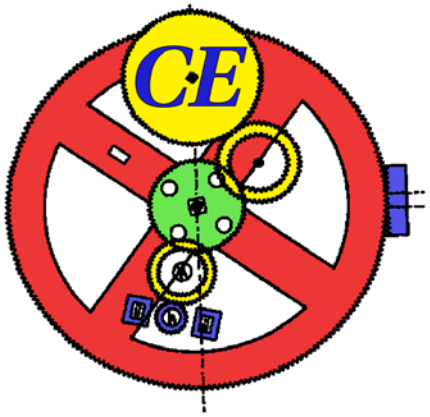

**Computer Engineering**
Mekelweg 4,
2628 CD Delft
The Netherlands
**http://ce.et.tudelft.nl/**

2011

# MSc THESIS

# Reliable In-Vehicle FlexRay Network Scheduler Design

**Aijie Zhao**

**Abstract**

The rapid developments of the automobile followed with the in-vehicle applications increase in both number and complexity. Those automotive technologies bring substantial demands of in-vehicle data communication. The need for data communication is growing beyond the capability of the existing automotive network. FlexRay protocol emerges to adapt the needs of the growing capacity. It is the next generation automotive communication protocol that offers fast data rate, reliable and fault-tolerant transmission. The network that uses FlexRay protocol is called FlexRay network. There are two types of FlexRay network. Simple FlexRay network is the one uses buses or active stars to connect the Electronic Control Unit (ECU). Switched FlexRay network is the one uses network switch to replace the active star in simple FlexRay network. In this thesis, we first introduce FlexRay protocol specifications and the real-time scheduling theory as the background information. Then we discuss the scheduling problems and configurations in static (ST) and dynamic (DYN) segments in communication cycle (CC). This thesis presents the scheduling algorithms for the ST and DYN segment in simple and switch FlexRay network respectively. The experimental results obtained from computer simulations show the schedulability of the switch ST scheduler is far better than the simple ST scheduler. Also, the results show the worst-case response times of the DYN messages in switch FlexRay network are shorter than the ones in simple FlexRay network.

**CE-MS-2011-05**

TU Delft Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

# Reliable In-Vehicle FlexRay Network Scheduler Design

THESIS

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

ELECTRICAL ENGINEERING

by

Aijie Zhao
born in Sichuan, China

Telecommunications
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

# Reliable In-Vehicle FlexRay Network Scheduler Design

By Aijie Zhao

**Abstract**

The rapid developments of the automobile followed with the in-vehicle applications increase in both number and complexity. Those automotive technologies bring substantial demands of in-vehicle data communication. The need for data communication is growing beyond the capability of the existing automotive network. FlexRay protocol emerges to adapt the needs of the growing capacity. It is the next generation automotive communication protocol that offers fast data rate, reliable and fault-tolerant transmission. The network that uses FlexRay protocol is called FlexRay network. There are two types of FlexRay network. Simple FlexRay network is the one uses buses or active stars to connect the Electronic Control Unit (ECU). Switched FlexRay network is the one uses network switch to replace the active star in simple FlexRay network. In this thesis, we first introduce FlexRay protocol specifications and the real-time scheduling theory as the background information. Then we discuss the scheduling problems and configurations in static (ST) and dynamic (DYN) segments in communication cycle (CC). This thesis presents the scheduling algorithms for the ST and DYN segment in simple and switch FlexRay network respectively. The experimental results obtained from computer simulations show the schedulability of the switch ST scheduler is far better than the simple ST scheduler. Also, the results show the worst-case response times of the DYN messages in switch FlexRay network are shorter than the ones in simple FlexRay network.

**Laboratory** : Computer Engineering

**Code number** : CE-MS-2011-05

**Committee Members**

**Advisor** : Dr. Ir. Zaid Al-Ars, CE, TU Delft

**Advisor** : Dr. Lotfi Mhamdi, CE, TU Delft

**Chairperson** : Dr. Koen Bertels, CE, TU Delft

**Member** : Dr. Ertan Onur, WMC, TU Delft

*Dedicated to my beloved family*

# Contents

# Abbreviations

| | |
|---|---|
| **ABS** | Anti-Lock Braking |
| **ACC** | Adaptive Cruise Control |
| **ACU** | Airbag Control Unit |
| **AD** | Airbag Deployment |
| **AS** | Active Suspensions |
| **BD** | Bus Driver |
| **BDM** | Body Control Module |
| **BG** | Bus Guardian |
| **CC** | Communication Cycle |
| **CCS** | Chassis Control System |
| **CRC** | Cyclic Redundancy Check |
| **DEM** | Differential Electronic Module |
| **DYN** | Dynamic |
| **ECM** | Engine Control Module |
| **ECS** | Engine Control System |
| **ECU** | Electronic Control Unit |
| **EPS** | Electric Power Steering |

**ESC** Electronic Stability Control

**FTDMA** Flexible Time Division Multiple Access

**NIT** Network Idle Time

**NMV** Network Management Vector

**MT** Macrotick

**OBD** On-Board Diagnostics

**ST** Static

**SYM** Symbol Window

**TCS** Transmission Control system

**TDMA** Time Division Multiple Access

**TPM** Tire Pressure Monitoring

# List of Figures

# List of Tables

# Acknowledgement

Doing something in a new field is really a challenge for me, but also I see it as an opportunity to learn new things and to put my analyzing ability into practical use. I really appreciate the experience and knowledge on automotive network I gained from doing this thesis. This thesis would not be possible to create without the help from my supervisors Lotfi Mhamdi and Zaid Al-Ars. I learnt from them not only the methodology to solve the problems that you face when doing academic research but also the way to manage the progress of a project and the motivation of keeping learning. Therefore I want to thank them for their advices, comments and reviewing. I would also like to thank Diomidis Katzourakis for the advices on automotive networks. Thanks to Shuofei Yang for the helpful advices on programming. Thanks to Hugo Poley for the careful proof reading. Thanks to Gebei He for always supporting me. Thanks to my beloved parents for supporting my study aboard and always giving me courage and bravery to face difficulties. Finally, thanks to all give me the memory of my life in Delft.

Aijie Zhao

Delft, the Netherlands

April 2011

# 1

# Introduction

The modern car is a combination platform of the safety, easy-to-drive and the entertainment. Consumers demand more and more automotive automation, driver assistance and safety from the vehicle. Car manufacturers developed many complex systems and technologies to satisfy these demands. Over the past forty years, electronic systems in vehicles have an exponential increase in number and complexity. The analysis indicates that more than 80 percent of the automotive innovation now stems from electronics [1]. The development trend of in-vehicle control systems changes from mechanical gradually to electronic. The manufacturers have concentrated on developing electronic systems that safely and efficiently replace in-vehicle mechanical and hydraulic applications. Figure 1-1 shows an example of the electronic systems and applications in a modern vehicle.

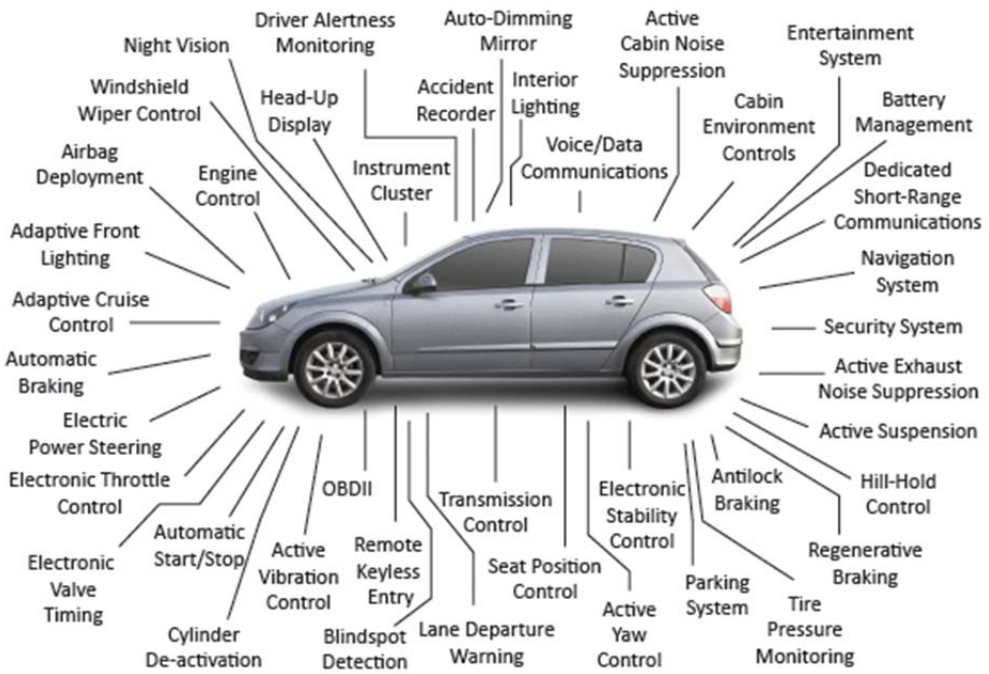


**Figure 1-1 Example of Automotive Electronic Systems [2]**

The main purposes of the in-vehicle electronic systems are to assist the driver to control the vehicle, to avoid some potentially dangerous operations as well as to increase the system efficiency and stability. There are lots of existing driver assistance systems such as Electric Power Steering (EPS), Active Suspensions (AS), Electronic Stability Control (ESC), and Adaptive Cruise Control (ACC). These systems offer the driver easy and accurate experience of control. There are also systems that give the greatest degree of safety protection to people such as Antilock Braking System (ABS), Airbag Deployment (AD), and Tire Pressure Monitoring (TPM). Moreover, there are systems that control core functional devices such as Engine Control System (ECS), Transmission Control system (TCS), and Chassis Control System (CCS). These systems have been widely implemented already. Other systems like Entertainment Systems: multimedia and internet access, Communication and Navigation Systems, Seat Position Control, and Cabin Environment Controls offer comfort and convenience to people. These systems are implemented in some models either.

The rest of the chapter is organized as follows: Section 1.1 provides a brief description of the problem being addressed in this thesis. Section 1.2 presents the motivation behind the thesis. The goals of this thesis are provided in Section 1.3. The last Section 1.4, states the overview of the thesis.

## 1.1 Problem Statement

The in-vehicle electronic systems depend on the successful exchange of a massive amount of signals and interconnected wires between electronic control units (ECU). Those ECUs are core equipment in the electronic control system. The exchange of information motivates the use of in-vehicle control networks.

Control networks in the vehicle facilitate the information and resources sharing among the distributed ECUs. Connections between vehicles' electronic elements in the control networks usually are wiring in the past. In today's luxury cars, up to 2500 signals are exchanged for up to 70 ECUs [3]. The wiring needs to be increased to support the exchange of this enormous amount of signal. However, added wiring increases vehicle weight which increases fuel consumption and complex wiring harnesses take up large amounts of vehicle space which limit the expanding of functionality. Beginning in the early 1980s, centralized and then distributed networks have replaced point-to-point wiring [4]. Today's control and communications networks base on serial protocols. So it reduces the total wire length used to interconnect ECUs. Moreover it counters the problem of large amounts of discrete wiring.

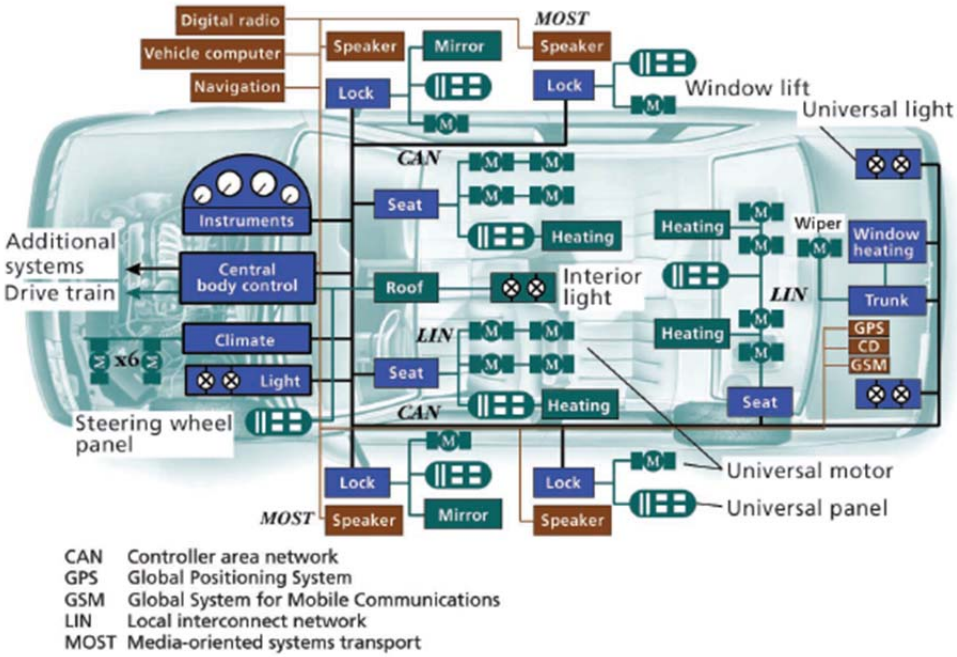


**Figure 1-2 One subset of a modern vehicle's networks [1]**

One of the main objectives of the design step of an in-vehicle embedded system is to ensure a proper execution of the vehicle functions, with a predefined level of safety, in the normal functioning mode but also when some components fail [5] (e.g., reboot of an ECU) or when the environment of the vehicle creates perturbations. Networks play a central role in maintaining the electronic control systems working properly, because most of the core elements are distributed and need to communicate with each other.

As a result, there is a need for designing different automotive networks capable of meeting different applications' requirements. Such as Local Interconnected Network (LIN) [6] designs for transmitting simple control data with data rates lower than 10kb/s. The Low-speed Controller Area Network (CAN) [7] designs for data sharing and exchanges between sensors and ECUs. The High-speed CAN [8] designs for high speed real-time communications. The Media Oriented System Transport (MOST) network [9] designs for the multimedia data which has high data rates.

New applications like x-by-wire need new features from the control networks such as predictability, fault tolerance and flexibility. These motivate the development of new automotive control networks, for example, Time Triggered Protocol (TTP) [10] and FlexRay [11] . Furthermore, in order to adapt different in-vehicle applications, the control networks should be able to support both time-triggered (TT) and event-triggered (ET) transmission. This type of networks, for example, are TTCAN [12] and FlexRay.

## 1.2 Motivation

The vast increase in automotive electronic systems has created new engineering opportunities and challenges. The resulting demands on design have led to innovations in

electronic networks for automobiles. To be capable of meeting current and next generation automotive systems requirements, one of the promising technologies envisioned in this project is the FlexRay. FlexRay is a time-triggered communication technology that provides high speed fault tolerant communications by combining time-triggered TDMA and the event-triggered FTDMA. Traffic passing through the FlexRay network is scheduled either statically with bounded communication latency (e.g. TDMA) or dynamically (e.g., FTDMA) [5]. It is the next generation in-vehicle network which is also the future replacement for CAN in many vehicle network architectures. FlexRay has already been implemented in some vehicle models such as the BMW X5, BMW 7-Series and AUDI A8, etc.[13].

However, even the FlexRay network is one of the most flexible and optimal automotive networks up to now, it still cannot guarantee to meet in-vehicle real-time constraints under system high loads, such as required response times. As a result, a new concept called FlexRay switch emerges to address these problems.

## 1.3 Project Goal

The goal of this thesis is to design two schedulers for the FlexRay network, one for simple FlexRay network and another for switch FlexRay network. For a given automotive network, including the ECUs, the communication patterns and the real-time communication constraints, design the schedulers to ensure a predictable and safe performance of the automotive network. The schedulers consist of scheduling both the TDMA communication (ST segment) and the FTDMA communication (DYN segment).

## 1.4 Thesis Overview

All the contents presented in this thesis will go into details in the following chapters:

Chapter 2 presents the necessary background information of this thesis, the FlexRay network components, network topology and the FlexRay protocol. Especially emphasizes on the FlexRay protocol's critical timing unit – the Communication Cycle. In this chapter, we can get a general idea about what the FlexRay is. Chapter 3 discusses the real-time system and different scheduling algorithms. Chapter 4 investigates the timeline of the FlexRay transmission and presents the method to evaluate schedulability of ST segment and DYN segment in simple FlexRay networks. It divides one knotty problem into many small problems to get a solution. Chapter 5 introduces the bus optimization and configuration for the ST segment and the DYN segment respectively. Moreover, this chapter gives the scheduling algorithms for each segment. Chapter 6 introduces the concept of the switch FlexRay network. This chapter presents the origination of the switch idea followed by the switching principle in the ST segment and DYN segment respectively. Chapter 7 provides the scheduling algorithms for the ST and DYN segment in switched networks and the schedulability analysis for the DYN segment. Chapter 8 concludes this thesis and recommends the future developments of this topic.

# 2

# Background

The FlexRay Protocol is a new communication protocol for automotive networks. FlexRay protocol is originated from the successful experience of BMW ByteFlight protocol. It is developed by the FlexRay consortium which consists of car manufacturers like BMW, DaimlerChrysler, General Motors, Ford and Volkswagen. The FlexRay consortium also cooperates with semiconductor companies like Bosch, Freescale, and Philips. The aim of the FlexRay protocol is to create a faster and more reliable automotive network system that would suit current and future needs.

In this chapter, the key components and their functions in the FlexRay node are introduced and different possible topologies for the FlexRay network are explained. The final part presents the FlexRay protocol based on different segments in the Communication Cycle (CC).

## 2.1 FlexRay Network

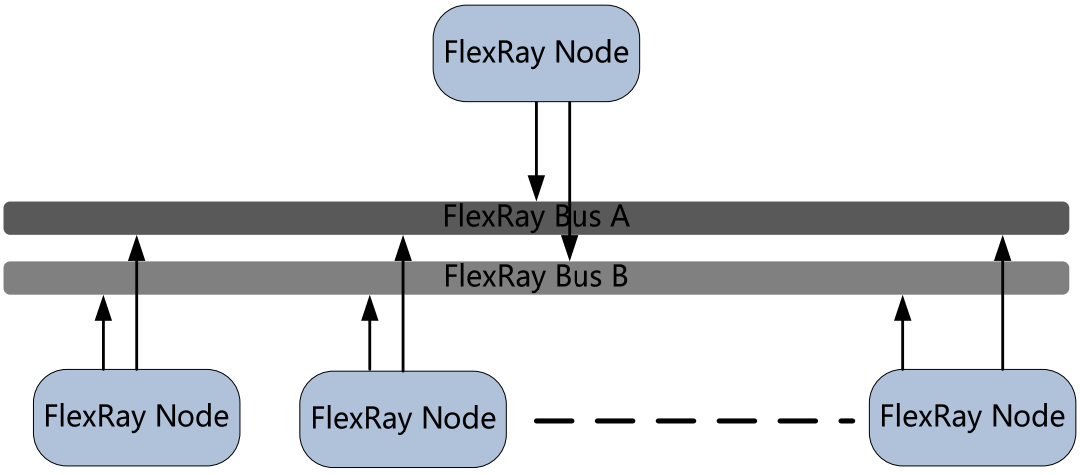


**Figure 2-1 FlexRay node cluster connected by FlexRay bus**

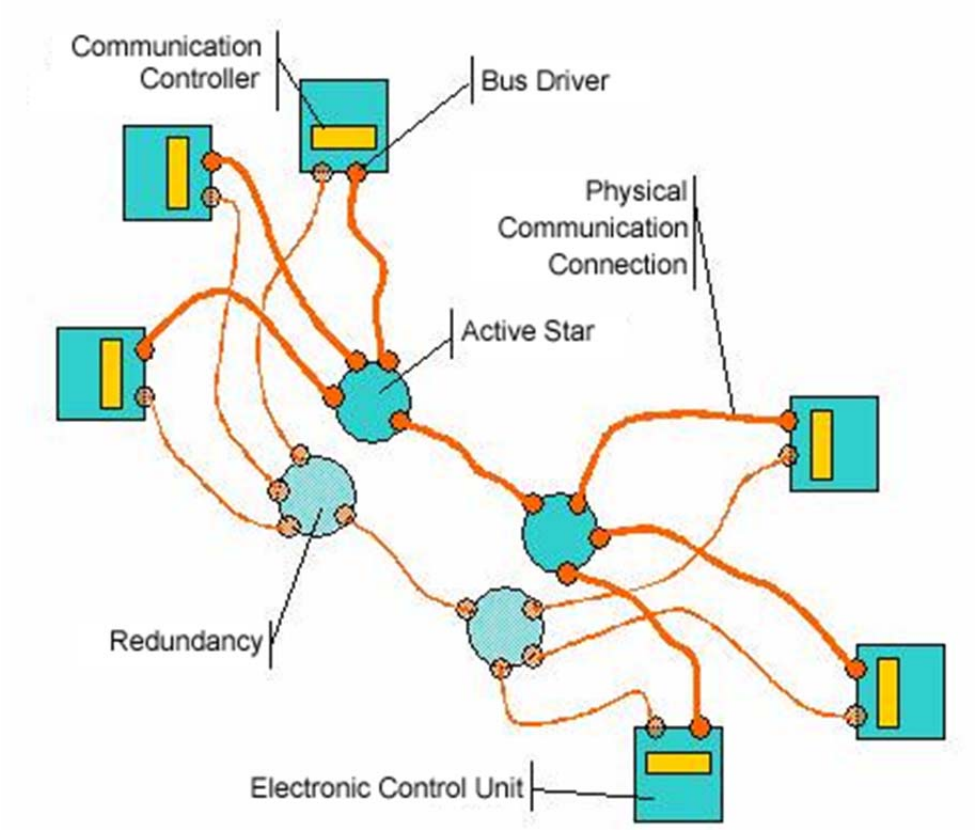


**Figure 2-2 ECUs nodes cluster connected by FlexRay active star [14]**

Figure 2-1 and Figure 2-2 illustrates two most common FlexRay networks that consist of ECUs clusters. The cluster is a distributed system where nodes are connected via at least one communication channel directly [15], like a bus or an active star.

Figure 2-1 shows a bus network and Figure 2-2 shows an active star network. These two figures show that FlexRay nodes and physical connections compose the FlexRay network. Section 2.1.1 will introduce the FlexRay node structure and working mechanism. Section 2.1.2 will comprehensively introduce the probable FlexRay topologies according to FlexRay 2005 specifications.

As previously introduced, the FlexRay protocol are designed especially for the communication of automotive networks applications. Figure 2-2 shows a typical FlexRay network with the redundant capacity. A central control component - active star [11] connects ECUs. In this figure, it can be seen that the FlexRay protocol controls the data transmission between different Electronic Control Unit (ECU). The dual-channel feature of the FlexRay protocol can be used as a redundancy channel in case of system failure. We can see these ECUs as different nodes in FlexRay network. The definition of the ECU and the concept of the node explain in Section 2.1.1. Section 2.1.2.2 provides the definition of the active star.

### 2.1.1 FlexRay Node

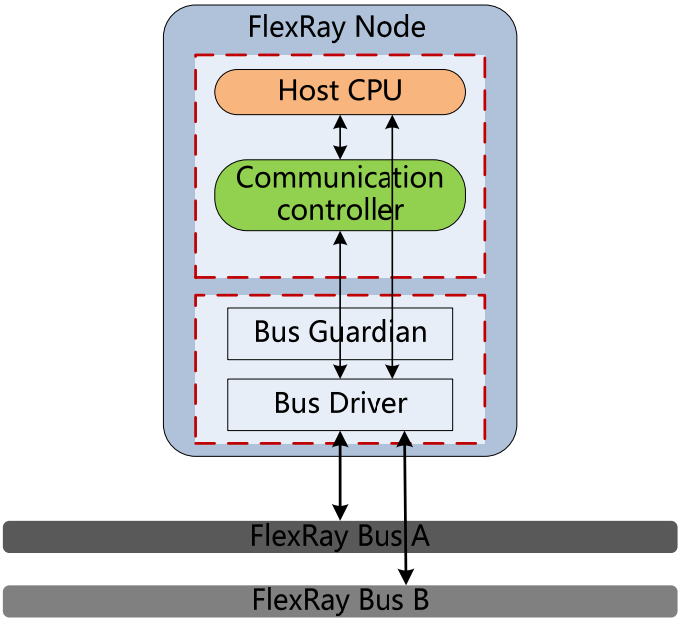


**Figure 2-3 Structure of a FlexRay network node**

As introduced in Section 2.1, the ECUs are the essential components in a FlexRay network. So what is ECU? ECU is a generic term for any embedded system that controls one or more of the electrical systems or subsystems [16] in an automotive vehicle. Different ECUs have different functions. The main ECUs in a vehicle include Engine Control Module (ECM), Anti-Lock Braking (ABS), Differential Electronic Module (DEM), On-Board Diagnostics (OBD), Airbag Control Unit (ACU), Body Control Module (BDM), etc. These different ECUs consist of the core control center of a vehicle. According to the FlexRay specifications, the node shall provide at least one absolute timer that may be set to an absolute time, in terms of cycle count and Macrotick. The following paragraphs will explain what components inside an ECU are essential for the transmission.

Figure 2-3 shows a generic structure of a FlexRay node. We can see from Figure 2-3, the node generally has two parts, the controller part and the driver part. The controller consists of a Communication Controller (CC) and a Host CPU. The driver part consists of a Bus Driver (BD), optional to have a Bus Guardian (BG).

**Host CPU**

The Host CPU is a part of an ECU where the application software is executed. The Host CPU provides the control and configuration information to the CC, also provides payload data transmitted during the CC.

**Communication Controller (CC)**

The Communication controller is an electronic component in a node that is responsible for implementing the protocol aspects of the FlexRay communications system. It provides

status information to the host and delivers payload data received from communication frames [11].

**Bus Driver (BD)**

Bus driver is an electronic component consisting of a transmitter and a receiver that connects a communication controller to one communication channel [11]. It can be used in electrical encoding/decoding, remote wake-up and error detection on OSI layer 0 (voltage, temperature).

**Bus Guardian (optional)**

Bus Guardian protects a channel from interference caused by communication that is not temporally be scheduled within limited of the times in a schedule [11]. Namely, it restricts transmissions of CC to defined slots, fault containment for fail-safe node communication, supports error detection and fault tolerance.

The main processes of a node accesses to the bus are as following: BD first connects the CC and the bus. The BG monitors the connection which accesses to the bus. The Host CPU informs the BG which time slots are allocated by the CC. The BG only allows CC to transfer data at these time slots. It also activates the BD. If the BG detects an idle interval in time, then it disconnects the node with the communication channel.

## 2.1.2 Network Topology

The FlexRay protocol supports a variety of topologies while providing a flexible configuration. This section presents some commonly used topologies in simple FlexRay network and explains the differences between these topologies.

### 2.1.2.1 Linear Passive bus

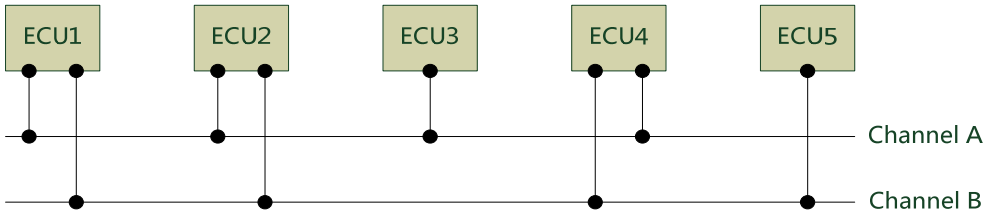


**Figure 2-4 Passive bus topology**

Like the network shows in Figure 2-4, nodes in FlexRay can be connected to either both channels or only one of them. The figure shows a possible configuration of the network as a dual bus system. Nodes that connected by the same bus can only transmit data once at a time. Similarly, the FlexRay network could be a single-bus system. In this case, all the nodes should connect to the bus.

#### 2.1.2.2 Star topology

According to the FlexRay specifications, the star is a device that allows information to be transferred from one physical communication link to one or more other physical communication links. It duplicates information present on one of its links to the other links connected to the star. [11] There are passive and active star in FlexRay protocol.

**Passive Star**

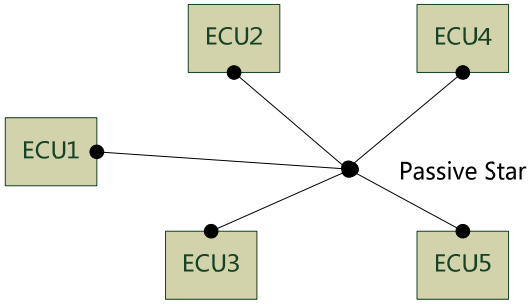


**Figure 2-5 Passive star topology**

If there are more than two ECUs need connect with each other, it is good to use a passive star structure like Figure 2-5 demonstrated. The passive-star structure is a special case of the linear passive bus. In a passive star structure, all ECUs are connected to a single splice [17].

**Active Star**

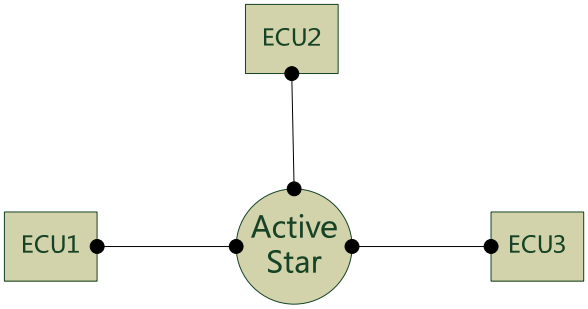


**Figure 2-6 Simple active star topology**

As Figure 2-6 shows, this network uses point-to-point connections between the active star and ECUs. The active star has the function to transfer data from one branch to all other branches, like a Hub in the Ethernet network. Since it has the transmitter and receiver circuit for each branch, the branches are electrically decoupled from each other [17]. The minimum number of branches at an active star is 2, no maximum according to the specifications.

**Cascaded Active Star**

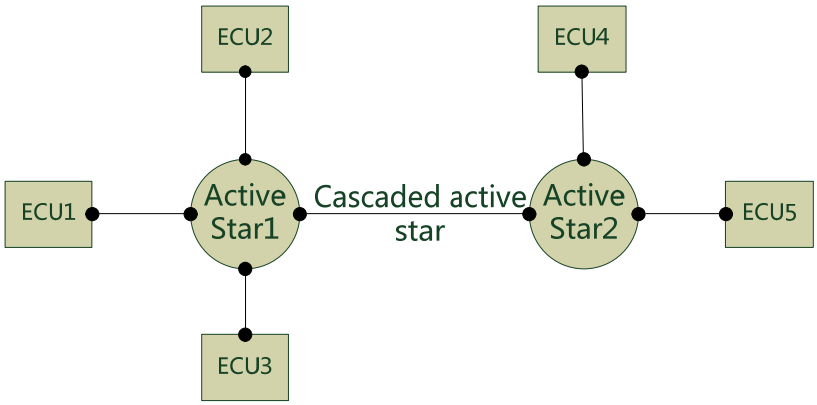


**Figure 2-7 Cascaded active star topology**

There are a few possible ways to configure the active-star topology because the FlexRay network can have multiple stars. Figure 2-7 provides one sample of the topology configuration. This topology is called cascaded active star. There are some constraints for this topology. Firstly, it cannot have a closed ring. Secondly, it cannot have more than 2 stars in one channel. The cascaded active star topology supports redundant channel either. The star actively sends the incoming signal to all nodes.

### 2.1.2.3 The hybrid topology

Besides the two topologies mentioned above, the FlexRay network also has the third topology, which is the combination of those two. The hybrid topology needs to follow the constraints of every elementary topology. There are many different ways to combine them. Figure 2-8 and Figure 2-9 provide the most representative two examples.

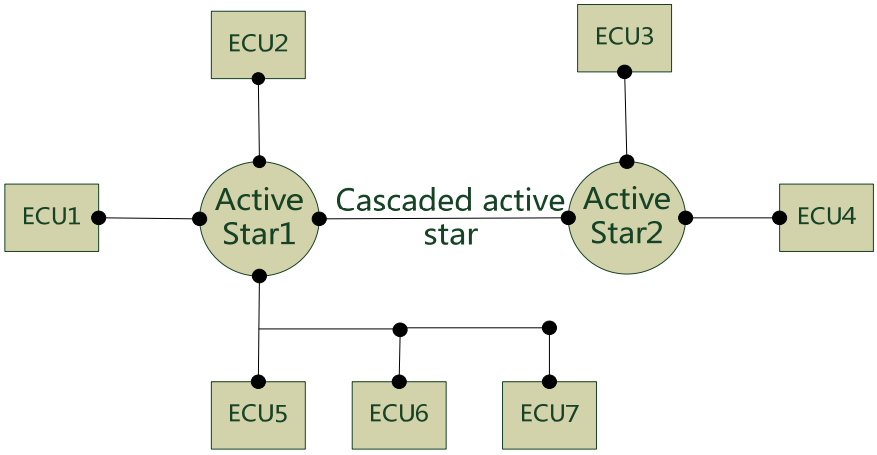


**Figure 2-8 Single channel hybrid topology**

In Figure 2-8, ECU1, 2, 3 and 4, use point-to-point connections connect to the active stars. The other ECUs connect with each other use a bus. The bus connects to the active star 1, in order to enable the communication between ECU5, 6 and 7.

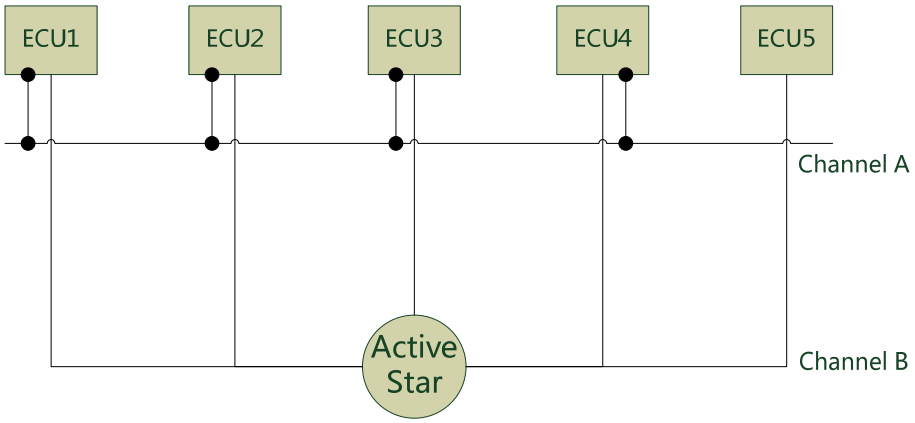


**Figure 2-9 Dual-channel hybrid example**

Figure 2-9 shows another sample of the hybrid topology. In this topology, each ECU is connecting to two channels A and B but use different ways. Channel A is a passive bus topology while channel B is an active star topology.

## 2.2 FlexRay Protocol

The FlexRay protocol is an in-vehicle communication protocol especially for the fast speed and high reliability data transmission. Compared to other in-vehicle communication protocols, such as CAN and TTP, the FlexRay protocol has significant improvements in a variety of aspects. Table 2-1 shows the general overview of the FlexRay protocol.

| | |
|---|---|
| Transmission channels | 1 or 2 |
| Gross data rate | 10Mbit/s |
| Effective data rate/channel | 5Mbit/s |
| Max payload per frame | 254 bytes |
| Max data rate effectively | ca.5000 Kbit/s |
| Transmission duration/frame | ca. 60 µs (40 bytes @ 10Mbit/s) |
| Buffer memory | Typ. 8kBytes |
| Transmission cable | Twisted pair cable |
| Length of the cable | Max. 24m |

**Table 2-1 FlexRay protocol overview**

The FlexRay protocol offers the possibility to serve up to two channels. The two-channel capacities increase the system bandwidth as well as introduce a redundant channel to increase the fault tolerance level. The maximum data rate of the channel is 10Mbps. The total data rate in FlexRay network maximum can be 20Mbits/s. In the psychical layer, the FlexRay protocol uses twisted pair for sending and receiving. The maximum length of the cable is 24m.

The introduction of the Communication Cycle (CC) is to adapt different communication, such as time-triggered communication and event-triggered communication. The transmission in FlexRay network consists of many CCs. Each CC is divided mainly into a ST segment and a DYN segment. The ST communication provides bounded delay, while the DYN communication provides flexible transmission. The ST segment uses the static time-trigger scheme, namely Time Division Multiple Access (TDMA), to transmit data, while the DYN segment uses the flexible time-triggered scheme, namely the DYN Minislot based scheme to transmit data. The detail about the CC and Minislots will be introduced in Section 2.2.3.

### 2.2.1 Physical Frame Format

The data needed to be packed into frames before sending to the physical channel. This section introduces the FlexRay frame and the functions of different parts in a frame.

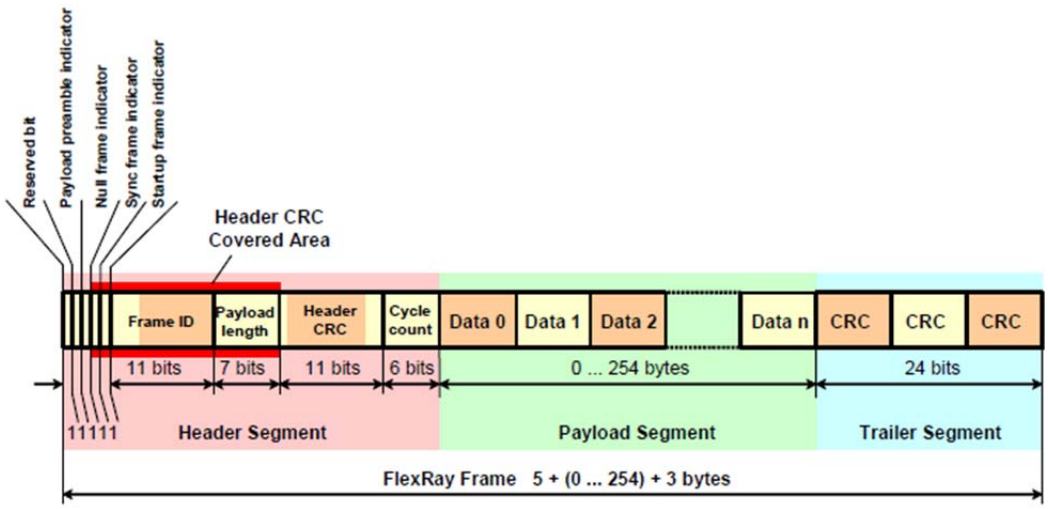


**Figure 2-10 FlexRay Frame Format [11]**

Figure 2-10 shows the FlexRay frame format. It is the only frame in FlexRay protocol. As we can see in Figure 2-10, the FlexRay frame consists of three parts, the header segment, the payload segment and the trailer segment. In the following paragraphs, we will introduce these three parts respectively.

#### 2.2.1.1 FlexRay header segment

The FlexRay header segment is 5 bytes long. It consists of 9 parts. Each part has the different function.

**Reserved bit (1 bit)**

This 1 bit is reserved for future protocol use.

**Payload preamble indicator (1 bit)**

This 1 bit indicates whether or not an optional vector is contained within the payload [11]; “1” means contained, “0” means not.

If the frame is transmitted in the ST segment, this position indicates the presence of a *network management vector* at the beginning of the payload [11]. If the frame is transmitted in the DYN segment, this position indicates the presence of a *frame ID* at the beginning of the payload.

**Null frame indicator (1 bit)**

This 1 bit indicates whether or not the frame is an empty frame. ”0” means the payload segment contains no valid data; “1” means the payload segment contains valid data.

**Sync frame indicator (1 bit)**

This 1 bit indicates whether or not the frame is a sync frame. “0” means no synchronization for node; “1” means all receiving nodes shall use the frame for synchronization if it meets synchronization conditions. This will discuss in detail later.

**Startup frame indicator (1 bit)**

This 1 bit indicates whether or not a frame is a *startup frame*. Only cold start nodes[1] are allow to transmit startup frame. “0” means this frame is not a startup frame; “1” means this frame is a startup frame. This part shall set to “1” in the sync frames of cold start nodes. A cold-start node can only transmit one frame per CC with startup frame indicator set to “1”.

