Markus Nager, BSc MA

Real-Time Multiplexing of Mixed-Criticality Data Streams for Automotive Multi-Core Test Systems

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Software Development and Business Administration

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Inf. Univ. Dr.rer.nat. Marcel Carsten Baunach

Institute of Technical Informatics
Embedded Automotive Systems Group

Graz, November 2017

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Abstract

With the integration of sophisticated features in modern cars, such as automatic parking, traffic sign recognition, or advanced driver assistance systems (ADAS), vehicles are becoming increasingly complex. To keep up with in-vehicle systems, automotive verification and validation (V&V) systems need to be redesigned and enhanced.

Historically, automotive test systems were designed for a single core architecture. This, however, limited the utilization of shared resources and generated high hardware costs.

In this thesis we present a redesigned AVL automotive test system that was upgraded to a multi-core architecture. As part of the redesign, we implemented a Connectivity Manager (CM) for the V&V system that is in charge of multiplexing mixed-criticality data streams from multiple cores across a shared network. Particularly, we used a controller are network (CAN) as shared communication network, since CAN is the most common and utmost widespread automotive communication network. Due to the increased complexity of our system, a more flexible scheduling approach is required.

Our solution to this problem is a dynamic priority communication scheduling approach that adapts to bandwidth changes on the shared communication network. Through simulations with realistic workloads, we prove the proper functioning of our algorithm with the result that higher critical data streams are favoured over less critical data streams in case of an overloaded system caused by a bottleneck on the CAN bus.

In a long term test-run we prove the stability of our demonstrator, which persisted functioning throughout 88 hours with a CAN bus utilization of 67.80%. Via an external device we also simulated other nodes on the bus.

As a result, we were able to demonstrate the real world applicability of our system, as the demonstrator seeks for maximum CAN bus utilization and therefore extends the utilization limit of 40% in live systems [Davis et al., 2007].

Kurzfassung

Durch die Integration von hochentwickelten Funktionen, wie zum Beispiel Parkassistent, Verkehrszeichen-Detektion oder Fahrerassistenzsysteme (ADAS), wurden moderne Autos zunehmend komplexer. Um mit den Fahrzeug internen Systemen Schritt zu halten müssen somit auch die in Verbindung stehenden Testsysteme erweitert werden.

Historisch bedingt sind Fahrzeug-Testsysteme für eine single-core Architektur ausgelegt. Das allerdings schränkt die Nutzung von gemeinsamen Ressourcen sehr ein und resultiert daher in hohen Hardware Kosten.

In dieser Diplomarbeit präsentieren wir ein umgestaltetes AVL Fahrzeug-Testsystem, welches auf eine Mehrkern-Architektur hochgestuft wurde. Als Teil dieser Umgestaltung haben wir eine Connectivity Manager Komponente für das Testsystem implementiert. Dieser Connectivity Manager ist verantwortlich für das Multiplexen von unterschiedlich kritischen (mixed-criticality) Datenströmen, die von mehreren Kernen stammen, über ein gemeinsames Netzwerk. Insbesondere haben wir das Controller Area Network (CAN) als gemeinsames Netzwerk verwendet, da CAN das meist verbreitetste Kommunikationsnetzwerk im Automotive Bereich ist. Auf Grund der erhöhten Komplexität unseres Systems, musste ein dynamischer Verteilungsalgorithmus (scheduling algorithm) gefunden werden.

Unsere Lösung für dieses Problem ist ein dynamisch priorisierenden Verteilungsalgorithmus, der flexibel auf Bandbreitenveränderungen des gemeinsamen Kommunikationsnetzwerkes reagiert. Durch Simulationen unter realistischen Bedingungen, können wir aufzeigen, dass der Algorithmus ordnungsgemäß funktioniert, sodass hochkritische Datenströme weniger kritischen Datenströmen vorgezogen werden sofern das System auf Grund eines Bottelnecks am CAN Bus überlastet ist.

In einem Langzeit-Testlauf konnten wir die Stabilität unseres Demonstrators, der über 88 Stunden hinweg bei einer CAN Bus Auslastung von 67,80% funktionsfähig blieb, unter Beweis stellen. Des Weiteren konnten wir mittels externen Gerät zusätzliche Busteilnehmer simulieren. Hiermit konnten wir eine reale Anwendbarkeit unseres Systems demonstrieren. Da unser Demonstrator eine maximale CAN Busauslastung anstrebt, kann dadurch die Buslastbegrenzung von 40% in Produktivsystemen erhört werden [Davis et al., 2007].

