

SOFTWARE TOOL DEVELOPMENT FOR THE AUTOMATED CONFIGURATION OF
FLEXRAY
NETWORKS FOR IN-VEHICLE COMMUNICATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

CAN ÖZTÜRK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

JANUARY 2013

Approval of the thesis:

**SOFTWARE TOOL DEVELOPMENT FOR THE AUTOMATED CONFIGURATION OF
FLEXRAY NETWORKS FOR IN-VEHICLE COMMUNICATION**

Submitted by **CAN ÖZTÜRK** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. İsmet Erkmn
Head of Department, **Electrical and Electronics Engineering**

Assoc. Prof. Dr. Şenan Ece Schmidt
Supervisor, **Electrical and Electronics Engineering Dept., METU**

Assoc. Prof. Dr. Klaus Werner Schmidt
Co-supervisor, **Mechatronics Engineering Dept., Çankaya University**

Examining Committee Members:

Prof. Dr. Semih Bilgen
Electrical and Electronics Engineering Dept., METU

Assoc. Prof. Dr. Şenan Ece Schmidt
Electrical and Electronics Engineering Dept., METU

Assoc. Prof. Dr. Cüneyt Bazlamaçcı
Electrical and Electronics Engineering Dept., METU

Assoc. Prof. Dr. Halit Oğuztüzün
Computer Engineering Dept., METU

M.Sc. Vakkas Çelik
TÜBİTAK UZAY

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Can ÖZTÜRK

Signature :

ABSTRACT

SOFTWARE TOOL DEVELOPMENT FOR THE AUTOMATED CONFIGURATION OF FLEXRAY NETWORKS FOR IN-VEHICLE COMMUNICATION

Öztürk, Can

M.Sc., Department of Electrical and Electronics Engineering

Supervisor: Assoc. Prof. Dr. Şenan Ece Schmidt

Co-supervisor: Assoc. Prof. Dr. Klaus Werner Schmidt

January 2013, 58 pages

The increasing use of electronic components in today's automobiles demands more powerful in-vehicle network communication protocols. FlexRay protocol, which is expected to be the de-facto standard in the near future, is a deterministic, fault tolerant and fast protocol designed for in vehicle communication. For proper operation of a FlexRay network the communication schedule needs to be computed and the nodes need to be configured before startup. Current software tools that are geared towards FlexRay only deal with the configuration process. The schedule needs to be computed by a network designer manually and it is necessary to input the designed schedule and the configurable parameters by hand. This thesis improves upon a previous scheduling software to automatically compute the network schedule, and then generate a universally acceptable FIBEX file that can be imported to available software tools to produce the necessary FlexRay node configuration files.

Keywords: In vehicle communication, FlexRay, automatic configuration, tool chain

ÖZ

ARAÇ İÇİ İLETİŞİMDE FLEXRAY AĞLARININ OTOMATİK OLARAK YAPILANDIRILMASI İÇİN YAZILIM ARACI GELİŞTİRİLMESİ

Öztürk, Can

Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü

Tez Yöneticisi: Doç. Dr. Şenan Ece Schmidt

Ortak Tez Yöneticisi: Doç. Dr. Klaus Werner Schmidt

Ocak 2013, 58 sayfa

Günümüz otomobillerinde artan elektronik birim kullanımı daha güçlü araba içi haberleşme protokollerine olan ihtiyacı doğurmaktadır. FlexRay protokolü ortaya çıkan bu ihtiyacı karşılayabilecek özelliklere sahip, gerekirci, hatalara dayanıklı ve hızlı bir haberleşme protokolüdür. FlexRay ağlarının doğru şekilde çalışabilmesi için çalıştırılma aşamasından önce ağın iletişim çizelgesinin hesaplanması ve FlexRay birimlerinin yapılandırılması gerekmektedir. FlexRay ağlarına yönelik var olan yazılımlar sadece yapılandırma görevini yerine getirebilmektedirler. Zaman çizelgesinin ağ tasarımcısı tarafından el ile hesaplanması ve oluşturulan çizelge ile hesaplanan ağ parametrelerinin el ile bu yazılımlara girilmesi gerekmektedir. Bu tez çalışmasında önceden var olan bir çizelge oluşturma yazılımı üstünde geliştirmeler yapılarak ağ çizelgesini otomatik olarak oluşturan ve bütün ağ yapılandırmasını diğer yazılımlara yüklenilebilen bir FIBEX dosyası derleyen bir yazılım aracı geliştirilmektedir.

Anahtar Kelimeler: Araç içi haberleşme, FlexRay, otomatik yapılandırma, araç zinciri

ACKNOWLEDGMENTS

I wish to thank, first and foremost, to my supervisor Assoc. Prof. Dr. Şenan Ece Schmidt for her immensely considerate supervision, unerring suggestions and mainly for keeping my spirits high.

I thank my co-supervisor Assoc. Prof. Dr. Klaus Werner Schmidt for guiding me through the path of the programmer, and teach me the trade secrets of writing a thesis work.

I would also like to thank my parents for carrying whatever burden they could during this process to make it unfold as smooth and less painful as possible.

TABLE OF CONTENTS

| | |
|--|-------------|
| ABSTRACT | v |
| ÖZ | vi |
| ACKNOWLEDGMENTS..... | vii |
| TABLE OF CONTENTS..... | viii |
| LIST OF TABLES..... | x |
| LIST OF FIGURES..... | xi |
| CHAPTERS | |
| 1.INTRODUCTION | 1 |
| 2.RELATED WORK..... | 3 |
| 2.1 In Vehicle Networking..... | 3 |
| 2.2 FlexRay Standard..... | 5 |
| 2.2.1 Protocol Specific Parameters | 9 |
| 2.2.2 Schedule Specific Input Parameters | 12 |
| 2.2.3 Schedule Dependent Parameters | 14 |
| 2.3 Configuration of FlexRay | 15 |
| 2.4 File and Software Standards | 17 |
| 2.5 Current FlexRay Configuration Tools | 18 |
| 2.6 Shortcomings of the Current Design Workflow..... | 18 |
| 3.AUTOMATED CONFIGURATION TOOL CHAIN DEVELOPMENT..... | 21 |
| 3.1 The Tool-Chain Workflow | 21 |
| 3.2 The Scheduler | 23 |
| 3.2.1 The Existing Infrastructure | 23 |
| 3.2.2 Improvements and Additions | 23 |
| 3.3 The FIBEX Project..... | 27 |
| 3.3.1 GUID References | 30 |
| 3.3.2 Creation of Network Elements..... | 31 |
| 3.3.3 Creating Frame Triggerings (Timings)..... | 36 |
| 3.3.4 Writing the FIBEX File | 42 |
| 4.TESTS AND EVALUATION | 45 |
| 4.1 Experiment Setups and Parameters | 45 |
| 4.2 Experiment with Three FlexRay Nodes..... | 45 |
| 4.3 Experiment with One Removed ECU | 52 |
| 5.CONCLUSION AND FUTURE WORK..... | 55 |
| REFERENCES | 57 |

LIST OF TABLES

TABLES

| | |
|---|----|
| Table 2-1: FlexRay Protocol constants [3]. | 9 |
| Table 2-2: Configurable parameters that are unrelated to schedule computation, that are considered fixed for our study [3]. | 12 |
| Table 2-3: FlexRay parameters that are dependent on the application and are configured by user input [3]. | 13 |
| Table 2-4: Parameters that are computed by the scheduler [3]. | 14 |
| Table 4-1: Schedule specific input parameter values used in the experiments. | 44 |
| Table 4-2: The signal set used in the first experiment. | 46 |
| Table 4-3: Table depicting the static schedule as computed by <code>ComputeScheduleForFibex</code> . | 48 |
| Table 4-4: The signal set used in the second experiment. | 53 |

LIST OF TABLES

TABLES

| | |
|---|----|
| Table 2-1: FlexRay Protocol constants [3]. | 9 |
| Table 2-2: Configurable parameters that are unrelated to schedule computation, that are considered fixed for our study [3]. | 12 |
| Table 2-3: FlexRay parameters that are dependent on the application and are configured by user input [3]. | 13 |
| Table 2-4: Parameters that are computed by the scheduler [3]. | 14 |
| Table 4-1: Schedule specific input parameter values used in the experiments. | 44 |
| Table 4-2: The signal set used in the first experiment. | 46 |
| Table 4-3: Table depicting the static schedule as computed by <code>ComputeScheduleForFibex</code> . | 48 |
| Table 4-4: The signal set used in the second experiment. | 53 |

CHAPTER 1

INTRODUCTION

The electronic systems used in automobiles need to exchange information among each other. These systems are made up of Electronic Control Units (ECU) transmitting data signals that are packed into messages [1]. Messages can be periodic or sporadic. Periodic messages are generated at fixed generation time periods where sporadic messages are generated during certain events, such as pushing a button. The exchange of these messages among ECUs is governed by in vehicle networks rather than point to point links. The amount of increasing ECUs and the number of exchanged messages signify the importance of in vehicle networking, as direct links mean more cabling, physical weight, cost and complexity as opposed to a shared medium network [1].

In-vehicle networks can be classified in two types: Event triggered networks and time triggered networks. Controller Area Network (CAN) [2] is an event triggered network standard, which is currently the most widely used network in current automobiles [1]. FlexRay [3], however, is a standard that employs both event and time triggered networking, and is gradually replacing CAN as a widespread network type due to its support of deterministic network access and a higher data rate than CAN.

In vehicle networks as well as ECUs need to be configured in the vehicle design phase. That is, no further configuration change can be performed after startup. The configuration depends on the chosen network and the required signal transmissions and is usually performed manually by an experienced network designer. The result of this configuration is denoted as the schedule and is then implemented on the ECU by configuring the relevant registers and setting the desired transmission times. In the process from specifying the signal set to be transmitted on the in-vehicle network to implementing the network configuration on the ECUs, the following steps have to be carried out in the network design workflow.

1. Packing signals into message frames according to the network specification. The result of this step is a set of messages to be transmitted on the network
2. Computing a schedule for the message set. As the result of this step, all configuration parameters of the network are found.
3. Translating the configuration parameters into a format that can be used to implement the network configuration on the ECU hardware. The result of this step is a network that should operate as designed.
4. Verifying the correct operation of the configured network

There are several tools available to help configuring ECUs [10,11,12,13,14]. Such tools are provided with the assumption that the schedule and related configuration parameters are already determined by a human network designer. That is, the support of such tools starts from step 3. of the network design workflow with manually entering each configuration parameter. In addition, errors and warnings are given by such tools in case wrong parameters are entered or recommended

settings are not met. These tools are then able to carry out step 3. of the network design workflow by generating a standardized file which can then be used in the compilation of machine code to be loaded on the ECUs.

It is readily observed that, although available configuration tools aid the network designer in implementing the network configuration, the main design task of finding a message schedule and determining the relevant configuration parameters is not automated. In addition, even if a message schedule is found, it has to be manually entered into one of the configuration tools. Considering the increasing amount of signal data to be transmitted on in-vehicle networks, this manual aspect of the configuration becomes very time consuming and is very much prone to error.

In order to address the issues discussed above, this thesis contributes a tool chain that automates the most cumbersome part of the FlexRay configuration process – the schedule computation and parameter input into available configuration tools. That is, our tool performs step 1. and 2. of the network design workflow and additionally replaces the manual input of configuration parameters. Our tool builds on the open scheduling software by Klaus W. Schmidt [5], which takes the network topology and the signal set to pack signals into messages and compute the necessary network parameter. Our tool then takes the calculated values from this software to generate the other necessary network parameters and assigns the message timing. At the end of the automated process, we create a single configuration file in FIBEX (Fieldbus Exchange Format) which can be imported to any existing software tool. As a result, the need for manually entering configuration parameters is mitigated. We further verify the validity of the tool chain by testing with example laboratory setups.

The main original contributions of this thesis can be summarized as follows.

1. Improvement of the scheduling software in [5]. As a result, the software now meets all details of the FlexRay specification [3]. In addition, algorithms for the computation of implementable schedules are added to this software package
2. Automated conversion of the network configuration parameters to a standardized (FIBEX) format. As a result, the automatically computed configuration parameters can be imported into any configuration tool.
3. Development of test cases that illustrate the benefits of the developed tool chain. These correctness of the test results is verified by importing the computed configuration parameters in the configuration tool FlexConfig [13] and by conducting hardware experiments.

The remainder of this thesis is organized as follows. In CHAPTER 2 of this thesis, we give the necessary background for in vehicle networking and the FlexRay standard, with the calculation of the parameters we use and the general configuration workflow. In CHAPTER 3, we describe our workflow for the automated configuration process and the details of our developed tool chain. In CHAPTER 4 we verify correct operation of the tool chain by setting up tests with ECU networks. CHAPTER 5 gives conclusions and suggestions for future work.

CHAPTER 2

RELATED WORK

From their invention until the 1970s, automobiles have been composed of purely mechanical components. The developments in electronics, however, discouraged this trend as electrical systems are more efficient, maintainable, cheap and safer than their mechanical counterparts. Since the communications between these electrical systems need to be achieved by integrating a way of networking, several in vehicle networks have been developed. In this section, a background for these in vehicle networks and their configurations is established. Among these network standards, FlexRay, is expected to be dominant in the future. Because of this reason, a detailed description of FlexRay is given, including its protocol and scheduling parameters, its configuration workflow and the file and software standard specifics [1].

2.1 In Vehicle Networking

When electronic systems were introduced into cars, separate ECUs (Electronic Control Units) were used for each separate function of the car. The data exchange between these ECUs was maintained by point to point links. As the use of electrical systems in cars increased, so did the number of ECUs needed to manage the added functions. With huge numbers of ECUs in cars, point to point links would not continue to be a viable option due to the immense increase in properties such as cost, complexity, power consumption, space allocation and weight. This problem was then solved by using dedicated shared-medium communication networks and several standard protocols for such networks were founded [1].

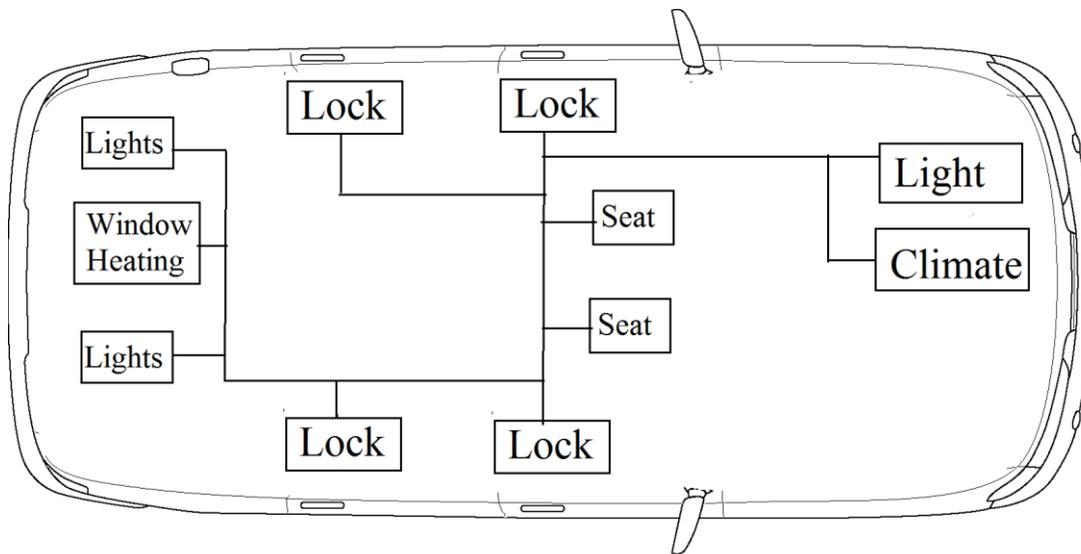


Figure 2-1: A sample ECU network [21].

The smallest data item in a network is called a signal. Signals in an in vehicle ECU network contain information such as data from sensors or actuators. These signal data are exchanged among the ECUs during the network operation. The signal data are transmitted in the form of messages, which can comprise one or more of these signals. The messages to be sent also need to be encapsulated with header and trailer segments as required by the used protocol. These protocol data units are then called frames, where the payload of a frame is the sent message.

The links between ECU nodes are established by communication channels. Channels define the medium through which the communication signals are conveyed. That is, signals generated by ECUs are packed into frames and transmitted via these channels.

It has to be noted that the performance requirements for the respective communication network depend on the particular applications running on the connected ECUs. Applications that are lenient such as mirrors, heating and other comfort functions can be handled by a cheap solution such as LIN (Local Interconnect Network) [1] [6]. Applications that need high bandwidth and security are connected by the MOST (Media Oriented System Transport) for most car brands [7]. The currently most widely standards is the CAN (Controller Area Network) bus due to its low cost, low power consumption and relatively higher speed rates. However, CAN is gradually being replaced or supplemented by the FlexRay standard due to several advantages such as higher data rate and support of hard real-time safety critical applications [1].

When choosing a bus for a specific application, the timing behaviors of the sent messages need to be considered. Regarding timings, messages can either be periodic or sporadic. Periodic messages are generated at fixed periods of time. Sporadic messages are generated as a response to certain events and do not have fixed generation times. As for the network types, there are two forms of message transmission: time or event triggered. Time triggered networks allocate time slots for each message for a predetermined time interval to avoid collisions on the shared medium. This timing scheme is called TDMA (Time Division Multiple Access) and is most suitable for

transmission of periodic messages. In a TDMA based network, when one node has a message to be sent at the allocated time slot, the other nodes wait until their allocated time slot. This causes a decrease in bandwidth utilization, but on the other hand the periodic nature of the scheduling makes properties such as delay and jitter predictable. Event triggered processes differ from the time triggered variant as when a message is sent in a reaction to an event immediately. During simultaneous events the nodes have to contend for the bus. CAN is an event triggered protocol where these contenders are assigned priorities and the ones with higher priority have precedence. Event triggered protocols are better suited for transmission of sporadic messages due to the emergency of an immediate reaction. Even though utilization is also increased by negating the needless periodic waiting time for time slots, the timing behavior is not predictable and messages may be lost in case of high collisions for a low priority message [1], [2].

FlexRay is a special type of protocol which can use both event and time triggered messaging, simultaneously if needed. In addition to this flexibility, its high data rate (up to 10Mbps compared to CAN's 1Mbps) and redundancy that employs two channels makes it a better choice for safety critical applications [1]. The advantages of FlexRay over other protocols are currently swaying most car brands in its direction. It is expected that FlexRay will be the dominant in vehicle network protocol in the future.

Embedded networks are configured ahead of time and do not change after the assembly. This reduces cost and increases reliability by removing the necessity of network discovery and configuration during runtime. In addition to this, for TDMA schemes like FlexRay to work correctly, each of the nodes of the network should be configured correctly. Apart from the protocol parameters, the time instants when a node is entitled to use the shared medium for transmitting a certain message is also preconfigured, which is called message scheduling. The schedule for each message needs to be computed beforehand to meet its transmission requirements such a periodicity or deadlines.

2.2 FlexRay Standard

The FlexRay protocol incorporates both time triggered and event triggered frames within a predetermined FlexRay cycle. All FlexRay nodes within the same network cluster are synchronized to the same cycle length and cycle count. FlexRay cycles are composed of four main parts. The static segment is where the time triggered frames are allocated within the cycle. Event triggered events are handled in the dynamic segment which can transmit messages of different length as opposed to the static segment, where all messages have the same length. The symbol window (SW) is where network management occurs within the cycle. Network idle time (NIT) is a preset idle time used to maintain synchronization between the nodes. The succession of these parts of the cycle can be observed in Figure 2-2.