**Frame ID (11 bits)**

This position defines the slot in which the frame should be transmitted. Each slot has a slot number. If the slot number equals to frame ID, this slot can use for transmission of this frame. A frame ID is unique on each channel in one CC. The frame ID ranges from 1 to 2047. 0 is invalid.

---

[1] Cold-start node: a node capable of initiating the communication startup procedure on the cluster by sending startup frames.

**Payload length (7 bits)**

This part is used to indicate the size of the payload segment. The value of the payload length position is set to the number of payload bytes divided by 2. Its range is from 0 to 254 bytes.

**Header CRC (11 bits)**

This part contains a cyclic redundancy check code (CRC) that is computed over the sync frame indicator, the startup frame indicator, the frame ID, and the payload length [11]. The header CRC of transmitted frames is computed offline and provided to the Communication Controller (CC) by means of configuration. It is not computed by transmitting CC. The CC shall calculate the received frame's header CRC in order to check that the CRC is correct [11].

**Cycle count (6 bits)**

This part indicates the value of cycle counter, from the transmitting node's view, at the time of frame transmission happened.

#### 2.2.1.2 FlexRay payload segment

The FlexRay payload segment contains 0 to 254 bytes data (0 to 127 two-byte words). It is import to notice that the payload segment contains only even number of bytes because the unit used in this segment is two-byte.

**Network Management Vector**

A number of data in the payload segment that transmits in the ST slot can be used as network management vector (NMVector). This vector is optional.

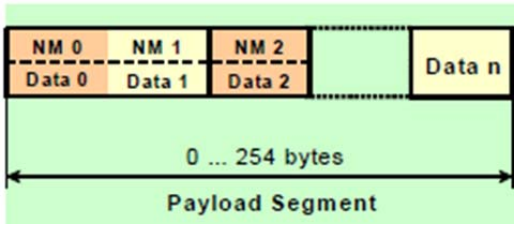


**Figure 2-11 NM vector in payload segment [11]**

If the payload segment uses as NMVector, the format of payload segment is like Figure 2-12shows. NMVector is written by the Host CPU in the transmission node as application data. The length of the NMVector is configurable. All nodes in a cluster must be configured with the same value for this parameter.

**Frame ID**

The first two bytes of the payload segment of the FlexRay frame transmitted in the dynamic segment can be used as frame ID. It is also optional.

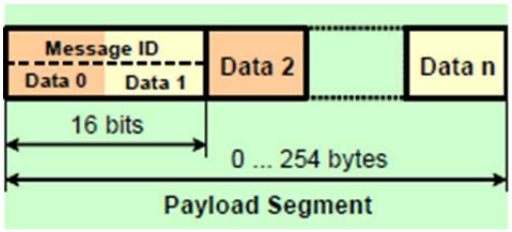


**Figure 2-12 Frame ID in payload segment [11]**

If the payload segment uses as frame ID, the format of payload segment is like Figure 2-12 shows. The frame ID is an application determined number that identifies the contents of the data segment. It is 16 bits long. Frame ID is written by the transmission node's Host CPU as application data. The CC has no knowledge about the frame ID.

#### 2.2.1.3 FlexRay trailer segment

The FlexRay trailer segment is a 24-bit cyclic redundancy check code (CRC) for the frame. It is computed with the data in the header segment and the payload segment of the frame.

### 2.2.2 Media Access

There are two different ways that use in different segments in CC to trigger the data transmission. In the ST segment, the FlexRay protocol uses Time Division Multiple Access (TDMA) as the media access mechanism. In the DYN segment, the FlexRay protocol uses Flexible Time Division Multiple Access (FTDMA) as the media access mechanism. These two different ways are the fundamental principles in media access control. Before we introduce those two media-access mechanisms, we will present two basic trigger modes of the transmission, the time-trigger and event-trigger.

#### 2.2.2.1 Time-trigger System and Event-trigger System

**Time-trigger system**

A real-time system is time-triggered if a schedule and a clock determine the transmissions performed by this system. External events, like interrupts, do not significantly influence the system operation.

The time-triggered communication is a synchronous transmission that controlled by distributed fault tolerant clocks. The transmissions are performed according to a predefined schedule executed on a global time-base. The global time-base can be established on-line by using the global clock synchronization. The scheduler is an off-line scheduler that is not run time determined by application behavior.

The time-triggered system can ensure the data transmission to follow the predetermined time. The transmissions are determined in advance. They are not flexible but very predictable. This system is suitable for the data that have high-reliability requirements. The time-triggered

transmission needs to determine a reasonably coordinated schedule before the transmission start.

**Event-trigger system**

In the event-triggered system, transmissions happen when a significant change of state occurs. In FlexRay, the Minislot based arbitration controls the event-triggered transmissions. The transmission time depends on network load in the DYN segment. The event-triggered transmission is very flexible, but not very predictable in case of system under peak load. The FlexRay protocol uses both the time-trigger method and the event-trigger method.

#### 2.2.2.2 TDMA in ST segment

The ST segment uses static time-trigger, namely Time Division Multiple Access (TDMA), as the media access control. The fundamental principle is the time-trigger method presented in Section 2.2.2.1. TDMA is a channel access method for shared medium networks. It allows several users to share the same frequency channel by dividing the signal into different time slots [18]. It assigns the network capacity to the nodes in a static and permanent way. Frames are sent at predetermined instances called slots. A schedule of slots is created offline. The schedules can also be created online. Once the schedule has been determined, it is then followed and repeated online.

Because TDMA is very deterministic and predictable, it is suitable for safety-critical tasks with hard real-time requirements. However, the inflexibility is one of the drawbacks of TDMA. TDMA frames cannot be sent at the arbitrary time. Furthermore, TDMA wastes the bandwidth. If the transmission of a frame only needs half of the predefined slot, other frames cannot use the other half of the slot.

#### 2.2.2.3 FTDMA in DYN segment

The DYN segment uses flexible time-trigger, namely Flexible Time Division Multiple Access (FTDMA), as the media access control. The fundamental principle is the event-trigger task transmission with dynamic Minislot based arbitration presented in Section 2.2.2.1. Once a task is triggered by a significant change of state, it needs to use FTDMA to arbitrate the media and be transmitted.

FTDMA enables frames have chances to be sent whenever they require. There are no static slot allocations in advance. The media access is priority based. FTDMA is similar to TDMA except the slot size. The slot size in FTDMA is not fixed. It will vary depending on whether the slot is used or not. If a slot is not used within a small time offset, which is a Minislot in FlexRay protocol, the scheduler will progress to the next slot. This is called Minislot based arbitration. The slot size depends on the frame length transmitted in that slot.

### 2.2.3 Timing Hierarchies of CC

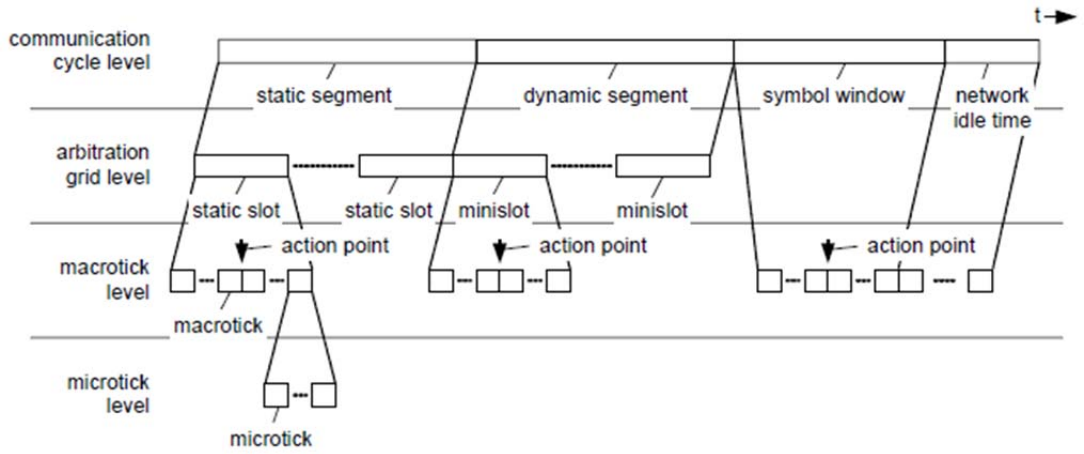


**Figure 2-13 Timing Hierarchies of CC [11]**

The definition of CC in FlexRay protocol is that one complete instance of the communication structure that is periodically repeated to comprise the media access method of the FlexRay system [11]. As we can see from Figure 2-13, CC can be further divided into 4 levels, CC level, arbitration grid level, Macrotick level and Microtick level. The lengths of the basic units in different levels are configurable. According to FlexRay specifications, the header segment of the frame has 6 bits that indicate the value of cycle counter with the notation *vCycleCounter* . The maximum value of the cycle counter is $2^6$ =64. We define the period of 64 cycles as the global static-schedule ($T_{ss}$).

**Communication cycle level**

Communication cycle level defines the basic time unit in the FlexRay protocol, CC. It is the highest level in the timing hierarchy. As we can see from Figure 2-13, CC consists of ST segment, DYN segment (optional), symbol window (optional) and network idle time (NIT). The length of the CC, expressed in μs, is from 10 μs to 16000 μs. The typical length is 5ms. Each CC has the same length and layout. More discussions about the ST segment and DYN segment, which are the most import parts in CC, will present in Section 2.2.3.1 and Section 2.2.3.2.

**Arbitration grid level**

Arbitration grid level defines the grid of the media access control in FlexRay. There is an important concept, slot, needed to mention first. Slot is an interval of time that accessing to a communication channel is granted exclusively to a frame. The FlexRay protocol has two types of slots, ST slots and DYN slots. The arbitration grid in ST segment is the consecutive time intervals called ST slots. The arbitration grid in DYN segment is the consecutive time intervals called Minislots.

**Macrotick level**

Macrotick level consists of Macroticks. A Macrotick (MT) is an interval of time derived from the cluster-wide clock synchronization algorithm. Macrotick is also known as the global time[2] in the cluster. Different ECUs use Macrotick to synchronize in their clusters. A Macrotick consists of an integral number of Microticks. The clock synchronization algorithm can adjust the actual number of Microticks in a given Macrotick. The Macrotick represents the smallest granularity unit of the global time [11].

| Number of Microticks in a Macrotick | Microtick length [µs] | | | |
|---|---|---|---|---|
| | 0.0125 | 0.0250 | 0.050 | 0.100 |
| 40 | - | 1µs | 2µs | 4µs |
| 60 | - | 1.5µs | 3µs | |
| 80 | 1µs | 2µs | 4µs | - |
| 120 | 1.5µs | 3µs | 6µs | - |
| 240 | 3µs | 6µs | - | - |

**Table 2-2 Possible nominal Macrotick length**

The numbers of Macrotick in the CC are 10 to 16000. Table 2-2 shows the possible duration of the cluster wide nominal Macrotick is 1 to 6 µs. The typical length of a Macrotick is 1µs. Action point is the designated Macrotick boundaries. In other words, it is the instants that the transmission should start and end. FlexRay determines the global clock by averaging the times in synchronization nodes – the SYNC frame senders. The header segment of SYNC frame contains an indicator. The indicator is the deviation that measured between the frame's arrival time and its expected arrival time. It is used by the clock synchronization algorithm, [11].

**Microtick level**

Microtick is the time units derived directly from CC's external oscillator. They are not affected by the clock synchronization. It is a node-local concept. Microtick is controller-

[2] Global time: the cycle time. Cycle time is the time within the current CC, expressed in units of Macroticks. Cycle time is reset to zero at the beginning of each CC.

specific units. Different nodes can have different duration of Microticks. Microtick is known as local clock, only visible on the local CC. The granularity of a node's local time is a Microtick [11].

| Possible sample clock period [µs] | Number of samples per Microtick | | |
|---|---|---|---|
| | 1 | 2 | 4 |
| 0.0125 | 0.0125µs | 0.0250µs | 0.0500µs |
| 0.0250 | 0.0250µs | 0.0500µs | 0.100µs |
| 0.0500 | 0.0500µs | 0.100µs | - |

**Table 2-3 Possible Microtick length**

The Microtick length usually is not a configuration parameter. It is an implementation-dependent parameter that may be different for each node. Table 2-3 shows the possible Microtick length in FlexRay protocol. The duration of a Microtick length is from 0.0125µs to 0.05 µs according to the specification. The typical length is 0.025 µs.

#### 2.2.3.1 ST segment

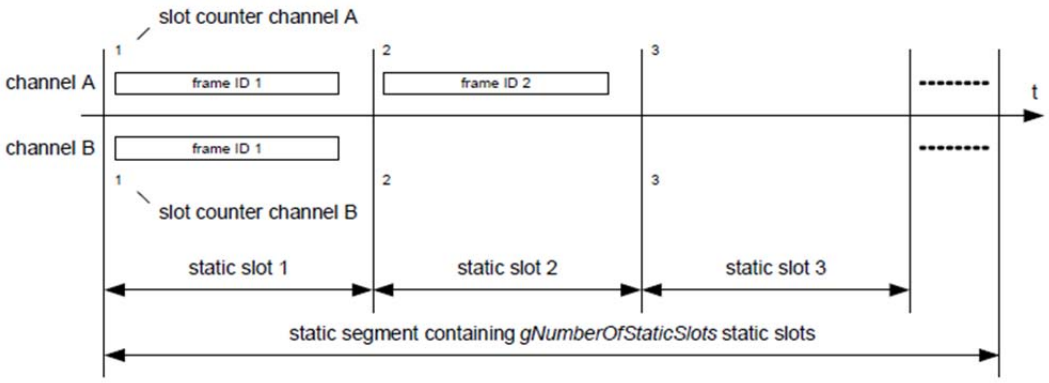


**Figure 2-14 Structure of ST segment [11]**

ST segment is a compulsory part in CC. It uses TDMA as the media access scheme. The length of ST segment is configurable but unchanged over the cycles. As it shown in Figure 2-14, ST segment consists of ST slots. In ST segment, the number of ST slots is fixed. The length of each ST slots is equal, despite the presence of ST frame in the slot or not. The transmission of the ST frames on the channel follows the predefined scheduling table. A ST slot only transmits one ST frame. Each frame has a unique frame ID associated with the slot. According to the FlexRay specifications, the maximum ST frame ID is 1023.

Each FlexRay node maintains two slot counters for two channels respectively. Both slot counters are initialized with 1 at the start of each CC and increased at the end of each slot. The clocks of the two channels are synchronized. Different nodes use the global synchronization time to decide when to start to send or to receive the frame.

Figure 2-14 illustrates a sample of the transmission patterns that are possible for the FlexRay node. In ST slot 1, the node transmits a frame on channel A and channel B. In ST slot 2, the node only transmits a frame on channel A. In ST slot 3, there is no frame transmit on either of the channels. If a slot has no frame to send, the slot stays empty, like the slot 3 in Figure 2-14.

The communication in ST segment should follow the following constraints [11]:

- Sync frames shall be transmitted on all connected channels.
- Non-sync frames may be transmitted on either channel, or both.
- Only one node shall transmit a given frame ID on a given channel. It is not acceptable to configure a cluster such that different nodes transmit in the same slot/channel combination in different cycles.

**Length of ST frame**

The data needed to transmit in the ST segment is packed into ST frame. All ST frames in a cluster have the same payload length. The size of ST frames is fixed in advance by the designers.

| Bit Rate [MBit/s] | 2.5 MBit/s | 5 MBit/s | 10 MBit/s |
|---|---|---|---|
| Minimum ST frame length [gdBit] | 86 | 87 | 89 |
| Maximum ST frame length [gdBit] | 2628 | 2631 | 2638 |

**Table 2-4 ST frame length [11]**

Table 2-4 gives the typical values of ST frame length under three different bit rates, in terms of $gdBit$. $gdBit$ is the nominal bit time. According to FlexRay specifications, the length of the payload in ST frame is 0 to 254 bytes.

**Number of ST slots**

The number of ST slots in ST segment is a configurable parameter with the notation of $gNumberOfStaticSlot$. It is a global constant for a given cluster. According to FlexRay specifications, the number of ST slots is from 2 to 1023 and at least is 2.

**ST slot length**

| Bit Rate [MBit/s] | 2.5 MBit/s | 5 MBit/s | 10 MBit/s |
|---|---|---|---|
| Minimum ST slot length [MT] | 9 | 6 | 4 |
| Maximum ST slot length [MT] | 658 | 661 | 397 |

**Table 2-5 Typical length of ST slot [11]**

Table 2-5 gives the typical length of ST slot under three different bit rates in terms of Macrotick (MT). ST slots in a cluster have the same number of Macroticks. The length of ST slot is 4 to 661 MT.

### 2.2.3.2 DYN segment4

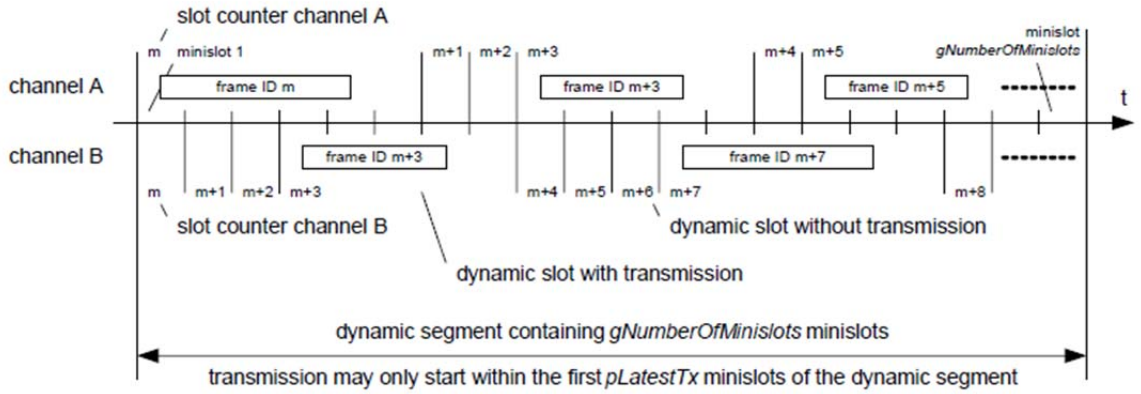


**Figure 2-15 Structure of DYN segment [11]**

DYN segment is optional in CC. The arbitration grid in DYN segment is Minislot. The Minislots length is fixed. Each Minislot contains an identical number of Macroticks.

Figure 2-15 shows a transmission sample in DYN segment. As we can see in Figure 2-15, the DYN frames and slots have different length. DYN frames have variable length. It depends on the data needed to transmit in that frame, namely the frame's payload length. A DYN slot could consist of one Minislot or multiple Minislots. The length of DYN slot varies to accommodate different DYN frames' size. If there is no transmission happened in a slot, the duration of the slot equals to the duration of a Minislot. Otherwise, the duration of the slot equals to the length of the DYN frame transmitted in that slot. Because the number of Minislots in the DYN segment is fixed and the slot size is changeable, the number of DYN slot is changeable.

FlexRay nodes maintain a slot counters for each channel. The counter adds 1 either after the end of a frame, in case of the presence of the data, or after one Minislot passed, in case of no data. The DYN slot counters are independent with each other. As we can see in Figure 2-15, the Minislots in two channels are synchronous, but the DYN slots in two channels are

not. In dual-channel system, the frame ID allocation can be different in 2 channels. The highest slot ID number is 2047.

It is import to notice that the actual start and end of transmission is not the nominal start and end of the slot. There is an action-point-offset we need to take into account. According to FlexRay specification, the delay in transmission must be greater than clock precision that is the limitation of fault tolerance in clock synchronization.

**Length of DYN frame**

The data transmitted in DYN segment is packed into DYN frames. Differing from fixed-payload ST frames, the DYN frames could have different payload lengths in a cluster. Even the same-frame-ID frames could have different frame sizes. The size of the frame depends on the payload length in the frame. According to FlexRay specifications, the payload length of DYN frame is 0 to 254 bytes.

**Minislot number and length**

| Bit Rate [MBit/s] | 2.5 | 5 | 10 |
|---|---|---|---|
| Minimum number of Minislot [Minislot] | 0 | | |
| Maximum number of Minislot [Minislot] | 3977 | 7977 | 7986 |
| Minimum length of a Minislot [MT] | 2 | | |
| Maximum length of a Minislot [MT] | 63 | | |

**Table 2-6 Maximum values of Minislot number**

Table 2-6 shows the maximum and minimum number of Minislots in DYN segment under three different bit rates. It also shows the minimum and maximum length of a Minislot in terms of Macrotick. According to FlexRay specifications, the number of Minislots in the DYN segment is 0 to 7986. The length of a Minislot is 2 to 63 MT.

**Parameter $pLatestTx$**

In DYN segment, there is an important concept $pLatestTx$ need to be mentioned. During the DYN segment, only if there is enough time until the end of DYN segment, a DYN frame could be transmitted. FlexRay protocol uses a parameter $pLatestTx$ to indicate the last instant the frame could be transmitted. $pLatestTx$ is the ID of the last Minislot that a frame could start to transmit. If the remaining time in DYN segment, at this instant, is shorter than $pLatestTx \times gdMinislot$, then this frame cannot be transmitted. Different frames have different $pLatestTx$. The range of $pLatestTx$ is 0 to 7980 Minislot according to FlexRay specifications. Table 2-7 presents the maximum value of $pLatestTx$ under different bit rates.

| Bit Rate [MBit/s] | 2.5 | 5 | 10 |
|---|---|---|---|
| Maximum $pLatestTx$ [Minislot] | 3967 | 7967 | 7980 |

**Table 2-7 Maximum values of $pLatestTx$**

#### 2.2.3.3 Symbol window (SYM) and Network Idle Time (NIT)

Symbol window (SYM) is an optional part in CC. It is a communication period in which a symbol can be transmitted on the network. The node shall transmit a symbol on a channel if the media access is in the ALL mode[3] and if a symbol is released for transmission. This period can be used for tests. The duration of the symbol window is 0 to 142 MT.

Network idle time (NIT) is a compulsory part in CC. It contains the remaining number of Macroticks within the CC which are not allocated to the ST segment, DYN segment, and symbol window. The network idle time is a communication-free period that concludes each CC. This period is required for clock synchronization, depends on system requirements. The duration of the network idle time use to define is 2 to 805 MT.

## 2.3 Conclusion

In this chapter, we gave an overview of FlexRay network, including the basic facts about the FlexRay network and the FlexRay protocol.

The first part overviews the core components in a FlexRay node: Host CPU, CC, BD, and BG. Next, we introduced the main FlexRay topology types such as bus topology, passive and active star topology, and hybrid topology. The FlexRay frame format is also provided in this chapter, which may give us the understanding about the physical layer in FlexRay protocol. Two typical tasks' trigger-modes: time-trigger and event-trigger, and two typical media access schemes: TDMA and FTDMA were introduced. The introductions give us a good understanding about the media access layer in FlexRay protocol. The timing hierarchies of CC, the basic timing unit in FlexRay protocol, were presented also. This part is the key background of this thesis, which could help to understand the schedulers presented later. After this chapter, the FlexRay knowledge related with scheduler design is clear.

[3] The ALL mode is one of the six operating modes of media access control. In the ALL mode frames and symbols are sent in accordance with the node's transmission slot allocation.

# 3

# Real-time Scheduling

The concept ‘real-time’ in computing refers to a time frame that is very brief, appearing to be immediate. A real-time system is a system that has to respond to externally generated input stimuli within a finite and specified period [19]. In other words, a real-time system is one that must process input information and produce a response or output result within a specified and reasonable time. Otherwise, might risk severe consequences including system failure. In a system with a real-time constraint, if the correct response or results are produced after a certain deadline, they expired and become useless. For example, in some widely implemented real time systems such as ABS, aircraft control, and over-temperature monitor in nuclear power station, any outdated response or results of these systems are useless even they are correct.

This chapter introduces the key concepts in the real-time system and real-time scheduling. At the end of this chapter, we present three commonly used scheduling algorithms and analyze their advantages and disadvantages respectively.

## 3.1 Real-time Tasks

Real-time tasks have three classifications: periodic task, sporadic task, and aperiodic task. Periodic tasks have a period, which means have a certain inter-arrival time. Sporadic tasks only have a minimum inter-arrival time. Aperiodic tasks have no known inter-arrival time requirement.

Based on the tolerance of violation of real-time requirements, tasks can also be divided into hard real-time task, soft real-time task and non-real-time task.

**Hard-real-time task**

Hard-real-time tasks have the hard deadline. It must be guaranteed to complete within a predefined amount of time. If there is a hard real-time task, we must try to avoid violating the requirement to the best ability. Safety-critical task is a typical sample of this type. We assume the tasks have deterministic deadlines. The system failure happens when any task is missed.

**Soft-real-time task**

Soft-real-time tasks also have real-time requirements. However, the result is not so severe when violation happens. Statistical distribution of response time of tasks is acceptable. A number of deadline violations can be tolerated.

**Non-real-time task**

Non-real-time tasks have no real-time requirement. For examples, entertainment and email service are non-real-time tasks.

## 3.2 Real-time Scheduling Policies

The real-time operation system is an operation system that is intended for real-time applications. Such operating systems serve application requests nearly real-time [20]. It is the computing system that must react within precise time constraints to inputs to the system. A reaction that occurs too late could be useless or even dangerous.

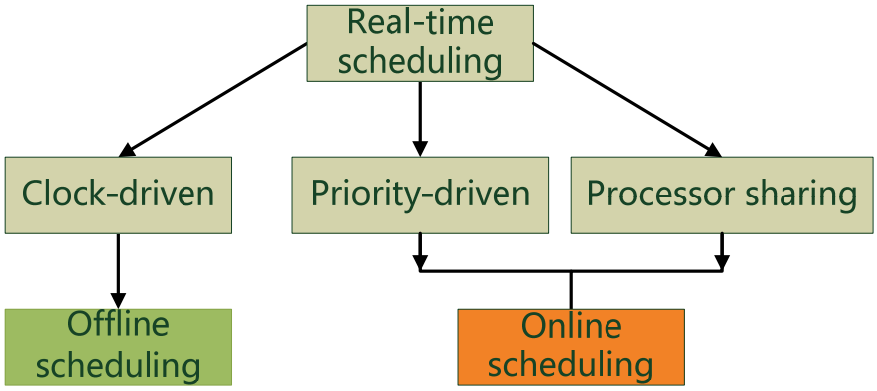


**Figure 3-1 Real-time scheduling Algorithms**

As shown in Figure 3-1, there are different criteria for the classifications of real-time scheduling algorithm. Based on the time to generate scheduling table, we classify scheduling algorithms into online and offline scheduling. Also there are other criteria to classify the real-time schedulers, such as clock-driven scheduler, priority-driven scheduler, and processor sharing scheduler. In the following paragraphs, we will discuss these classifications in detail.

### 3.2.1 Preemptive and Non-preemptive Scheduling

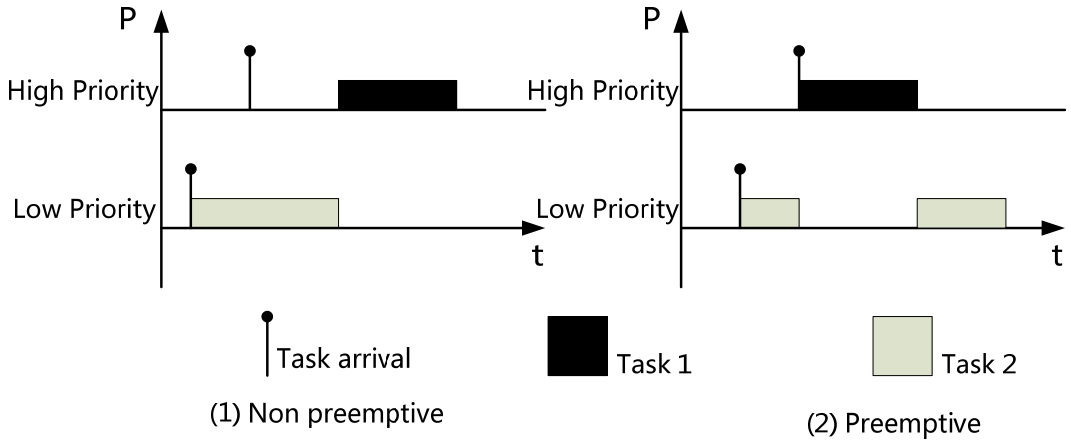


**Figure 3-2 Difference between non pre-emptive and pre-emptive systems**

There are two types of real-time systems: preemptive and non-preemptive. From Figure 3-2, it is clear that the main difference between these two types is the transmission continuity.

**Non-preemptive system**

In non-preemptive system, a task that has started will execute until its completion without any interruptions. It will defer execution of any higher-priority tasks.

**Preemptive system**

Preemption is the act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation, and with the intention of resuming the task at a later time. It is carried out by a preemptive scheduler, which has the power to preempt, or interrupt, and later resume other tasks in the system [21]. Tasks can preempt each other. The system allows the task with the highest priority to execute as soon as possible. Preemptive scheduling incurs more system overhead than non-preemptive scheduling, e.g. context switching time caused by preemption. Preemptive scheduling has the advantage that it has higher processor utilization than non-preemptive scheduling.

### 3.2.2 Offline and Online Scheduler

Scheduler creates a scheduling table. The system's CPU follows the scheduling table to transmit data. Scheduler can either offline or online creates the scheduling table. It depends on whether the input data known beforehand or not.

**Offline Scheduler**

Offline scheduler should have all of the input data and generate scheduling table before the system start. At run-time, a dispatcher is used to activate tasks according to the schedule generated before run-time [22].

**Online Scheduler**

If inputs of a scheduler do not know before the system start, the scheduler is an online scheduler. Online scheduler generates scheduling table at run-time.

### 3.2.3 Different Scheduling Approaches

Except the classifications presented above, we can also classify the scheduler based on other criteria. Mainly, there are three types of real-time schedulers. They are clock-driven scheduler, priority-driven scheduler, and processor sharing scheduler as shown in Figure 3-1. We will discuss the details about these three types in the following sections.

**Clock-driven Scheduling**

In clock-driven scheduling, the time instant to execute send/receive operations are initiated at predetermined points in time. So the clock-driven scheduling is also called time-triggered scheduling. Time- triggered systems are typically implemented non-preemptive static cyclic scheduling (SCS). Scheduling tables are built offline and stored in the memory before the system start to operate. Dispatchers transmit data and activate tasks based on predefined scheduling tables. In a distributed time-triggered system, we assume that the nodes' clocks are synchronized to provide a global reference of time.

At run-time, a dispatcher follows the schedule and makes sure that tasks are only executing at their predetermined time slots [22]. After the scheduler dispatches a task, it sets the periodic timer to generate an interrupt at the next task switching time. The scheduler will then go to sleep until the timer expires. This process is repeated throughout the whole operation. For each task, the time instant to execute is fixed, so the response time for each task is very predictable. Therefore, it suits the safety-critical applications.

Time-triggered scheduling is very simply and reliable, but lack of flexibility. It cannot deal with inputs' changes at runtime. A small change of input completely changes the whole scheduling table.

**Round-robin Scheduling**

Round-robin scheduling is one of the time-triggered scheduling algorithms. There are two types of round-robin scheduling approaches: regular round robin and weighted round robin.

Regular round-robin scheduling is commonly used in scheduling time-shared applications. Messages queue in the FIFO queue when they are ready to execute. The transmission starts from the beginning of the queue. If the task has not completed by the end of its timeslot, it is preempted and placed at the end of the queue. When there are N ready messages in the queue, each message gets one timeslot every N timeslots. A round is N timeslots.

In weighted round-robin, every message is assigned a weight $w_i$. The message will get $w_i$ timeslots in each round, and the duration of a round is $\sum_{i=1}^{n} w_i$. It is simpler than priority-driven scheduling, because weighted round-robin scheduling does not require different priority queues. Real-time networking commonly uses weighted round-robin scheduling.

**Priority-driven Scheduling**

Priority-driven scheduling is an important scheduling algorithm. Priority-driven systems typically implement by using preemptive priority-based scheduling. In priority-driven scheduling, each task is assigned a priority. System starts to execute from the highest-priority task among all of the ready tasks.

Based on the priority allocation, we can categorize priority-driven scheduling into Fixed Priority Scheduling (FPS) and Dynamic Priority Scheduling (DPS). The major difference of these two schedulers is whether the priorities of tasks can change at runtime. The commonly used priority-based scheduling algorithms are Fixed Priority Scheduling (FPS) and Earliest Deadline First (EDF). Section 3.3 will introduce these scheduling algorithms in detail.

Priority-driven scheduling is very flexible but not predictable. It can cope with work-load changes such as adding or removing the tasks. The predictability of the response time of a task decreases, as the flexibility increases. Therefore, safety-critical applications not often use the priority-driven scheduling.

**Processor Sharing Scheduling**

Processor-sharing-scheduling is a scheduling algorithm that assigns different fractions of the processor to the messages. Fraction means part of the time interval. In a processor, one message or job is executed at one time. The processor-sharing means assigning a fraction of processor's time to messages or jobs. For example, scheduler assigns 0.2 fraction of the processor to a job. This means this job can use 0.2 percent of the processor's time. In order to obtain the accurate fraction of processor-sharing, the timeslot has to be very small. However, when the timeslot is very small, the context switching spends a significant amount of time. This is a major drawback of the Processor-Sharing scheduling [23].

## 3.3 Classic Scheduling Algorithms

In this section, we introduce three classic priority-driven scheduling algorithms in the real-time system. Priority-driven scheduling is one of the most widely implemented and studied scheduling types. The algorithms we present in this section are milestones in the development of scheduling theory. Understanding these algorithms is very important to getting the general view of scheduling theory.

As we introduced in Section 3.2.3, based on the priority allocation, the priority-driven scheduling is categorized into fixed priority and dynamic priority scheduling. Before the discussion, we need to distinguish the concepts of jobs and tasks. Jobs compose tasks. In fixed-priority scheduling, all jobs in a task have the same priority that is computed offline.

The priorities are assigned to the tasks before the system start. This is called task-level priority. The highest-priority task is scheduled first. We use 1 to represent the highest priority. The priority decreases when the integer increases.

In dynamic-priority scheduling, the priority of the task is changeable, and the priority of the job is changeable either. The priorities are assigned to individual job instead of task. It can be further divided into job-level fixed-priority scheduling and job-level dynamic priority scheduling. The dynamic-priority scheduling has higher processor utilization and incurs more system overhead than the fixed-priority scheduling, because dynamic-priority scheduling needs to determine the priority at runtime.

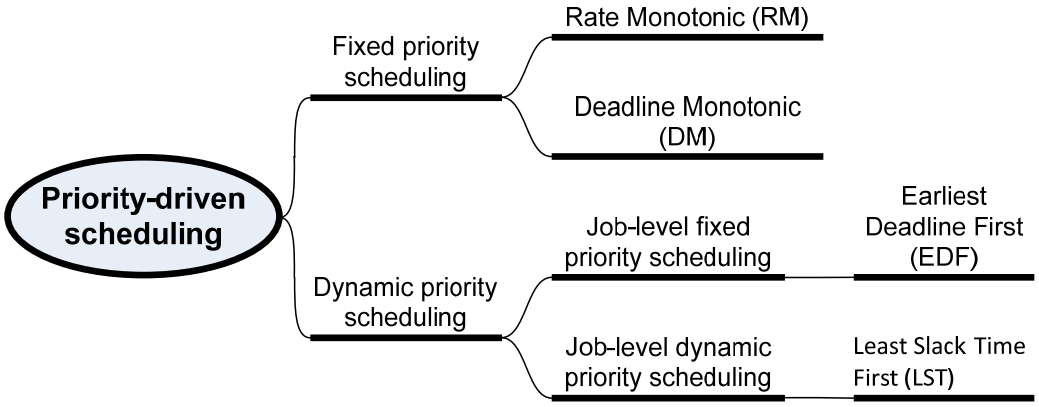


**Figure 3-3 Priority-driven scheduling algorithms**

Figure 3-3 presents the hierarchy of the classification of priority-driven scheduling. In the following sections, we will introduce two algorithms, *Rate Monotonic (RM)* and *Deadline Monotonic (DM)*, which are the optimal algorithms of fixed-priority scheduling. Also, we will introduce *Earliest Deadline First (EDF)* scheduling that is the optimal job-level fixed-priority scheduling.

### 3.3.1 Rate Monotonic scheduling (RM)

Liu and Layland [24] shows that the Rate Monotonic scheduling is the optimal fixed priority scheduling algorithm in terms of schedulability under the following restrictions:

- All tasks are independent of each other (e.g. they do not interact).
- All tasks are periodic.
- No task can block waiting for an external event.
- All tasks share a common release time (called the critical instant).
- All tasks have a deadline equal to their period.

These first four restrictions have already been relaxed by different approaches, except the last one. Therefore, we can say Rate Monotonic scheduling is the optimal fixed-priority scheduling algorithm when tasks' deadline equal to their periods. The optimal algorithm means it always generate a feasible schedule if the task set is schedulable by any static priority algorithms.

RM refers to assigning priorities as a monotonic function of the rate (frequency of the occurrence) of these tasks. In RM scheduling, the individual task priority assignment bases on the periods of the tasks. The priorities are inversely proportional to the periods, in other words, the shorter the period the higher the priority.

RM scheduling can be used statically on any hard real-time systems to decide if the system is schedulable. In fixed-priority scheduling, the upper bound of schedulability is 69.314% of the processor utilization. However, the large runtime overhead is one of the shortcomings of RM scheduling. Furthermore, only periodic tasks can apply RM scheduling.

### 3.3.2 Deadline Monotonic scheduling (DM)

Many systems need tasks' deadlines shorter than tasks' periods, which violate the assumption in RM scheduling mentioned in the previous section. In 1982, Leung and Whitehead [25] shows that Deadline Monotonic (DM) scheduling is another optimal algorithm of fixed-priority scheduling when tasks have deadlines less than (or equal) to periods.

In DM scheduling, tasks' priorities are inversely proportional to the order of tasks' deadlines. If happens that tasks have the same deadlines, the priority assignment will be arbitrary ordered among the same deadline tasks.

### 3.3.3 Earliest Deadline First (EDF)

One of the most widely used optimal dynamic-priority scheduling algorithms is Earliest Deadline First (EDF) scheduling. EDF processors are priority-driven and preemptive. Dertouzos [26] showed that EDF is optimal among all preemptive scheduling algorithms. Mok also presents another optimal algorithm, Least Laxity First (LLF) [27], which assigns the processor to the active task with the smallest laxity[4]. However, LLF has larger overhead than EDF. That is the reason why EDF is the most commonly used algorithm in dynamic-priority scheduling. EDF is also called Deadline Driven Scheduling Algorithm. The earliest deadline task, among all tasks ready for execution, gets the highest priority. The priorities assigned to tasks are inversely proportional to the absolute deadlines of the tasks. Dynamic-priority scheduling is still schedulable when the processor utilization approaches 100%. It is more flexible than fixed priority scheduling algorithm.

---

[4] The laxity is the difference between the absolute deadline and the estimated worst-case finishing time.

## 3.4 Conclusion

This chapter focuses on the real-time scheduling theory. We first introduced the key scheduling policies. There are lots of criteria to classify scheduling policies. Based on the transmission continuity, input data certainty and the driven type of the transmission, we introduced three methods. These introductions give the basic knowledge about the scheduling methods. Then we presented three representative scheduling algorithms. RM scheduling is an optimal scheduling algorithm of fixed priority scheduling with the constraint that all tasks have a deadline equal to their period. DM scheduling is another optimal algorithm of fixed-priority scheduling when tasks have deadlines less than (or equal) to periods. The third scheduling algorithm presented is EDF scheduling, which is the optimal priority-based dynamic scheduling algorithm. We have known the main factors in the scheduling theory after the discussion of this chapter.

