRay bus, the FlexRay register space is represented as a global array of 32-bit integers and the R/W accesses are performed on that array.

Composer application composes the response JSON object, which structure and contents are determined in Section 4.2 Communication Data Objects. This object is then transmitted back to the client. If an error occurs during parsing or composing process, an *error object* is returned (Listing 4.10 in Section 4.2).

The TCP/IP port number is assigned in the 'start_application' function, which is placed in the provided by the 'Lwip Echo Server' program *echo.c* file. By default it is set to 7, which is the Telnet port. Therefore, Telnet must be enabled on the PC client or another port has to be used. The device IP address, subnet mask and default gateway can be modified in 'main.c' file. The parsing and composing functions are stored in the *parser.c* file. The two files of the 'jsmn' library are *jsmn.c* and *jsmn.h*.

6. Evaluation

The current project task is evaluated based on the requirements specified in Section 3.4 Evaluation. The investigation of the correct operation of most of the implemented algorithms can only be performed via demonstration. Having the available information contained in the configuration JSON files we can manually pre-calculate the expected outcome of the different use cases and compare it to the actual result of the implemented application algorithms.

The correct operation of the software assumes that the provided configuration JSON files are constructed according to the standards set in the Concept chapter and they contain reasonable FlexRay data. Based on that data we can make conclusions whether the parsing and composing algorithms, algorithms for composition and decomposition of register values, as well as encoding and decoding of FlexRay messages, are correctly implemented. The correct execution of the requested R/W accesses can be determined by monitoring the global ‘register space’ array and its values.

The TCP/IP communication can be examined with the help of a network protocol analyzing tool. In our case it is the WireShark tool [34]. It enables the user to investigate the data packets that are sent over the network and provides detailed information about their size, contents and so on.

On Fig 7.1 (a) we can see the request of the client (with IP address 192.168.1.9) to read server (with IP 192.168.1.10) register address “0x714”. As the test mock-up register space array consists of only 16 addresses, the address value is calculated as modulo 16. Therefore, the server response shown on Fig 7.1 (b) has the address “0x4” (array index 4) and the value “0x3” read from that address (index).

No.	Time	Source	Destination	Protocol	Length	Info
8	5.975031	192.168.1.9	192.168.1.10	ECHO	77	Request
9	5.975353	192.168.1.10	192.168.1.9	ECHO	80	Response
10	5.978947	192.168.1.9	192.168.1.10	ECHO	82	Request
11	5.979260	192.168.1.10	192.168.1.9	ECHO	80	Response
12	6.006056	192.168.1.9	192.168.1.10	ECHO	84	Request
13	6.006448	192.168.1.10	192.168.1.9	ECHO	82	Response
14	6.037241	192.168.1.9	192.168.1.10	ECHO	77	Request
15	6.037585	192.168.1.10	192.168.1.9	ECHO	80	Response
16	6.068480	192.168.1.9	192.168.1.10	ECHO	84	Request

<ul style="list-style-type: none"> ▷ Frame 8: 77 bytes on wire (616 bits), 77 bytes captured (616 bits) on interface 0 ▷ Ethernet II, Src: Dell_da:f0:e2 (34:17:eb:da:f0:e2), Dst: Xilinx_00:01:02 (00:0a:35:00:01:02) ▷ Internet Protocol Version 4, Src: 192.168.1.9, Dst: 192.168.1.10 ▷ Transmission Control Protocol, Src Port: 49305, Dst Port: 7, Seq: 1, Ack: 1, Len: 23 ▷ Echo 						
0000	00 0a 35 00 01 02 34 17	eb da f0 e2 08 00 45 00	..5...4.E.			
0010	00 3f 23 fb 40 00 80 06	00 00 c0 a8 01 09 c0 a8	.?.@...			
0020	01 0a c0 99 00 07 24 d6	02 a5 00 00 27 8c 50 18\$.'P.			
0030	f8 7a 83 95 00 00 7b 22	41 22 3a 5b 22 30 78 37	.z....{" A":["0x7			
0040	31 34 22 5d 2c 22 56 22	3a 5b 5d 7d 0a	14"],"V" :[]}.}			

(a)

```

▶ Frame 9: 80 bytes on wire (640 bits), 80 bytes captured (640 bits) on interface 0
▶ Ethernet II, Src: Xilinx_00:01:02 (00:0a:35:00:01:02), Dst: Dell_da:f0:e2 (34:17:eb:da:f0:e2)
▶ Internet Protocol Version 4, Src: 192.168.1.10, Dst: 192.168.1.9
▶ Transmission Control Protocol, Src Port: 7, Dst Port: 49305, Seq: 1, Ack: 24, Len: 26
▶ Echo

0000  34 17 eb da f0 e2 00 0a  35 00 01 02 08 00 45 00  4..... 5.....E.
0010  00 42 00 85 00 00 ff 06  37 cd c0 a8 01 0a c0 a8  .B..... 7.....
0020  01 09 00 07 c0 99 00 00  27 8c 24 d6 02 bc 50 18  ..... '$...P.
0030  05 9d 6d 26 00 00 7b 22  41 22 3a 5b 22 30 78 34  ..m&..{" A":["0x4
0040  22 5d 2c 22 56 22 3a 5b  22 30 78 33 22 5d 7d 0a  "],"V":[" 0x3"]}.

No.: 9 · Time: 5.975353 · Source: 192.168.1.10 · Destination: 192.168.1.9 · Protocol: ECHO · Length: 80 · Info: Response

```

(b)

Fig 7.1 Monitoring of data packets via WireShark

From Fig 7.1 (a) we can make conclusions about the time needed for the server application to receive the client request, process it, compose a JSON response object and send it back. The medial value for a single register request is calculated to be approximately 326µs.

The request-response implementation design established in Section 4.2 implies that relatively small objects are transmitted between the network participants. More specifically, the largest data object that is being transmitted is the message data contained in a FlexRay message buffer, assuming that all data registers are used. For the E-Ray (2.3.3 E-Ray) the number of data registers is 64, which means that the maximum size of a communication JSON data object is less than 1 kB²³. That ensures that no buffer overflows will occur during transmission as long as the receive message buffer is at least of 1 kB of size.

²³ A string consisting of 64 register addresses and 64 register values, each 32 bits long, plus JSON format symbols (parentheses, brackets, commas, colons, etc.), plus protocol data equals to less than 1 kB of size

7. Conclusion

The result of this project is a software application, developed to meet the needs for establishing a bidirectional communication between a computer and a microcontroller device, part of a FlexRay cluster. As the project title implies, the purpose of this software is to enable a client to perform initial configuration, control and monitoring of a FlexRay controller's registers. Following the project requirements, the implementation of this application is realised by a graphical user interface (GUI), designed to provide services that enables the user to carry out complex operations with little user interactions.

Being one of the newest automotive communication standards the FlexRay protocol has a high level of complexity. Creating a software application that covers all aspects of the FlexRay standard is barely achievable by one person for the time bounds set for completion. Therefore, the current project is targeted only towards some particular parts of the whole realization. As being a continuation of a previous research made on that topic [26], this project is intended to be extended in the future to provide improved and complete services, including compatibility to other automotive protocols.

Due to the lack of a real FlexRay module, the testing and evaluation of the developed software application is done according to the correct implementation of the software algorithms, as defined by the project requirements. The next step would be testing it with a real FlexRay module. The configuration and control of the Communication Controller (CC) is protocol specific, i.e. once the configuration files are created and the header register-address macro is present, the implemented algorithms shall work for every FlexRay device. The Controller Host Interface (CHI) is, however, implementation specific. Therefore, the procedure for monitoring the FlexRay message buffers may differ in each distinct device. As for this project the access to the FlexRay message RAM is implemented according to the E-Ray user manual [11] provided by Bosch, testing it with such a device is most recommended. Otherwise, additional changes in the programming code might be necessary to enable compatibility to other monitoring procedures.

