



CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering
Department of Control Engineering

Vehicle test platform

Master's thesis

Study program: Open informatics
Major: Computer engineering
Supervisor: Doc. Ing. Jiří Novák, Ph.D.

Martin Zeman

Prague 2014

I confirm that I've completed this thesis myself and have only used the sources (including literature, projects and software) listed in the corresponding section.

In Prague 30.12. 2014

signature

Abstrakt

Tato práce si klade za cíl návrh struktury a implementaci programového vybavení pro testovací platformu vozidlových komunikačních standardů. Náplň se skládá z vývoje obslužné aplikace pro platformu PC Windows. Další součástí je firmware řídicího mikrokontroléru samotné platformy postavený nad operačním systémem reálného času. K propojení obou částí je zapotřebí navrhnout komunikační protokol pro jejich interakci. Systém musí být schopen vzdálené rekonfigurace hradlového pole, jež je součástí testovací platformy a obsluhy radičů standardu FlexRay v něm implementované.

Abstract

This thesis aims to design the structure of and to implement the software component of a test platform for vehicular communication standards. The task breaks down into two main areas. The first entails the development of a PC platform-based application responsible for coordinating all platform functionality. The second part involves the development of firmware on the platform side. The system is required to run a real-time operating system. In order to enable interaction between the parts a communication protocol needs to be developed. The system must be capable of remote reconfiguration of the design in the FPGA, which is a part of the platform.

Acknowledgements

First and foremost a sincere thank you goes to Doc. Ing. Jiří Novák, Ph.D. I appreciate his patience and time devoted to helping me succeed. I'm equally grateful for his tolerance for broken hardware. I'd also like to thank my friend and former classmate Ing. Jiří Blecha, the author of the platform's hardware, for always taking the time to explain the intricacies of his design. Last but not least, I express my gratitude to my family who have supported me during my studies and continue to do so.

Table of contents

Abstrakt.....	2
Abstract.....	2
Acknowledgements.....	3
Table of contents.....	1
List of figures and tables.....	4
1. Introduction.....	1
1.1. Analysis of the assignment.....	1
1.2. Analysis of the solution.....	2
2. FlexRay.....	3
2.1. Brief description.....	3
2.2. Physical layer.....	4
2.3. Link layer.....	8
2.3.1. Architecture of a node.....	9
2.4. Communication cycle.....	10
2.4.1. Microtick – μT	11
2.4.2. Macrotick – MT.....	11
2.4.3. Static segment.....	11
2.4.4. Dynamic segment.....	12
2.5. Frame format.....	13
2.6. Clock synchronization.....	14
2.6.1. Measurement.....	14
2.6.2. FTM algorithm.....	14
2.6.3. Rate Correction calculation.....	15
2.6.4. Offset correction calculation.....	16
2.7. Startup mechanism.....	16
3. System architecture.....	18

4.	FreeRTOS	19
4.1.	Port settings	19
4.2.	Features used	20
4.2.1.	Tasks	20
4.2.2.	Queues.....	21
4.2.3.	Mutexes, semaphores and binary semaphores	21
5.	Firmware	22
5.1.	Porting of the lwIP stack	22
5.2.	Command processing	22
5.3.	Task declaration macros	24
5.4.	TCP server implementation	25
5.5.	EMIF.....	26
5.6.	MCU state machines.....	27
5.6.1.	MCU Connection state machine	27
5.6.2.	MCU FR state machine	29
5.6.3.	FPGA FR state machine.....	32
6.	PC Application.....	34
6.1.	State machines	34
6.1.1.	General state machine	34
6.1.2.	FPGA State Machine.....	37
6.2.	Features.....	38
6.2.1.	Monitoring and frame transmission	38
6.2.2.	Remote reconfiguration of the FPGA	42
6.2.3.	Remote task control.....	42
6.2.4.	Fibex parsing.....	43
6.2.5.	FPGA FlexRay controller and Testing.....	46
6.2.6.	Target I.P. setting	47
6.2.7.	ECU Mapping	47

Table of contents

6.2.8. Saving and loading of Cluster and MCU parameters.....	48
6.2.9. Other useful features	49
6.3. TCP client implementation.....	51
6.4. Database.....	51
7. Communication protocol	55
7.1. Purpose of the protocol.....	55
7.2. Requirements	55
7.3. Negotiation of supported functions	56
7.4. Message format.....	56
7.4.1. PC to MCU.....	57
7.4.2. MCU to PC.....	64
8. Conclusion	66
References.....	67

List of figures and tables

List of Figures

Figure 2-1: FlexRay transceiver with two channels	4
Figure 2-2: Levels of FlexRay's electrical signals	5
Figure 2-3: A Bus with two channels	6
Figure 2-4: Dual channel single star configuration	6
Figure 2-5: Single channel cascaded star configuration	6
Figure 2-6: Dual channel cascaded star configuration.....	7
Figure 2-7: Single channel hybrid example	7
Figure 2-8: Dual channel hybrid example	8
Figure 2-9: Physical layer and link sub-layers.....	9
Figure 2-10: Architecture of a FlexRay node	10
Figure 2-11: Communication cycle	11
Figure 2-12: FlexRay frame format	13
Figure 2-13: An example of FTM calculation for $k = 2$	15
Figure 3-1: System architecture	18
Figure 5-1: Command dispatching	23
Figure 5-2: EMIF Settings	27
Figure 5-3: MCU Connection state machine	28
Figure 5-4: State machine for handling the MCU's FlexRay controller	29
Figure 5-5: Details of message exchange at the end of data definition	31
Figure 5-6: MCU FPGA State machine.....	32

Figure 6-1: PC application's main state machine.....	35
Figure 6-2: PC FPGA State machine.....	37
Figure 6-3: How to select a frame to be monitored	38
Figure 6-4: Difference between FPGA and MCU frames	39
Figure 6-5: Message Editor tab example	40
Figure 6-6: Monitoring example.....	41
Figure 6-7: Remote configuration.....	42
Figure 6-8: Task manager window	43
Figure 6-9: UML diagram of a FlexRay fibex file	44
Figure 6-10: Cluster Setup after loading a fibex file	45
Figure 6-11: Cycle settings in the MCU.....	46
Figure 6-12: FGPA Status window.....	47
Figure 6-13: IP Address settings window.....	47
Figure 6-14: ECU Mapping example.....	48
Figure 6-15: Load and Save menu options	49
Figure 6-16: Absolute vs relative time	50
Figure 6-17: Monitoring record from Figure 6-6 saved as CSV file and displayed in MS Excel.....	50
Figure 7-1: MCU slot definition message format - first word.....	59
Figure 7-2: MCU slot definition message format - second word	60
Figure 7-3: MCU Tx Data Definition Message Format	60
Figure 7-4: MCU Tx Data Update Message Format	60
Figure 7-5: FPGA TX frame data definition message format - first word	61

List of figures and tables

Figure 7-6: FPGA TX frame data definition message format - Timestamp - second and third word	62
Figure 7-7: FPGA TX frame data definition message format - Macrotick - second word	62
Figure 7-8: FPGA TX frame data definition message format - Macrotick and Cycle - second word.....	62
Figure 7-9: FPGA TX frame data update message format	62
Figure 7-10: Send FPGA design message format - bytes 0 to 3.....	63
Figure 7-11: Send FPGA design message format - bytes 4 to 5.....	63
Figure 7-12: Send FPGA design data message format	63
Figure 7-13: Supported Commands and Version Response message format.....	65
Figure 7-14: FlexRay Data Message Format	65

List of Tables

Table 2-1: An example of measured values.....	14
Table 2-2: Number of entries to eliminate.....	15
Table 6-1: Contents of the ECUs table	52
Table 6-2: Contents of the FPGAs table	53
Table 6-3: Contents of the Frames table	54
Table 6-4: Contents of the Signals table	54
Table 6-5: Contents of the Triggers table	55
Table 7-1: PC to MCU command table	57
Table 7-2: MCU FlexRay parameters.....	59
Table 7-3: FPGA FlexRay parameters.....	61

List of figures and tables

Table 7-4: Trigger Type values	62
Table 7-5: MCU to PC response table	64
The three bytes in Figure 7-13 are followed by a number of bytes defined in the "Number of commands" section. Each byte contains a supported command code from Table 7-1.....	64

1. Introduction

1.1. Analysis of the assignment

The aim of this thesis is to integrate various projects from a number of authors into a functional and flexible vehicular testing platform. The integration consists of a development of a PC-Windows-based application responsible for providing an interface between the testing platform and the user. The application must be capable of controlling the function of the FlexRay controller present in the platform's MCU such as defining the outgoing communication and monitoring of the incoming frames.

Another important feature is the remote reconfiguration of the design in the FPGA. The application has to be able to send an arbitrary design file from the PC's file system to be loaded into the FPGA or be capable of loading the default design present in the MCU's flash memory.

Next, the application needs to provide the functionality to remotely run and suspend user-defined tasks in the MCU. Said tasks are to be defined at compile time as part of the MCU's firmware.

The application is also responsible for the management of the FlexRay controllers implemented in the FPGA. These controllers provide some unique features for the testing of the parameters of the FlexRay networks. The application's role is to trigger these tests and present the user with results. Specifically, the application needs to provide an interface in which to configure and manage all the controllers present in the FPGA separately.

On the hardware side the MCU is required to run a real-time operating system to provide the programmer with methods of task synchronization and resource protection. The use of a real-time operating system also offers a higher flexibility in terms of task separation, better control over timing requirements and an overall richer selection of tools for building non-trivial systems.

The MCU's firmware acts as the execution centre for all the system's functionality. It needs to receive and decode requests from the PC application and carry out corresponding actions. In order to facilitate these functions, it is necessary to develop a protocol for communication between the PC application and the MCU.

1.2. Analysis of the solution

From a hardware standpoint the platform provides a TMS570LS3137ZWT microcontroller from Texas Instruments. This chip has been chosen due to previous experience with it in which it has proven to be both powerful and cost-efficient. The chip provides a wide variety of interfaces and modules, as well as a flash memory of sufficient capacity (3072 KB) for the purposes of the platform.

Another significant advantage of this microcontroller is that Texas Instruments provide developers with a great tool called HalcoGen to generate code for its configuration, initiation and libraries for all its peripherals, with the exception of the FlexRay interface. This tool is also capable of generating a port of the freeRTOS real-time operating system specifically for the TMS570 MCU family.

This makes the choice of the real-time operating systems simple. Our requirements for the OS are:

- Real-time capability
- Must be lightweight
- Must be able to provide methods of task-synchronization and resource protection
- Must be able to provide an interface to report defined tasks and run/suspend them flexibly
- Must be free
- Must be open-sourced with a reasonable license
- Needs to be well-documented

FreeRTOS meets all these requirements and therefore has been selected as the OS for the platform. Detailed settings of the used port are described in section 4.1.

The firmware can take advantage of a FlexRay library written for the E-Ray controller by the author of this thesis for a previous project (source [11]). It provides a basic API which simplifies the handling of the controller's functions.

A decision has been made to use the TCP/IP protocol as a basis for building the communication protocol between the PC application and the MCU. The main reason for this decision over the USB, which is in reality the only other candidate, is previous experience with the lwIP stack. Another factor is TCP's simplicity compared to USB. Its advantage

compared to UDP lies in its reliability while maintaining high enough throughputs. This decision requires the lwIP stack to be ported for the combination of freeRTOS with TMS570.

A couple of possible ways to implement the communication protocol between the PC application and the MCU were considered. The first option was to opt for a RPC-type protocol. This category broke down further into binary-based or human-readable protocols, Bert and Apache Thrift are examples of the former and JSON-RPC or SOAP of the latter. The alternative to RPC is a custom-made binary protocol. Considering the rather small computational frequency of the MCU's core (160 MHz) and the number of tasks it has to perform, the XML or JSON-based options needed to be dismissed due to the difficulty of their parsing. A custom binary protocol has been selected since it offers the best performance while maintaining simplicity.

The C# programming language with its .NET framework has proven to be a powerful, flexible and versatile choice for our projects in the past. It offers an easy-to-use networking and database API together with a vast variety of GUI components to build an application that is both user-friendly and visually pleasing. Therefore, it has been chosen for this project as well.

The FlexRay protocol defines a large number of parameters, which are necessary for a network to function. The Fieldbus Exchange Format (FIBEX) is a network description standard defined by the Association for Standardization of Automation and Measuring Systems (ASAM) which encompasses all common automotive communication standards including FlexRay. A FIBEX file contains all the information needed to describe an entire on-board network and has become the standard input file format for commercial software. For this reason, a decision has been made to use FIBEX as the input format for the PC application.

2. FlexRay

2.1. Brief description

The FlexRay standard is a communication protocol released by the FlexRay Consortium in the year 1999. Members of the consortium are leading companies of the automotive industry such as GM, Bosch, BMW, Motorola, Volkswagen, Freescale, Daimler Chrysler and others. The standard has been developed primarily for the automotive industry.

Its intended field of application lies chiefly in safety-critical applications for instance steer-by-wire or brake-by-wire. In contrast to standards like CAN, TTCAN or LIN it offers higher bandwidth up to 10 Mbit/s. It is based on the time-division multiplexing (TDMA) principle the determinism of which is crucial for real-time applications. This also presents its main advantages over aforementioned standards which use the master/slave (LIN) or CSMA/CR (CAN) methods of arbitration.

However, FlexRay combines both deterministic and stochastic approaches to communication which makes it flexible. The standard only defines the physical and the link layers as defined by the ISO/OSI reference model.

2.2. Physical layer

The physical layer is represented by a transceiver and an unshielded twisted-pair cabling. The standard also offers the possibility to use optical fibers with optical transceivers. FlexRay supports the usage of two separate channels (commonly referred to as A and B). Those can act as completely independent media or can provide redundancy to achieve better reliability. However, data consistency on redundant channels isn't secured by the controller intrinsically and thus has to be implemented by the host.

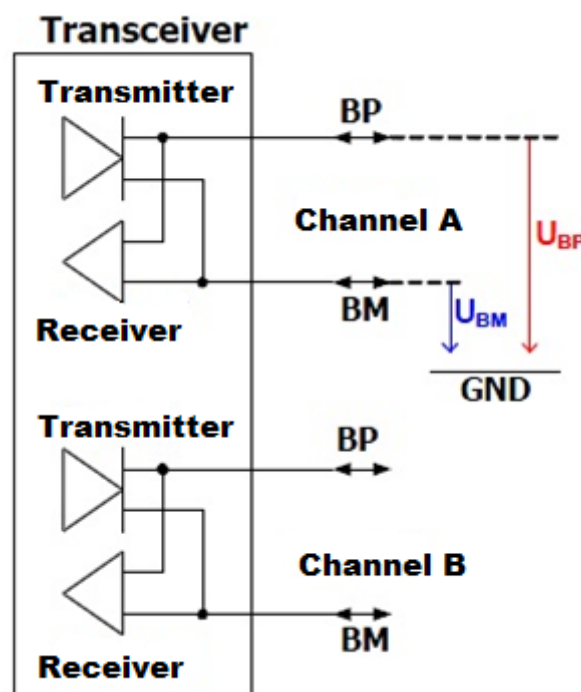


Figure 2-1: FlexRay transceiver with two channels

Source: [1]

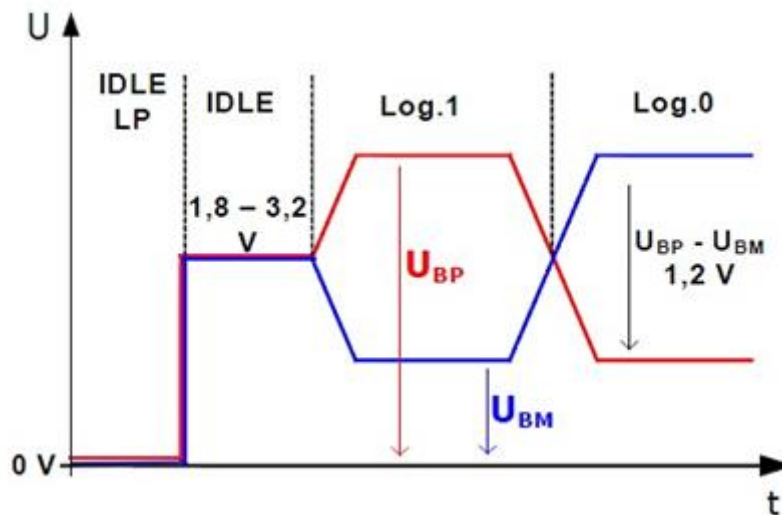


Figure 2-2: Levels of FlexRay's electrical signals

Source: [1]

Furthermore, FlexRay introduces the bus guardian. It is an optional part of the physical layer. The bus guardian is an element responsible for the protection of the channel from interference caused by communication that is not in compliance with the cluster's communication schedule. It is capable of blocking outgoing communication in time slots that are not assigned to its hosting node. This prevents a potential break down of communication between all the nodes of a cluster.

There are several approaches to designing a FlexRay cluster depending on its topology. In addition to pure topologies like a bus or a star, FlexRay also supports their hybrid variants which are a combination of the two. The number of channels used and their configuration presents another point of decision. There's a plethora of possibilities. The thing to keep in mind is the maximum length of a segment between two nodes which is 24 meters. These are some examples of the possible topologies:

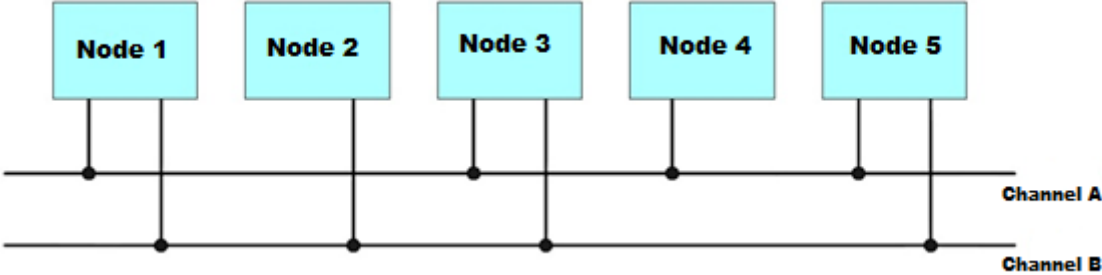


Figure 2-3: A Bus with two channels

Source: [2]

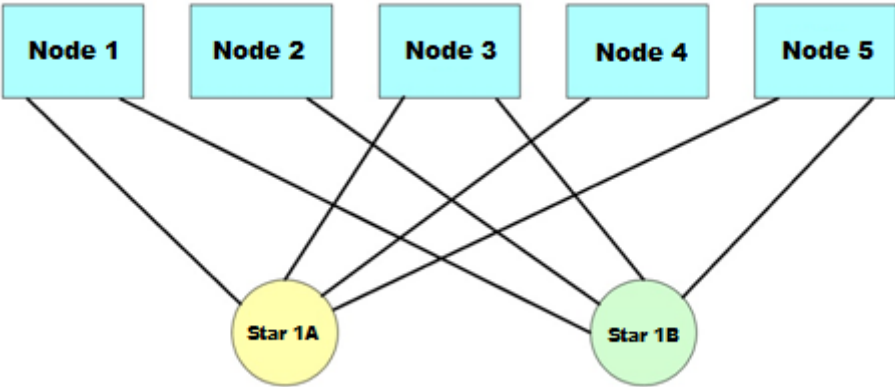


Figure 2-4: Dual channel single star configuration

Source: [2]