# 4

# Schedulability of Simple FlexRay Networks

The important issue in scheduler design is to find whether all tasks are schedulable during the peak-load. In order to know the system schedulability, we can calculate the total utilization of the CPU or the response-time of all tasks in the worst-case scenarios (at peak-load) [22]. These measurements could use to determine whether the tasks are schedulable.

In Chapter 4, we provide the basic constraint of the ST segment schedulability analysis, which calculates the minimum number of ST slots required in the segment. Then we analyze the worst-case response time of the DYN messages to verify whether the tasks in a given task set can meet the deadlines.

## 4.1 System Architecture

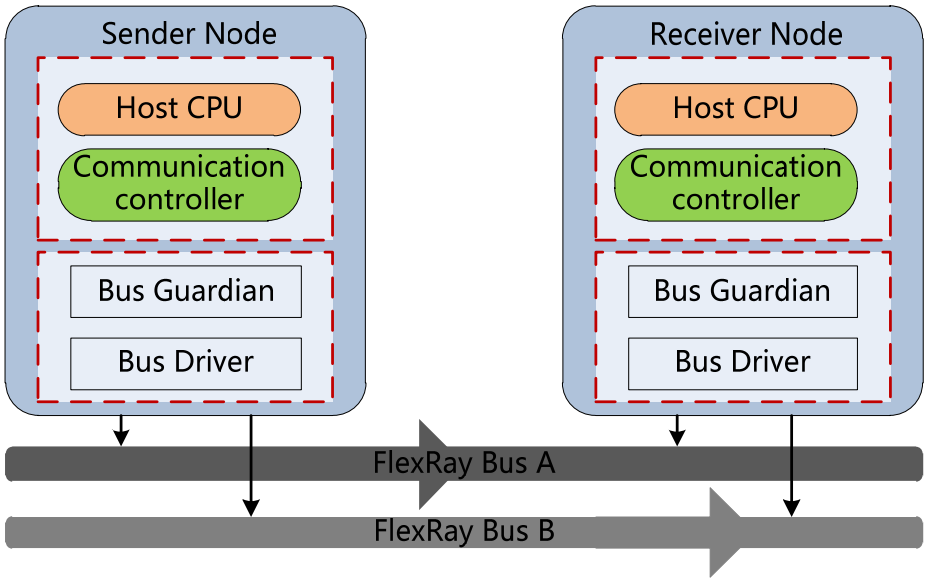


**Figure 4-1 Simple FlexRay Node to Node Communication**

As shown in Figure 4-1, there are two nodes in the network, which are all connected with the FlexRay bus[5]. The nodes might produce some safety-critical tasks or diagnostic tasks. The tasks have different timing requirements and generation frequencies, such as time-critical tasks and non-time-critical tasks. The outputs messages of tasks inherit the timing requirements and generation frequencies from their sending tasks. As we introduced in Chapter 2, the CC of FlexRay protocol consists of a ST segment and a DYN segment. Each segment has different media access control. The working principles of these two segments can adapt different tasks' transmissions. Therefore, two segments should have different real-time kernels that consist of two different schedulers. Figure 4-2 shows the two-scheduler concept.

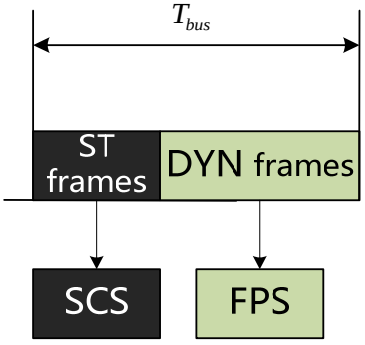


**Figure 4-2 Two schedulers for two segments**

The nodes may have different tasks to finish or frames to send. For time-critical tasks, namely ST tasks, we use ST cyclic scheduling (SCS). The working principle of SCS is that the set of tasks or messages follows a static scheduling table to transmit and repeat. For DYN tasks, we use fixed priority scheduling (FPS) that transmits tasks or messages by the predefined priorities.

ST tasks use SCS scheduler and have the highest priority among all the tasks. These SCS tasks are preemptive. Their start time is off-line fixed in the scheduling table. SCS activities are triggered based on a node's local clock. Other tasks use FPS scheduler. FPS tasks are scheduled based on priorities. The high-priority task preempts a lower-priority task. These tasks can only be executed in the idle time of SCS scheduling table. When several tasks are ready on a node, the highest-priority task is activated and preempts the other tasks.

[5] FlexRay can serve as a dual-channel system, here we only use one channel to communication, the other channel serve as the redundant channel to make sure fault tolerance in the system.

## 4.2 Task & Message

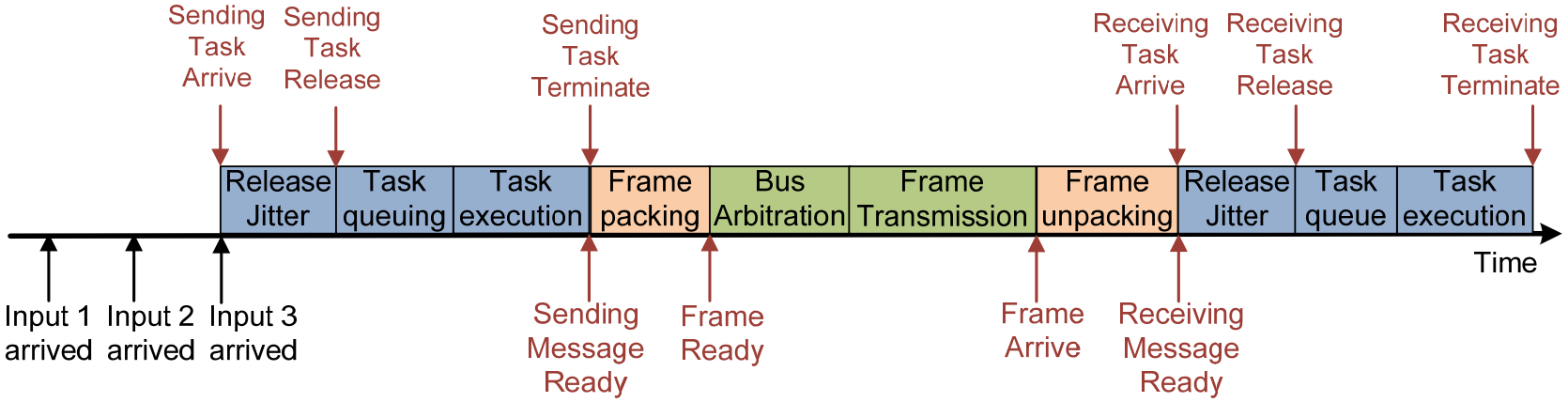


**Figure 4-3 Timeline of FlexRay DYN data transmission**

Before analyzing the transmission timing in detail, we have to distinguish the concepts of task from message[6]. The relationship between the task and the message is shown in Figure 4-3. The point 'Sending Task Arrive' shows that a task becomes ready after all its inputs have arrived. We assume tasks issue the output messages at the end of their executions. The point 'Sending Task terminated' shows this instant. Output messages become ready after the sender task has finished, as the point 'Sending Message ready' shown in Figure 4-3. The point "Receiving message ready" shows the messages become ready at the receiver processor after the physical transmission has ended. Moreover, we need to notice that a task could produce several massages.

## 4.3 Definition of Latency

Latency is the duration between data ready for transmission at the sender and data ready for consumption at the receiver [28]. Let us consider the timeline in Figure 4-3. The blue boxes represent the time to execute tasks inside a single processor, which is from the tasks' arrival to the tasks' termination, and issue the output messages. The green boxes represent the time interval to transmit a frame on the bus. The interval is from the frames that are ready to be transmitted at the sending bus driver to the frame ready for consumption at the receiving bus driver.

There are two types of latency. One is end-to-end latency, which is from sending task arrive to receiving task terminate. We can see this latency in Figure 4-3 from red arrow "Sending task Arrive" from red arrow "Receiving task terminates". Other latency is communication latency which is from sending Communication Controller to receiving Communication Controller [28]. This latency is only related with communication protocol and physical media. It is the latency in data link layer and physical layer. We can see this latency is the length of two green boxes in Figure 4-3.

The reason why we only do the calculation of communication latency is that the ECUs might from different venders and have different hardware specifications, so it is very difficult to do the holistic schedulability analysis. There are a lot of existed researches on processor schedulability analysis, for readers who interested can refer to [29-32].

There is one thing need to mention. The "message" in Figure 4-3 is the data sequence need to be transmitted which is the payload in FlexRay frame. In communication latency, the data which ready to transmit is the FlexRay frames. In FlexRay, Communication Controller is responsible for implementing the protocol aspects to the payload data received from Host CPU. In other word, Communication Controller is responsible for frame packing or segmentation. As mentioned before, in this thesis, we do not deal with frame packing or

[6] This thesis doesn't consider the frame packing or frame segmentation. These functions are realized at higher layer which are out of the scope of this thesis. To simplify the problem, this thesis assumes that one message is packed into one frame.

segmentation, so the time for frame packing or segmentation is only considered including in end-to-end latency.

## 4.4 ST Segment Schedulability Analysis

In timing analysis of the FlexRay network, we considered the ST messages are schedulable if it is possible to generate a valid static scheduling table. So we need to build the scheduling tables for SCS tasks and ST frames. Before doing this, we first simply compare the available ST slots and the minimum required ST slots. This comparison gives the general view of available bus data rate that can use to system design.

The comparisons of available ST slots and required ST slots are the necessary but insufficient condition to see whether a given set of messages is schedulable. Necessary but insufficient condition means that if a schedule cannot pass this test, it is not schedulable. However, there might be some schedules that can pass this test but still cannot generate a valid schedule.

The comparisons of the minimum required ST slots and available ST slots are the basic schedulable constraint of the system. If the available ST slots are smaller than the minimum required ST slots, the system is not schedulable. In this section, we would like to calculate the minimum required ST slots in the system.

In FlexRay system, each message assigned one transmission buffer based on the slot ID and the cycle ID. The identifier of the channel on which the transmission shall occur is not necessary because the dual-channel is only been used as redundancy channel. Aiming to increase the resource-utility rate, this thesis chooses to use slot multiplexing in ST segment. Slot multiplexing means messages do not have the exclusive right of the slots. More messages can share one slot in the ST segment.

Hence message $m's$ schedule is defined by the value of the cycle ID and slot ID of the transmission slot. The value of the cycle ID is represented by the 2-tuple vector: base cycle $b_m$ and cycle repetition $r_m$ [33]. Base cycle $b_m$ is the cycle ID in which the first message-arrival sends. It has the constraint $b_m \in [0,63], b_m \in Z$. Cycle repetition $r_m$ is the multiple of CC between two successive transmission cycles in which the message-arrivals send. It has the constraint $r_m = 2^n, n \in [0,6], n \in Z$ to allow a periodic occurrence in the 64 cycles.

The transmission-slot ID is represented by the 2-tuple vector: base slot $s_m$ and slot repetition $p_m$. $s_m$ is the ST slot ID in which the first message-arrival sends. Slot repetition $p_m$ is the multiple of ST slots between two successive message-arrivals within one CC. It has the constraint $p_m = 2^n, n \in [0, \log_2 gNumberOfStaticSlots], n \in Z$ to allow the successful slot multiplexing.

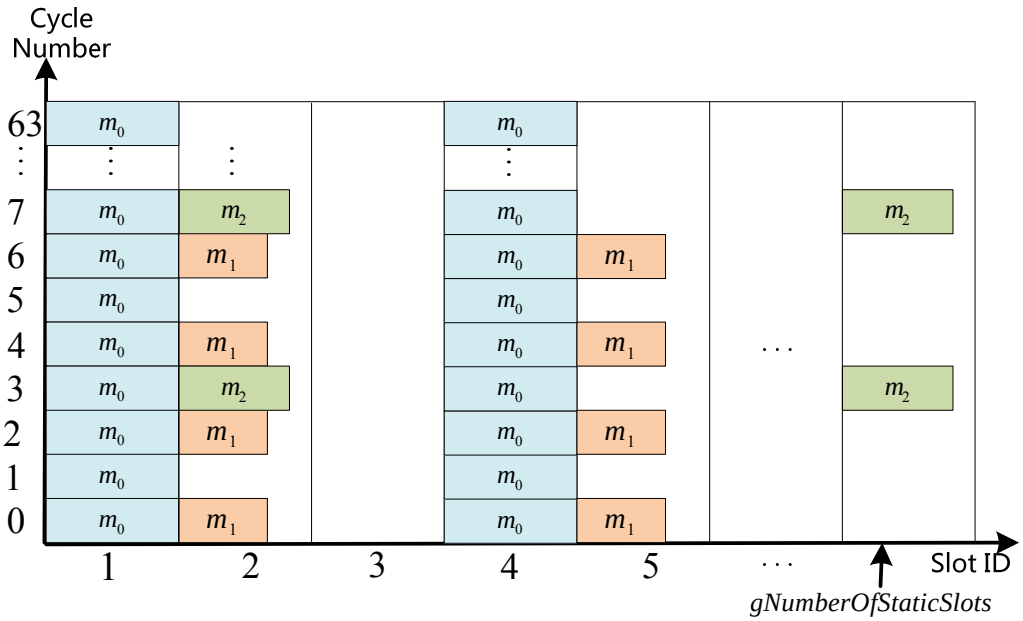


**Figure 4-4 Example schedule**

As illustrate in Figure 4-4, the base cycle $b_m$ is 0 for $m_0$ and $m_1$, 3 for $m_2$. The cycle repetition $r_m$ is 1 for $m_0$, 2 for $m_1$ and 4 for $m_2$. $s_m$ is 1 for $m_0$ and 2 for $m_0$ and $m_1$. The Slot repetition $p_m$ is 3 for $m_0$ and $m_1$, $gNumberOfStaticSlot - 2$ for $m_2$.

The message set $M_{ST}$ is the ST messages waiting to send in a cluster. The previous study [33] shows that the minimum number of ST slot $gNumberOfStaticSlots_{min}$ required[7] for the transmission of the messages set $M_{ST}$ is:

$$gNumberOfStaticSlots_{min} = \sum_{\{m \in M_{ST}\}} \frac{1}{r_m} \tag{4.1}$$

This equation only suits for the schedule that every message transmits once per cycle. This fact limits the length of CC at least should equal to the shortest period of ST messages, in order to accommodates the transmissions of ST messages. Furthermore, according to FlexRay specification, the maximum cycle ID is 64. Therefore, this constraint limits the maximum system bandwidth and reduces the system capacity. In order to increase system bandwidth, the messages should be able to transmit multiple times per cycle. The calculation of the minimum ST slots required for the transmission then becomes the following formula:

[7] This thesis assumes that each message fits within one frame and uses a full slot to transmit.

$$gNumberOfStaticSlots_{min} = \sum_{\{m \in M_{ST}\}} \frac{gNumberOfStaticSlot_m}{r_m} \quad (4.2)$$

$gNumberOfStaticSlot_m$ is the total ST slots message $m$ is allocated. $M_{ST}$ is the ST message set in the cluster. It is import to notice that this result is under the assumption that one message-arrival packs into one frame and sends in one slot. Furthermore, the FlexRay specifications regulate that the minimum number of ST slots in a system is 2. However, the actual number of the ST slots needed does not always equal to the minimum ST slots required. The minimum value only happens when there is no empty slot left between any two ST slots. It means that the transmission patterns of the ST messages match each other perfectly. However, this idealized case does not happen in every case.

According to FlexRay specifications, system can have maximum 1023 ST slots. If $gNumberOfStaticSlots_{\min} > 1023$ happens, the system is non-schedulable. The minimum number of ST slots $gNumberOfStaticSlots_{\min}$ can only be used as the basic schedulable condition. The actual number of ST slots should base on the scheduling result.

## 4.5 DYN Segment Schedulability Analysis

First thing in schedulability analysis is to decide what scheduling policy we want to use. There are many exist scheduling algorithms as we introduced in Section 3.3. Although dynamic-priority scheduling algorithms can give optimized system utility rate, but the implementation of dynamic-priority scheduling are far more complex than fixed-priority scheduling algorithms. Therefore, this thesis uses the fixed-priority scheduling as the scheduling policy.

There are three different approaches of schedulability analysis: utilization-based test, demand-based test, and response-time test. Utilization-based test is based on the utilization of the task-set under analysis. Demand-based test is based on the processor demand at a given time interval. Response-time test is based on the worst-case response-time of each task in the task-set. The worst-case response time calculation is the best approach to test system schedulability in the circumstances that no hardware can use for analysis. If all the messages can meet their deadlines, the system is schedulable.

FlexRay is a hybrid-bus system where TT and ET tasks share the same media and time resources. Tasks are scheduled using both TT and ET scheduling. However, the TT and ET tasks do not exchange time-critical communication [34]. Because the TT and ET activities share the same resource, and TT activities have the highest priority, thus the ET tasks can only execute in the slack of the TT scheduling table.

In the following sections, we discuss the method of the DYN messages' worst-case response time calculation in FlexRay network.

### 4.5.1 Worst-case Communication Latency

According to the latency analysis in Section 4.3, a frame's communication latency consists of two parts (the two green boxes): the bus arbitration delay and the transmission delay. Therefore, the worst-case communication latency is given by the following formula:

$$R_m = w_m + C_m \tag{4.3}$$

$w_m$ is the longest delay caused by the media contention before the transmission. We can see this part as the waiting delay. $C_m$ is the longest time taken to send the frame *m* on the bus. This part is the transmission delay in Figure 4-3.

In order to calculate $R_m$, we should know these two delays. In the following paragraphs, we will present the methods to calculate these unknown parameters.

### 4.5.2 Transmission Delay $C_m$

Based on the definition of $C_m$, we know that $C_m$is related to the message length and the FlexRay bus bit rate. So we can get:

$$C_m = Frame\_size_m / bus\_speed \tag{4.4}$$

In FlexRay protocol, the bus gross data rate is 10Mbit/s and the effective data rate is 5Mbit/s.

### 4.5.3 Bus Arbitration Delay $w_m$

$w_m$ represents the longest delay of the bus arbitration before transmission. Three possible reasons may contribute to this delay: the transmission of ST segment, the transmission of the DYN frames having lower Frame ID than message $m$, denoted by $lf(m)$, and the transmission of the DYN frames having same Frame ID but higher priority than message $m$, denoted by $hp(m)$. However, the delay $w_m$ is the multiple of CC. It is hard to define $w_m$ in terms of ST segment, $hp(m)$, or $lf(m)$.

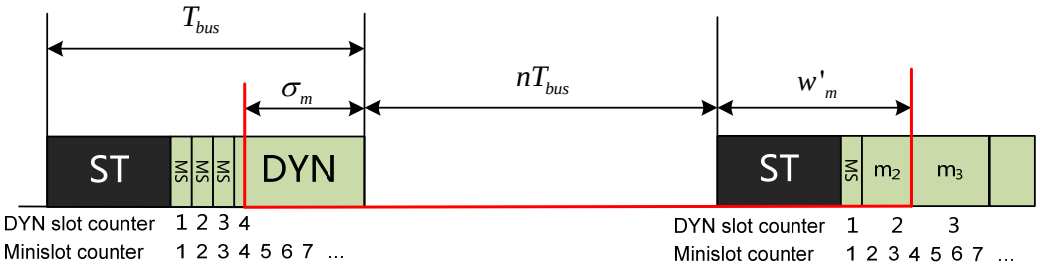


**Figure 4-5 Response time of the DYN messages**

The delay $w_m$ is the duration between two red lines in Figure 4-5. As we can see in Figure 4-5, we divide this delay into three parts to simply the analysis. We introduce the following notations:

- $\sigma_m$ denotes the time interval that starts from the instant that sender task generates the message to the end of the generation CC.
- $ms(m)$ denotes the unused Minislots set that have lower frame ID than message $m$. Although these Minislots are not used for data transmission, they still delay the transmission for the length of a Minislot. $nT_{bus}$ denotes the number of CCs that cannot transmit message $m$ because of the transmission of messages from sets $hp(m)$, $lf(m)$ and $ms(m)$.
- $w'_m$ denotes the delay that starts from the beginning of the message $m$'s transmission CC to the transmit point of message $m$. $gdCycle$ denotes the length of the CC.

Therefore, the formula of delay $w_m$ is:

$$w_m = \sigma_m + nT_{bus} + w'_m \tag{4.5}$$

**Parameter $\sigma_m$**

The maximum value of $\sigma_m$ appears when the generation of the frame just after the frame's allocated slot and there is no data transmission happen before this its generation. Therefore, the formula of worst-case $\sigma_m$ is [35]:

$$\sigma_m = gdCycle - (ST_{bus} + (Frame\ ID_m - 1) \times gdMinislot) \tag{4.6}$$

$ST_{bus}$ is the length of the ST segment. $FrameID_m$ is the frame ID of message $m$. The value of $FrameID$ is from 1 to 2047 in binary format. $gdMinislot$ is the duration of a Minislot. We assume the CC only consist of ST segment and DYN segment, regardless the symbol window and network idle time. In the following discussion, $T_{bus}$ is used as the approximation of the length of CC. It does not introduce any significant pessimism.

**Heuristic solution of $w'_m$**

$w'_m$ is the notation of the delay that starts from the beginning of the CC in which the message m sent until the real transmission of message m happens. The holistic solution gives the exact value of $w'_m$. However, the computation times of the highly complex algorithms are not practical. Therefore, we choose the heuristic solutions instead of the holistic solution. The

heuristic solution has significantly lower complexity and needs extremely short computation times, while at the same time producing results close to the ones offered by the optimal implementations [35].

The maximum value of $w'_m$ happens when the messages is transmitted in the last possible slot. The value of Minislot counter at this instant is smaller than the value of $pLatestTx$. $pLatestTx$ is the number of the last Minislot in which a frame transmission can start in the dynamic segment [11]. The value of $pLatestTx$ depends on the size of DYN frames and the length of the Minislot. Therefore, the heuristic solution of the worst-case $w'_m$ [35] is:

$$w'_m = ST_{bus} + pLatestTx_m \times gdMinislot \tag{4.7}$$

**Parameter** $nT_{bus}$

Message $m$ that is packed into frame $m$ with $FrameID_m$ cannot be sent during the CC if at least one of the following conditions is fulfilled:

- One of the messages from messages set $hp(m)$ occupies the DYN slot that corresponds to $FrameID_m$ in the cycle.
- The remaining time in the DYN segment is shorter than the frame length of message $m$. This might because of the large latency that caused by the data or Minislots that have the *FrameID* smaller than $FrameID_m$. This latency delays the message so that it misses the last instant that can transmit in the CC. These elements belong to sets $lf(m)$ and $ms(m)$.

Based on the discussions above, the delay $nT_{bus}$ can be written as[35]:

$$nT_{bus}(t) = \left( BusCycles_m(hp(m,t)) + BusCycles_m(lf(m,t), ms(m,t)) \right) \times gdCycle \tag{4.8}$$

The reason why introduce the time interval $t$ is that the message in the sets $hp(m)$, $lf(m)$ and $ms(m)$ are changing all the times. Messages keep coming into the queue and sending out to the scheduler. The analysis should focus on a given time interval. So we need to consider all the messages in sets $hp(m)$, $lf(m)$ and $ms(m)$ during this time interval, which would delay the message $m$. $hp(m,t)$ is the number of messages that have higher priority than message $m$ during the time interval t. $lf(m,t)$ is the number of messages that have lower $FrameID$ than $FrameID_m$ during the time interval t. $ms(m,t)$ is the number of slots which have no data to transmit but have lower $FrameID$ than $FrameID_m$ during the time interval t, so that the message $m$ has to wait until the DYN slot counter reaching the $FrameID_m$.

**Heuristic computation for $\boldsymbol{BusCycles_m(hp(m,t))}$**

If there is a message that has the same $FrameID$ but higher priority than message m, it delays the transmission of m for one CC. Therefore, the total delay caused by the messages in the set $hp(m,t)$ is [35]:

$$BusCycles_m(hp(m,t)) = \left|hp(m,t)\right| \tag{4.9}$$

**Heuristic computation for $\boldsymbol{Delay_m(ms(m,t))}$**

The holistic solution of $BusCycles_m(ms(m,t))$, namely the optimal solution, can calculate the tightest worst-case response time. However, the high complexity of the optimal algorithm leads to the long computational time, which is not practical when calculate the response times of many messages. The holistic solution only can calculate the results in the reasonable time for up to 20 DYN frames [35].

We also use the heuristic solution to compute the $BusCycles_m(lf(m,t), ms(m,t))$. If the massage m is transmitted in the DYN slot corresponded to $FrameID_m$ there are up to $FrameID_m - 1$ unused Minislots before the transmission of message $m$ in the worst-case. We use the approximate number of Minislots in the worst-case scenario instead of the actual number of Minislots. The duration the Minislot is very small, compared to the duration of CC. The approximation does not introduce the crucial pessimism [35]. Therefore, the delay caused by messages in the set $ms(m,t)$ is [35]:

$$Delay_m(ms(m,t)) = (FrameID_m - 1) \times gdMinislot \tag{4.10}$$

**Heuristic computation for $\boldsymbol{BusCycles_m(lf(m,t))}$**

The next step we need to compute the delay $BusCycles_m(lf(m,t))$. This problem transforms into a *one dimension bin packing (1DBP)* problem. The *bin packing problem* tries to maximize the number of bins that can be filled with a fixed minimum capacity by a given set of items with specified weights. More details about *bin packing problem* can refer to [36-38].

There are many existing bin packing algorithms such as First-Fit, Best-Fit, Next Fit, Worst-Fit and Last-Fit. These existing algorithms generally can be divided into two categories: optimal algorithm and heuristic algorithm. Optimal algorithm can get exact result but always is the NP-hard algorithm that has unacceptable computational time, especially for a large number of inputs. Heuristic algorithm can significantly decrease the algorithm complexity and computational time. Although it cannot offer tightest results, the algorithm outputs are very close to the optimal algorithm when there are large numbers of inputs. For this reason, this thesis chooses a greedy heuristic bin packing algorithm, the First-Fit Decreasing Algorithm (FFD), as the algorithm to calculate $BusCycles_m(lf(m,t))$.

The FFD algorithm first sorts the items in decreasing order by size, and then inserting each item into the first bin in the list with sufficient space. This algorithm packs the largest items

first and is more likely to produce an optimal solution than the simple First-Fit method [39]. The First-Fit packing algorithm is one of the bin packing algorithms. For a number of bins, it always places the next box into the lowest-numbered bin it will fit into [40]. In our case, the elements in $lf(m,t)$ are the items. The DYN segments are bins, namely $BusCycles_m(lf(m,t))$. The minimum capacity required to fill a bin is $pLatestTx_m \times gdMinislot$.

For any given DYN message sets, if the $FrameID_m$ and all the message sizes are known, the delay $BusCycles_m\big(lf(m,t)\big)$ only varies with the DYN segment length $DYN_{bus}$. In conclusion, the analysis of the worst-case response time $R_m$ of the DYN message $m$ shows that $R_m$ only varies with the CC length $gdCycle$.

## 4.6 Conclusion

In this chapter, we first discussed the different scheduling algorithms that should use in two segments in CC. Then we differentiated the concept between task and message. Differentiation two confusing concepts are crucial to help to understand the scheduling timeline. Next to this, the timeline of the DYN was provided. Then we defined the concept of the latency, which gave the clear understanding of the transmission timeline. Furthermore, we discussed the basic scheduling constraint in ST segment, namely the minimum required ST slots. The basic constraint is the first condition that the scheduler should check in order to produce a valid scheduling table. Finally, the worst-case response time of the DYN messages was analyzed and calculated in detail. We chose the FFD algorithm to calculate a part of the delay.

# 5

# Scheduler Design for Simple FlexRay Networks

After all the background knowledge and schedulability analyses, this chapter we will design the scheduler for simple FlexRay network. This chapter is generally divided into two parts: ST segment and DYN segment schedulers design. The structure in each part is formed by three sections. The first section of each segment provides some examples to show the different schedulability resulting from different values of those configurable parameters in FlexRay protocol. After the comparisons of examples, the second section defines the problems needed to configure. Then follow with analysis for each parameter and the suggested solutions for these problems. The final section is the scheduling algorithm for ST segment and DYN segment respectively.

## 5.1 Scheduler Design for Simple FlexRay ST Segment

Before the discussion we would like to present some notations which are used later.

- $b_m$ Base cycle is the value of the cycle counter in which the first message[8] arrival sends with the constraint $b_m \in [0,63], b_m \in Z$;

- $D_m$ [μs] Message $m$'s deadline is the duration from message generation to message expiration;

[8] The concept of message is equivalent to the signal in the discussion of this thesis. The signal may send from a sensor or processor, etc.

- *gdActionPointOffset* [MT] is the number of Macroticks the action point is offset from the beginning of a static slot or symbol window;
- *gdBit* [μs] is the nominal bit time;
- *gdBitMax* [μs] is the maximum bit time taking into account the allowable clock deviation of each node;
- *gdCycle* [μs] is the duration of the communication cycle;
- *gdMacrotick* [μs] is the duration of the cluster wide nominal Macrotick;
- *gdMinPropagationDelay* [μs] is the minimum propagation delay of a cluster;
- *gdMaxPropagationDelay* [μs] is the maximum propagation delay of a cluster;
- *gdStaticSlot* [μs] is the length of a ST slot;
- *gdTSSTransmitter* [gdBit] is the number of bits in the Transmission Start Sequence;
- *gNumberOfStaticSlots* is the total number of ST slots in the ST segment;
- $gNumberOfStaticSlots_m$ is the total number of ST slot IDs which message $m$ is allocated, $gNumberOfStaticSlots_m \in \mathrm{N}$;
- $gNumberOfCycle_m$ is the total number of cycles IDs message $m$ allocated, $gNumberOfCycle_m \in \mathrm{N}$;
- $MessageLengthST_m$ is the number of bits constituting the static message $m$ in the cluster;
- $p_m$ Slot repetition is the multiple of ST slots in between two successive message arrivals within one CC with the constrain $p_m \in [1,1022]$, $p_m \in \mathrm{N}$;
- $r_m$ Cycle repetition is the multiple of CC in between two successive transmission cycles in which message arrivals send. It has the constraint $r_m = 2^n, n \in [0,6], n \in Z$ to allow a periodic occurrence in the 64 cycles;
- $ST_{bus}$ is the length of the ST segment;

- $s_m$ Base slot is the value of the ST slot counter in which the first message arrival sends with the constraint $s_m \in [1,1023], s_m \in Z$;

- $T_m$ Message $m$ 's minimum inter-arrival time is the minimum time interval between two successive values of message $m$ which inherited from its sender task;

### 5.1.1 Problem definition

In automotive communication, the time-triggered tasks and messages require the guaranteed communication latency. They have strictly timing constraint and highly predictability requirements. ST segment in FlexRay protocol uses TDMA as the media access mechanism. The messages that transmitted in ST segment have very predictable response time. Therefore, ST segment suits the time-triggered tasks and messages. The messages transmitted in ST segment are called ST messages.

Every system has an optimal system configuration. However, the goal of the ST scheduler design in this thesis is not to find an optimal configuration for a system but to find a schedulable configuration for any FlexRay systems. Thus, the schedulable constraints defined in this thesis may vary depend on the system requirements.

From the aspect of system schedulability, we want to accommodate the transmission of every ST messages to meet their timing constraint. From the aspect of system expansibility, we want to accommodate the messages by using as less system resource as possible. Therefore, the idea of the scheduler design is to find the best way to accommodate the ST messages in order to satisfy the timing constraints and use as less resource as possible.

For any sporadic ST message $m$ generated by the sporadic task, the known parameters are the minimum inter-arrival time $T_m$ and the message size $MessageLengthST_m$. Thus, a ST message $m$ can be represented by a 2-tuple vector:

$$m = \{T_m, MessageLengthST_m\} \tag{5.1}$$

A ST message is schedulable only if it has a valid static scheduling table. The valid static scheduling table means that the ST messages can meet their communication constraints as long as they follow the scheduling table to transmit. These communication constraints are also the schedulable constraints for the scheduler. Therefore, we should analyze the schedulable constraints first in order to schedule the ST messages.

#### 5.1.1.1 Configurable Parameters

Let's consider three examples of different system configuration first. Then we will discuss the different schedulabilities and related parameters.

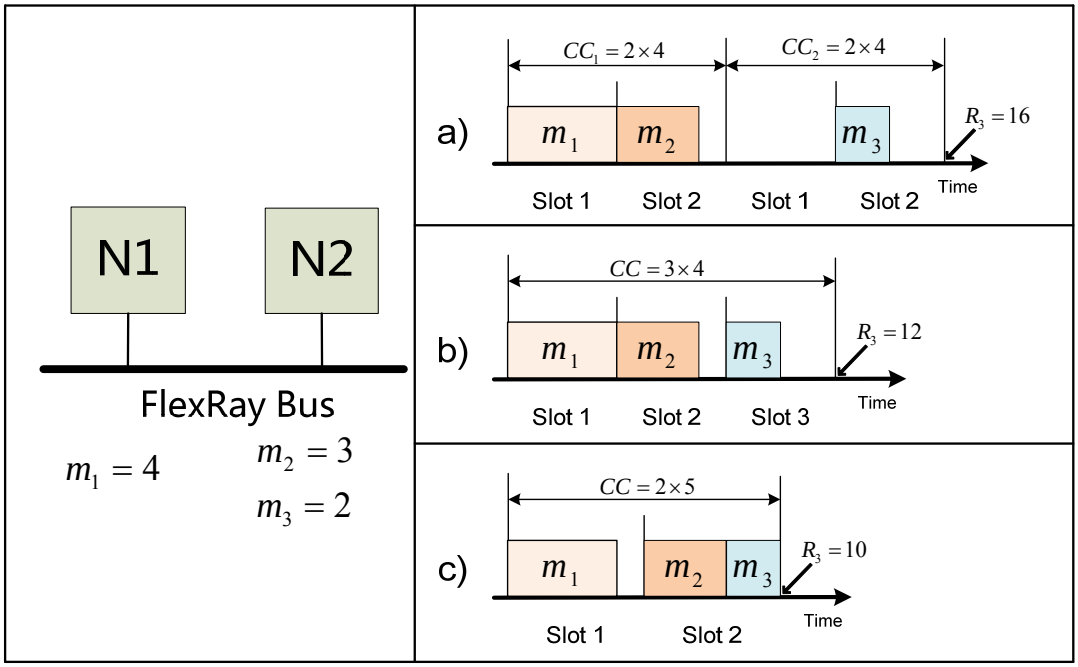


**Figure 5-1 Examples of the configuration of ST segment**

The Figure 5-1 gives a system with two nodes, N1 and N2, connected with a bus. Message $m_1$ with length of 4 is ready at N1 and messages $m_2$ and $m_3$ with length of 3 and 2 respectively are ready at N2. The messages' length and *SlotID* are also shown in figure. Please note that the examples in Figure 5-1 only shows the ST segment in the CC.

Firstly, let us compare the examples a) and b). As we can see in the example a) in Figure 5-1, there are 2 ST slots available. Each node gets 1 slot. Consequently in N2, $m_3$ have to share the same slot with $m_2$. This fact leads to the transmission of $m_3$ delayed to the second cycle because of the transmission of the higher-priority message $m_2$. While in example b), there are 3 ST slots available so that each message gets one slot. Consequently $m_3$ is able to send during the first cycle. From Figure 5-1, we know that this results the shorter response time of $m_3$, which is from 16 in a) to 12 in b).

Let's consider the example c) and a). c) has the same slot number as example a) does so that $m_2$ and $m_3$ in c) need to share the same slot. a) has the slot length of 4 while c) has longer slot length of 5. Even there are not enough slots for the messages in c), the change of slot length leads to the fact that one slot can accommodate both $m_2$ and $m_3$, so that $m_3$ still can send in the first cycle. As we can see from Figure 5-1, the response time of $m_3$ in c) is even shorter than b). The drawback of changing the slot length is that it delays the response time of the message $m_1$ and $m_2$.

From the examples illustrated above, it is clear to tell that the different number of ST slots and slot lengths could affect the system schedulability. Different configurations of the parameters lead to the totally different system capacities. According to FlexRay specifications, there are regulations about the configurable parameters' value. However, there always are optimal values of the parameters for a network. Therefore, it is very important to configure the system appropriately. The scheduler should not only can apply in FlexRay networks, but also can maximize the system capacity and increase system schedulability.

#### 5.1.1.2 Schedule Parameters

As introduced in Section 4.4, this thesis uses slot multiplexing to increase the resource utility rate. The transmission schedule of a ST message $m$ is represented by the 6-tuple vector:

$$\text{Schedule}_m = \{s_m, p_m, gNumberOfStaticSlots_m, b_m, r_m, gNumberOfCycle_m\} \quad (5.2)$$

Every ST message starts to transmit at the base slot ID $s_m$ in base cycle ID $b_m$. The transmissions repeat at a $p_m$-slots interval for $gNumberOfStaticSlots_m$ times and at a $r_m$-cycles interval for $gNumberOfCycles_m$ times. We need to set these unknown parameters in order to get the schedule of the ST messages.

Therefore, all the factors needed to set in ST segment are:

- The length of the ST slot $gdStaticSlot$;
- The number of ST slots $gNumberOfStaticSlots$;
- The schedulable constraints;
- The values of parameters in the schedule of ST message which can meet the schedulable constraints;
- The optimal allocation order of ST messages to reduce algorithm complexity;

The following sections will analysis and give the solution of these problems one by one.

### 5.1.2 Motivation for the Solutions

The motivation for the solutions is to define the known factors first and tries to use the known factors to define the unknown parameters and problems defined in section 5.1.1.

For a ST message $m$ needed to be scheduled, the known parameters introduced in Section 5.1.1 are the minimum inter-arrival time $T_m$ and the message length $MessageLengthST_m$. The following paragraphs will solve the unknown parameters, which are base cycle $b_m$, cycle repetition $r_m$, base slot $s_m$ and slot repetition $p_m$, by using the known factors. Furthermore, this section presents the optimization configurations and schedulable constraints.

#### 5.1.2.1 Length of ST slots *gdStaticSlot*

From the examples illustrate in Figure 5-1, we can tell that in order to have the shorter message response time we need to have either enough slots or enough length slots.

It is easy to tell by comparing the examples a) and c) that longer slot length shortens the response time of the share-slot low-priority messages. However, this configuration increases other messages' response time. The analysis shows a trade-off situation that it is hard to increase the response time of all messages at the same time. The improvement of the response time of a part of the messages leads to the degradation in another part of the messages. Therefore, we need to determine whether it is worth to do that. Alternatively, we could maintain the response time for most of the messages while sacrifice a few messages' response time.

As we can see in Figure 5-1, c) has the shortest response time of the message $m_3$ because c) uses the frame packing to transmit multiple messages in one slot. However, it is clear that frame packing significantly increases the system's computational complexity. Additionally, the response time of $m_3$ in c) does not significant decrease compared with the one in b). Therefore, this thesis does not consider the frame packing.

The configuration of the ST slot length *gdStaticSlot* must assure that the ST frame and the channel idle delimiter and any potential safety margin fit within the static slot under worst-case assumptions [35]. In order to fit at least any ST messages in the slot, the payload of the ST frame should equal to the longest ST message in a cluster. According to the FlexRay specifications appendix B.4.9, the length of ST frame *aFrameLengthStatic* has the following calculation formula:

$$\begin{aligned} aFrameLengthStatic[gdBit] \\ = gdTSSTransmitter[gdBit] + cdFSS[gdBit] \\ + 80[gdBit] + gPayloadLengthStatic\,[2 - byte - word] \\ * 20[gdBit] + cdFES[gdBit] \end{aligned} \tag{5.3}$$

If the lengths of the ST messages are known in advance, the payload of the ST frame *gPayloadLengthStatic* can be set to the longest ST message length:

$$gPayLoadLengthStatic = MessageLengthST_{\max} \tag{5.4}$$

The length of the ST slot is calculated as following:

$$\begin{aligned} gdStaticSlot[MT] &= 2 * gdActionPointOffset[MT] \\ &+ \boldsymbol{ceil}(\,(((aFrameLengthStatic[gdBit] \\ &+ cChannelIdleDelimiter[gdBit]) * gdBitMax[\mu s/gdBit] \\ &+ gdMinPropagationDelay[\mu s] \\ &+ gdMaxPropagationDelay[\mu s]))/(gdMacrotick[\mu s/MT] \\ &* (1 - cClockDeviationMax))\,) \end{aligned} \tag{5.5}$$

Function $ceil(x)$ returns the nearest integer greater than or equal to x. From formula (5.5) we can tell that the length of the ST slot $gdStaticSlot$ not only related with the ST frame length $aFrameLengthStatic$ but also related with some system parameters. The values of the system parameters in formula(5.3) and formula(5.5) are regulated in FlexRay specifications.