For this project, a connection to only one FlexRay node at a time is required. As following, the program shall be extended to simultaneously support connection to multiple nodes, by introducing a multilayered GUI structure that opens a new panel for every newly connected node and potentially closes it when node is disconnected.

The last step would be migrating to other than FlexRay protocols. As per the project requirements, the software is designed to provide simple user control by executing most of the operations complexity in the background. However, many of these implemented procedures are FlexRay specific and cannot be applied to other protocols. Therefore, for integration of other communication protocols, it might be necessary whole parts of the current software application to be modified. In such cases the development of a completely new GUI application shall be considered.

Bibliography

- [1] (2017, Dec.) HAW Hamburg. [Online]. <https://www.haw-hamburg.de/fakultaeten-und-departments/ti/unsere-fakultaet/zukunftsprogramm-der-fakultaet-ti/geoerderte-projekte.html#c125639>
- [2] (2017, Dec.) FlexRay. [Online]. <https://en.wikipedia.org/wiki/FlexRay>
- [3] (2017, Dec.) Network Switch. [Online]. https://en.wikipedia.org/wiki/Network_switch
- [4] (2017, Nov.) System Bus. [Online]. https://en.wikipedia.org/wiki/System_bus
- [5] (2017, Nov.) Serial Bus Systems in the Motor Vehicle. [Online]. https://elearning.vector.com/index.php?&wbt_ls_seite_id=507973&root=378422&seite=vl_sbs_introduction_en
- [6] (2017, Nov.) Serial Bus Systems In The Motor Vehicle. [Online]. https://elearning.vector.com/index.php?&wbt_ls_seite_id=507955&root=378422&seite=vl_sbs_introduction_en
- [7] (2017, Nov.) The OSI Model Layers from Physical to Application. [Online]. <https://www.lifewire.com/layers-of-the-osi-model-illustrated-818017>
- [8] (2017, Nov.) The 7 Layers of the OSI Model. [Online]. https://www.webopedia.com/quick_ref/OSI_Layers.asp
- [9] Charles M. Kozierok, *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference.*, 2005.
- [10] FlexRay TM. FlexRay Communications System Protocol Specification Version 2.1.pdf.
- [11] (2017, Oct) FlexRay IP-Module User's Manual - Bosch Semiconductors and Sensors. [Online]. http://www.bosch-semiconductors.de/media/en/automotive_electronics/pdf_2/ipmodules_3/flexray_1/eray_users_manual_1_2_7.pdf
- [12] (2011, Nov.) Finite State Machines. [Online]. https://en.wikipedia.org/wiki/Finite-state_machine
- [13] Francoise Simonot-Lion Nicolas Navet, Ed., *Automotive Embedded Systems Handbook.*: CRC Press, 2009.
- [14] (2011, Nov.) Ethernet. [Online]. <https://en.wikipedia.org/wiki/Ethernet>
- [15] Charles E. Spurgeon, *Ethernet: The Definite Guide*, 1st ed. USA: O'Reilly Media, Inc., 2000.
- [16] (2017, Nov.) The IEEE 802.3 Standard. [Online]. https://en.wikipedia.org/wiki/IEEE_802.3
- [17] Gilbert Held, *Ethernet Networks: Design, Implementation, Operation, & Management*, 4th ed.

USA: John Wiley& Sons, Ltd., 2003.

- [18] IBM, *TCP/IP Tutorial and Technical Overview.*, 2006.
- [19] (2017, Nov.) TCP vs UDP - Difference and Comparisson. [Online]. http://www.diffen.com/difference/TCP_vs_UDP
- [20] (2017, Nov.) What is DHCP? [Online]. [https://technet.microsoft.com/en-us/library/dd145320\(v=ws.10\).aspx](https://technet.microsoft.com/en-us/library/dd145320(v=ws.10).aspx)
- [21] (2017, Nov.) LwIP: Overview. [Online]. http://www.nongnu.org/lwip/2_0_x/index.html
- [22] (2017, Nov.) LwIP - Application API Layers. [Online]. http://lwip.wikia.com/wiki/Application_API_layers
- [23] (2017, Nov.) LwIP - Raw/TCP. [Online]. <http://lwip.wikia.com/wiki/Raw/TCP>
- [24] (2017, Nov.) Zynq-7000 All Programable SoC Data Sheet. [Online]. https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf
- [25] (2017, Nov.) MicroZed HW User Guide. [Online]. http://microzed.org/sites/default/files/documentations/MicroZed_HW_UG_v1_4.pdf
- [26] Laxmi Kumar Ghising Tamang, *Development of a TCP/IP based diagnostic access to an automotive FlexRay network.* Hamburg: HAW, 2016.
- [27] (2017, Oct.) JSON. [Online]. <http://json.org/>
- [28] (2017, Oct.) Computer Configuration. [Online]. https://en.wikipedia.org/wiki/Computer_configuration
- [29] (2017, Dec.) Control flow. [Online]. https://en.wikipedia.org/wiki/Control_flow
- [30] (2017, Oct) Model-View-Controller. [Online]. <https://www.tomdalling.com/blog/software-design/model-view-controller-explained/>
- [31] (2017, Oct.) The Ultimate JSON Library: JSON.simple vs GSON vs Jackson vs JSONP. [Online]. <http://blog.takipi.com/the-ultimate-json-library-json-simple-vs-gson-vs-jackson-vs-json/>
- [32] (2017, Oct.) JSON Simple. [Online]. <https://code.google.com/archive/p/json-simple/>
- [33] (2017, Oct.) JSMN. [Online]. <http://zserge.com/jsmn.html>
- [34] (2017, Dec.) Wire Shark Tool. [Online]. <https://www.wireshark.org/>
- [35] (2017, Dec.) Board Definition Files for MicroZed Board. [Online]. <http://zedboard.org/content/board-definition-files-0>

Appendix A

FlexRay parameter and variable names are self-describing for better readability. In order to include additional information about the type of the parameter, the name is preceded by one or two prefixes (Listing A1). The meaning of each prefix is shown in Table A-1.

<variable> ::= <prefix_1> [<prefix_2>] Name

<prefix_1> ::= a | c | v | g | p | z

<prefix_2> ::= d | s

Listing A1. Parameter prefix convention [10] (p. 18)

Naming Convention	Information Type	Description
a	Auxiliary Parameter	Used in the definition or derivation of other parameters or in the derivation of constraints.
c	Protocol Constant	Used to define characteristics or limits of the protocol. These values are fixed for the protocol and cannot be changed.
v	Node Variable	Its value varies depending on time, events, etc.
g	Cluster Parameter	Must have the same value in all nodes in a cluster. It is initialized in the <i>POC:default config</i> state and can only be changed while in the <i>POC:config</i> state.
p	Node Parameter	Parameter that may have different values in different nodes in the Cluster. It is initialized in the <i>POC:default config</i> state and can only be changed while in the <i>POC:config</i> state.
z	Local SDL Process Variable	Variables used in SDL processes to facilitate accurate representation of the necessary algorithmic behaviour. Their scope is local to the process where they are declared and their existence in any particular implementation is not mandated by the protocol.
d	Time Duration	Value (variable, parameter, etc.) describing a time duration, the time between two points in time
s	Set	Set of values (variables, parameters, etc.)