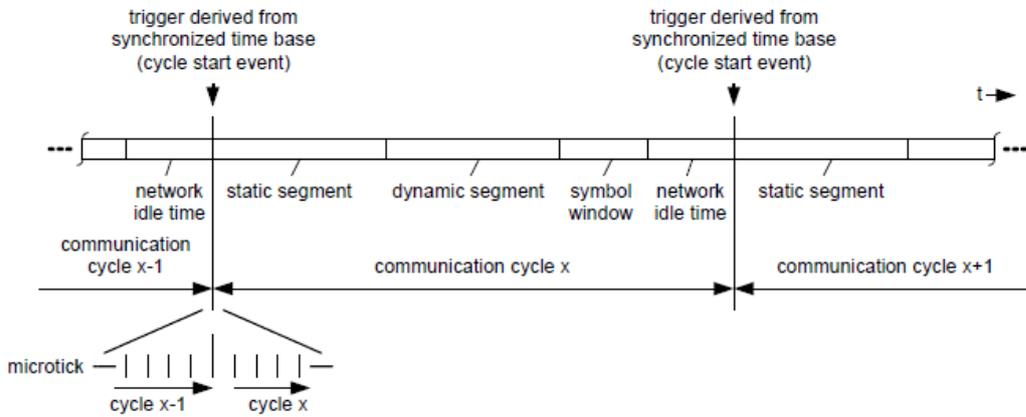


Figure 2-2: A communication cycle in FlexRay as depicted in the specifications [3].

The time representation within the FlexRay cycle is based on so-called macroticks and microticks. Each cycle is composed of an integer number of macroticks, which are composed of an integer number of microticks. Each node in the cluster is synchronized to have identical macroticks. Microticks, on the other hand, are computed from the controllers' oscillators and hence can be different among nodes depending on the controller. Thus, the number of microticks per macrotick may be different among nodes within the same cluster [3]. A visual representation of the timing hierarchy is given in Figure 2-3.

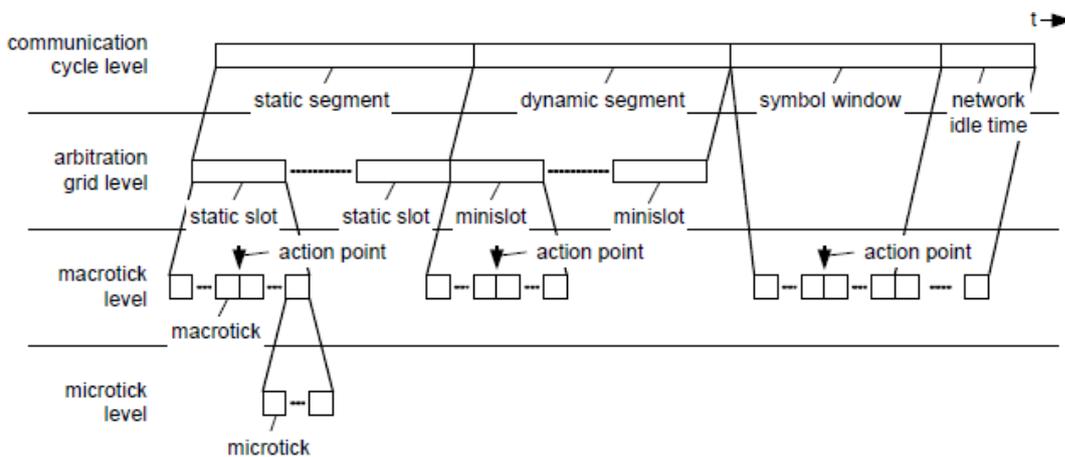


Figure 2-3: The timing hierarchy of a FlexRay cycle [3].

The static segment of the FlexRay cycle is composed of static slots that have an identical, predefined length; which is measured in macroticks. The number of static slots for every node in the same cluster is the same, which is referred as *gNumberOfStaticSlots* in the specifications. Each node has a slot counter for each channel to keep track of the occurring static slots. Only the frame that has been allocated to the slot indicated by the slot counter can be sent at that time. Even though the slot counters operate the same way in both channels, the slot allocation per channel

may be different. Slot counters are incremented after each slot time until the final static slot for that cycle is reached. FlexRay cycles always have a static segment as startup frames are sent in this segment for synchronization between nodes. A depiction of the static slot and the mentioned operations can be seen in Figure 2-4.

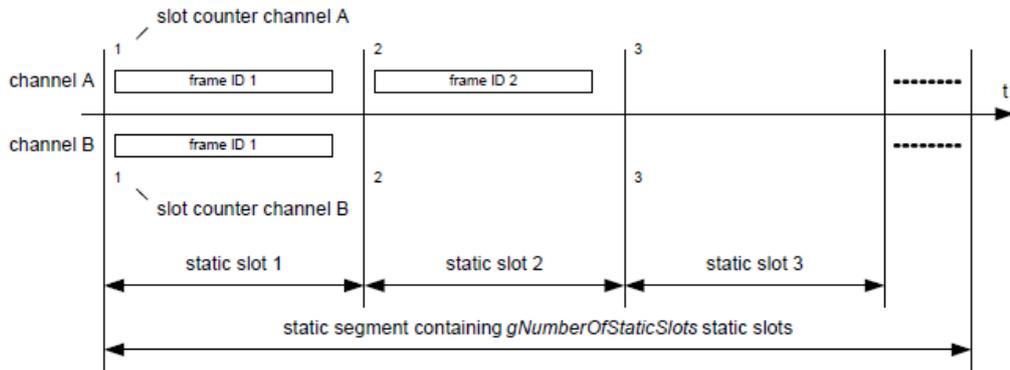


Figure 2-4: The static segment structure [3].

The timings of each frame are represented by three values: FID (Frame Identifier), repetition and offset. FID indicates which static slot in the segment the frame is allocated. A frame can only be transmitted during the slot where its FID corresponds to the slot counter. Repetition is the period of the allocation of the frame to that static slot, in terms of how many FlexRay cycles are in between each repeated frame. Offset is the base cycle where the first instance of that frame is sent. Using appropriate offsets according to each frames' repetition allows for multiplexing several frames of the same node in the same static slot. To clarify, the allocation of four periodic frames in two FIDs is depicted in Figure 2-5 over the course of 8 FlexRay cycles. The first frame depicted, Frame A, has a repetition value of 2. In our example its offset is 0 so the very first slot is allocated to this frame (slot number 0). Since its repetition is 2, every other slot after slot 0 is also allocated to this frame: Slot numbers 0, 2, 4 and 6. Then remains 4 empty slots out of the total 8 that can still be allocated to other frames. Both frames B and C have repetitions of 4 so they can be allocated 2 slots each, and can be fitted in these empty slots with the proper offsets. Here we assign frame B an offset of 1 so every fourth slot beginning from 1 is allocated. Frame C can then be assigned an offset of 3 which fills up the remaining free slots. Since there are no vacant slots in FID 1 for the final frame, Frame D, it is allocated in FID 2.

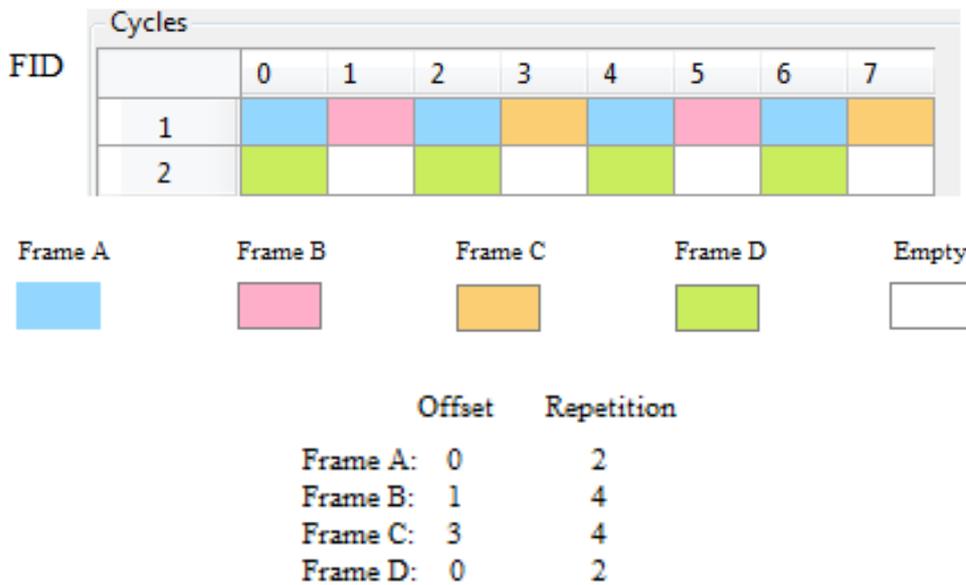


Figure 2-5: The slot allocation of four periodic frames in a static segment for 2 slots over the course of 8 cycles. The frames are color coded to represent their allocated time slots.

Unlike the static segment, a cycle may or may not have a dynamic segment. It is composed of a number of dynamic slots, which are composed of minislots that are made up of the same number of macroticks. Like the static segment, each node makes use of a slot counter to keep track of the current dynamic slot. Frames are allocated to these dynamic slots with FIDs. The number of minislots in the dynamic segment is the same for each node in the same cluster. The dynamic slots composed of minislots, however, may change depending on whether or not a frame is being sent. As seen in Figure 2-6, when a frame is being transmitted for one channel, the slot counter stops increasing until the transmission is complete. After the transmission, the counter is incremented by one. If there is no transmission, the dynamic slot only consists of one minislot and the slot counter is incremented after the minislot period.

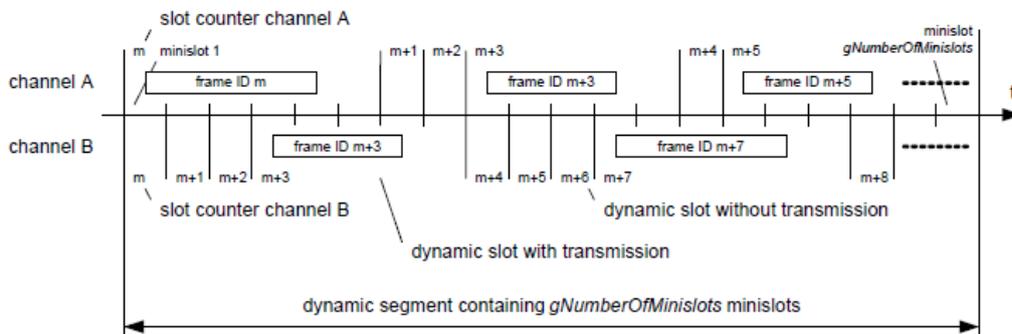


Figure 2-6: Dynamic segment structure [3].

Several parameters for each FlexRay node have to be appropriately configured according to the FlexRay specifications and the application requirements. For our purposes we separate these parameters in three categories: Protocol specific parameters, schedule specific input parameters and schedule parameters. The parameters for each category are explained in the following subsections.

2.2.1 Protocol Specific Parameters

The parameters we deem protocol specific are a collection of two types of FlexRay parameters. The first are named system parameters in the specifications and consist of constants that are related to the current FlexRay standard, such as fixed register values, bit durations of header or trailer sequences and maximum values for certain parameters allowed by FlexRay. Table 2-1 depicts the protocol constants are used in the computation of another parameter or are limiting factors for such a parameter.

Table 2-1: FlexRay Protocol constants [3].

| Name | Description | Value |
|------------------------------|--|--------------------|
| <i>cChannelIdleDelimiter</i> | Duration of the channel idle delimiter. | 11 gdBIt |
| <i>cClockDeviationMax</i> | Maximum clock frequency deviation, equivalent to 1500 ppm (1500ppm = 500/1000000 = 0.0015). | 0.0015 |
| <i>cCycleCountMax</i> | Maximum cycle counter value in any cluster. | 63 |
| <i>cPayloadLengthMax</i> | Maximum length of the payload segment of a frame. | 127 two-byte words |
| <i>cPropagationDelayMax</i> | Maximum allowable propagation delay arising from the physical layer and analog effects inherent in the FlexRay CC's involved in the transmission and reception of a communication element. | 2.5 μ s |
| <i>cSlotIDMax</i> | Highest slot ID number. | 2047 |
| <i>cStaticSlotIDMax</i> | Highest static slot ID number. | 1023 |
| <i>cdCycleMax</i> | Maximum cycle length. | 16000 μ s |
| <i>cdBSS</i> | Duration of the Byte Start Sequence. | 2 gdBIt |
| <i>cdFES</i> | Duration of the Frame End Sequence. | 2 gdBIt |
| <i>cdFSS</i> | Duration of the Frame Start Sequence. | 1 gdBIt |
| <i>cdWakeupTxActive</i> | Duration of the LOW phase of a transmitted wakeup symbol and the active | 6 μ s |
| <i>cdWakeupTxIdle</i> | Duration of the idle phase between two low phases inside a wakeup pattern. | 18 μ s |

The second collection of parameters we regard as protocol specific are non-constant configurable parameters. However, these parameters are fixed for each application. Hence, we consider them as constant as they are unrelated to the schedule computation. We set these parameters to default values within their configurable ranges. It should also be noted that configurable parameters are either global parameters that are the same for all nodes in a cluster, or node parameters that are configured separately for each node. Global parameters appear with ‘g’ or ‘gd’ prefixes and node parameters appear with ‘p’ or ‘pd’ prefixes in the FlexRay specifications [3].

Table 2-2: Configurable parameters that are unrelated to schedule computation, that are considered fixed for our study [3].

| Name | Description | Value |
|---|--|------------------------|
| <i>pdAcceptedStartup-Range</i> | Expanded range of measured clock deviation allowed for startup frames during integration. | 29 - 2743 μ T |
| <i>pdListenTimeout</i> | Value for the startup listen timeout and wakeup listen timeout. Although this is a node local parameter, the real time equivalent of this value should be the same for all nodes in the cluster. | 1926 - 2567692 μ T |
| <i>gClusterDriftDamping</i> | The cluster drift damping factor, based on the longest microtick <i>adMicrotickMax</i> used in the cluster. Used to compute the local cluster drift damping factor <i>pClusterDriftDamping</i> . | 0 - 5 μ T |
| <i>pClusterDriftDamping</i> | Local cluster drift damping factor used for rate correction. | 0 - 10 μ T |
| <i>pDecodingCorrection</i> | Value used by the receiver to calculate the difference between primary time reference point and secondary time reference point. | 12 - 136 μ T |
| <i>pDelayCompensation[A], pDelayCompensation[B]</i> | Value used to compensate for reception delays on the indicated channel. | 4 - 211 μ T |
| <i>pMacroInitialOffset[A], pMacroInitialOffset[B]</i> | Integer number of macroticks between the static slot boundary and the following macrotick boundary of the secondary time reference point based on the nominal macrotick duration. | 2 - 68 MT |

Table 2-2 (Cont'd)

| | | |
|-------------------------------|---|-------------------------|
| <i>pOffsetCorrectionStart</i> | Start of the offset correction phase within the NIT, expressed as the number of macroticks from the start of cycle. | 7 - 15999 MT |
| <i>pOffsetCorrectionOut</i> | Magnitude of the maximum permissible offset correction value. | 15 - 16082 μ T |
| <i>pRateCorrectionOut</i> | Magnitude of the maximum permissible rate correction value and the maximum drift offset between two nodes operating with non-synchronized clocks for one communication cycle. | 3 - 3846 μ T |
| <i>pSamplesPerMicrotick</i> | Number of samples per microtick. | [1, 2] |
| <i>pWakeupPattern</i> | Number of repetitions of the wakeup symbol that are combined to form a wakeup pattern when the node enters the <i>POC:wakeup send</i> state. | 0 – 63 ^a |
| <i>gColdstartAttempts</i> | Maximum number of times a node in the cluster is permitted to attempt to start the cluster by initiating schedule synchronization. | 2 - 31 |
| <i>gdBit</i> | Nominal bit time. | [0.1, 0.2, 0.4] μ s |
| <i>gdCASRxLowMax</i> | Upper limit of the CAS acceptance window. | 28 - 254 gdBit |
| <i>gdWakeupRxIdle</i> | Number of bits used by the node to test the duration of the 'idle' or HIGH phase of a received wakeup. | 8 - 59 gdBit |
| <i>gdWakeupRxLow</i> | Number of bits used by the node to test the duration of the LOW phase of a received wakeup. | 8 - 59 gdBit |
| <i>gdWakeupRxWindow</i> | The size of the window, expressed in bits, used to detect wakeups. | 76 - 485 gdBit |
| <i>gdWakeupTxActive</i> | Number of bits used by the node to transmit the LOW phase of a wakeup symbol. | 15 - 60 gdBit |
| <i>gdWakeupTxIdle</i> | Number of bits used by the node to transmit the 'idle' part of a wakeup symbol. | 45 - 180 gdBit |
| <i>gListenNoise</i> | Upper limit for the startup listen timeout and wakeup listen timeout in the presence of noise. | 2 - 16 |

Table 2-2 (Cont'd)

| | | |
|--|---|-----------------------------|
| <i>gMaxWithoutClockCorrectionFatal</i> | Threshold used for testing the <i>vClockCorrectionFailed</i> counter. | 1 - 15 even/odd cycle pairs |
| <i>gMaxWithoutClockCorrectionPassive</i> | Threshold used for testing the <i>vClockCorrectionFailed</i> counter. | 1 - 15 even/odd cycle pairs |
| <i>gNetworkManagementVectorLength</i> | Length of the Network Management vector in a cluster. | 0 - 12 bytes |
| <i>gClusterDriftDamping</i> | The cluster drift damping factor, based on the longest microtick <i>adMicrotickMax</i> used in the cluster. | 0 - 5 μ T |

2.2.2 Schedule Specific Input Parameters

These parameters are configurable over a range of values that depend on the application and are regarded as input to the tool chain. These are the parameters that define the specifics of a certain application and the schedules change when parameters are configured differently. Table 2-3 lists the configurable range of these parameters that should be defined when the network is being designed.

Table 2-3: FlexRay parameters that are dependent on the application and are configured by user input [3].

| Name | Description | Value |
|------------------------------------|--|--|
| <i>gPayloadLengthStatic</i> | Payload length of a static frame. | 0 - <i>cPayloadLengthMax</i> twobyte words |
| <i>gdDynamicSlotIdlePhase</i> | Duration of the idle phase within a dynamic slot. | 0 - 2 Minislot |
| <i>gdMinislotActionPointOffset</i> | Number of macroticks the minislot action point is offset from the beginning of a minislot. | 1 - 31 MT |
| <i>gdActionPointOffset</i> | Number of macroticks the action point is offset from the beginning of a static slot. | 1 - 63 MT |
| <i>gdCycle</i> | Length of the cycle. | 24 μ s - <i>cdCycleMax</i> |
| <i>gdSymbolWindow</i> | Duration of the symbol window. | 0 - 162 MT |
| <i>gdMacrotick</i> | Duration of the cluster wide nominal macrotick. | 1 - 6 μ s |
| <i>gdMinislot</i> | Duration of a minislot. | 2 - 63 MT |
| <i>gdTSSTransmitter</i> | Number of bits in the Transmission Start Sequence. | 1 - 15 gdBIt |
| <i>pChannels</i> | Channels to which the node | [A, B, A&B] |

Table 2-3 (Cont'd)

| | | |
|-----------------------|---|-------------------------------|
| | is connected. | |
| <i>pMicroPerCycle</i> | Nominal number of microticks in the communication cycle of the local node. If nodes have different microtick durations this number will differ from node to node. | 960 - 1280000 μ T |
| <i>pdMicrotick</i> | Duration of a microtick. | [0.0125, 0.025, 0.05] μ s |

Every static frame has a set payload length called *gPayloadLengthStatic*. Each of these frames are sent a number of macroticks after the beginning of the static slot. This amount is called *gdActionPointOffset* and configured the same for all nodes in a cluster. A depiction of these offsets can be seen in Figure 2-7.