Various errors influence the system attainable precision of the clock synchronization that can be achieved. The system precision is mainly influenced by the network topology. The large and complex network will result in low synchronization accuracy [41]. $gdActionPointOffset$ has the range from 1 to 63 MT and must be greater than the attainable precision [11]. Thus, the simpler the network is, the smaller the $gdActionPointOffset$ value and ST slot size are. Moreover, the length of the ST segment is fixed, so the smaller slot size leads to more slots available in the CC.

According to FlexRay specifications, $cChannelIdleDelimiter$ is fixed to 11 gdBit. $gdBit$ and $gdBitMax$ have different values for different bus bit rates. The values of $gdBit$ are 0.4, 0.2, 0.1μs and the value of $gdBitMax$ are 0.4006μs, 0.2003μs and 0.10015μs for bus bit rate 2.5, 5 and 10Mbit/s respectively. $gdMinPropagationDelay$ and $gdMaxPropagationDelay$ both have the ranges from 0 to 2.5 μs. $gdMacrotick$ is from 1 to 6 μs. $cClockDeviationMax$ is fixed to 0.0015. $gdTSSTransmitter$ is from 3 to 15 gdBit. $cdFSS$ and $cdFES$ are fixed to 1 and 2 gdBit respectively. $gPayloadLengthStatic$ equals to $MessageLengthST_m^{\max}$, which is from 0 to 127 two-byte-words.

It is important to notice that the payload length of the ST frame $gPayloadLengthStatic$ increases only in two-byte-word unit, which equal to 20 gdBit in FlexRay specifications. However, the unit should unify to bit for the calculations, which equals to 1.25 gdBit.

#### 5.1.2.2 Number of ST slots *gNumberOfStaticSlots*

If the length of the ST segment and the length of the ST slot $gdStaticSlot$ are known, the number of ST slots $gNumberOfStaticSlots$ is known.

$$gNumberOfStaticSlot = \frac{ST_{bus}}{gdStaticSlot} \tag{5.6}$$

Equation (5.6) calculates the maximum number of ST slots available in the ST segment. Since this thesis does not consider the frame packing, the scenario c) is not possible. From the comparison of example a) and b), it is clear that enough number of ST slots could shorten the response time of the messages. However, the actual used slots number cannot excess the maximum available number of slots. The actual number of the used slots is not known until the generation of the scheduling table.

On one hand, the scheduler need to guarantee the messages have opportunities to transmit within their deadlines, on the other hand, it should avoid the slot over allocations to increase the system capacity. Therefore, the ST slots allocation should base on the minimum requirements of the messages. In other words, the number of ST slots allocated to a message should close to the required slots number by the messages.

#### 5.1.2.3 Schedulable Constraints

A ST message $m$ is scheduled if it satisfies the following constraints:

1. The message transmission finishes before the message's deadline $D_m$ for every message-arrival. The worst-case response time is smaller than the deadline $D_m$. The mathematic expression is:

$$\forall m \in M_{ST}, R_m \leq D_m \tag{5.7}$$

$M_{ST}$ is the ST message set in a cluster. $R_m$ is the worst-case response time of message m. This constraint suits for any schedulable systems, not only FlexRay system.

2. The FlexRay 2005 specifications regulate the transmission buffer between the host and the CC is non-queued buffers. A non-queued transmit buffer is a data storage structure in which new values overwrite former values [11]. To avoid buffer overwriting, the message transmission should finish before the next message-arrival. Therefore, in the worst-case scenario, message m's deadline $D_m$, namely the expiration time, equals to the minimum inter-arrival time $T_m$ of the messages. The mathematic expression is:

$$\forall m \in M_{ST}, D_m \leq T_m \tag{5.8}$$

#### 5.1.2.4 Base cycle $b_m$ and base slot $s_m$

$V$ denotes the maximum vector space of the schedule. The X-axis of $V$ is the ST slot ID with the maximum value of $gNumberOfStaticSlots$. The Y-axis of $V$ is the cycle ID with the maximum value of 63. $V$ represents the maximum system capacity of FlexRay protocol. Although the arrivals of ST messages are aperiodic, the transmission schedules of the ST messages are periodic. If the first transmission position $(s_m, b_m)$ for the message $m$ is

determined, the following transmission slots are known. It is clear that the position $(s_m, b_m)$ is very important to the message's schedule. Point $(s_m, b_m)$ is called reference point. This thesis defines a subset of $V$, denoted by $V_m$, to represent all possible positions of the reference point of message m, $(s_m, b_m) \in V_m$. In other words, the searching space of message m's schedule is $V_m$, $V_m \subseteq V$.

If the message is transmitted periodically, the transmission timeline, from this message's point of view, becomes the multiples of its period. Every small interval has the same duration as its transmission period and all are the same. The possible position for its reference point only has to choose from one of the spare slots in this interval. The following transmission instances are just as the repetition of the reference point in every interval. Therefore once the spare slot for the reference point is found, all of the following transmission instances can be guaranteed to successfully schedule. If we assume the schedule for message $m$ repeats every $p_m$-slots and every $r_m$-cycles, the possible values of the base slot $s_m$ are $[1, p_m]$ and the possible values of the base cycle $b_m$ are $[0, r_m - 1]$ because of the periodic transmission of messages.

Here we use the notation $S_m$ to represents the possible values of base slot $s_m$ which defined as follow:

$$S_m = [1, p_m], s_m \in S_m, s_m \in Z \tag{5.9}$$

And use the notation $CC_m$ to represents the possible values of base cycle $b_m$. It is defined as follow:

$$CC_m = [0, r_m - 1], b_m \in CC_m, b_m \in Z \tag{5.10}$$

Therefore the searching space of the reference point of messages $m$ is defined as:

$$V_m = (S_m, CC_m) \tag{5.11}$$

#### 5.1.2.5 Cycle repetition $r_m$ and slot repetition $p_m$

The analysis of base cycle $b_m$ and base slot $s_m$ show that these two values are directly related to the values $r_m$ and $p_m$. The previous section already set the searching space of $s_m$ and $b_m$. So this section will discuss the constraints of $r_m$ and $p_m$.

It is obvious that the worst-case response time $R_m$ is related to $r_m$ and $p_m$. $\forall m \in M_{ST}, \left\{ R_m \le \min\left( T_m \middle| m \in M_{N_s} \right) \right\}$ is the constraint of $R_m$ defined in Section 5.1.1. Furthermore, the design goal of the ST scheduler is allocating the slots to messages in the resource-saving way. To avoid ST slots wasting caused by over allocation, the cycle

repetition $r_m$ should be the maximum value that satisfies equation(5.7). Similarly, the slot repetition $p_m$ also should be the maximum value that satisfies equation(5.7). Furthermore, to ensure the messages' transmission patterns match with each other better and the concurrent transmissions, we deduce the constraint for $r_m$ and $p_m$ as followings:

$$r_m = \max\left\{2^n \middle| n \in [0,6], n \in \mathrm{Z}, R_m \le D_m\right\} \tag{5.12}$$

$$p_m = \max\left\{2^n \middle| n \in \left[0, \log_2\left(gdCycle / gdStaticSlot\right)\right], n \in \mathrm{Z}, R_m \le D_m\right\} \tag{5.13}$$

The slots that occupied by the schedule of message $m$ are represented as a 2-tuple vector set $buffer_m$:

$$\mathrm{buffer}_m = \left\{s_m + (m-1) \times p_m, b_m + (n-1) \times r_m\right\} \tag{5.14}$$

$$m \in [1, gNumberOfStaticSlots_m], m \in \mathrm{N}, \quad n \in [1, gNumberOfCycle_m], n \in \mathrm{N}$$

$gNumberOfStaticSlots_m$ is the number of slots per cycle message $m$ occupied. $gNumberOfCycle_m$ is the number of cycles per 64 cycles message $m$ occupied. After the message $m$ is scheduled, the slots occupied by this schedule needs to be eliminated from the reference point's searching space $V_{m+1}$ of the next message m+1.

#### 5.1.2.6 Detailed parameters based on the message types

Section 5.1.1 defined that the schedule of message m is represented by a 6-tuple vector $\left(s_m, p_m, gNumberOfStaticSlots_m, b_m, r_m, gNumberOfCycle_m\right)$. The messages need to be scheduled are divided into three types, based on the relationship between their deadlines and the CC length $gdCycle$.

The following paragraphs illustrate examples of three message-types' schedules and analyze the unknown parameters and schedulable constraints based on the transmission characteristics.

**$D_m > gdCycle$**

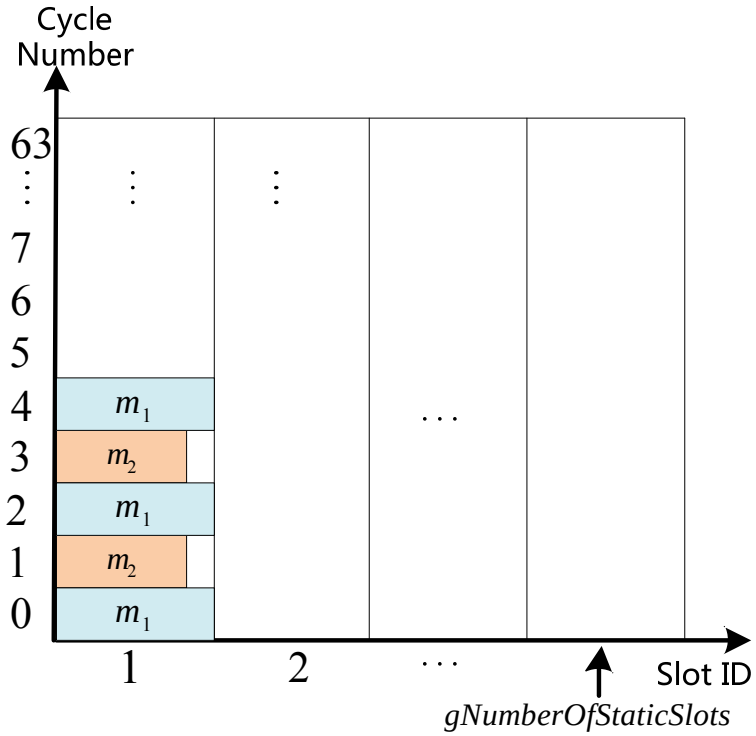


**Figure 5-2 Example message schedule for $D_m > gdCycle$**

If $D_m$ is longer than the CC length $gdCycle$, the message m only needs one ST slot per cycle and does not need to use all 64 cycles. Therefore, the slot repetition $p_m = gNumberOfStaticSlots$. Because of $D_m \le T_m$, the minimum inter-arrival period $T_m$ is also longer than the CC length. Thus, this message type uses the slot multiplexing to increase the shared-resource utility rate.

An example schedule is shown in Figure 5-2. $m_1$ and $m_2$ share the same ST slot ID 1. The cycle repetition $r_1$ and $r_2$ are 2 so that these two messages take turns to transmit. In general, this type of messages only gets one ST slot per cycle and transmits once per $r_m$ cycles.

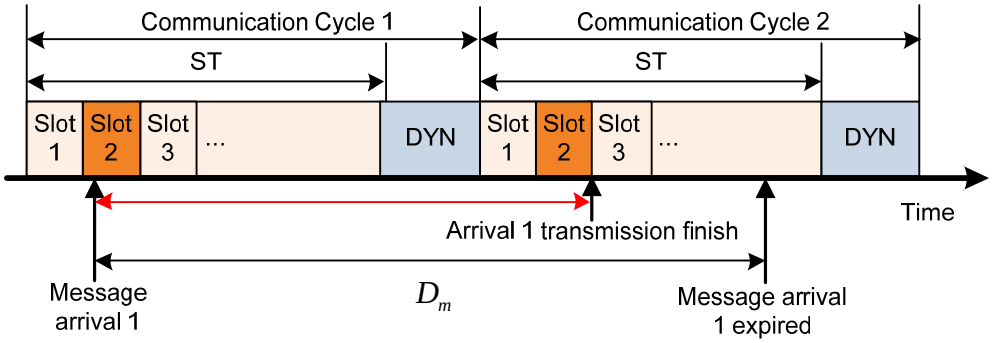


**Figure 5-3 Worst case response time of $D_m > gdCycle$**

As shown in Figure 5-3, the worst-case response time happens when message arrives just after the start of its allocated slot. The worst-case response time is the red duration in Figure 5-3. The mathematic expression for this type of worst case response time is:

$$R_m = r_m \times gdCycle \tag{5.15}$$

Equation(5.12) sets $r_m = \max\left\{2^n \middle| n \in [0,6], n \in Z, R_m \le D_m\right\}$. Therefore, the value of $r_m$ for this type of messages can be deduced from equation(5.12) and equation(5.15):

$$r_m = \max\left\{2^n \middle| n \in [0,6], n \in Z, r_m \le \frac{D_m}{gdCycle}\right\} \tag{5.16}$$

The number of cycles that message occupied in 64 cycles is:

$$gNumberOfCycle_m = \frac{64}{r_m} \tag{5.17}$$

In conclusion, the schedule for the message set$\{m|D_m > gdCycle\}$ has the characteristic:

- $gNumberOfStaticSlots_m = 1$
- $gNumberOfCycle_m = \frac{64}{r_m}$
- $r_m = max\left\{2^n \middle| n \in [0,6], n \in \mathbb{Z}, r_m \le \frac{D_m}{gdCycle}\right\}$
- $p_m = gNumberOfStaticSlots$

$\boldsymbol{D_m = gdCycle}$

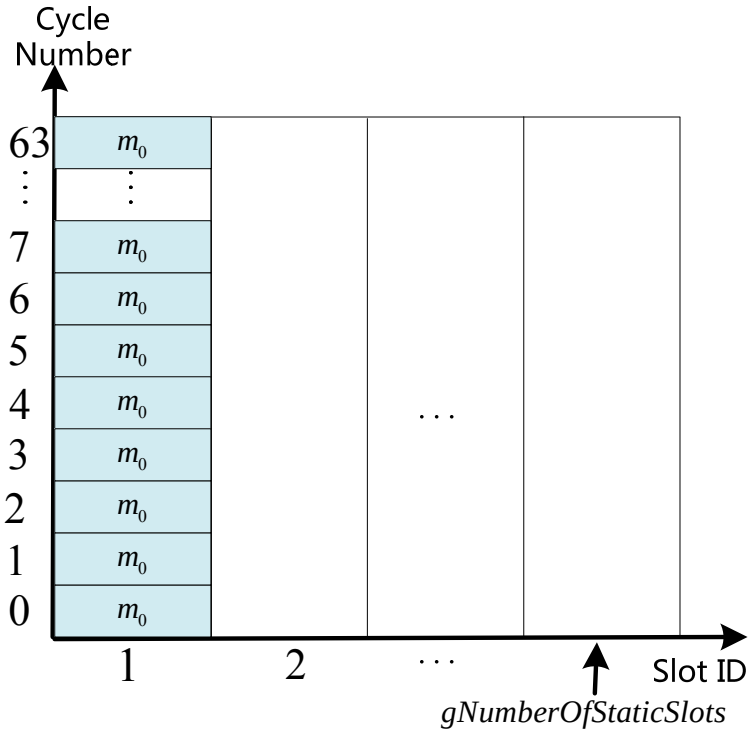


**Figure 5-4 Example message schedule for $\boldsymbol{D_m = gdCycle}$**

Messages that $D_m$ equals to $gdCycl$ are considered as a special case of $\{m|D_m > gdCycle\}$ with $r_m = 1$. The message deadline$D_m$equals to the CC length $gdCycle$. An example schedule is shown in Figure 5-4. $m_0$ uses ST slot ID 1. The cycle repetition $r_0$ is 1 because $D_m = gdCycle$. In general, this type of messages only gets one ST slot per cycle and transmits in every cycle. Therefore $r_m = 1$ and $gNumberOfStaticSlots_m = 1$.

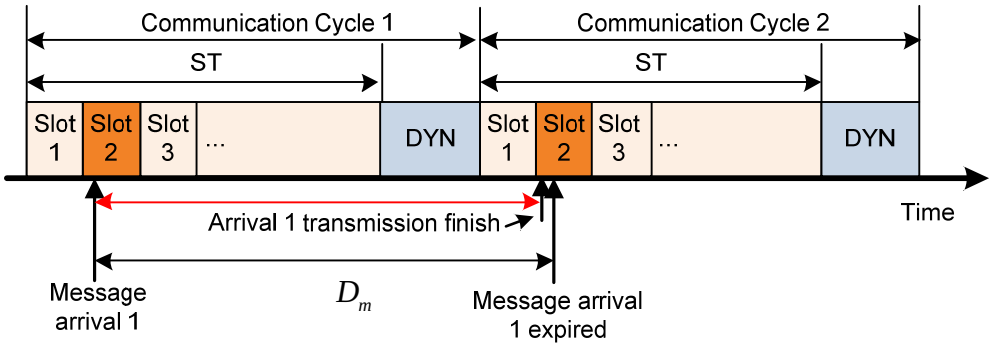


**Figure 5-5 Worst case response time of $\boldsymbol{D_m = gdCycle}$**

In conclusion, the schedule for the message set $\{m|D_m = gdCycle\}$has the characteristic:

- $gNumberOfStaticSlots_m = 1$

- $gNumberOfCycle_m = 64$
- $r_m = 1$
- $p_m = gNumberOfStaticSlots$

**$D_m < gdCycle$**

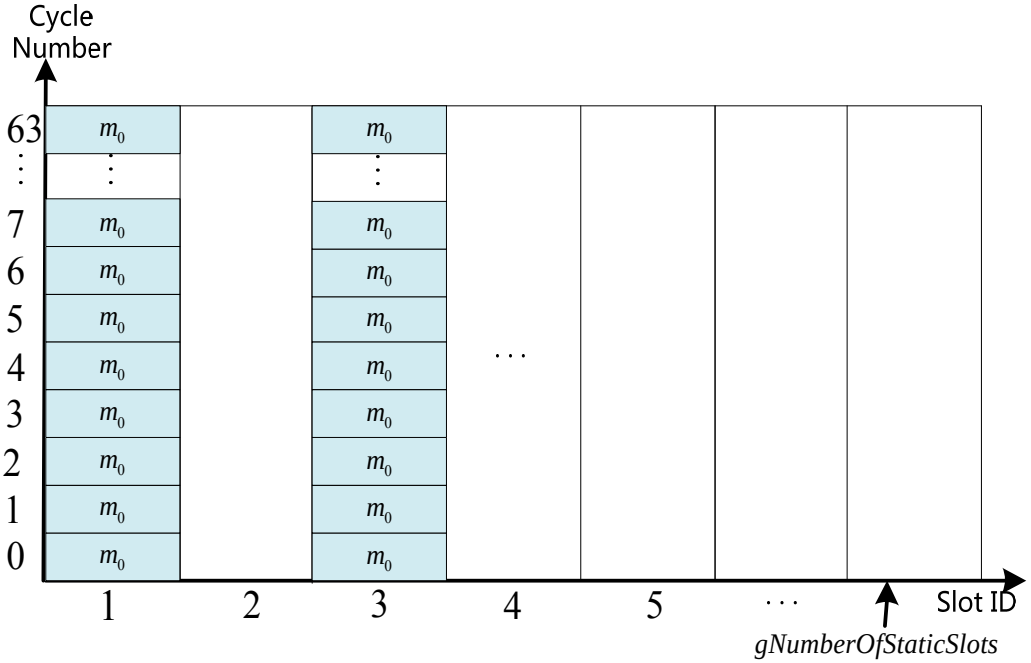


**Figure 5-6 Example message schedule for $D_m < gdCycle$**

The messages that $D_m$ shorter than $gdCycle$ has to assign multiple ST slots per cycle to be able to transmit every message-arrival within the deadline. The slot repetition $p_m$ has the constraint $p_m = 2^n, n \in [0,9], n \in Z$ in order to better match with other messages' schedules. An example schedule is shown in Figure 5-6. $m_0$ is allocated ST slot ID 1 and 3. The cycle repetition $r_0$ is 1 because $D_m < gdCycle$. In general, this type of messages gets multiple ST slots per cycle and transmits in every cycle. Therefore $r_m = 1$.

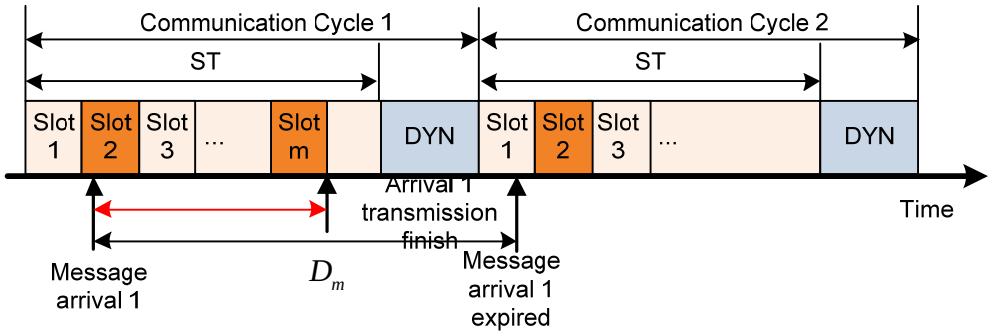


**Figure 5-7 Worst case response time of $D_m < gdCycle$**

The worst case response time is the red duration shown in Figure 5-7. The mathematic expression for this type of worst case response time is:

$$R_m = p_m \times gdStaticSlot \tag{5.18}$$

Equation(5.13) sets $p_m = max\left\{2^n \middle| n \in \left[0, log_2\left(\frac{gdCycle}{gdStaticSlot}\right)\right], n \in \mathbb{Z}, R_m \leq D_m\right\}$. Therefore, the value $p_m$ of this type of messages can be deduced from equation(5.13) and equation(5.18):

$$p_m = max\left\{2^n \middle| n \in \left[0, log_2\left(\frac{gdCycle}{gdStaticSlot}\right)\right], n \in \mathbb{Z}, p_m \leq \frac{D_m}{gdStaticSlot}\right\} \tag{5.19}$$

Because $D_m < gdCycle$, the number of cycles assigned to message m is the maximum number of cycle counter 64:

$$gNumberOfCycle_m = 64 \tag{5.20}$$

The number of ST slots message $m$ occupied per cycle equals to the maximum number of transmissions during one cycle:

$$gNumberOfStaticSlots_m = \left\lceil \frac{gdCycle}{p_m \times gdStaticSlot} \right\rceil \tag{5.21}$$

In conclusion, the schedule for the message set$\{m|D_m < gdCycle\}$ has the characteristic:

- $gNumberOfStaticSlots_m = \left\lceil \frac{gdCycle}{p_m \times gdStaticSlot} \right\rceil$
- $gNumberOfCycle_m = 64$
- $r_m = 1$
- $p_m = max\left\{2^n \middle| n \in [0, log_2(gdCycle/gdStaticSlot)], n \in \mathbb{Z}, p_m \leq \frac{D_m}{gdStaticSlot}\right\}$

#### 5.1.2.7 Allocation order

The preceding discussion determined the ST slot length, the number of ST slot, and the constraints of $r_m$ and $b_m$. The next question will be: which message should be scheduled first so that the system might have a better schedulability?

In ST segment scheduling, all of the ST messages should meet their deadlines in order to have a schedulable system. The scheduling order of the ST messages follows the principle that the least flexible message is allocated first. In other words, the messages need to be scheduled first are the ones that have the least feasible schedules, namely the flexibility of the message. The message's flexibility is represented by the searching-space size of the reference point $V_m = (S_m, CC_m)$. $S_m = [0, p_m - 1]$ and $CC_m = [0, r_m - 1]$. Therefore the size is $p_m \times r_m$.

Furthermore, the 2D area is less flexible than the 1D area. Since the transmissions of the messages in set $\{m | D_m < gdCycle\}$ form a 2D area, they are less flexible than other types of messages. Therefore, the messages in set $\{m | D_m < gdCycle\}$ are scheduled first, which have smaller value of $p_m$. In conclusion, the allocation order follows the ascending order of the values $p_m$ and then in ascending order of values $r_m$.

### 5.1.3 ST Segment Scheduling Algorithm

The motivations to solve the problems are already presented in section 5.1.1. Here we provide the ST scheduling algorithm for a known network in Algorithm 1 based on these solutions.

**Input:**

- Bus bit rate $\boldsymbol{bus\_speed}$
- ST message set $\boldsymbol{M_{ST}}$, $\boldsymbol{M_{ST}}$ is the ST message set waiting to send in a cluster, $m = (T_m, MessageLengthST_m), m \in M_{ST}$

**Output:**

$m \times 6$ **Scheduling matrix** $\textbf{SimpleScheduler}_{\textbf{ST}}$ **:**

$$\begin{bmatrix} s_1 & p_1 & gNumberOfStaticSlots_1 & b_1 & r_1 & gNumberOfCycle_1 \\ s_2 & p_2 & gNumberOfStaticSlots_2 & b_2 & r_2 & gNumberOfCycle_2 \\ s_3 & p_3 & gNumberOfStaticSlots_3 & b_3 & r_3 & gNumberOfCycle_3 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ s_m & p_m & gNumberOfStaticSlots_m & b_m & r_m & gNumberOfCycle_m \end{bmatrix}$$

Simple FlexRay ST scheduling algorithm

for $\forall m \in M_{ST}$ do // assign values to different parameters based on message types

$gdStaticSlot = f\left(MessageLengthST_{\max}, \text{ number of active stars in the topology}\right)$

$$gNumberOfStaticSlot = \frac{ST_{bus}}{gdStaticSlot}$$

$$D_m = T_m$$

if $D_m > 30000$

$$D_m = 30000$$

end if

if $D_m < gdCycle$

$$r_m = 1$$

$$p_m = \max\left\{2^n \middle| n \in \left[0, \log_2\left(gdCycle / gdStaticSlot\right)\right], n \in Z, p_m \le \frac{D_m}{gdStaticSlot}\right\}$$

$$gNumberOfStaticSlots_m = \left\lceil \frac{gdCycle}{p_m \times gdStaticSlot} \right\rceil$$

$$gNumberOfCycle_m = 64$$

else

$$p_m = gNumberOfStaticSlots$$

$$gNumberOfStaticSlots_m = 1$$

if $D_m > gdCycle$

$$r_m = \max\left\{2^n \middle| n \in [0, 6], n \in Z, r_m \le \frac{D_m}{gdCycle}\right\}$$

$$gNumberOfCycle_m = \frac{64}{r_m}$$

else // message type $D_m = gdCycle$

$r_m = 1$

$gNumberOfCycle_m = 64$

end if

end if

end for

$$gNumberOfStaticSlots_{\min} = \max\left(2, \left\lceil \sum_{\forall m \in M_{ST}} \frac{gNumberOfStaticSlots_m}{r_m} \right\rceil\right)$$

if $gNumberOfStaticSlots_{\min} > gNumberOfStaticSlots$

output non-schedulable, exit //system non-schedulable

end if

Sort the messages $\forall m \in M_{ST}$ in ascending order of the values $p_m$ and then in ascending order of values $r_m$ store them in the message list $L_{ST}$

$p_{min} = \min(p_m)$ //sort finish

$V_{use} = (S_{use}, CC_{use}) = (0,0)$

for $m \in L_{ST}$ do

Slot ID set $S_m = [1, p_m]$

CC set $CC_m = [0, r_m - 1]$

$V_m = (S_m, CC_m)$

$V_m = V_m - V_{use}$ // update the available space of the reference point

if $V_m$ is empty set

output system non-schedulable, exit

end if

for $(s_m, b_m) \in V_m$

for $\forall m \in [1, gNumberOfStaticSlots_m], m \in \mathrm{N}$

for $\forall n \in [1, gNumberOfCycle_m], n \in \mathrm{N}$

$\mathrm{buffer}_m = \{s_m + (m-1) \times p_m, b_m + (n-1) \times r_m\}$

end for

end for

$V_{use} = V_{use} + buffer_m$

$gNumberOfStaticSlots_{new} = s_m + (gNumberOfStaticSlots_m - 1) \times p_m$
//update used maximum ID ST slots

if $gNumberOfStaticSlots_{new} > gNumberOfStaticSlots_{use}$

$gNumberOfStaticSlots_{use} = gNumberOfStaticSlots_{new}$

end if

take m out of $L_{ST}$, update $L_{ST}$

if $L_{ST}$ is empty

output $\mathbf{SimpleScheduler_{ST}}$ matrix, exit //system schedulable

end if

continue with next $m \in L_{ST}$

end for

end for // continue with next value of $gdCycle$

**Algorithm 1 Pseudocode for Simple FlexRay ST scheduling algorithm**

In the worst-case scenario, the complexity of the algorithm is $O(n)$. $n$ is the number of ST messages.

## 5.2 Scheduler Design for Simple FlexRay DYN Segment

Before analysis of this section, we would like to present some notations which are uses later.

- $DYN_{bus}$ is the length of the DYN segment;
- *gdMinislot* is the duration of a Minislot;

- $MessageLengthDYN_m$ is the number of bits constituting the dynamic message $m$ in the cluster;
- $pLatestTx_m$ is the number of the last Minislot in which a frame transmission can start in the DYN segment;
- $R_m$ is the worst-case response time of the DYN message m;

### 5.2.1 Problem definition

The analysis of ST segment shows that the ST segment communication is designed for the periodic tasks and messages that have a minimum inter-arrival time. These tasks and messages are time-triggered, which require highly predictability and guaranteed latency. However, automotive communication not only has the periodic tasks and messages but also has the aperiodic ones. The aperiodic tasks and messages do not have a maximum inter-arrival bit rate. It is necessary to have another media access scheme to adapt the transmission and improve the network efficiency. The media access mechanism in DYN segment of the FlexRay CC uses event-triggered FTDMA. It is the Minislot-based scheme to adapt the flexible transmissions. The messages transmitted in DYN segment are called DYN messages.

There are two differences between ST segment and DYN segment communications. Firstly, the length of the ST slot is fixed in a schedule while the length of the DYN slot varies based on the size of the frame transmitted in that slot. Secondly, the transmissions of other messages do not affect the response time of the ST messages since every ST message is allocated a ST slot to transmit. On the contrary, the transmission of other messages affect the response time of the DYN message[9] since the DYN messages do not have a fixed slot for transmission. The DYN message only has a FrameID that can be used to arbitrate the shared resource, in this case is the FlexRay bus.

The aperiodic tasks or messages arrive unpredictably. They do not have an average inter-arrival time. Therefore, the known parameters of the DYN message $m$ are message length $MessageLength_m$ and the deadlines $D_m$. Thus, a DYN message $m$ can be represented by 2 parameters:

$$m = \{D_m, MessageLengthDYN_m\} \tag{5.22}$$

Since the DYN message is the event-triggered transmission, there is no fixed scheduling table for the message in advance of the system start. So we need to find another method to

[9] The discussion of DYN segment don't consider the frame packing problem as the consistent policy of the ST segment. Therefore the DYN message can be seen as DYN frame.

evaluate the system schedulability which is different from the method to see whether the scheduler can generate a valid scheduling table. Because the unpredictability of the DYN message arriving, to guarantee a system is schedulable, the best way is to ensure the system is schedulable in the worst-case scenario, even this method is a little pessimistic. To see if the communication network can complete the transmission of every DYN message within its deadline, even in the worst case, is the constraint used to evaluate the system schedulability. In other words, the worst-case response time of the message should be shorter than its deadline:

$$\forall m \in M_{DYN}, R_m \leq D_m \tag{5.23}$$

$M_{DYN}$ is the DYN message set in a cluster. $R_m$ is the worst-case response time of message m.

Just like the analysis in Section 4.5, the worst-case scenario of a DYN message transmission happens when the higher-priority messages and the lower frame ID messages are all queuing for transmission. We assume all the DYN messages are ready for transmission. We calculate the worst-case response time $R_m$ of every DYN message to see whether it is smaller than the deadline. The calculation of the worst-case response time is presented in Section 4.5. Thus, the message is scheduled if the scheduler can guarantee the response time at least equals to the deadline.

**Configurable Parameters**

Let us consider the examples first and discuss the different message response times and the related parameters.

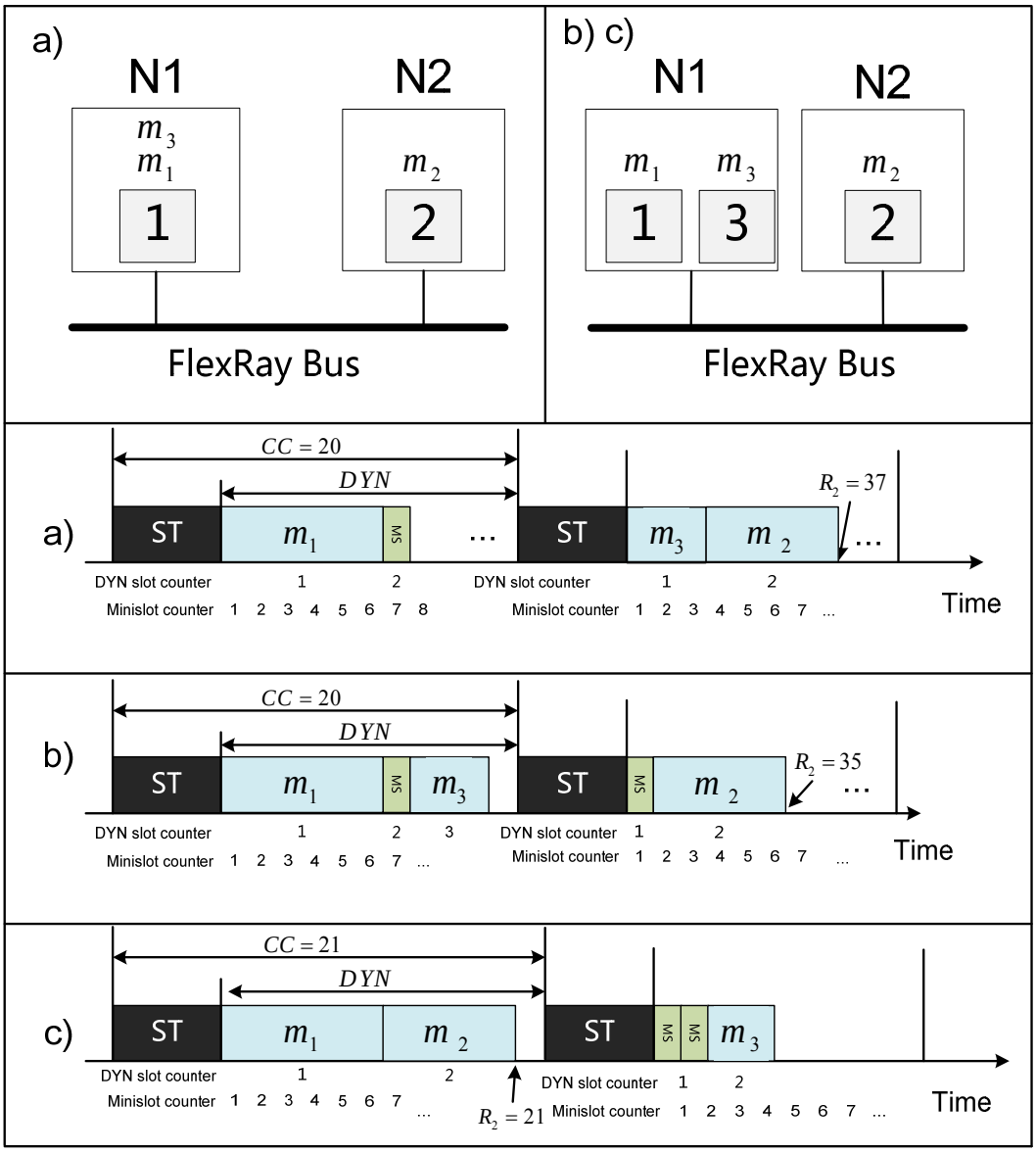


**Figure 5-8 Examples of the configuration of DYN segment**

The examples of the different configurations in DYN segment are shown in Figure 5-8. The network of the examples consists of two nodes N1 and N2. They are connected by a bus. The messages transmitted on the nodes are shown in figure.

Let us compare example a) and b). These two examples have the same DYN segment length but different Frame ID allocations. As we can see in Figure 5-8, $m_1$ and $m_3$ are messages in node N1. They share the same Frame ID=1 in example a) while use the different Frame ID 1 and 3 in examples b) and c). In example a), $m_3$ is the lowest priority among all shared Frame ID messages. If they both ready at the same time, the transmission of $m_3$ is delayed one cycle by the transmission of higher-priority $m_1$. This situation is similar to ST segment that different ST messages share the same Slot ID. We use Frame ID instead of Slot

ID in DYN segment. In example a), the delay of $m_3$ suspends the transmission of $m_2$. In example b), $m_3$ has its own Frame ID so that it has the right to arbitrate the bus access and can transmit in the first cycle. This reduces the response time of $m_2$ since it does not need to wait for $m_3$.

Let us consider example b) and c). These two examples have the same Frame ID allocation but different DYN lengths. The response time of $m_2$ is shorter in c) than it in b). Because b) does not have enough time in DYN segment after the transmission of $m_1$ to accommodate $m_2$ while c) has enough time in DYN segment to accommodates $m_2$ . $m_3$ still cannot fit in the first cycle in c) even it has its own Frame ID. However, at least we can say the longer DYN length gives more possibilities to the messages to reduce the response time.

The results are very similar to ST segment. The capacity of DYN segment is strongly affected by the values of the DYN length and the frame ID. Furthermore, the transmission order and response time of message $m_1$ does not change in example a), b) and c). Since it has the highest priority among all of the messages, it cannot be affected by the other messages' transmission. However, this does not indicate that the higher-priority messages have shorter response time. Message $m_3$ and $m_2$ in example b) and c) have the same Frame ID, however, the response time of $m_2$ is not always shorter than $m_3$.The reason is the size of the message also affect the transmission order and the response time.

The problems in the DYN segment scheduler design are:

- Setting the order of the DYN messages' Frame ID assignment to achieve the best system schedulability;
- Calculation $\boldsymbol{pLatestTx}$;
- Calculating the message worst-case response time $\boldsymbol{R_m}$ ;
- Setting the length of the DYN segment $\boldsymbol{DYN_{bus}}$ ;

## 5.2.2 Motivation for the Solution

### 5.2.2.1 Assigning the Frame ID

The worst-case response time analysis in Section 4.5 gives the conclusion that the messages in sets $hp(m)$, $lf(m)$ and $ms(m)$ may cause the delay of the message m. Delay reduction optimizes the DYN messages' transmission. The illustration in Figure 5-8 leads to the conclusion that shared $FrameID$ causes large delay. Therefore, allocating a Frame ID to each message in the cluster reduce the worst-case response time of the lower priority messages.

The value of Frame ID corresponds to the priority. The next question is how to allocate the priority to the DYN messages. The discussions of previous sections show that the message size and the priority both could affect the message's response time. We can observe from Figure 5-8 that the response time of a larger size message that has low priority is expected very long. The long response time is not a problem as long as the message can meet the deadline. Frame ID '1' denotes the highest priority in this thesis.

Therefore, we allocate the FrameID to messages from 1, namely the highest priority, follows the descending order of the value $\frac{MessageLengthDYN_m}{D_m}$. If the messages have the same value $\frac{MessageLengthDYN_m}{D_m}$, the Frame-ID assignment is in descending order of the value $MessageLengthDYN_m$.