Acknowledgments

First of all I would like to thank my supervisor Prof. Dr. Marcel Baunach of the Institute of Technical Informatics at the Technical University of Graz, for his tremendous support. Without his great commitment, critical reviewing and advising this research would not have yielded such great findings.

Equally grateful I am for the precious support of Peter Priller. Not just for giving me the opportunity to accomplish this thesis in cooperation with AVL List GmbH, but also for his invaluable dedication and encouragement. His devoted commitment was crucial for the success of this thesis and its scientific contribution. In addition, I would also like to thank Thomas Strunz, Markus Strobich and Georg Macher, for their help and assistance with AVL frameworks and tools. This leads me to the great support of the members of the Technology Research department. I want to thank you for all the constructive discussions and ideas.

Further acknowledgements go to Jürgen Wurzinger and Lisa Kandlhofer, who were not just AVL colleagues, but friends who encouraged and motivated me. Especially our coffee breaks were absolutely inspiring.

Besides these persons I want to thank my friends and flatmates for supporting me and dealing with my moods. Especially, I would like to mention Theresa Rauchberger, who was a really good friend in tough times and provided me with food regularly.

Finally, I would like to thank my family for their patience and support. They always encouraged me to pursue my goals.

Furthermore, research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement 621429 (project EMC2), and from the Endowed Professorship "Embedded Automotive Systems" (bmwfw, AVL List GmbH, and TU Graz).

Contents

List of Figures

Index of Abbreviations

ABS Antilock Braking System

ACC Adaptive Cruise Control

ADAS Advanced Driver Assistant Systems

API Application Programming Interfaces

ASC Automatic Stability Control

CAN Controller Area Network

CIM Computer-Integrated Manufacturing

CRC Cyclical Redundancy Checks

CSMA/CR Carrier-Sense Multiple Access/Collision Resolution

DLC Data Length Code

DM Deadline Monotonic

ECU Electronic Control Unit

EDF Earliest Deadline First

EOF End of Frame

ESP Electronic Stability Program

FIFO First In First Out

GDT Global Descriptor Table

GPOS General-Purpose Operating System

HiL Hardware-in the-Loop System

HMI Human Machine Interface

ICC Inter-Core Communication

IDE IDentifier Extension

IFS Inter Frame Space

IPC Inter-Process Communication

ISO/OSI International Organization for Standardization/Open System Interconnection

ISO International Organization for Standardization

LIN Local Interconnected Network

LVDS Low-Voltage Differential Signaling

MAU Medium Access Unit

MCU Microcontroller

MDI Medium Dependent Interface

MOST Media Oriented Systems Transport

MTS Mixed Traffic Scheduler

NRZ Non Return to Zero

QoS Quality of Service

RED Robust Earliest Deadline

RM Rate-Monotonic

RT Real-Time

RTOS Real-Time Operating System

RTR Remote Transmission Request

SAE Society of Automotive Engineers

SOF Start of Frame

SRR Substitute Remote Request

TDMA Time-Division Multiple Access

TPMS Tire Pressure Monitoring System

V&V Verification and Validation

VMM Virtual Machine Manager

VSEG Virtual Segment

1 Introduction

1.1 Motivation

In the past few years the components within a vehicle shifted increasingly from mechanical to hydraulic and now to electrical components [Leen and Heffernan, 2002]. Nowadays cars can be seen as highly distributed control systems, considering that in modern luxury cars up to 100 electronic control units (ECUs) are interconnected [Albert, 2004], [Charette, 2009]. Due to upcoming features such as automatic parking, traffic sign recognition or advanced driver assistance systems (ADAS), vehicles are becoming increasingly sophisticated. To cope with the rising number of ECUs, while keeping weight, costs, and complexity at a minimum, networks with multiplexed communication over a shared medium were introduced [Navet and Simonot-Lion, 2013].

However, with greater sophistication comes greater complexity of those in-vehicle systems, networks, but also automotive test systems [Kraus et al., 2016]. For the development, verification and validation (V&V) of such complex, distributed control systems, the industries use powerful multicore test systems like HiL (Hardware in the loop) or powertrain test-beds systems. Each test bed is controlled by automotive test systems, which are, amongst other things, responsible for providing the embedded in-vehicle system with simulation data. Additionally, the embedded in-vehicle system returns diagnosis as well as analysis data with the result of a prevalent bidirectional communication [Nager et al., 2017].