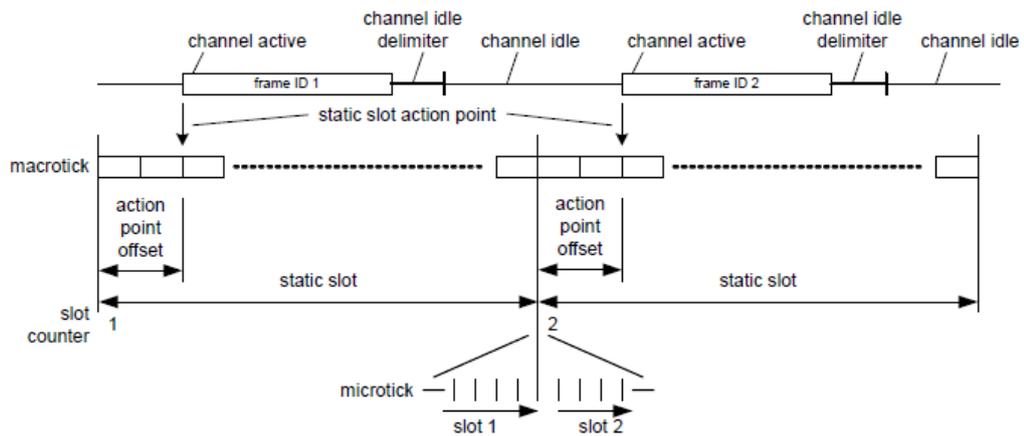


Figure 2-7: A depiction of the timings and the action point offset of static slots [3].

Similarly for the dynamic segment, every dynamic frame is offset by *gdMinislotActionPointOffset* for each minislot as seen in Figure 2-8.

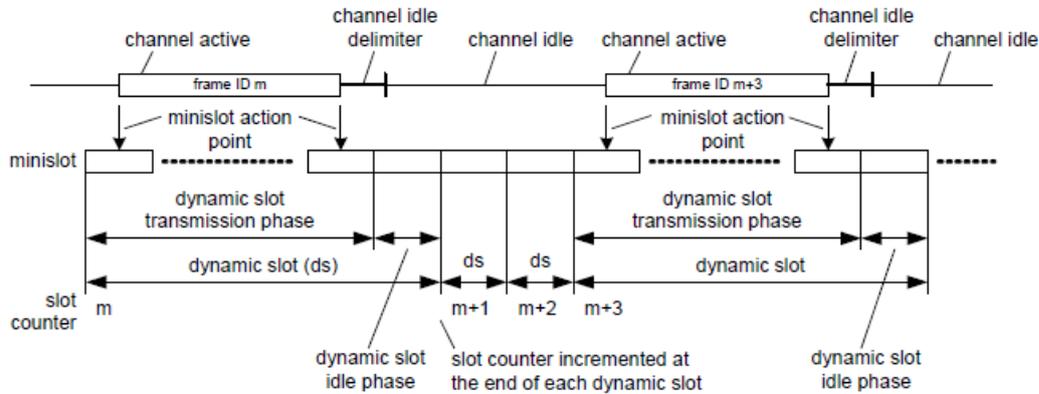


Figure 2-8: A depiction of the timings and the action point offset of minislots [3].

2.2.3 Schedule Dependent Parameters

The schedule dependent parameters are FlexRay parameters that are computed by the scheduler using the constraints present in the FlexRay specifications and the properties of the message set to be transmitted on the bus. Depending on the other configurable parameters and the message set, we compute these parameters to come up with a viable schedule before setting them. Table 2-4 lists these parameters that we compute in the scheduler automatically.

Table 2-4: Parameters that are computed by the scheduler [3].

| Name | Description | Value |
|-----------------------------|---|-----------------------------|
| <i>gMacroPerCycle</i> | Number of macroticks in a communication cycle. | 8 - 16000 MT |
| <i>gNumberOfMinislots</i> | Number of minislots in the dynamic segment. | 0 - 7988 |
| <i>gNumberOfStaticSlots</i> | Number of static slots in the static segment. | 2 - <i>cStaticSlotIDMax</i> |
| <i>gdNIT</i> | Duration of the Network Idle Time. | 2 - 15978 MT |
| <i>gdStaticSlot</i> | Duration of a static slot. | 3 - 664 MT |
| <i>pLatestTx</i> | Number of the last minislot in which a frame transmission can start in the dynamic segment. | 0 - 7988 Minislot |

gMacroPerCycle is computed by dividing *gdCycle* by *gdMacrotick*, both of which should have been previously set by the designer to define the network.

gNumberOfMinislots and *gNumberOfStaticSlots* are parameters that relate to the schedule, as in the number of minislots and static slots respectively in each cycle. Both of these parameters are

calculated by the scheduler to allow for the needed amount of static and dynamic slots. The specifics for coming up with feasible values for these parameters can be found in [16] and [17].

$gdNIT$ is the communication free period of a cycle. This period is used for various computations for the communication cycle and clock synchronization. It is computed as the remaining number of macroticks from the static segment, dynamic segment, and symbol window.

In order to compute the static slot duration, $gdStaticSlot$, first the static framelength should be computed. Each frame consists of $gdTSSTransmitter$, $cdFSS$, 80 $gdBit$ of header and trailer, the payload and $cdFES$. The static frame payload is called $gPayloadLengthStatic$ and counted in two-byte words, where each payload byte consists of 8 bits plus $cdBSS$ (2 bits). Therefore each two-byte payload is 20 bits. Then the final equation for the frameneqth, which is called $aFrameLength$ is

$$aFrameLength[gdBit] = gdTSSTransmitter[gdBit] + cdFSS[gdBit] + 80\ gdBit + aPayloadLength[two\text{-}byte\ word] * 20\ gdBit / two\text{-}byte\ word + cdFES[gdBit] \quad (1)$$

where substituting $gPayloadLengthStatic$ with $aPayloadLength$ yields the static frame length, $aFrameLengthStatic$. $gdTSSTransmitter$ is a configurable parameter where $cdFSS$ and $cdFES$ are protocol constants (1 and 2 bits respectively).

Once $aFrameLengthStatic$ is known, a minimum number for $gdStaticSlot$ can be computed using the formula given in the specifications:

$$gdStaticSlot[MT] \geq 2 * gdActionPointOffset[MT] + \text{ceil}((aFrameLengthStatic[gdBit] - 0.5 * cdFES[gdBit] + cChannelIdleDelimiter[gdBit]) * adBitMax[\mu s/gdBit] + adPropagationDelayMin[\mu s] + adPropagationDelayMax[\mu s] + adMaxIdleDetectionDelayAfterHIGH[\mu s]) / (gdMacrotick[\mu s/MT] / (1 + ClockDeviationMax))) \quad (2)$$

Another parameter computed by the scheduler is $pLatestTx$, which is the last allowed transmission slot in the dynamic segment for each node. This parameter is calculated according to the largest sporadic message sent by that node. It is configured to be a value so that there is enough of a time frame before $pLatestTx$ for any of the dynamic frames to be transmitted successfully. It is calculated after the calculation of $gNumberOfMinislots$ by subtracting from this number the amount needed to transmit the maximum payload dynamic frame as follows:

$$pLatestTx[Minislot] \leq \text{floor}(gNumberOfMinislots[Minislot] - ((aFrameLengthDynamic[gdBit] + adDTSLowMin[gdBit]) * adBitMax[\mu s/gdBit] + adMicrotickDistError[\mu s]) / ((gdMacrotick[\mu s/MT] / (1 + gClockDeviationMax)) * gdMinislot[MT/Minislot])) - gdDynamicSlotIdlePhase[Minislot] \quad (3)$$

2.3 Configuration of FlexRay

The FlexRay configuration workflow is the sequence of processes that use the signal set as input to compile the necessary ECU files to be loaded for proper operation. The signal set is the set of

signals that will be sent when the network is operational. It contains for each signal the sender and receiver nodes, signal IDs, signal types (periodic or sporadic), size, period, deadline and generation time. The goal is to configure all FlexRay nodes within the cluster using this set to meet the deadlines. The processes involved in the configuration of a FlexRay network are depicted in Figure 2-9.

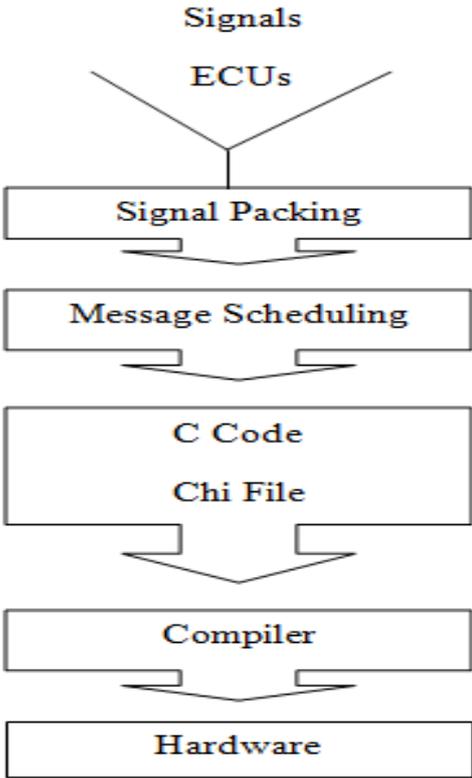


Figure 2-9: The FlexRay configuration workflow.

The configuration workflow starts with signal packaging. Signals are the main input of the whole system. They are generated when an ECU needs to communicate in the network. Packaging is necessary to form the messages that can be scheduled using the available standards. For example for the static segment of a FlexRay cycle, multiple signals can be fit into a message, which can then be fit into static slots if the sizes are adequate. This part of the workflow deals with the generation of a suitable message set from the input signal set, which includes information necessary for the scheduler such as IDs, sizes, periods and destinations.

The resulting messages then need to be scheduled with their proper timings of transmission. For message scheduling in FlexRay, periodic messages are allocated in the proper static slots and the sporadic messages are allocated in the dynamic slots. A viable schedule is acquired when all the messages in the message set are allocated in the time slots so that the defined requirements are set, such as message deadlines.

After acquiring the schedule, the configuration files of each node need to be created. This is generally done by manually entering the schedule and the necessary parameters into a configuration tool. There are several commercially available such tools, some of which will be explained in Section 2.5. Using these tools, the configuration files for each ECU can be generated. The generated files from these tools are called controller host interface (CHI) files. The controller host interface manages the data exchange between the FlexRay protocol engine and the host processor [3]. The chi file containing this information can then be used to compile the actual binary data that needs to be uploaded per ECU. Once the ECUs are loaded and the network is booted up, no further configuration takes place.

2.4 File and Software Standards

The different processes forming the configuration workflow need to be handled by several tools in sequence, forming a tool chain. The data flow between these tools needs some sort of compatibility in order to establish data exchange. The ASAM organization (Association for Standardisation of Automation and Measuring Systems) was founded to develop and maintain interface standards for vehicle electronics. For our approach in configuration, to forward the data from the Message Scheduler to any standardised configuration tool, we use the ASAM developed FIBEX (Fieldbus Exchange Format) standard [15]. FIBEX is an XML based data exchange format. It is a standard that is used to exchange data with multiple protocols in message oriented networks. Since it was developed by ASAM, the FIBEX format is compatible for conversion between most standard tools. It supports several in-vehicle transmission protocols and is the established exchange format for FlexRay. FIBEX shares many principles with AUTOSAR (Automotive Open System Architecture), another standard for automotive software. This fact indicates that FIBEX tools are appropriate for many applications since many automotive and electronics are affiliated with AUTOSAR [18], [19].

A FIBEX file consists of several objects that define the aspects of the network to be configured. Network topology is defined in cluster objects, where the ECUs and the channels connecting them are configured. The timing parameters of communication objects are configured in the channels. The communication objects that can be configured this way are signals, frames and PDUs (Protocol Data Units). Signals are grouped into PDUs, which belong to the Interaction Layer. PDUs are then grouped into frames which belong to the Data Link layer (FIBEX versions below 3.0 do not support PDUs; signals are directly grouped into frames). The physical aspects of the signals are stored as 'codings' which are separate data objects. Codings include information such as bit length and computation method. FIBEX also allows the configuration of Gateway objects as an interface to other bus systems [15].

The interfacing of the protocol configuration of FlexRay and the message schedule to the hardware are handled by .chi (Controller Host Interface) files for each node in the network. These files can be generated by the use of commercially available FlexRay configuration tools. Such configuration tools allow the user to define a network, set the protocol parameters and enter a schedule to create the .chi files for the defined nodes. Once the necessary .chi files are created, these need to be converted into a hex file format that can be loaded to the hardware using software specific to the hardware vendors. This final hex file to be loaded can be of different formats depending on the vendor. The software to be used to generate these files from the .chi files also are specific to the model of the boards used.

2.5 Current FlexRay Configuration Tools

There are several configuration tools that can be used for FlexRay Networks. One is EB Tresos Designer by the company Elektrobit. This tool can be used to plan the signals and the network schedule for FlexRay, as well as CAN and LIN. The parameters are checked for interdependencies during the design. This software also allows user created AUTOSAR compliant software components to be linked with Tresos projects [10].

FlexRay Configuration Package by dSpace provides a configuration tool for simulating FlexRay nodes on dSpace hardware. This software also supports Protocol Data Units (PDU) as described by AUTOSAR. By entering the signal set and the schedule, the communication code and the node configurations are generated. This data can then be used to create MATLAB[®]/Simulink[®] models using the RTI FlexRay Configuration Blockset, which can then be used in the applications [11].

The Network Designer FlexRay by Vector is a tool where schedules can be generated and both cluster and node parameters can be configured. Vector exclusively uses the FIBEX format to exchange information with other tools such as Network Designer CAN. The configuration files generated by the design tools (and possible other Vector tools) are then as a whole diagnosed by the next software Vector's design tool chain, CANoe, for ECU operation. The final ECU files can then be loaded by using Vector Bus Interfaces for specific nodes [12].

TTXPlan by TTTech is a FlexRay configuration tool with the added application of automatically mapping signals to frames and generating a schedule for time-triggered networks. The empty slots in the schedule can then be modified by the user without disrupting the frames allocated by the previous schedule [14].

FlexConfig by Eberspächer is the FlexRay configuration tool we use in this study. It allows the configuration of FlexRay and CAN nodes along with gateways between different nodes. FlexConfig also simulates wakeup and startup signals, CRC algorithms and network management. The controller register values that appear in the .chi files can also directly be changed, if the user is able [13].

All of these tools are AUTOSAR compliant and projects can be exported to other software as long as they use the same exchange standards. Since all these tools support the FIBEX exchange format, FIBEX files can be imported or exported with any of these software tools.

2.6 Shortcomings of the Current Design Workflow

Even though the current workflow is adequate for configuring FlexRay nodes, some shortcomings still need to be pointed out. One such liability is that configuration tools do not support any scheduling. The tools only take whatever schedule is entered to create the configuration files, possibly giving out errors and warnings in the case of a schedule that does not meet the specifications of the used standard. The schedule itself, however, needs to be computed by the engineer to be entered in to the configuration tool. This brings us to a second problem, the schedule has to be manually entered in to the tool, as in defining the all the frames and their timings by hand. The scheduling parameters also need to be entered manually by looking at the computed schedule. Manual entry by an engineer takes effort and time, in addition to being prone to human errors.

For the message packing and scheduling parts of the workflow, there are no commercial tools available. There is an open-source scheduling tool by Klaus W. Schmidt [5], which analyses a given signal set and comes up with a viable schedule, but the resulting schedule still needs to be entered to a configuration tool manually as the tool does not provide a compatible output for commercial configuration tools.

In this thesis, we try to automate as many of the processes of the current design workflow as possible to deal with these shortcomings. Here we design a tool to create a file that includes both the schedule and all the configuration parameters that can be imported to all commercial tools following the same standard (FIBEX xml format). We also make alterations to the work previously done in [5] so that the computed schedule can be automatically integrated into the tool chain. Thus, once the signal set and the ECUs are introduced to the tool chain, the signal packing, scheduling and configuration processes are sequentially done by the tool chain automatically. The resulting file can then be imported to any standard commercial tool to generate the chi files.

CHAPTER 3

AUTOMATED CONFIGURATION TOOL CHAIN DEVELOPMENT

We describe our tool chain for the automated FlexRay configuration process from a given network topology and signal set, to the generation of the .chi files in this chapter. We begin by describing our workflow and the tasks performed by each working block. Then we describe the first block, the scheduler, which is mostly previous work and explain the additions to the previous work and why they are added. After that, we go into the details of the writing of the FIBEX xml file from the output of the scheduler. This part contains the generation of the network topology in the file, inserting the configuration parameters and the message timings, in addition to some specifics related to the FIBEX file format.

3.1 The Tool-Chain Workflow

The overall configuration process we employ can be summarized in three main blocks. The first block is the scheduler, where the signal set is packed into messages and scheduled using a scheduling algorithm. This part is handled by an API developed by Klaus W. Schmidt [5]. We make some improvements and adjustments to the previous API to make it usable by the next set of tools. The second block is the FIBEX file scheme we have developed. This part uses the schedule and the parameters obtained from the scheduler to create the single FIBEX file (xml extension) which represents the whole FlexRay network. The conversion from the computed schedule to the configured FIBEX file is automatic, as opposed to current configuration schemes where the schedule and the configuration has to be manually entered to an available tool. In our method, the universal FIBEX file is created automatically which can be exported to all the professional tools that use this standard. The final block of the whole setup makes use of FlexConfig as such commercially available tool. This part deals with the generation of the loadable hex files for the hardware from the configuration file. An overview of the tool chain is given in figure 3-1.

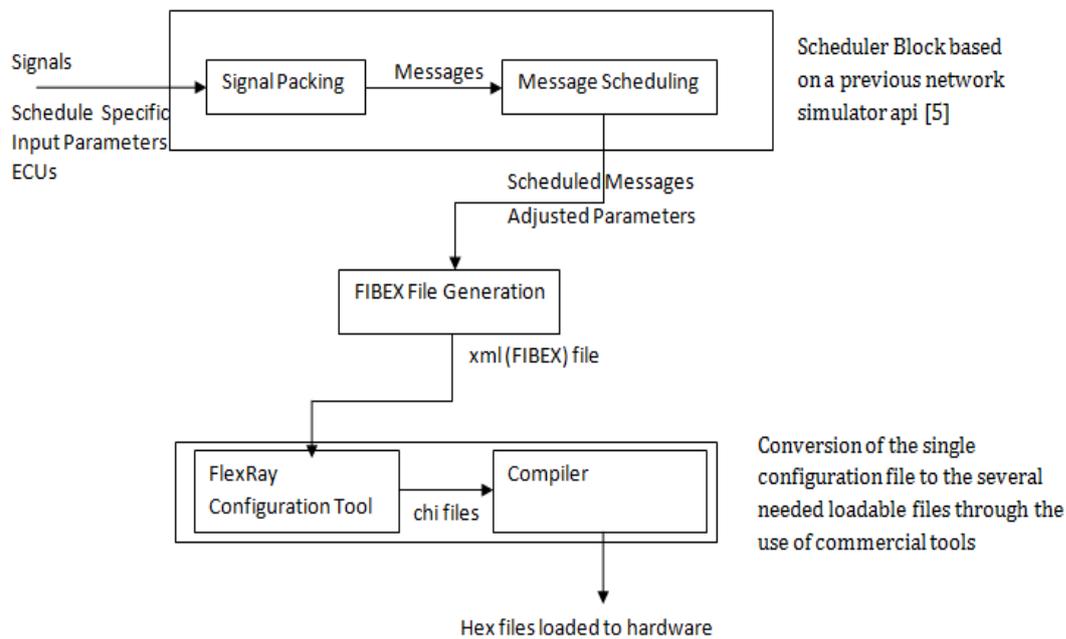


Figure 3-1: An overview of our workflow.