#### 5.2.2.2 Calculation $pLatestTx$

Based on the discussion of the message response time, we can tell that for a given DYN message sets, if the $FrameID_m$ and the message sizes $MessageLength_m$ are known, the worst-case response time $R_m$ only varies with the DYN segment length $DYN_{bus}$, namely the CC length $gdCycle$ and ST segment length $ST_{bus}$.

To calculate the worst-case response time $R_m$, the parameter $pLatestTx_m$ needs to be determined first. Section 4.5.3 already introduced the concept of $pLatestTx_m$ that the parameter $pLatestTx_m$ shows the number of the last Minislot in which a frame transmission can start in the DYN segment [11]. A DYN message cannot be transmitted at the instant that the value of Minislot counter is smaller than $pLatestTx_m$.The value of $pLatestTx$ depends on the size of the DYN frames[10] and the size of the Minislot. According to the FlexRay specification, the approximate calculation formula of $pLatestTx$ is:

$$pLatestTx\ [Minislot] \leq gNumberOfMinislots - aFrameLengthDynamic\ [\mu s] / gdMinislot\ [\mu s] \quad (5.24)$$

The value of $pLatestTx$ mainly related with the number of Minislots in the DYN segment $gNumberOfMinislots$ and the DYN frame length. This thesis ignores the length of network idle time (NIT) and symbol window (SYM). The number of Minislots $gNumberOfMinislots$ in DYN segment can be approximately calculated as following:

[10] The calculation simplified the frame length into the message length.

$$gNumberOfMinislots \approx DYN_{bus}/gdMinislot[\mu s] \tag{5.25}$$

FlexRay specifications define the DYN frame length $aFrameLengthDynamic$ as following:

$$\begin{aligned} &aFrameLengthDynamic\ [\mu s] \\ &\quad = (\ gdTSSTransmitter[gdBit] + 83\ gdBit \\ &\quad + \ MessageLengthDYN\ [bit] * 1.25\ gdBit) * gdBit[\mu s] \end{aligned} \tag{5.26}$$

As we can see in formula(5.24), $pLatestTx$ relates with other system parameters. According to FlexRay specification, the value of the system parameter $gdMinislot$ is from 2 to 63 MT. $MessageLengthDYN_m$ is from 0 to 127 two-byte-words.

#### 5.2.2.3 Calculate the message worst-case response time $R_m$

The discussion in Section 4.5 has already presented the detail of the heuristic calculation formula of worst-case response time:

$$\begin{aligned} R_m = w_m + C_m \quad &= (\sigma_m + nT_{bus} + w'_m) + (aFrameLengthDynamic_m/bus\_speed) \\ &= \\ &\big(gdCycle - (ST_{bus} + (FrameID_m - 1) * gdMinislot)\big) + \\ &BusCycles_m\big(lf(m,t)\big) * gdCycle + (FrameID_m - 1) * gdMinislot + \\ &\big(ST_{bus} + pLatestTx_m * gdMinislot\big) + \\ &(aFrameLengthDynamic * gdBit/bus\_speed) \end{aligned} \tag{5.27}$$

In this thesis, there are no shared Frame ID DYN messages, namely the message set $hp(m,t)$ is empty. Thus, the delay $BusCycles_m\big(hp(m,t)\big) = 0$. The solution of $BusCycles_m\big(lf(m,t)\big)$ transforms into the *1DBP problem.* In this case, the message set $lf(m,t)$ represents the items, the DYN segments $DYN_{bus}$ are bins, and the minimum capacity required to fill a bin is $pLatestT_m \times gdMinislot$.

#### 5.2.2.4 Length of the DYN segment $DYN_{bus}$

The CC consists of ST segment and DYN segment in this thesis. The length of DYN segment $DYN_{bus}$ can be calculated as following:

$$DYN_{bus} = gdCycle - ST_{bus} \tag{5.28}$$

According to FlexRay specification, the range of $gdCycle$ is from 10 μs to 16000 μs. Moreover, the increment of the cycle length aims to accommodate different sizes of messages. The payload of the frame can increase only in two-byte-word unit, which equals to 20 $gdBit$

in FlexRay protocol. Therefore, the algorithm should increase the cycle length in the same unit as the increment of the FlexRay frame payload.

### 5.2.3 DYN Segment Scheduling Algorithm

The motivations of the simple DYN scheduler design are presented in the previous section. Based on the solutions, we provide the simple DYN scheduling algorithm for a known network in Algorithm 2.

**Input:**

- Bus bit rate $\boldsymbol{bus\_speed}$
- The DYN message set$\boldsymbol{M_{DYN}}$, $\boldsymbol{M_{DYN}}$ is maximum DYN message set waiting to send in a cluster, $m = \left(D_m, MessageLengthDYN_m\right), m \in M_{DYN}$

**Output:**

System schedulable or non-schedulable

---

Simple FlexRay DYN scheduling algorithm

---

Sort the DYN messages $\forall m \in M_{DYN}$ in descending order of the value $\frac{MessageLength_m}{D_m}$ and then in descending order of the value $MessageLength_m$, store them in the message list $L_{DYN}$

Assign the Frame ID to the DYN messages with the order in list $L_{DYN}$ from 1

// sort the messages and assign the Frame IDs

for $gdCycle = 10\ to\ 16000\mu s$ step $20 \times gdBit$ μs

simple FlexRay ST scheduling algorithm

$DYN_{bus} = gdCycle - ST_{bus}$

$gNumberOfMinislots = DYN_{bus}/gdMinislot$

for $m \in L_{DYN}$

$aFrameLengthDynamic\ (bit) = gdTSSTransmitter + 83 + MessageLengthDYN * 1.25$

$pLatestTx_m = gNumberOfMinislots - (aFrameLengthDynamic * gdbit)/gdMinislot$

---

FFD bin packing algorithm calculates the delay $BusCycles_m(lf(m,t), ms(m,t))$

$$R_m = \big(gdCycle - (ST_{bus} + (FrameID_m - 1) * gdMinislot)\big) + BusCycles_m\big(lf(m,t)\big) * gdCycle + (FrameID_m - 1) * gdMinislot + (ST_{bus} + pLatestTx_m * gdMinislot) + (aFrameLengthDynamic * gdBit/bus\_speed)$$

if $R_m \leq D_m$

take $m$ out of list $L_{DYN}$

if $L_{DYN}$ is not empty

continue with next message in list $L_{DYN}$

else

output schedulable, exit

end if

else

next value of $gdCycle$

end if

end for

end for

output non-schedulable, exit //system non-schedulable

**Algorithm 2 Pseudocode for Simple FlexRay DYN scheduling algorithm**

The complexity of the algorithm is $O(n)$. $n$ is the number of DYN messages.

## 5.3 Conclusion

This chapter focuses on the ST and DYN segment scheduler-design in simple FlexRay network. We first analyzed schedulable constraints and configurable parameters of messages and decided the way to determine the value of these constraints. In the analysis of the ST segment, we classified the ST messages into three types and gave the values of the parameters in detail for each of these types. Afterwards, we decided the allocation order of the slot ID. In the analysis of the DYN segment, we focused on the allocation order of the frame ID and the calculation of the worst-case response time. In the calculation, the key parameter $pLatestTx$ is specially discussed. In the end of DYN segment discussion, we presented the scheduling algorithm.

# 6

# Scheduler Design for Switched FlexRay Networks

The simple FlexRay schedulers design has already presented in Chapter 5. This chapter introduces a new FlexRay network that includes a new component called ‘FlexRay switch’. It has the similar conceptual features like the Ethernet network switch.

The switch can isolate or combine different branches into separate clusters based on communication needs. Each cluster can work parallel. So each cluster has different schedules to maximize the use of the shared resource. During different slots, the clusters consist of different nodes. This is done by reforming clusters in each slot. In other words, ECUs belong to the different cluster during different slots. The switch FlexRay network is no longer communicates in broadcast mode. The clustering step has a crucial influence to increase the slot utility rate.

This chapter we first will introduce the concept of switch FlexRay network. Then we will present scheduling algorithms for ST and DYN segment in switch FlexRay network respectively.

## 6.1 Concept of Switched FlexRay Network

According to FlexRay specifications, the existing component active star is a central component in the network. It is the good starting point to realize the switching function. Therefore the network replaces the active star with a switch in order to realize multiple paths of data synchronized transmission.

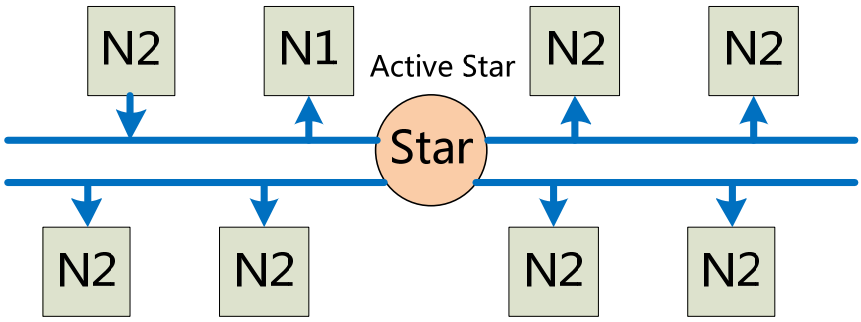


**Figure 6-1 Simple FlexRay network with 4 nodes connected by active star**

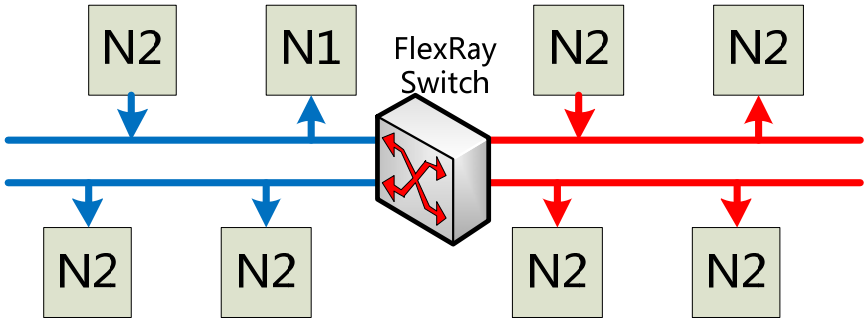


**Figure 6-2 Switched FlexRay network with 4 ports[11]**

Figure 6-2 shows a simple FlexRay network included eight nodes that are connected by an active star while Figure 6-2 shows a network includes a FlexRay switch that the eight nodes are connected by the switch. The blue and red lines represent the data flows. As we can see from two figures, the simple FlexRay network only has one data flow while switch FlexRay can has two independent data flows. The active star allows only one source node at a time while FlexRay switch allows multiple source nodes. It is clear to tell that switch FlexRay network significantly increase the system bandwidth.

FlexRay switch cannot use the packet switching since it is used in the real time system. Predefined scheduling matrix stored in the switch decides the operation order of the ports.

[11] The concept of port basiclly is equivalent to branch in this thesis. But port is more emphasis on the physical interface while branch is more emphasis on the whole system behind the interface.

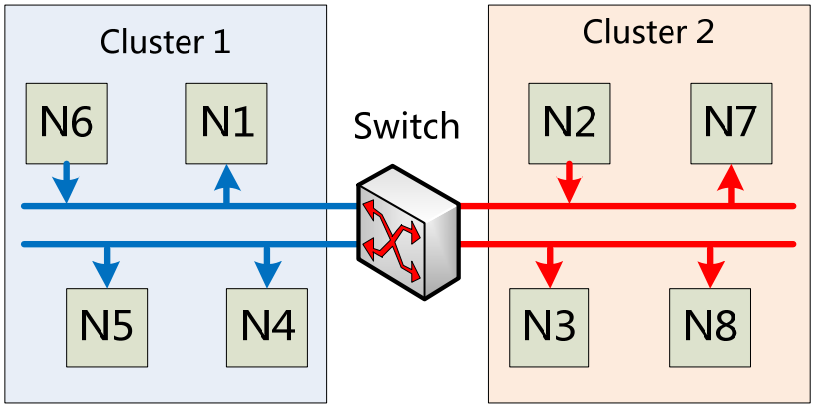


**Figure 6-3 Two clusters separately communicate during one slot**

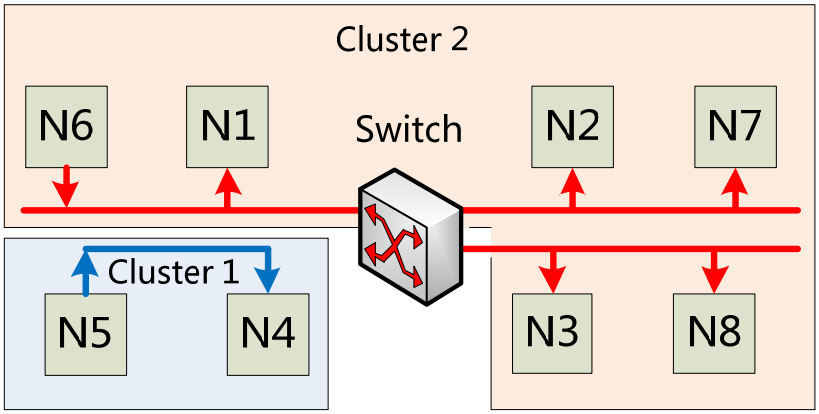


**Figure 6-4 One port close during one slot**

Figure 6-3 and Figure 6-4 are two different clustering examples of the same switched network. The switch has different branches which can isolate or group different branches into different clusters, namely the clusters, and to send data simultaneously based on the communication needs. The topology of the cluster could be bus topology, star topology or mixed topology. The maximum number of clusters depends on the number of switch ports. There is only one source node at a time in any cluster.

After the introduction of switch FlexRay network, the following sections will discuss the related problems and the answers concerning the scheduler design for ST segment and DYN segment.

## 6.2 Scheduler Design for Switched FlexRay ST Segment

Before the discussion we would like to present some notations which are used later.

- **$Matrix_m$** is the matrix of message m indicating the message transmission path;
- **$Matrix_{CycleNumber,SlotID}$ is the switching matrix for a specific slot ID and cycle number;**

- $Port_{available}$ is the set of unoccupied ports in the switch;
- $Port_{m}$ is the set of source and destination ports of message m;
- $Port_{use}$ is the These three constraints indicate that if a switch port functions as the disconnected port, the source port or the destination port involving any communication during a slot it is seen as an occupied port. The occupied port set is notated
- $Port_{switch}$ is the set of the ports for the FlexRay switch;

### 6.2.1 Problem definition

The best way to solve a problem is to analyze the differences between the existing problems and the new problems and get the solutions from the existing solutions. After the analysis of simple FlexRay ST scheduler design, problems and solutions of this scheduler are clear. This section deals with the ST segment scheduler design in switch FlexRay network. Therefore, the best starting point of the new scheduler is to consider the differences between the working principle of switch FlexRay network and the simple FlexRay network. Then start to find solutions for these differences and design the new scheduler.

The main difference of these two networks is the FlexRay switch. The simple FlexRay network supports one data source sending at a time. Switched FlexRay network includes a FlexRay switch that enables multiple data sources to send data simultaneously.

Every message has a message transmission path that consists of one source node and one or more destination nodes. Each node is associated with a fixed switch port. From the message's point of view, the switching path is a set of source and destination nodes. From the switch's point of view, the switching path is a set of source and destination ports. The nodes connected with the same port can be seen as one simple FlexRay branch.

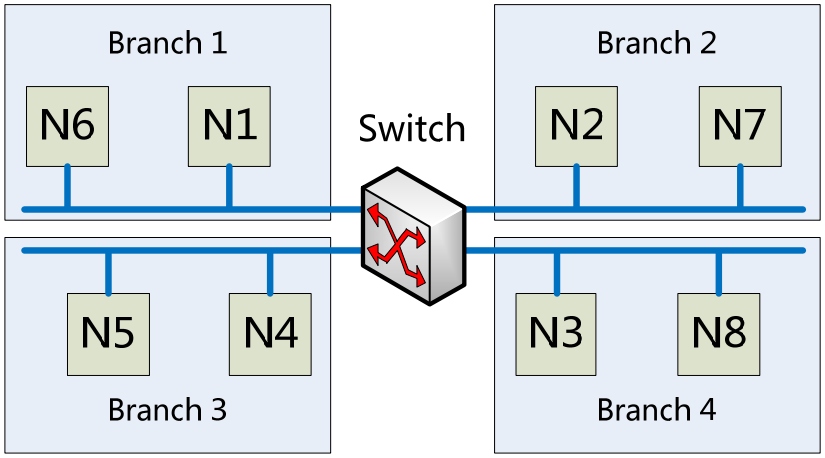


**Figure 6-5 Concept of branch**

The concept of the branch is shown in Figure 6-5. As it is seen from the figure, the network is divided into four branches since the switch has four ports. The nodes N4 and N5 are connected with the same port in the switch. These two nodes form a simple FlexRay branch. It is clear that the maximum number of the branches equals to the number of the switch ports.

The ST messages transmitted in the switch FlexRay network are called switched ST message. The switched ST messages can be further divided into two types: the one transmits through the switch is called the after-switch message, alternatively, the one does not transmit through the switch is called the local communication message.

The after switch ST messages have the source and destination nodes in the different branches. Thus, the messages' source and destination information need to transmit to the FlexRay switch to indicate the switching path. This thesis introduces the notation $\mathbf{Matrix_m}$ to represent the switching path of message m. We assume the messages transmitted in the network have fixed switching path at different slots. Furthermore, the switching path needs to be known by the scheduler in advance. Therefore, on the basis of simple ST message's representation, the switched ST messages can be represented by a 4-tuple vector:

$$m = \left(T_m, MessageLengthST_m, Port_m, \mathbf{Matrix_m}\right) \tag{6.1}$$

$Port_m$ is the set of source and destination ports of message m. $\mathbf{Matrix_m}$ is the matrix of message m indicating the message transmission path.

The local communication ST messages have the source and destination nodes in one branch. The set of transmission related ports $Port_m$ only contains one element. The transmission of the message never passes through the switch in the network. Hence the switch does not need to know the transmission path $\mathbf{Matrix_m}$. Therefore, the local communication ST messages can be seen as a special case of the switch ST messages that the transmission path $\mathbf{Matrix_m}$ is 0.

The concept of cluster introduced in Section 6.1 is the key change in the switch FlexRay network. Multipath transmission significantly increases the system bandwidth. To enable multipath transmissions, we need to group ports to form the clusters, namely a broadcast group, based on the communications. The communication inside one cluster is the same as the simple FlexRay network. The time grid of the ST segment is the ST slot, so the clusters in the network regroup every slot. For each ST slot, there should be a corresponding switching matrix.

Like the simple ST messages, the switched ST messages are considered schedulable if it is possible to generate a valid static scheduling table. Hence the switched ST scheduler aims to generate a valid ST scheduling table after the clustering. The schedulable constraints of the switched ST scheduler are the same as the ones in the simple ST scheduler.

**Schedule Parameters**

In simple FlexRay network, the transmission follows the scheduling table stored in the hosts. In switch FlexRay network, the simultaneous transmissions in ST segment are realized by the scheduling tables stored in the nodes' hosts and the switching matrix table stored in the switch. The scheduling table consists of the same parameters as the simple FlexRay scheduling table:

$$\text{Schedule}_m = \{s_m, p_m, gNumberOfStaticSlots_m, b_m, r_m, gNumberOfCycle_m\} \quad (6.2)$$

The switching matrix table consists of a set of switching matrixes of different slot IDs in different cycles during the global static scheduling period $T_{ss}$. Therefore, the number of matrixes in the matrix table depends on the number of ST slots and number of CCs. Each matrix in the matrix table represents the connections between the switch ports of a particular instance. One example of the switching matrix [42] for cycle number $CycleNumber$ and Slot ID $SlotID$ shows below:

$$\mathbf{Matrix}_{\mathbf{CycleNumber,SlotID}} = \begin{pmatrix} 0 & Port_{1,2} & Port_{1,3} & Port_{1,4} \\ Port_{2,1} & 0 & Port_{2,3} & Port_{2,4} \\ Port_{3,1} & Port_{3,2} & 0 & Port_{3,4} \\ Port_{4,1} & Port_{4,2} & Port_{4,3} & 0 \end{pmatrix} \quad (6.3)$$

Matrix (6.3) is one of the switching matrixes in the switching matrix table of a four ports switch. In this matrix, the column represents the source port. The row represents the destination port. Therefore, it is a $4\times4$ matrix. The source port $i$ is marked by a "1" in the $i^{th}$ column. For example, if there are two messages transmitted in the network. One of the messages is transmitted from port 3 to port 2 during a slot. The other message is transmitted from port 1 to port 4 during the same slot. $Port_{2,3}$ and $Port_{4,1}$ in the matrix are marked by a "1". The switching matrix for this slot then becomes:

$$\mathbf{Matrix}_{\mathbf{CycleNumber,SlotID}} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \quad (6.4)$$

The 3rd and 1st column represent the source ports 3 and 1 and the 2nd and 4th row represent the destination ports 2 and 4. If the data transmits from port 3 to braches 1, 2 and 4, then the switching matrix for this slot becomes:

$$\mathbf{Matrix}_{\mathbf{CycleNumber,SlotID}} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad (6.5)$$

If an after-switch ST message transmits during some slots in some cycles, the switching matrix of these slots should have the switching path information of these messages so that the switch knows how to transfer the data. Besides the communications between the branches, the communications also happen locally inside one branch during some slots. If a branch communicates locally during some slots, it is disconnected by not having the switching path information in the switching matrix for those slots.

The discussion above shows that the data transmissions in the switched ST segment not only follow the clustering constraints but also need to satisfy the basic schedulable constraints. Therefore, the problems need to solve in the switched ST segment scheduler design are:

- Clustering and generating the switching matrix $\mathbf{Matrix}_{\mathbf{CycleNumber,SlotID}}$ for each ST slot;
- The value of the parameters in the schedule which can meet the schedulable constraints need to be set;

### 6.2.2 Motivation for the Solution

Section 6.2.1 pointed out the problem of switched ST scheduler design is to cluster the switch ports efficiently. The first goal of the switched scheduler is to allocate the ST messages in the schedule. The second goal is to reduce the system utility rate. Therefore, the priori consideration of the switched ST scheduler design is the schedulable constraints. The clustering constraints are further considered.

#### 6.2.2.1 Clustering Constraints

The switched ST scheduler aims to maximize the number of non-used slots to increase the system capacity. The scheduler tries to merge different slots' transmissions into one slot as much as possible. However, in order to avoid the data collision, clustering the possible messages needs to satisfy the clustering constraints listed in the following.

- A port cannot be the source and destination port at the same time;

  According to the FlexRay specifications, the interface between the CC and the BD has different signal lines for the data transmission and receiving. The interface between the host and the CC only has one signal line for the data transmission and receiving. Therefore, if a port is a destination port during some slots, it cannot be the source port during these slots [42]. This means a port of the switch cannot be the source port and the destination port during the same slot.

- A destination port cannot have more than one source port;

  According to the FlexRay specifications, the receive buffer in a FlexRay node might be the queued buffer. However, there is only one signal line in the node for transmission. Multiple sources will cause data collision. Therefore, a destination

port may have zero source ports when it communicates locally or one source port when it uses the switch to communicate.

- A branch that communicates locally during a slot needs to disconnect with the switch;

  A branch is a simple FlexRay cluster. The cluster's bus is occupied by the local communication data if there is local data communication. Therefore, other data cannot transmit on the bus at that slot. This branch cannot participate in the switched communication between other branches during that slot.

#### 6.2.2.2 Different valued parameters with simple ST scheduler

The goal of the switch FlexRay ST scheduler design is to maximize the utility rate of the used slots. If one slot is assigned to a message, the scheduler tries to find other messages that are able to transmit during this slot satisfied the constraints presented in Section 6.2.2.1. If successfully find the messages, the scheduler multiplex their transmissions.

$Port_{available}$ denotes a new concept called available port set in switch FlexRay network. It is defined as the unoccupied ports in the switch during that slot. This concept is very important because the clustering finds the transmission that can happen simultaneously by using different available ports $Port_{available}$, slot ID and cycle number. The vector space $V_m$ changes to represent the possible positions of the reference point with 3 parameters: base cycle $b_m$, base slot $s_m$ and the available ports $Port_{available}$. It is defined as following:

$$V_m = \left(S_m, CC_m, Port_{available}\right) \tag{6.6}$$

The slot ID set $S_m$ and the CC set $CC_m$ have the same definitions as the ones in the simple FlexRay network.

Section 6.2.2.1 defined three clustering constraints. Constraint 3 states that the local communication messages affect the clustering of the after-switch ST messages. The three constraints indicate that a switch port is occupied in a slot when it functions as the disconnected port, the source port or the destination. The occupied port set is notated as $Port_{use}$. These ports cannot be the source or the destination ports in other clusters during the same slot. It is clear to get the conclusion that $Port_{use}$ should be eliminated from $Port_{available}$ in the slot clustering. We also define the notation $Port_{switch}$ to represent the ports a FlexRay switch has. Thus we have:

$$Port_{available} = Port_{switch} - Port_{use} \tag{6.7}$$

### 6.2.3 Switched ST Segment Scheduling Algorithm

**Input:**

- Bus bit rate $\boldsymbol{bus\_speed}$
- Switch port set $\boldsymbol{Port_{switch}}$
- ST message set $\boldsymbol{M_{ST}}$ is the set of ST messages waiting to send in a network. $m = (T_m, MessageLengthST_m, Port_m, \mathbf{Matrix_m}), m \in M_{ST}$. The matrix $\mathbf{Matrix_m}$ is 0 if the message port set $\boldsymbol{Port_m}$ only contains one element.

**Output:**

- $m \times 6$ scheduling matrix $\mathbf{SwitchScheduler_{ST}}$:

$$\begin{bmatrix} s_1 & p_1 & gNumberOfStaticSlots_1 & b_1 & r_1 & gNumberOfCycle_1 \\ s_2 & p_2 & gNumberOfStaticSlots_2 & b_2 & r_2 & gNumberOfCycle_2 \\ s_3 & p_3 & gNumberOfStaticSlots_3 & b_3 & r_3 & gNumberOfCycle_3 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ s_m & p_m & gNumberOfStaticSlots_m & b_m & r_m & gNumberOfCycle_m \end{bmatrix}$$

- Switching matrix table:

$$\mathbf{MatrixTable_{ST}} = \begin{Bmatrix} Matrix_{63,1} & Matrix_{63,1} & \dots & Matrix_{63,gNumberOfStaticSlot} \\ \dots & \dots & \dots & \dots \\ Matrix_{1,1} & Matrix_{1,2} & \dots & Matrix_{1,gNumberOfStaticSlot} \\ Matrix_{0,1} & Matrix_{0,2} & \dots & Matrix_{0,gNumberOfStaticSlot} \end{Bmatrix},$$

in which:

$$\mathbf{Matrix_{CycleNumber,SlotID}} = \begin{pmatrix} 0 & Port_{1,2} & Port_{1,3} & Port_{1,4} \\ Port_{2,1} & 0 & Port_{2,3} & Port_{2,4} \\ Port_{3,1} & Port_{3,2} & 0 & Port_{3,4} \\ Port_{4,1} & Port_{4,2} & Port_{4,3} & 0 \end{pmatrix}$$

- $SlotID \in [1, gNumberOfStaticSlots], SlotID \in \mathrm{N}$

  $CycleNumber \in [0, 63], CycleNumber \in Z$

---

Switched FlexRay ST scheduling algorithm

---

for $\forall m \in M_{ST}$ do // assign values to different parameters based on message types

---

$$gdStaticSlot = \frac{gdCycle}{gNumberOfStaticSlot}$$

$D_m = T_m$

if $D_m > 30000$

$D_m = 30000$

end if

if $D_m < gdCycle$

$r_m = 1$

$$p_m = \max\left\{2^n \middle| n \in [0,9], n \in Z, p_m \le \frac{D_m}{gdStaticSlot}\right\}$$

$$gNumberOfStaticSlots_m = \left\lceil \frac{gdCycle}{p_m \times gdStaticSlot} \right\rceil$$

$gNumberOfCycle_m = 64$

else

$p_m = 1023$

$gNumberOfStaticSlots_m = 1$

if $D_m > gdCycle$

$$r_m = \max\left\{2^n \middle| n \in [0,6], n \in Z, r_m \le \frac{D_m}{gdCycle}\right\}$$

$$gNumberOfCycle_m = \frac{64}{r_m}$$

else // message type $D_m = gdCycle$

$r_m = 1$

$gNumberOfCycle_m = 64$

end if

end if

end for

$$gNumberOfStaticSlots_{\min} = \max\left(2, \left\lceil \sum_{\forall m \in M_{ST}} \frac{gNumberOfStaticSlots_m}{r_m} \right\rceil\right)$$

if $gNumberOfStaticSlots_{\min} > gNumberOfStaticSlots$

output non-schedulable, exit //system non-schedulable

end if

for $\forall m \in M_{ST}$ do

Sort the messages $\forall m \in M_{ST}$ in ascending order of the values $p_m$ and then in ascending order of values $r_m$ store them in the message list $L_{ST}$

$p_{min} = \min(p_m)$

end for //sort finish

$V_{use} = (S_{use}, CC_{use}, Port_{use}) = (0,0,0)$

$\mathbf{MatrixTable_{ST}} = 0$

for $m \in L_{ST}$ do

Slot ID set $S_m = [0, p_m - 1]$

CC set $CC_m = [0, r_m - 1]$

$V_m = (S_m, CC_m, Port_{switch})$

$V_m = V_m - V_{use}$ // update the available space of the reference point

if $V_m$ is empty set

output system non-schedulable, exit

end if

for $(s_m, b_m, Port_m) \in V_m$

for $\forall m \in [1, gNumberOfStaticSlots_m], m \in \mathrm{N}$

for $\forall n \in [1, gNumberOfCycle_m], n \in \mathrm{N}$

$\text{buffer}_m = \{s_m + (m-1) \times p_m,\ b_m + (n-1) \times r_m,\ Port_m\}$

end for

end for

if elements in set $Port_m$ >1 //update the matrix of that slot

$\mathbf{Matrix_{CycleNumber,SlotID}} = \mathbf{Matrix_{CycleNumber,SlotID}} + \mathbf{Matrix_m}$ ,

$SlotID = s_m + (m-1) \times p_m \quad , \quad \forall m \in [1, gNumberOfStaticSlots_m], m \in \mathrm{N}$

$CycleNumber = b_m + (n-1) \times r_m, \forall n \in [1, gNumberOfCycle_m], n \in \mathrm{N}$

end if

$V_{use} = V_{use} + buffer_m$

$gNumberOfStaticSlots_{new} = s_m + (gNumberOfStaticSlots_m - 1) \times p_m$
//update used maximum ID ST slots

if $gNumberOfStaticSlots_{new} > gNumberOfStaticSlots_{use}$

$gNumberOfStaticSlots_{use} = gNumberOfStaticSlots_{new}$

end if

take m out of $L_{ST}$, update $L_{ST}$

if $L_{ST}$ is empty

output $\mathbf{SwitchScheduler_{ST}}$ and $\mathbf{MatrixTable_{ST}}$, exit //system schedulable

end if

continue with next $m \in L_{ST}$

end for

end for // continue with next value of $gdCycle$

**Algorithm 3 Pseudocode for Switched ST scheduling algorithm**

## 6.3 Scheduler Design for Switched FlexRay DYN Segment

### 6.3.1 Problem definition

By grouping sub-set of ports into clusters, different switched ST messages can transmit simultaneously. Each cluster functions like a simple FlexRay network. Therefore, a switch FlexRay network can be seen as the combination of a few simple FlexRay networks. The clustering increases the system bandwidth and enables simultaneous collision-free transmissions. The switched DYN segment communication can consult the method in switched ST communication and make necessary modifications. To distinguish the simple DYN message from the messages transmitted in the FlexRay network, the latter is called switched DYN message.

Section 2.2.2.3 introduced that FTDMA is the media access mechanism in DYN segment. In DYN segment, the messages could send in any instant in terms of the number of Minislots with an arbitrary length of payload (maximum is 127 2-byte-words). There are two major differences between ST segment and DYN segment. The first difference is the length of the slot. The DYN slot length is varied by the size of the messages while the ST slot length is fixed in a cluster. The second difference is the message response time. The transmission of other messages could affect the response time of the switched DYN message. The ST messages use fixed slot transmission to prevent the interferences from other messages. Every switched DYN message is allocated a FrameID for bus arbitration.

To enable the switched transmission, it is necessary to known the source and destination ports of the switched DYN message. The known parameters of a switched DYN message $m$ are the message length$MessageLengthDYN_m$ , the message deadline$D_m$, the set of source and destination ports $Port_m$ and the message switching path $\mathbf{Matrix_m}$ . Therefore the switched DYN messages can be represented by a 4-tuple vector:

$$m = \{MessageLengthDYN_m, D_m, Port_m, \mathbf{Matrix_m}\} \tag{6.8}$$

Local communication message is a special switched DYN message that does not pass through the switch. So the message ports set only contains one port that is both the source and destination port. It can be seen as no switching information. Thus, the switching path $\mathbf{Matrix_m}$ is 0.

The goal of the switched DYN scheduler is to increase the utility rate of the shared resource and to improve the system bandwidth. Therefore, efficiently build the broadcast groups based on the communication demands is the crucial issue. The clustering constraints are the same as the ones in switched ST segment. The regrouping period of the clusters in the switched DYN segment is slightly different from the period in the switched ST segment. The fixed time grid is Minislot in the DYN segment instead of ST slot in the ST segment. However, the duration of the Minislot is too short to be the regrouping period. So the regrouping period should be multiple of the Minislot duration. For each period, there should be a corresponding switching matrix stored in the switch.

The schedulable constraints are the same as the ones in simple DYN segment. The switched DYN segment is considered schedulable if every DYN messages' worst-case response time $R_m$ is shorter than or equal to the deadline $D_m$. Hence the switched DYN scheduler should calculate the worst-case response time after the clustering step and compare them with the deadlines to determine the system schedulability.

In conclusion, the problems needed to solve in the switched DYN scheduler design are:

- Clustering;
- Calculating the switched DYN message worst-case response time $R_m$;

## 6.3.2 Motivation for the Solution

### 6.3.2.1 Clustering

The analysis in the previous section has already shown that, for the DYN segment, the regrouping period should be multiple of the Minislot. There are two methods of choosing the regrouping period.

- Regrouping the cluster every CC

Since the duration of the CC is fixed, it is suitable to use as the regrouping period [42]. Therefore, the switch regroups the nodes at the interval of a CC. In other words, the switch can have different clusters during each CC and up to 64 clusters in total.

Besides the regrouping period, there is one more difference between the switched ST scheduler and the switched DYN scheduler. In the switched DYN scheduler, the branch could be both source and destination branch in a cluster. Although the regrouping period of the cluster is one cycle, the timing of data transmission is Minislot-based. Therefore, the node can change part in different Minislots. Furthermore, the branches in one cluster should have the same value of the slot counter $vSlotCounter_{DYN}$.

- Flexible regrouping period

The second method tries to refine the clusters by reducing the set of branches belonged to a cluster. The refinement will increase system bandwidth since it allows more transmissions happen at the same time than the big cluster. The refinement is done by reducing the regrouping period of the clusters. Just as the discussion in the previous section, the newly refined period is a specific period that is multiple of the Minislot duration and is shorter than the duration of a CC. Each regrouping period could be different. The regrouping points need to be defined in advance based on practical communication demands to obtain the maximum system bandwidth. The switch builds the new clusters at every regrouping point. The clustering is based on the switching matrix pre-stored in the switch that describes the connection of the switch ports. Theoretically, this can help to gain more bandwidth but may require lot more configuration memories.

This thesis chooses the duration of the CC as the regrouping period because the switched DYN scheduler is a general scheduler, not a scheduler for a particular implementation. The flexible regrouping points cannot be set without the detail transmission information. Furthermore, the first method still can gain more bandwidth without increasing too many configurations and hardware requirements.

#### 6.3.2.2 Calculating the worst-case response time $R_m$

In the schedulability analysis, it is sure that applying the switch in FlexRay network introduces additional switching delay. The worst-case response time becomes:

$$R_m = w_m + C_m + S_{delay\_m} \tag{6.9}$$

$S_{delay\_m}$ is the delay of message $m$ caused by FlexRay switch. Section 6.2.1 introduced two types of messages in the switch FlexRay network. One called after-switch message, the other called local communication message. For the local communication message, it does not pass through the switch. Therefore, the switching delay $S_{delay\_m}$ equals to 0.

The switching delay $S_{delay\_m}$ is defined as from the transceiver of the sending port to the transceiver of the destination port [42]. This switching delay's definition not considers the upper layer delays in the switch such as searching-switching-entry delay. It is the physical switching delay. According to FlexRay specifications, the maximum delay allowed for the active star is $150ns$. So the switching delay is required to keep within this bound. In the experimental setup in [42], the delay is proven to successfully stay within the bound. Therefore, in the calculation of worst-case response time, the switching delay $S_{delay\_m}$ can be ignored.

Introducing the switch does not change the bus arbitration scheme. The mathematic model of the bus arbitration delay $w_m$ and transmission delay $C_m$ is the same as the simple FlexRay analysis. Switched FlexRay network enables parallel communication between branches, which virtually transform one communication network into several networks. From the aspect of the system schedulability, this equals to reduce the system size and increase the chance to access the bus.

In simple FlexRay analysis, the *1DBP* algorithm is used to calculate the delay $BusCycles_m(lf(m,t))$. The bin is the DYN segment. The items to fill the bin are the elements in $lf(m,t)$. Since the messages are clustered, the messages in one cluster can send simultaneously. The original transmission order of the messages changes. The analysis of the DYN message transmission in Section 4.5 shows that the message transmission order is decided by the Frame ID. Therefore, we can see the transmission order changes as the Frame ID changes.

For the messages in the same cluster, the Frame IDs of messages update to the smallest Frame ID in the cluster. For messages not included in a cluster, from their point of views, the messages in a cluster are considered as one representative message. The size of the representative message is the longest message length in that cluster. For the messages that

change the Frame ID, the messages in set $lf(m,t)$ are decreased, and the delay $BusCycles_m(lf(m,t))$ is reduced.

In order to calculate the worst-case response time $R_m$ of the switched DYN messages, the worst-case scenario is that all DYN messages are ready for transmission at the same time. After the clustering step, based on the new Frame ID, the method to calculate $R_m$ is the same as the one in simple DYN network.

### 6.3.3 Switched DYN Segment Scheduling Algorithm

The motivations of solutions already presented in the previous section. Based on the discussions, we provide the DYN segment scheduling algorithm for a known network in Algorithm 4.