Hence, not only the communication within the car is of high relevance, but also the information exchange between in-vehicle communication networks and external real-time test systems. The most common and utmost widespread automotive communication network is the Controller Area

Network (CAN) [Navet and Simonot-Lion, 2013]. There might be various applications in the automotive system which vary in demands on computational power and latency. Multiple applications might run simultaneously, requiring data exchange with varying criticality. Thus, the test system (V&V system) must be capable of dealing with workloads of different criticality (mixed-criticality data) but also with data in different time intervals [Nager et al., 2017].

With this thesis we introduce a Connectivity Manager (CM) for the V&V system that is in charge of multiplexing several data streams across a shared communication network. Equally important for this research is the dynamic priority communication scheduling mechanism that is implemented as part of the Connectivity Manager. In the following chapters the Connectivity Manager approach is described.

1.2 Problem

At AVL List GmbH, the communication among embedded systems and automotive test systems is designed for a single core architecture. This results in a 1:1 relationship between the ECU on the embedded system and the test-application running on the automotive test system. Hence, for every further application, an additional core with a separate network interface would be required. Obviously, this approach would result in high hardware costs, what can be seen as a major problem.

As already mentioned before, in-vehicle systems become increasingly sophisticated. Inherently, this trend is also valid for automotive test systems to keep up with the in-vehicle systems. Therefore, V&V systems are required to be enhanced accordingly, which can be seen as a major challenge of this thesis.

A further challenge, this thesis focuses on, is the variety of data the Connectivity Manager has to deal with. As in automotive systems the data varies in terms of latency and criticality, such as periodic and aperiodic data as well as real-time and non-real-time data, a flexible approach with regard to multiplexing is required.

1.3 Solution

With this thesis we redesigned AVL's status quo system and created an automotive test system that is based on a multi-core architecture. Consequently, this approach enables the sharing of resources, such as memory management, network interfaces, or global wall-clock time. The impact of this redesign is twofold. On the one hand side, this approach effectively reduces the hardware costs since less network interfaces and CPUs are required. Furthermore, the data exchange among cores is enhanced due to our inter-core communication mechanism. On the other hand side, the multi-core architecture likewise increases the complexity of our system.

Our solution to this problem is a dynamic priority communication scheduling algorithm that considers the importance of data streams and is therefore able to handle mixed-criticality data. Furthermore, our dynamic scheduling approach is capable of adapting to bandwidth changes on the network by applying a prioritization policy. Meaning that higher critical data streams are favoured over less critical data streams in case of an overloaded system caused by a bottleneck on the CAN bus.

1.3.1 Objectives

The major goals for this thesis are:

- Upgrading the status quo system to an automotive test system (V&V system) that is based on a multi-core architecture
- Developing a fast and efficient communication mechanism for inter-core communication
- Designing a scheduling algorithm that considers mixed-criticality data and dynamically adapts to bandwidth changes
- Creating a system design that maximizes the sharing of resources, such as memory, network connections, or time management
- Implementing a demonstrator that can be used for evaluation and analysis

- Proving the proper functioning of our scheduling algorithm in the sense that higher critical data streams are favoured over less critical data streams in case of an network bottleneck
- Analysing the performance of the demonstrator in terms of maximum bandwidth utilization, long term stability and minimum cycle-time
- Evaluating the impact of demonstrator with regard to CPU utilization and memory consumption

1.3.2 Outlook on Results

In this thesis we demonstrate a redesigned automotive test system (V&V system) which is based on a multi-core architecture and therefore enables a n:m relationship between the ECU on the embedded system and the test-application running on the automotive test system. Due to the increased complexity of the system, a central managing component, called Connectivity Manager, is introduced. This Connectivity Manager is responsible for providing shared resources as well as managing communications among cores of the test system. To handle mixed-criticality data of the data streams, a dynamic priority communication scheduling mechanism is implemented as part of the Connectivity Manager.

With the help of well-defined test cases, we prove the correct functioning of the Connectivity Manager implementation, referred to as demonstrator. One test case for example yields to find the maximum CAN bus utilization of the demonstrator without generating failures. Moreover, we point out the minimum cycle-time that our demonstrator can successfully handle. Another test case focuses on testing the demonstrator on its long-term stability.