The input to the system is the signal set, along with the schedule specific input parameters defined in Section 2.2.2 and the ECUs. The signal set is the set of signals that will be sent throughout the network during operation. The signal set includes information such as signal type (periodic or sporadic), size, generation time, deadline, sending and receiving nodes per signal. The parameters input to the system are the schedule specific parameters that were mentioned previously in chapter 2.2.2. We introduce the ECUs into the system as separate objects called node items, where the configuration parameters and schedules for each ECU are kept.

The collection of inputs is first subject to the signal packing process of the scheduler API. Signals that are viable to be sent simultaneously are grouped into a single message. Each sporadic signal constitutes a single message. As for periodic signals, signals being sent from the same ECU can be packed into a message, according to their receivers and deadlines. The API groups the periodic signals and sets the static slot duration using an algorithm to minimize the bandwidth utilization [8].

The outcome of the signal packaging block is the message set. The message set is the set of messages and their relevant parameters that will be scheduled in the network. The message set is then run through the scheduler where we compute the schedule. The schedule for each message in the message set is represented in terms of their FIDs, offset, and repetitions. This representation is the same for both periodic and sporadic messages; where we set the offset and repetition of all messages scheduled in the dynamic segment as 0 and 1 respectively. Several network parameters are also adjusted for schedulability during the computation. The final output of the scheduler is then, these parameters and the overall FlexRay schedule.

Conventionally, this generated schedule and the necessary parameters are then manually entered to an existing network design tool manually so that the configuration files can be generated. The

FIBEX generator project we developed, on the other hand, generates a universal xml file that can be imported to such design tools. Therefore, manually entering the parameters and the schedule into the tools is now not needed. The automatically generated xml file can be imported to any network design tool that uses the FIBEX standard (which most do), from where the chi (Controller Host Interface) files for each ECU can be generated.

For our case, we use FlexConfig™ Developer for the generation of chi files. Once we import the FIBEX file we generate into FlexConfig, we can observe the parameters and the schedule in the FlexConfig interface as if we had manually registered all the necessary values. From here, we use the export functionality to generate the .chi file for each node. The final step is to convert these .chi files into machine code (hex files) and load them to the network nodes. It should be noted that we do not fully automate the processes of importing/exporting with FlexConfig and conversion of the exported .chi files to machine code. Even though automating FIBEX import and .chi export processes of the FlexConfig could provide some convenience, the time and effort used is insignificant compared to the previous processes. Furthermore, automating this process as is would limit our tool chain with specifically integrating FlexConfig, whereas now any configuration software employing the same standard can be substituted with ease. The same reasoning applies to the conversion of files to the machine codes as well since automation of this process too would restrict us to using the same file formats and hardware for all applications.

3.2 The Scheduler

3.2.1 The Existing Infrastructure

The scheduler code is where we define the nodes and insert the signals. These signals are then packed into messages and the schedule is obtained from these messages using a scheduling algorithm. For scheduling and message packing purposes, we use tools readily available in [5]. In this network scheduling API, ECUs are represented by objects called node items (`FXNodeItem`). The information stored in these objects is used both in the message packing and scheduling processes. Signals that are generated by the ECUs are represented by `Signal` objects and are stored in the `mSignalVector` vector of their respective `FXNodeItem` objects. The `Signal` objects contain the parameters needed to define a signal: type (periodic or sporadic), period, deadline, size, generation time etc. The `Message` objects that represent the frames that these signals will be grouped into also have the same attributes. For each specific network, a scheduler object of the class `FXScheduler` is created, which is where the methods that are used to pack frames and compute the schedule are held. This scheduler operates on the set of defined node items. The node items to be used need to be inserted into the scheduler objects node item vector (`FXNodeItemVector`) to be included in the network.

After defining the network and the signal set, the schedule is computed using the computation functions. The packed messages for each scheduled node are stored in the `mOptimalMessages` vector of each `FXNodeItem`. The FID numbers and the maximum repetition allowed for each node is stored in the map `mFIDAssignmentMap`. For the dynamic segment the number of minislots needed are computed and the scheduled dynamic messages are displayed in the output with their message IDs and response times.

3.2.2 Improvements and Additions

For incorporating this tool into our automated configuration tool chain, we make several improvements due to several missing features of [5] that are needed for our case. One such improvement is that the previous version did not take into account the receivers of each inserted

signal, since only the senders were needed to come up with a viable schedule. When node configuration comes into play however, each node has to be configured to know when they are in the receiving end. Therefore, we change the signal objects to include the receiving node in terms of an integer node ID (we also do the same for the message objects). When we then pack these signals into messages, we also include the destinations for each signal packed within the message. The destination for each message is then the union of the destinations of all the packed signals. We also updated the correct formulas for the computation of frame lengths according to the equation (1) given in 2.2.3.

A bulk of the changes added are the computation of the necessary information that were not present before and interface all the output to the next block. We accomplish these functionalities using the `ComputeScheduleForFibex` method for each node. In this method we first extract the contents of the `mFIDAssignmentMap`, which holds the FIDs and their maximum available repetitions. We also extract the `mOptimalMessages` vector of messages, which will be placed in these FIDs according to the schedule. The FIDs and repetitions were already assigned in the previous scheduler, from where we added the computation of the offsets for the extraction.

A pseudocode for the general flow of this method is given below:

```

FIDRep = Map of FIDs to maximum repetitions
While FIDRep still has an entry
    Get one FID and maximum repetition from FIDRep
    Loop over all messages
        Get repetition of current message
        Set offset = 0
        Set FreeSlotNotFoundFlag = true
    While offset < message repetition and FreeSlotNotFoundFlag = true and message
    repetition <= maximum repetition
        Check if the current message is viable for assignment to this FID with its repetition and current
        offset
        If viable do the assignment and remove the message
        Else increase offset by 1
    Remove the current FIDRep entry

```

What we do here is taking all available FIDs according to their maximum repetitions and filling available slots with the sent messages. For each FID, we first take one of the messages and see if it can be allocated to this FID. We consider all available offsets while checking, which is an integer between 0 and the message repetition. If the message is available with one of the counted offset values, the allocation takes place and we mark the corresponding slots as occupied. We also remove the message from this process by setting its repetition as 0, so that it does not go into the while loop for the subsequent FIDs. Once we check all of the messages for one FID, all of the possible messages would have been allocated so we discard this FID from this process (by deleting the `FIDRep` map entry) and move on to the next FID. If all the message repetitions and FIDs that are computed by the scheduler are indeed correct, no message is left out at this block.

An example scenario for one node is given in Figure 3-2 for clarification. This part of the output from the scheduler says that there are four FIDs for messages sent from that particular node and seven different messages. The first rows of the output are the content of `FIDRep` matching FIDs to their maximum repetitions. The rows below are for each message entry of the `mOptimalMessages` vector and their repetitions. From this content, our algorithm first takes FID 1 and loops through the seven messages to check if one of them fits with 0 offset. Incidentally, the maximum repetition allowed in FID 1 is a repetition of 1, which is the repetition

of the first message so the allocation is done for this message and the message entry is removed from the process. Since no other message can be allocated to FID 1, its entry is also removed from `FIDRep` and the process continues for FID 2 with maximum repetition 2. From the remaining six messages, three of them have repetitions of 2 so one of them is allocated to FID 2 with offset 0 and its vector entry is removed. Then the algorithm checks the remaining 5 messages to find a message that can be allocated to the remaining slots with offset 1. At this run one of the messages with repetition 2 is allocated to the remaining slots with offset 1 and repetition 2. Afterwards since further messages cannot be allocated to FID 2, this entry is removed from `FIDRep` and the process continues for the remaining two FIDs and four messages.

```

****
NodeItem Number: 2
FID: 1   Max Repetition: 1
FID: 2   Max Repetition: 2
FID: 3   Max Repetition: 4
FID: 4   Max Repetition: 16

FID: 1   with maxrep: 1
Message with rep: 1
Message with rep: 2
Message with rep: 2
Message with rep: 2
Message with rep: 4
Message with rep: 4
Message with rep: 16

```

Figure 3-2: A sample screenshot from the scheduler output. The FID to Max Repetition pairings are the content of the `FIDRep` map structure. The following message repetitions are the entries of the `mOptimalMessages` vector, duplicate lines indicate different messages with the same repetition.

During the inspection of each message for this segment, we represent the static slots of the current FID with an array called `slotmap` which keeps track of the availability of the slots. The size of the array is the maximum repetition value as there are no slots beyond that. Whenever a message is checked for viability, we compute the slots that are occupied by the message. If all the necessary slots are available, then the message is assigned and all these slots are marked as unavailable. The code segments that handle the checking and allocation parts are given below.

```

/* Checking for availability */
for(availableflag=true, j=offset; j<maxrep && availableflag;j=j+currentrep){
    if (slotmap[j] == false) availableflag= false;}

/* Marking the used up slots (if assignment takes place) */
for(j=offset;j<maxrep;j=j+currentrep){
    slotmap[j]=false; }

```

Figure 3-3 displays the output for a node with a single FID with maximum repetition 32 and 4 messages to be allocated. Since the maximum repetition is 32, the `slotmap` array also has 32 elements, which are all set to `true` initially.

```

FID: 1      with maxrep: 32
Message with rep: 4
Message with rep: 16
Message with rep: 32
Message with rep: 16

```

Figure 3-3: A sample screenshot from the scheduler during message assignment to FID 3 when there are four messages currently unassigned.

The algorithm again sets `offset` as 0 and begins checking the messages. The first message with repetition 4 is suitable for allocation so the assignment is made, the message entry is removed from the message vector and the `slotmap` elements representing the occupied slots are marked as false. A visual representation of the marked array elements is given in Figure 3-4.

| | | | | | | | | | |
|---|----|----|----|----|---|----|----|----|----|
| X | 0 | 1 | 2 | 3 | X | 4 | 5 | 6 | 7 |
| X | 8 | 9 | 10 | 11 | X | 12 | 13 | 14 | 15 |
| X | 16 | 17 | 18 | 19 | X | 20 | 21 | 22 | 23 |
| X | 24 | 25 | 26 | 27 | X | 28 | 29 | 30 | 31 |

Figure 3-4: The `slotmap` elements marked as false for a message with offset 0 and repetition 4.

Next, the remaining three messages are tried with offset 1. Since the messages are checked in the order they are present in the `mOptimalMessages` vector, the first available message is the one with repetition 16. After this message is assigned and the `slotmap` entries are marked, the remaining messages are again tried for the array elements that are not yet marked as false. The visual progression of the `slotmap` array elements for the remaining messages is given in Figure 3-5.

| | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|
| | 0 | X | 1 | 2 | 3 | | 4 | 5 | 6 | 7 |
| | 8 | | 9 | 10 | 11 | | 12 | 13 | 14 | 15 |
| | 16 | X | 17 | 18 | 19 | | 20 | 21 | 22 | 23 |
| | 24 | | 25 | 26 | 27 | | 28 | 29 | 30 | 31 |
| .. | .. | .. | .. | .. | .. | .. | .. | .. | .. | .. |

| | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|----|
| | 0 | | 1 | X | 2 | 3 | | 4 | 5 | 6 | 7 |
| | 8 | | 9 | | 10 | 11 | | 12 | 13 | 14 | 15 |
| | 16 | | 17 | | 18 | 19 | | 20 | 21 | 22 | 23 |
| | 24 | | 25 | | 26 | 27 | | 28 | 29 | 30 | 31 |
| .. | .. | .. | .. | .. | .. | .. | .. | .. | .. | .. | .. |

| | | | | | | | | |
|----|----|----|---|----|----|----|----|----|
| 0 | 1 | 2 | X | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | X | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | | 27 | 28 | 29 | 30 | 31 |

Figure 3-5: The slotmap array visual representation for each run of the FID assignment process. Darkened blocks depict elements that have been marked previously by preceding messages. Blocks marked with X represent the slots taken by the current message at hand. From top to bottom the three pictures represent the allocation of three messages in sequence: Offset 1 repetition 16, offset 2 repetition 32, offset 3 repetition 16.

After computing the static segment frames, we extend both the static segment and the dynamic segment to fill the entire cycle time. First we add two static slots to the current schedule as a reservation for the two necessary synchronization frames. Then more static slots and minislots are added. We add 100 minislots for each added static slot, mainly because minislots are generally much smaller than static slots. We stop adding when the remaining time in the cycle is less than the duration of 1 static slot and 100 minislots. At this point, the remaining time is added to the NIT. After updating the slots, the dynamic frames in the `mScheduledMessagesDynamic` vector are updated with their new FIDs starting with the new starting FID of the dynamic segment.

When we compute the schedule, we also store the information for each message for the uses in the upcoming blocks. The information needed per message are FID, offset, repetition and the destination nodes. These are stored in the `ScheduledMessages` vector. Each entry to this vector represents each message. We use this vector as an interface of the schedule for the static segment and the FIBEX block.

We pass on the dynamic schedule to the FIBEX block in a similar manner. The difference is that there is a single `mScheduledMessagesDynamic` in the scheduler object as opposed to separate `ScheduledMessages` vectors for each node item. As per the FIBEX standard, the message parameters are namely the same. The FIDs are assigned to the dynamic messages in order of precedence, where offsets are set to 0 and repetitions are set to 1. The needed minimum duration for the dynamic segment is also calculated here, but not the needed amount of minislots for each sporadic message.

These two message vectors, `ScheduledMessages` (for static messages) and `mScheduledMessagesDynamic`, are the main input of the next block. These schedules along with the necessary protocol and schedule parameters need to be entered to a network tool, so the hardware compatible files can be produced. We make this process automatic in the next step by generating a standardized FIBEX file that imports all necessary parameters to the tool for us.

3.3 The FIBEX Project

The FIBEX block in Figure 3-1 is where we create the xml file using the FIBEX format. XML (Extensive Markup Language) is a format which is formed around data objects called elements and their attributes. The content of these data objects are encapsulated within their start and end tags in the file. Each file has to contain one root element, which is the element that does not occur as any other elements' content through the whole document. The elements can also have attributes which are defined in the start tags. The start and end tags are encapsulated within `<>` characters and the value stored (called declarations in xml) in that element is encapsulated by these tags. The FIBEX

standard also employs this standard and a valid file has to follow the XML definitions [9] [15] [20]. A blank FIBEX file is given in Figure 3-6 to clarify the encasings of the elements and document form of xml.

```
<?xml version="1.0" encoding="utf-8"?>
<fx:FIBEX xmlns:asr30nm="http://www.asam.net/xml/fbx/nm/asr30">
  <fx:PROJECT ID="ProjectType_06cd2a8294164474bfcda54927ff139e">
    <ho:SHORT-NAME>ShortName</ho:SHORT-NAME>
  </fx:PROJECT>
  <fx:ELEMENTS />
</fx:FIBEX>
```

Figure 3-6: A blank FIBEX file as a sample xml file. Element tags are marked in blue, attribute names in red, attribute values in purple and stored values in black.

The structure of the FIBEX block of our code is similar to the xml node hierarchy of the standard FIBEX file. This hierarchy is given in Figure 3-7. Each block in the figure represents the entity with the respective name and appears as xml node (tags) in the written file. How these tags are written into the xml-file is elaborated on in the sequel.

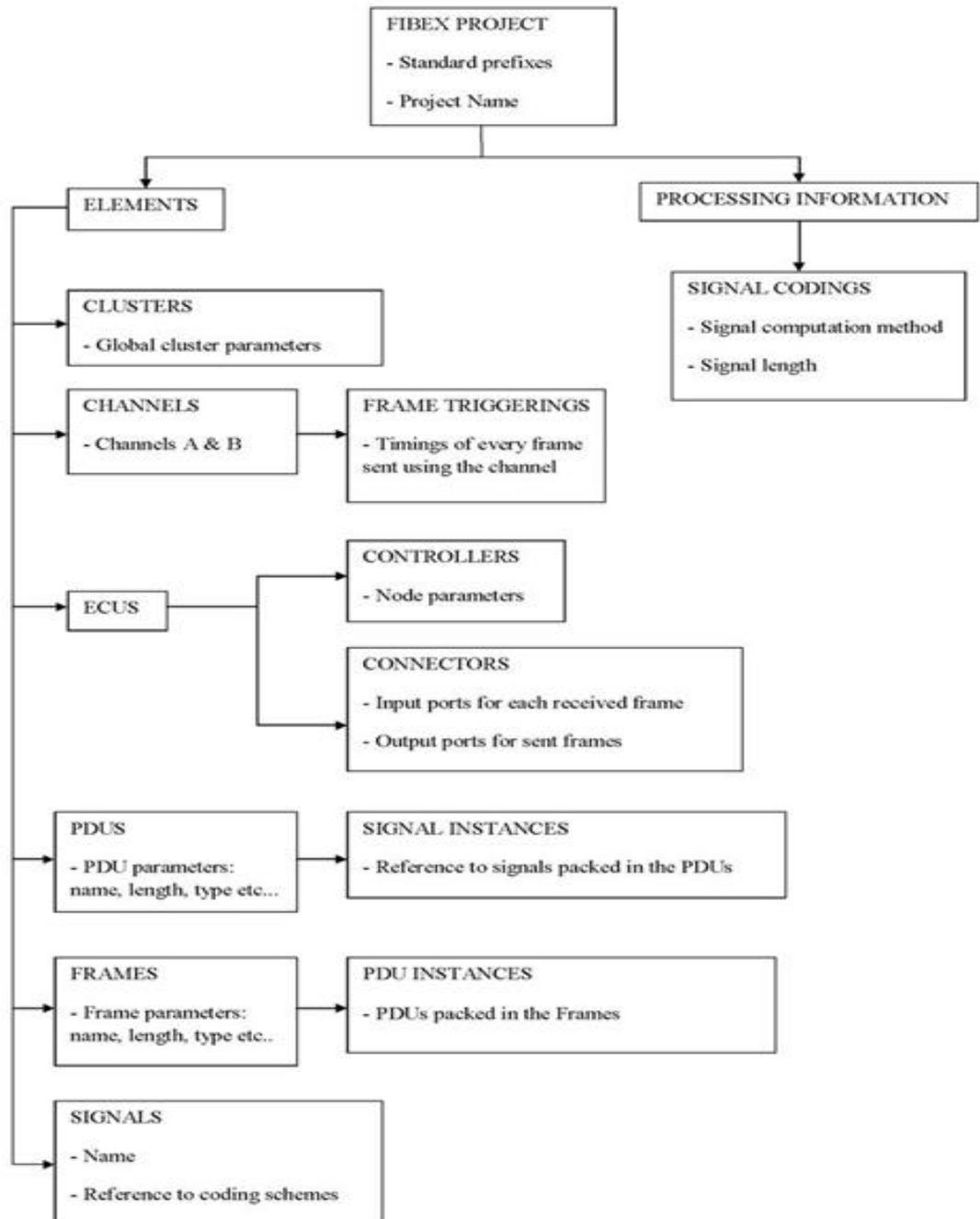


Figure 3-7: The FIBEX standard structure [15].