**Input:**

- Bus bit rate $\boldsymbol{bus\_speed}$
- Switch port set $\boldsymbol{Port_{switch}}$
- The DYN message set$\boldsymbol{M_{DYN}}$, $\boldsymbol{M_{DYN}}$ is maximum DYN message set waiting to send in a cluster, $m = \{MessageLengthDYN_m, D_m, Port_m, \mathbf{Matrix_m}\}, \forall m \in M_{DYN}$

**Output:**

System schedulable or non-schedulable

---

Switched FlexRay DYN scheduling algorithm

---

for $\forall m \in M_{DYN}$

Sort the DYN messages $\forall m \in M_{DYN}$ in descending order of the value $\frac{MessageLength_m}{D_m}$ and then in descending order of the value $MessageLength_m$, store them in the message list $L_{DYN}$

Assign the Frame ID to the DYN messages with the order in list $L_{DYN}$ from 1

end for // sort the messages and assign the Frame IDs

// start of clustering and Frame IDs updating

for $\forall m \in L_{DYN}$

---

if $m$ is the first message in $L_{DYN}$

create set $cluster_1 = \{\text{empty}\}$ //create the first message set $cluster_1$

$ClusterSet = \{cluster_1\}$ // create a set represented all the available clusters

$Port_{use\_cluster_1} = Port_m$

$m \in cluster_1$ // put $m$ in set $cluster_1$

else // $m$ is not the first message

for all $cluster_1$ to $cluster_m$ in the set $ClusterSet$

//for all valid cluster in $ClusterSet$

if $Port_m \subseteq Port_{valid\_cluster_n}$ , $n \in [1, m], n \in \mathbb{N}$

$Port_{use\_cluster_n} = Port_{use\_cluster_n} + Port_m$

$m \in cluster_n$ // put message m in $cluster_n$

go to the next message

// if message is able to put in any one of the existed $cluster_n$, finish the clustering step of message m

end if

end for

create set $cluster_{m+1} = \{\text{empty}\}$ //create a new set $cluster_{m+1}$

/* if search all available clusters message still cannot find a cluster can fit in, create a new cluster */

$Port_{use\_cluster_{m+1}} = Port_m$

$m \in cluster_{m+1}$

// put $m$ in the new created cluster $cluster_{m+1}$

go to the next message //finish the cluster step of this message

end if

$Port_{valid\_cluster_n} = Port_{switch} - Port_{use\_cluster_n}$

//update the valid ports for any set $cluster_n$ included message m

if $Port_{valid\_cluster_n} = 0$

$ClusterSet = ClusterSet - cluster_n$

//update the valid cluster set $ClusterSet$

end if

end for // finish the clustering step for all the DYN messages

for $\forall m \in L_{DYN}$ // update the Frame ID for each DYN message

$FrameID_m = FrameID_{min}\ in\ cluster_n,\ message\ m \in cluster_n$

end for

// start the calculation of the worst-case response time

for $gdCycle = 10$ to $16000\mu$s step $20 \times gdBit$ µs

simple FlexRay ST scheduling algorithm

$DYN_{bus} = gdCycle - ST_{bus}$

for $m \in L_{DYN}$

$pLatestTx_m =$
$(DYN_{bus} - (gdTSSTransmitter + 83 + MessageLengthDYN\ [\text{bit}] * 1.25) * gdBit) / (gdMacrotick * gdMinislot)$

FFD bin packing algorithm calculates the delay $BusCycles_m(lf(m,t))$

$R_m =$
$(gdCycle - (ST_{bus} + (FrameID_m - 1) * gdMinislot)) +$
$BusCycles_m(lf(m,t)) * gdCycle + (FrameID_m - 1) * gdMinislot +$
$(ST_{bus} + pLatestTx_m * gdMinislot) +$
$((gdTSSTransmitter + 83 + MessageLengthDYN) * 1.25 * gdBit/bus\_speed)$

if $R_m < D_m$

take $m$ out of list $L_{DYN}$

if $L_{DYN}$ is not empty

continue with next message in list $L_{DYN}$

else

output schedulable, exit

end if

else

next value of $gdCycle$

```
            end if
        end for
    end for
    output non-schedulable, exit  //system non-schedulable
```

**Algorithm 4 Pseudocode for Switched DYN scheduling algorithm**

## 6.4 Conclusion

The method of the switch FlexRay scheduler design is solving the differences and following the similarities. Firstly, the concept of switch FlexRay network was introduced. Secondly, we analyzed the differences between simple and switched network and found the solutions of these differences for the ST and DYN segment respectively. The main difference between the switched and the simple scheduler design is that it needs an extra step before the simple scheduling algorithm, called clustering. Forming the ECUs' clusters depends on the transmission path of the messages and the switch ports number. If none of the ports in two messages' port set is the same, then these two messages can transmit in the same slot, namely slot-sharing messages. For ST messages, clusters are regrouped at the interval of a slot. Different slot has the different clusters. For DYN messages, clusters are regrouped at the interval of a CC. Different cycle has the different clusters. At the end of the discussion, the switched scheduling algorithms were presented.

# 7

# Experimental Results

After theoretical analyses and conceptual designs of schedulers for two segments in two types of networks, this chapter aims at the performance evaluations for these four schedulers and comparisons in between the simple scheduler and the switched scheduler of ST and DYN segment respectively by using four C++ programs designed for four schedulers.

The evaluations of ST schedulers are executed in two aspects: schedulability analysis and system resource utilization analysis. The worst-case response time is the key characteristic for the DYN segment scheduling. Therefore, the response time analyses are the evaluation standard of the DYN schedulers.

## 7.1 Experimental Setup

The Electronic Damper Control (EDC) technology in BMW X5 is the pilot application of FlexRay technology in BMW's productions. Therefore, we choose this application as the evaluative scenario. This section aims at performance evaluation of the scheduling algorithm presented in section 5.1.3 in the BMW EDC application.

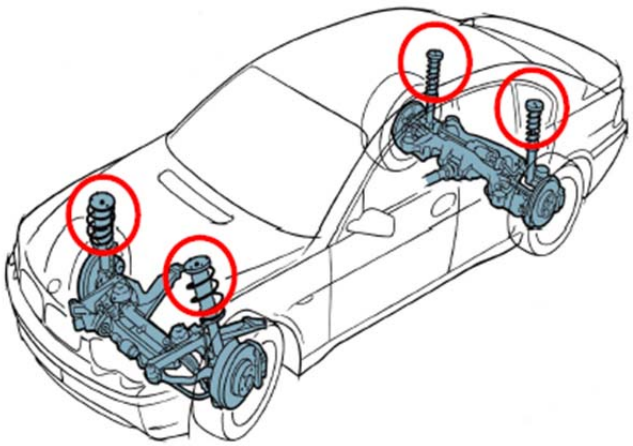

**Figure 7-1 Hardware Distribution of EDC in BMW X5 [43]**

Figure 7-1 shows the general hardware distribution of the EDC in BMW X5. Each red circle represents a FlexRay node.

The experiment uses the following FlexRay configuration settings, which are in line with the design settings disclosed by BMW [43, 44].

- The duration of the CC fixes to 5ms, denoted by $gdCycle$. The duration of ST segment fixes to 3ms, denoted by $ST_{bus}$. This thesis ignores the duration of the SYM and the NIT. So the DYN segment fixes to 2ms, denoted by $DYN_{bus}$.
- The data payload of a ST frame sets to 8 2-bytes words, which is 16 bytes, denoted by $gPayloadLengthStatic$.
- The length of the Macrotick set to 2μs, denoted by $gdMacrotick$.
- The length of the Minislot set to 5MT, denoted by $gdMinislot$.
- The bus bit rate set to 10Mbit/s.

## 7.2 ST Segment Scheduler Performance Evaluation

This thesis uses C++ language programs two programs, which correspond to the simple ST scheduler and the switched ST scheduler. Section 5.1.3 and Section 6.2.3 introduced these two schedulers. The source codes of these programs can refer to Appendix A and B.

The following sections will present four evaluations: the system loads, used ST slots, percentage of scheduled system and the system schedulability. By analyzing these tests, we will know the performances of the two ST schedulers.

Different number of ST messages input to the schedulers, which generate varying results. For every level of message number, the programs randomly generate 100 sets of messages for the evaluations. Each set represents a different system. In the scenario, each ST message has the same length 8 2-byte-word. The number of star sets to 0 in the simple FlexRay network and sets to 1 in the switch FlexRay network. Other system related parameters are set in the program. The user can change the value of the parameters in the source code in the Appendix if the system parameters changed. All the statistics results are made from 100 sets of messages.

### 7.2.1 System Loads

The goal of system loads evaluation is that can show the average used system bandwidth with different number of ST messages inputted to two schedulers. The results show in Figure 7-2.

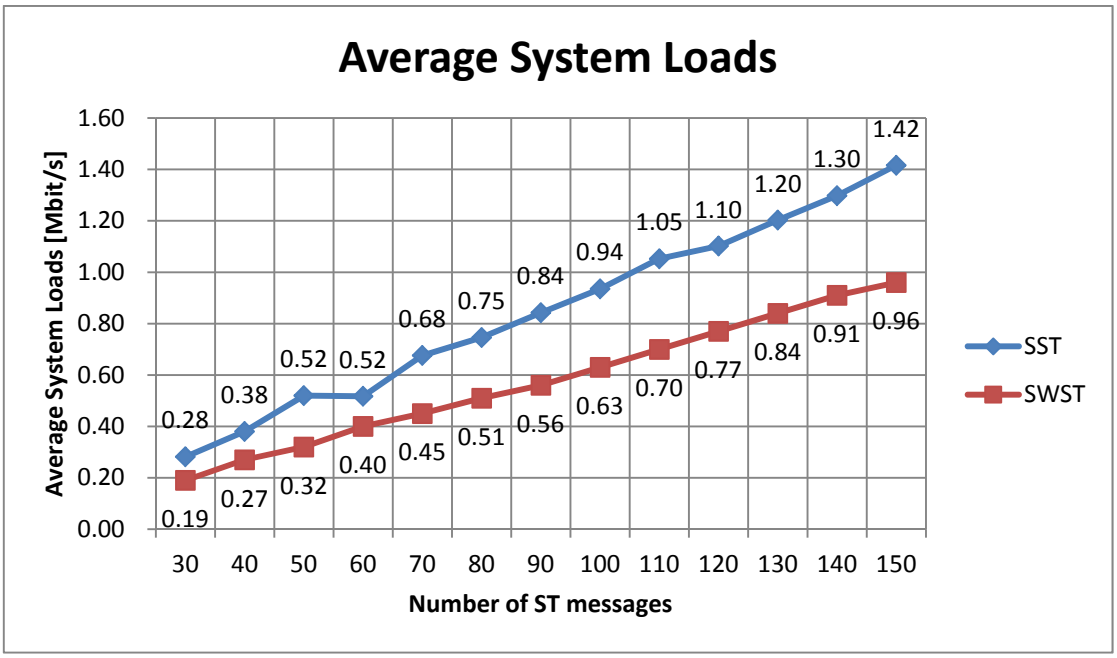


**Figure 7-2 Average System Loads**

Formula (7.1) calculates the system load, namely the system bit rate.

$$Load = gNumberOfStaticSlots_{\min} \times MessageLengthST_{max} / gdCycle \qquad (7.1)$$

The minimum required number of ST slots $gNumberOfStaticSlots_{\min}$ is the number of slots occupied by ST messages in the ST segment. For example, to schedule a ST message set successfully requires the minimum number of ST slots 60, the message length for each message is 16 bytes = 64 bits. The system load is 60 * 64/5000 = 0.768 Mbit/s. By summing up the 100 systems' loads and averaging by the test time100, we can get the average system load.

As we can see in Figure 7-2, the system loads are lower in the switched ST (SWST) scheduler than the ones in the simple ST (SST) scheduler under each level of input messages. This is because of the slot-sharing mechanism in the SWST scheduler. The SWST scheduler schedules each message based on its transmission pattern and path which contains the information of the source and destination ports. The source and destination ports form the message's ports set. If two messages have different port set, then they can share the transmission in one slot. This slot-sharing mechanism is the reason why the SWST scheduler has lower system load for each level of message number.

### 7.2.2 Number of Slots Used

The number of slots used means the maximum ST slot ID required by a set of ST messages' transmission. The used ST slots are not always equals to the minimum number of ST slots as we explained in Section 4.4. The transmission patterns of messages cannot match with each other perfectly all the time. There might be some wasted slots. So the used ST slots usually more than the minimum slots required. By summing up the used number of slots in

the scheduled systems and averaging by the number of scheduled systems, we can get the average number of used slots.

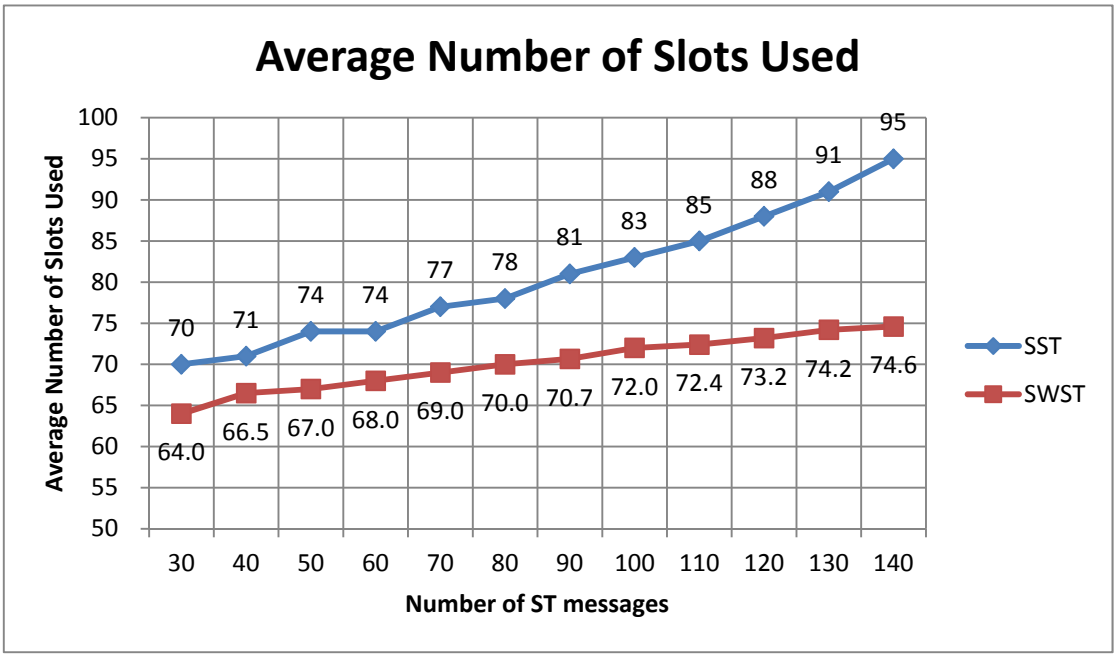


**Figure 7-3 Average Number of Slots Used**

The results shown in Figure 7-3 are the same as expected. It shows that the SWST scheduler requires less number of slots than the SST scheduler to schedule the same amount of ST messages. This fact indicates the same that the SWST scheduler has better ability to save the system resource than the SST scheduler.

### 7.2.3 Percentage of Schedulable Systems

By evaluating 100 sets of messages, we get the percentage of successfully schedulable systems. In other words, this test shows the number of schedulable systems in 100 testing systems. The useful bandwidth is the maximum bandwidth the system could make use of. The bandwidth defines in formula(7.2):

$$Bandwidth = gNumberOfStaticSlots \times MessageLengthST_{max} / gdCycle \quad (7.2)$$

Since the number of ST is 96 in our scenario, the useful bandwidth is 96 * 64/5000 ≈ 1.23 Mbit/s. So we can get the conclusion that the system has better schedulability when the system load is less than the useful system bandwidth 1.23 Mbit/s.

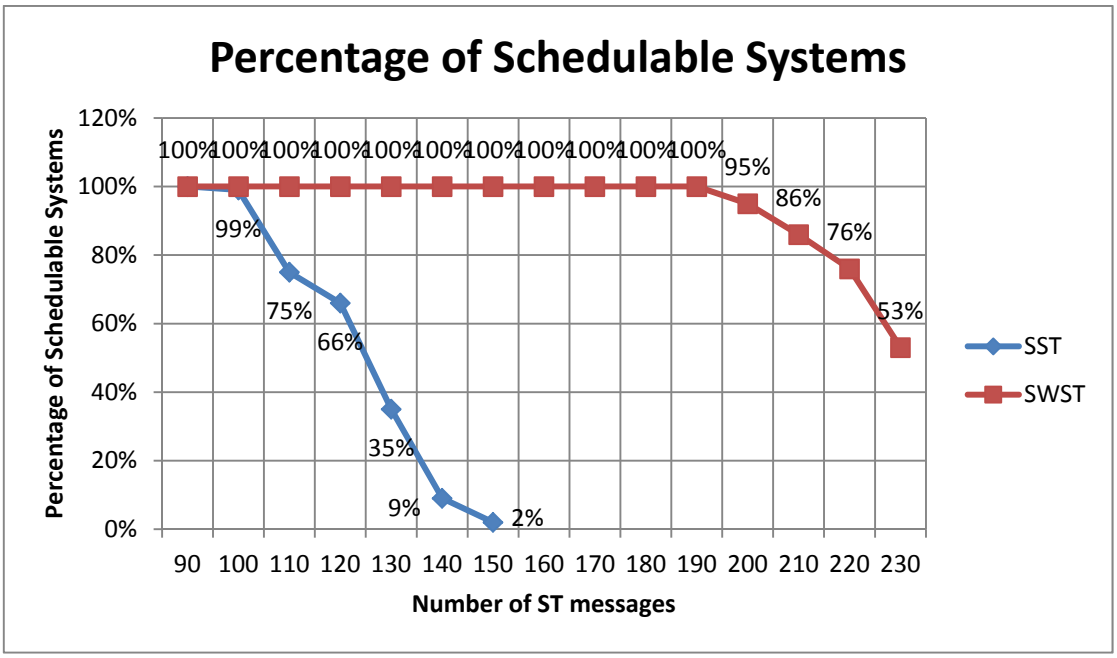


**Figure 7-4 Percentage of Schedulable System**

Figure 7-4 shows that more systems scheduled by using the SWST scheduler than scheduled by the SST scheduler. This fact indicates the SWST scheduler has better capability of scheduling than the SST scheduler when has the same number of input messages.

### 7.2.4 System Schedulability

The system schedulability evaluates the percentages of scheduled messages in a message set. The evaluations are done for different number of input ST messages. The results are the average values of the system schedulabilities of 100 systems. In each system, we count the total number of messages that are scheduled by the scheduler and calculate the percentage of the scheduled messages among the whole message set. By averaging 100 systems, we get the average system schedulability of different number of input messages.

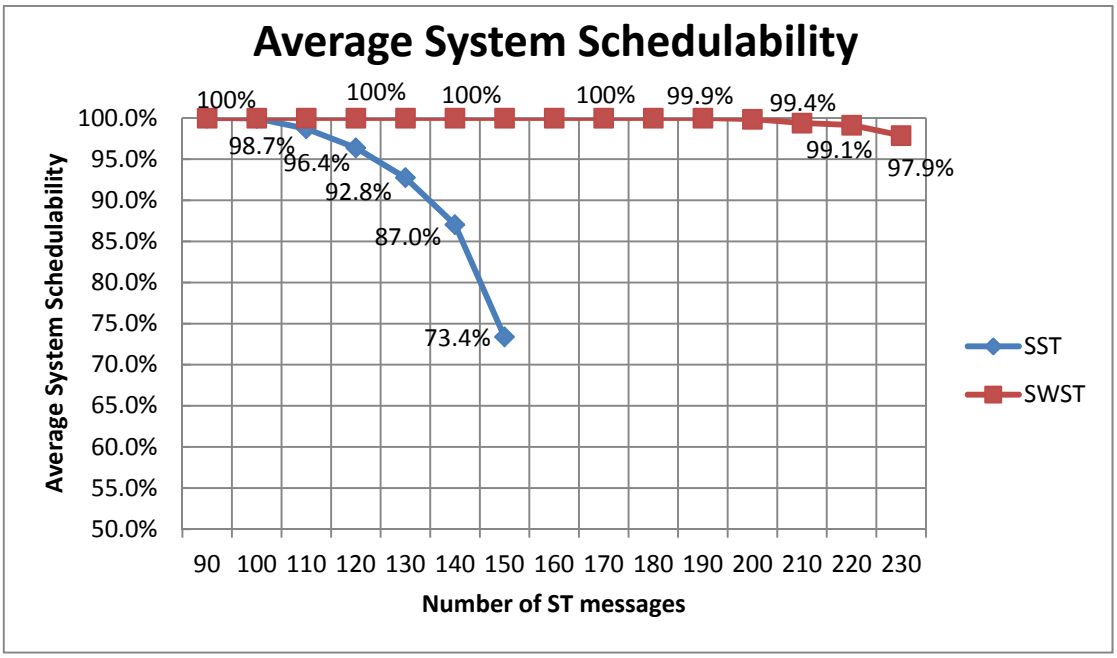


**Figure 7-5 Average System Schedulability**

As we can see from Figure 7-5, the SWST scheduler has the better system schedulability than the SST scheduler. This is also because of the slot-sharing mechanism in SWST scheduler as explained Section 7.2.1.

The schedulability has a modest decreasing from 140 messages to 150 messages when the system applies the SST scheduler. Because there are more messages requiring the same system capacities from the system, but the system cannot provide these capacities when it reaches some limit. This may cause the scheduled failures of messages after a certain message and the modest decreasing of the schedulabilities.

## 7.3 DYN Segment Scheduler Performance Evaluation

### 7.3.1 Worst-case Response Time without topology information

The same as the performance evaluations conducted for the ST scheduler, evaluations for the DYN scheduler are also done by two C++ programs. The key characteristic of the DYN message is the worst-case response time $R_m$. Therefore, we focus on the calculation of $R_m$ in our evaluations. Chapter 4 already presented the method to determine the worst-case response times $R_m$. In the evaluation, the SWDYN scheduler randomly allocates switch ports to DYN messages, and then follows the method in Chapter 4 to calculate $R_m$. Before the calculation, the SWDYN scheduler has one more step which uses the slot-sharing mechanism to cluster the messages and reallocate FrameIDs. The switch ports messages occupied and message length will affect the worst-case response times.

We use 10 DYN messages in the evaluations and assign them different message lengths. The smallest messages size is 500 bits.

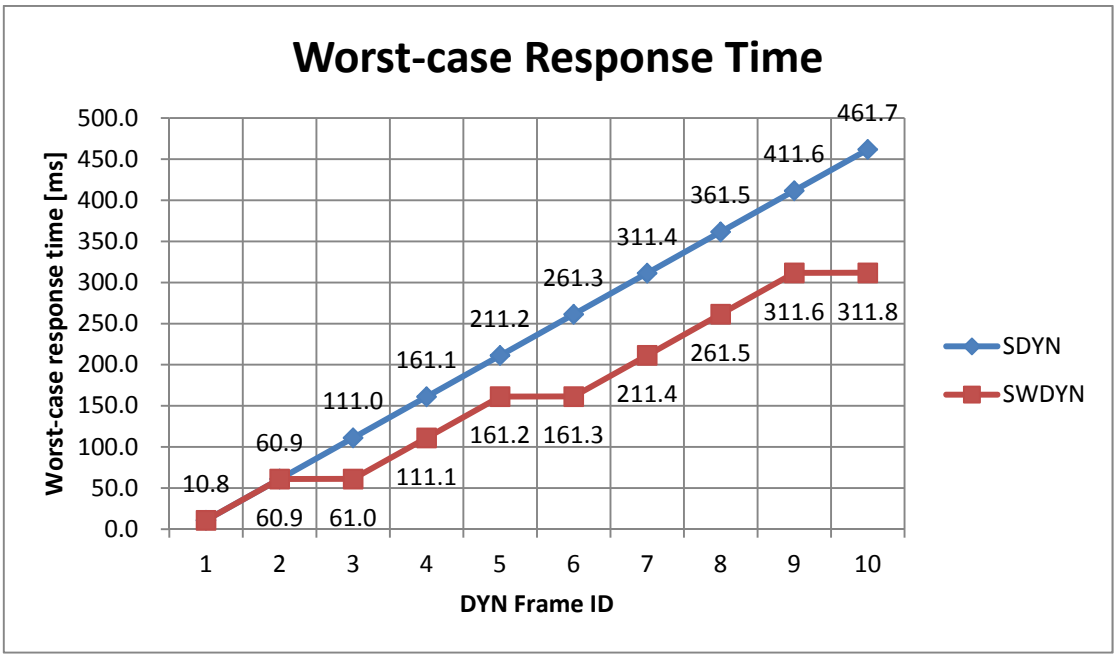


**Figure 7-6 Worst-case Response Times**

As we can see from Figure 7-6, the message with the original FrameID 5 is sharing the slot with message has FrameID 1. The message with the original FrameID 9 is sharing the slot with the message has FrameID 10. Hence message 5 has the same response time as message 1 and message 10 has the same response time as message 9. The clustering and FrameID reallocation reduce the response time of these two messages. It is clear to tell from Figure 7-6 that the worst-case response times of the SWDYN scheduler are equal or less than the SDYN scheduler. Therefore, we can draw the conclusion that if there is any message satisfies the slot-sharing condition, the response time of the message with large FrameID will decrease. The response time of the other messages which have large FrameIDs will decrease either.

### 7.3.2 Worst-case Response Time with Path Delays

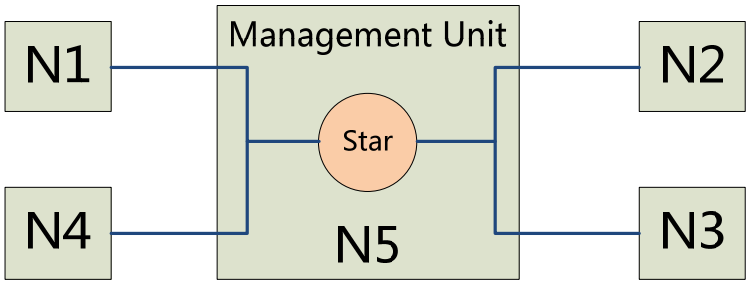


**Figure 7-7 Abstract Topology of EDC**

Figure 7-7 shows the abstract topology of the EDC in BMW X5. The nodes N1 to N4 are slave ECUs, which correspond to the four red circles in Figure 7-1. There is a central management unit which connects these four nodes by an active star. The management unit can also be seen as a node N5.

Taking topology into consideration, there is one parameter will affect the response time: the longest length of the wire between the source mode and the destination node. The wire causes the part of the delay called path delay. The following paragraphs will evaluate the worst-case response time with path delays.

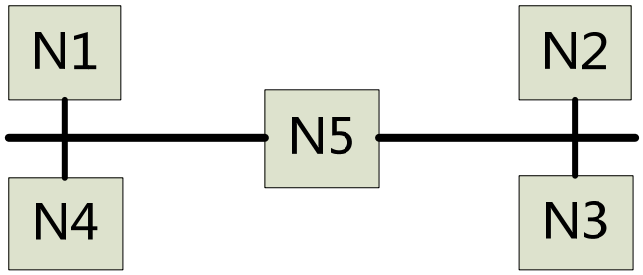


**Figure 7-8 Experimental Simple FlexRay Network**

As the topology shown in Figure 7-8, the experimental network removes the active star and replaces it with a bus. The bus connects five ECUs. Figure 7-9 shows a topology that a FlexRay switch replaces the active star connected four ECUs. The switch locates in node 5, which is also the management unit in the network.

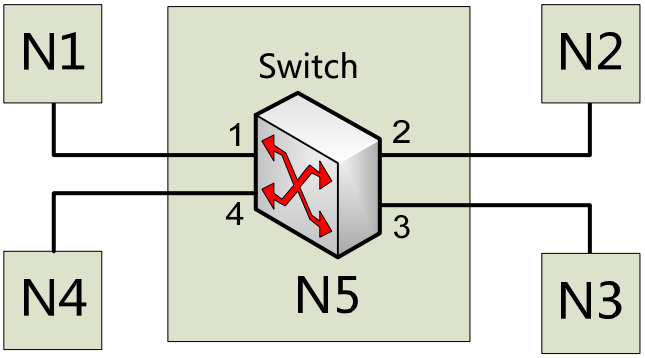


**Figure 7-9 Experimental Switched FlexRay Network**

We assume the longest wire length between two ECUs is 300 meters. If the transmission happens between ports 1 and 4 or 2 and 3, the wire length is 200 meters. If the transmission only happens locally, the wire length is 100 meters. Therefore, based on the wire length we can calculate the path delay. For example, the worst path delay, which is the wire transmission delay of 300 meters, equals to $300\text{m}/\ 3 \times 10^8\ \text{m/s} = 1\text{ms}$. $3 \times 10^8\ \text{m/s}$ is the speed of light.

Figure 7-10 illustrates the results that the response times of the SDYN scheduler and the SWDYN scheduler. We use the same set of messages as in Figure 7-6 for testing.

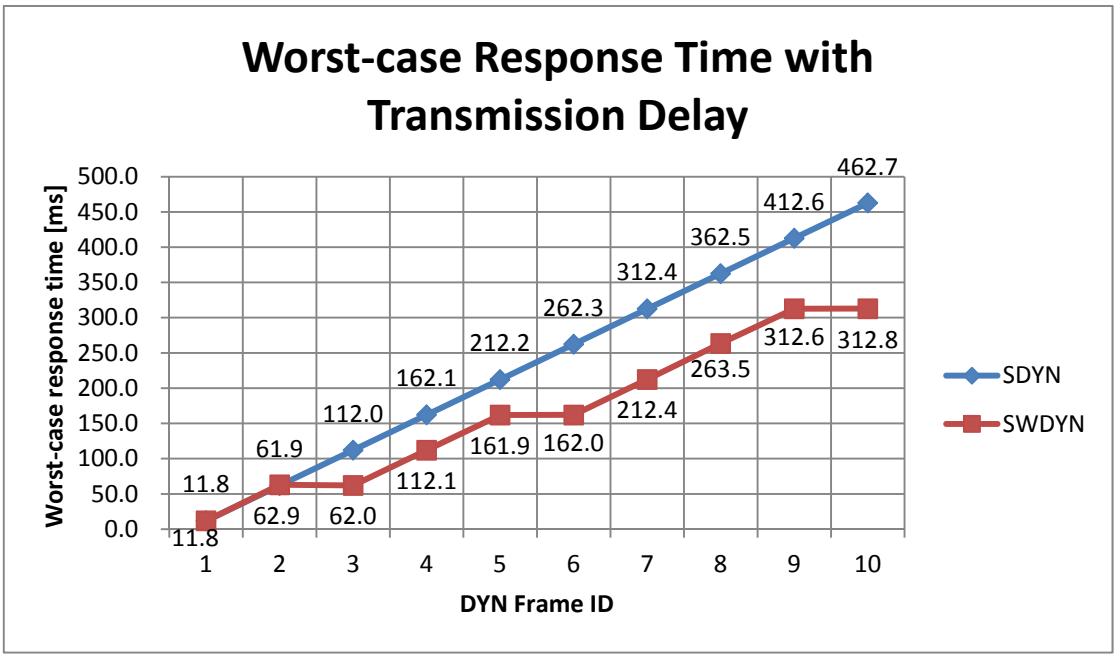


**Figure 7-10 Worst-case Response Times with Media Transmission Delay**

Through the comparison of Figure 7-6 and Figure 7-10, we can tell that the patterns of the response times vary not much. Compared with bus arbitration delays, the path delays are relatively small.

## 7.4 Conclusion

This chapter presented the test data in the EDC application in the simple and switch FlexRay schedulers. In the evaluation, we used the actual data from automotive manufacturer to generate the test data. We compared these two types of schedulers for ST and DYN segment respectively. In the evaluation of the ST segment, four parameters, average system loads, average used ST slots, the percentage of scheduled system and the system schedulabilities, were used to analyze the data under different number of input messages. In the evaluation of the DYN segment, the evaluated parameter is the worst-case response time without and with the path delays.

# 8

# Conclusion and Future Work

The next generation of automotive control network uses the switch FlexRay networks as its core networks and incorporates with other automotive control networks, to connect the vast amount of electronic systems. This thesis designs two schedulers for simple and switch FlexRay networks each. These schedulers can evaluate the effectiveness of the parameters in the design and measure the system upper bound. These evaluation results can be used as design references that assist the in-vehicle embedded systems design and further improvement. In this chapter, we will conclude the works we did in this thesis and will discuss the future work that can be done to improve this project.

Section 8.1 will summarize the content of the thesis. The contributions and some future work suggestions of this thesis will give in section 8.2.

## 8.1 Conclusion

Chapter 2 gives the background information of this thesis. This chapter discusses the FlexRay network and its working principle and briefly introduces the types of topologies and components in the node. The protocol physical and media access layers were introduced, as well. Furthermore, Chapter 2 presents the timing hierarchies of the basic timing unit, CC, in details. On one hand, the FlexRay protocol offers scalable dependability and fault tolerance, which realize by redundancy transmission; on the other hand, the FlexRay protocol is also very flexible. By using the time-triggered communication in ST segment, nodes are assigned as many slots as needed to ensure the reliable ST communication. Meanwhile, the DYN messages only transmit when it is necessary to save the resource. The FlexRay network supports various topologies like bus topology, star topology, mixed topology and switched topology.

Chapter 3 discussed the real-time scheduling theory. Common classifications of real-time scheduling were given. We can classify the scheduling policies into preemptive and non-preemptive, offline and online, or clock-driven, priority-driven, etc. This chapter also introduces a few representative scheduling algorithms in the fix-priority and dynamic priority scheduling. The previous studies indicate that the RM scheduling is the optimal fix-priority

algorithm when the tasks' deadlines are equal to their periods. The DM scheduling is another fix-priority scheduling algorithm which elevates the constraint of the deadline. It is the optimal algorithm when tasks have deadlines less than (or equal) to periods. The EDF scheduling is one of the optimal algorithms in dynamic-priority scheduling.

In Chapter 4, the differences of two segments in the CC were discussed. We draw the conclusion that different schedulers should be used for different segments. Then we distinguished the concepts of task and message. This chapter also analyzes the timeline of the transmission and discusses the latency in detail, to help to understand the discussion about response time in the following contents. In this chapter, we determine that the bus arbitration and data transmission compose the latency discussed in this thesis. The final part of this chapter investigates schedulabilities of the ST and DYN segments in the simple FlexRay networks.

Chapter 5 and Chapter 6 bring up the scheduling algorithms of the ST and DYN segments, for the simple and switch FlexRay networks respectively. By analyzing the schedulable constraints and configurable parameters, we determine the methods to generate the ST scheduling table which depends on the characteristics of the input messages. For DYN communication, the transmission is event triggered. The DYN scheduler is an online scheduler. It produces the responses or outputs based on the runtime inputs. Therefore, it does not have a ST scheduling table. As a result, the DYN scheduler without runtime inputs only can calculate the worst-case response time to estimate the conservative system schedulability.

Chapter 7 evaluates the performances of the four schedulers. The results for the simple and switch FlexRay network were compared. It is obvious that the switch FlexRay network has significant improvement in system schedulability. Meanwhile, it decreases the resource utility rate. Therefore, the conclusion we drew from the performance evaluations is that the switch FlexRay network increases the system capacity and efficiency. It is the trend in the future.

## 8.2 Contributions and Future Work

This section summaries the contributions and concludes this thesis by suggesting some ideas for future improvements.

The main contribution of this thesis is the developments of the scheduling algorithms for the simple and switch FlexRay networks. By using these schedulers, we are able to analyze the schedulabilities and upper bound of automotive networks. The results generated from the schedulers are very helpful in the designing step of automotive systems.

From all the discussions and analyses in this thesis, it is clear to form the concept about FlexRay protocol and the concept of clustering in switch FlexRay network, also to understand about the factors that affect the worst-case response time of the DYN messages. However, there are still spaces of the improvements. The following sections provide some of the suggestions for future works:

- Test the schedulers with hardware implementations;

  The scheduling algorithms are only on the conceptual level, with the results from the computer simulations. The schedulers provided in this thesis have not yet been experimentally proved. For future developments, it is suggested that combine the software with hardware implementations to evaluate the performances of the schedulers, including schedulability, efficiencies and system utilization. The evaluations should conduct under different system configurations and network loads.

- Holistic analysis of the worst-case response time for DYN messages;

  To simplify the calculation, this thesis uses the heuristic solutions to calculate the worst-case response time. By calculating the communication latency with holistic analyses, we can get more accurate system schedulabilities.

- Refine the clusters of the switched DYN messages by using flexible regrouping period;

  The discussion in Section 6.3.2.1 shows that there exists a regrouping time of the DYN clusters. This thesis uses the duration of a CC as the regrouping period. More multiplexing in the system undoubtedly increases system bandwidth. By refining the communication clusters, the utility rate of the system resource will increase. Reducing the regrouping period, which should base on the transmission data, can refine the clusters.

- Increase the accuracy of the schedulability analysis in DYN segment;

  This thesis calculates the worst-case response time of the DYN messages and then compares with the messages deadlines. The comparison results indicate the system schedulability. However, this is only a conservative estimation for the system schedulability. In order to generate a DYN scheduling table for an application, the parameters extracted from the ECUs, such as clock times, should input to the scheduler presented in Appendix E. The input data affects the generated DYN scheduling table. Consequently, we can get a more accurate evaluation of the system schedulability.

# Bibliography

[1] G. Leen, and D. Heffernan, “Expanding automotive electronic systems,” *Computer,* vol. 35, no. 1, pp. 88-93, 2002.

[2] "Automotive Electronic Systems," The Clemson University Vehicular Electronics Laboratory.

[3] A. Albert, “Comparison of event-triggered and time-triggered concepts with regards to distributed control systems,” *Embedded World,* vol. 2, pp. 235-252, 2004.

[4] G. Leen, D. Heffernan, and A. Dunne, “Digital networks in the automotive vehicle,” *Computing & Control Engineering Journal,* vol. 10, no. 6, pp. 257-266, 1999.

[5] "Two Msc. Thesis Positions:Reliable In-Vehicle FlexRay Network Scheduler Design."

[6] "LIN Specification Package, Revision 2.0," LIN Consortium September 2003.

[7] "Road Vehicles—Low Speed Serial Data Communication—Part 2: Low Speed Controller Area Network," ISO 11 519-2, 1994.

[8] "Road Vehicles—Interchange of Digital Information—Controller Area Network for High-Speed Communication," ISO 11 898, 1994.

[9] N. Navet, Y. Song, F. Simonot-Lion *et al.*, “Trends in Automotive Communication Systems,” *Proceedings of The IEEE,* vol. 93, no. 6, pp. 1204-1223, 2005.

[10] "Time-Triggered Protocol TTP/C, High-Level Specification Document, Protocol Version 1.1.," TTTech Computertechnik GmbH., November, 2003.

[11] F. Consortium, "FlexRay Communication System, Protocol Specification, Version 2.1 Rev.A," FlexRay Consortium, 12 December 2005.

[12] "Road Vehicles—Controller Area Network (CAN)—Part 4: Time-Triggered Communication," ISO 11 898-4, 2000.

[13] Wikipedia. "FlexRay," http://en.wikipedia.org/wiki/FlexRay.

[14] "FlexRay Overview," F. N. A. F. B. a. N. T. A. Modeling, ed., Mirabilis Design Inc., 2006.

[15] S. Kosov, *FlexRay communication protocol (Wakeup and Startup)*, Institute for Computer Architecture and Parallel Computing, Universitat Des Saarlandes, 2005.

[16] Wikipedia. "Electronic control unit," http://en.wikipedia.org/wiki/Electronic_control_unit.

[17] "FlexRay Electrical Physical Layer Specification V2.1 Rev.B," FlexRay Consortium, November 2006, p. 27.

[18] Wikipedia. "Time division multiple access," http://en.wikipedia.org/wiki/Time_division_multiple_access.

[19] S. J. Young, *Real time languages: Design and development*, New York: Halsted Press, 1982.

[20] Wikipedia. "Real-time operating system," http://en.wikipedia.org/wiki/Real-time_operating_system.

[21] Wikipedia. "Preemption (computing)," http://en.wikipedia.org/wiki/Preemption_%28computing%29.

[22] T. Nolte, "Share-Driven Scheduling of Embedded Networks," Department of Computer Science and Electronics, Malardalen University, Västerås, Sweden, 2006.

[23] J. Leung, and H. Zhao, *Real-Time Scheduling Analysis,* DOT/FAA/AR-05/27, New Jersey Institute of Technology, Newark, NJ, 2005.

[24] C. L. Liu, and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the Association for Computing Machinery,* vol. 20, no. 1, pp. 40-61, January, 1973.

[25] J. Y. T. Leung, and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic real-time tasks," *Performance Evaluation,* vol. 2, no. 4, pp. 237-250, December, 1982.

[26] M. L. Dertouzos, "Control Robotics: The Procedural Control of Physical Processes.," in IFIP Congres, Stockholm, Sweden, 1974, pp. 807-81.

[27] A. K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment," Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1983.

[28] TTTech, "Welcome to FlexRay," TTTech Computertechnik AG, 2009.

[29] K. Tindell, A. Burns, and A. Wellings, "An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks," *Real-Time Systems,* vol. 6, no. 1, pp. 133-151, March, 1994.