Table A-1 Parameter prefixes [10] (p. 18)

The relation between the CHI, POC and the core protocol mechanisms is shown on Fig A.1

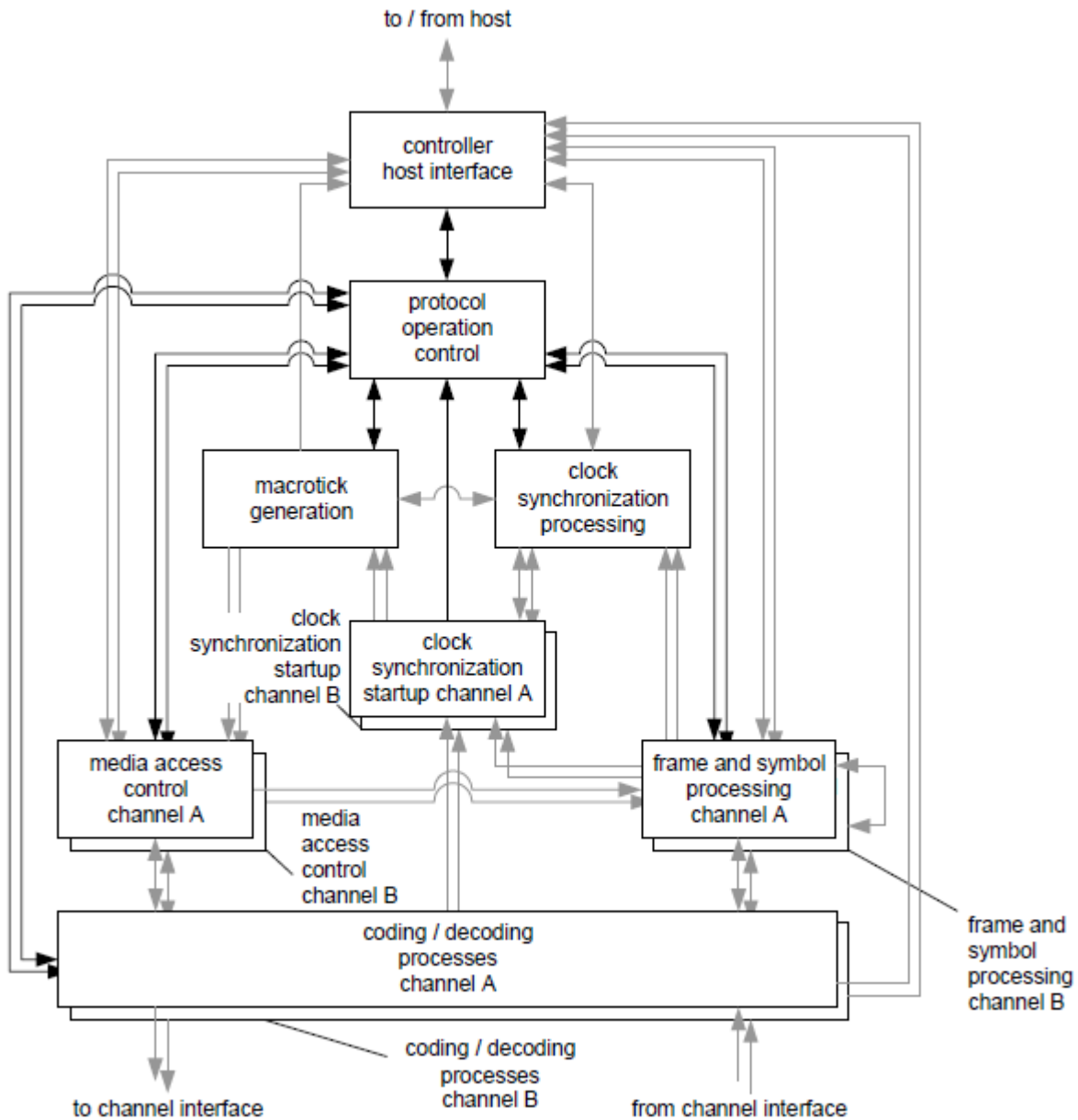


Fig A.1 Protocol operation control context ([10] p. 32)

List of CHI commands according to the Bosch E-Ray User Manual v.1.2.7 [11]:

Register name: SUCC1

Bit field: CMD[3:0]

Offset: 0

- | | | |
|-----------------------------|--------------------------------|-------------------|
| 0000 = command_not_accepted | 0110 = HALT | 1100 = CLEAR_RAMs |
| 0001 = CONFIG | 0111 = FREEZE | 1101 = reserved |
| 0010 = READY | 1000 = SEND_MTS | 1110 = reserved |
| 0011 = WAKEUP | 1001 = ALLOW_COLDSTART | 1111 = reserved |
| 0100 = RUN | 1010 = RESET_STATUS_INDICATORS | |
| 0101 = ALL_SLOTS | 1011 = MONITOR_MODE | |

List of POC states according to the Bosch E-Ray User Manual v.1.2.7 [11]:

Register name: CCSV

Bit field: POCS[5:0]

Offset: 0

Indicates the actual state of operation of the CC Protocol Operation Control

00 0000 = DEFAULT_CONFIG state

00 0001 = READY state

00 0010 = NORMAL_ACTIVE state

00 0011 = NORMAL_PASSIVE state

00 0100 = HALT state

00 0101 = MONITOR_MODE state

00 0110 ... 00 1110 = reserved

00 1111 = CONFIG state

Indicates the actual state of operation of the POC in the wakeup path

01 0000 = WAKEUP_STANDBY state

01 0001 = WAKEUP_LISTEN state

01 0010 = WAKEUP_SEND state

01 0011 = WAKEUP_DETECT state

01 0100 ... 01 1111 = reserved

Indicates the actual state of operation of the POC in the start-up path

10 0000 = STARTUP_PREPARE state

10 0001 = COLDSTART_LISTEN state

10 0010 = COLDSTART_COLLISION_RESOLUTION state

10 0011 = COLDSTART_CONSISTENCY_CHECK state

10 0100 = COLDSTART_GAP state

10 0101 = COLDSTART_JOIN State

10 0110 = INTEGRATION_COLDSTART_CHECK state

10 0111 = INTEGRATION_LISTEN state

10 1000 = INTEGRATION_CONSISTENCY_CHECK state

10 1001 = INITIALIZE_SCHEDULE state

10 1010 = ABORT_STARTUP state

10 1011 = STARTUP_SUCCESS state

10 1100 ... 11 1111 = reserved

Message RAM

Header Partition

Bit Word	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	P			MBI	TXM	PPIT	CFG	CHB	CHA		Cycle Code														Frame ID										
1	P		Payload Length Received									Payload Length Configured													Tx Buffer: Header CRC Configured Rx Buffer: Header CRC Received										
2	P		RES	RPIS	NFIS	SFIN	SFIS	RCIS				Receive Cycle Count													Data Pointer										
3	P		RES	RPIS	NFIS	SFIN	SFIS	RCIS				Cycle Count Status						FTB	FTA		MLST	ESB	ESA	ETB	ETA	STB	STA	CTB	CTA	VTB	VTA	CTB	CTA	VTB	VTA
...	P		...																																
...	P		...																																