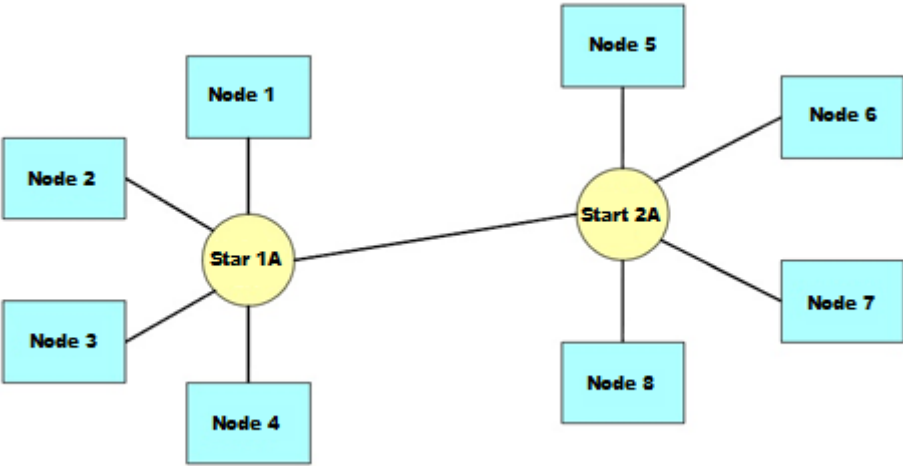


Figure 2-5: Single channel cascaded star configuration

Source: [2]

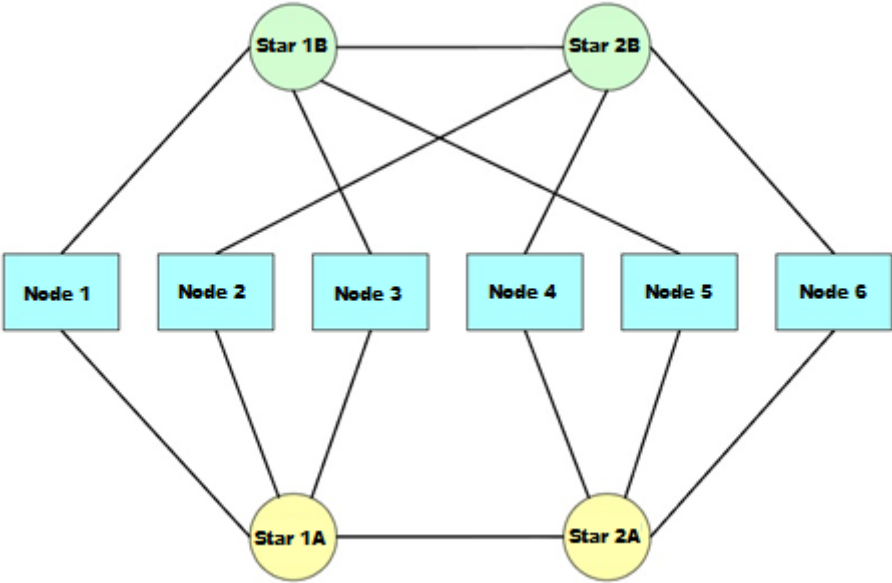


Figure 2-6: Dual channel cascaded star configuration

Source: [2]

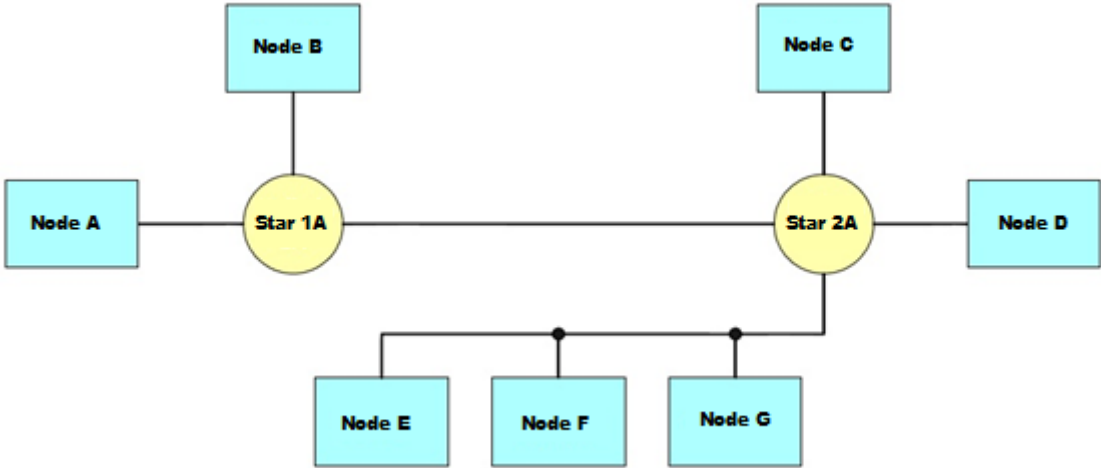


Figure 2-7: Single channel hybrid example

Source: [2]

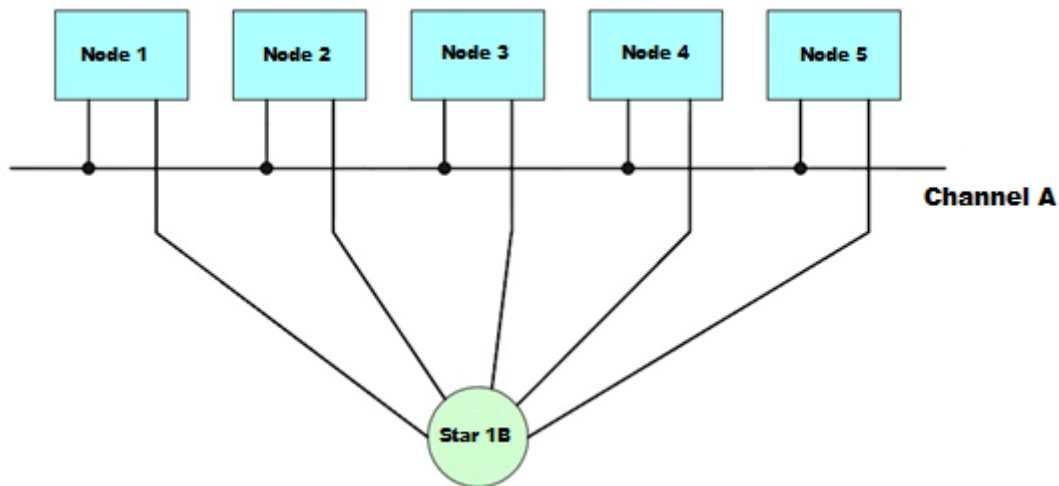


Figure 2-8: Dual channel hybrid example

Source: [2]

From a reliability standpoint it is no doubt preferable to use both channels as redundant media. The bus topology is the simplest option and also the cheapest making it suitable for simple applications. In this case the bus transmission line represents a weak link. In case of its severance the whole network goes out of commission due to faulty line termination.

The active star topology enables us to reduce the consequences of a failure which occurs in a part of the network. Failure of the active star element is nevertheless still a potential risk. Hybrid topologies combine the advantages of the bus and the active star. Their weaknesses can be partially made up for through a correct combination of both topologies (example Figure 2-8).

2.3. Link layer

This layer of the ISO/OSI model is responsible for is the equivalent of a communication controller and can be sub-divided into three sub-layers. These layers represent the core of the standard itself and provide an interface to layers below and above. These include:

- Coding/decoding layer - responsible for modifying the data and physical coding of the transmitted bits.
- Protocol execution layer - implements the core of the protocol, puts data into frames, controls the media.
- Controller host interface layer - interface between the node and the host.

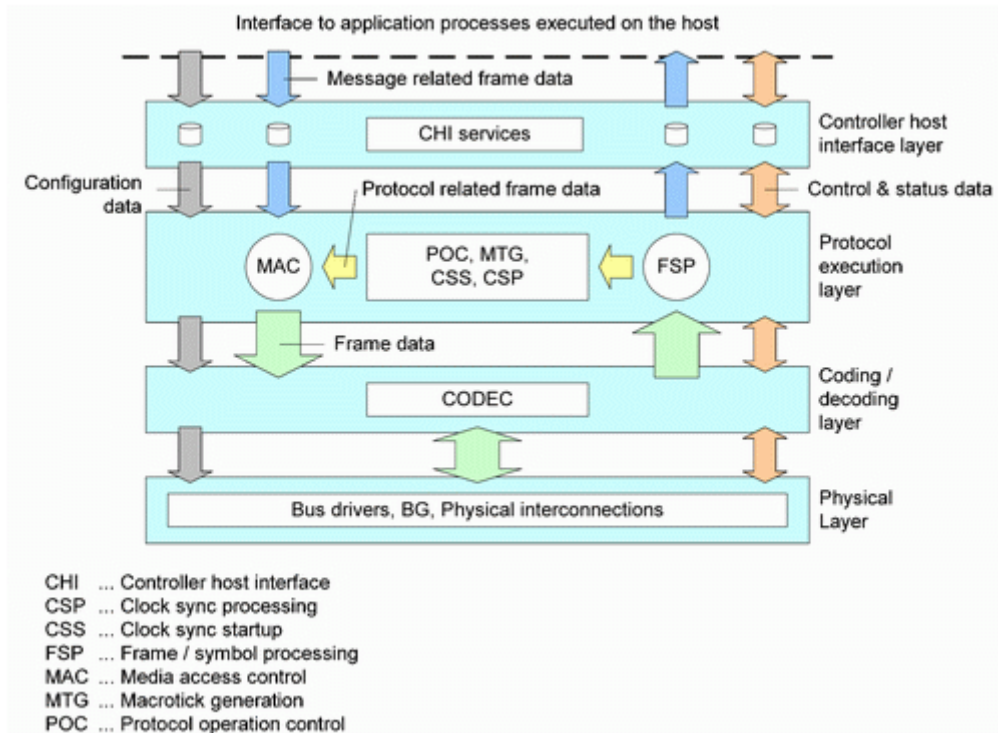


Figure 2-9: Physical layer and link sub-layers

Source web: <http://automatizace.hw.cz/sbernice-komunikace-flexray-nejen-pro-automobily>

2.3.1. Architecture of a node

- Host
 - > Contains the node's firmware
 - > Sets the parameters of the communication controller
 - > Enables/disables the usage of the bus guardian (if it is physically present)
- Communication controller
 - > Implements the protocol's core
 - > Provides an interface to the host
 - > Generates interrupts
 - > Generates the local time base - macro-tick (refer to section 2.4.1)
 - > Synchronizes the local time base with the global time base
 - > Controls access to media
- Bus driver
 - > Drives and receives various bus signals
 - > Detects and reports error states

- > Provides support for a remote node wakeup triggered by communication on the bus
- > Two independent channels (A and B)
- Bus Guardian
 - > Provides protection against unauthorized access to the bus
 - > Optional

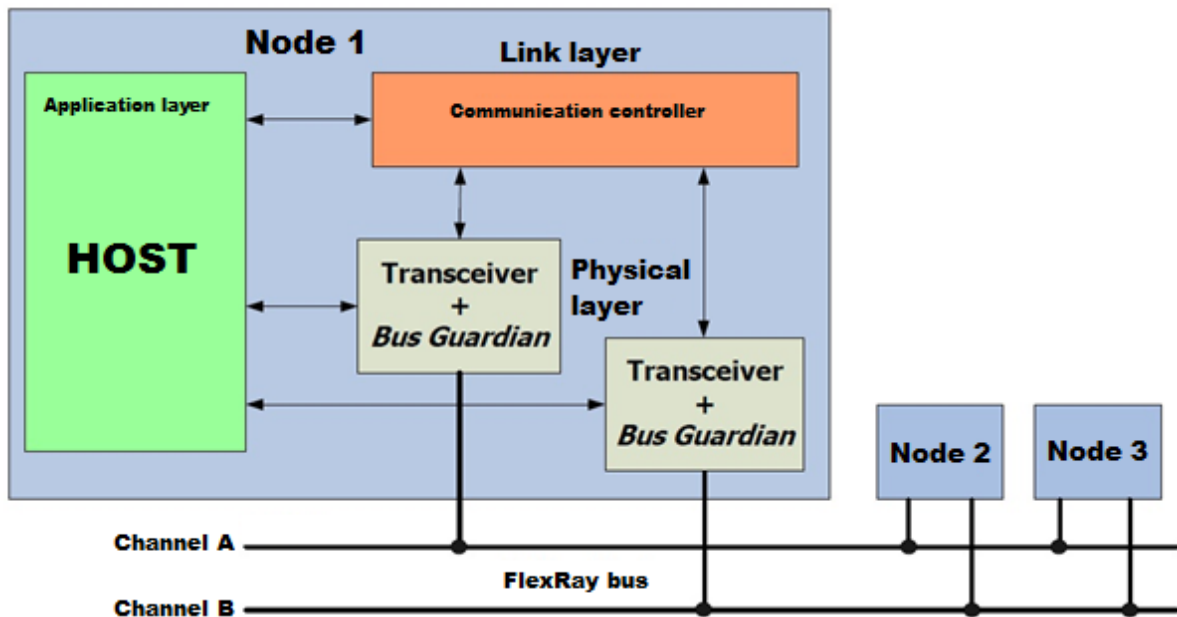


Figure 2-10: Architecture of a FlexRay node

Source: [1]

2.4. Communication cycle

The FlexRay protocol divides time into so called communication cycles. The length of a communication cycle is parameter which needs to be determined by the network-designer. This length is constant during run-time. One communication cycle then breaks down into four different segments. These are the static segment, the dynamic segment, the symbol window and the network idle time.

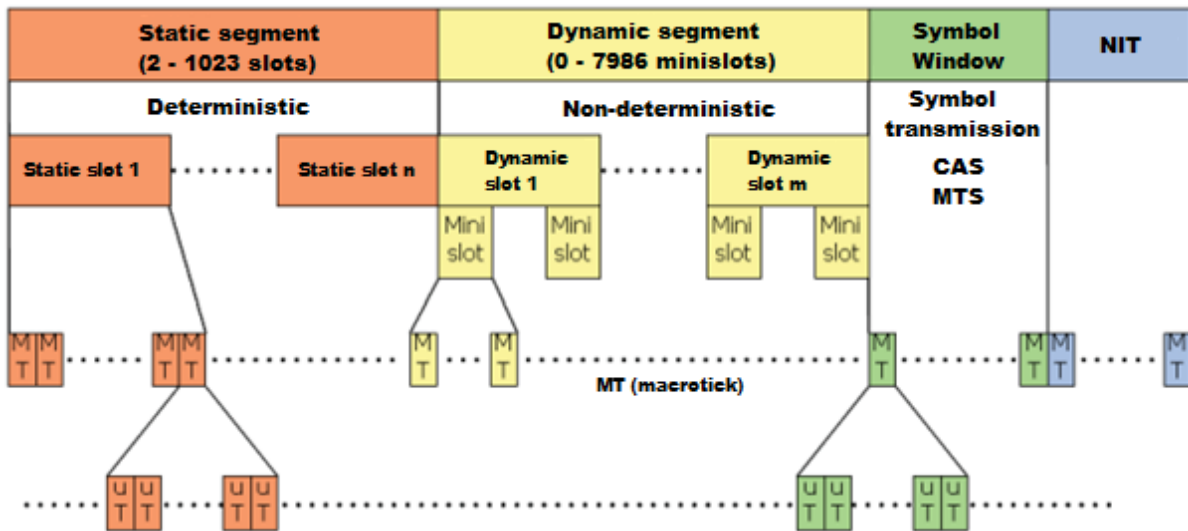


Figure 2-11: Communication cycle

Source: [1]

Only the static segment and the network idle time are compulsory parts of the communication cycle. The length of the communication cycle and its division into segments must be identical in all the nodes of the cluster.

2.4.1. Microtick – μT

The microtick represents the smallest and atomic time interval. It is derived from the controller's oscillator and therefore node-specific. The usual length is equal to the period of controller's time base. It is not a subject to the global clock synchronization mechanism.

2.4.2. Macrotick – MT

Macrotick is a time interval identical for all the nodes in the cluster. It represents the common perception of time in the FlexRay network. A macrotick consists of an integral number of microticks, formally $MT = \mu T \cdot k; k \in \mathbb{N}$, where the constant k can differ in different nodes of the cluster depending on the frequency of their time bases.

2.4.3. Static segment

The communication cycle always starts with the static segment. It is a compulsory part of the communication cycle. It consists of $n; n \in \mathbb{N}$ number of static slots. The maximum number of static slots is defined by the standard as 1024. Each slot belongs to exactly one node, however, one node can own multiple slots. The beginning and the end of a static slot is

time wise preset and cannot change at run-time according to the payload length transmitted inside the slot.

The static segment represents the deterministic part of the communication cycle and is thus suitable for the exchange of time-critical data. The guaranteed latency, however, comes at the cost of lower utilization of the communication channel.

2.4.4. Dynamic segment

The dynamic segment is an optional part of the communication cycle. It's made up of m ; $m \in \mathbb{N}$ dynamic slots. Each dynamic slot is then further made up of minislots the number of which varies depending on the current frame's payload length. The duration of a minislot must be the same for all nodes in the cluster and it is defined by the number of MT which it consists of. The length of a dynamic slot is therefore not constant and its beginning and end cannot be known at network design time.

The slot counter of a communication controller then, as opposed to the static segment, holds the count for the duration of frame reception or transmission so that all the nodes in the cluster share the same value of the slot counter. The length of the dynamic segment is constant, however, for every communication cycle. It is possible for this reason that a frame assigned to one of the later dynamic slots will not be transmitted and it delayed until the next cycle (the transmission would cause the frame to overstep the dynamic segment boundary). This can happen multiple times in a row.

Cluster's behavior in this respect can be influenced by the parameter $pLatestTx$ as defined by the FlexRay standard v2.1. This parameter sets the time in minislots when a node is allowed to start transmitting at the latest in the dynamic segment. The communication controller checks before the transmission of any frame in the dynamic segment if the minislot counter has exceeded the $pLatestTx$ threshold. If so, then the transmission is suspended until the next communication cycle.

The dynamic segment represents the non-deterministic part of the communication cycle. It is therefore most suitable for the exchange of time-noncritical data. Its advantage lies in the high degree of utilization of communication channel compared to the static segment. For this reason it can reach a much higher throughput.

2.5. Frame format

A FlexRay frame is made up for three segments. First is the header segment which begins by the reserved bit and is followed by a series of indicator bits. Frame ID determines the time slot of the communication cycle in which the frame is being transmitted. It can either be a static slot or a dynamic slot but no other frames are allowed to have the same Frame ID in the same communication cycle. The payload length is indicated by a number of half-words (16 bits). The range is then from 0 to 254 bytes. The header CRC is calculated from the last two indicator bits, Frame ID and payload length. It serves as a means of verification of the transmission's correctness. The cycle count denotes the cycle number. It ranges from 0 to 63. When it reaches its maximum value it starts over from zero again. This is useful mostly for the so called cycle filtering. Nodes can for instance only transmit data every n -th cycle with a possible offset. In the same manner, the protocol supports also filtering of received frames. This practice, however, should not be exploited for sharing slots between nodes despite it being technically feasible.