[30] N. Audsley, A. Burns, M. Richardson *et al.*, "Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling," *Software Engineering Journal,* vol. 8, no. 5, pp. 284-292, September, 1993.

[31] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronisation," *IEEE Transactions on Computers,* vol. 39, no. 9, pp. 1175-1185, September, 1990.

[32] K. Tindell, and J. Clark, “Holistic Schedulability Analysis for Distributed Hard Real-Time Systems,” *Parallel Embedded Real-Time Systems*, no. Micrprocessors and Microprogramming, March, 1994.

[33] M. Lukasiewycz, M. Glaß, P. Milbredt *et al.*, "FlexRay Schedule Optimization of the Static Segment," *Proceedings of the 7th International Conference on Hardware/Software Codesign and System Synthesis.* pp. 363-372.

[34] P. Pop, P. Eles, Z. Peng *et al.*, “Analysis and Optimization of Distributed Real-Time Enbedded Systems,” *ACM Transactions on Design Automation of Electronic Systems,* vol. 11, no. 3, pp. 593-625, July, 2006.

[35]T. Pop, P. Pop, P. Eles *et al.*, “Timing Analysis of the FlexRay Communication Protocol,” in 18th Euromicro Conference on Real-Time Systems, Dresden, Germany, 2006.

[36] Wikipedia. "Bin packing problem," http://en.wikipedia.org/wiki/Bin_packing_problem#First-fit_algorithm.

[37] M. Labbé, G. Laporte, and S. Martello, “An exact algorithm for the dual bin packing problem,” *Operations Research Letters,* vol. 17, no. 1, pp. 9-18, February, 1995.

[38] H2G2. "Packing Algorithms," http://www.bbc.co.uk/dna/h2g2/A954722.

[39] JRank. "Science Encyclopedia: Mathematics—first-fit decreasing packing algorithm," http://science.jrank.org/pages/50761/first-fit-decreasing-packing-algorithm.html.

[40] JRank. "Science Encyclopedia: Mathematics —First -fit packing algorithm," http://science.jrank.org/pages/50760/first-fit-packing-algorithm.html.

[41] M. Grenier, L. Havet, and N. Navet, “Configuring the communication on FlexRay – the case of the static segment,” in 4th European Congress on Embedded Real Time Software (ERTS 2008), Toulouse, France, 2008.

[42] P. Milbredt, B. Vermeulen, G. o. Tabanoglu *et al.*, "Switched FlexRay: Increasing the Effective Bandwidth and Safety of FlexRay Networks."

[43] A. Schedl, “Goals and architecture of FlexRay at BMW,” in Vector FlexRay Symposium, Stuttgart, Germany, 2007.

[44] M. Peteratzinger, F. Steiner, and R. Schuermans, “Use of XCP on FlexRay at BMW,” *HANSER automotive*, September, 2006.

# A

# Source Code of Simple ST Scheduler

```
1  /*-----------------------------------------------------------------
               Simple FlexRay ST scheduling algorithm
   -----------------------------------------------------------------*/
2  #include <iostream>
3  #include <iomanip>
4  #include <vector>
5  #include <set>
6  #include <algorithm>
7  #include <time.h>
8  #include <math.h>
9
10 using namespace std;
11
12 #define max(a,b) (((a) > (b)) ? (a) : (b))
13
14 //-----------------System parameters-------------------
15 const int pSamplesPerMicrotick = 1;//Number of samples per
   microtick
16 const double gdSampleClockPeriod = 0.0125;//Sample clock
   period[µs]
17 double pdMicrotick;//Duration of a microtick[µs]
18 double gdMaxMicrotick;//Maximum microtick length of all
   microticks configured within a cluster[µs]
19 double gdMacrotick = 2;// Duration of the cluster wide nominal
   macrotick[µs]
20 double gdBit, gdBitMax, gdBitMin; // gdBit:Nominal bit time[µ
   s], gdBitMax[µs], gdBitMin[µs]
21 int gdActionPointOffset;//Number of macroticks the action point
   is offset from the beginning of a static slot or symbol
   window[MT]
22 double gAssumedPrecision;//Assumed precision of the application
   network[µs];  aBestCasePrecision[µs] <= gAssumedPrecision[µs]
   <= aWorstCasePrecision[µs]=(34µT + 20 * gClusterDriftDamping[µ
   T]) * gdMaxMicrotick[µs /µT] +2 * gdMaxPropagationDelay
```

```
23 double aBestCasePrecision, aWorstCasePrecision;
24 int gdTSSTransmitter; //Number of bits in the Transmission Start
   Sequence[gdBit]
25 double gdMaxInitializationError;//maximum initialization
   error[µs]; 2 * (gdMaxMicrotick[µs] * (1 + 0.0015)) +
   gdMaxPropagationDelay <= gdMaxInitializationError<=
   gAssumedPrecision= aWorstCasePrecision= 11.7 µs
26 int gClusterDriftDamping = 5;//cluster drift damping factor=0~
   5[']
27 double dStarTruncation = 0.45; //Interval by which the
   transmission of a frame is shortened by one star = 0.45[µs]
28 double dBDRxia = 0.3;//Activity reaction time. Time by which a
   transmission becomes shortened in a receiving node = 0 ~ 0.45[µ
   s];It is the truncation that occurs in the BD of the receiving
   node. It is present even if the frame does not pass through any
   active stars.
29 int nStarPath = 0;//0~2; the maximum number of active stars
   between any two nodes
30 double gdMinPropagationDelay = 0, gdMaxPropagationDelay =
   2.5;//A minimum/maximum propagation delay of the network as seen
   by the local node[µs], 0 ~ 2.5
31
32 double BusSpeed = 10;//Bus data rate[Mbit/s]
33 double gdCycle = 5000, STbus = 3000;//gdCycle:CC length[µs],
   STbus:ST segment length[µs]
34
35
36 //---------------ST segment parameters----------------
37 int num_ST;//number of ST messages
38 double gdStaticSlot; //ST-Slot length[µs]
39 int gNumberOfStaticSlots, gNumberOfStaticSlots_min,
   gNumberOfStaticSlots_use = 0;//gNumberOfStaticSlots_min: minimum
   required number of ST slots;gNumberOfStaticSlots_use:used number
   of ST slots
40 double gPayloadLengthStatic;//payload length of the ST
   frame[bit]
41 int aFrameLengthStatic;//ST Frame length[dBit]
42 int scheduledCount = 0; //counter of scheduled messages
43 bool Vuse[1023][64] = {0}; //used vector space
44
45
46 struct buf
47 {
48     int x;
49     int y;
50 };
51
52 struct MessageST
53 {
54     int FrameID_ST;//FrameID of ST messages
55     int MessageLengthST; // MessageLengthST[bit]
56     double T_ST; //Period of production of static message [µs]
57     double D_ST; //Deadline of static message [µs]
58
59     int s; //base slot
60     int p; //slot repetition
```

```
61      int gNumberOfStaticSlot_m;//Number of ST slots for m
62      int b; //base cycle
63      int r; //cycle repetition
64      int gNumberOfCycle;//Number of cycles allocated to
65
66      bool buffer[1023][64];//the slots m's transmission occupied
67      bool V[1023][64];
68      bool isScheduled;
69
70      int SlotIDMax; //Max slot ID
71
72      buf SlotID;
73      buf CC;
74  };
75
76
77  int main()
78  {
79   /*-------------------------------------------------------------
                              user input
     -------------------------------------------------------------*/
80      /*cout << "Please choose the duration of the samples
    clock[μs]: 0.0125, 0.0250, 0.0500";
81      cin >> gdSampleClockPeriod;
82      cout << "Please choose the number of samples per Microtick:
    1,2,4";
83      cin >> pSamplesPerMicrotick;*/
84      cout << "Please input the number of ST messages:";
85      cin >> num_ST;
86      /*cout << "Please input the maximum number of active stars
    between any 2 nodes:";
87      cin >> nStarPath;
88      cout << "Please input the FlexRay bus bit rate [MBit/s]:";
89      cin >> BusSpeed;
90      cout << "Please input the min bit rate of the ST messages
    [MBit/s]:";
91      cin >> min_bitRate;
92      cout << "Please input the max bit rate of the ST messages
    [MBit/s]:";
93      cin >> max_bitRate;*/
94
95      srand((unsigned)time(NULL));
96      for (int j = 0; j < 10; j++)
97      {
98          rand();
99      }
100
101     vector<MessageST> MST;
102     vector<MessageST> Lst;
103     vector<MessageST>::iterator iterST;
104
105  /*-------------------------------------------------------------
                    value the system parameters
     -------------------------------------------------------------*/
106     if (BusSpeed == 2.5)
107     {
108         gdBit = 0.4;
```

```
109         gdBitMax = 0.4006;
110         gdBitMin = 0.3994;
111     }
112     if (BusSpeed == 5)
113     {
114         gdBit = 0.2;
115         gdBitMax = 0.2003;
116         gdBitMin = 0.1997;
117     }
118     if (BusSpeed == 10)
119     {
120         gdBit = 0.1;
121         gdBitMax = 0.10015;
122         gdBitMin = 0.09985;
123     }
124 
125     pdMicrotick = pSamplesPerMicrotick * gdSampleClockPeriod;
126     gdMaxMicrotick = pdMicrotick;
127     aWorstCasePrecision=(34    +  20  *  gClusterDriftDamping)  *
    gdMaxMicrotick +2 * gdMaxPropagationDelay;
128 
129     gAssumedPrecision = aWorstCasePrecision;
130     gdMaxInitializationError  =  2  *  (gdMaxMicrotick*  (1  +
    0.0015)) + gdMaxPropagationDelay;
131 
132     gdTSSTransmitter  =  ceil(  (gdBitMax+  dBDRxia+  nStarPath  *
    dStarTruncation)/gdBitMin);
133     gdActionPointOffset     =     ceil((2*gAssumedPrecision     -
    gdMinPropagationDelay    +2    *    gdMaxInitializationError)    /
    (gdMacrotick* (1 - 0.0015)));
134 //------------------output message info-------------------

135     cout << "ID_ST" << '\t' << "ML_ST" << '\t' << "T_ST" << '\t'
    << "D_ST" << '\t' << endl;
136 
137 //-----------------messages input parameters-----------

138     int ID = 1;
139     for (int n = 0; n < num_ST; n++)
140     {
141         MessageST *temp = new MessageST;
142         temp->FrameID_ST = ID++;
143 
144         //-------------random T generation--------------
145         int a = rand() % 11;
146         if (a == 0)
147         {
148             temp->T_ST = 5000;
149         }
150         else if (a == 1)
151         {
152             temp->T_ST = 10010;
153         }
154         else if (a == 2)
155         {
156             temp->T_ST = 20010;
157         }
158         else if (a == 3)
```

```
159         {
160             temp->T_ST = 40010;
161         }
162         else if (a == 4)
163         {
164             temp->T_ST = 80010;
165         }
166         else if (a == 5)
167         {
168             temp->T_ST = 160010;
169         }
170         else if (a == 6)
171         {
172             temp->T_ST = 320010;
173         }
174         else if (a == 7)
175         {
176             temp->T_ST = 4000;
177         }
178         else if (a == 8)
179         {
180             temp->T_ST = 2000;
181         }
182         else if (a == 9)
183         {
184             temp->T_ST = 1000;
185         }
186         else if (a == 10)
187         {
188             temp->T_ST = 500;
189         }
190 
191         temp->D_ST = temp->T_ST;
192         if (temp->D_ST > 40000)
193         {
194             temp->D_ST = 40000;
195         }
196 
197         temp->MessageLengthST = 64;
198 
199         for (int i = 0; i <= 1022; i++)
200         {
201             for (int j = 0; j <= 63; j++)
202             {
203                 temp->buffer[i][j] = 0;
204                 temp->V[i][j] = 1;
205             }
206         }
207         temp->isScheduled = false;
208         MST.push_back(*temp);
209         cout << setw(3) << temp->FrameID_ST << '\t' << temp-
    >MessageLengthST << '\t' << temp->T_ST << '\t' << temp->D_ST <<
    '\t' << endl;
210     }
211 
212  /*-------------------------------------------------------------
            value the message schedule's main parameters
213 -------------------------------------------------------------*/
```

```
214     int MessageLengthSTMax;
215     for (iterST = MST.begin(); iterST != MST.end(); iterST++)
216     {
217         if (MessageLengthSTMax < iterST->MessageLengthST)
218             MessageLengthSTMax = iterST->MessageLengthST;
219     }
220
221     gPayloadLengthStatic = MessageLengthSTMax;
222     aFrameLengthStatic    =    (gdTSSTransmitter+    1    +    80
    +gPayloadLengthStatic* 1.25 + 2);//gdBit
223
224     gdStaticSlot     =     2     *     gdActionPointOffset     +
    ceil(                    ((aFrameLengthStatic                    +
    11)*gdBitMax+gdMinPropagationDelay+gdMaxPropagationDelay)/(gdMac
    rotick *(1-0.0015)));
225     gNumberOfStaticSlots = STbus / gdStaticSlot;
226
227         for   (iterST   =   MST.begin();   iterST   !=   MST.end();
    iterST++)
228         {
229             if (iterST->D_ST < gdCycle)
230             {
231                 iterST->r = 1;
232                 for (int n = 0; n <= log(gdCycle / gdStaticSlot)
    / log(2); n++)
233                     {
234                         if    (iterST->p    <=    iterST->D_ST    /
    gdStaticSlot)
235                         {
236                             iterST->p = pow(2,n);
237                         }
238                     }
239                 iterST->gNumberOfStaticSlot_m                    =
    ceil((double)gNumberOfStaticSlots / iterST->p);
240                 iterST->gNumberOfCycle = 64;
241             }
242             else
243             {
244                 iterST->p = gNumberOfStaticSlots;
245                 iterST->gNumberOfStaticSlot_m = 1;
246                 if (iterST->D_ST > gdCycle)
247                 {
248                     for (int n = 0; n <= 6; n++)
249                     {
250                         if (iterST->r <= iterST->D_ST / gdCycle)
251                         {
252                             iterST->r = pow(2,n);
253                         }
254                     }
255                     iterST->gNumberOfCycle = 64 / iterST->r;
256                 }
257                 else
258                 {
259                     iterST->r = 1;
260                     iterST->gNumberOfCycle = 64;
261                 }
262             }
263         }
```

```
264
265 //------------------basic  constraint----------------------
266         for  (iterST  =  MST.begin();  iterST  !=  MST.end();
    iterST++)
267         {
268             gNumberOfStaticSlots_min      +=      (double)iterST-
    >gNumberOfStaticSlot_m/iterST->r;
269         }
270         gNumberOfStaticSlots_min                                    =
    ceil((double)gNumberOfStaticSlots_min);
271         if (gNumberOfStaticSlots_min <= 2)
272         {
273             gNumberOfStaticSlots_min = 2;
274         }
275         if (gNumberOfStaticSlots_min > gNumberOfStaticSlots)
276         {
277             cout << "non-schedulable ,gNumberOfStaticSlots_min>
    gNumberOfStaticSlots" << endl;
278             return 0;
279         }
280 //------------------sort-------------------------------
281         while (MST.size() != 0)
282         {
283             vector<MessageST>::iterator temp = MST.begin();
284             for  (iterST  =  MST.begin();  iterST  !=  MST.end();
    iterST++)
285             {
286                 if (iterST->p < temp->p || (iterST->p == temp->p
    && iterST->r < temp->r))
287                 {
288                     temp = iterST;
289                 }
290             }
291             Lst.push_back(*temp);
292             MST.erase(temp);
293         }
294 //------------------scheduling process------------------
295         for  (iterST  =  Lst.begin();  iterST  !=  Lst.end();
    iterST++)
296         {
297             if (iterST->isScheduled)
298             {
299                 continue;
300             }
301             iterST->SlotID.x = 0;
302             iterST->SlotID.y = iterST->p - 1;
303             iterST->CC.x = 0;
304             iterST->CC.y = iterST->r - 1;
305
306             for  (int  i  =  iterST->SlotID.x;  i  <=  iterST-
    >SlotID.y; i++)
307             {
308                 for (int j = iterST->CC.x; j <= iterST->CC.y;
    j++)
309                 {
310                     if (Vuse[i][j] == 0) //space not be used
```

```
311                     {
312                         iterST->V[i][j] = 0;
313                     }
314                 }
315             }
316             for (int sm = iterST->SlotID.x; sm <= iterST->SlotID.y; sm++)
317             {
318                 if (iterST->isScheduled)
319                 {
320                     break;
321                 }
322                 for (int bm = iterST->CC.x; bm <= iterST->CC.y; bm++)
323                 {
324                     if (iterST->isScheduled)
325                     {
326                         break;
327                     }
328                     if (iterST->V[sm][bm] == 0)
329                     {
330                         iterST->s = sm;
331                         iterST->b = bm;
332                         for (int m = 1; m <= iterST->gNumberOfStaticSlot_m; m++)
333                         {
334                             for (int n = 1; n <= iterST->gNumberOfCycle; n++)
335                             {
336                                 iterST->buffer[iterST->s+(m-1)*iterST->p][iterST->b+(n-1)*iterST->r] = 1;
337                             }
338                         }
339                         for (int k = 0; k <= 1022; k++)
340                         {
341                             for (int j = 0; j <= 63; j++)
342                             {
343                                 Vuse[k][j] = Vuse[k][j] | iterST->buffer[k][j]; //update Vuse
344                             }
345                         }
346 
347                         iterST->SlotIDMax = iterST->s+1+(iterST->gNumberOfStaticSlot_m - 1)*iterST->p;
348 
349                         if ( iterST->SlotIDMax > gNumberOfStaticSlots_use)
350                         {
351                             gNumberOfStaticSlots_use= iterST->SlotIDMax; //update used max slot ID
352                         }
353                         if (gNumberOfStaticSlots_use > gNumberOfStaticSlots)
354                         {
355                             cout << "system non-schedulable. gNumberOfStaticSlots_use > gNumberOfStaticSlots." << endl;
356                             return 0;
357                         }
```

```
358
359
360                         iterST->isScheduled = true;
361                         scheduledCount++;
362                         if (scheduledCount == Lst.size())
363                         {
364                             cout << "scheduled" << endl;
365                             cout    <<    "gdStaticSlot:"    <<
gdStaticSlot << endl;
366                             cout  <<  "gNumberOfStaticSlots:"  <<
gNumberOfStaticSlots << endl;
367                             cout << "number of ST slots used:"
<< gNumberOfStaticSlots_use << endl;
368 //-----------------output ST schedule----------------------

369                             vector<MessageST>::iterator
iterSTOut;
370                             cout << "ID_ST" << '\t' << "s" <<
'\t' << "p" << '\t' << setw(8) << "STSlotNum";
371                             cout << setw(5) << "b" << setw(5) <<
"r" << '\t' << "CycNum" << '\t' << "SlotIDMax" << '\t' << endl;
372                             for   (iterSTOut   =   Lst.begin();
iterSTOut != Lst.end(); iterSTOut++)
373                             {
374                                 cout  <<  setw(3)  <<  iterSTOut-
>FrameID_ST << '\t' << iterSTOut->s + 1 << '\t' << iterSTOut->p
<< '\t' << setw(4) << iterSTOut->gNumberOfStaticSlot_m;
375                                 cout << setw(10) << iterSTOut->b
<< setw(5) << iterSTOut->r << '\t' << setw(4) << iterSTOut-
>gNumberOfCycle << setw(9) << iterSTOut->SlotIDMax << '\t' <<
endl;
376                             }
377 //-------------------------------------------------------------

378                             return 0;
379                         }
380                         else
381                         {
382                             break;
383                         }
384                     }
385                 }
386             }
387         }
388     cout   <<   "system   non-schedulable.   Schedulability:"   <<
(double)scheduledCount/Lst.size()*100 << "%" << endl;
389     cout << "gdStaticSlot:" << gdStaticSlot << endl;
390     cout  <<  "gNumberOfStaticSlots:"  <<  gNumberOfStaticSlots  <<
endl;
391     return 0;
392 }
```

# B

# Source Code of Simple DYN Scheduler

```
1  /*-------------------------------------------------------------
             Simple FlexRay DYN scheduling algorithm
   -------------------------------------------------------------*/
2  #include <iostream>
3  #include <iomanip>
4  #include <vector>
5  #include <set>
6  #include <algorithm>
7  #include <time.h>
8  #include <math.h>
9
10 using namespace std;
11
12 //----------------System parameters-------------------
13 const double gdMacrotick = 2;//Duration of the cluster wide
   nominal macrotick[μs]
14 double gdTSSTransmitter;//Number of bits in the Transmission
   Start Sequence[gdBit]
15 double gdBit, gdBitMax, gdBitMin; // gdBit:Nominal bit time[μ
   s], gdBitMax[μs], gdBitMin[μs]
16 double dStarTruncation = 0.45; //Interval by which the
   transmission of a frame is shortened by one star = 0.45[μs]
17 double dBDRxia = 0.3;//Activity reaction time. Time by which a
   transmission becomes shortened in a receiving node = 0 ~ 0.45[μ
   s];It is the truncation that occurs in the BD of the receiving
   node. It is present even if the frame does not pass through any
   active stars.
18 int nStarPath = 0;//0~2; the maximum number of active stars
   between any two nodes
19
20 double BusSpeed = 10;//Bus data rate[Mbit/s]
21 double gdCycle = 5000, STbus = 3000, DYNbus;//gdCycle:CC
   length[μs], STbus:ST segment length[μs], DYN bus length[μs]
22
23
```

```
24 //---------------DYN segment parameters----------------
25 int num_DYN;//number of DYN messages
26 double gdMinislot;//Minislot length[μs]
27 int gNumberOfMinislots;
28 int scheduledCountDYN = 0;
29 int pLatestTx;//Number of the last minislot in which a frame
   transmission can start in the dynamic segment
30 double aFrameLength_DYN;//DYN frame length
31
32
33 struct MessageDYN
34 {
35     int FrameID_DYN; //FrameID
36     int MessageLengthDYN;// MessageLengthDYN[bit]
37     double FrameLengthDYN; //DYN frame length
38     double D_DYN;//Deadline of dynamic message [μs]
39     double R_DYN; //response time[μs]
40 };
41
42
43 int main()
44 {
45 //-------------------user input---------------------------
46     cout << "Please input the number of DYN messages:";
47     cin >> num_DYN;
48     gdMinislot = 5 * gdMacrotick;
49     /*cout << "Please input the length of a Minislot:";
50     cin >> gdMinislot;*/
51
52     srand((unsigned)time(NULL));
53     for (int j = 0; j < 10; j++)
54     {
55         rand();
56     }
57
58     vector<MessageDYN> MDYN;
59     vector<MessageDYN> Ldyn;
60     vector<MessageDYN>::iterator iterDYN;
61
62     vector<MessageDYN> lf;
63     vector<MessageDYN> lf_sort;
64     vector<MessageDYN>::iterator iterlf;
65     vector<MessageDYN>::iterator iterMessageLength;
66
67
68     if (BusSpeed == 2.5)
69     {
70         gdBit = 0.4;
71         gdBitMax = 0.4006;
72         gdBitMin = 0.3994;
73     }
74     if (BusSpeed == 5)
75     {
76         gdBit = 0.2;
77         gdBitMax = 0.2003;
78         gdBitMin = 0.1997;
79     }
```

```
80      if (BusSpeed == 10)
81      {
82          gdBit = 0.1;
83          gdBitMax = 0.10015;
84          gdBitMin = 0.09985;
85      }
86
87  //----------------system parameters-----------------------
88      gdTSSTransmitter = ceil( (gdBitMax+ dBDRxia+ nStarPath *
    dStarTruncation)/gdBitMin);
89      DYNbus = gdCycle - STbus;
90      gNumberOfMinislots = DYNbus / gdMinislot;//int = floor()
91  //-------------messages input parameters--------------------
92      for (int m = 0; m < num_DYN; m++)
93      {
94          MessageDYN *temp = new MessageDYN;
95
96          //-------------random D generation-------------------
97          int a = rand() % 7;
98          if (a == 0)
99          {
100             temp->D_DYN = 10000;
101         }
102         else if (a == 1)
103         {
104             temp->D_DYN = 20000;
105         }
106         else if (a == 2)
107         {
108             temp->D_DYN = 50000;
109         }
110         else if (a == 3)
111         {
112             temp->D_DYN = 100000;
113         }
114         else if (a == 4)
115         {
116             temp->D_DYN = 200000;
117         }
118         else if (a == 5)
119         {
120             temp->D_DYN = 1000000;
121         }
122         else if (a == 6)
123         {
124             temp->D_DYN = 2000000;
125         }
126
127         temp->MessageLengthDYN = rand() % gNumberOfMinislots *
    BusSpeed * gdMinislot;//maximum DYN data supported, changeable
128         MDYN.push_back(*temp);
129     }
130
131 //---------------sort the DYN messages----------------------
132     int ID = 1;
133     while (MDYN.size() != 0)
134     {
135         vector<MessageDYN>::iterator temp = MDYN.begin();
```

```
136         for (iterDYN = MDYN.begin(); iterDYN != MDYN.end();
    iterDYN++)
137         {
138             if (iterDYN->MessageLengthDYN/iterDYN->D_DYN > temp-
    >MessageLengthDYN/temp->D_DYN)
139             {
140                 temp = iterDYN;
141             }
142         }
143         temp->FrameID_DYN = ID++;
144         Ldyn.push_back(*temp);
145         MDYN.erase(temp);
146     }
147 //-----------------sort end----------------------------
148 
149     cout << "FrameID" << setw(8) << "ML_DYN" << setw(11) <<
    "D_DYN[ms]" << setw(12) << "R_DYN[ms]" << setw(18) <<
    "scheduled?" << '\t' << endl;
150 //---------------worst-case reponse time-----------------------
151     for (iterDYN = Ldyn.begin(); iterDYN != Ldyn.end();
    iterDYN++)
152     {
153         aFrameLength_DYN = gdTSSTransmitter + 83 + iterDYN-
    >MessageLengthDYN * 1.25;//gdBit
154         pLatestTx = gNumberOfMinislots - (aFrameLength_DYN *
    gdBit) / gdMinislot;
155 
156    //--------construct lf(m) list---------------------
157         lf_sort.clear();//initiate vector lf_sort for the next
    m_DYN
158         int FrameID_m = iterDYN->FrameID_DYN;
159         for (iterlf = Ldyn.begin(); iterlf!= Ldyn.end();
    iterlf++)
160         {
161             if (iterlf->FrameID_DYN < FrameID_m)
162             {
163                 bool hasSameID = false;
164                 for (iterMessageLength = lf.begin();
    iterMessageLength!= lf.end(); iterMessageLength++)
165                 {
166                     if(iterMessageLength->FrameID_DYN == iterlf-
    >FrameID_DYN)
167                     {
168                         hasSameID = true;
169                         if (iterMessageLength->MessageLengthDYN
    < iterlf->MessageLengthDYN)//check the same FrameID message size
170                         {
171                             iterMessageLength->MessageLengthDYN
    = iterlf->MessageLengthDYN;
172                             iterMessageLength->FrameID_DYN =
    iterlf->FrameID_DYN;
173                             iterMessageLength->D_DYN = iterlf-
    >D_DYN;
174                             iterMessageLength->R_DYN = iterlf-
    >R_DYN;
175                         }
176                     }
177                 }
```

```
178                 if (!hasSameID)
179                 {
180                     lf.push_back(*iterlf);//put into vector
181                 }
182             }
183             else
184             {
185                 continue;
186             }
187         }
188 
189 //----------sort the lf(m) in decreasing order--------------
190         while (lf.size() != 0)
191         {
192             vector<MessageDYN>::iterator temp = lf.begin();
193             for (iterlf= lf.begin(); iterlf != lf.end();
    iterlf++)
194             {
195                 if (iterlf->MessageLengthDYN > temp-
    >MessageLengthDYN)
196                 {
197                     temp = iterlf;
198                 }
199             }
200             lf_sort.push_back(*temp);
201             lf.erase(temp);
202         }
203 
204   /*-------------------------------------------------------
          first fit bin packing calculates BusCycles_lf
    -------------------------------------------------------*/
205         double bin[64];
206         for (int n = 1; n <= 64; n++)
207         {
208             bin[n] = DYNbus;
209         }
210         int BusCycles_lf = 1;
211         for (iterlf= lf_sort.begin(); iterlf != lf_sort.end();
    iterlf++)
212         {
213             iterlf->FrameLengthDYN = gdTSSTransmitter + 83 +
    iterlf->MessageLengthDYN * 1.25;
214             for (n = 1; n <= 64; n++)//searching all avilable
    bins
215             {
216                     if ( bin[n] >= iterlf->FrameLengthDYN &&
    bin[n] >= DYNbus-pLatestTx * gdMinislot)//enough space, put into
    this bin
217                 {
218                     if ( n > BusCycles_lf)//update BusCycles_lf
219                     {
220                         BusCycles_lf = n; //determine the number
    of cycles occupied by lf messages
221                     }
222                     bin[n] -= iterlf->FrameLengthDYN;//update
    the spare space of bin[n]
223                     break;
224                 }
```

```
225                 else
226                 {
227                     continue;//search next bin
228                 }
229             }
230         }
231 
232 //---------------worst-case response time--------------------
233         iterDYN->R_DYN = (gdCycle-(STbus+(iterDYN->FrameID_DYN-
    1)*gdMinislot))+ BusCycles_lf *gdCycle + (iterDYN->FrameID_DYN-
    1)*gdMinislot + (STbus+pLatestTx*gdMinislot)+(aFrameLength_DYN
    *gdBit/BusSpeed);
234         if (iterDYN->R_DYN <= iterDYN->D_DYN)
235         {
236             scheduledCountDYN++;
237             cout << setw(4) << iterDYN->FrameID_DYN << setw(11)
    << iterDYN->MessageLengthDYN << setw(9) << iterDYN->D_DYN/1000
    << setw(13) << iterDYN->R_DYN/1000 << setw(15) << "Yes" << '\t'
    << endl;
238             if (scheduledCountDYN == Ldyn.size())
239             {
240                 cout << "System schedulable. DYN messages are
    100% scheduled." << endl;
241                 return 0;
242             }
243         }
244         else
245         {
246             cout << setw(4) << iterDYN->FrameID_DYN << setw(11)
    << iterDYN->MessageLengthDYN << setw(9) << iterDYN->D_DYN/1000
    << setw(13) << iterDYN->R_DYN/1000 << setw(15) << "No" << '\t'
    << endl;
247         }
248     }
249     cout << "system non-schedulable. Schedulability:" <<
    (double)scheduledCountDYN/Ldyn.size()*100 << "%"<< endl;
250     return 0;
251 }
```