	Frame Configuration
	Filter Configuration
	Message Buffer Control
	Message RAM Configuration
	Updated from received Data Frame
	Message Buffer Status MBS
	Parity Bit
	unused

Fig A.2 Header sections of a message buffer in the Header Partition of the message RAM ([11] p. 144)

Header 1 (word 0)

- Frame ID - Slot counter filtering configuration
- Cycle Code - Cycle counter filtering configuration
- CHA, CHB - Channel filtering configuration
- CFG - Message buffer direction configuration: receive / transmit
- PPIT - Payload Preamble Indicator Transmit
- TXM - Transmit mode configuration: single-shot / continuous
- MBI - Message buffer receive / transmit interrupt enable

Header 2 (word 1)

- Header CRC - Transmit Buffer: Configured by the Host (calculated from frame header)
- Receive Buffer: Updated from received frame
- Payload Length Configured - Length of data section (2-byte words) as configured by the Host

- Payload Length Received - Length of payload segment (2-byte words) stored from received frame

Header 3 (word 2)

- Data Pointer - Pointer to the beginning of the corresponding data section in the data partition

Valid for receive buffers only, updated from received frames:

- Receive Cycle Count - Cycle count from received frame
- RCI - Received on Channel Indicator
- SFI - Startup Frame Indicator
- SYN - Sync Frame Indicator
- NFI - Null Frame Indicator
- PPI - Payload Preamble Indicator
- RES - Reserved bit

Message Buffer Status MBS (word 3) – updated by the CC at the end of the configured slot:

- VFRA - Valid Frame Received on channel A
- VFRB - Valid Frame Received on channel B
- SEOA - Syntax Error Observed on channel A
- SEOB - Syntax Error Observed on channel B
- CEOA - Content Error Observed on channel A
- CEOB - Content Error Observed on channel B
- SVOA - Slot boundary Violation Observed on channel A
- SVOB - Slot boundary Violation Observed on channel B
- TCIA - Transmission Conflict Indication channel A
- TCIB - Transmission Conflict Indication channel B
- ESA - Empty Slot Channel A
- ESB - Empty Slot Channel B
- MLST - Message Lost
- FTA - Frame Transmitted on Channel A
- FTB - Frame Transmitted on Channel B
- Cycle Count Status- Actual cycle count when status was updated
- RCIS - Received on Channel Indicator Status
- SFIS - Startup Frame Indicator Status
- SYNS - Sync Frame Indicator Status
- NFIS - Null Frame Indicator Status
- PPIS - Payload Preamble Indicator Status
- RESS - Reserved bit Status

Data Partition

Bit Word	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
...	P	unused								unused								unused								unused							
...	P	unused								unused								unused								unused							
...	P	MBn Data3								MBn Data2								MBn Data1								MBn Data0							
...	P							
...	P							
...	P	MBn Data(m)								MBn Data(m-1)								MBn Data(m-2)								MBn Data(m-3)							
...	P							
...	P							
...	P	MB1 Data3								MB1 Data2								MB1 Data1								MB1 Data0							
...	P							
...	P	MB1 Data(k)								MB1 Data(k-1)								MB1 Data(k-2)								MB1 Data(k-3)							
2046	P	MB0 Data3								MB0 Data2								MB0 Data1								MB0 Data0							
2047	P	unused								unused								MB0 Data5								MB0 Data4							

Fig A.3 Data partition in the message RAM ([11] p. 145)

The beginning and the end of a message buffer's data section is determined by the data pointer and the payload length configured in the message buffer's header section, respectively. This enables a flexible usage of the available RAM space for storage of message buffers with different data length. If the size of the data section is an odd number of 2-byte words, the remaining 16 bits in the last 32-bit word are unused. [11]

Appendix B

Example of configuration files, according to the E-Ray user manual provided by Bosch [11].

Cluster File

```
{
  "general": {
    "id": 5,
    "protocol": "FR",
    "node_ids": [1,4,9,11],
    "port": 7
  },
  "parameters": {
    "gColdStartAttempts": "0xC",
    "gListenNoiseMinusOne": "0x2",
    "gMacroPerCycle": 10000,
    "gMaxWithoutClockCorrectionFatal": 11,
    "gMaxWithoutClockCorrectionPassive": 4,
    "gNetworkManagementVectorLength": "0xF",
    "gdTSSTransmitter": "0x1",
    "gdCASRxLowMax": "0x21",
    "gStrobePointPosition": 3,
    "gBaudRatePrescaler": 2,
    "gdWakeupSymbolRxWindow": "0xA0",
  }
}
```

```

    "gNumberOfMinislots":          512,
    "gNumberOfStaticSlots":        1024,
    "gOffsetCorrectionStart":      "0x1234",
    "gPayloadLengthStatic":        "0x55",
    "gSyncNodeMax":                 "0xE",
    "gdActionPointOffset":         "0x2",
    "gdDynamicSlotIdlePhase":      1,
    "gdMinislot":                   "0xA",
    "gdMinislotActionPointOffset": "0x0",
    "gdNITStart":                   666,
    "gdStaticSlot":                 500,
    "gdWakeupSymbolRxIdle":        "0xB",
    "gdWakeupSymbolRxLow":         "0x8",
    "gdWakeupSymbolTxIdle":        "0x4",
    "gdWakeupSymbolTxLow":         "0x6",
    "gNetworkIdleTimeStart":       "0x76"

```

```
    }
```

```
}
```

Node File

```
{
```

```

    "general": {
        "id": 1,
        "ip": "192.168.1.10",
        "device_id": 31542563
    },
    "command": {
        "CONFIG":          "0x01",
        "READY":           "0x02",
        "WAKEUP":          "0x03",
        "RUN":              "0x04",
        "ALL SLOTS":       "0x05",
        "HALT":             "0x06",
        "FREEZE":          "0x07",
        "SEND MTS":        "0x08",
        "ALLOW COLDSTART": "0x09",
        "RESET STATUS INDICATORS": "0x0A",
        "MONITOR MODE":    "0x0B",
        "CLEAR RAMS":      "0x0C"
    },
    "lock": ["0xCE", "0x31"],
    "control": {
        "cc_control": {
            "pKeySlotUsedForStartup": 0,
            "pKeySlotUsedForSync":    1,
            "pAllowPassiveToActive":  "0xA",
            "pWakeupChannel":          3,
            "pSingleSlotEnabled":      0,
            "pAllowHaltDueToClock":    1,
            "pChannelsA":               1,

```

```

        "pChannelsB": 0,
        "pdListenTimeout": "0x3210",
        "pWakeupPattern": "0x33",
        "pLatestTx": "0xAAA",
        "pMicroPerCycle": "0xBF9",
        "pMicroInitialOffset[A]": "0x12",
        "pMicroInitialOffset[B]": "0x34",
        "pMacroInitialOffset[A]": "0x11",
        "pMacroInitialOffset[B]": "0x22",
        "pDelayCompensation[A]": "0x56",
        "pDelayCompensation[B]": "0x78",
        "pClusterDriftDamping": "0xD",
        "pDecodingCorrection": "0x9A",
        "pdAcceptedStartupRange": "0x123",
        "pdMaxDrift": "0x456",
        "pOffsetCorrectionOut": "0x1928",
        "pRateCorrectionOut": "0xAF",
        "vExternOffsetControl": 1,
        "vExternRateControl": 2,
        "pExternOffsetCorrection": 3,
        "pExternRateCorrection": 4
    },
    "mb_control": {
        "pFirstDynamicBuffer": "0x5",
        "pFirstFIFOBuffer": 10,
        "pLastConfiguredBuffer": 20,
        "pChannelFilter": 3,
        "pFrameIDFilter": "0xAB",
        "pCycleCounterFilter": "0x78",
        "pRejectStaticSegment": 0,
        "pRejectNullFrames": 1,
        "pMaskFrameIDFilter": "0x80",
        "pCriticalLevel": "0xEF"
    }
},
"message_buffers": {
    "5": {
        "vRF!Header!FrameID": 3,
        "vRF!Header!CycleCode": 1,
        "vRF!Header!ChannelA": 1,
        "vRF!Header!ChannelB": 1,
        "vRF!Header!Direction": 0,
        "vRF!Header!PPIndicator": 0,
        "vRF!Header!TxMode": 0,
        "vRF!Header!BufferInterrupt": 1,
        "vRF!Header!HeaderCRC": 111,
        "vRF!Header!Length": 10,
        "vRF!Header!DataPointer": 1024,
        "vRF!CommandMask!LoadHeaderSection": 1,