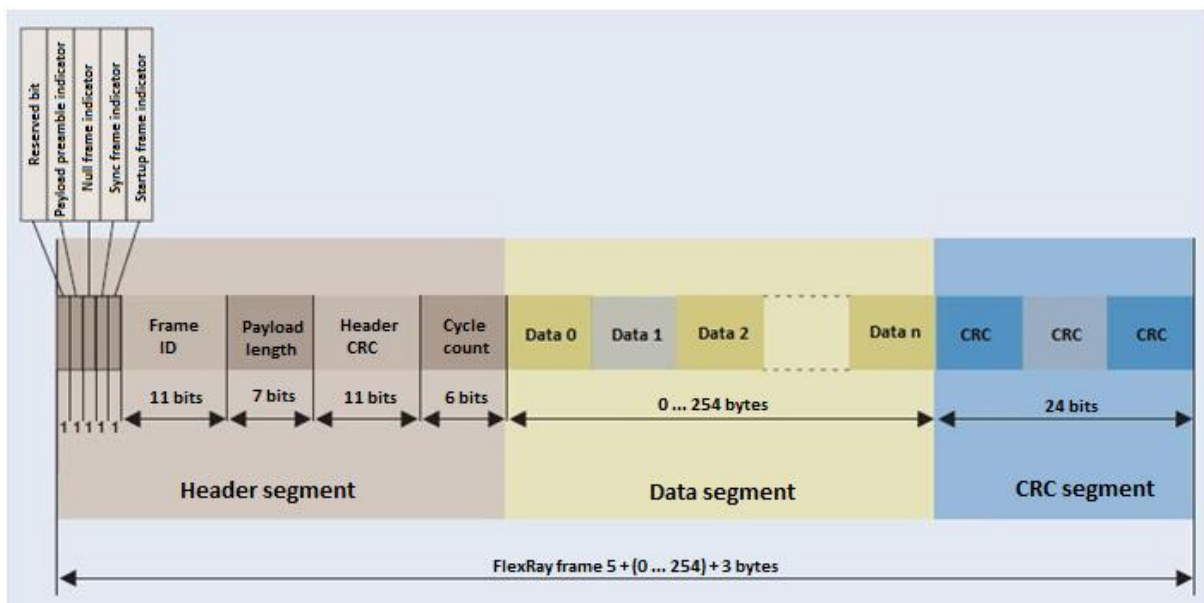


Figure 2-12: FlexRay frame format

Source web: <http://www.coleparmer.com/TechLibraryArticle/1112>

The data segment contains 0 to 254 bytes of payload data. The payload length may vary in frames with the same Frame ID cycle to cycle. Therefore it's necessary to always read the payload length field. The CRC segment contains a value calculated over the entire header and data segments. The presence of two CRCs in a single frame is of the FlexRay's security

features. For details on the generator polynomials and their initialization vectors for the CRC segment and the header CRC refer to the FlexRay standard (source [2]).

2.6. Clock synchronization

In order to enable the use of TDMA all nodes in a FlexRay cluster must have a common perception of time with a fairly high precision regardless of their individual oscillator frequencies. To accomplish this FlexRay introduces the aforementioned global time unit - macrotick. Nevertheless, real oscillators are imperfect and their frequencies fluctuate with time. Therefore, it is necessary that all nodes constantly adjust the lengths of their macroticks (Rate Correction) and the offset of individual cycles (Offset correction).

2.6.1. Measurement

At the beginning of each communication cycle the controller measures the time deviations between the expected reception time and the actual reception time. This is performed for every so called synchronization frame of every synchronization node (a node that transmits a synchronization frame). Measurements are done separately for both channels. The measured values for the last two cycles are stored.

	Even cycle		Odd cycle	
	ΔT of channel A [μT]	ΔT of channel B [μT]	ΔT of channel A [μT]	ΔT of channel B [μT]
Node 1	5	13	25	32
Node 2	11	10	14	13
...
Node n	35	30	41	29

Table 2-1: An example of measured values

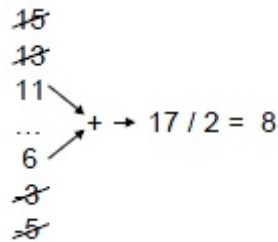
2.6.2. FTM algorithm

The input for the Fault Tolerant Midpoint algorithm is a list of integral values. Those values are first sorted in descending order. Depending on the number of entries we eliminate k highest and lowest values. From the remaining values we chose the highest and the lowest ones. Their arithmetical mean is the output of the FTM algorithm.

Number of entries	k
1-2	0
3-7	1
> 7	2

Table 2-2: Number of entries to eliminate

Source: [2]

Figure 2-13: An example of FTM calculation for $k = 2$

Source: [2]

2.6.3. Rate Correction calculation

The value of Rate Correction is calculated during the network idle time of every odd cycle from two consecutive measurements. Even cycles borrow values from their previous odd cycles. First we calculate the difference of deviations for even and odd cycles separately for channels A and B. We then take their arithmetic mean. The result is a single table of values which is then used as an input for the FTM algorithm. The FTM's output is consequently a subject to the *pClusterDriftDamping* parameter (range of insensitivity). Let's denote the result of which as g . The final value for Rate Correction is then saturated by $\min(g, pRateCorrectionOut)$ where the second argument represents the maximum admissible value for Rate Correction.

The final output of RC is an integral value which denotes by how many microticks should the next communication cycle be adjusted. Positive values represent extension and negative values shortening. The change is applied to the next two consecutive cycles and is evenly distributed over macroticks so that no two consequent macroticks differ by more than one microtick.

2.6.4. Offset correction calculation

The Offset Correction value is computed in each cycle. First the minimum value for each row is taken from the table of deviations. This produces a list of values as an input for the FTM algorithm. Subsequently, we denote the FTM's output as g . The resulting value is then saturated by $\min(g, pOffsetCorrectionOut)$ where the second argument represents the maximum admissible value for Offset Correction.

2.7. Startup mechanism

All nodes in a FlexRay cluster need to set up a common perception of time in order to be able to stick to their scheduled time slots and be able to receive from others. The startup mechanism thus has to perform an initialization of the time base. This is done by so called coldstart nodes. To startup a FlexRay network at least two coldstart nodes are required. One of the two becomes a leading coldstart node and the other a following coldstart node.

Prior to the startup all the nodes must be in the *ready* state meaning that they already need to be configured and if required also woken up. As the startup commences all nodes enter the *coldstart-listen* state. Each node stays in this state for a random amount of time during which it listens to the communication channel. The first node to leave this state transmits its CAS (*Collision Avoidance Symbol*). By doing so it becomes the leading coldstart node. Other coldstart nodes assume the roles of following coldstart nodes.

It can occur that two nodes transmit their CAS at the same time. For this reason a leading coldstart node always transitions to the *collision resolution stage* after transmitting the CAS. During this phase the leading coldstart node transmits a startup frame in four consecutive cycles and listens for possible collisions. In case a collision occurs all nodes have to recognize this and return to the *coldstart-listen* state and the startup process is repeated while each node decreases its counter of remaining coldstart attempts. The *coldstart-listen* state may only be entered if the number of remaining coldstart attempts is greater than zero.

If there are no conflicts detected then just after two cycles are the following coldstart nodes able to determine the correctness of the time schedule by computing the time interval between the startup frames. All nodes must know the schedule beforehand. They merely verify its correctness. The other two cycles are essential to perform rate and offset corrections. Following the four cycles all other coldstart nodes proceed to transmit their startup frames. The leading coldstart node enters the *coldstart consistency check* stage in which it checks whether the frames transmitted by following coldstart nodes comply with its schedule. This

again takes four cycles. At the end of the fourth cycle, if not interrupted by consistency check errors, the cluster is successfully started and other non-coldstart nodes can join starting with the next cycle.

3. System architecture

The platform is to serve as a flexible and unique tool for monitoring and testing of vehicular networks. In its current state it is capable of monitoring FlexRay clusters by using the integrated FlexRay controller as one of the cluster's nodes. The FPGA is a key part of the system since it's responsible for the mapping of all communication outputs from the MCU to their respective drivers. But equally important are the FlexRay controllers contained within. Their number can be changed by loading a different design into the FPGA. However, the system reacts flexibly to this and reads the number from the FPGA's special register. Both the firmware and the PC application then recognize this number and provide control to the user of each controller without the need to recompile. The latest hardware version offers two physical FlexRay drivers (2 x 2 channels) to which either controller type (MCU or FPGA) can be mapped. At the time of development the mapping was static.

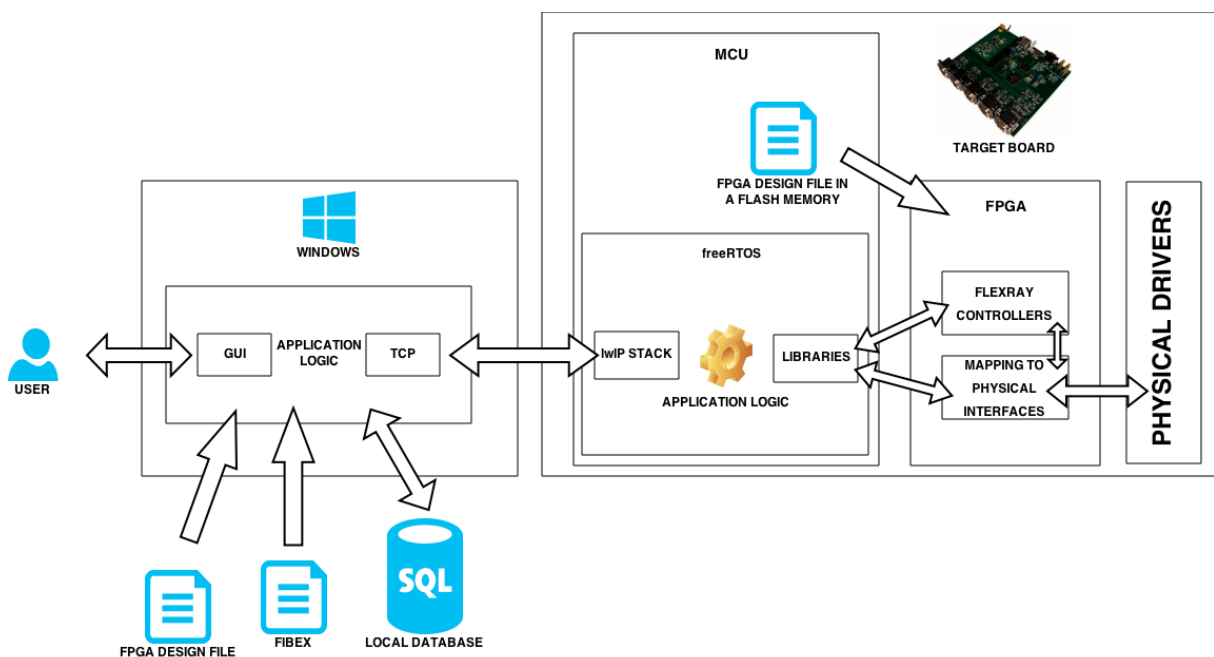


Figure 3-1: System architecture

All FlexRay controllers in the FPGA are capable of non-standard operations when compared to commercially available controllers. Those capabilities are aimed at the testing of the parameters of FlexRay networks. The ability to change node's parameters at runtime represents the core principal behind the tests. The MCU partakes in the tests by executing sets of commands responsible for coordinating the tests and reading the results. FlexRay

controllers in the FPGA are taken from source [14] and stand as a key component which this thesis integrates into a complex testing system.

Apart from FlexRay the target board is also equipped with CAN and LIN drivers. Their utilization is not within the scope of this thesis but they are ready for future applications. Their usage can either be added to all components of the system meaning the PC application, the communication protocol and the firmware. Or they can be controlled purely from custom tasks which are described in more detail in section 5.3.

4. FreeRTOS

FreeRTOS is an open source real time operating system targeted at microcontrollers and small microprocessors. It has been largely successful over its 12 years of existence. A vibrant community has been formed around freeRTOS providing free professional-level support. The kernel has a very small binary image. The exact size varies depending on the components used. Despite being free freeRTOS has successfully made it into commercial applications and is known to be reliable. Such a track-record combined with the fact that a port for our chosen architecture can be easily generated with the Halcogen tool made freeRTOS a clear choice for this platform over its only considered competitor – RTEMS.

4.1. Port settings

The Halcogen tool from Texas Instruments offers an easy way of generating a freeRTOS port with the desired parameters. Here is a list of chosen settings:

- **Tick Rate – 1000 Hz**, that means one tick equals one millisecond
- **Minimum Stack Size – 128 words**
- **Preemption – Enabled**
- **Number of Priorities – 3**, despite only two being actually used. Wasting of processor time is prevented by tasks blocking while waiting for resources. While not being blocked tasks share processor time equally. The remaining priority is provided for possible future use.
- **Heap Size – 32 768 bytes**, this current setting may be adjusted according to need. For instance additional heap space might be needed if a large number of user tasks were defined. More about user tasks in section 5.3.

- **Memory sections** – are not adjustable by the Halcogen’s GUI but can be considered part of port settings. However, as of now memory sections are irrelevant for this project since the memory protection unit (MPU) of freeRTOS is disabled. Though they do need to be considered if that were to change as the project expands.

4.2. Features used

The features of the real-time operating system that were used to build the firmware are listed in this section. These include tasks, queues and mutexes. All of these elements have to be dynamically allocated and therefore can fail to be created. It is a good practice to check the handler after allocation to see whether it has been successful. This can save a lot of time debugging for any programmer expanding the firmware with new features (such as the planned CAN and LIN).

4.2.1. Tasks

A task in freeRTOS just as in any other operating system represents a small program in and of itself. There are two basic ways of creating tasks in freeRTOS depending whether we take advantage of the MPU or not. The two functions to create tasks are *xTaskCreateRestricted* and *xTaskCreate* respectively. Since the MPU is not utilized in this project only the latter function is used. Here is a list of tasks used in this thesis:

- **Command_dispatcher** – stack size = 128 words, function = `commandDispatcher`, priority = 1, parameters = none
- **Mcu_controller_task** – stack size = 128 words, function = `mcuStateMachineTask`, priority = 1, parameters = none
- **Fpga_controller_task** – stack size = 128 words, function = `fpgaStateMachineTask`, priority = 1, parameters = index of the controller, the number of these tasks is determined by the value read from the FPGA. One task for each FlexRay controller is created up to a maximum defined by the macro `MAX_FPGA_FR_CONTROLLERS`.
- **Lwip_server_task** – stack size = 2048 words, function = `lwipTask`, priority = 1 (later lowered to 0), parameters = none
- **Tcp_send_task** – stack size = 128 words, function = `tcpSenderTask`, priority = 2, parameters = none
- **User defined tasks** –refer to section 5.3.

4.2.2. Queues

Queues are a means of passing data between tasks and can also serve as a way of synchronization since queues in freeRTOS are capable of blocking for a certain period of time or indefinitely. When inserting data into a queue or when retrieving it the data is always copied. Therefore, special care must be taken when dealing with large data structures. In such a situation it is advised to design the program's architecture in such a way that only pointers are stored in the queue. This approach is not used in scope of this project because the largest queued data structure is 263 bytes long and it happens very sparsely. This size has been chosen because it is required to accommodate data of largest possible FlexRay payload (254 bytes). However, most commands are much shorter than that so only the required number of bytes is copied. Here is a list of used queues:

- **commandQueue** – element size = 263 bytes, number of elements = 3
- **mcuFrControllerQueue** – element size = 263 bytes, number of elements = 2
- **fpgaFrControllerQueue** – element size = 263 bytes, number of elements = 2, the number of these tasks is determined by the value read from the FPGA. One task for each FlexRay controller is created up to a maximum defined by the macro `MAX_FPGA_FR_CONTROLLERS`.
- **tcpSendQueue** - element size = 2 bytes, number of elements = 1

4.2.3. Mutexes, semaphores and binary semaphores

Mutexes, semaphores and binary semaphores in freeRTOS all use the same handler type *xSemaphoreHandle*. The way to distinguish them is through the method called to initialize them. Mutex is a binary semaphore that employs the priority inheritance mechanism. Mutexes are suitable for mutual exclusion. Semaphores and binary semaphores are very similar to mutexes but they do not include priority inheritance. Binary semaphores can same as mutexes only be either taken or free (not-taken, unlocked, etc.). A regular (counting) semaphore contains a counter which determines how many times it can be taken without releasing it. Both semaphore types are best suited for synchronization. Mutex named **tcpSendProtection** is used in the firmware. It is created as the binary semaphore type and is responsible for the synchronization of requests to send data from multiple state machines (both MCU and FR). Specifically, it protects the access to the TX buffer which is shared by all tasks. The mutex is released as soon as the data is written to the EMAC buffers.

5. Firmware

5.1. Porting of the lwIP stack

In order to build a protocol based on TCP between the PC application and the MCU an IP stack needs to be ported for the combination of the architecture and the real-time operating system. This can be done in many ways. For example one of the decisions that need to be made is what kind of API we want to use. LwIP offers three layers from which we can choose.

- BSD socket API – the primary advantage is its portability to other stacks. It is sequential which means that it requires threading to operate it. One thread uses the API and the second thread runs the stack itself (takes care of timers, incoming packets etc.)
- Netconn API – not portable to other stacks, sequential
- Raw API – not portable to other stacks, uses callbacks, best performance since it doesn't have to deal with thread switching

First two options are more complicated to implement and require a deeper knowledge of the lwIP stack. Also, issues with performance might arise if more user tasks were defined. Since the porting of the lwIP stack is not the objective of this thesis but only means to an end a decision has been made to use the raw API.

5.2. Command processing

Figure 5-1 depicts the flow of command processing in the MCU. Whenever a TCP packet is received a callback function is invoked. The callback does not interpret the data. Its only job is to copy the received data into the command queue and signal the reception of data to the lwIP stack.

Considering the frequency of incoming commands it is not expected that there should be more than one element in the queue at any time. However, the queue is set to a capacity of three to allow for extreme cases.

Data from the command queue is read by the **Command_dispatcher** task. It is set to wait indefinitely so it doesn't waste any processing time when there is no command to be processed. Upon successful retrieval of a command from the queue the dispatcher reads the command type coded in the first byte. Depending on the command type it has three options. It

can process the command itself, pass it to the **Mcu_controller_task** or pass to one of the **Fpga_controller_tasks** identified by the index in the second byte.

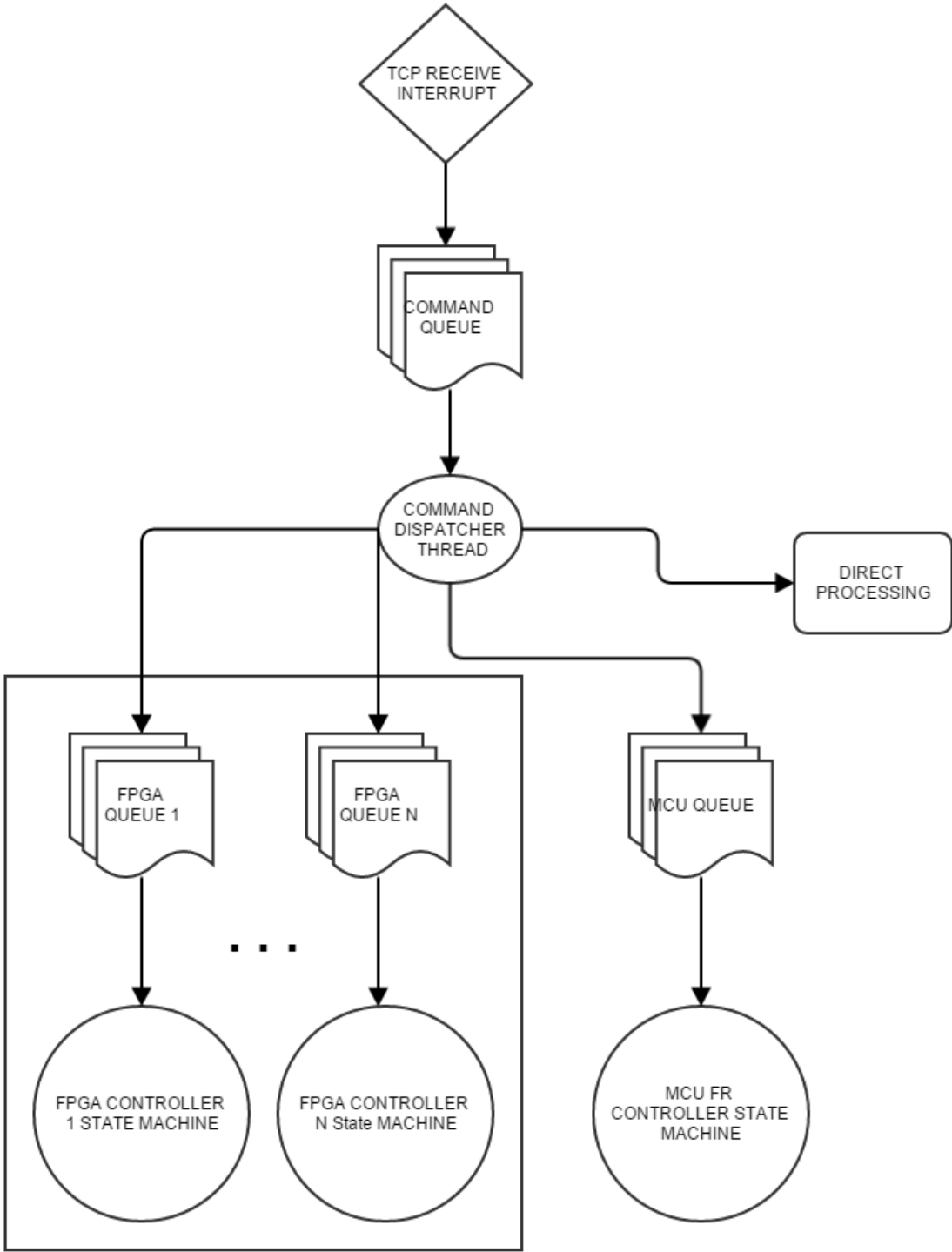


Figure 5-1: Command dispatching

Commands that do not belong to any FlexRay controllers are processed immediately by the dispatcher. For example when the PC application requests the number of FlexRay

controllers in the FPGA. This information is has already been stored during initialization so the dispatcher simply replies with the value. Another example would be the request for the list of available user tasks.