# C

# Source Code of Switched ST Scheduler

```
1  /*-------------------------------------------------------------
             Switched FlexRay ST scheduling algorithm
   -------------------------------------------------------------*/
2  #include <iostream>
3  #include <iomanip>
4  #include <vector>
5  #include <set>
6  #include <algorithm>
7  #include <time.h>
8  #include <math.h>
9
10 using namespace std;
11
12 #define max(a,b) (((a) > (b)) ? (a) : (b))
13
14 //----------------System parameters-------------------
15 const int pSamplesPerMicrotick = 1;//Number of samples per
   microtick
16 const double gdSampleClockPeriod = 0.0125;//Sample clock
   period[µs]
17 double pdMicrotick;//Duration of a microtick[µs]
18 double gdMaxMicrotick;//Maximum microtick length of all
   microticks configured within a cluster[µs]
19 double gdMacrotick = 2;// Duration of the cluster wide nominal
   macrotick[µs]
20 double gdBit, gdBitMax, gdBitMin; // gdBit:Nominal bit time[µ
   s], gdBitMax[µs], gdBitMin[µs]
21 int gdActionPointOffset;//Number of macroticks the action point
   is offset from the beginning of a static slot or symbol
   window[MT]
22 double gAssumedPrecision;//Assumed precision of the application
   network[µs];  aBestCasePrecision[µs] <= gAssumedPrecision[µs]
   <= aWorstCasePrecision[µs]=(34µT + 20 * gClusterDriftDamping[µ
   T]) * gdMaxMicrotick[µs /µT] +2 * gdMaxPropagationDelay
```

```
23 double aBestCasePrecision, aWorstCasePrecision;
24 int gdTSSTransmitter; //Number of bits in the Transmission Start
   Sequence[gdBit]
25 double gdMaxInitializationError;//maximum initialization
   error[µs]; 2 * (gdMaxMicrotick[µs] * (1 + 0.0015)) +
   gdMaxPropagationDelay <= gdMaxInitializationError<=
   gAssumedPrecision= aWorstCasePrecision= 11.7 µs
26 int gClusterDriftDamping = 5;//cluster drift damping factor=0~
   5[']
27 double dStarTruncation = 0.45; //Interval by which the
   transmission of a frame is shortened by one star = 0.45[µs]
28 double dBDRxia = 0.3;//Activity reaction time. Time by which a
   transmission becomes shortened in a receiving node = 0 ~ 0.45[µ
   s];It is the truncation that occurs in the BD of the receiving
   node. It is present even if the frame does not pass through any
   active stars.
29 int nStarPath = 1;//0~2; the maximum number of active stars
   between any two nodes
30 double gdMinPropagationDelay = 0, gdMaxPropagationDelay =
   2.5;//A minimum/maximum propagation delay of the network as seen
   by the local node[µs], 0 ~ 2.5
31
32 double BusSpeed = 10;//Bus data rate[Mbit/s]
33 double gdCycle = 5000, STbus = 3000;//gdCycle:CC length[µs],
   STbus:ST segment length[µs]
34
35
36 //---------------ST segment parameters----------------
37 int num_ST;//number of ST messages
38 double gdStaticSlot; //ST-Slot length[µs]
39 int gNumberOfStaticSlots, gNumberOfStaticSlots_min,
   gNumberOfStaticSlots_use = 0;//gNumberOfStaticSlots_min: minimum
   required number of ST slots;gNumberOfStaticSlots_use:used number
   of ST slots
40 double gPayloadLengthStatic;//payload length of the ST
   frame[bit]
41 int aFrameLengthStatic;//ST Frame lengthg[dBit]
42 int scheduledCount = 0; //counter of scheduled messages
43 bool Vuse[1023][64] = {0}; //used vector space
44
45
46 struct buf
47 {
48     int x;
49     int y;
50 };
51
52 struct MessageST
53 {
54     int FrameID_ST; //FrameID of ST messages
55     int MessageLengthST; // Static message length[bit]
56     double T_ST; //Period of production of static message [µs]
57     double D_ST; //Deadline of static message [µs]
58
59     bool Port[4]; //ports message m occupied
60     bool Source[4]; //message m's source ports
```

```
61      bool Sink[4]; //message m's sink ports
62      bool Matrix[4][4]; //messages m's transmission path
63
64      int s; //base slot
65      int p; //slot repetition
66      int gNumberOfStaticSlot_m;//Number of ST slots for m
67      int b; //base cycle
68      int r; //cycle repetition
69      int gNumberOfCycle;//Number of cycles allocated to
70
71
72      bool buffer[1023][64][4];//slots message m occupied
73      bool V[1023][64][4];
74      bool isScheduled;
75
76      int SlotIDMax; //Max slot ID
77
78      buf SlotID;
79      buf CC;
80  };
81
82
83
84  int main()
85  {
86   /*-----------------------------------------------------------------
                              user input
    -----------------------------------------------------------------*/
87      /*cout  <<  "Please  choose  the  duration  of  the  samples
    clock[µs]: 0.0125, 0.0250, 0.0500";
88      cin >> gdSampleClockPeriod;
89      cout << "Please choose the number of samples per Microtick:
    1,2,4";
90      cin >> pSamplesPerMicrotick;*/
91      cout << "Please input the number of ST messages:";
92      cin >> num_ST;
93      /*cout << "Please input the maximum number of active stars
    between any 2 nodes:";
94      cin >> nStarPath;
95      cout << "Please input the FlexRay bus bit rate [MBit/s]:";
96      cin >> BusSpeed;
97      cout << "Please input the min bit rate of the ST messages
    [MBit/s]:";
98      cin >> min_bitRate;
99      cout << "Please input the max bit rate of the ST messages
    [MBit/s]:";
100     cin >> max_bitRate;*/
101
102     srand((unsigned)time(NULL));
103     for (int j = 0; j < 10; j++)
104     {
105         rand();
106     }
107
108     vector<MessageST> MST;
109     vector<MessageST> Lst;
110     vector<MessageST>::iterator iterST;
```

```
111
112  /*-------------------------------------------------------------
                    value the system parameters
     -------------------------------------------------------------*/
113     if (BusSpeed == 2.5)
114     {
115         gdBit = 0.4;
116         gdBitMax = 0.4006;
117         gdBitMin = 0.3994;
118     }
119     if (BusSpeed == 5)
120     {
121         gdBit = 0.2;
122         gdBitMax = 0.2003;
123         gdBitMin = 0.1997;
124     }
125     if (BusSpeed == 10)
126     {
127         gdBit = 0.1;
128         gdBitMax = 0.10015;
129         gdBitMin = 0.09985;
130     }
131
132     pdMicrotick = pSamplesPerMicrotick * gdSampleClockPeriod;
133     gdMaxMicrotick = pdMicrotick;
134     aWorstCasePrecision=(34    +  20  *  gClusterDriftDamping)  *
gdMaxMicrotick +2 * gdMaxPropagationDelay;
135
136     gAssumedPrecision = aWorstCasePrecision;
137     gdMaxInitializationError   =   2   *   (gdMaxMicrotick*   (1   +
0.0015)) + gdMaxPropagationDelay;
138
139     gdTSSTransmitter  =  ceil(  (gdBitMax+  dBDRxia+  nStarPath  *
dStarTruncation)/gdBitMin);
140     gdActionPointOffset     =     ceil((2*gAssumedPrecision     -
gdMinPropagationDelay    +2    *    gdMaxInitializationError)    /
(gdMacrotick* (1 - 0.0015)));
141 //-----------------output message info-----------------------
142     cout << "ID_ST" << '\t' << "ML_ST" << '\t' << "T_ST" << '\t'
<< "D_ST" << '\t' << endl;
143
144 //--------------------messages input parameters-------------
145     int ID = 1;
146     for (int n = 0; n < num_ST; n++)
147     {
148         MessageST *temp = new MessageST;
149         temp->FrameID_ST = ID++;
150
151         //-------------random T generation-------------
152         int a = rand() % 11;
153         if (a == 0)
154         {
155             temp->T_ST = 5000;
156         }
157         else if (a == 1)
158         {
159             temp->T_ST = 10010;
160         }
```

```
161             else if (a == 2)
162             {
163                 temp->T_ST = 20010;
164             }
165             else if (a == 3)
166             {
167                 temp->T_ST = 40010;
168             }
169             else if (a == 4)
170             {
171                 temp->T_ST = 80010;
172             }
173             else if (a == 5)
174             {
175                 temp->T_ST = 160010;
176             }
177             else if (a == 6)
178             {
179                 temp->T_ST = 320010;
180             }
181             else if (a == 7)
182             {
183                 temp->T_ST = 4000;
184             }
185             else if (a == 8)
186             {
187                 temp->T_ST = 2000;
188             }
189             else if (a == 9)
190             {
191                 temp->T_ST = 1000;
192             }
193             else if (a == 10)
194             {
195                 temp->T_ST = 500;
196             }
197 
198             temp->D_ST = temp->T_ST;
199             if (temp->D_ST > 40000)
200             {
201                 temp->D_ST = 40000;
202             }
203 
204             temp->MessageLengthST = 64;
205 
206             for (int i = 0; i <= 1022; i++)
207             {
208                 for (int j = 0; j <= 63; j++)
209                 {
210                     for (int k = 0; k <= 3; k++)
211                     {
212                         temp->buffer[i][j][k] = 0;
213                         temp->V[i][j][k] = 1;
214                     }
215                 }
216             }
217             //-------------random message Port generation---------
218             for (int k = 0; k <= 3; k++)
```

```
219         {
220             temp->Port[k] = 0;
221             temp->Source[k] = 0;
222             temp->Sink[k] = 0;
223         }
224         int first = rand() % 4;
225         temp->Source[first] = 1;
226         for (int m = 0; m <= 3; m++)
227         {
228             if(m != first)
229             {
230                 if (rand() % 2 == 0)
231                 {
232                     temp->Sink[m] = 0;
233                 }
234                 else
235                 {
236                     temp->Sink[m] = 1;
237                 }
238             }
239             temp->Port[m] = temp->Source[m] + temp->Sink[m];
240         }
241 
242         temp->isScheduled = false;
243         MST.push_back(*temp);
244         cout << setw(3) << temp->FrameID_ST << '\t' << temp->MessageLengthST << '\t' << temp->T_ST << '\t' << temp->D_ST << '\t' << endl;
245     }
246 //------------------output message port----------------------
247     cout <<"Port"<<'\t'<<endl;
248     for (iterST = MST.begin(); iterST != MST.end(); iterST++)
249     {
250         for (int m = 0; m <= 3; m++)
251         {
252             if(iterST->Port[m] == 1) // "1" means been occupied
253             {
254                 cout<<m<<',';
255             }
256         }
257         cout << endl;
258     }
259 
260  /*-------------------------------------------------------------
           value the message schedule's main parameters
     -------------------------------------------------------------*/
261     int MessageLengthSTMax;
262     for (iterST = MST.begin(); iterST != MST.end(); iterST++)
263     {
264         if (MessageLengthSTMax < iterST->MessageLengthST)
265             MessageLengthSTMax = iterST->MessageLengthST;
266     }
267 
268     gPayloadLengthStatic = MessageLengthSTMax;
269     aFrameLengthStatic    =    (gdTSSTransmitter+    1    +    80    +gPayloadLengthStatic* 1.25 + 2);//gdBit
270 
271     gdStaticSlot     =     2     *     gdActionPointOffset     +
```

```
    ceil(                    ((aFrameLengthStatic                    +
    11)*gdBitMax+gdMinPropagationDelay+gdMaxPropagationDelay)/(gdMac
    rotick *(1-0.0015)));
272     gNumberOfStaticSlots = STbus / gdStaticSlot;
273
274         for  (iterST  =  MST.begin();  iterST  !=  MST.end();
    iterST++)
275         {
276             if (iterST->D_ST < gdCycle)
277             {
278                 iterST->r = 1;
279                 for (int n = 0; n <= log(gdCycle/gdStaticSlot) /
    log(2); n++)
280                 {
281                         if   (iterST->p   <=   iterST->D_ST   /
    gdStaticSlot)
282                         {
283                             iterST->p = pow(2,n);
284                         }
285                 }
286                 iterST->gNumberOfStaticSlot_m                   =
    ceil((double)gNumberOfStaticSlots / iterST->p);
287                 iterST->gNumberOfCycle = 64;
288             }
289             else
290             {
291                 iterST->p = gNumberOfStaticSlots;
292                 iterST->gNumberOfStaticSlot_m = 1;
293                 if (iterST->D_ST > gdCycle)
294                 {
295                     for (int n = 0; n <= 6; n++)
296                     {
297                         if (iterST->r <= iterST->D_ST / gdCycle)
298                         {
299                             iterST->r = pow(2,n);
300                         }
301                     }
302                     iterST->gNumberOfCycle = 64 / iterST->r;
303                 }
304                 else
305                 {
306                     iterST->r = 1;
307                     iterST->gNumberOfCycle = 64;
308                 }
309             }
310         }
311 //--------------------basic  constraint--------------------

312         for  (iterST  =  MST.begin();  iterST  !=  MST.end();
    iterST++)
313         {
314             gNumberOfStaticSlots_min      +=      (double)iterST-
    >gNumberOfStaticSlot_m/iterST->r;
315         }
316         gNumberOfStaticSlots_min                                =
    ceil((double)gNumberOfStaticSlots_min);
317         if (gNumberOfStaticSlots_min <= 2)
318         {
```

```
319             gNumberOfStaticSlots_min = 2;
320         }
321         if (gNumberOfStaticSlots_min > gNumberOfStaticSlots)
322         {
323             cout << "non-schedulable ,gNumberOfStaticSlots_min>
    gNumberOfStaticSlots" << endl;
324             return 0;
325         }
326 //----------------------sort------------------------------
327         while (MST.size() != 0)
328         {
329             vector<MessageST>::iterator temp = MST.begin();
330             for (iterST = MST.begin(); iterST != MST.end();
    iterST++)
331             {
332                 if (iterST->p < temp->p || (iterST->p == temp->p
    && iterST->r < temp->r))
333                 {
334                     temp = iterST;
335                 }
336             }
337             Lst.push_back(*temp);
338             MST.erase(temp);
339         }
340 //--------------------scheduling process--------------------
341         for (iterST = Lst.begin(); iterST != Lst.end();
    iterST++)
342         {
343             if (iterST->isScheduled)
344             {
345                 continue;
346             }
347             iterST->SlotID.x = 0;
348             iterST->SlotID.y = iterST->p - 1;
349             iterST->CC.x = 0;
350             iterST->CC.y = iterST->r - 1;
351
352             for (int i = iterST->SlotID.x; i <= iterST-
    >SlotID.y; i++)
353             {
354                 for (int j = iterST->CC.x; j <= iterST->CC.y;
    j++)
355                 {
356                     for (int k = 0; k <= 3; k++)
357                     {
358                         if (Vuse[i][j][k] == 0) //space not be
    used
359                         {
360                             iterST->V[i][j][k] = 0;
361                         }
362                     }
363                 }
364             }
365             for (int sm = iterST->SlotID.x; sm <= iterST-
    >SlotID.y; sm++)
366             {
367                 if (iterST->isScheduled)
```

```
368                 {
369                     break;
370                 }
371                 for (int bm = iterST->CC.x; bm <= iterST->CC.y;
    bm++)
372                 {
373                     if (iterST->isScheduled)
374                     {
375                         break;
376                     }
377                     bool flag = true;
378                     for (int k = 0; k <= 3; k++)
379                     {
380                         if    (iterST->Port[k]    ==    1&&iterST-
    >V[sm][bm][k] == 1)
381                         {
382                             flag = false;
383                             break;
384                         }
385                     }
386                     if (!flag)
387                     {
388                         continue;
389                     }
390                     if (flag)
391                     {
392                         iterST->s = sm;
393                         iterST->b = bm;
394                         for   (int   m   =   1;   m   <=   iterST-
    >gNumberOfStaticSlot_m; m++)
395                         {
396                             for   (int   n   =   1;   n   <=   iterST-
    >gNumberOfCycle; n++)
397                             {
398                                 for (k = 0; k <= 3; k++)
399                                 {
400                                     if (iterST->Port[k] == 1)
401                                     {
402                                         iterST->buffer[iterST-
    >s+(m-1)*iterST->p][iterST->b+(n-1)*iterST->r][k] = 1;
403                                     }
404                                 }
405                             }
406                         }
407                         for (int i = 0; i <= 1022; i++)
408                         {
409                             for (int j = 0; j <= 63; j++)
410                             {
411                                 for (k = 0; k <= 3; k++)
412                                 {
413                                     Vuse[i][j][k]                =
    Vuse[i][j][k] | iterST->buffer[i][j][k]; //update Vuse
414                                 }
415                             }
416                         }
417
418                         iterST->SlotIDMax   =   iterST->s   +   1
    +(iterST->gNumberOfStaticSlot_m - 1)*iterST->p;
```

```
419
420                         if      (       iterST->SlotIDMax       >
    gNumberOfStaticSlots_use)
421                         {
422                             gNumberOfStaticSlots_use=    iterST-
    >SlotIDMax; //update used max slot ID
423                         }
424                         if      (gNumberOfStaticSlots_use       >
    gNumberOfStaticSlots)
425                         {
426                             cout   <<   "system   non-schedulable.
    gNumberOfStaticSlots_use > gNumberOfStaticSlots." << endl;
427                             return 0;
428                         }
429                         iterST->isScheduled = true;
430                         scheduledCount++;
431                         if (scheduledCount == Lst.size())
432                         {
433                             cout << "scheduled" << endl;
434                             cout     <<     "gdStaticSlot:"     <<
    gdStaticSlot << endl;
435                             cout  <<  "gNumberOfStaticSlots:"  <<
    gNumberOfStaticSlots << endl;
436                             cout << "number of ST slots used:"
    << gNumberOfStaticSlots_use << endl;
437     //------------------output ST schedule-------------------------

438                             vector<MessageST>::iterator
    iterSTOut;
439                             cout << "ID_ST" << '\t' << "s" <<
    '\t' << "p" << '\t' << setw(8) << "STSlotNum";
440                             cout << setw(5) << "b" << setw(5) <<
    "r" << '\t' << "CycNum" << '\t' << "SlotIDMax" << '\t' << endl;
441                             for   (iterSTOut   =   Lst.begin();
    iterSTOut != Lst.end(); iterSTOut++)
442                             {
443                                 cout  <<  setw(3)  <<  iterSTOut-
    >FrameID_ST << '\t' << iterSTOut->s + 1 << '\t' << iterSTOut->p
    << '\t' << setw(4) << iterSTOut->gNumberOfStaticSlot_m;
444                                 cout << setw(10) << iterSTOut->b
    << setw(5) << iterSTOut->r << '\t' << setw(4) << iterSTOut-
    >gNumberOfCycle << setw(9) << iterSTOut->SlotIDMax << '\t' <<
    endl;
445                             }
446     //------------------------------------------------------------
447                             return 0;
448                         }
449                         else
450                         {
451                             break;
452                         }
453                     }
454                 }
455             }
456         }
457     cout   <<   "system   non-schedulable.   Schedulability:"   <<
    (double)scheduledCount/Lst.size()*100 << "%" << endl;
458     cout << "gdStaticSlot:" << gdStaticSlot << endl;
```

```
459     cout << "gNumberOfStaticSlots:" << gNumberOfStaticSlots <<
    endl;
460     return 0;
461 }
```

# D

# Source Code of Switched DYN Scheduler

```
1  /*-------------------------------------------------------------
             Switched FlexRay DYN scheduling algorithm
   -------------------------------------------------------------*/
2  #include <iostream>
3  #include <iomanip>
4  #include <vector>
5  #include <set>
6  #include <algorithm>
7  #include <time.h>
8  #include <math.h>
9
10 using namespace std;
11
12 //----------------System parameters-------------------
13 const double gdMacrotick = 2;//Duration of the cluster wide
   nominal macrotick[μs]
14 double gdTSSTransmitter;//Number of bits in the Transmission
   Start Sequence[gdBit]
15 double gdBit, gdBitMax, gdBitMin; // gdBit:Nominal bit time[μ
   s], gdBitMax[μs], gdBitMin[μs]
16 double dStarTruncation = 0.45; //Interval by which the
   transmission of a frame is shortened by one star = 0.45[μs]
17 double dBDRxia = 0.3;//Activity reaction time. Time by which a
   transmission becomes shortened in a receiving node = 0 ~ 0.45[μ
   s];It is the truncation that occurs in the BD of the receiving
   node. It is present even if the frame does not pass through any
   active stars.
18 int nStarPath = 1;//0~2; the maximum number of active stars
   between any two nodes
19
20 double BusSpeed = 10;//Bus data rate[Mbit/s]
21 double gdCycle = 5000, STbus = 3000, DYNbus;//gdCycle:CC
   length[μs], STbus:ST segment length[μs], DYN bus length[μs]
22
23
```

```
24 //---------------DYN segment parameters----------------
25 int num_DYN;//number of DYN messages
26 double gdMinislot;//Minislot length[μs]
27 int gNumberOfMinislots;
28 int scheduledCountDYN = 0;
29 int pLatestTx;//Number of the last minislot in which a frame
   transmission can start in the dynamic segment
30 double aFrameLength_DYN;//DYN frame length
31
32
33 struct MessageDYN
34 {
35     int FrameID_DYN; //FrameID
36     int FrameID_DYNnew; //FrameID after clustering
37     int MessageLengthDYN;// MessageLengthDYN[bit]
38     double FrameLengthDYN; //DYN Frame length
39     double D_DYN;//Deadline of dynamic message [μs]
40     double R_DYN; //response time[μs]
41
42     bool Port_DYN[4]; //the ports message m occupied
43     int clusterID_DYN; //cluster ID messages m belonged
44     bool isCluster;
45 };
46
47
48 int main()
49 {
50 //-------------------user input----------------------------
51     cout << "Please input the number of DYN messages:";
52     cin >> num_DYN;
53     /*cout << "Please input the length of a Minislot:";
54     cin >> gdMinislot;*/
55
56     srand((unsigned)time(NULL));
57     for (int j = 0; j < 10; j++)
58     {
59         rand();
60     }
61
62     vector<MessageDYN> MDYN;
63     vector<MessageDYN> Ldyn;
64     vector<MessageDYN>::iterator iterDYN;
65     vector<MessageDYN>::iterator iterFrameID;
66
67     vector<MessageDYN> lf;
68     vector<MessageDYN> lf_sort;
69     vector<MessageDYN>::iterator iterlf;
70     vector<MessageDYN>::iterator iterMessageLength;
71
72     if (BusSpeed == 2.5)
73     {
74         gdBit = 0.4;
75         gdBitMax = 0.4006;
76         gdBitMin = 0.3994;
77     }
78     if (BusSpeed == 5)
79     {
```

```
80          gdBit = 0.2;
81          gdBitMax = 0.2003;
82          gdBitMin = 0.1997;
83      }
84      if (BusSpeed == 10)
85      {
86          gdBit = 0.1;
87          gdBitMax = 0.10015;
88          gdBitMin = 0.09985;
89      }
90
91  //-------------------------------------------------------------
                          system parameters
     -------------------------------------------------------------
92      gdTSSTransmitter = ceil( (gdBitMax+ dBDRxia+ nStarPath *
    dStarTruncation)/gdBitMin);
93      DYNbus = gdCycle - STbus;
94      gNumberOfMinislots = DYNbus / gdMinislot;//int = floor()
95
96   //-------------------------------------------------------------
                        messages input parameters
     -------------------------------------------------------------
97      for (int m = 0; m < num_DYN; m++)
98      {
99          MessageDYN *temp = new MessageDYN;
100
101 //-------------random D generation----------------
102         int a = rand() % 7;
103         if (a == 0)
104         {
105             temp->D_DYN = 100000;
106         }
107         else if (a == 1)
108         {
109             temp->D_DYN = 200000;
110         }
111         else if (a == 2)
112         {
113             temp->D_DYN = 500000;
114         }
115         else if (a == 3)
116         {
117             temp->D_DYN = 1000000;
118         }
119         else if (a == 4)
120         {
121             temp->D_DYN = 2000000;
122         }
123         else if (a == 5)
124         {
125             temp->D_DYN = 10000000;
126         }
127         else if (a == 6)
128         {
129             temp->D_DYN = 20000000;
130         }
131
132         temp->MessageLengthDYN = rand() % 128;//maximum 16 bytes
```

```
DYN data supported, changeable
133         temp->isCluster = false;
134   //-------------random message Port generation----------
135         int first = rand() % 4;
136         temp->Port_DYN[first] = 1;
137         for (int i = 0; i <= 3; i++)
138         {
139             if(i != first)
140             {
141                 if (rand() % 2 == 0)
142                 {
143                     temp->Port_DYN[i] = 0;
144                 }
145                 else
146                 {
147                     temp->Port_DYN[i] = 1;
148                 }
149             }
150         }
151         MDYN.push_back(*temp);
152     }
153
154  //-------------------------------------------------------------
                        sort the DYN messages
  -------------------------------------------------------------
155     int ID = 1;
156     while (MDYN.size() != 0)
157     {
158         vector<MessageDYN>::iterator temp = MDYN.begin();
159         for (iterDYN = MDYN.begin(); iterDYN != MDYN.end();
iterDYN++)
160         {
161             if (iterDYN->MessageLengthDYN/iterDYN->D_DYN > temp-
>MessageLengthDYN/temp->D_DYN)
162             {
163                 temp = iterDYN;
164             }
165         }
166         temp->FrameID_DYN = ID++;
167         Ldyn.push_back(*temp);
168         MDYN.erase(temp);
169     }
170
171   //-------------------------------------------------------------
                              clustering
    -------------------------------------------------------------
172     int cluster[100][4];
173     int cluster_max = 1;
174     for (int i = 0; i < 100; i++)
175     {
176         for (int j = 0; j < 4; j++)
177         {
178             cluster[i][j] = -1;
179         }
180     }
181
182     for (iterDYN = Ldyn.begin(); iterDYN != Ldyn.end();
iterDYN++)
```

```
183     {
184         if (iterDYN == Ldyn.begin())
185         {
186             for (int j = 0; j < 4; j++)
187             {
188                 if ( iterDYN->Port_DYN[j] == 1 )
189                 {
190                     cluster[1][j] = iterDYN->FrameID_DYN;
191                 }
192             }
193             iterDYN->clusterID_DYN = 1;
194             iterDYN->isCluster = true;
195         }
196         else
197         {
198             for (int i = 1; i <= cluster_max; i++)//search all
    valid clusters
199             {
200                 if (iterDYN->isCluster)
201                 {
202                     break;
203                 }
204                 bool flag =true;
205                 for (int j = 0; j < 4; j++)
206                 {
207                     if ( iterDYN->Port_DYN[j] == 1 &&
    cluster[i][j] > 0 )//one port has been occupied
208                     {
209                         flag = false;
210                     }
211                 }
212                 if (flag)//every port needed by the transmission
    is valid
213                 {
214                     for (int j = 0; j < 4; j++)
215                     {
216                         if ( iterDYN->Port_DYN[j] == 1)
217                         {
218                             cluster[i][j] = iterDYN-
    >FrameID_DYN;
219                         }
220                     }
221                     iterDYN->clusterID_DYN = i;
222                     iterDYN->isCluster = true;
223                 }
224             }
225             if (!iterDYN->isCluster)//cannot find the valid
    cluster, create a new one
226             {
227                 cluster_max++;
228                 for (int j = 0; j < 4; j++)
229                 {
230                     if ( iterDYN->Port_DYN[j] == 1 )
231                     {
232                         cluster[cluster_max][j] = iterDYN-
    >FrameID_DYN;
233                     }
234                 }
```

```
235                 iterDYN->clusterID_DYN = cluster_max;
236                 iterDYN->isCluster = true;
237             }
238         }
239     }
240  //-------------------------------------------------------------
                          update FrameIDs
  -------------------------------------------------------------
241     cout << "FrameID" << endl;
242     for (iterDYN = Ldyn.begin(); iterDYN != Ldyn.end();
iterDYN++)
243     {
244         cout << setw(4) << iterDYN->FrameID_DYN << endl;//output
original FrameID as reference
245     }
246
247     cout << "FrameID_new" << '\t' << endl;
248     for (iterDYN = Ldyn.begin(); iterDYN != Ldyn.end();
iterDYN++)
249     {
250         iterDYN->FrameID_DYNnew = iterDYN->FrameID_DYN;
251         for (int j = 0; j < 4; j++)
252         {
253             if (cluster[iterDYN->clusterID_DYN][j] > 0 &&
cluster[iterDYN->clusterID_DYN][j] < iterDYN->FrameID_DYNnew)
254             {
255                 iterDYN->FrameID_DYNnew = cluster[iterDYN-
>clusterID_DYN][j];
256             }
257         }
258         if(iterDYN->FrameID_DYNnew < iterDYN->FrameID_DYN)
259         {
260             for (iterFrameID= Ldyn.begin(); iterFrameID!=
Ldyn.end(); iterFrameID++)
261             {
262                 if(iterFrameID->FrameID_DYN > iterDYN-
>FrameID_DYN)
263                 {
264                     iterFrameID->FrameID_DYN = iterFrameID-
>FrameID_DYN - 1;
265                 }
266             }
267         }
268         cout << setw(4) << iterDYN->FrameID_DYNnew << '\t' <<
endl;//output updated FrameID
269         iterDYN->FrameID_DYN = iterDYN->FrameID_DYNnew;//use new
FrameID in the following calculation
270     }
271
272   //-------------------------------------------------------------
                      update the Frame IDs end
   -------------------------------------------------------------
273
274     cout << "FrameID" << setw(8) << "ML_DYN" << setw(11) <<
"D_DYN[ms]" << setw(12) << "R_DYN[ms]" << setw(18) <<
"scheduled?" << '\t' << endl;
275   //-------------------------------------------------------------
                      worst-case reponse time
```

```
  ------------------------------------------------------------
276     for (iterDYN = Ldyn.begin(); iterDYN != Ldyn.end();
    iterDYN++)
277     {
278         aFrameLength_DYN = gdTSSTransmitter + 83 + iterDYN-
    >MessageLengthDYN * 1.25;
279         pLatestTx = gNumberOfMinislots - (aFrameLength_DYN *
    gdBit) / gdMinislot;
280
281  //----------------construct lf(m) list----------------------
282
283         lf_sort.clear();//initiate vector lf_sort for the next
    m_DYN
284         int FrameID_m = iterDYN->FrameID_DYN;
285         for (iterlf = Ldyn.begin(); iterlf!= Ldyn.end();
    iterlf++)
286         {
287             if (iterlf->FrameID_DYN < FrameID_m)
288             {
289                 bool hasSameID = false;
290                 for (iterMessageLength = lf.begin();
    iterMessageLength!= lf.end(); iterMessageLength++)
291                 {
292                     if(iterMessageLength->FrameID_DYN == iterlf-
    >FrameID_DYN)
293                     {
294                         hasSameID = true;
295                         if (iterMessageLength->MessageLengthDYN
    < iterlf->MessageLengthDYN)//check the same FrameID message size
296                         {
297                             iterMessageLength->clusterID_DYN =
    iterlf->clusterID_DYN;
298                             iterMessageLength->MessageLengthDYN
    = iterlf->MessageLengthDYN;
299                             iterMessageLength->FrameID_DYN =
    iterlf->FrameID_DYN;
300                             iterMessageLength->FrameID_DYNnew =
    iterlf->FrameID_DYNnew;
301                             iterMessageLength->D_DYN = iterlf-
    >D_DYN;
302                             iterMessageLength->R_DYN = iterlf-
    >R_DYN;
303                             for (int port_i = 0; port_i < 4;
    port_i++)
304                             {
305                                 iterMessageLength-
    >Port_DYN[port_i] = iterlf->Port_DYN[port_i];
306                             }
307                             iterMessageLength->isCluster =
    iterlf->isCluster;
308                         }
309                     }
310                 }
311                 if (!hasSameID)
312                 {
313                     lf.push_back(*iterlf);//put into vector
314                 }
315             }
```

```
316             else
317             {
318                 continue;
319             }
320         }
321 
322  //-------------------------------------------------------------
                sort the lf(m) in decreasing order
     -------------------------------------------------------------
323         while (lf.size() != 0)
324         {
325             vector<MessageDYN>::iterator temp = lf.begin();
326             for (iterlf= lf.begin(); iterlf != lf.end();
    iterlf++)
327             {
328                 if (iterlf->MessageLengthDYN > temp-
    >MessageLengthDYN)
329                 {
330                     temp = iterlf;
331                 }
332             }
333             lf_sort.push_back(*temp);
334             lf.erase(temp);//initiate lf for the next m_DYN
335         }
336 
337 //-------------------------------------------------------------
          first fit bin packing calculates BusCycles_lf
     -------------------------------------------------------------
338         double bin[64];
339         for (int n = 1; n <= 64; n++)
340         {
341             bin[n] = DYNbus;
342         }
343         int BusCycles_lf = 1;
344         for (iterlf= lf_sort.begin(); iterlf != lf_sort.end();
    iterlf++)
345         {
346             iterlf->FrameLengthDYN = gdTSSTransmitter + 83 +
    iterlf->MessageLengthDYN * 1.25;
347             for (n = 1; n <= 64; n++)//searching all avilable
    bins
348             {
349                 if ( bin[n] >= iterlf->FrameLengthDYN &&
    bin[n] >= DYNbus-pLatestTx * gdMinislot)//enough space, put into
    this bin
350                 {
351                     if ( n > BusCycles_lf)//update BusCycles_lf
352                     {
353                         BusCycles_lf = n; //determine the number
    of cycles occupied by lf messages
354                     }
355                     bin[n] -= iterlf->FrameLengthDYN;//update
    the spare space of bin[n]
356                     break;
357                 }
358                 else
359                 {
360                     continue;//search next bin
```

```
361                 }
362             }
363         }
364
365   //-------------------------------------------------------------
                      worst-case response time
      ------------------------------------------------------------
366         iterDYN->R_DYN = (gdCycle-(STbus+(iterDYN->FrameID_DYN-
    1)*gdMinislot))+ BusCycles_lf *gdCycle + (iterDYN->FrameID_DYN-
    1)*gdMinislot + (STbus+pLatestTx*gdMinislot)+(aFrameLength_DYN
    *gdBit/BusSpeed);
367         if (iterDYN->R_DYN <= iterDYN->D_DYN)
368         {
369             scheduledCountDYN++;
370             cout << setw(4) << iterDYN->FrameID_DYN << setw(9)
    << iterDYN->MessageLengthDYN << setw(11) << iterDYN->D_DYN/1000
    << setw(13) << iterDYN->R_DYN/1000 << setw(15) << "Yes" << '\t'
    << endl;
371             if (scheduledCountDYN == Ldyn.size())
372             {
373                 cout << "System schedulable. DYN messages are
    100% scheduled." << endl;
374                 return 0;
375             }
376         }
377         else
378         {
379             cout << setw(4) << iterDYN->FrameID_DYN << setw(9)
    << iterDYN->MessageLengthDYN << setw(11) << iterDYN->D_DYN/1000
    << setw(13) << iterDYN->R_DYN/1000 << setw(15) << "No" << '\t'
    << endl;
380         }
381     }
382     cout << "system non-schedulable. Schedulability:" <<
    (double)scheduledCountDYN/Ldyn.size()*100 << "%"<< endl;
383     return 0;
384 }
```

# E

# Pseudocode for DYN Schedulers with ECUs' Output

## E.1 Notations

Here we would like to present some notations which may use in the discussion.

- $CycleID_m$ is the ID of the cycle in which the DYN message is scheduled;
- $CycleID_{m\_g}$ is the value of cycle counter at the instant of the DYN message m is generated;
- $CycleID_{m\_t}$ is the value of cycle counter at the instant of the DYN message m is transmitted, $CycleID_{m\_t} \in [0,63]$, $CycleID_{m\_t} \in Z$;
- $DYN_{bus}$ is the length of the DYN segment;
- $gdMinislot$ is the number of Macroticks constituting the duration of a Minislot;
- $Macrotick_{m\_g}$ is the value of Macrotick in the node timer at the instant of the DYN message m is generated;
- $Macrotick_{m\_t}$ is the value of Macrotick in the node timer at the instant of the DYN message m is transmitted;
- $MessageLengthDYN_m$ is the number of bits constituting the dynamic message $m$ in the cluster;

- $MinislotID_m$ is the number of Minislot from the beginning of the DYN segment to the transmission of message m;
- $pLatestTx_m$ is the number of the last Minislot in which a frame transmission can start in the DYN segment;
- $R_m$ is the worst-case response time of the DYN message m;
- $ST_{bus}$ is the length of the ST segment;
- $vSlotCounter_{DYN}$ is the value of slot counter of DYN segment;

## E.2 Representation of the Schedule

Unlike the fixed starting time of any ST slot in the ST segment, the starting time of the DYN slot is not fixed in a schedule. It depends on the transmission data in different networks. Therefore we cannot imitate the way of schedule representing in ST segment to use the Frame ID as a part of the schedule.

According to FlexRay specifications, the node shall provide at least one absolute timer that may be set to an absolute time in terms of cycle count and Macrotick, i.e. the timer is set to expire at a determined Macrotick in a determined communication cycle [11]. This means any instant in time can represent as 2-tuple vector $(vCycleCounter, Macrotick)$. Moreover, the length of a Minislot is fixed in a schedule so each Minislot ID has a fixed starting time. Therefore the Minislot ID is a good reference grid for any message's schedule. The transmission schedule of the DYN message $m$ can be represented by a 2-tuple vector:

$$\text{Schedule}_m = \{MinislotID_m, CycleID_m\} \tag{E.1}$$

$\text{Schedule}_m = \{MinislotID_m, CycleID_m\}$ $\text{Schedule}_m = \{MinislotID_m, CycleID_m\}$

Every DYN message starts to transmit at position $MinislotID_m$ of cycle number $CycleID_m$.

Therefore the problem of the scheduler design in DYN segment needs to consider these two factors. Here list them below:

- Length of the DYN segment $DYN_{bus}$;
- The way to assign the Frame ID and the optimal way to set the order of Frame ID;
- Parameters $MinislotID_m$ and $CycleID_m$;

## E.3 Motivation for the Solution

If there is an idle instant which with the value of DYN slot counter equal to message m's FrameID and it satisfies the $pLatestTx_m$ condition, the DYN message m can start to transmit. We denote this instant of the DYN message generation as $\left(CycleID_{m\_g}, Macrotick_{m\_g}\right)$. Similarly, we denote the instant of the DYN message transmission as $\left(CycleID_{m\_t}, Macrotick_{m\_t}\right)$. The interval between these two instants is the message response time $R_m$. It cannot longer than message deadline $D_m$. The mathematic expression for this constraint is:

$$R_m \le D_m \tag{E.2}$$

The response time can write as follow:

$$\begin{aligned} R_m &= \left|\left(CycleID_{m\_t}, Macrotick_{m\_t}\right) - \left(CycleID_{m\_g}, Macrotick_{m\_g}\right)\right| \\ &= \left(CycleID_{m\_t} - CycleID_{m\_g}\right) \times gdCycle[\mu s] + \\ &\quad \left(Macrotick_{m\_t} - Macrotick_{m\_g}\right) \times gdMacrotick[\mu s] \end{aligned} \tag{E.3}$$

$MinislotID_m$ is related with $Macrotick_{m\_t}$ by:

$$MinislotID_m = \frac{Macrotick_{m\_t}[MT]}{gdMinislot[MT]} \tag{E.4}$$

## E.4 Pseudocode for Simple DYN Scheduler with ECUs' Output

**Input:**

- Bus bit rate $\boldsymbol{bus\_speed}$
- The DYN message set $\boldsymbol{M_{DYN}}$, $\boldsymbol{M_{DYN}}$ is maximum DYN message set waiting to send in a cluster, $m = \left(D_m, MessageLengthDYN_m\right), m \in M_{DYN}$

**Output:**

- $m \times 2$ **scheduling matrix** $\mathbf{SimpleScheduler_{DYN}}$

$$\begin{bmatrix} MinislotID_1 & CycleID_1 \\ MinislotID_2 & CycleID_2 \\ MinislotID_3 & CycleID_3 \\ \cdots & \cdots \\ MinislotID_m & CycleID_m \end{bmatrix}$$

Simple DYN scheduler with ECUs' Output

for $\forall m \in M_{DYN}$

Sort the DYN messages $\forall m \in M_{DYN}$ in descending order of the value $\frac{MessageLength_m}{D_m}$ and then in descending order of the value $MessageLength_m$, store them in the message list $L_{DYN}$

Assign the Frame ID to the DYN messages with the order in list $L_{DYN}$ from 1

end for

for $gdCycle = 10$ to $16000\mu$s step $20 \times gdBit$ μs

Simple FlexRay ST scheduling algorithm

$ST_{bus} = gdStaticSlot * gNumberOfStaticSlots$

$DYN_{bus} = gdCycle - ST_{bus}$

for $gdMinislot = 2$ to 63MT step 1 $gdMacrotick$ [MT]

for $m \in L_{DYN}$

$pLatestTx_m =$
$(DYN_{bus} - (gdTSSTransmitter + 83 +$
$MessageLengthDYN$ [bit] $* 1.25) *$
$gdBit) / (gdMacrotick * gdMinislot)$

$CycleID_{m\_g}$ = the value of cycle counter in the timer at this instant

$Macrotick_{m\_g}$ = the value of Macrotick in the timer at this instant

$CycleID_m = CycleID_{m\_g}$

if m is the first message

$DYN_{use} = 0$

$vSlotCounter_{DYN} = 1$

end if

$DYN_m = DYN_{bus} - DYN_{use}$

// update the valid time for message m in DYN segment

if $CycleID_m \le vCycleCounter_{\max}$

// $vCycleCounter_{\max}$ is the value get after ST segment scheduling

if $vSlotCounter_{DYN} = FrameID_m$

if $\left\lfloor \frac{DYN_m}{gdMinislot} \right\rfloor \ge pLatestTx_m$

$CycleID_{m\_t}$ = the value of cycle counter in the timer at this instant

//read the new value from the timer

$Macrotick_{m\_t}$ = the value of Macrotick in the timer at this instant

//read the new value from the timer

$R_m = \left(CycleID_{m\_t} - CycleID_{m\_g}\right) \times gdCycle[\mu s] + \left(Macrotick_{m\_t} - Macrotick_{m\_g}\right) \times gdMacrotick[\mu s]$

if $R_m \le D_m$

$$MinislotID_m = \frac{Macrotick_{m\_t}}{gdMinislot}$$

store $\left(MinislotID_m, CycleID_m\right)$ in the matrix $\mathbf{SimpleScheduler_{DYN}}$

$DYN_{use} = DYN_{use} + MessageLength_m$

take m out of $L_{DYN}$, update $L_{DYN}$

if $L_{DYN}$ is empty

output matrix $\mathbf{SimpleScheduler_{DYN}}$, exit //system schedulable

end if

continue with next message $m+1 \in L_{DYN}$

$vSlotCounter_{DYN} = vSlotCounter_{DYN} + 1$

else

continue with next *gdMinislot* value

end if

else

$DYN_{use} = DYN_{use} + gdMinislot$

$vSlotCounter_{DYN} = vSlotCounter_{DYN} + 1$

continue with next message $m+1 \in L_{DYN}$

end if

else

$vSlotCounter_{DYN} = vSlotCounter_{DYN} + 1$

end if

$CycleID_m = CycleID_m + 1$

else

continue with next *gdMinislot* value

end if

end for

end for

end for

output non-schedulable, exit //system non-schedulable

**Algorithm E Pseudocode for** Simple DYN scheduler with ECUs' Output

## E.5 Pseudocode for Switched DYN Scheduler with ECUs' output

**Input:**

- Bus bit rate $\boldsymbol{bus\_speed}$
- Switch port set $\boldsymbol{Port_{switch}}$

- The DYN message set $\boldsymbol{M_{DYN}}$, $\boldsymbol{M_{DYN}}$ is maximum DYN message set waiting to send in a cluster, $m = \{MessageLengthDYN_m, D_m, Port_m \ \mathbf{Matrix_m}\} \ m \in M_{DYN}$

**Output:**

System schedulable or non-schedulable

---

Switched DYN scheduler with ECUs' output

---

for $\forall m \in M_{DYN}$

Sort the DYN messages $\forall m \in M_{DYN}$ in descending order of the value $\frac{MessageLength_m}{D_m}$ and then in descending order of the value $MessageLength_m$, store them in the message list $L_{DYN}$

Assign the Frame ID to the DYN messages with the order in list $L_{DYN}$ from 1

end for // sort the messages and assign the Frame IDs

// start of clustering and Frame IDs updating

for $\forall m \in L_{DYN}$

if $m$ is the first message in $L_{DYN}$

create set $cluster_1 = \{\text{empty}\}$ //create the first message set $cluster_1$

$ClusterSet = \{cluster_1\}$ // create a set represented all the available clusters

$Port_{use\_cluster_1} = Port_m$

$m \in cluster_1$ // put $m$ in set $cluster_1$

else // $m$ is not the first message

for all $cluster_1$ to $cluster_m$ in the set $ClusterSet$

//for all valid cluster in $ClusterSet$

if $Port_m \subseteq Port_{valid\_cluster_n}$, $n \in [1, m], n \in \mathbb{N}$

$Port_{use\_cluster_n} = Port_{use\_cluster_n} + Port_m$

$m \in cluster_n$ // put message m in $cluster_n$

go to the next message

// if message is able to put in any one of the existed $cluster_n$, finish the

clustering step of message m

end if

end for

create set $cluster_{m+1} = \{\text{empty}\}$ //create a new set $cluster_{m+1}$

/* if search all available clusters message still cannot find a cluster can fit in, create a new cluster */

$Port_{use\_cluster_{m+1}} = Port_m$

$m \in cluster_{m+1}$

// put $m$ in the new created cluster $cluster_{m+1}$

go to the next message //finish the cluster step of this message

end if

$Port_{valid\_cluster_n} = Port_{switch} - Port_{use\_cluster_n}$

//update the valid ports for any set $cluster_n$ included message m

if $Port_{valid\_cluster_n} = 0$

$ClusterSet = ClusterSet - cluster_n$

//update the valid cluster set $ClusterSet$

end if

end for // finish the clustering step for all the DYN messages

for $\forall m \in L_{DYN}$ // update the Frame ID for each DYN message

$FrameID_m = FrameID_{min}\ in\ cluster_n,\ message\ m \in cluster_n$

end for

// start the calculation of the worst-case response time

for $gdCycle = 10$ to $16000\mu$s step $20 \times gdBit$ μs

simple FlexRay ST scheduling algorithm

$DYN_{bus} = gdCycle - ST_{bus}$

for $gdMinislot = 2$ to 63MT step 1 $gdMacrotick$ [MT]

for $m \in L_{DYN}$

$pLatestTx_m =$

$(DYN_{bus} - (gdTSSTransmitter + 83 + MessageLengthDYN\ [\text{bit}] * 1.25) * gdBit) / (gdMacrotick * gdMinislot)$

$CycleID_{m\_g}$ = the value of cycle counter in the timer at this instant

$Macrotick_{m\_g}$ = the value of Macrotick in the timer at this instant

$CycleID_m = CycleID_{m\_g}$

if m is the first message

$DYN_{use} = 0$

$vSlotCounter_{DYN} = 1$

end if

$DYN_m = DYN_{bus} - DYN_{use}$

// update the valid time for message m in DYN segment

if $CycleID_m \le vCycleCounter_{\max}$

// $vCycleCounter_{\max}$ is the value get after ST segment scheduling

if $vSlotCounter_{DYN} = FrameID_m$

if $\left\lfloor \frac{DYN_m}{gdMinislot} \right\rfloor \ge pLatestTx_m$

$CycleID_{m\_t}$ = the value of cycle counter in the timer at this instant

//read the new value from the timer

$Macrotick_{m\_t}$ = the value of Macrotick in the timer at this instant

//read the new value from the timer

$R_m = \left(CycleID_{m\_t} - CycleID_{m\_g}\right) \times gdCycle[\mu s] + \left(Macrotick_{m\_t} - Macrotick_{m\_g}\right) \times gdMacrotick[\mu s]$

if $R_m \le D_m$

$$MinislotID_m = \frac{Macrotick_{m\_t}}{gdMinislot}$$

store $(MinislotID_m, CycleID_m)$ in the matrix **SimpleScheduler$_{\mathbf{DYN}}$**

$DYN_{use} = DYN_{use} + MessageLength_m$

take m out of $L_{DYN}$, update $L_{DYN}$

if $L_{DYN}$ is empty

output matrix **SimpleScheduler$_{\mathbf{DYN}}$**, exit //system schedulable

end if

continue with next message $m+1 \in L_{DYN}$

$vSlotCounter_{DYN} = vSlotCounter_{DYN} + 1$

else

continue with next *gdMinislot* value

end if

else

$DYN_{use} = DYN_{use} + gdMinislot$

$vSlotCounter_{DYN} = vSlotCounter_{DYN} + 1$

continue with next message $m+1 \in L_{DYN}$

end if

else

$vSlotCounter_{DYN} = vSlotCounter_{DYN} + 1$

end if

$CycleID_m = CycleID_m + 1$

else

continue with next *gdMinislot* value

```
                end if
            end for
        end for
    end for
output non-schedulable, exit  //system non-schedulable
```

**Algorithm F Pseudocode for Switched DYN scheduler with ECUs' output**