```

```

        "vRF!CommandMask!LoadDataSection":1,
        "vRF!CommandMask!TxRequest": 0
    }
},
"status": {
    "cc_status": {
        "vPOC!State": {
            "0x00": "DEFAULT CONFIG",
            "0x01": "READY",
            "0x02": "NORMAL ACTIVE",
            "0x03": "NORMAL PASSIVE",
            "0x04": "HALT",
            "0x05": "MONITOR MODE",
            "0x0F": "CONFIG",
            "0x10": "WAKEUP STANDBY",
            "0x11": "WAKEUP LISTEN",
            "0x12": "WAKEUP SEND",
            "0x13": "WAKEUP DETECT",
            "0x20": "STARTUP PREPARE",
            "0x21": "COLDSTART LISTEN",
            "0x22": "COLDSTART COLLISION_RESOLUTION",
            "0x23": "COLDSTART CONSISTENCY_CHECK",
            "0x24": "COLDSTART GAP",
            "0x25": "COLDSTART JOIN",
            "0x26": "INTEGRATION COLDSTART_CHECK",
            "0x27": "INTEGRATION LISTEN",
            "0x28": "INTEGRATION CONSISTENCY_CHECK",
            "0x29": "INITIALIZE SCHEDULE",
            "0x30": "ABORT STARTUP",
            "0x31": "STARTUP SUCCESS"
        },
        "vPOC!Freeze": {
            "0x01": "FREEZE COMMAND SET"
        },
        "vPOC!CHIHaltRequest": {
            "0x01": "HALT COMMAND SET"
        },
        "vPOC!SlotMode": {
            "0x00": "SINGLE",
            "0x02": "ALL PENDING",
            "0x03": "ALL"
        },
        "vPOC!ColdstartNoise": {
            "0x01": "COLDSTART NOISE"
        },
        "vColdstartAbort": {
            "0x01": "COLDSTART ABORTED"
        },
        "vColdStartInhibit": {

```

```

        "0x00": "COLDSTART ENABLED",
        "0x01": "COLDSTART DISABLED"
    },
    "vPOC!WakeupStatus": {
        "0x00": "UNDEFINED",
        "0x01": "RECEIVED HEADER",
        "0x02": "RECEIVED WUP",
        "0x03": "COLLISION HEADER",
        "0x04": "COLLISION WUP",
        "0x05": "COLLISION UNKNOWN",
        "0x06": "TRANSMITTED"
    },
    "vPOC!ErrorMode": {
        "0x00": "ACTIVE",
        "0x01": "PASSIVE",
        "0x02": "COMM HALT"
    }
},
"mb_status": {
    "vSS!ValidFrameA": {
        "0x00": "No valid frame received on channel A",
        "0x01": "Valid frame received on channel A"
    },
    "vSS!ValidFrameB": {
        "0x00": "No valid frame received on channel B",
        "0x01": "Valid frame received on channel B"
    },
    "vSS!SyntaxErrorA": {
        "0x00": "No syntax error observed on channel A",
        "0x01": "Syntax error observed on channel A"
    },
    "vSS!SyntaxErrorB": {
        "0x00": "No syntax error observed on channel B",
        "0x01": "Syntax error observed on channel B"
    },
    "vSS!ContentErrorA": {
        "0x00": "No content error observed on channel A",
        "0x01": "Content error observed on channel A"
    },
    "vSS!ContentErrorB": {
        "0x00": "No content error observed on channel B",
        "0x01": "Content error observed on channel B"
    },
    "vSS!BViolationA": {
        "0x00": "No slot boundary violation observed on channel A",
        "0x01": "Slot boundary violation observed on channel A"
    },
    "vSS!BViolationB": {
        "0x00": "No slot boundary violation observed on channel B",

```

```

        "0x01": "Slot boundary violation observed on channel B"
    },
    "vSS!TxConflictA": {
        "0x00": "No transmission conflict occurred on channel A",
        "0x01": "Transmission conflict occurred on channel A"
    },
    "vSS!TxConflictB": {
        "0x00": "No transmission conflict occurred on channel B",
        "0x01": "Transmission conflict occurred on channel B"
    },
    "vSS!EmptySlotA": {
        "0x00": "Bus activity detected in the assigned slot on channel A",
        "0x01": "No bus activity detected in the assigned slot on channel A"
    },
    "vSS!EmptySlotB": {
        "0x00": "Bus activity detected in the assigned slot on channel B",
        "0x01": "No bus activity detected in the assigned slot on channel B"
    },
    "vSS!MsgLost": {
        "0x00": "No message lost",
        "0x01": "Unprocessed message was overwritten"
    },
    "vSS!FrameTransmittedA": {
        "0x00": "No data frame transmitted on channel A",
        "0x01": "Data frame transmitted on channel A"
    },
    "vSS!FrameTransmittedB": {
        "0x00": "No data frame transmitted on channel B",
        "0x01": "Data frame transmitted on channel B"
    },
    "vSS!Channel": {
        "0x00": "Frame received on channel B",
        "0x01": "Frame received on channel A"
    },
    "vRF!Header!SuFIndicator": {
        "0x00": "No startup frame received",
        "0x01": "The received frame is a startup frame"
    },
    "vRF!Header!SyFIndicator": {
        "0x00": "No sync frame received",
        "0x01": "The received frame is a sync frame"
    },
    "vRF!Header!NFIndicator": {
        "0x00": "Received frame is a null frame",
        "0x01": "Received frame is not a null frame"
    },
    "vRF!Header!PPIndicator": {
        "0x00": "No NM vector or message ID in payload segment",

```

```
"0x01": "Static segment: NM vector at the beginning of the payload.  
Dynamic segment: Message ID at the beginning of the  
payload"
```

```
}
```

```
}
```

```
}
```

```
}
```

Device File

```
{
```

```
  "general": {
```

```
    "id": 42685350,
```

```
    "name": "Arduino µC",
```

```
    "name_id": 2,
```

```
    "type": "E-Ray",
```

```
    "type_id": 1,
```

```
    "version": "Atmel v.3.1.4",
```

```
    "version_id": 3,
```

```
    "header": "FR register-address map.h"
```

```
  },
```

```
  "command": {
```

```
    "SUCC1": {
```

```
      "*": [4, 0]
```

```
    }
```

```
  },
```

```
  "state": {
```

```
    "CCSV": {
```

```
      "vPOC!State": [6, 0]
```

```
    }
```

```
  },
```

```
  "lock": {
```

```
    "LCK": {
```

```
      "*": [8, 0]
```

```
    }
```

```
  },
```

```
  "control": {
```

```
    "cc_control": {
```

```
      "SUCC1": {
```

```
        "pKeySlotUsedForStartup": [1, 8],
```

```
        "pKeySlotUsedForSync": [1, 9],
```

```
        "gColdStartAttempts": [5, 11],
```

```
        "pAllowPassiveToActive": [5, 16],
```

```
        "pWakeupChannel": [1, 21],
```

```
        "pSingleSlotEnabled": [1, 22],
```

```
        "pAllowHaltDueToClock": [1, 23],
```

```
        "pChannelsA": [1, 26],
```

```
        "pChannelsB": [1, 27]
```

```
      }
```

```
    },
```

```
  {
```

```

    "SUCC2": {
        "pdListenTimeout":      [21, 0],
        "gListenNoiseMinusOne": [4, 24]
    }
},
{
    "SUCC3": {
        "gMaxWithoutClockCorrectionPassive": [4, 0],
        "gMaxWithoutClockCorrectionFatal": [4, 4]
    }
},
{
    "NEMC": {
        "gNetworkManagementVectorLength": [4, 0]
    }
},
{
    "PRTC1": {
        "gdTSSTransmitter":      [4, 0],
        "gdCASRxLowMax":         [7, 4],
        "gStrobePointPosition":  [2, 12],
        "gBaudRatePrescaler":     [2, 14],
        "gdWakeupSymbolRxWindow": [9, 16],
        "pWakeupPattern":        [6, 26]
    }
},
{
    "PRTC2": {
        "gdWakeupSymbolRxIdle":  [6, 0],
        "gdWakeupSymbolRxLow":   [6, 8],
        "gdWakeupSymbolTxIdle":  [8, 16],
        "gdWakeupSymbolTxLow":   [6, 24]
    }
},
{
    "MHDC": {
        "gPayloadLengthStatic":  [7, 0],
        "pLatestTx":             [13, 16]
    }
},
{
    "GTUC1": {
        "pMicroPerCycle":        [20, 0]
    }
},
{
    "GTUC2": {
        "gMacroPerCycle":        [14, 0],
        "gSyncNodeMax":          [4, 16]
    }
}