5.3. Task declaration macros

One of the required features is the possibility for a programmer to define arbitrary tasks and to be able to run or suspend these tasks from the PC application. To make the definition of user tasks easier a macro has been written which serves as a sort of a task declaration API which wraps the freeRTOS task declaration API with additional code. This way the programmer doesn't need to understand exactly how the management of user tasks is implemented.

To declare a user task the programmer has to look for a section bounded by `/*----USER TASK DECLARATION----*/` and `/*----END OF USER TASK DECLARATION----*/`. All the user tasks are supposed to be declared within this area using the following macro:

```
DECLARE_TASK_MANAGER_TASK( function, name, stack, params, priority, handle )
```

The passed arguments are:

- **Function** – a pointer to a function which the task is going to perform. The function must never exit.
- **Name** - const char * const type variable that is going to be displayed in the PC application as the name of the task. The programmer must avoid using the '|' character in the name since it is used as a separation character in the communication protocol.
- **Stack** – A value that represents the size of the stack in words that the operating system has to allocate for the task.
- **Params** – parameters that will be passed to the task
- **Priority of the task** – it is recommended to use 1 but if a higher number is chosen, the programmer needs to make sure that the task either blocks or yields often enough not to starve other tasks.
- **Handle** – here the programmer has to put in `taskManager[x]` where x is the index to the `taskManager` array. This value should be ascending for every declared task going from 0 to `SIZE_OF_TASK_MANAGER - 1`.

Here is an example of a correct user task declaration area:

```
uint32 ledTimeOne = 1000;
uint32 ledTimeTwo = 3000;
uint32 sciTime = 2000;
uint32 stack = configMINIMAL_STACK_SIZE;
...
...
...
/*-----USER TASK DECLARATION-----*/
    DECLARE_TASK_MANAGER_TASK( ledTask, "led_flash_one", stack, ledTimeOne, 1,
taskManager[0] )

    DECLARE_TASK_MANAGER_TASK( ledTask, "led_flash_two", stack, ledTimeTwo, 1,
taskManager[1] )

    DECLARE_TASK_MANAGER_TASK( consoleTask, "console_task", stack, sciTime, 1,
taskManager[2] )

/*-----END OF USER TASK DECLARATION-----*/
```

When passing parameters to tasks it is the programmer's responsibility to cast them correctly in the task function since they are always being passed as pointers to the void type.

5.4. TCP server implementation

The server part of the system is implemented in the MCU. Upon power up or reset the MCU initializes all necessary peripherals. This happens before starting the scheduler in all cases except the EMAC. The initialization of EMAC and the lwIP stack is performed by the **Lwip_server_task** before entering the endless loop every freeRTOS task is required to have. The last function called by this task is **server_init** which allocates the **struct tcp_pcb** variable. After that it lowers its own priority to that of the idle task and enters an infinite loop. Packet reception is then handled through callbacks.

It is done this way because all the components of the lwip used in this project are designed to pass around a pointer to the pcb. The pcb variable needs to be kept valid, which means we cannot allow the stack space to be freed. It would require a lot of extra time and effort to rewrite the lwip which also invites a number of potential errors.

Packet transmission is handled by a special task (*tcpSenderTask*). Whenever a task wants to send data over the TCP it has to acquire the *tcpSendProtection* mutex which protects access to the TX buffer. After writing the data into the buffer the task has to enqueue the data length into *tcpSendQueue*. This causes the *tcpSenderTask* to immediately preempt any other running task since it has the highest priority. And it writes the requested amount of data into the EMAC buffers. Consequently, it releases the *tcpSendProtection* mutex. The queue can

only hold one element at a time. There is no point making this queue any bigger because no other task can acquire the *tcpSendProtection* mutex until the *tcpSenderTask* releases it.

5.5. EMIF

The External Memory Interface (EMIF) is a controller integrated in the TMS570LS3137ZWT chip. The purpose of EMIF is to provide a means for the MCU's core to connect to a variety of external devices including SDRAMs or asynchronous devices such as NOR Flash and SRAM. In case of this project it is used to interface to the FPGA's registers. The registers are mapped to CPU's address space and can be accessed simply by reading from and writing to an address using pointers. The following macros are provided to facilitate the access:

```
#define READ_CONTROLLER_VALUE(BASE, CONTROLLER_INDEX, OFFSET) \  
(*((volatile unsigned int *) ((BASE) + ((CONTROLLER_INDEX)*(FPGA_FR_CONTROLLER_LENGTH)) + (OFFSET))))  
  
#define WRITE_CONTROLLER_VALUE(BASE, CONTROLLER_INDEX, OFFSET, VALUE) \  
(*((volatile unsigned int *) ((BASE) + ((CONTROLLER_INDEX)*(FPGA_FR_CONTROLLER_LENGTH)) + (OFFSET))) = (VALUE))  
  
#define READ_VALUE_8BIT(BASE, OFFSET) \  
(*((volatile unsigned char *) ((BASE) + (OFFSET))))  
  
#define WRITE_VALUE_8BIT(BASE, OFFSET, VALUE) \  
(*((volatile unsigned char *) ((BASE) + (OFFSET))) = (VALUE))  
  
#define READ_VALUE_32BIT(BASE, OFFSET) \  
(*((volatile unsigned int *) ((BASE) + (OFFSET))))  
  
#define WRITE_VALUE_32BIT(BASE, OFFSET, VALUE) \  
(*((volatile unsigned int *) ((BASE) + (OFFSET))) = (VALUE))
```

It should be noted that tasks in freeRTOS normally only allow access to their own stack and to the heap. Other memory regions are accessible according to the setting of MPU regions. The MPU is not needed in this project so it is left uninitialized. Otherwise either the memory region access right would have to be changed or a special restricted type of task would have to be used instead (using *xTaskCreateRestricted*). Restricted tasks are an unnecessary over complication. Leaving the MPU off allows for the use of regular tasks even when accessing the EMIF memory regions.

Communication constants of the EMIF controller have to be set in mutual compliance with the FPGA's EMIF module. It is configured for asynchronous access. Here are the parameters used:

EMIF: ASYNC1 Config	
<input type="checkbox"/> Select Strobe Mode	<input type="checkbox"/> Page Mode
<input type="checkbox"/> NOR Flash	Page Delay: <input type="text" value="0"/> Cycles
<input type="checkbox"/> Extended Wait	Page Size: <input type="text" value="4_words"/>

EMIF: ASYNC1 Timings	
W_SETUP: <input type="text" value="1"/> Cycles	ASIZE: <input type="text" value="16_bit"/>
W_STROBE: <input type="text" value="10"/> Cycles	ASYNC1 WAIT: <input type="text" value="pin0"/>
W_HOLD: <input type="text" value="1"/> Cycles	
R_SETUP: <input type="text" value="1"/> Cycles	
R_STROBE: <input type="text" value="9"/> Cycles	
R_HOLD: <input type="text" value="1"/> Cycles	
TA: <input type="text" value="1"/> Cycles	

Figure 5-2: EMIF Settings

5.6. MCU state machines

All state machines in the MCU are event-driven. That means they can only perform actions on edges i.e. while receiving a command.

5.6.1. MCU Connection state machine

This state machine is the simplest in the MCU. Its only purpose is to track the connection state of the protocol. The state of the connection has three phases. First one is IDLE, which is the initial state. Upon receiving a request for the number of FlexRay controllers in the FPGA the dispatcher task sends the response and transitions to the FPGA_READOUT_SENT state. Here it remains until the PC application requests the firmware version together with a list of supported commands. It then sends the needed answer and finally transitions to the PAIRED state. At this point it is allowed to pass commands to their respective FlexRay state machines.

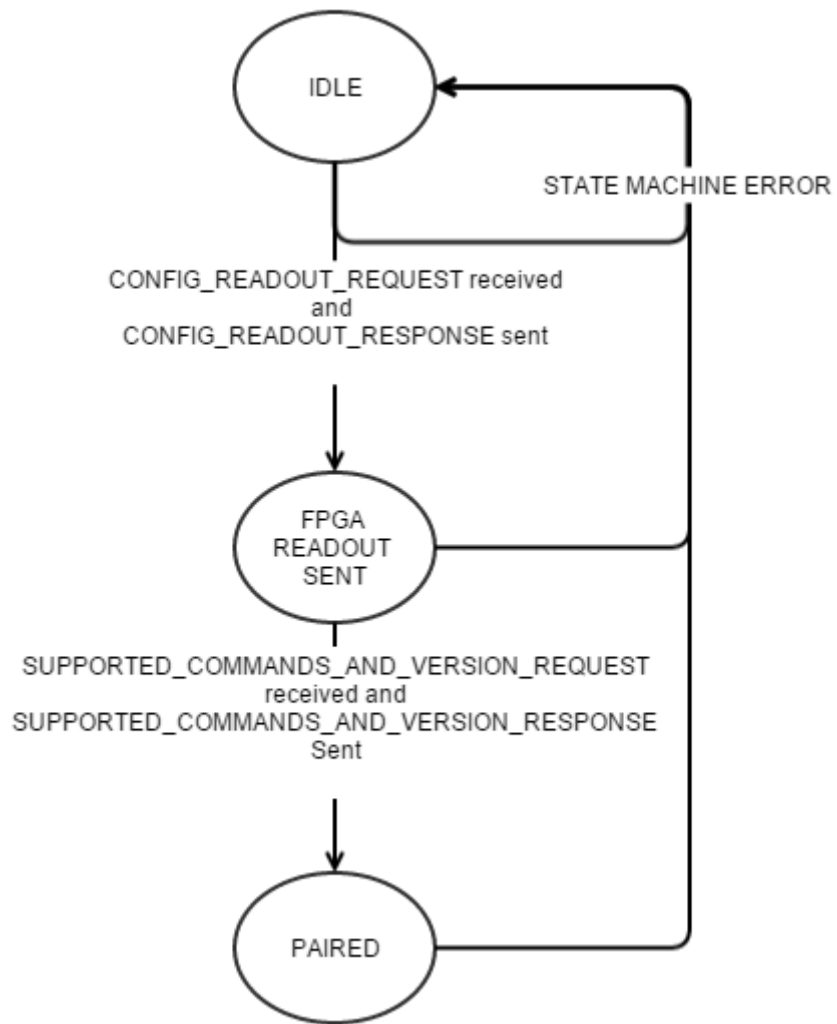


Figure 5-3: MCU Connection state machine

5.6.2. MCU FR state machine

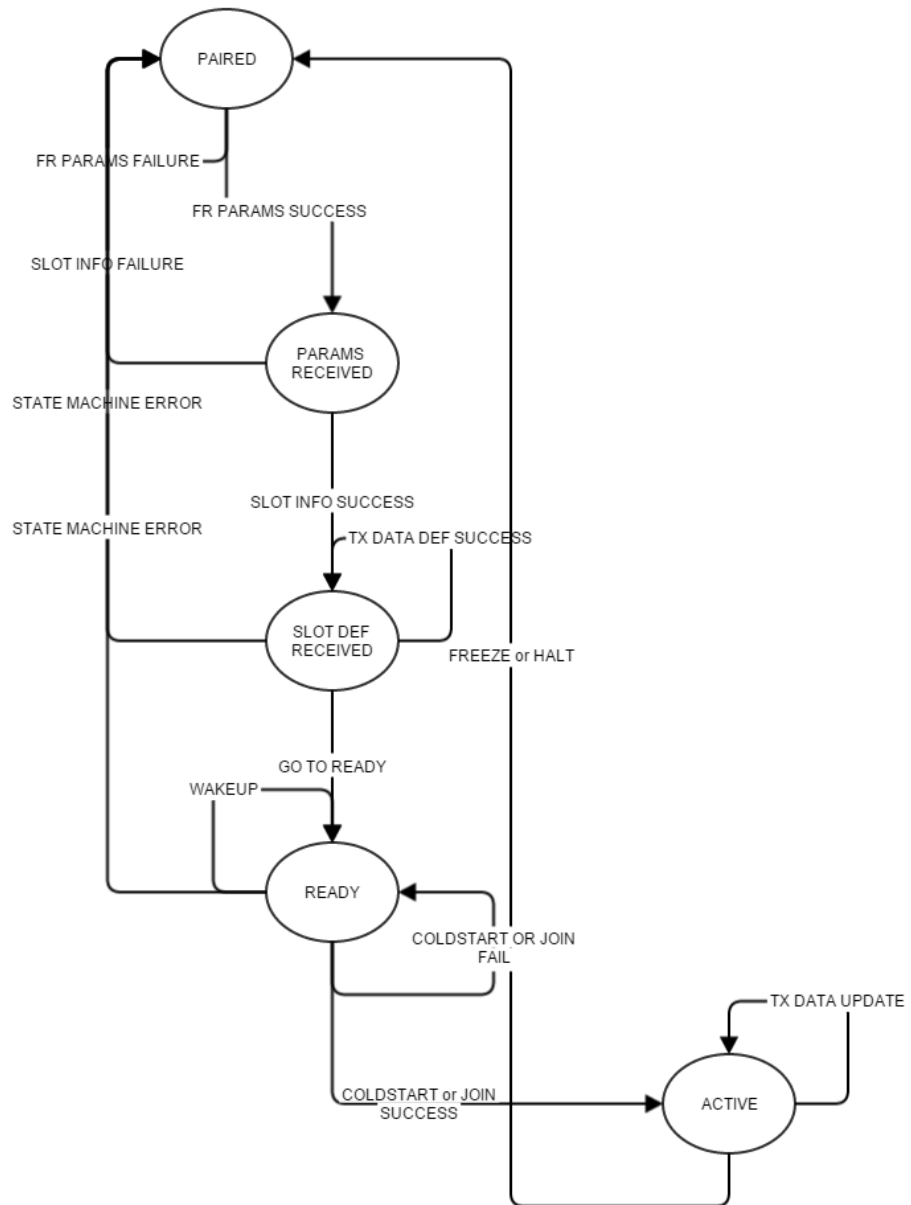


Figure 5-4: State machine for handling the MCU's FlexRay controller

State machine depicted in Figure 5-4 is responsible for implementing the parts of the communication protocol that concern the FlexRay controller in the MCU. It starts with the PAIRED state which is where the connection state machine left off and started passing commands to FlexRay state machines. The states can be divided into two phases - configuration and operation. Configuration includes the PAIRED, PARAMS_RECEIVED and SLOT_DEF_RECEIVED states. Upon receiving the following constants:

- *pKeySlotUsedForStartup*
- *pKeySlotUsedForSync*
- *gColdStartAttempts*
- *pAllowPassiveToActive*
- *pWakeupChannel*
- *pSingleSlotEnabled*
- *pAllowHaltDueToClock*
- *pChannels*
- *pdListenTimeOut*
- *gListenNoise*
- *gMaxWithoutClockCorrectionPassive*
- *gMaxWithoutClockCorrectionFatal*
- *gNetworkManagementVectorLength*
- *gdTSSTransmitter*
- *gdCASRxLowMax*
- *gdSampleClockPeriod*
- *pSamplesPerMicrotick*
- *gdWakeupSymbolRxWindow*
- *pWakeupPattern*
- *gdWakeupSymbolRxIdle*
- *gdWakeupSymbolRxLow*
- *gdWakeupSymbolTxIdle*
- *gdWakeupSymbolTxLow*
- *gPayloadLengthStatic*
- *pLatestTx*
- *pMicroPerCycle*
- *gMacroPerCycle*
- *gSyncNodeMax*
- *pMicroInitialOffset[A]*
- *pMicroInitialOffset[B]*
- *pMacroInitialOffset[A]*
- *pMacroInitialOffset[B]*
- *gdNIT*
- *gOffsetCorrectionStart*
- *pDelayCompensation[A]*
- *pDelayCompensation[B]*
- *pClusterDriftDamping*
- *pDecodingCorrection*
- *pdAcceptedStartupRange*
- *pdMaxDrift*
- *gdStaticSlot*
- *gNumberOfStaticSlots*
- *gdMinislot*
- *gNumberOfMinislots*
- *gdActionPointOffset*
- *gdMinislotActionPointOffset*
- *gdDynamicSlotIdlePhase*
- *pOffsetCorrectionOut*
- *pRateCorrectionOut*
- *pExternOffsetCorrection*
- *pExternRateCorrectio*

Along with additional information about channel usage, startup and synchronization the state machine writes them into the FlexRay controller. Subsequently, the state advances to PARAMS_RECEIVED. In this state the information about RX and TX frames is expected. The order of buffers is important. RX frames go first and TX frames second. This is necessary to make the firmware much simpler and more elegant. This information is stored and the firmware uses it to calculate pointers to the FlexRay message RAM for all the message buffers. RX buffers are configured right away since they don't require any payload data. Afterwards, the machine shifts to the SLOT_DEF_RECEIVED state. Here it receives the data that is to be transmitted out of the TX buffers. The order of the buffers has to follow the same order in which the frames were defined in the previous stage. For example if the following four frames were defined:

- slotID = 12, RX
- slotID = 2, RX

- slotID 5, TX
- slotID 23, TX

Then the first TX data must be for the frame transmitted in slot 5 and the second in slot 23. At the end of data definition a double acknowledge mechanism is used. See Figure 5-5 for better descriptiveness.

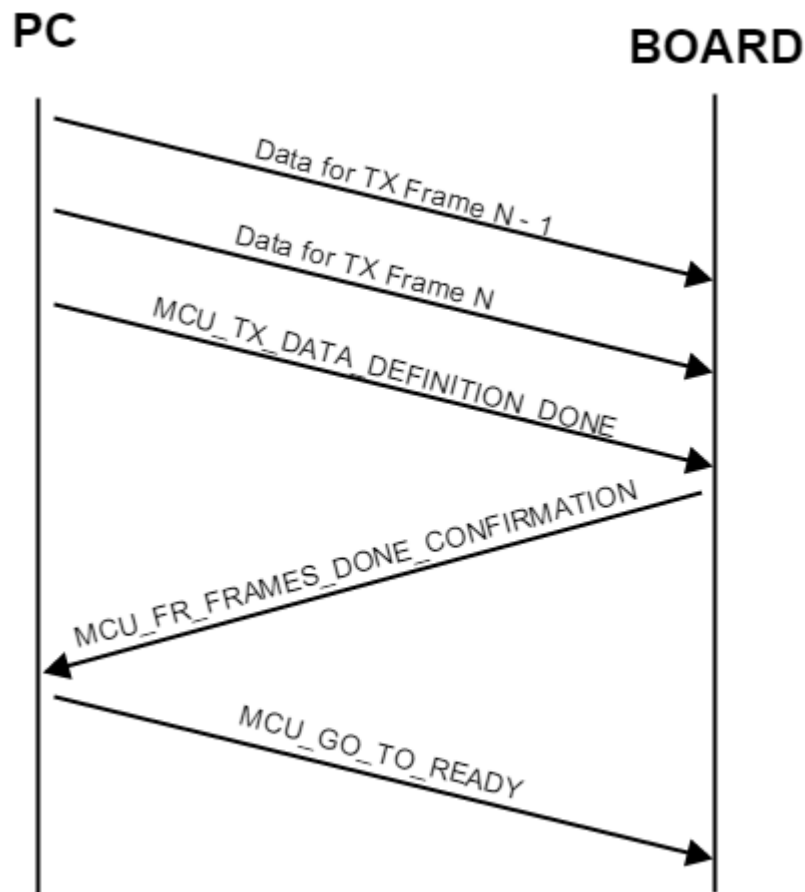


Figure 5-5: Details of message exchange at the end of data definition

When the MCU receives the `MCU_GO_TO_READY` command it can finally transition to the `READY` state. Now, the FlexRay controller can optionally perform a cluster wakeup. Other than that, it waits until it is instructed to either coldstart or to join a running network. The startup procedure may fail. The result is reported to the PC application. In case of success the controller now finds itself in the `ACTIVE` state. In case of failure it remains in the `READY` state. Then a new startup command may be issued by the PC application. Details about the message format can be found in section 7.4.