Additionally, through simulations with realistic workloads, we demonstrate that with our dynamic priority communication scheduling approach we are able to multiplex mixed-criticality data streams over a shared CAN bus while dynamically adapting to bandwidth changes.

1.4 Outline

This thesis starts with preliminaries (Chapter 2). This chapter is dedicated to provide a common understanding of terms and definitions, technologies, protocols as well as concepts that are used within this thesis. In Chapter 3 we give a brief discussion on related work that influenced this thesis. Chapter 4 is devoted to introduce our real-time multiplexing component, called Connectivity Manager (CM). In the following sub-chapters the concept, design, and implementation of the Connectivity Manager is discussed. In Chapter 5 the functioning as well as the performance of our demonstrator is evaluated. Finally, Chapter 6 is dedicated to summarize our work and give an outlook on possible future work.

2 Preliminaries

In this chapter the major concepts, technologies and systems are being introduced. Moreover, important terms are defined and an overview of relevant approaches, with regard to real-time scheduling, is given.

2.1 Terminology and Definitions

This section is dedicated to describe different terms and systems, which are used within this thesis.

2.1.1 Electronic Control Unit

ECU stands for Electronic Control Unit, which is a self-contained automotive embedded system. An ECU can be seen as a subsystem composed of a microcontroller, and a set of sensors and actuators [Navet et al., 2005]. ECUs are increasingly used in the automotive domain to control almost every aspect within a car [Mishra and Gurumurthy, 2014].

Historically, the first ECU was introduced in the late 1970s to fulfil the California Clean Air Act (and subsequent federal legislation) for US vehicles. By dynamically measuring the oxygen present in exhaust fumes, the ECU was able to adjust the mixture of fuel and oxygen before combustion. Consequently, this led to improved efficiency and reduced pollution. Due to this applicability, the automotive embedded system was initially called Engine Control Unit. However, the term ECU was generalized, since nowadays these control units operate various aspects of a car and not only the engine [Koscher, 2014]. Commonly, there are multiple ECUs within a car, each with

a different task. Two examples for ECUs are the aforementioned Engine Control Unit and the Speed Control Unit [Kraus et al., 2016].

These days, such embedded systems have been integrated into virtually every aspect of the functioning and diagnostics of a vehicle. Such as throttle, brakes, transmission, telematics, passenger climate, lighting control and entertainment support [Koscher, 2014]. This leads to 30 to 50 ECUs in low-end cars and up to 100 ECUs in a modern luxury sedan [Charette, 2009].

These embedded systems are also used for technologies supporting the driver, passengers and assist in certain driving situations. Those features encompass technologies such as Anti-lock Braking System (ABS), Electronic Stability Control (ESP) or Advanced Driver Assistant Systems (ADAS) [Leen and Heffernan, 2002]. Furthermore, it is common practice to separate the different systems according to their functionality and temporal demands [Kraus et al., 2016]. This system distribution can further be seen in Section 2.2.

As it can be seen in Figure 2.1, since 1978, ECUs are being increasingly used for multiple applications within a vehicle. Further, this illustration emphasises that modern vehicles are mainly based on electronic components rather than on mechanical or hydraulic components.

2.1.2 ECU System

Within this thesis we differentiate between ECU System and V&V system. As already mentioned above, ECU stands for Electronic Control Unit which is an automotive embedded system composed of a micro-controller and a set of sensors and actuators. The ECU system, in particular the specific ECUs of the system, is responsible to independently control almost every aspect within a vehicle without requiring external input. Thus, the ECU system consists of multiple ECUs connected through a in-vehicle network, such as the Controller Area Network (CAN). Hence, the ECU system can be seen as an independently functioning system that is embedded into the vehicle, also referred to as automotive embedded in-vehicle system.