```



```

    }
  },
  {
    "GTUC3": {
      "pMicroInitialOffset[A]": [8, 0],
      "pMicroInitialOffset[B]": [8, 8],
      "pMacroInitialOffset[A]": [7, 16],
      "pMacroInitialOffset[B]": [7, 24]
    }
  },
  {
    "GTUC4": {
      "gNITStart": [14, 0],
      "gOffsetCorrectionStart": [14, 16]
    }
  },
  {
    "GTUC5": {
      "pDelayCompensation[A]": [8, 0],
      "pDelayCompensation[B]": [8, 8],
      "pClusterDriftDamping": [5, 16],
      "pDecodingCorrection": [8, 24]
    }
  },
  {
    "GTUC6": {
      "pdAcceptedStartupRange": [11, 0],
      "pdMaxDrift": [11, 16]
    }
  },
  {
    "GTUC7": {
      "gdStaticSlot": [10, 0],
      "gNumberOfStaticSlots": [10, 16]
    }
  },
  {
    "GTUC8": {
      "gdMinislot": [6, 0],
      "gNumberOfMinislots": [13, 16]
    }
  },
  {
    "GTUC9": {
      "gdActionPointOffset": [6, 0],
      "gdMinislotActionPointOffset": [5, 8],
      "gdDynamicSlotIdlePhase": [2, 16]
    }
  },
},

```

```

{
    "GTUC10": {
        "pOffsetCorrectionOut": [14, 0],
        "pRateCorrectionOut": [11, 16]
    }
},
{
    "GTUC11": {
        "vExternOffsetControl": [2, 0],
        "vExternRateControl": [2, 8],
        "pExternOffsetCorrection": [3, 16],
        "pExternRateCorrection": [3, 24]
    }
}],
"mb_control": [{
    "MRC": {
        "pFirstDynamicBuffer": [8, 0],
        "pFirstFIFOBuffer": [8, 8],
        "pLastConfiguredBuffer": [8, 16]
    }
}],
{
    "FRF": {
        "pChannelFilter": [2, 0],
        "pFrameIDFilter": [11, 2],
        "pCycleCounterFilter": [7, 16],
        "pRejectStaticSegment": [1, 23],
        "pRejectNullFrames": [1, 24]
    }
},
{
    "FRFM": {
        "pMaskFrameIDFilter": [11, 0]
    }
},
{
    "FCL": {
        "pCriticalLevel": [8, 0]
    }
}
}],
"status": {
    "cc_status": [{
        "CCSV": {
            "vPOC!State": [6, 0],
            "vPOC!Freeze": [1, 6],
            "vPOC!CHIHaltRequest": [1, 7],
            "vPOC!SlotMode": [2, 8],
            "vPOC!ColdstartNoise": [1, 12],

```

```

        "vColdstartAbort":      [1, 13],
        "vColdStartInhibit":    [1, 14],
        "vPOC!WakeupStatus":    [3, 16],
        "*vRemainingColdstartAttempts": [5, 19]
    }
},
{
    "CCEV": {
        "*vClockCorrectionFailed": [4, 0],
        "vPOC!ErrorMode":          [2, 6],
        "*vAllowPassiveToActive":  [5, 8]
    }
},
{
    "SCV": {
        "*vSlotCounter[A]":        [11, 0],
        "*vSlotCounter[B]":        [11, 16]
    }
},
{
    "MTCCV": {
        "*vMacrotick":              [14, 0],
        "*vCycleCounter":           [6, 16]
    }
},
{
    "RCV": {
        "*vRateCorrection":         [12, 0]
    }
},
{
    "OCV": {
        "*vOffsetCorrection":       [19, 0]
    }
}],

"mb_status": [{
    "MBS": {
        "vSS!ValidFrameA":         [1,0],
        "vSS!ValidFrameB":         [1,1],
        "vSS!SyntaxErrorA":        [1,2],
        "vSS!SyntaxErrorB":        [1,3],
        "vSS!ContentErrorA":       [1,4],
        "vSS!ContentErrorB":       [1,5],
        "vSS!BViolationA":         [1,6],
        "vSS!BViolationB":         [1,7],
        "vSS!TxConflictA":         [1,8],
        "vSS!TxConflictB":         [1,9],
        "vSS!EmptySlotA":          [1,10],
    }
}

```

```

        "vSS!EmptySlotB":      [1,11],
        "vSS!MsgLost":         [1,12],
        "vSS!FrameTransmittedA": [1,14],
        "vSS!FrameTransmittedB": [1,15],
        "*vRF!Header!CycleCount": [6,16],
        "vSS!Channel":         [1,24],
        "vRF!Header!SuFIndicator": [1,25],
        "vRF!Header!SyFIndicator": [1,26],
        "vRF!Header!NFIndicator": [1,27],
        "vRF!Header!PPIndicator": [1,28]
    }
}
},
"input_buffer": {
    "header": [{
        "WRHS1": {
            "vRF!Header!FrameID": [11, 0],
            "vRF!Header!CycleCode": [7, 16],
            "vRF!Header!ChannelA": [1, 24],
            "vRF!Header!ChannelB": [1, 25],
            "vRF!Header!Direction": [1, 26],
            "vRF!Header!PPIndicator": [1, 27],
            "vRF!Header!TxMode": [1, 28],
            "vRF!Header!BufferInterrupt": [1, 29]
        }
    },
    {
        "WRHS2": {
            "vRF!Header!HeaderCRC": [11, 0],
            "vRF!Header!Length": [7, 16]
        }
    },
    {
        "WRHS3": {
            "vRF!Header!DataPointer": [11, 0]
        }
    }
    ]},
    "command_mask": {
        "IBCM": {
            "vRF!CommandMask!LoadHeaderSection": [1, 0],
            "vRF!CommandMask!LoadDataSection": [1, 1],
            "vRF!CommandMask!TxRequest": [1, 2]
        }
    },
    "command_request": {
        "IBCR": {
            "*": [7, 0]
        }
    },
},

```

```

"data": ["WRDS1", "WRDS2", "WRDS3", "WRDS4", "WRDS5", "WRDS6", "WRDS7",
        "WRDS8", "WRDS9", "WRDS10", "WRDS11", "WRDS12", "WRDS13",
        "WRDS14", "WRDS15", "WRDS16", "WRDS17", "WRDS18", "WRDS19",
        "WRDS20", "WRDS21", "WRDS22", "WRDS23", "WRDS24", "WRDS25",
        "WRDS26", "WRDS27", "WRDS28", "WRDS29", "WRDS30", "WRDS31",
        "WRDS32", "WRDS33", "WRDS34", "WRDS35", "WRDS36", "WRDS37",
        "WRDS38", "WRDS39", "WRDS40", "WRDS41", "WRDS42", "WRDS43",
        "WRDS44", "WRDS45", "WRDS46", "WRDS47", "WRDS48", "WRDS49",
        "WRDS50", "WRDS51", "WRDS52", "WRDS53", "WRDS54", "WRDS55",
        "WRDS56", "WRDS57", "WRDS58", "WRDS59", "WRDS60", "WRDS61",
        "WRDS62", "WRDS63", "WRDS64"
],
"busy_control": {
    "IBCR": {
        "*": [1, 15]
    }
}
},
"output_buffer": {
    "header": [{
        "RDHS1": {
            "vRF!Header!FrameID": [11, 0],
            "vRF!Header!CycleCode": [7, 16],
            "vRF!Header!ChannelA": [1, 24],
            "vRF!Header!ChannelB": [1, 25],
            "vRF!Header!Direction": [1, 26],
            "vRF!Header!PPIndicator": [1, 27],
            "vRF!Header!TxMode": [1, 28],
            "vRF!Header!BufferInterrupt": [1, 29]
        }
    }
},
{
    "RDHS2": {
        "vRF!Header!HeaderCRC": [11, 0],
        "vRF!Header!Length": [7, 16],
        "vRF!Header!LengthRcvd": [7, 24]
    }
},
{
    "RDHS3": {
        "vRF!Header!DataPointer": [11, 0],
        "vRF!Header!CycleCount": [6, 16],
        "vSS!Channel": [1, 24],
        "vRF!Header!SuFIndicator": [1, 25],
        "vRF!Header!SyFIndicator": [1, 26],
        "vRF!Header!NFIndicator": [1, 27],
        "vRF!Header!PPIndicator": [1, 28]
    }
}
}],