5.6.3. FPGA FR state machine

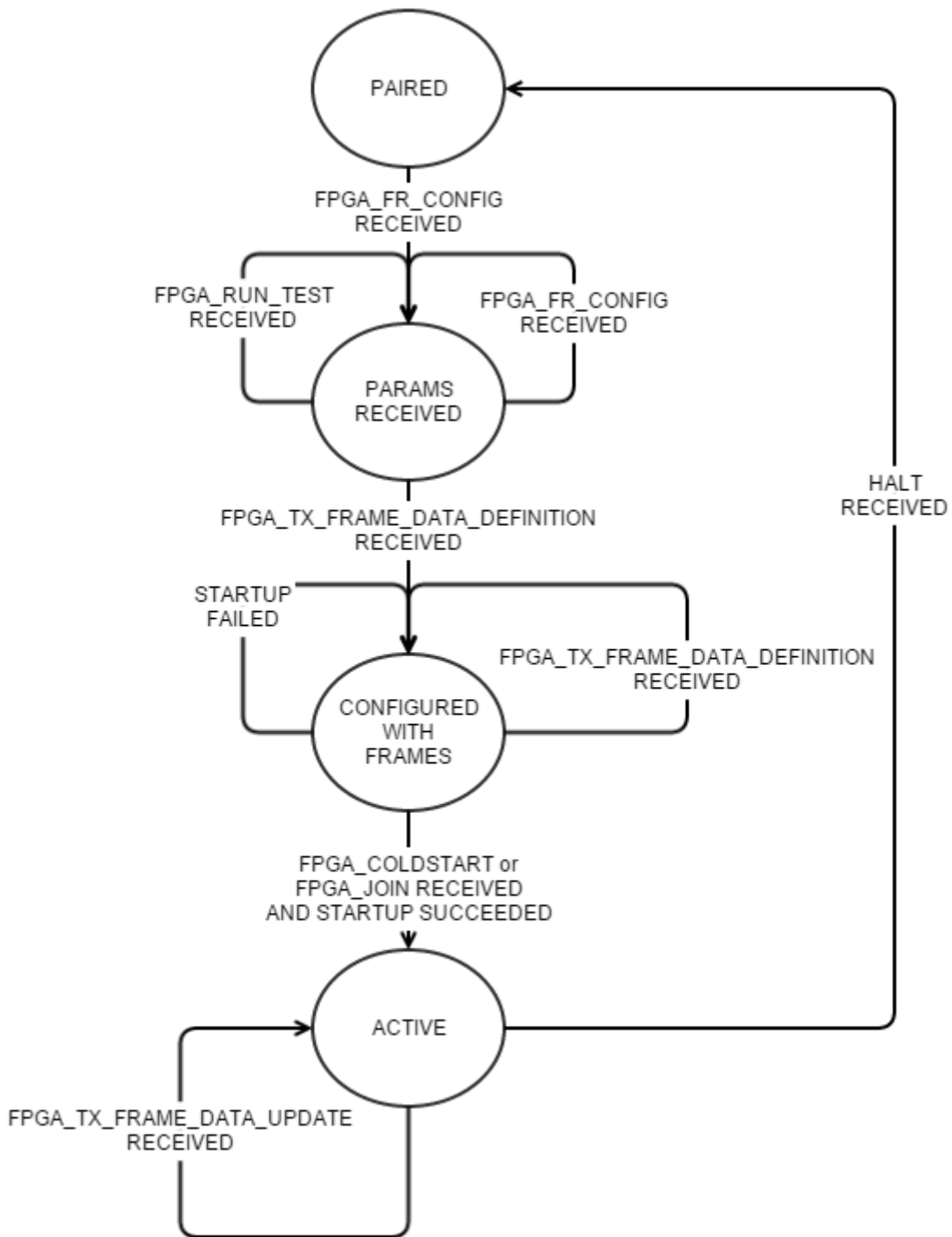


Figure 5-6: MCU FPGA State machine

Each **Fpga_controller_task** (section 4.2.1) manages a state machine shown in Figure 5-6. In addition to what can be seen in the figure each state also has an edge to the PAIRED

state as a reaction an internal reset command which is issued every time a new configuration is loaded into the FPGA. With the exception of this internal command the state machine takes action exclusively in reaction to commands passed to it by the command dispatcher. When an action is finished the task blocks on its receive queue waiting for a new command. This way it doesn't consume any CPU time when it's not needed.

First the state machine expects cluster and node parameters in the format which is detailed in section 7.4. After receiving the first definition of a TX frame it transitions to the CONFIGURED WITH FRAMES state. Other TX frames sent by the client are processed in this state. However, there is a limit to the number of TX frames that can be defined in a single FPGA FlexRay controller. The current number is 4. This constant can be adjusted in the VHDL code of the controller. If the client tries to define more frames than that the state machine responds with an error message.

The PARAMS RECEIVED state is intended for the future implementation of running tests. Configuring TX or RX buffers is usually included in the tests themselves. Only FlexRay parameters need to be set. That is why PARAMS RECEIVED is the correct launching state and not CONFIGURED WITH FRAMES.

6. PC Application

6.1. State machines

Just like in the case of MCU the state machines in the PC application are edge oriented. Only, in this case, they can react not only to received frames over TCP but also to user interactions. In addition to what can be seen in the figures, each state possesses an edge to the IDLE state in case of an unexpected action (state machine error). Those edges were left out to keep the graphs neatly arranged.

The PC application manages only two state machines. The first one is solely responsible for managing the connection to the board, keeping track whether the fibex file has been loaded and the MCU's FlexRay controller itself. It also spawns the instances of state machines responsible for the FPGA controllers. The decision to merge all these functions into one state machine has been made for several reasons:

- It saves lines of code
- It avoids having to coordinate more state machines with one another
- It makes it clearer (which might of course be subjective)

So the state machine contains all the functionality that could reasonably possible be fit in. However, it was not feasible to include the FPGA state machines since there are multiple instances of those and the number of them is not known beforehand. And even if it were the resulting states would be a Cartesian product of all the states starting at a certain point. This would unacceptably inflate the number of states.

6.1.1. General state machine

At the beginning the state machine splits into two branches (see Figure 6-1). One of them is where the fibex file is loaded before connecting to the board and the other one after. A fibex file is a compulsory input for this application. It has been chosen as the standard way of describing FlexRay networks in the industry. However, the user is still allowed to edit the parameters even after loading the fibex. This gives the user freedom to experiment without having to edit the fibex file itself but it can also compromise the ability of controllers to integrate into a cluster. There is no mechanism in place which would check whether the new parameters are still compatible with the originally loaded fibex. A user with at least basic knowledge of the FlexRay standard is assumed.

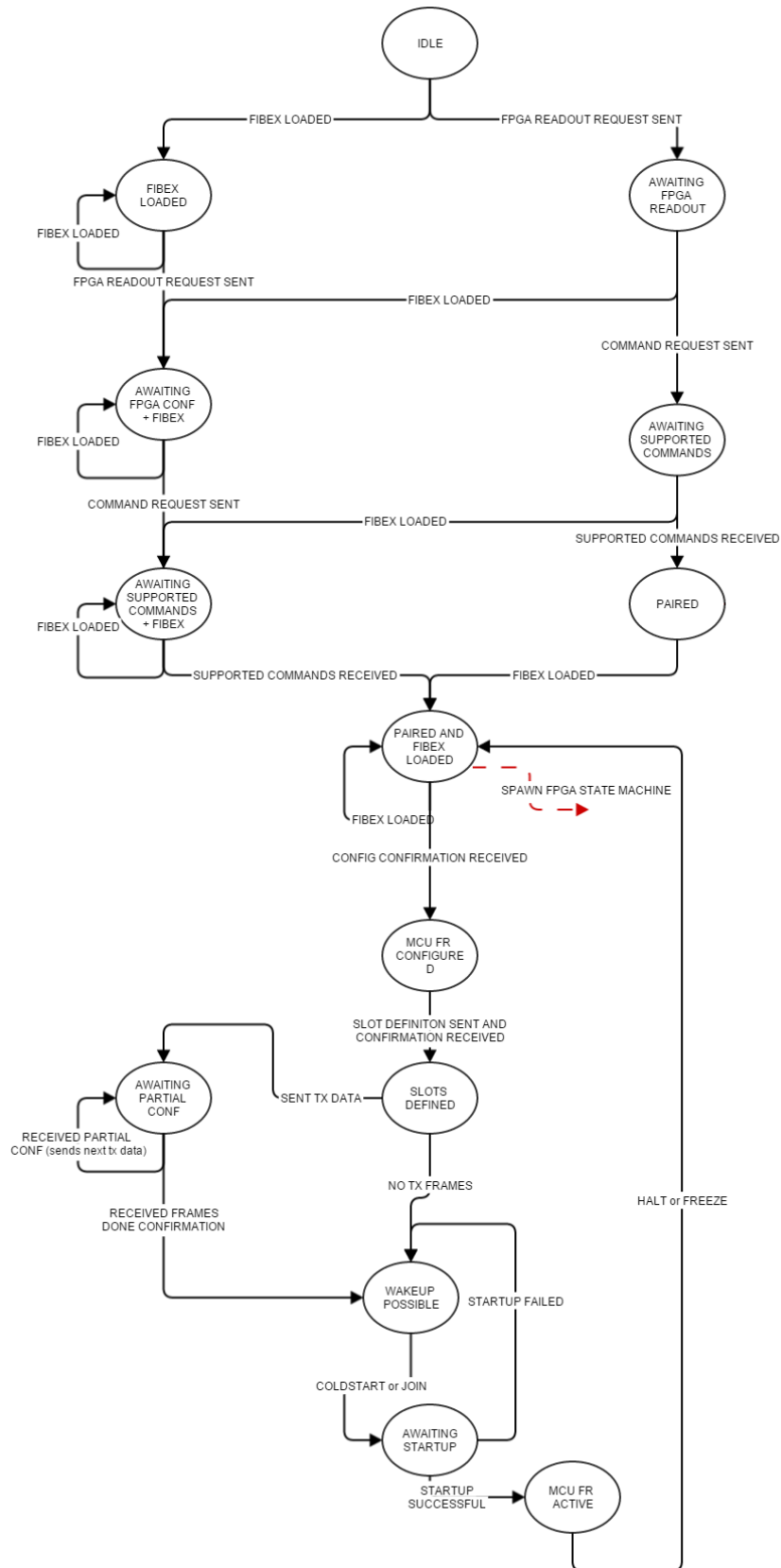


Figure 6-1: PC application's main state machine

Once the machine reaches the PAIRED or the PAIRED AND FIBEX LOADED state the "FPGA Status" window (see 6.2.5) can be opened. This spawns one FPGA state machine (see 6.1.2) for each controller in the FPGA. These state machines are managed through the UI of this window.

Next comes the configuration phase which practically mirrors the state machine in 5.6.2. The MCU_FR_CONFIGURED state is entered once a confirmation from the MCU is received that setting of the cluster constants is finished. Without any user interaction the state machine then proceeds to send details about monitored frames followed by frames added in the Message Editor tab. As mentioned previously, the order of frames matters! Then again without any user interaction, provided no errors were detected, the state machine starts sending data for the TX frames defined in the previous step (in the same order). This data is stored in their corresponding message buffers in the FlexRay controller and will be scheduled for transmission as soon as the node comes online. For the double acknowledgement mechanism which follows this data exchange refer back to Figure 5-5.

Now the UI enables the user to perform a wakeup of the cluster or select one of the startup options. After sending a command to perform a coldstart or to integrate itself to a running network the application waits for a confirmation from the MCU that the startup was successful. Monitoring is automatically triggered in case of successful startup. Issuing a halt or freeze command will set it back to PAIRED_AND_FIBEX_LOADED state and a new configuration can be used. If the startup fails the state machine transitions back to WAKEUP_POSSIBLE and the user can repeat the attempt.

6.1.2. FPGA State Machine

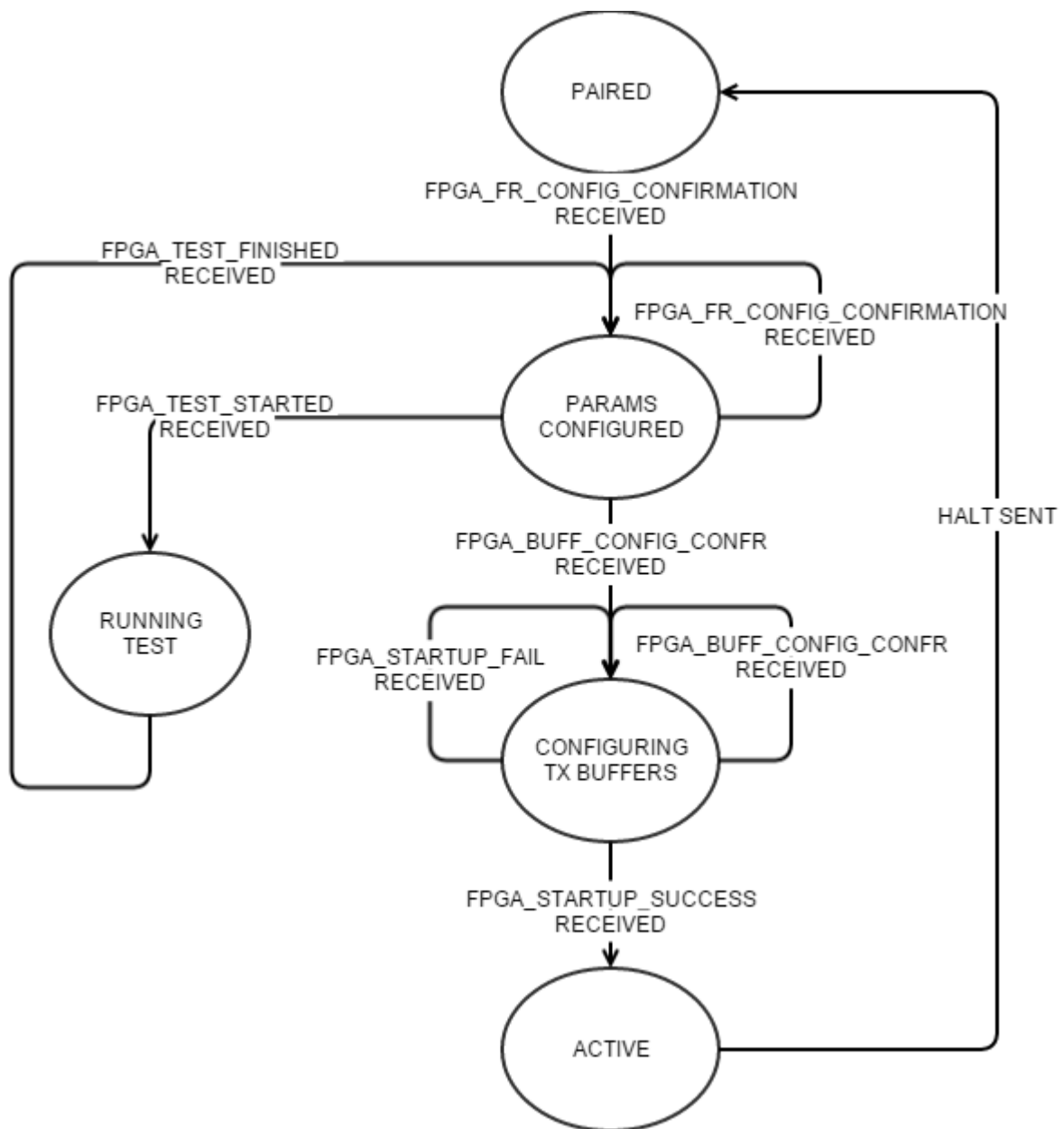


Figure 6-2: PC FPGA State machine

This state machine is practically a mirror copy to its MCU counterpart (see Figure 5-6). It has a separate state for testing since it's not in charge of the test's execution and has to wait for the server to report that the test is finished. In the meantime the state machine is not allowed to do anything else. State transition with the exception of sending a halt command is in all cases driven by receiving a confirmation for a successful transition on the server side.

6.2. Features

6.2.1. Monitoring and frame transmission

The primary purpose of the platform is to monitor communication of a FlexRay bus and to be able to transmit frames of its own. In order to monitor a frame it needs to be selected as a monitored frame in the Cluster Setup tab as seen in Figure 6-3.

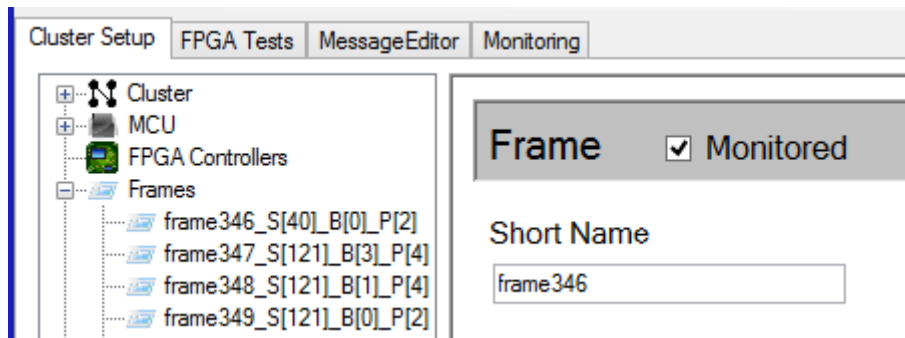


Figure 6-3: How to select a frame to be monitored

The next step is to define the outgoing communication. In the Message Editor tab the user can take advantage of the "Copy from Cluster Setup" button. In order to do that the user must first setup controller mapping. In Cluster Setup in the "MCU→Local Settings" menu choose one of the available ECUs in the "ECU Mapping" combo box. This also copies all the parameters from the ECU to the Local Settings panel. The same kind of mapping can be performed for the FPGA Controllers. Only the application must first be connected to the board (Actions→Connect to Board). This is necessary because the application must first find out how many FlexRay controllers are actually present in the FPGA. Once the mapping is done the button "Copy from Cluster Setup" will add all frames from the Cluster Setup, that belong to ECUs to which a physical controller is mapped. This way is much preferable to adding frames manually which is also supported.

The application distinguishes between two types of frames - MCU frames and FPGA frames. The difference can be noticed when clicking on frames of both types in the Message Editor tab. It is also shown in Figure 6-4.