We use the GNU library libxml2 to write an xml file. This library defines a specific type of pointer called xml node pointer (`xmlNodePtr`) for the writing process. Each node pointer represents an xml node (or tag) that should appear in the file. The created node pointers are related to each other with a tree structure. As in the nature of xml files, each file has a root xml node which has one or more children xml nodes which in return may or may not have children nodes and so forth. The `Write()` function of the library then starts writing from the specified root node and goes down the tree recursively, writing the proper tags and attributes for each node in the file [9].

As for creating the FIBEX files using this library, it is required to define all xml nodes. The standard requires one main node with the `fx:FIBEX` tag with some attributes regarding the version number as is shown in Figure 3-2. Then we create the nodes for each block appearing in Figure 3-1. The children of these nodes then represent whatever entity they represent in the FIBEX file; for example xml nodes that hold the attributes of signals are children of the `fx:Signals` node.

For each main xml node that should appear in the file, we define a C++-class that points to the respective node by an xml node pointer. Each object that is instantiated from such class contains the configurable parameters. We name the main class `FXFIBEX`, which contains the xml node pointer called `fibex_node`. Each FIBEX project has to have one `FXFIBEX` object as it contains the root node of the xml file, which is `fibex_node`. The ‘elements’ and ‘processing information’ nodes are children of this `fibex_node` as seen in Figure 3-1. The children of the ‘elements’ node are the network elements: ‘Clusters’, ‘ECUs’, ‘channels’, ‘frames’, ‘PDUs’ and ‘signals’. For each element in the network, we create the appropriate objects and then connect them to the main `FXFIBEX` object. The connection is made by creating the xml node tree, where each xml node pointer for each object is the child of the appropriate type. For example, for a network with 3 ECUs, 3 ECU objects are created, whose xml nodes are added as children of the `ecus_node` in the `FXFIBEX` class so all these objects are written under the `fx:ECUS` tab. The `ecus_node` is the child of the `elements_node`, so it appears under the `fx:ELEMENTS` tab, which is under the main FIBEX tab.

We use two parameters for the constructor of the `FXFIBEX` class. The first is the version number as our code supports both version 2.1 and 3.1. The other parameter is the scheduler object that we used previously for computing the schedule. The `FXFIBEX` object contains the xml nodes where the other objects are pointed so that they appear in their respective tabs. We also set the prefixes that need to occur as xml properties in the `fx:FIBEX` tab according to the version number. Once we create this object, the relations of the necessary xml nodes are automatically set. Then for every other object created that needs to appear in this file, we add them using their respective methods.

The method to write the xml file is also included in the `FXFIBEX` class. What this method does is to actually add the xml nodes contained in each object to their respective parents. Then the information contained in these objects is written as the children xml nodes using the syntax that should be used in FIBEX files.

3.3.1 GUID References

Most tags that occur in the FIBEX file format need to have unique identifiers which are stored as attributes in the start tag. To keep track of these identifiers, we use a GUID (Global unique identifier) set that is specific to the current `FXFIBEX` object. We create these unique identifiers by appending a 32 byte random hex number to the identifier type. We create these unique random hex numbers by adding four 8-digit hex numbers back to back. We check for uniqueness by comparing

the newly generated random ID with all the other IDs in the GUID set. The code segments handling GUID operations are given below.

```
std::string GenerateGUID();
std::string GenerateGUID(){

int flag = 1;
int idno1, idno2, idno3, idno4;
stringstream ss;

while (flag){
    ss.str( std::string() );
    ss.clear();
    idno1 = rand(), idno2 = rand(), idno3 = rand(), idno4 =
rand();
    ss << hex << idno1 << hex << idno2 << hex << idno3 << hex
<< idno4;
    if (FXFIBEX::GUIDset.count(ss.str()) == 0) flag = 0;
}

FXFIBEX::GUIDset.insert(ss.str());
return ss.str();
}
```

These identifiers are used to identify each unique entity in the FIBEX file. When certain connections need to be established between these entities, they are referred by using these unique identifiers.

3.3.2 Creation of Network Elements

We represent each cluster by a `FibexCluster` object. For every cluster in the network, we create a `FibexCluster` object. We add these cluster objects into the `FXFIBEX` object by inserting the `FibexCluster` object into the `Clusters` vector in the `FXFIBEX` object. When the `Write()` method is operated in the `FXFIBEX` object, it goes through this vector to write all the present clusters one by one.

The `FibexCluster` object contains the global parameters necessary for the configuration of the cluster. We migrate the schedule specific parameters that are present in the scheduler to created `FibexCluster` objects by using the `ConfigureFibexCluster` method of `FXFIBEX`. The input of this method is the `FibexCluster` object that needs to inherit the parameters. These parameters are either variables in the created `FXScheduler` object, or static variables of the `FXNodeItem` class. The following is an example for both of these kinds of variables configured in this method.

```
Void FXFIBEX :: ConfigureFibexCluster ( FibexCluster & Cluster){
    /** Variable from FXScheduler */
Cluster.SetCStaticSlot(FibexScheduler.GetmSSlotInMT() );
...
    /** Static variable of FXNodeItem */
Cluster.SetCSWindow(FXNodeItem::GetSWDuration());
```

We also refer up to two channels that are connected to the cluster. `ChannelRefVector` in the `FibexCluster` object is a vector of strings where we insert the channel GUIDs. When we write the file, the tool goes through this vector to write the IDs of the channels under the `fx:CHANNEL-REFS` tab.

Figure 3-8 depicts the `fx:CLUSTERS` segment written when we write the configured `FibexCluster` object into the FIBEX file.

```
<fx:ELEMENTS>
  <fx:CLUSTERS>
    <fx:CLUSTER xsi:type="flexray:CLUSTER-TYPE" ID="FlexRayCluster_599e7abb60754e4917f1cb1577dbd1e">
      <ho:SHORT-NAME>Cluster1</ho:SHORT-NAME>
      <fx:SPEED>10000000</fx:SPEED>
      <fx:IS-HIGH-LOW-BIT-ORDER>true</fx:IS-HIGH-LOW-BIT-ORDER>
      <fx:BIT-COUNTING-POLICY>SAWTOOTH</fx:BIT-COUNTING-POLICY>
      <fx:PROTOCOL xsi:type="flexray:PROTOCOL-TYPE">FlexRay</fx:PROTOCOL>
      <fx:CHANNEL-REFS>
        <fx:CHANNEL-REF ID-REF="FlexRayChannel_4783c28f38134be5bb7ea9357afb94de" />
        <fx:CHANNEL-REF ID-REF="FlexRayChannel_6147508f7fdd41b9bf5b2f66bf838fc0" />
      </fx:CHANNEL-REFS>
      <fx:MAX-FRAME-LENGTH>8</fx:MAX-FRAME-LENGTH>
      <flexray:COLD-START-ATTEMPTS>31</flexray:COLD-START-ATTEMPTS>
      <flexray:ACTION-POINT-OFFSET>16</flexray:ACTION-POINT-OFFSET>
      <flexray:DYNAMIC-SLOT-IDLE-PHASE>1</flexray:DYNAMIC-SLOT-IDLE-PHASE>
      <flexray:MINISLOT>11</flexray:MINISLOT>
      <flexray:MINISLOT-ACTION-POINT-OFFSET>5</flexray:MINISLOT-ACTION-POINT-OFFSET>
      <flexray:N-I-T>12</flexray:N-I-T>
      <flexray:STATIC-SLOT>53</flexray:STATIC-SLOT>
      <flexray:SYMBOL-WINDOW>0</flexray:SYMBOL-WINDOW>
      <flexray:T-S-S-TRANSMITTER>7</flexray:T-S-S-TRANSMITTER>
      <flexray:WAKE-UP>
        <flexray:LISTEN-NOISE>2</flexray:LISTEN-NOISE>
        <flexray:MACRO-PER-CYCLE>8000</flexray:MACRO-PER-CYCLE>
        <flexray:MACROTICK>1</flexray:MACROTICK>
        <flexray:MAX-INITIALIZATION-ERROR>4.21</flexray:MAX-INITIALIZATION-ERROR>
        <flexray:MAX-WITHOUT-CLOCK-CORRECTION-FATAL>2</flexray:MAX-WITHOUT-CLOCK-CORRECTION-FATAL>
        <flexray:MAX-WITHOUT-CLOCK-CORRECTION-PASSIVE>2</flexray:MAX-WITHOUT-CLOCK-CORRECTION-PASSIVE>
        <flexray:NETWORK-MANAGEMENT-VECTOR-LENGTH>1</flexray:NETWORK-MANAGEMENT-VECTOR-LENGTH>
        <flexray:NUMBER-OF-MINISLOTS>646</flexray:NUMBER-OF-MINISLOTS>
        <flexray:NUMBER-OF-STATIC-SLOTS>16</flexray:NUMBER-OF-STATIC-SLOTS>
        <flexray:OFFSET-CORRECTION-START>7989</flexray:OFFSET-CORRECTION-START>
        <flexray:PAYLOAD-LENGTH-STATIC>4</flexray:PAYLOAD-LENGTH-STATIC>
        <flexray:SYNC-NODE-MAX>15</flexray:SYNC-NODE-MAX>
        <flexray:CAS-RX-LOW-MAX>99</flexray:CAS-RX-LOW-MAX>
        <flexray:BIT>0.1</flexray:BIT>
        <flexray:CYCLE>8000</flexray:CYCLE>
        <flexray:CLUSTER-DRIFT-DAMPING>3</flexray:CLUSTER-DRIFT-DAMPING>
      </fx:CLUSTER>
    </fx:CLUSTERS>
  </fx:ELEMENTS>
```

Figure 3-8: A screenshot of a FlexRay cluster as seen in a FIBEX file.

We create a `FibexChannel` object for each channel that occurs in the file and connect up to two channels to a cluster. Each channel object created contains the unique `FlexRayChannel` ID used to identify each channel object and the type (A or B). Adding a channel to a cluster means that we add the IDs of the channels to the channel ID reference vector (`ChannelRefVector`) in the cluster object.

The main configuration that takes place after the channel creation is the frame triggering section. This is where the timings of each frame are defined. One example of a frame triggering instance in

a sample FIBEX xml file is given in figure 3-9. This is a block of the beginning of the fx:FRAME-TRIGGERINGS tab under a fx:CHANNEL tab, which details the frame timing of a single frame for that channel. Here we see the timing of a frame assigned to FID 4 with offset 0 and repetition 1, as computed by the scheduler. The fx:FRAME-TRIGGERING event itself has a unique ID property as required by FIBEX. The frame that is linked to these timing parameters is referenced with its GUID in the fx:FRAME-REF tag. ID references are made as an xml property in the xml tag as ID-REF.

```

<fx:FRAME-TRIGGERINGS>
  <fx:FRAME-TRIGGERING ID="FrameTriggering_db82f582948c4b468a9dd02bce6f2db6">
    <fx:TIMINGS>
      <fx:ABSOLUTELY-SCHEDULED-TIMING>
        <fx:SLOT-ID xsi:type="flexray:SLOT-ID-TYPE">4</fx:SLOT-ID>
        <fx:BASE-CYCLE>0</fx:BASE-CYCLE>
        <fx:CYCLE-REPETITION xsi:type="flexray:CYCLE-REPETITION-TYPE">1</fx:CYCLE-REPETITION>
      </fx:ABSOLUTELY-SCHEDULED-TIMING>
    </fx:TIMINGS>
    <fx:FRAME-REF ID-REF="Frame_ae347ed9b0484270b1771b12806a46c2" />
  </fx:FRAME-TRIGGERING>

```

Figure 3-9: A screenshot of a single frame triggering under an fx:CHANNEL tab.

We assign each triggering a GUID which is set as a property of the fx:FRAME-TRIGGERING named ID. This ID is later referenced in the creation of the related ECU ports. We extract the data to be filled in the timing section from the ScheduledFrames vector in the scheduler. We use the information in this vector extract the FID, offset and repetition for each frame, and add them to FrameTriggerMap of the related FibexChannel object. FrameTriggerMap is a map of strings containing the frame triggering IDs to a vector of integers containing the FID, offset and repetition. The contents of the vector for each triggering is written under the fx:SLOT-ID, fx:BASE-CYCLE and fx:CYCLE-REPETITION tabs respectively, for each triggering instance.

There are two parts to the configuration of each ECU in the cluster. The first is the ‘controller’ part where the node parameters for the ECU are configured under the fx:CONTROLLERS tag. The second is the ‘connectors’ part where virtual input and output ports are configured for each frame timing occurring in the network, tagged fx:CONNECTORS in FIBEX. We handle this part at the same time when the frame triggering is created for each channel. When we create a new FibexECU object representing each FlexRay node in the cluster, the unique ECU GUID for the object, ECU name and the controller parameters are also configured.

We record the ECU GUID, name and controller parameters in their respective data structures in the FibexECU objects. When we use this data to write the xml file, this part is shaped as in the following FIBEX sample. These parameters should be configured to reflect the properties of the used boards and the application at hand. Figure 3-10 shows the content of the fx:CONTROLLER tab for an ECU in a sample FIBEX file.

```

<fx:ECUS>
  <fx:ECU ID="Ecu_93562be263244a7ab6cb3b947c46edf7">
    <ho:SHORT-NAME>Ecu1</ho:SHORT-NAME>
    <fx:CONTROLLERS>
      <fx:CONTROLLER xsi:type="flexray:CONTROLLER-TYPE" ID="FlexRayController_ef08fc2fa22e4d70bee0c8a882482a4e">
        <ho:SHORT-NAME>Controller1</ho:SHORT-NAME>
        <fx:VENDOR-NAME>Bosch</fx:VENDOR-NAME>
        <fx:CHIP-NAME>ERay</fx:CHIP-NAME>
        <fx:TERMINATION-IMPEDANCE>0</fx:TERMINATION-IMPEDANCE>
        <fx:TRANSCIEVER />
        <flexray:KEY-SLOT-USAGE>
          <flexray:STARTUP-SYNC>4</flexray:STARTUP-SYNC>
        </flexray:KEY-SLOT-USAGE>
        <flexray:MAX-DYNAMIC-PAYLOAD-LENGTH>16</flexray:MAX-DYNAMIC-PAYLOAD-LENGTH>
        <flexray:CLUSTER-DRIFT-DAMPING>3</flexray:CLUSTER-DRIFT-DAMPING>
        <flexray:DECODING-CORRECTION>40</flexray:DECODING-CORRECTION>
        <flexray:LISTEN-TIMEOUT>641924</flexray:LISTEN-TIMEOUT>
        <flexray:MAX-DRIFT>962</flexray:MAX-DRIFT>
        <flexray:LATEST-TX>641</flexray:LATEST-TX>
        <flexray:MICRO-PER-CYCLE>320000</flexray:MICRO-PER-CYCLE>
        <flexray:OFFSET-CORRECTION-OUT>206</flexray:OFFSET-CORRECTION-OUT>
        <flexray:RATE-CORRECTION-OUT>962</flexray:RATE-CORRECTION-OUT>
        <flexray:SAMPLES-PER-MICROTICK>2</flexray:SAMPLES-PER-MICROTICK>
        <flexray:DELAY-COMPENSATION-A>0</flexray:DELAY-COMPENSATION-A>
        <flexray:DELAY-COMPENSATION-B>0</flexray:DELAY-COMPENSATION-B>
        <flexray:WAKE-UP-PATTERN>2</flexray:WAKE-UP-PATTERN>
        <flexray:ALLOW-HALT-DUE-TO-CLOCK>false</flexray:ALLOW-HALT-DUE-TO-CLOCK>
        <flexray:ACCEPTED-STARTUP-RANGE>82</flexray:ACCEPTED-STARTUP-RANGE>
        <flexray:MACRO-INITIAL-OFFSET-A>18</flexray:MACRO-INITIAL-OFFSET-A>
        <flexray:MACRO-INITIAL-OFFSET-B>18</flexray:MACRO-INITIAL-OFFSET-B>
        <flexray:MICRO-INITIAL-OFFSET-A>0</flexray:MICRO-INITIAL-OFFSET-A>
        <flexray:MICRO-INITIAL-OFFSET-B>0</flexray:MICRO-INITIAL-OFFSET-B>
        <flexray:MICRO-PER-MACRO-NOM>40</flexray:MICRO-PER-MACRO-NOM>
      </fx:CONTROLLER>
    </fx:CONTROLLERS>
  </fx:ECU>
</fx:ECUS>

```

Figure 3-10: A screenshot of the ECU controller parameters as seen in a FIBEX file.

We configure the connectors part once the frames to be sent are introduced, after the creation of FibexECU objects. This part will be discussed in section 3.3.3.

The creation of the FIBEX data units, namely frames, PDUs and signals, are handled automatically during the extraction of the schedule. This process will also be explained in section 3.3.3 although the data structures that are used need to be defined beforehand.

The data units have a different hierarchy for different versions of FIBEX. For v2 signals are packed in frames and PDUs are not supported. For v3 and higher, signals are packed into PDUs, which are then packed into frames. We call every sub data unit that is packed into another an instance. To determine which unit is packed into which, their GUIDs are used when defining the instances.

The FibexFrame objects have lengths as parameters and automatically assigned frame names. Each FibexFrame has a vector of objects we defined as SubInstance. SubInstance objects contain the GUIDs of the PDUs packed into this frame and the bit position indicating the place where the PDU is located (the SubInstance objects contain signal information instead of PDUs for versions lower than 3.0). These SubInstance objects also have their own GUIDs as required by the FIBEX standard. The final view of a frame appearing in a sample FIBEX file with two PDUs is given in Figure 3-11. This example is an 8 byte frame named Frame1 and contains two PDUs. Each PDU is placed under an fx:PDU-INSTANCE tag. The fx:PDU-REF tag has an

ID-REF property which holds the GUID for the fx:PDU tag representing the PDU related to that instance.

```
<fx:FRAMES>
  <fx:FRAME ID="Frame_ae347ed9b0484270b1771b12806a46c2">
    <ho:SHORT-NAME>Frame1</ho:SHORT-NAME>
    <fx:BYTE-LENGTH>8</fx:BYTE-LENGTH>
    <fx:PDU-INSTANCES>
      <fx:PDU-INSTANCE ID="PduInstance_1113c9fc7d05479393bcbcf38adf3f63">
        <fx:PDU-REF ID-REF="Pdu_b57bee57e88d42bbb3385b07afdd19f2" />
        <fx:BIT-POSITION>0</fx:BIT-POSITION>
        <fx:IS-HIGH-LOW-BYTE-ORDER>true</fx:IS-HIGH-LOW-BYTE-ORDER>
      </fx:PDU-INSTANCE>
      <fx:PDU-INSTANCE ID="PduInstance_b470a69b7a52462b9734dae57f42e192">
        <fx:PDU-REF ID-REF="Pdu_191a57ad90204b27903896e9c76c1fac" />
        <fx:BIT-POSITION>32</fx:BIT-POSITION>
        <fx:IS-HIGH-LOW-BYTE-ORDER>true</fx:IS-HIGH-LOW-BYTE-ORDER>
      </fx:PDU-INSTANCE>
    </fx:PDU-INSTANCES>
  </fx:FRAME>
</fx:FRAMES>
```

Figure 3-11: A screenshot of a single frame with two PDUs assigned to it in a FIBEX file.