```

```

"command_mask": {
  "OBCM": {
    "vRF!CommandMask!LoadHeaderSection": [1, 0],
    "vRF!CommandMask!LoadDataSection": [1, 1]
  }
},
"command_request": {
  "OBCR": {
    "vRF!CommandRequest!BufferRequest": [7, 0],
    "vRF!CommandRequest!View": [1, 8],
    "vRF!CommandRequest!MessageRAMTransfer": [1, 9]
  }
},
"data": ["RDDS1", "RDDS2", "RDDS3", "RDDS4", "RDDS5", "RDDS6", "RDDS7",
  "RDDS8", "RDDS9", "RDDS10", "RDDS11", "RDDS12", "RDDS13", "RDDS14",
  "RDDS15", "RDDS16", "RDDS17", "RDDS18", "RDDS19", "RDDS20",
  "RDDS21", "RDDS22", "RDDS23", "RDDS24", "RDDS25", "RDDS26",
  "RDDS27", "RDDS28", "RDDS29", "RDDS30", "RDDS31", "RDDS32",
  "RDDS33", "RDDS34", "RDDS35", "RDDS36", "RDDS37", "RDDS38",
  "RDDS39", "RDDS40", "RDDS41", "RDDS42", "RDDS43", "RDDS44",
  "RDDS45", "RDDS46", "RDDS47", "RDDS48", "RDDS49", "RDDS50",
  "RDDS51", "RDDS52", "RDDS53", "RDDS54", "RDDS55", "RDDS56",
  "RDDS57", "RDDS58", "RDDS59", "RDDS60", "RDDS61", "RDDS62",
  "RDDS63", "RDDS64"
],
"busy_control": {
  "OBCR": {
    "*": [1, 15]
  }
}
},
"new_data": ["NDAT1", "NDAT2", "NDAT3", "NDAT4"],
"mb_status_changed": ["MBSC1", "MBSC2", "MBSC3", "MBSC4"]
}

```

Application User Manual

Graphical User Interface of the application is shown in Figure 1. It is separated in different parts depending on their functionality and purpose. Each of these parts is discussed separately.

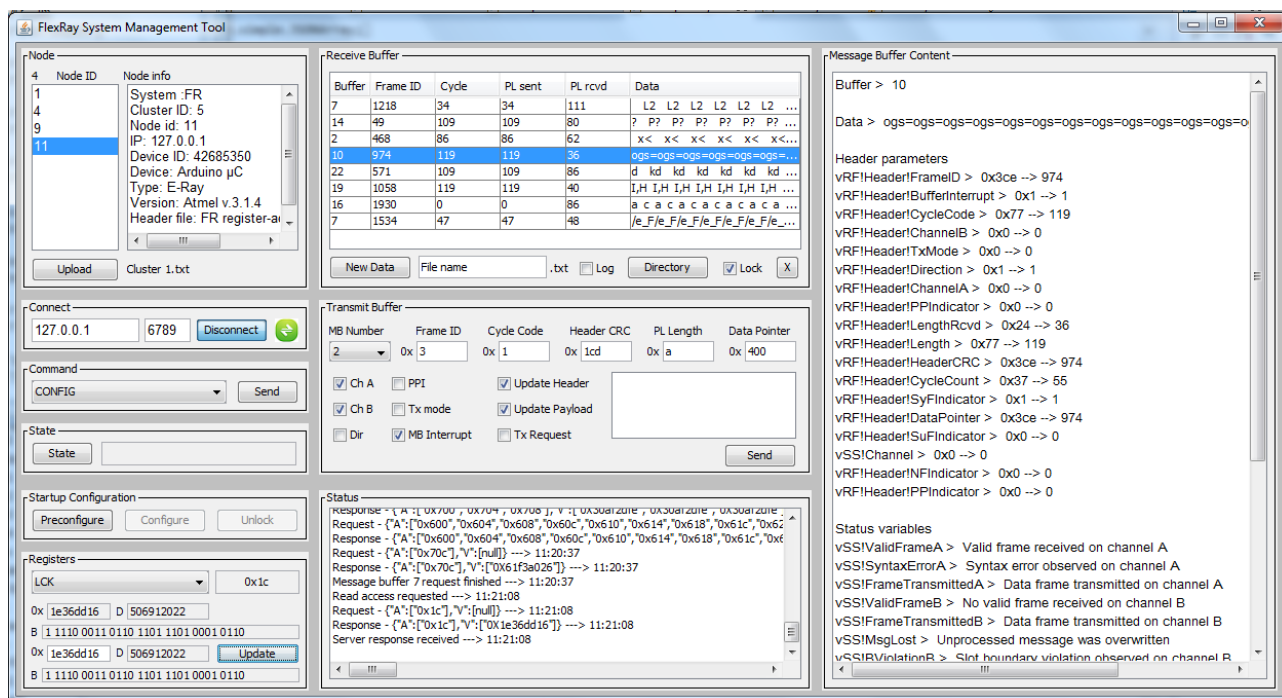


Figure 1. Graphical User Interface – full view

Node

The user uploads the Cluster config file and the Node ID list is populated with all node ids that are part of this cluster. User then selects the desired id and in Node info field node details are displayed. The name of the file is also displayed for reference and its file location on the computer is available as a tooltip text.

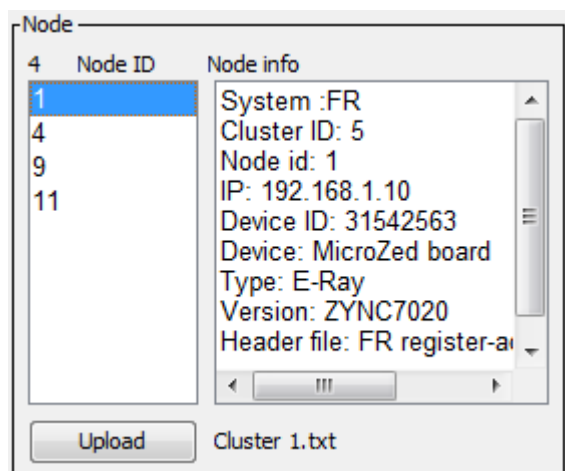


Figure 2. Node part

Connect

In Connect part the node IP address is displayed. It is taken automatically from the Node config file and it shall not be changed from the GUI. The port number is taken from the Cluster config file where a default port number is assigned but it can be changed dynamically by the user as for different machines there are different COM ports available. If the connection was successful a green icon is displayed.

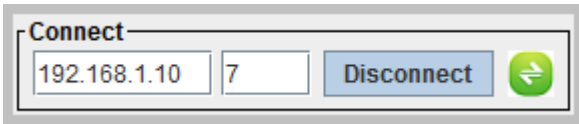


Figure 3. Connect part

Command and State

The user can send a FlexRay command from a predefined list of commands. After applying the selected command, current node state (potentially changed by the applied command) is automatically requested and displayed. The user can also request only the current node state.