Frame Name <input type="text" value="Frame0"/>	Startup/Sync <input type="text" value="No"/>	Channel <input type="text" value="Both"/>	Transmit mode <input checked="" type="radio"/> Continuous <input type="radio"/> One-shot
			Cycle Period <input type="text" value="1"/>
			Base Cycle <input type="text" value="0"/>
Slot ID <input type="text" value="0"/>	Dynamic <input type="text" value="No"/>	Length [B] <input type="text" value="2"/>	Data [Hex] <input type="text" value="ABCD"/>

VS

Trigger options <input type="radio"/> Timestamp <input type="radio"/> Macrotick <input checked="" type="radio"/> Macrotick and Cycle <input type="radio"/> Every Cycle <input type="radio"/> Every Odd Cycle	<input type="radio"/> Every Even Cycle <input type="radio"/> Immediately	Slot ID <input type="text" value="1"/>	Frame Name <input type="text" value="Frame 1"/>	Startup/Sync <input type="text" value="No"/>	Channel <input type="text" value="Both"/>
<input type="text" value="0"/> Macrotick	<input type="text" value="0"/> Cycle	Dynamic <input type="text" value="No"/>	Length [B] <input type="text" value="2"/>	Data <input type="text" value="ABCD"/>	
Source Controller <input type="text" value="FPGA - None"/>					

Figure 6-4: Difference between FPGA and MCU frames

These differences are needed due to different capabilities of said controllers to trigger frame's transmission. The FPGA controller offers more options. However, the MCU has the upper hand when it comes to periodicity. The cycle code is capable of expressing periods ranging from 1 to 64 cycles with offsets from 0 to 63. As opposed to the FPGA which can only send ever cycle or every even cycle.

An important thing to note is that if the user wants to see the frames transmitted by the MCU or one of the FPGA controllers he still needs to mark those frames as monitored. Having them in Message Editor is not enough. Figure 6-5 shows an example of the Message Editor tab with the MCU mapped to an ECU which transmits thirteen different frames and a FPGA controller which only has one frame. In order to coldstart a network one of the frames belonging to the coldstart node must be defined as "Startup & Sync". There may only be one such or "Sync" frame per node. The application takes care of this and doesn't allow the user to define more.

Frame Name	Source	Slot ID	Channel	Base Cycle	Cycle Period	Startup/Sync	Dynamic	Length [B]	Data
frame346_S[40]_B[0]_P[2]	MCU	40	Both	0	2	No	No	2	
frame347_S[121]_B[3]_P[4]	MCU	121	Both	3	4	No	Yes	2	
frame348_S[121]_B[1]_P[4]	MCU	121	Both	1	4	No	Yes	2	
frame349_S[121]_B[0]_P[2]	MCU	121	Both	0	2	No	Yes	2	
frame398_S[1]_B[0]_P[1]	MCU	1	Both	0	1	Startup & Sync	No	2	
frame398_S[74]_B[0]_P[1]	MCU	74	Both	0	1	No	No	2	
frame624_S[40]_B[1]_P[4]	MCU	40	Both	1	4	No	No	4	
frame1000_S[224]_B[0]_P[8]	MCU	224	Both	0	8	No	Yes	9	
frameAB_S[13]_B[1]_P[4]	MCU	13	Both	1	4	No	No	5	
frameA_S[13]_B[0]_P[2]	MCU	13	Both	0	2	No	No	2	
frameA_S[13]_B[3]_P[4]	MCU	13	Both	3	4	No	No	2	
frame_25_S[2]_B[0]_P[1]	MCU	2	Both	0	1	No	No	2	
frame_25_S[75]_B[0]_P[1]	MCU	75	Both	0	1	No	No	2	
frame397_S[88]_B[0]_P[1]	FPGA_Controller1	88	Both	0	1	No	No	2	

Figure 6-5: Message Editor tab example

Monitoring is automatically activated by coldstarting a network or joining one. It can be paused and started again at any time while the cluster is running. To change data being sent, select a frame from the combo box in the bottom panel of the Monitoring tab. Now the data can be adjusted. The controller is notified of the change by clicking on the green arrow. The selection in the combo boxes is filled automatically by frames belonging to the MCU or the FPGA. In case of the MCU frames this happens as it transitions into the WAKEUP_POSSIBLE state. As for the FPGA frames, all frames belonging to a particular FPGA controller are added to the selection when the application receives a confirmation of the "Configure Frames" action in the "FPGA Status" window. Those frames are also removed when the node leaves the active state.

Time [ms]	Cycle	Slot ID	Channel	Startup	Sync	Dynamic	Length [B]	Data
2	0	9	A	Yes	Yes	No	32	d769a09d9881b0d8166a637c3437200c0f72f77c002e4ede0f7254b6cec5225
3	0	8	A	Yes	Yes	No	32	f34e50adefc2a8a00c2b657d44669fccc067c7a3edebf94457c1209076e3ff7
13	1	9	A	Yes	Yes	No	32	d769a09d9881b0d8166a637c3437200c0f72f77c002e4ede0f7254b6cec5225
13	1	8	A	Yes	Yes	No	32	f34e50adefc2a8a00c2b657d44669fccc067c7a3edebf94457c1209076e3ff7
31	2	9	A	Yes	Yes	No	32	d769a09d9881b0d8166a637c3437200c0f72f77c002e4ede0f7254b6cec5225
31	2	8	A	Yes	Yes	No	32	f34e50adefc2a8a00c2b657d44669fccc067c7a3edebf94457c1209076e3ff7
50	3	9	A	Yes	Yes	No	32	d769a09d9881b0d8166a637c3437200c0f72f77c002e4ede0f7254b6cec5225
50	3	8	A	Yes	Yes	No	32	f34e50adefc2a8a00c2b657d44669fccc067c7a3edebf94457c1209076e3ff7
62	4	9	A	Yes	Yes	No	32	d769a09d9881b0d8166a637c3437200c0f72f77c002e4ede0f7254b6cec5225
62	4	8	A	Yes	Yes	No	32	f34e50adefc2a8a00c2b657d44669fccc067c7a3edebf94457c1209076e3ff7
80	5	9	A	Yes	Yes	No	32	d769a09d9881b0d8166a637c3437200c0f72f77c002e4ede0f7254b6cec5225
80	5	8	A	Yes	Yes	No	32	f34e50adefc2a8a00c2b657d44669fccc067c7a3edebf94457c1209076e3ff7
92	6	9	A	Yes	Yes	No	32	d769a09d9881b0d8166a637c3437200c0f72f77c002e4ede0f7254b6cec5225
92	6	8	A	Yes	Yes	No	32	f34e50adefc2a8a00c2b657d44669fccc067c7a3edebf94457c1209076e3ff7
110	7	9	A	Yes	Yes	No	32	d769a09d9881b0d8166a637c3437200c0f72f77c002e4ede0f7254b6cec5225
110	7	8	A	Yes	Yes	No	32	f34e50adefc2a8a00c2b657d44669fccc067c7a3edebf94457c1209076e3ff7
122	8	9	A	Yes	Yes	No	32	d769a09d9881b0d8166a637c3437200c0f72f77c002e4ede0f7254b6cec5225
122	8	8	A	Yes	Yes	No	32	f34e50adefc2a8a00c2b657d44669fccc067c7a3edebf94457c1209076e3ff7
140	9	9	A	Yes	Yes	No	32	d769a09d9881b0d8166a637c3437200c0f72f77c002e4ede0f7254b6cec5225
140	9	8	A	Yes	Yes	No	32	f34e50adefc2a8a00c2b657d44669fccc067c7a3edebf94457c1209076e3ff7
158	10	8	A	Yes	Yes	No	32	f34e50adefc2a8a00c2b657d44669fccc067c7a3edebf94457c1209076e3ff7
158	10	9	A	Yes	Yes	No	32	d769a09d9881b0d8166a637c3437200c0f72f77c002e4ede0f7254b6cec5225

Figure 6-6: Monitoring example

Internally, the frame reception is handled by a combination of callbacks and a dedicated thread. First a callback function of the TCP client is invoked upon frame reception. It then calls the state machine which in the MCU_FR_ACTIVE state extracts the data from the incoming packet and enqueues it in a buffer. While monitoring is active a dedicated thread dequeues the data from the buffer and adds it into the monitoring UI. It has to be done this way because the UI add method cannot keep up with the rate at which the data flows in. Addition of new data can fail and most importantly the responsiveness of the application suffers. A dedicated thread solves this problem by adding the messages at a pace the UI can handle.

6.2.2. Remote reconfiguration of the FPGA

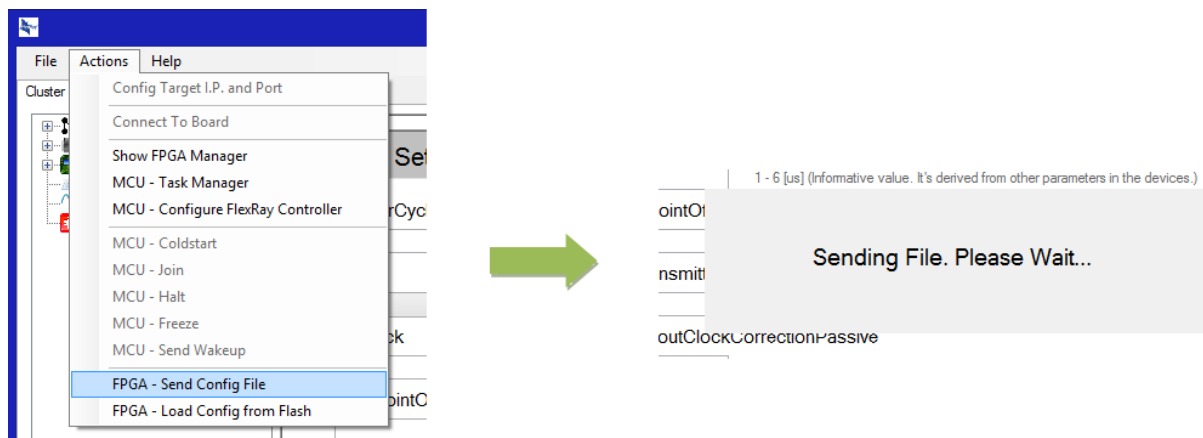


Figure 6-7: Remote configuration

After power up or reset the board's firmware automatically configures the FPGA over serial interface with a design stored in the MCU's flash memory. However, the FPGA can be reconfigured using the PC application by selecting "Actions→FPGA - Send Config File". This opens up a dialog window to select a .rbf file containing the FPGA's new configuration. This file is then sent to the board and loaded into the FPGA.

The default configuration from MCU's flash can always be restored by selecting "Actions→FPGA - Load Config from Flash". Every time a new design is loaded, the new number of FlexRay controllers is read from the FPGA. Afterwards, all tasks in the firmware managing those controllers are reset to their default state. PC application's UI is also adjusted to the new number and all its FPGA FlexRay state machines are reverted to their initial state.

6.2.3. Remote task control

One of the key features of the whole system is the ability to define and run custom user tasks in the MCU. These tasks represent a flexible way for a programmer to expand the functionality of the system without having to adjust the communication protocol and without deeper understanding of the MCU's firmware. The macro API for the user task definition on the MCU side is described in chapter 5.3.

Once the tasks are present in the MCU they will be reported to the application when selecting "Actions → MCU - Task Manager". The UI lists all available tasks as is shown in Figure 6-8. Task is activated by selecting it in the list and clicking "Run". Only one task may run at a time. In case another task had already been active it is now suspended and the

selected task becomes the active one. The currently running task is highlighted in the UI. A task may also be suspended without having to run another task. When the active task is selected the "Run" button becomes a "Suspend" button. The decision to only allow one active user task was a design decision and can easily be changed by a small change to the firmware if desired.

Even when this window is closed and reopened the request for available tasks is not repeated. The same list of tasks will be presented again.

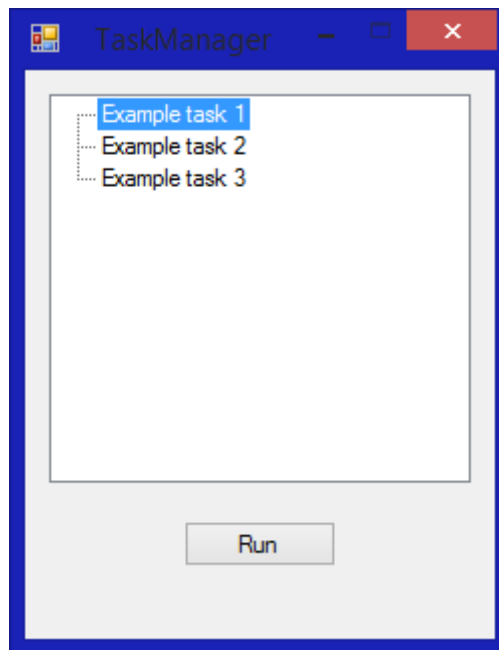


Figure 6-8: Task manager window

6.2.4. Fibex parsing

A fibex file describing the target FlexRay cluster is required to use any functionality of the platform. Fibex is a XML-based format and has been established in the industry as the standard way of defining automotive networks. The application parses only the data that is relevant for the purposes of its functionality with the exception of signals. The parsing is conveniently designed to report any missing cluster parameters. The UML diagram for the FlexRay fibex format can be seen in Figure 6-9.

All mentioned files were either examples distributed with the fibex standard (by the BMW group) or generated by third party software (such as NI-XNET Database Editor from National Instruments). Despite their incorrectness it is still useful to be able to parse them so the parsing algorithm is designed to handle such cases.

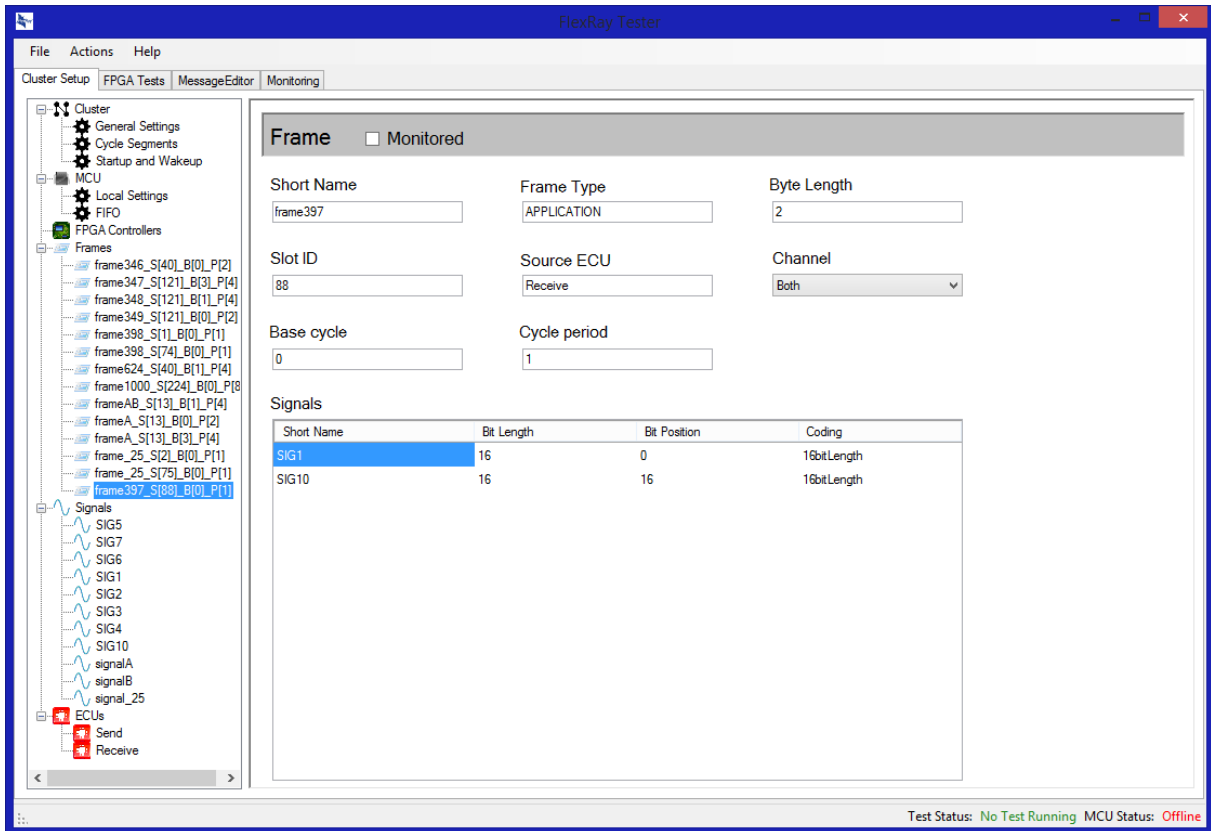


Figure 6-10: Cluster Setup after loading a fibex file

Constants *gdSymbolWindow* and *gdNIT* in the "Cycle Segments" section are parsed from the fibex file. However, the MCU's FlexRay controller requires a value according to Figure 6-11. That means that for the sake of setting it correctly a derived value of the *gdNIT* parameter needs to be calculated. This value is displayed next to the regular one for convenience and represents what is actually loaded into the MCU. Moreover, the application checks for the correctness of all cycle parameters. Following equation must hold true:

$$gNumberOfStaticSlots * gdStaticSlot + dymanicSegmentOffset + gNumberOfMinislots * gdMinislot - 1 + gdSymbolWindow + gdNIT = gMacroPerCycle$$

where $dymanicSegmentOffset = gdActionPointOffset - gdMinislotActionPointOffset$

or zero if the value is negative. The application lets the user know if this equation is violated.

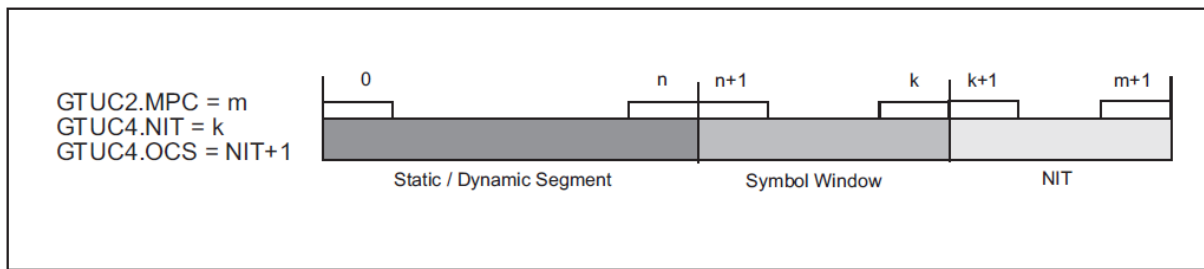


Figure 6-11: Cycle settings in the MCU

6.2.5. FPGA FlexRay controller and Testing

FlexRay controllers of the FPGA are managed from the separate window. This window can be opened after the connection with the target board has been established by selecting "Action→Show FPGA Manager". Upon connecting to the board the FPGA controllers also appear in the Cluster Setup. It is necessary to map a controller to an ECU in Cluster Setup before trying to configure its parameters. Similarly, frames for transmission need to be defined in the Message Editor before loading them into the controller (option Configure Frames).

The "FPGA Status" window lets the user select a FlexRay controller and execute actions related to that controller. These include parameters configuration, frame configuration, coldstarting, joining a running cluster, halting and running a test. The last option is out of the scope of this thesis and will be supported in the future. Once a FPGA controller is in the NORMAL_ACTIVE state it takes part in the communication according to the trigger settings of its frames. The payload of the frames can be updates in the Monitoring tab of the main window. Doing so also refreshes the flag which tells the FPGA's TX buffers to transmit the frame. This is relevant for frames that are only sent once since their flags don't automatically reset after transmission.