PDUs have a similar format to frames. Therefore we have defined the `FibexPDU` objects in a similar way to the `FibexFrame` objects. One difference is an extra parameter, PDU type, which we added into the data structure. The instances are instead signal instances that appear as `SubInstance` objects that contain signal information. Figure 3-12 is an example view of an 8 byte PDU with a single signal assigned to it. The only difference to the structure of a FIBEX frame, apart from the signal instances, is the addition of `fx:PDU-TYPE`. We set the PDU type as the default `APPLICATION` type, since although the type is mandatory to be set in the FIBEX standard, it is an irrelevant parameter for our test cases.

```
<fx:PDUS>
  <fx:PDU ID="Pdu_b57bee57e88d42bbb3385b07afdd19f2">
    <ho:SHORT-NAME>Pdu1</ho:SHORT-NAME>
    <fx:BYTE-LENGTH>8</fx:BYTE-LENGTH>
    <fx:PDU-TYPE>APPLICATION</fx:PDU-TYPE>
    <fx:SIGNAL-INSTANCES>
      <fx:SIGNAL-INSTANCE ID="PduSignalInstance_2777ad2a6883483c855f1f74afdb978c">
        <fx:BIT-POSITION>0</fx:BIT-POSITION>
        <fx:IS-HIGH-LOW-BYTE-ORDER>true</fx:IS-HIGH-LOW-BYTE-ORDER>
        <fx:SIGNAL-REF ID-REF="Signal_648e7323fe134df5a59c6da7393fef25" />
      </fx:SIGNAL-INSTANCE>
    </fx:SIGNAL-INSTANCES>
  </fx:PDU>
</fx:PDUS>
```

Figure 3-12: A screenshot of a single PDU with a single signal assigned in a FIBEX file.

Signals in the FIBEX standard have a different structure than the other data units. Signals are defined under the elements tab but the coding schemes of the signals are defined as coding objects,

which are not network elements. The codings are instead defined under the fx:PROCESSING-INFORMATION tab as opposed to the fx:ELEMENTS tab which we write for all the network elements. The codings are then associated to the signals using the GUID of the coding object.

The parameters included in the FibexSignal object are the signal GUID, signal name and a string which holds the GUID of the selected FibexCoding object. The codings in FIBEX define many schemes and computation methods but most are optional. For our purposes the FibexCoding objects contain the coding GUID and the bit length, which defines the signal length for the signals which select this coding. We then write the information as in the FIBEX sample given in Figure 3-13. As with all GUID references, the coding ID should match the related coding reference ID-REF.

```

<fx:SIGNALS>
  <fx:SIGNAL ID="Signal_648e7323fe134df5a59c6da7393fef25">
  <fx:SIGNAL ID="Signal_b1e80c8d8c7848a6b50816fbf23d215f">
    <ho:SHORT-NAME>Signal2</ho:SHORT-NAME>
    <fx:CODING-REF ID-REF="Coding_bd33aa73c8844dc9871797ed7f6e4765" />
  </fx:SIGNAL>
</fx:SIGNALS>
</fx:ELEMENTS>
<fx:PROCESSING-INFORMATION>
  <fx:CODINGS>
    <fx:CODING ID="Coding_1f5a6e45df994b56a12fffff8adaf620">
    <fx:CODING ID="Coding_bd33aa73c8844dc9871797ed7f6e4765">
      <ho:SHORT-NAME>Coding2</ho:SHORT-NAME>
      <ho:CODED-TYPE CATEGORY="STANDARD-LENGTH-TYPE">
        <ho:BIT-LENGTH>8</ho:BIT-LENGTH>
      </ho:CODED-TYPE>
    </fx:CODING>
  </fx:CODINGS>
</fx:PROCESSING-INFORMATION>

```

Figure 3-13: A screenshot depicting both a signal and its related coding in a FIBEX file. The relation is established by referring to the coding ID as highlighted.

When we use the `FXFIBEX.Write()` method, the code goes through all the created objects and the contained information in order. After setting up the network by creating the necessary objects and establishing their connections, we then extract the schedule from the scheduler code and insert the information into the appropriate objects.

3.3.3 Creating Frame Triggerings (Timings)

The timing of each frame in the network has to be configured for the configuration to be complete. For the static segment, we use a function in the `FXFIBEX` object called `CreateTriggeringsA` for timings involving channel A and `CreateTriggeringsB` for channel B. The tasks handled by these functions are as follows:

1. Extract the static schedule for one node item (Get the `ScheduledFrames` vector).
2. Create a frame triggering entity with its own GUID.

3. Insert the timing of this entity into the channel (With frame triggering GUID, FID, offset and repetition)
4. Add an output port for the ECU from where this frame is set, link the GUID of this port with the frame triggering GUID.
5. Add an input port for each destination ECU and link the GUIDs.
6. Create a frame and a PDU instance to this frame (no PDUs are needed for versions lower than 3.0)
7. Create the signals and their respective codings using the size information in the ScheduledFrames vector. Link the signals to the frame by creating signal instances in the PDU belonging to that frame.

The function takes one `FXNodeItem` object from the scheduler and a `FibexEcu` object from the FIBEX part to establish that these items represent the same sender node. The function also takes a `FibexChannel` object to operate on. `FibexChannel` objects of type A should be input to `CreateTriggeringsA` function and type B for its own function.

Since this function links a `FXNodeItem` with a `FibexEcu`, we compute the `EcuLatestTX` parameter here even though it is related to the dynamic segment. We compute `LatestTx` by subtracting the duration of the longest sporadic message for that node (in minislots) from the total total minislot number as in equation (3) in section 2.3.3.

The complete configuration information for each scheduled message is gathered in three different places in the FIBEX file. The first is the `FibexChannel` where the timing parameters per frame triggers are set. The second is the connectors tab of `FibexECUs` where the triggers are associated with the virtual input and output ports of the nodes according to their delivery. The final is the `FibexFrame` object carrying the message parameters and the grouped signals, which is linked to the frame triggering. This three-way-linkage via GUID referrals is depicted in the following figures. Figure 3-14 shows the view of a frame triggering event as an example.

```
<?xml version="1.0" encoding="utf-8"?>
<fx:FIBEX xmlns:asr30nm="http://www.asam.net/xml/ebx/nm/asr30" xmlns:ho="http://www.asam.net/xml" xmlns:lin="
  <fx:PROJECT ID="ProjectType_770b989263269f59669b10e443cc47e0">
    <ho:SHORT-NAME>ShortName</ho:SHORT-NAME>
  </fx:PROJECT>
  <fx:ELEMENTS>
    <fx:CLUSTERS>
      <fx:CHANNELS>
        <fx:CHANNEL xsi:type="flexray:CHANNEL-TYPE" ID="FlexRayChannel_4a578de3a826968509374ffe5fe999">
          <ho:SHORT-NAME>ChannelA</ho:SHORT-NAME>
          <fx:FRAME-TRIGGERINGS>
            <fx:FRAME-TRIGGERING ID="FrameTriggering_115722c4788016842b2049e1589e481d">
              <fx:TIMINGS>
                <fx:ABSOLUTELY-SCHEDULED-TIMING>
                  <fx:SLOT-ID xsi:type="flexray:SLOT-ID-TYPE">2</fx:SLOT-ID>
                  <fx:BASE-CYCLE>0</fx:BASE-CYCLE>
                  <fx:CYCLE-REPETITION xsi:type="flexray:CYCLE-REPETITION-TYPE">2</fx:CYCLE-REPETITION>
                </fx:ABSOLUTELY-SCHEDULED-TIMING>
              </fx:TIMINGS>
              <fx:FRAME-REF ID-REF="Frame_c632fc43ed82ab85d1b4bd6766c2955"/>
            </fx:FRAME-TRIGGERING>
            <fx:FRAME-TRIGGERING ID="FrameTriggering_180b731c4a0969f2216b8f7567851c2c">
              <fx:TIMINGS>
                <fx:ABSOLUTELY-SCHEDULED-TIMING>
                  <fx:SLOT-ID xsi:type="flexray:SLOT-ID-TYPE">2</fx:SLOT-ID>
```

Figure 3-14: A screenshot of a single frame triggering event in a complete FIBEX file sample. The frame triggering ID in question is highlighted.

When we create a frame triggering, we also create the relevant `FibexFrame` object and the ECU input and output ports simultaneously. FIBEX requires ECUs to have an output port for each sent frame with its unique ID. Under these `fx:OUTPUT-PORT` tags, we enter the ID of the triggering event in the `fx:FRAME-TRIGGERING-REF` tag. Figure 3-15 shows this referencing for the sender of the frame described in Figure 3-14.

```

</fx:CHANNEL>
</fx:CHANNELS>
<fx:ECUS>
  <fx:ECU ID="Ecu_61dbe83f3d39b6ac53f7cb3746ffe416">
    <ho:SHORT-NAME>Ecu1</ho:SHORT-NAME>
    <fx:CONTROLLERS>
      <fx:CONTROLLER xsi:type="flexray:CONTROLLER-TYPE" ID="FlexRayController_12aaaf432a2b803d552671043fbb">
    </fx:CONTROLLERS>
    <fx:CONNECTORS>
      <fx:CONNECTOR xsi:type="flexray:CONNECTOR-TYPE" ID="FlexRayConnector_420f4ccb363d5b053d89bfeeba5890f">
        <fx:CHANNEL-REF ID-REF="FlexRayChannel_4a578de3a826968509374ffe5fe999"/>
        <fx:CONTROLLER-REF ID-REF="FlexRayController_12aaaf432a2b803d552671043fbb58e1"/>
    </fx:CONNECTORS>
    <fx:INPUTS>
    </fx:INPUTS>
    <fx:OUTPUTS>
      <fx:OUTPUT-PORT ID="EcuPort_58e7e9d565c7d498785ac51e289c82f0">
        <fx:FRAME-TRIGGERING-REF ID-REF="FrameTriggering_2568860476229f28687febc03e70404e"/>
        <fx:COMPLETE-FRAME />
      </fx:OUTPUT-PORT>
      <fx:OUTPUT-PORT ID="EcuPort_6e7f8455288734eb3b3a9d70444e2f55">
        <fx:FRAME-TRIGGERING-REF ID-REF="FrameTriggering_5afed5a958a26f2865de2597f56f0e5"/>
        <fx:COMPLETE-FRAME />
      </fx:OUTPUT-PORT>
      <fx:OUTPUT-PORT ID="EcuPort_7541630e631d8aa67e63cdae5fb9b678">
        <fx:FRAME-TRIGGERING-REF ID-REF="FrameTriggering_180b731c4a0969f2216b8f7567851c2c"/>
        <fx:COMPLETE-FRAME />
      </fx:OUTPUT-PORT>
      <fx:OUTPUT-PORT ID="EcuPort_77cfd41b3d36b5ec6e00abb970ca7082">
        <fx:FRAME-TRIGGERING-REF ID-REF="FrameTriggering_115722c4788016842b2849e1589e481d"/>
        <fx:COMPLETE-FRAME />
      </fx:OUTPUT-PORT>
      <fx:OUTPUT-PORT ID="EcuPort_df4b5bb3058419a48bc324d13b19238">

```

Figure 3-15: A screenshot of the `fx:CONNECTORS` tab of the ECU sending the frame described in Figure 3-14. The `fx:INPUTS` tag has been collapsed to simplify the view. The highlighted ID-REF matches the unique ID of the frame triggering event given in Figure 3-14.

We also do the same referencing for the input ports of the receiving ECUs. The input port of the ECU receiving our example frame can be seen in Figure 3-16.

```

</fx:ECU>
<fx:ECU ID="Ecu_70b80fb64cc5cd0745224e774c2cbff5">
  <ho:SHORT-NAME>Ecu3</ho:SHORT-NAME>
  <fx:CONTROLLERS>
  <fx:CONNECTORS>
    <fx:CONNECTOR xsi:type="flexray:CONNECTOR-TYPE" ID="FlexRayConnector_323a48f7763436753e2173b83584f">
      <fx:CHANNEL-REF ID-REF="FlexRayChannel_4a578de3a826968509374ffe5fe999"/>
      <fx:CONTROLLER-REF ID-REF="FlexRayController_7810dbf61aa647fd6ce3cb669a5a016"/>
      <fx:INPUTS>
        <fx:INPUT-PORT ID="EcuPort_1b57d3a81b6dbfe17710d0db45891826">
          <fx:FRAME-TRIGGERING-REF ID-REF="FrameTriggering_2568860476229f28687feb03e70404e"/>
          <fx:COMPLETE-FRAME />
        </fx:INPUT-PORT>
        <fx:INPUT-PORT ID="EcuPort_252f243fd78c44f4b924d8a7604a7fb">
          <fx:FRAME-TRIGGERING-REF ID-REF="FrameTriggering_115722c4788016842b2849e1589e481d"/>
          <fx:COMPLETE-FRAME />
        </fx:INPUT-PORT>
        <fx:INPUT-PORT ID="EcuPort_2747efd754709a6845652cbf432d10a5">

```

Figure 3-16: A screenshot of the fx:CONNECTORS tab of the ECU receiving the frame sent in Figure 3-14. The highlighted ID-REF matches the unique ID of the frame triggering event given in Figure 3-14.

Figure 3-17 shows the actual frame in the FIBEX file that we use in this example. The ID of the frame is the same in the ID-REF property of the fx:FRAME-REF in Figure 3-14.

```

</fx:PDU>
</fx:PDUS>
<fx:FRAMES>
  <fx:FRAME ID="Frame_76fe8ae425863d816bad9ec8366042ff">
    <ho:SHORT-NAME>Frame1</ho:SHORT-NAME>
    <fx:BYTE-LENGTH>24</fx:BYTE-LENGTH>
    <fx:PDU-INSTANCES>
      <fx:PDU-INSTANCE ID="PduInstance_702d7851697249e31d9be946caa0f3e">
        <fx:PDU-REF ID-REF="Pdu_3ab63f002c6a776027182bd65f7b169"/>
        <fx:BIT-POSITION>0</fx:BIT-POSITION>
        <fx:IS-HIGH-LOW-BYTE-ORDER>false</fx:IS-HIGH-LOW-BYTE-ORDER>
      </fx:PDU-INSTANCE>
    </fx:PDU-INSTANCES>
  </fx:FRAME>
  <fx:FRAME ID="Frame_c632fc43ed82ab85d1b4bd6766c2955">
    <ho:SHORT-NAME>Frame2</ho:SHORT-NAME>
    <fx:BYTE-LENGTH>24</fx:BYTE-LENGTH>
    <fx:PDU-INSTANCES>
      <fx:PDU-INSTANCE ID="PduInstance_7be6acc9532d1aa34c7a491011dfa2a8">
        <fx:PDU-REF ID-REF="Pdu_359c77d515bccbed7ce5d3f83cf72e70"/>
        <fx:BIT-POSITION>0</fx:BIT-POSITION>
        <fx:IS-HIGH-LOW-BYTE-ORDER>false</fx:IS-HIGH-LOW-BYTE-ORDER>
      </fx:PDU-INSTANCE>
    </fx:PDU-INSTANCES>
  </fx:FRAME>
  <fx:FRAME ID="Frame_87214eada802337993a57113e103a7">

```

Figure 3-17: A screenshot of a sample frame in a FIBEX file. The highlighted frame is referred in the triggering event described in Figure 3-14 and the unique ID here matches the frame ID-REF in said figure.

As previously mentioned, each frame, PDU, signal and signal coding object have to have a unique ID and a short-name which need not be unique. We use static integers to number the consequently created objects to avoid confusion. These integers are stored in the `FXFIBEX` object as `framecount`, `signalcount`, `pducount` and `codingcount` which are all initialized as zero. When the objects are created in one of the `CreateTriggerings` functions, these incremented integers are appended to the short-name text. For example created frames go `Frame1`, `Frame2` and so on. When we run the `CreateDynamicTriggerings` function the naming continues from where the previous function left off.

Both `CreateTriggerings` functions need three inputs to operate. The first is the `FXNodeItem` object representing the node from where the static frames are sent called `sendernode` in the function. The second input is the `FibexEcu` object to be associated with that particular `FXNodeItem` called `senderecu`. The last is the respective `FibexChannel` object as in the channel object that represents channel A is the input to the `CreateTriggeringsA` function and is named `fchannel`.

The function loops through all the `Message` objects that are present in the `ScheduledMessages` vector of the input `NodeItem`. For each message we create a frame triggering identifier with its GUID. We then make an entry in the output port of the `senderecu` object with the following lines.

```
std::string FrameTrig ="FrameTriggering_";
FrameTrig.append( GenerateGUID() );
senderecu.InsertOutputA(FrameTrig);
```

The `InsertOutputA` function creates a new port entry for each entered unique frame trigger and pairs the port ID and the trigger ID in the `EcuOutputsA` set. We use sets to store that information so that we can check if any frame triggering ID has already been assigned a port, to avoid duplicates.

```
void FibexEcu::InsertOutputA(std::string ftrig){
std::string ecuport = "EcuPort_";
ecuport.append(GenerateGUID());
if (CurrentFrames.count(ftrig)==0)
    EcuOutputsA.insert (pair<std::string, std::string> (ecuport,
ftrig) );
CurrentFrames.insert(ftrig);
}
```

After this we create the PDU where the signals belonging to the current frame are to be grouped. We group the signals into this PDU by adding them back to back considering their bit-lengths. In FIBEX the signals are placed in a PDU by setting its bit position. For each signal grouped in the current message, we create the `FibexSignal` object with its short-name and GUID and the `FibexCoding` object with the same required identifiers and the signal length in bits. We give the first signal the bit position 0 (stored in the integer `signalbits`). We then add the signal size to determine the next bit position. The `AssignSignal` method of the `FibexPDU` class sets the signal object in the PDU object with the entered bit position.

```
codingcount++;
FibexCoding aCode("Coding" + boost::lexical_cast<string>(
codingcount ), SignalIt->GetSize() );
Codings.push_back(aCode);
```

```

signalcount++;
FibexSignal aSignal("Signal" + boost::lexical_cast<string>(
signalcount ) );
aSignal.SelectCoding(aCode);
Signals.push_back(aSignal);

aPDU.AssignSignal(aSignal, signalbits, "false");
signalbits = signalbits + SignalIt->GetSize();

```

The destinations of the signals are also extracted during this process. We utilize a set called `DestinationSet` in the `CreateTriggerings` function where every new destination node number that occurs is inserted. Therefore the final content is the union of the destinations of all the signals in the message. The frame encapsulating this message should be sent to all the destinations in this set. We specify this by entering the current triggering GUID into the `EcuInputsA` (or `B`) vectors of the `FibexEcu` objects belonging to those destinations. We again use the set scheme here to eliminate duplicates.

```

if ( DestinationSet.count( SignalIt->GetDestinationId() ) == 0 ) {
    Ecus.at( (SignalIt->GetDestinationId() ) - 1 )->
    InsertInputA(FrameTrig);
}
DestinationSet.insert (SignalIt->GetDestinationId() );

```

After grouping the signals and determining the destinations, we finally create the `FibexFrame` object and group everything together. Both the frame and the PDU objects are set the same length in bytes according to the size of the current message. The `AssignPDU` method then assigns the PDU into the created frame. We then add the frame and PDU objects into their relevant vectors in the `FXFIBEX` object to be used in the `Write` function in the end.