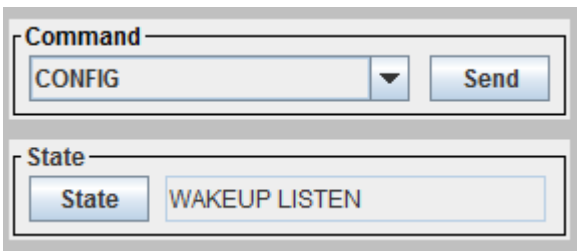


Figure 4. Command and State parts

Registers

All registers and their corresponding addresses contained in the node Header file are present in a GUI list. The user can request the current register value and can also update it.

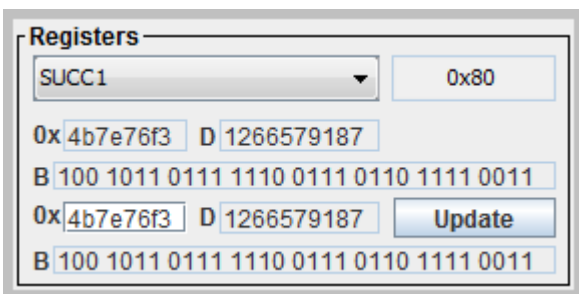


Figure 5. Registers part

Transmit Buffer

All configured message buffers are present in a list. When user selects one buffer, its configuration data is displayed. The user can then send a message or attempt to reconfigure the message buffer during runtime, if reconfiguration is enabled. The text field dedicated to the actual message has a maximum length of 255 symbols (64 data registers, 4 bytes each). Message Buffer 0 is reserved.

Figure 6. Transmit Buffer part

Receive Buffer

Once the user presses the New Data button, the new data that was received by the node is requested and displayed in a loop. The most relevant message buffer parameters are displayed in a table in their receive order (Figure 7). The user can select a row to see all the parameters and status variables related with this buffer. They are displayed in a text field (Figure 8).

Another possibility that the GUI offers is to save all that information in a text file. For this purpose the user has to write the file name and check the Log checkbox. The destination folder of that file is also a click away. The 'X' button clears the table but does not affect the log file.

Buffer	Frame ID	Cycle	PL sent	PL rcvd	Data
0	1570	6	6	96	UVKtUVKtUVKtUVKtUVKt...
2	44	121	121	70	*:r=:r=:r=:r=:r=:r=
3	1572	23	23	123	s s s s s s s ...
7	624	104	104	96	2xi!2xi!2xi!2xi!2xi!2xi!2xi...
13	1565	118	118	55	4 CW4 CW4 CW4 CW4...
15	1495	59	59	122	n h9n h9n h9n h9n h9...
25	2046	91	91	99	h ^ h ^ h ^ h ^ h ^ h ^ h ^...
26	126	10	10	107	: J : J : J : J : J : J : J : J...
32	650	61	61	53	'Kg `Kg `Kg `Kg `Kg `K...

Figure 7. Receive Buffer part

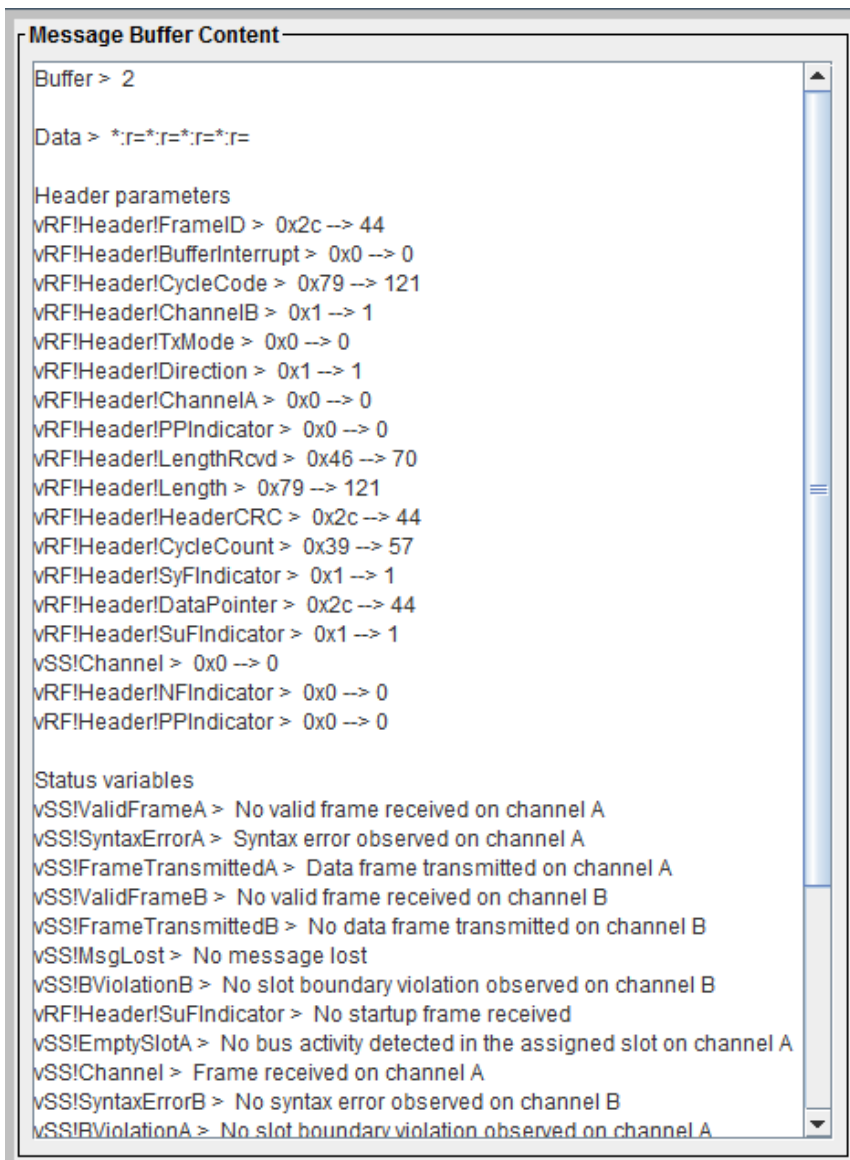


Figure 8. Message Buffer Content part

Status

In Status text field are shown the JSON objects that are requested to and received from the server, as well as the errors that have occurred during operation. When an error occurs the text is coloured in red.

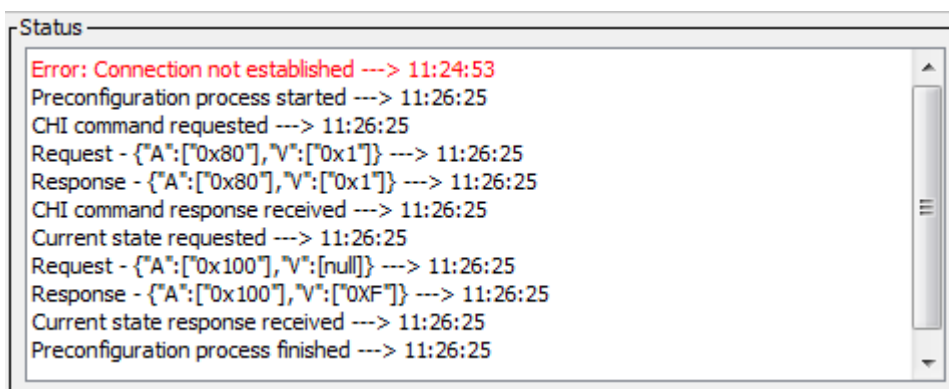


Figure 9. Status part