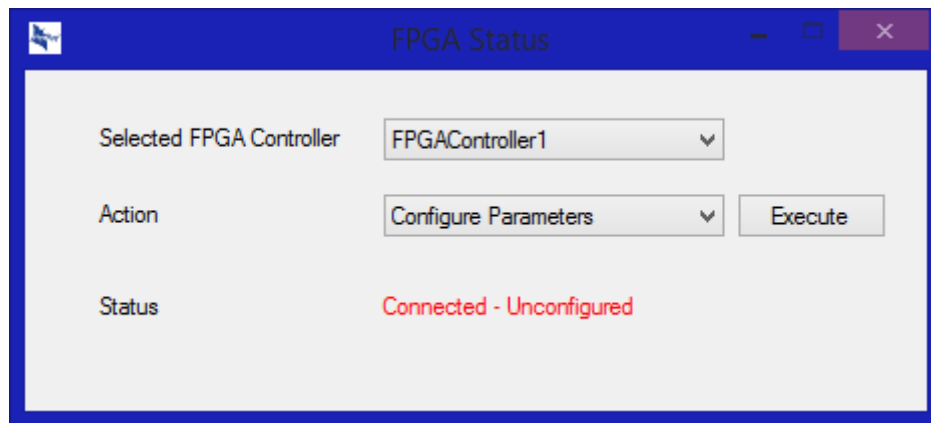


Figure 6-12: FGPA Status window

6.2.6. Target I.P. setting

Figure 6-13 shows the UI responsible for setting the target IP address and port. Those need to be input correctly before attempting to connect to the target board. The default values match the values statically set in the MCU's firmware. The application checks for validity of the input and won't allow for any violations of the IP address format.

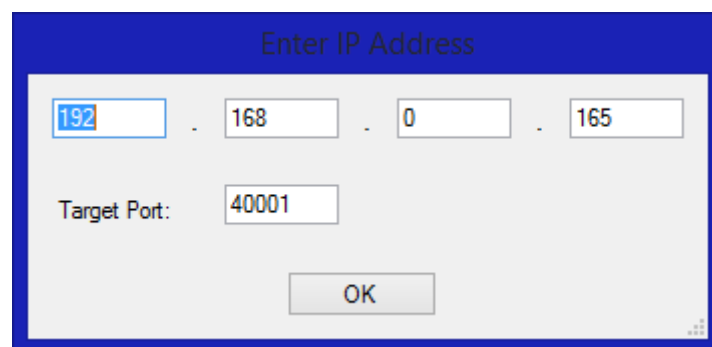


Figure 6-13: IP Address settings window

6.2.7. ECU Mapping

The mapping feature is a means of pairing a physical FlexRay controller with an ECU defined by the fibex file. All the ECUs of a cluster will appear in the ECUs section of the Cluster Setup tab as well as in the “ECU Mapping” combo boxes of the MCU and all FPGA controllers. By selecting a mapping all the node-specific parameters are automatically copied from the corresponding ECU to the controller's UI. Not all required parameters may be present in the ECUs section simply because they might not be defined by the fibex file. Those are highlighted upon selection to notify the user that they need his attention and must be filled

manually. There is no mechanism that would prevent the user from mapping two physical controllers to the same ECU as this might be his intention.



Figure 6-14: ECU Mapping example

6.2.8. Saving and loading of Cluster and MCU parameters

Apart from loading the parameters of a cluster from a fibex file it is also possible to save them into an xml file using serialization and load them when needed. Loading such a file does not have the same effect as loading a fibex file. It doesn't add Frames, Signals and ECUs defined in the network. It merely sets the values of the cluster parameters and settings for the FlexRay controller in the MCU. This can be useful for instance when the user wants to experiment with the cluster or node settings but wants to keep all the elements and parameters defined by the fibex. In this case the fibex file would be loaded first and the xml file with the alterations afterwards.

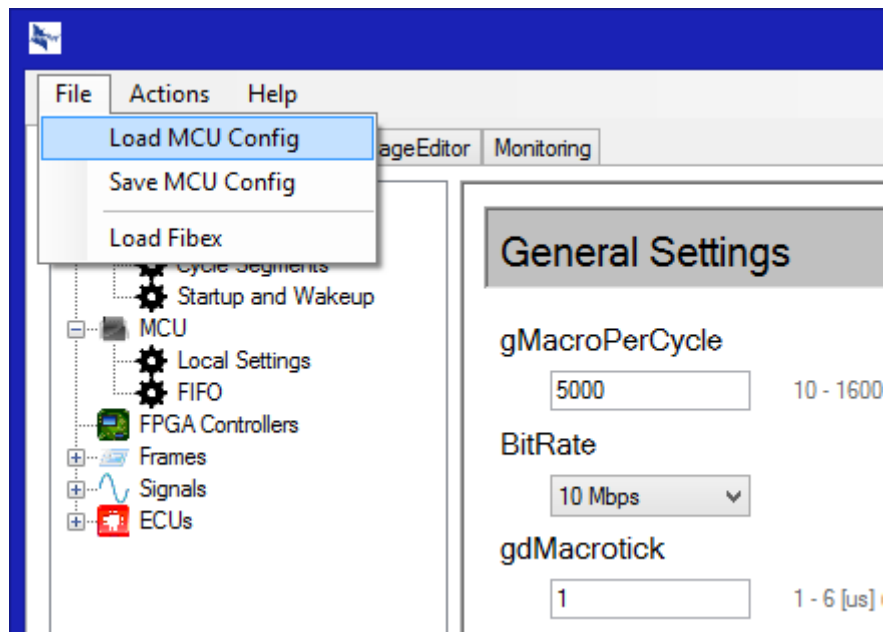


Figure 6-15: Load and Save menu options


6.2.9. Other useful features

The application offers a handful of other useful features for user convenience. Not all are listed since some of the minor features will never be observed by the user because they only manifest in special cases such as an incorrect input.

Relative and absolute time

This function belongs to the Monitoring interface. It simply switches between what kind of a timestamp is displayed in the monitoring window. The resolution of time is in milliseconds. It uses the stopwatch class of the .NET platform which is the most accurate way of measuring time it offers. The time can either be absolute measured roughly from MCU's successful integration into the cluster (or a coldstart) or from the last reset (using the Reset time button). Or it can be relative. In which case, each frame gets a value relative to the reception time of the previous frame. The timestamp marks the point in time when the frame was processed by the application so there may be significant fluctuations and deviations from cycle periodicity.

Time [ms]	Cycle	Slot ID
2	0	9
3	0	8
13	1	9
13	1	8
31	2	9
31	2	8
50	3	9
50	3	8
62	4	9
62	4	8



Delta time [ms]	Cycle	Slot ID
2	0	9
1	0	8
10	1	9
0	1	8
18	2	9
0	2	8
19	3	9
0	3	8
12	4	9
0	4	8

Figure 6-16: Absolute vs relative time

Saving of the monitoring log

Any software dealing with communication monitoring must have a feature to save the communication log for future reference and offline inspection. The application offers to save the records from the Monitoring tab to a CSV file. This simple format can easily be open by a vast number of software tools and viewed in a table form. These tools can also effortlessly convert it into different format of choice.

	A	B	C	D	E	F	G	H	I	J	K
1	TIME [MS]	DELTA TIME [MS]	CYCLE	SLOT ID	CHANNEL	STARTUP	SYNC	DYNAMIC	LENGTH [B]	DATA	ECU
2	2	2	0	9	A	Yes	Yes	No	32	d769a090d9881b0d8166a637c3	ECU4
3	3	1	0	8	A	Yes	Yes	No	32	f34e50adfec2a8a00c2b657d44	ECU2
4	13	10	1	9	A	Yes	Yes	No	32	d769a090d9881b0d8166a637c3	ECU4
5	13	0	1	8	A	Yes	Yes	No	32	f34e50adfec2a8a00c2b657d44	ECU2
6	31	18	2	9	A	Yes	Yes	No	32	d769a090d9881b0d8166a637c3	ECU4
7	31	0	2	8	A	Yes	Yes	No	32	f34e50adfec2a8a00c2b657d44	ECU2
8	50	19	3	9	A	Yes	Yes	No	32	d769a090d9881b0d8166a637c3	ECU4
9	50	0	3	8	A	Yes	Yes	No	32	f34e50adfec2a8a00c2b657d44	ECU2
10	62	12	4	9	A	Yes	Yes	No	32	d769a090d9881b0d8166a637c3	ECU4
11	62	0	4	8	A	Yes	Yes	No	32	f34e50adfec2a8a00c2b657d44	ECU2
12	80	18	5	9	A	Yes	Yes	No	32	d769a090d9881b0d8166a637c3	ECU4
13	80	0	5	8	A	Yes	Yes	No	32	f34e50adfec2a8a00c2b657d44	ECU2
14	92	12	6	9	A	Yes	Yes	No	32	d769a090d9881b0d8166a637c3	ECU4
15	92	0	6	8	A	Yes	Yes	No	32	f34e50adfec2a8a00c2b657d44	ECU2

Figure 6-17: Monitoring record from Figure 6-6 saved as CSV file and displayed in MS Excel

Resetting of monitoring time

As mentioned, the monitoring time-base can be manually reset by clicking on the "Reset time" button of the top panel in the Monitoring tab. The time-base is also reset with each new integration of the MCU FlexRay controller into a cluster.

Resizing

The graphical user interface strives to accommodate users with various screen resolutions. Therefore it is paramount that all the UI elements resize correctly to maintain their usefulness. This is ensured through a proper setting of anchors which are a feature provided by the Windows Forms API of the .NET platform.

6.3. TCP client implementation

The PC application implements the client side of the platform. A combination of threads and callbacks is used to provide a simple API to the rest of the application. Specifically, the Client class is made up of two partial classes - the Receiver and the Sender along with a method to register a callback. Instances of both of these classes are created while creating the client Class instance.

The Sender class contains a thread which constantly checks for new data to send. It provides the `SendData(byte[] data)` method which is used by the state machine to send data. Importantly, the Nagle algorithm is disabled in the client so the data is sent immediately. Otherwise it would be buffered and sent only after filling the buffers to reduce network traffic. This behavior is undesirable in case of our platform since it would disrupt the communication protocol.

The receiver class also runs a thread. It continuously checks for incoming packets. Each received frame is then converted into a small Command class instance which separates the command type from the rest of the data. The received invokes the registered callback with the command as an argument. The function registered as the callback then separates the only state-machine-independent command `AVAILABLE_TASK_RESPONSE` from others and forwards those to their intended destination which is either the main state machine or one of the FPGA state machines.

6.4. Database

The WPF graphical subsystem natively supports the separation of data and presentation. This is not the case with Windows Forms used in this project. That's why a database has to be used in order to provide a similar experience. Windows Forms were selected as an already familiar technology. WPF have a rather steep learning curve and having to get familiar with it would delay the project. Nevertheless, Windows Forms combined with a database provide a completely sufficient solution for the purposes of this platform. The database is contained in a

MDF file which basically is a SQL server data file. It is named FlexRayDB.mdf and has to be distributed with the binary of the application. It holds the following tables:

ECUs

<u>Name</u>	<u>Data Type</u>	<u>Allow Nulls</u>	<u>Default</u>
Id	int	no	none
pMicroPerCycle	int	no	none
pAllowPassiveToActive	int	no	none
pdListenTimeout	int	no	none
pWakeupPattern	int	no	none
pMicroInitialOffsetA	int	no	none
pMicroinitialOffsetB	int	no	none
pMacroInitialOffsetA	int	no	none
pMacroInitialOffsetB	int	no	none
pOffsetCorrectionOut	int	no	none
pRateCorrectionOut	int	no	none
pExternOffsetCorrection	int	no	none
pExternRateCorrection	int	no	none
pClusterDriftDamping	int	no	none
pDecodingCorrection	int	no	none
pdAcceptedStartupRange	int	no	none
pdMaxDrift	int	no	none
pDelayCompensationA	int	no	none
pDelayCompensationB	int	no	none
pLatestTx	int	no	none

Table 6-1: Contents of the ECUs table

This table is responsible for storing the values of node constants for all ECUs. It is first loaded with values parsed from the fibex file. Changes made through the GUI are also saved.

FPGAs

<u>Name</u>	<u>Data Type</u>	<u>Allow Nulls</u>	<u>Default</u>
Id	int	no	none
pMicroPerCycle	int	no	none
InterfaceMapping	int	no	none
pAllowPassiveToActive	int	no	none

pdListenTimeout	int	no	none
pdCASRxLowMax	int	no	none
pWakeupPattern	int	no	none
pMicroInitialOffsetA	int	no	none
pMicroInitialOffsetB	int	no	none
pMacroInitialOffsetA	int	no	none
pMacroInitialOffsetB	int	no	none
pOffsetCorrectionOut	int	no	none
pRateCorrectionOut	int	no	none
pExternOffsetCorrection	int	no	none
pExternRateCorrection	int	no	none
pClusterDriftDamping	int	no	none
pDecodingCorrection	int	no	none
pdAcceptedStartupRange	int	no	none
pdMaxDrift	int	no	none
pDelayCompensationA	int	no	none
pDelayCompensationB	int	no	none

Table 6-2: Contents of the FPGAs table

This table contains all the information about parameter values for every FPGA FlexRay controller. When selecting a FPGA controller the data is fetched from the table and filled into the UI.

Frames

<u>Name</u>	<u>Data Type</u>	<u>Allow Nulls</u>	<u>Default</u>
Id	int	no	none
ShortName	nvarchar(50)	no	none
FrameType	nvarchar(50)	no	none
ByteLength	int	no	none
SlotId	int	no	none
SourceECU	nvarchar(50)	no	none
CyclePeriod	int	no	none
BaseCycle	int	no	none
Mapping	nvarchar(50)	no	"None"
MappingType	nvarchar(50)	no	none
Monitored	nvarchar(50)	no	"No"
ExtendedFrame	nvarchar(50)	no	none
Channel	nvarchar(50)	no	none

Table 6-3: Contents of the Frames table

The Frames table is filled by values parsed from the fibex file. Additional parameters are set from the UI such as *Mapping*, *MappingType* and *Monitored*.

Signals

<u>Name</u>	<u>Data Type</u>	<u>Allow Nulls</u>	<u>Default</u>
Id	int	no	none
ShortName	nvarchar(50)	no	none
FrameName	nvarchar(50)	no	none
BitLength	int	no	none
BitPosition	int	no	none
Coding	nvarchar(50)	no	none

Table 6-4: Contents of the Signals table

The *FrameName* value from the Signals table is used to identify all signals belonging to a certain frame.

Triggers

Name	Data Type	Allow Nulls	Default
Id	int	no	none
FrameName	nvarchar(50)	no	none
TriggerType	nvarchar(50)	no	none
Timestamp	bigint	yes	none
Macrotick	int	yes	none
Cycle	int	yes	none

Table 6-5: Contents of the Triggers table

The Triggers table specifically relates to the FPGA TX frames. The decision to store this information in a database rather than in the Message Editor table was made to keep the Message Editor neatly arranged. Including this information would require the addition of extra columns that would be irrelevant for MCU frames. The Timestamp entry necessitates the use of bigint due to its size (64 bits).

7. Communication protocol

7.1. Purpose of the protocol

Since the testing platform is composed of a target board and a PC application it is essential to have a form of communication in place between the two parts. All of the platform's functions are controlled from the PC application which then manages the platform using the communication protocol.

7.2. Requirements

First and foremost the protocol must be based on a standard capable of a throughput high enough to handle reporting of FlexRay frames in real-time. The maximum throughput of FlexRay is 20 Mbit/s taking into consideration the two independent channels (10 Mbit/s per channel). This basically leaves two interfaces that a standard PC has - USB and Ethernet. TCP/IP has been opted for due to previous experience with a high-quality, lightweight open-source TCP/IP stack for embedded systems - the lwIP stack. This choice also opens a lot of potential options to choose from in the area of RPC protocols.

The architecture dictates that the MCU assumes the role of a server and the application acts as a client. Only a single client is required. The protocol is responsible for configuring

MCU's peripherals, managing user tasks, reporting received frames etc. A new FPGA design can be loaded into the FPGA from the application through the MCU.

Taking into account that the frequency of the MCU's core is rather low (160MHz) a protocol with an overhead as little as possible is preferred. Bearing in mind that the system is likely to be extended in the future, all of the RCP protocols were rejected. Protocols such as SOAP bring too much of a parsing overhead. JSON-RPC was a serious candidate but no good open-source embedded server implementation was found. Another downside of JSON-RPC is that the communication is always initiated by the client. This presents an issue since in case of this platform the server needs to retransmit the FlexRay frames as they come. To work around this the server would have to be regularly polled by the client. Such a solution is clumsy at best. This leaves binary-based alternatives. Since the protocol has to be binary anyway a decision has been made to design a custom protocol instead of using an existing standard which would require porting of a third-party code.

A vital point is to design the protocol as simple as possible while keeping it extensible. A compromise has been found between the simplicity of parsing and the efficiency of channel utilization. To keep the protocol extensible the command type is coded into one byte which leaves a plenty of free values to use in the future. Extensibility is also considered in the sense of adding support for other interfaces than FlexRay. This can easily be achieved in the same way that the FPGA FlexRay controllers are differentiated from the MCU FlexRay controllers by spawning separate state machines in the PAIRED_AND FIBEX_LOADED state.

7.3. Negotiation of supported functions

It has been requested that the protocol should be capable of enabling the server to report its release version and also its supported functionality. This is implemented by the SUPPORTED_COMMANDS_AND_VERSION_REQUEST and SUPPORTED_COMMANDS_AND_VERSION_RESPONSE commands during pairing of the devices. See 7.4 Message format section for details.

7.4. Message format

This section describes the message format of the communication protocol in detail. The endianness of the two platform's parts needs to be taken into account. The PC application uses little endian and the MCU uses big endian. This is irrelevant when assembling the commands byte by byte but it comes into effect when examining any larger data types by bytes. An example of this would be the MCU_FR_CONFIG message.

Each command starts with a command type coded in the first byte. This first byte is implicit and won't be depicted. Table 7-1 shows all command codes and Table 7-5 all response codes. Some commands and responses contain only the first byte which codes the meaning of it. Others have data following after. This is expressed in the table. Only commands and responses with additional data are described in the following section in detail.