The last line creates the triggering under the channel object using all the needed parameters for the `FibexChannel::InsertFrameTriggering` method: Frame triggering ID, FID, repetition, offset and the ID of the related `FibexFrame` object.

To finalize the static segment configuration, we configure the synchronization frames that should be exchanged. We use the `FXFIBEX::ConfigureStartup` method to automatically generate two synchronization frames that are transmitted between ECUs at the first two elements of the `FXFIBEX.Ecus` vector. A single `FibexFrame` is created to represent both synchronization frames. The frame is configured as a periodic frame with 0 offset and 1 repetition. We automatically set the FIDs of these frames as the last and the second to last slots of the static segment. We configure the timings of these frames in the `FibexChannel` objects in the `Channels` vector of `FXFIBEX` with the same `FibexChannel::InsertFrameTriggering` method as before.

The configuration of the triggerings of the dynamic segment is much simpler than its static counterpart. This process is handled by the `CreateDynamicTriggerings` function. This function takes each entry in the `mScheduledMessagesDynamic` and creates a frame triggering entry with its own GUID as before. It then creates an output port for the node sending that message where the GUID is referred. In the same manner input ports for each receiving node are also created with the same referral. The frame and PDU objects are also created in the same manners as before and timing information is input to the related `FibexChannel` objects.

3.3.4 Writing the FIBEX File

After we configure every necessary entity in the network, the data has to be translated into a configuration tool for the generation of the configuration files for each node. While normally this data is entered by hand to the configuration tool, we instead generate the standard FIBEX file which can be imported to all configuration tools.

In order to generate the FIBEX file which is of .xml format, we use the tree structure of the libxml2 free library [9]. The library works by assigning a document pointer to the file to be created. Then the root nodes are defined as xml pointers of type `xmlNodePtr`, which is a property of this library. The children of these nodes are then created and added as necessary, again as `xmlNodePtr` types. After all nodes are entered to according to their parents, the `xmlSaveFormatFileEnc` function is run, using a file name, coding standard and the document pointer. This function recursively goes through each `xmlNodePtr` and its children to write out the xml file using the standard xml format.

In our project, when we create a new `FXFIBEX` object, the document pointer and the main `xmlNodePtr` pointers are automatically created. Whenever we introduce a new network element as a new Fibex object in the code, we store the configured parameters as data entries in this object. When at the end before we write the file, we enter all the data in all the present objects in their respective nodes in the xml tree.

To provide an example, code segments that we use to configure the cluster are given below.

```
/* fx_fibex.h */
class FXFIBEX {
...
/** Node Pointers to be used during the writing operations */
xmlNodePtr root_node, node, node1, node2, node3, node4;
/** Node Pointers for the main nodes occurring in a FIBEX file */
xmlNodePtr fibex_node;
xmlNodePtr clusters_node;

/** document pointer for the libxml2 project */
xmlDocPtr fibexdoc;

/** Method to write the document in the specified xml file */
void Write();
}

/* fx_fibex.cpp */
/* Fibex object constructor */
FXFIBEX::FXFIBEX(float fVersion, FXScheduler sced){
...
fibexdoc = xmlNewDoc(BAD_CAST "1.0");
fibex_node = xmlNewNode(NULL, BAD_CAST "fx:FIBEX");
...
/* fibex_node is set as the root node of the project */
xmlDocSetRootElement(fibexdoc, fibex_node);
...
/* Elements is a child of Fibex, Clusters is a child of Elements
*/
elements_node = xmlNewNode(NULL, BAD_CAST "fx:ELEMENTS");
```

```

xmlAddChild(fibex_node, elements_node);

clusters_node = xmlNewNode(NULL, BAD_CAST "fx:CLUSTERS");
xmlAddChild(elements_node, clusters_node);
...
}

/* The Write() method is run at the end */
void FXFIBEX::Write() {
...
/* An iterator named It goes through all clusters in the FXFIBEX
object */

for( ; It != EndIt; It++){

root_node      =      xmlNewChild(clusters_node,      NULL,      BAD_CAST
"fx:CLUSTER", NULL);

xmlNewProp(root_node,      BAD_CAST      "xsi:type",      BAD_CAST
"flexray:CLUSTER-TYPE");

xmlNewProp(root_node,      BAD_CAST      "ID",      BAD_CAST      (*It)-
>GetCID().c_str() );

xmlNewChild(root_node, NULL, BAD_CAST "ho:SHORT-NAME",
BAD_CAST (*It)->GetCName().c_str() );

xmlNewChild(root_node, NULL, BAD_CAST "fx:SPEED",
BAD_CAST (boost::lexical_cast<string>      ((*It)->GetCSpeed()      )
).c_str());
...
}

```

As seen in the cluster example, the parameter values stored in the relevant `FibexCluster` object are acquired using the appropriate `Get` functions. We then enter these values into their respective nodes in the xml tree as properties, with the appropriate FIBEX tags. The other network elements are also configured and then entered into the document in the same manner. We then run the method which creates the xml file using the following command.

```
xmlSaveFormatFileEnc(FibexName.c_str(), fibexdoc, "utf-8", 1);
```

The parameters are the file name, document pointer, encoding and formatting spaces. This command begins by adding the set prefixes to the xml file and writing the contents of the root node, which we had set as `fibex_node`. Then the three children of this node are written `project_node`, `elements_node` and `processinfo_node`. Afterwards the next level of children is written until the leafes of the tree structure are reached. A sample view of the final result is given in Figure 3-17. This sample file configures a cluster with two channels and six ECUs. The configuration parameters and the data unit configuration are not given to have a better look at the xml tree.

```

<?xml version="1.0" encoding="utf-8"?>
<fx:FIBEX xmlns:asr30nm="http://www.asam.net/xml/fbx/nm/asr30" xmlns:hc="http://www.asam.net/xml" xmlns:lin="htt
<fx:PROJECT ID="ProjectType e39731367d11437b8d6d53c4824ca8b2">
  <fx:ELEMENTS>
    <fx:CLUSTERS>
      <fx:CLUSTER xsi:type="flexray:CLUSTER-TYPE" ID="FlexRayCluster 599e7abb60754e4e917f1cb1577dbdle">
    </fx:CLUSTERS>
    <fx:CHANNELS>
      <fx:CHANNEL xsi:type="flexray:CHANNEL-TYPE" ID="FlexRayChannel 4783c28f38134be5bb7ea9357afb94de">
      <fx:CHANNEL xsi:type="flexray:CHANNEL-TYPE" ID="FlexRayChannel 6147508f7fdd41b9bf5b2f66bf838fc0">
    </fx:CHANNELS>
    <fx:ECUS>
      <fx:ECU ID="Ecu 93562be263244a7ab6cb3b947c46edf7">
      <fx:ECU ID="Ecu fec0066191914b9fb1e8fc8da392fac7">
      <fx:ECU ID="Ecu d198eace817643f78d581015c3b8e692">
      <fx:ECU ID="Ecu 6d3671c50e7b469dbeaaf549bc825651">
      <fx:ECU ID="Ecu f87a2f909656456193ff377c1d352e44">
      <fx:ECU ID="Ecu e609622fcf8e423781941169188cc775">
    </fx:ECUS>
    <fx:PDUS>
    <fx:FRAMES>
    <fx:SIGNALS>
  </fx:ELEMENTS>
  <fx:PROCESSING-INFORMATION>
</fx:FIBEX>

```

Figure 3-18: The final overview of a complete FIBEX file. The content of the main xml tags have been collapsed so only the network elements are seen.

To summarize the entirety of this project, we first provide the network nodes, schedule specific input parameters and the signal set as the input to the scheduler. The scheduler then packs the signals into messages and computes the schedule. We then use the data structures resulting from the scheduler code to come up with the exact slot allocations for each frame. The remaining necessary parameters are then computed and entered into their respective Fibex code objects. Fibex objects for the data units and their timings are also created here. Finally the Fibex code uses the resulting data structure to generate the text file using the xml library. The resulting xml file is a proper FIBEX file that contains all the necessary information needed to represent the network and the schedule and is compatible with any FlexRay configuration software.

CHAPTER 4

TESTS AND EVALUATION

4.1 Experiment Setups and Parameters

For our first experiment we use a FlexRay network with 3 ECUs. We will refer to these nodes as ECU 1, 2 and 3 from now on, in addition to naming the related software objects with the same numbering. To define our network, we first create three FXNodeItem objects to represent the ECUs and set the input parameters using their respective set functions in the scheduler. We use the same set of input parameter values in the upcoming experiments, which can be seen in Table 4-1.

Table 4-1: Schedule specific input parameter values used in the experiments.

| Name | Value |
|---|-------------------|
| <i>gPayloadLengthStatic</i> | 12 two byte words |
| <i>gdDynamicSlotIdlePhase</i> | 1 Minislot |
| <i>gdMinislotActionPointOffset</i> | 1 MT |
| <i>gdActionPointOffset</i> | 5 MT |
| <i>gdCycle</i> | 5000 μ s |
| <i>gdSymbolWindow</i> | 52 MT |
| <i>gdMacrotick</i> | 1 μ s |
| <i>gdMinislot</i> | 3 MT |
| <i>gdTSSTransmitter</i> | 7 gdBit |
| <i>pChannels</i> | A&B |
| <i>pMicroPerCycle</i> (node parameter; is set the same for all nodes) | 400000 μ T |
| <i>pdMicrotick</i> (node parameter; is set the same for all nodes) | 0.0125 μ s |

For the second experiment, we remove ECU 3 from the system and add more signals to the signal set to see the changes in the schedule, in addition to displaying the convenience of the automatic configuration process.

4.2 Experiment with Three FlexRay Nodes

Having defined the network as such, we input the signal set to be used in the communication. The signal set provides the following information: Signal type (periodic / sporadic), sending and

receiving node ID, signal ID, size, period and deadline. The signal ID is a unique number representing the signal. The sending and receiving node IDs are for the numbers representing the ECUs, which are ECU number 1, 2 and 3 as mentioned before. Both periodic and sporadic signals are represented with the same parameters although period has a different meaning for both. For periodic signals the period is the fixed amount of time between each generation of that particular signal. For sporadic signals, the period value is the minimum time taken between two consequent generations of a sporadic signal. The deadlines are the timing requirements of each signal and the signals must reach their receivers within that time limit. The existing scheduler also takes into account these deadlines when performing signal packing. The signals set we used for the experiment is given in Table 4-2.

Table 4-2: The signal set used in the first experiment.

| Signal ID | Signal Type | Sending Node | Receiving Node | Size (Bits) | Repetition (μ s) | Deadline (μ s) |
|-----------|-------------|--------------|----------------|-------------|-----------------------|---------------------|
| 1 | Periodic | 2 | 1 | 64 | 10 | 10 |
| 2 | Periodic | 2 | 1 | 64 | 5 | 5 |
| 3 | Periodic | 2 | 1 | 64 | 20 | 20 |
| 4 | Periodic | 2 | 1 | 64 | 10 | 10 |
| 5 | Periodic | 1 | 3 | 64 | 10 | 10 |
| 6 | Periodic | 2 | 1 | 64 | 10 | 10 |
| 7 | Periodic | 1 | 3 | 64 | 10 | 10 |
| 8 | Periodic | 1 | 3 | 64 | 10 | 10 |
| 9 | Periodic | 2 | 1 | 64 | 10 | 10 |
| 10 | Periodic | 2 | 1 | 64 | 10 | 10 |
| 11 | Periodic | 2 | 1 | 64 | 10 | 10 |
| 12 | Periodic | 2 | 1 | 64 | 20 | 20 |
| 13 | Periodic | 2 | 1 | 64 | 10 | 10 |
| 14 | Periodic | 2 | 1 | 64 | 20 | 20 |
| 15 | Periodic | 2 | 1 | 64 | 10 | 10 |
| 16 | Periodic | 1 | 3 | 64 | 10 | 10 |
| 17 | Periodic | 1 | 3 | 64 | 10 | 10 |
| 18 | Periodic | 1 | 3 | 32 | 10 | 10 |
| 19 | Periodic | 1 | 2 | 64 | 100 | 100 |
| 20 | Periodic | 1 | 2 | 64 | 50 | 50 |
| 21 | Periodic | 2 | 1 | 16 | 100 | 100 |
| 22 | Periodic | 2 | 3 | 64 | 100 | 100 |
| 23 | Periodic | 2 | 3 | 64 | 100 | 100 |
| 24 | Periodic | 3 | 1 | 16 | 250 | 250 |
| 25 | Periodic | 3 | 1 | 8 | 500 | 500 |
| 26 | Periodic | 1 | 2 | 24 | 250 | 250 |
| 27 | Periodic | 2 | 3 | 64 | 10 | 10 |
| 28 | Periodic | 1 | 2 | 32 | 100 | 100 |
| 29 | Periodic | 1 | 2 | 32 | 100 | 100 |
| 30 | Periodic | 1 | 2 | 32 | 100 | 100 |
| 31 | Periodic | 1 | 2 | 64 | 2000 | 2000 |
| 32 | Periodic | 1 | 3 | 64 | 2000 | 2000 |
| 33 | Periodic | 1 | 3 | 48 | 1000 | 1000 |
| 34 | Periodic | 1 | 3 | 16 | 1000 | 1000 |
| 35 | Periodic | 3 | 2 | 64 | 20 | 20 |
| 36 | Periodic | 1 | 3 | 64 | 2000 | 2000 |
| 37 | Periodic | 1 | 3 | 64 | 2000 | 2000 |
| 38 | Periodic | 2 | 3 | 64 | 2000 | 2000 |

Table 4-2 (Cont'd)

| | | | | | | |
|----|----------|---|---|----|------|------|
| 39 | Periodic | 2 | 3 | 64 | 2000 | 2000 |
| 40 | Periodic | 3 | 2 | 64 | 2000 | 2000 |
| 41 | Periodic | 1 | 3 | 64 | 100 | 100 |
| 65 | Sporadic | 1 | 2 | 80 | 10 | 5 |
| 66 | Sporadic | 1 | 2 | 40 | 10 | 10 |
| 67 | Sporadic | 2 | 3 | 64 | 20 | 15 |
| 68 | Sporadic | 2 | 1 | 96 | 20 | 15 |
| 69 | Sporadic | 3 | 2 | 32 | 20 | 20 |
| 70 | Sporadic | 3 | 1 | 64 | 20 | 20 |

After providing the necessary inputs we now run the program. Even though both the scheduler and the Fibex code blocks work automatically in quick succession, we can still observe the output of the scheduler from the console output. Regarding the dynamic segment, the scheduler displays the minimum number of minislots needed as 25 and also tells the respective receivers for each sporadic message. For the static segment, we are interested in the output of the `ComputeScheduleForFibex` function, which acts as the interface for the Fibex code block. A view of this output which was partly mentioned previously in Section 3.2.2 is given in Figure 4-1. In the figure, we see that the scheduler computes 2 FIDs, one with maximum repetition 2 and the other with maximum repetition 64 for Node 1. We also see that the signals sent from Node 1 are packed into 9 messages with their repetitions calculated. We also see the allocation of the first two messages with repetition 2 into FID 1, one with offset 0 and the other one with offset 1 (named base cycle in the figure). The destination nodes of each signal packed into the messages are examined to determine the set of destinations for the frame (as the union of the destinations of all signals). The size of each signal is considered to allow the calculation of bit positions of each packed signal in a frame at the next code block.

```

****
NodeItem Number: 1
FID: 1 Max Repetition: 2
FID: 2 Max Repetition: 64

FID: 1 with maxrep: 2
Message with rep: 2
Message with rep: 2
Message with rep: 8
Message with rep: 4
Message with rep: 4
Message with rep: 16
Message with rep: 64
Message with rep: 16
Message with rep: 16

Frame allocation:
Contained signals
Dest: 3
Size: 64
Dest: 3
Size: 64
Dest: 3
Size: 64
ID: 1 Base cycle: 0 Rep: 2

Frame allocation:
Contained signals
Dest: 3
Size: 64
Dest: 3
Size: 64
Dest: 3
Size: 32
ID: 1 Base cycle: 1 Rep: 2

```

Figure 4-1: A sample from the console output of `ComputeScheduleForFibex`.

By examining the remaining parts of the output given in Figure 4-1, the schedule for the static segment and the signal content of each frame can be deduced. In Table 4-3, we summarize this information in tabular form.

Table 4-3: Table depicting the static schedule as computed by `ComputeScheduleForFibex`.

| Node 1 | | | | |
|---------------|------------------------------|---------------|---|---------------------|
| FID 1 | Maximum Repetition 2 | | | |
| Frame | Repetition | Offset | Size of Contained Signals (Bits) | Destinations |
| Frame 1 | 2 | 0 | 64, 64, 64 | 3 |
| Frame 2 | 2 | 1 | 64, 64, 32 | 3 |
| FID 2 | Maximum Repetition 64 | | | |
| Frame 3 | 8 | 0 | 32 | 2 |
| Frame 4 | 4 | 1 | 64, 32, 32, 32 | 2 |
| Frame 5 | 4 | 2 | 64 | 3 |
| Frame 6 | 16 | 3 | 24 | 2 |
| Frame 7 | 64 | 4 | 48, 16 | 3 |
| Frame 8 | 16 | 7 | 64, 64, 64 | 2, 3 |
| Frame 9 | 16 | 11 | 64 | 3 |
| Node 2 | | | | |
| FID 1 | Maximum Repetition 1 | | | |

Table 4-3 (Cont'd)

| | | | | |
|---------------|------------------------------|---|------------|------|
| Frame 10 | 1 | 1 | 64 | 1 |
| FID 2 | Maximum Repetition 2 | | | |
| Frame 11 | 2 | 0 | 64, 64, 64 | 1 |
| Frame 12 | 2 | 1 | 64, 64, 64 | 1 |
| FID 3 | Maximum Repetition 4 | | | |
| Frame 13 | 2 | 0 | 64, 64, 64 | 1, 3 |
| Frame 14 | 4 | 1 | 64, 64, 64 | 1 |
| Frame 15 | 4 | 3 | 16, 64, 64 | 1, 3 |
| FID 4 | Maximum Repetition 16 | | | |
| Frame 16 | 16 | 0 | 64, 64 | 3 |
| Node 3 | | | | |
| FID 1 | Maximum Repetition 32 | | | |
| Frame 17 | 4 | 0 | 64 | 2 |
| Frame 18 | 16 | 1 | 16 | 1 |
| Frame 19 | 32 | 2 | 8 | 1 |
| Frame 20 | 16 | 3 | 64 | 2 |

Finally at the end of `ComputeScheduleForFibex`, more static and dynamic slots are added to the calculated minimum values, as described in Section 3.2. We observe the resulting parameters as seen in Figure 4-2.

```

minislots: 25
minislot dur: 3
DS: 75
staticslots: 7
staticslot dur: 46
SS: 322
SW: 1
NIT: 100
cycle: 5000
    
```

```

minislots: 1225
minislot dur: 3
DS: 3675
staticslots: 19
staticslot dur: 46
SS: 874
cycle: 5000
NIT: 399
    
```

Figure 4-2: The two sequent output displays during the extension of the FlexRay segments. The parameters and their acronyms are: Minislot number, Minislot duration (macrotick), Dynamic Segment duration (DS, in macroticks), Static slot number, Static Segment duration (SS, in macroticks), Symbol Window (SW, in macroticks, NIT (macrotick) and Cycle time (μ s).

It has to be noted that the scheduler output is only an intermediate result in order to assess the resulting schedule. It is automatically converted to the FIBEX file format by the `Fibex` code block. The FIBEX file contains all the information that is required for import into the `FlexConfig Developer` for testing. Figure 4-3 shows the resulting cluster view imported from this file. The figure shows a collection of fixed, input or schedule computed global parameters. Figure 4-4 shows the same parameter overview tool of `FlexConfig` to display the configuration of the three nodes. Since we compute the parameters according to their constraints and interdependencies, `FlexConfig` does not display any errors or warnings when we import the file.

| | Cluster1 |
|------------------------------------|----------|
| gdBit (us) | 0.100 |
| gdBitMax (us) | 0.100 |
| gdBitMin (us) | 0.100 |
| gdCASRxLowMax (Bits) | 73 |
| gdCycle (us) | 5000.000 |
| gdDynamicSlotIdlePhase (Minislots) | 1 |
| gdMacrotick (us) | 1.000 |
| gdMaxInitializationError (us) | 0.870 |
| gdMaxMicrotick (us) | 0.0250 |
| gdMaxPropagationDelay (us) | 0.930 |
| gdMinislot (MT) | 3 |
| gdMinislotActionPointOffset (MT) | 1 |
| gdMinPropagationDelay (us) | 0.000 |
| gdNIT (MT) | 399 |
| gdSampleClockPeriod (us) | 0.0125 |
| gdStaticSlot (MT) | 46 |
| gdSymbolWindow (MT) | 52 |
| gdTSSTransmitter (Bits) | 7 |
| gdWakeupSymbolRxIdle (Bits) | 59 |
| gdWakeupSymbolRxLow (Bits) | 55 |
| gdWakeupSymbolRxWindow (Bits) | 301 |
| gdWakeupSymbolTxIdle (Bits) | 180 |
| gdWakeupSymbolTxLow (Bits) | 60 |
| gListenNoise | 2 |
| gMacroPerCycle (MT) | 5000 |
| gMaxWithoutClockCorrectionFatal | 2 |
| gMaxWithoutClockCorrectionPassive | 2 |
| gNetworkManagementVectorLength | 2 |
| gNumberOfMinislots | 1225 |
| gNumberOfStaticSlots | 19 |

Figure 4-3: Cluster parameters from the imported FIBEX file, observed in the parameter overview function of FlexConfig.

| | Ecu1 (Controller1) | Ecu2 (Controller2) | Ecu3 (Controller3) |
|-------------------------------------|-------------------------------------|-------------------------------------|--------------------------|
| pAllowHaltDueToClock | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| pAllowPassiveToActive | 0 | 0 | 0 |
| pChannels | 3 | 3 | 3 |
| pClusterDriftDamping (uT) | 0 | 0 | 0 |
| pdAcceptedStartupRange (uT) | 70 | 70 | 70 |
| pDecodingCorrection (uT) | 36 | 36 | 36 |
| pDelayCompensationA (uT) | 0 | 0 | 0 |
| pDelayCompensationB (uT) | 0 | 0 | 0 |
| pdListenTimeout (uT) | 641924 | 641924 | 641924 |
| pdMaxDrift (uT) | 962 | 962 | 962 |
| pdMicroTick (us) | 0.0250 | 0.0250 | 0.0250 |
| pExternOffsetCorrection (uT) | 0 | 0 | 0 |
| pExternRateCorrection (uT) | 0 | 0 | 0 |
| pKeySlotId | 19 | 18 | 0 |
| pKeySlotUsedForStartup | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | <input type="checkbox"/> |
| pKeySlotUsedForSync | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | <input type="checkbox"/> |
| pLatestTx (Minislots) | 16 | 15 | 17 |
| pMacroInitialOffsetA (MT) | 6 | 6 | 6 |
| pMacroInitialOffsetB (MT) | 6 | 6 | 6 |
| pMicroInitialOffsetA (uT) | 4 | 4 | 4 |
| pMicroInitialOffsetB (uT) | 4 | 4 | 4 |
| pMicroPerCycle (uT) | 400000 | 400000 | 400000 |
| pMicroPerMacroNom | 40 | 40 | 40 |
| pOffsetCorrectionOut (uT) | 81 | 81 | 81 |
| pPayloadLengthDynMax (2 Byte Words) | 127 | 127 | 127 |
| pRateCorrectionOut (uT) | 962 | 962 | 962 |
| pSamplesPerMicrotick | 2 | 2 | 2 |
| pSingleSlotEnabled | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| pWakeupChannel | 0 | 0 | 0 |
| pWakeupPattern | 2 | 2 | 2 |
| vColdstartInhibit | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Figure 4-4: The parameter overview of the three FlexRay nodes used in our experiment, imported from a generated FIBEX file to FlexConfig.

We check the correctness of the schedule in the schedule view of FlexConfig. In the several screenshots depicted in Figure 4-5, we see the schedule is the same as described in Table 4-3 with the addition of two static synchronization frames and the dynamic frames. We also see the correct receivers and senders marked with Rx and Tx. The startup/synchronization frames at slots 18 and 19 have a special flag SU;SY.

| Slot | | 1 | | 2 | | 3 | | 4 | | 5 | | 6 | | 7 | | 8 | | 9 | |
|------|---------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|
| ECU | Channel | A | B | A | B | A | B | A | B | A | B | A | B | A | B | A | B | A | B |
| | | | | | | | | | | | | | | | | | | | |
| Ecu1 | | Tx | Tx | Tx | Tx | Rx | Rx | Rx | Rx | Rx | Rx | | | Rx | Rx | | | | |
| Ecu2 | | | | Rx | Rx | Tx | Tx | Tx | Tx | Tx | Tx | Tx | Tx | Rx | Rx | | | | |
| Ecu3 | | Rx | Rx | Rx | Rx | | | | | Rx | Rx | Rx | Rx | Tx | Tx | | | | |

| Slot | | 10 | | 11 | | 12 | | 13 | | 14 | | 15 | | 16 | | 17 | | 18 | | 19 | |
|------|---------|----|---|----|---|----|---|----|---|----|---|----|---|----|---|----|---|-------|----|-------|----|
| ECU | Channel | A | B | A | B | A | B | A | B | A | B | A | B | A | B | A | B | A | B | A | B |
| | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | SU;SY | | SU;SY | |
| Ecu1 | | | | | | | | | | | | | | | | | | Rx | Rx | Tx | Tx |
| Ecu2 | | | | | | | | | | | | | | | | | | Tx | Tx | Rx | Rx |
| Ecu3 | | | | | | | | | | | | | | | | | | | | | |

| Slot | | 19 | | 20 | | 21 | | 22 | | 23 | | 24 | | 25 | | 26 | | 27 | |
|------|---------|-------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|
| ECU | Channel | A | B | A | B | A | B | A | B | A | B | A | B | A | B | A | B | A | B |
| | | | | | | | | | | | | | | | | | | | |
| | | SU;SY | | | | | | | | | | | | | | | | | |
| Ecu1 | | Tx | Tx | Tx | Tx | Tx | Tx | Rx | Rx | | | | | Rx | Rx | | | | |
| Ecu2 | | Rx | Rx | Rx | Rx | Rx | Rx | Tx | Tx | Tx | Tx | | | | | Rx | Rx | | |
| Ecu3 | | | | | | | | | | Rx | Rx | Tx | Tx | Tx | Tx | | | | |

Figure 4-5: Screenshots of the schedule view of the imported FIBEX file. Static slots are marked as blue, dynamic slots as red. The ECUs send frames at the slots that they are marked with Tx where the ECUs receiving those frames are marked with Rx. The coloring of the Tx blocks represent the utilization, more fully colored blocks with less empty slots for that FID.

We also see that each frame created for the FlexRay schedule also has a PDU of the same size attached to it. In addition, we observe the signals appearing in the signal set packed into these PDUs with appropriate bit positions. Furthermore, each signal is assigned to a coding where its bitlength is defined. All in all, a complete configuration project for FlexConfig is created and the chi files generated from this project are used to operate the hardware without any problems.

4.3 Experiment with One Removed ECU

For the second experiment we remove ECU 3 and the related items in the software. We slightly alter the same signal set we used before to account for the signals whose destination node was ECU 3. We also add more signals to the system to further demonstrate the changes in the schedule. The new signal set is given in Table 4-4.

Table 4-4: The signal set used in the second experiment.

| Signal ID | Signal Type | Sending Node | Receiving Node | Size (Bits) | Repetition (μ s) | Deadline (μ s) |
|-----------|-------------|--------------|----------------|-------------|-----------------------|---------------------|
| 1 | Periodic | 2 | 1 | 64 | 10 | 10 |
| 2 | Periodic | 2 | 1 | 64 | 5 | 5 |
| 3 | Periodic | 2 | 1 | 64 | 20 | 20 |
| 4 | Periodic | 2 | 1 | 64 | 10 | 10 |
| 5 | Periodic | 1 | 2 | 64 | 10 | 10 |
| 6 | Periodic | 2 | 1 | 64 | 10 | 10 |
| 7 | Periodic | 1 | 2 | 64 | 10 | 10 |
| 8 | Periodic | 1 | 2 | 64 | 10 | 10 |
| 9 | Periodic | 2 | 1 | 64 | 10 | 10 |
| 10 | Periodic | 2 | 1 | 64 | 10 | 10 |
| 11 | Periodic | 2 | 1 | 64 | 10 | 10 |
| 12 | Periodic | 2 | 1 | 64 | 20 | 20 |
| 13 | Periodic | 2 | 1 | 64 | 10 | 10 |
| 14 | Periodic | 2 | 1 | 64 | 20 | 20 |
| 15 | Periodic | 2 | 1 | 64 | 10 | 10 |
| 16 | Periodic | 1 | 2 | 64 | 10 | 10 |
| 17 | Periodic | 1 | 2 | 64 | 10 | 10 |
| 18 | Periodic | 1 | 2 | 32 | 10 | 10 |
| 19 | Periodic | 1 | 2 | 64 | 100 | 100 |
| 20 | Periodic | 1 | 2 | 64 | 50 | 50 |
| 21 | Periodic | 2 | 1 | 16 | 100 | 100 |
| 22 | Periodic | 2 | 1 | 64 | 100 | 100 |
| 23 | Periodic | 2 | 1 | 64 | 100 | 100 |
| 26 | Periodic | 1 | 2 | 24 | 250 | 250 |
| 27 | Periodic | 2 | 1 | 64 | 10 | 10 |
| 28 | Periodic | 1 | 2 | 32 | 100 | 100 |
| 29 | Periodic | 1 | 2 | 32 | 100 | 100 |
| 30 | Periodic | 1 | 2 | 32 | 100 | 100 |
| 31 | Periodic | 1 | 2 | 64 | 2000 | 2000 |
| 32 | Periodic | 1 | 2 | 64 | 2000 | 2000 |
| 33 | Periodic | 1 | 2 | 48 | 1000 | 1000 |
| 34 | Periodic | 1 | 2 | 16 | 1000 | 1000 |
| 36 | Periodic | 1 | 2 | 64 | 2000 | 2000 |
| 37 | Periodic | 1 | 2 | 64 | 2000 | 2000 |
| 38 | Periodic | 2 | 1 | 64 | 2000 | 2000 |
| 39 | Periodic | 2 | 1 | 64 | 2000 | 2000 |
| 41 | Periodic | 1 | 2 | 64 | 100 | 100 |
| 65 | Sporadic | 1 | 2 | 80 | 10 | 5 |
| 66 | Sporadic | 1 | 2 | 40 | 10 | 10 |
| 67 | Sporadic | 2 | 1 | 64 | 20 | 15 |
| 68 | Sporadic | 2 | 1 | 96 | 20 | 15 |
| 80 | Periodic | 1 | 2 | 16 | 100 | 100 |
| 81 | Periodic | 1 | 2 | 64 | 250 | 250 |
| 82 | Periodic | 1 | 2 | 32 | 1000 | 1000 |
| 83 | Periodic | 1 | 2 | 64 | 2000 | 2000 |
| 69 | Sporadic | 1 | 2 | 32 | 20 | 20 |
| 84 | Periodic | 2 | 1 | 16 | 10 | 10 |
| 85 | Periodic | 2 | 1 | 32 | 100 | 100 |

Table 4-4 (Cont'd)

| | | | | | | |
|----|----------|---|---|----|------|------|
| 86 | Periodic | 2 | 1 | 64 | 2000 | 2000 |
| 70 | Sporadic | 2 | 1 | 64 | 10 | 10 |
| 71 | Sporadic | 2 | 1 | 32 | 15 | 15 |

Removing an ECU and introducing a different signal set results in a different configuration setting. For our case, both the dynamic and static segments have different durations than before. The changes in the schedule can be observed in Figure 4-6. As seen in the figure the dynamic segment has a smaller starting FID since the static segment is shorter. The dynamic frames are automatically configured with this new starting FID and the new dynamic frames are added. The startup frames still appear in the two static slots before the dynamic segment, this time at slots 17 and 18. This relocation as with all the other changes to the network configuration was achieved automatically without any manual interference.

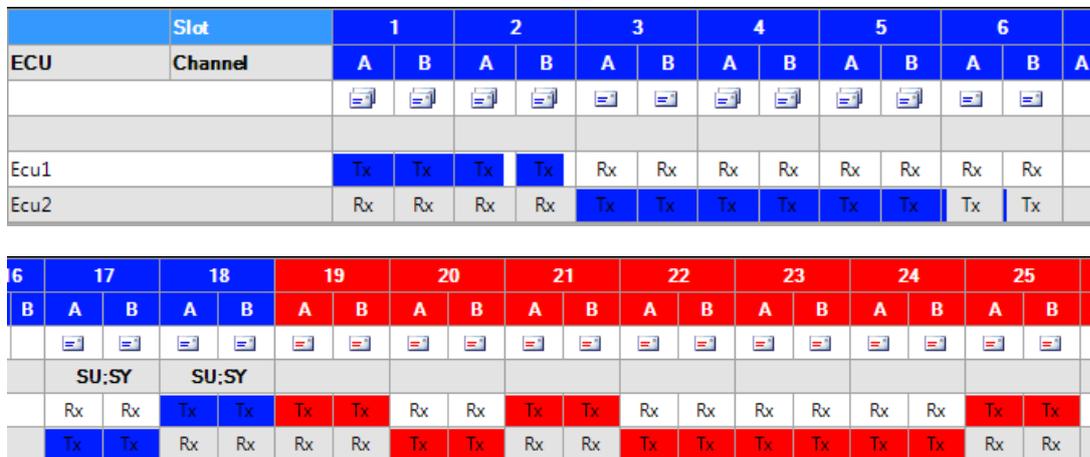


Figure 4-6: The Schedule view of the second generated FIBEX file, as seen in FlexConfig.

It has to be emphasized that the amount of manual input in a non-automated workflow would be considerably higher with respect to our workflow. Specifically, when the topology or the signal set changes, a new FlexConfig (or other software) project would need to be created manually and all data of the new schedule would have to be entered. With our automated tool chain, on the other hand, we only make the changes to the topology and the signal set and the new project is created automatically. The new configuration parameters and the schedule are configured effortlessly without any errors as well as the startup nodes needed for proper operation.

CHAPTER 5

CONCLUSION AND FUTURE WORK

The subject of this thesis is the network configuration of electronic control units (ECUs) that communicate on the FlexRay network. The design workflow of this configuration requires both the computation of all relevant configuration parameters and the generation of machine code from standardized configuration files.

In this thesis, we have developed a tool chain that automates the FlexRay network configuration process. Its input is the network topology and the signal set to be transmitted on the network. Its output is a standardized configuration file in the FIBEX format that can be imported to any commercial configuration tool in order to generate the necessary input files for the hardware implementation in the form of machine code. The required steps between input and output - signals, schedule optimization, slot allocation of message frames and computation of schedule related parameters are all done automatically. In particular, the developed tool chain improves on an existing software package for FlexRay in order to perform signal packing and message scheduling. Moreover, the resulting schedules are converted into a standardized file format (FIBEX) that can be imported into any software tool for machine code generation. The main advantage of the developed tool chain is that it automates processes such as schedule computation and entering data into existing configuration tools that were performed manually in the previous workflow. Hence, design time and error-proneness of the network configuration design workflow are effectively reduced. The developed method is successfully applied to several test cases including experiments on real hardware.

It has to be noted that the focus of this thesis is the FlexRay network. For future work CAN nodes and Gateways between CAN and FlexRay networks can also be incorporated into the developed tool-chain. Since FIBEX also supports CAN and Gateway nodes, our tool-chain can be enhanced with the ability to choose among these types of networks. For including CAN in our tool-chain, CAN scheduler of [5] can also be used in the scheduler block of our workflow.

REFERENCES

- [1] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in automotive communication systems," *Proceedings of the IEEE*, vol.93, no.6, pp. 1204-1223, June 2005.
- [2] CAN Specificaiton 2.0 <http://www.can-cia.org/>
- [3] The FlexRay Communications System Specifications, Ver. 3.1 Available: www.flexray.com
- [4] A. Demirci, "Performance Evaluation of Flexray Networks for In-Vehicle Communication," Master Thesis, Ankara, Nov. 2009.
- [5] K.W. Schmidt, Automotive Simulator C++ Api. 2007
- [6] Local Interconnect Network Specification, 2.2 Rev A Available: <http://www.lin-subbus.org/>.
- [7] Media Oriented System Transport Cooperation <http://www.mostcooperation.com>
- [8] K. Schmidt and E. G. Schmidt, "Message scheduling for the FlexRay protocol: The static segment," *Vehicular Technology, IEEE Transactions on*, vol. 58, no. 5, pp. 2160–2169, 2009.
- [9] <http://xmlsoft.org/>
- [10] EB Tresos Designer (Elektrobit). URL: <http://ww.automotive.elektrobit.com/home/ecu-software/autosar/eb-tresos-product-line/ecu-design-and-configuration-tools.html>
- [11] dSPACE FlexRay Configuration Package (dSpace). URL: <http://www.dspace.com/en/inc/home/products/sw/impsw/rtiflexray.cfm>
- [12] Network Designer FlexRay (Vector). URL: www.vector.com/vi_networkdesigner_flexray_en.html
- [13] FlexConfig RBS/Gateway (Eberspächer). URL: <http://www.eberspaecher-electronics.com/en/products/flexconfig-rbsgateway.html>
- [14] TTXPlan (TTTech). URL: <http://www.theautochannel.com/news/2006/06/01/009329.html>

- [15] ASAM e.V.: *FIBEX - Field Bus Exchange Format*
- [16] Klaus Schmidt, E. G. Schmidt, "Optimal Message Scheduling for the Static Segment of FlexRay", in 72nd IEEE Vehicular Technology conference Fall (VTC 2010-Fall), 2010.
- [17] K. Schmidt and E. G. Schmidt. Schedulability analysis and message schedule computation for the dynamic segment of FlexRay. In Vehicular Technology Conference, 2010.
- [18] AUTOSAR Release 4.0, <http://www.autosar.org/>
- [19] Association of Standardization of Automation and Measuring Systems, <http://www.asam.net/>
- [20] Extensible Markup Language (XML) 1.0 (Fifth Edition), <http://www.w3.org/TR/xml/>
- [21] G. Leen, D. Heffernan, "Expanding automotive electronic systems," IEE Proc.-Comput. Digit. Tech, vol.35, No.1, pp. 88-93, January 2002.