7.4.1. PC to MCU

<u>Command type</u>	<u>Code value</u>	<u>Contains Data</u>
CONFIG_READOUT_REQUEST	0x01	No
SUPPORTED_COMMANDS_AND_VERSION_REQUEST	0x02	No
MCU_FR_CONFIG	0x03	Yes
MCU_SLOT_DEFINITION	0x04	Yes
MCU_TX_DATA_DEFINITION	0x05	Yes
MCU_TX_DATA_DEFINITION_DONE	0x06	No
MCU_GO_TO_READY	0x07	No
MCU_SEND_WAKEUP	0x08	No
MCU_TX_FRAME_DATA_UPDATE	0x09	Yes
MCU_COLDSTART	0x0A	No
MCU_JOIN	0x0B	No
MCU_HALT	0x0C	No
MCU_FREEZE	0x0D	No
FPGA_FR_CONFIG	0x0E	Yes
FPGA_TX_FRAME_DATA_DEFINITION	0x0F	Yes
FPGA_TX_FRAME_DATA_UPDATE	0x10	Yes
FPGA_COLDSTART	0x11	Yes
FPGA_JOIN	0x12	Yes
FPGA_HALT	0x13	Yes
FPGA_RUN_TEST	0x14	Yes
AVAILABLE_TASKS_REQUEST	0x15	No
RUN_TASK	0x16	Yes
SUSPEND_TASK	0x17	Yes
SEND_FPGA_DESIGN	0x18	Yes
LOAD_FPGA_DESIGN_FROM_FLASH	0x19	No

Table 7-1: PC to MCU command table

MCU_FR_CONFIG

<u>Parameter</u>	<u>Bit field</u>	<u>Bit position</u>	<u>Range</u>	<u>Units</u>
pKeySlotUsedForStartup	SUCC1.TXST	8	0/1	-
pKeySlotUsedForSync	SUCC1.TXSY	9	0/1	-
gColdStartAttempts	SUCC1.CSA(4-0)	15.11	2.31	-
pAllowPassiveToActive	SUCC1.PTA(4-0)	20-16	0-31	-
pChannels	SUCC1.CCHA	26	0/1	-
	SUCC1.CCHB	27	0/1	-
pdListenTimeOut	SUCC2.LT(20-0)	20-0	1284- 1283846	μT
gListenNoise	SUCC2.LTN(3-0)	27-24	2.15	μT
gMaxWithoutClockCorrectionPassive	SUCC3.WCP(3-0)	3-0	1.15	-
gMaxWithoutClockCorrectionFatal	SUCC3.WCF(3-0)	7.4	1.15	-
gNetworkManagementVectorLength	NEMC.NML(3-0)	3-0	0-12	Bytes
gdTSSTransmitter	PRTC1.TSST(3-0)	3-0	3.15	Bit times
gdCASRxLowMax	PRTC1.CASM(6-0)	10.4	67-99	Bit times
gdSampleClockPeriod	PRTC1.BRP(1-0)	15-14	0-3	-
pSamplesPerMicrotick	PRTC1.BRP(1-0)		0-3	-
gdWakeupSymbolRxWindow	PRTC1.RXW(8-0)	24-16	76-301	Bit times
pWakeupPattern	PRTC1.RWP(5-0)	31-26	2.63	-
gdWakeupSymbolRxIdle	PRTC2.RXI(5-0)	5-0	14-55	Bit times
gdWakeupSymbolRxLow	PRTC2.RXL(5-0)	13.8	10.55	Bit times
gdWakeupSymbolTxIdle	PRTC2.TXI(7-0)	23-16	45-180	Bit times
gdWakeupSymbolTxLow	PRTC2.TXL(5-0)	29-24	15-60	Bit times
gPayloadLengthStatic	MHDC.SFDL(6-0)	6-0	0-127	16bit
pLatestTx	MHDC.SLT(12-0)	28-16	0- 7981	minislots
pMicroPerCycle	GTUC1.UT(19-0)	19-0	640- 640000	μT
gMacroPerCycle	GTUC2.MPC(13-0)	13-0	10-16000	MT
gSyncNodeMax	GTUC2.SNM(3-0)	19-16	2.15	Frames
pMicroInitialOffset[A]	GTUC3.UIOA(7-0)	7-0	0-240	μT
pMicroInitialOffset[B]	GTUC3.UIOB(7-0)	15.8	0-240	μT
pMacroInitialOffset[A]	GTUC3.MIOA(6-0)	22-16	2.72	MT
pMacroInitialOffset[B]	GTUC3.MIOB(6-0)	30-24	2.72	MT
gdNIT	GTUC4.NIT(13-0)	13-0	7- 15997	MT
gOffsetCorrectionStart	GTUC4.OCS(13-0)	29-16	8- 15998	MT
pDelayCompensation[A]	GTUC5.DCA(7-0)	7-0	0-200	μT
pDelayCompensation[B]	GTUC5.DCB(7-0)	15.8	0-200	μT
pClusterDriftDamping	GTUC5.CDD(4-0)	20-16	0-20	μT
pDecodingCorrection	GTUC5.DEC(7-0)	31-24	14-143	μT
pdAcceptedStartupRange	GTUC6.ASR(10-0)	10-0	0-1875	μT
pdMaxDrift	GTUC6.MOD(10-0)	26-16	2.23	μT



Figure 7-2: MCU slot definition message format - second word

MCU_TX_DATA_DEFINITION

1 = Periodic, 0 = Single

The two bytes are followed by the payload data as an array of bytes.

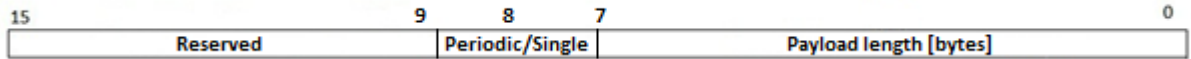


Figure 7-3: MCU Tx Data Definition Message Format

MCU_TX_FRAME_DATA_UPDATE

The Slot ID and Cycle code are necessary to identify the exact frame to update because there may be multiple frames with the same Slot ID but different Cycle codes.

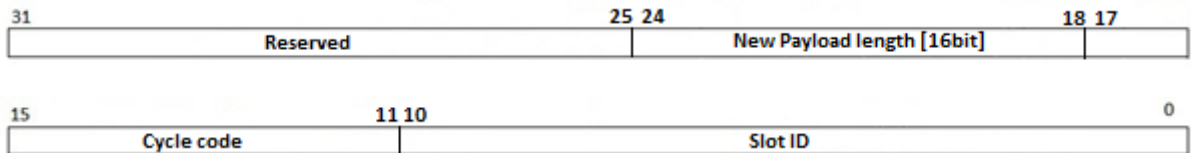


Figure 7-4: MCU Tx Data Update Message Format

FPGA_FR_CONFIG

Parameter	Offset	Value Range
gdActionPointOffset	0x02	0 to 63
gdStaticSlot	0x06	0 to 661
gMacroPerCycle	0x0A	0 to 16000
gNumberOfStaticSlots	0x0E	0 to 1023
gOffsetCorrectionStart	0x13	0 to 15999
pDecodingCorrection	0x17	0 to 143
pdMaxDrift	0x1B	0 to 1923
pMacroInitialOffsetA	0x1F	0 to 68
pMacroInitialOffsetB	0x24	0 to 68
pMicroPerCycle	0x28	0 to 640000
pOffsetCorrectionOut	0x2C	0 to 15567
pRateCorrectionOut	0x31	0 to 1923
gdSampleClockPeriod	0x35	0 to 7
pClusterDriftDamping	0x39	0 to 20

gdTssTransmitter	0x3D	0 to 15
pMicroInitialOffsetA	0x42	0 to 239
pMicroInitialOffsetB	0x46	0 to 239
pdAcceptedStartupRange	0x4A	0 to 1875
pDelayCompensationA	0x4E	0 to 200
pDelayCompensationB	0x53	0 to 200
pSamplesPerMacrotick	0x57	1 to 7
gdCasrxLowMax	0x5B	0 to 99
gdWakeupSymbolTxLow	0x5F	0 to 6
gdWakeupSymbolTxIdle	0x64	0 to 180
gdWakeupSymbolRxLow	0x68	0 to 60
gdWakeupSymbolRxIdle	0x6C	0 to 180
gdWakeupSymbolRxWindow	0x71	0 to 301
pWakeupPattern	0x75	0 to 63
pdListenTimeout	0x79	1284 to 1283846
vColdstartAttempts	0x7D	2 to 31
gMaxWithoutClockCorrectionPassive	0x82	1 to 15
gMaxWithoutClockCorrectionFatal	0x86	1 to 15
pAllowPassiveToActive	0x8A	1 to 31
externRateControl	0x8E	0 to 15567
externOffsetControl	0x93	0 to 1923

Table 7-3: FPGA FlexRay parameters

All commands related to the FPGA FlexRay controllers begin with the command code in the first byte followed by the controller index in the second byte. Table 7-3 shows how the parameters are put into the packet following the index. Each parameter is sent as a 4-byte word.

FPGA_TX_FRAME_DATA_DEFINITION

A command of this type is sent for each frame that a FPGA controller is supposed to transmit. In every case such a command starts with the word (first after command type and controller index) in Figure 7-5.

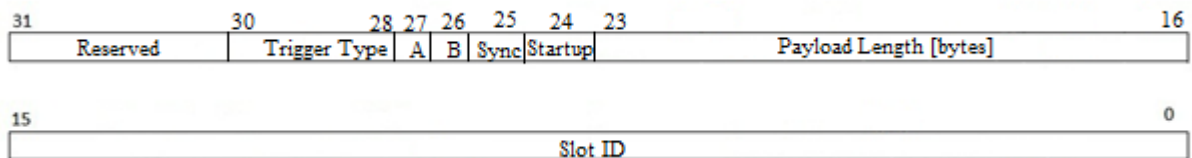


Figure 7-5: FPGA TX frame data definition message format - first word

The value of the Trigger Type field determines the format of the rest of the frame according to this table:

<u>Trigger Type value</u>	<u>Meaning</u>	<u>Followed by</u>
0	Single message	Data
1	Every Cycle	Data
2	Every Odd Cycle	Data
3	Every Even Cycle	Data
4	Timestamp	Timestamp low and high (Figure 7-6) and then data
5	Macrotick	Macrotick word (Figure 7-7) and then data
6	Macrotick and cycle	Macrotick and cycle word (Figure 7-8) and then data

Table 7-4: Trigger Type values

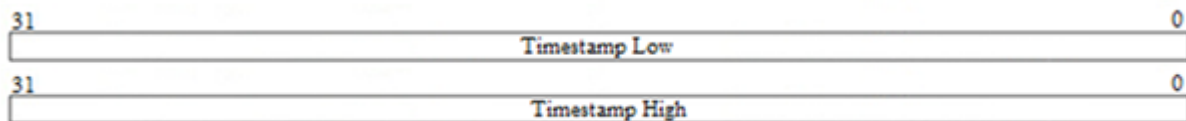


Figure 7-6: FPGA TX frame data definition message format - Timestamp - second and third word

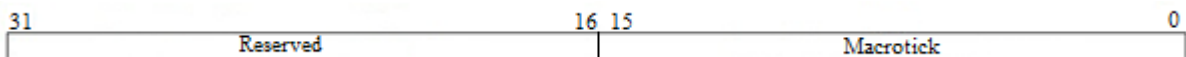


Figure 7-7: FPGA TX frame data definition message format - Macrotick - second word

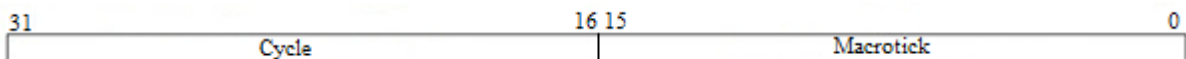


Figure 7-8: FPGA TX frame data definition message format - Macrotick and Cycle - second word

FPGA_TX_FRAME_DATA_UPDATE

The Slot ID is needed to identify the TX buffer in the FPGA in which we want to update the data

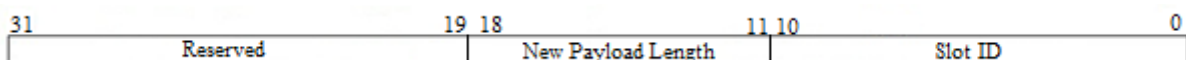


Figure 7-9: FPGA TX frame data update message format

FPGA_COLDSTART, FPGA_JOIN and FPGA_FREEZE

All of these commands only have one other index byte which identifies the target FPGA controller.

FPGA_RUN_TEST

This feature is not yet supported. The platform is prepared for this feature to be implemented in the future.

RUN_TASK and SUSPEND_TASK

Either command carries only one byte with the index of the task to run or suspend.

SEND_FPGA_DESIGN_START

By sending this command the application signals the start of sending a new FPGA configuration. First the command type is followed by a uint32 value of the design's total length in bytes. Afterwards comes the number of packets in which is the data going to be fragmented.

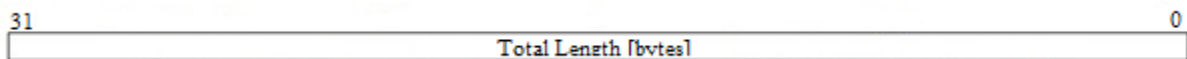


Figure 7-10: Send FPGA design message format - bytes 0 to 3

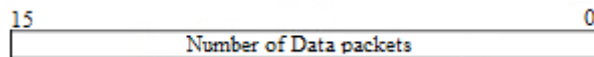


Figure 7-11: Send FPGA design message format - bytes 4 to 5

SEND_FPGA_DESIGN_DATA

The command code is followed by a uint16 value of an ordinal number. The MCU checks this number upon reception whether it is equal to the previous one increased by one. The data as an array of bytes comes after.

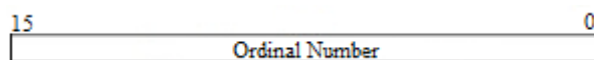


Figure 7-12: Send FPGA design data message format

7.4.2. MCU to PC

<u>Response type</u>	<u>Code value</u>	<u>Contains Data</u>
CONFIG_READOUT_RESPONSE	0x01	Yes
SUPPORTED_COMMANDS_AND_VERSION_RESPONSE	0x02	Yes
MCU_FR_CONFIG_FAIL	0x03	No
MCU_FR_PARAM_CONFIRMATION	0x04	No
MCU_FR_SLOT_INFO_CONFIRMATION	0x05	No
MCU_FR_FRAMES_PARTIAL_CONFIRMATION	0x06	No
MCU_FR_FRAMES_DONE_CONFIRMATION	0x07	No
FPGA_FR_CONFIG_CONFIRMATION	0x08	Yes
AVAILABLE_TASKS_RESPONSE	0x09	Yes
FPGA_DESIGN_CONFIRMATION	0x0A	No
FPGA_DESIGN_FROM_FLASH_CONFIRMATION	0x0B	No
FR_DATA	0x0C	Yes
STATE_MACHINE_ERROR	0x0D	No
CONSOLE_DATA	0x0E	Yes
FPGA_TEST_FINISHED	0x0F	Yes
MCU_STARTUP_SUCCESS	0x10	No
MCU_STARTUP_FAIL	0x11	No
FPGA_STARTUP_SUCCESS	0x12	Yes
FPGA_STARTUP_FAIL	0x13	Yes
FPGA_TEST_STARTED	0x14	Yes
FPGA_DESIGN_FAILED	0x15	No
FPGA_NEW_READOUT	0x16	Yes
FPGA_BUFF_CONFIG_CONF	0x17	Yes

Table 7-5: MCU to PC response table

CONFIG_READOUT_RESPONSE

The response has only one byte with the number of FlexRay controllers in the FPGA.

SUPPORTED_COMMANDS_AND_VERSION_RESPONSE

The three bytes in Figure 7-13 are followed by a number of bytes defined in the "Number of commands" section. Each byte contains a supported command code from Table 7-6.

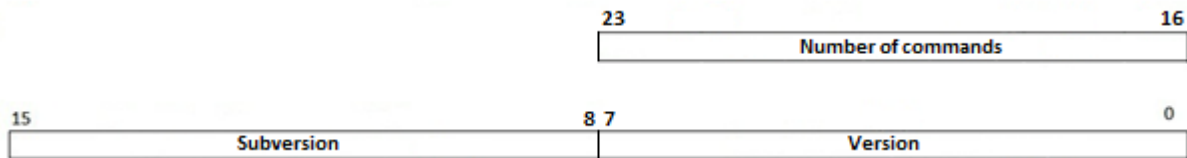


Figure 7-13: Supported Commands and Version Response message format

AVAILABLE_TASKS_RESPONSE

This response contains a list of available tasks as an array of characters. Tasks are separated by the '|' character. It is therefore forbidden from being used in task's name. Here is an example with three tasks:

"Task1|Task2|Task3|"

Each character is ASCII coded in one byte.

FR_DATA

The word in Figure 7-14 is followed by $\text{ceiling}(\frac{PLR}{2})$ bytes of data. This message reports received FlexRay frames to the application.

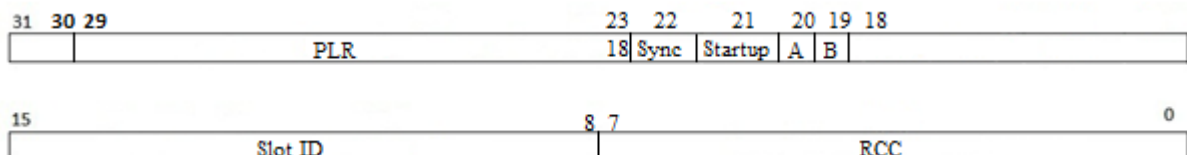


Figure 7-14: FlexRay Data Message Format

CONSOLE_DATA

This feature is not yet supported. The platform is prepared for this feature to be implemented in the future to display the messages during testing.

FPGA_NEW_READOUT

A message with the new number of FlexRay controllers in the FPGA is sent right after the last FPGA_FR_CONFIG_CONFIRMATION. The new number of the only payload byte.

FPGA Responses

All these commands contain only one byte identifying the source FPGA controller:

- FPGA_TEST_FINISHED
- FPGA_TEST_STARTED
- FPGA_STARTUP_SUCCESS
- FPGA_STARTUP_FAIL
- FPGA_FR_CONFIG_CONFIRMATION
- FPGA_BUFF_CONFIG_CONF

8. Conclusion

The objective of this thesis was to develop firmware and software for a platform custom designed for the testing of automotive communication networks. Specifically, this thesis only focuses on the FlexRay standard. From a hardware perspective the platform is, however, capable of communicating over CAN and LIN as well.

The system as a whole had to enable a remote management of tasks. Additionally, the possibility to remotely reconfigure the FPGA's design was required. The target platform of the control application is MS Windows. The application is responsible for managing the functionality of the whole platform. This includes the ability to monitor communication in a FlexRay cluster as well as to transmit arbitrary frames of data. Moreover, it was required that the application could manage FlexRay controllers in the FPGA.

On the firmware side a real-time operating system was to be used. The firmware had to be designed to support all the platform's features and acted as a server for the client Windows application.

All the components from various projects were successfully integrated into the platform. Namely, the FlexRay controllers from [14] wrapped in a design from [16]. Hardware of the platform [12] has proven to be fully functional. Performance of the entire system met expectations. However, shortcomings of the TMS570 microcontroller were discovered during the development of this thesis. Its extreme sensitivity caused multiple samples to malfunction. It would therefore be advisable not to use this particular microcontroller in the future.

FreeRTOS was chosen as the real-time operating system for this project. Ultimately, this was a satisfactory choice. FreeRTOS was capable of providing the exact set of features to fulfill the platform's needs. It also met all requirements performance-wise.

A decision was made to use TCP as the basis for communication between the server (firmware) and the client (Windows application) of the system. TCP was selected for its throughput, reliability and simplicity. The open-source lwIP stack used in this thesis was successfully ported for this purpose and met our requirements. A custom binary-based communication protocol was designed between the server and the client.

The Windows application offered a number of additional features for user's convenience. It utilized the .NET platform which limited the portability of the code. Nevertheless, it offered a rich API that facilitated the parsing of fibex files and provided an easy access to the local database file.

During the development of this thesis a couple of ideas for future improvement presented themselves. First of all, it was found that the platform could be ready for the implementation of testing sequences for which the FlexRay controllers in the FPGA were specifically designed. The tests were, however, out of the scope of this thesis. Moreover, the option to trigger on certain conditions could be added Monitoring tab. The task management feature could be expanded by adding configurable priorities and by the possibility to pass user data to the tasks through queues.

References

- [1] Malinský, J.: Intrusive Tests in FlexRay Standard, doktorská disertační práce ČVUT 2009
- [2] FlexRay Consortium, FlexRay Protocol Specification V2.1 Rev. A, 2005
- [3] IEEE standard 802.3 – 2008
- [4] Texas Instruments Inc., TMS570LS31x/21x 16/32-Bit RISC Flash Microcontroller, Technical Reference Manual, 2011
- [5] Robert Bosch GmbH, E-Ray FlexRay IP Module, Application Note AN002 Startup, 2007
- [6] FlexRay Consortium, FlexRay Protocol Specification V2.1 – errata sheet, 2005
- [7] FlexRay Consortium, FlexRay Electrical Physical Layer Specification V2.1, 2005

- [8] Pokorný V.: Metody měření vybraných parametrů komunikačního standardu FlexRay a jejich implementace, diplomová práce ČVUT 2007
- [9] Richard Barry, Using The freeRTOS Real Time Kernel, A practical guide, PIC32 Edition, 2009
- [10] Association for Standardisation of Automation and Measuring Systems, Field Bus Exchange Format, Version 3.0, 2008
- [11] Zeman M.: Firmware of Ethernet/FlexRay Gateway ,bachelor's thesis CTU Faculty of Electrical Engineering 2012
- [12] Blecha J.: Programmable test platform, master's thesis CTU 2014
- [13] lwIP stack documentation, http://lwip.wikia.com/wiki/LwIP_Wiki
- [14] Paták M.: Methods for Testing of the FlexRay Startup Mechanism, master's thesis CTU Faculty of Electrical Engineering 2012
- [15] Schäuffele, J., Zurawka, T.: Automotive Software Engineering, SAE International 2005, ISBN 0-7680-1490-5
- [16] Ille, O.: Programové vybavenie testovacej platformy, bakalářská práce CTU Faculty of Electrical Engineering, 2014