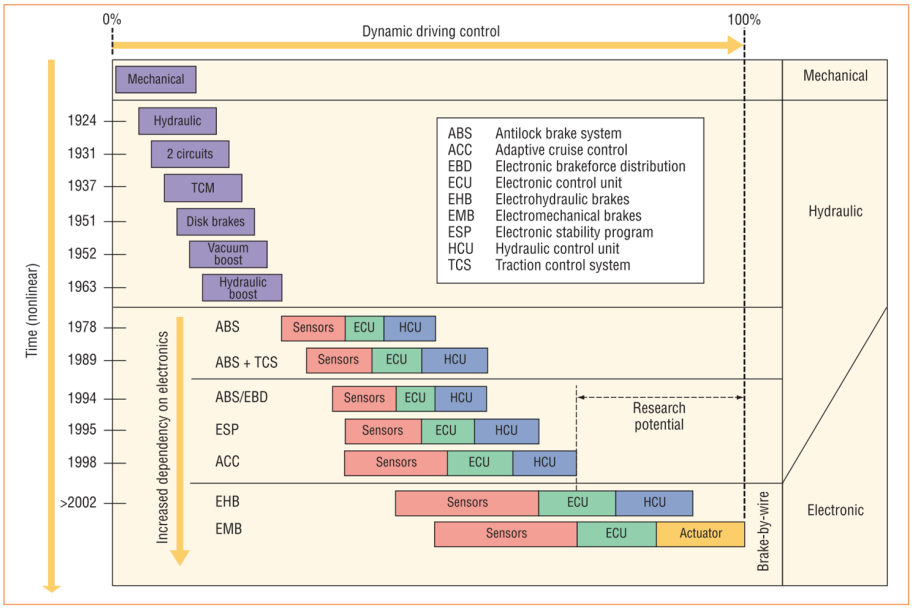


Figure 2.1: The increasing usage of ECUs presents a clear shift from mechanical to hydraulic components to almost exclusively electrical components [Leen and Heffernan, 2002].

In the literature, a general definition of an embedded system is as follows. According to the definition of Li Q. and Yao C. [Li and Yao, 2003], an embedded system is a computing system with tightly coupled hardware and software integration. Furthermore, embedded systems are designated to perform a dedicated function, whereas the term *embedded* reflects the fact that these systems are usually part of a larger system, the so called *embedding system*.

With regard to our research, this definition acknowledges that one particular ECU can be seen as the *embedded system* which is part of the larger in-vehicle distributed ECU system (*embedding system*). Since this thesis mainly focuses on the automotive test system (V&V system), which is the system that tests and analyse the ECU systems, we specifically do not further focus on the embedded in-vehicle system.

2.1.3 V&V System

While the ECU system is in charge of controlling the vehicle, the V&V system is responsible for *validating* and *verifying* the embedded in-vehicle systems. For the development, verification and validation (V&V) of such complex, distributed control systems, the industries increasingly use powerful (multi-core) test systems like HiL (Hardware-in the-loop) or powertrain test-bed systems. Each test bed is controlled by automotive test systems, which are, amongst other things, responsible for providing the embedded in-vehicle system with input data, such as simulation scenarios. Additionally, the embedded in-vehicle system returns diagnosis as well as analysis data with the result of a prevalent bidirectional communication [Nager et al., 2017].

Moreover, also Lawrenz W. [Lawrenz, 2013] describes HiL as a method for testing and validating embedded systems by an existing hardware that can be integrated into a simulated system to test the functionality.

Since AVL's V&V systems are neither embedded into the vehicle nor are they composed of micro-controllers, a regular x86 processor architecture usually builds the foundation. To fulfil the real-time demands of the automotive micro-controllers, a real-time operating system (RTOS) is used instead of a regular operating system (OS). Further information regarding the architecture of our V&V system is discussed in Section 4.4.1.

2.2 In-Vehicle Communication Systems

At the very beginning of automotive electronics, every function was implemented as stand-alone electronic control unit. However, this approach appeared to be insufficient with the need for functions to be distributed over several ECUs. Moreover, information exchange among functions was also a considerable need [Navet and Simonot-Lion, 2013]. For example, the vehicle speed estimated by the engine controller, needs to be known by the steering module to adapt the steering effort or to control the suspension. This information exchange can lead to 2500 shared variables, so called "signals", exchanged between up to 100 ECUs in modern luxury cars [Albert, 2004], [Charette, 2009].

In the past, ECUs exchanged data through point-to-point links, which required a vast number of communication channels and complex wiring. This strategy was unable to cope with the increasing number of ECUs due to the problem of weight, cost, complexity and reliability. Motivated by these issues, the use of a network where the communication channels are multiplexed over a shared medium, was aimed for [Navet and Simonot-Lion, 2013]. Since the early 1980s, almost all automobile manufacturers had started intensive efforts to find or develop a suitable communication protocol which fulfils the increasing need for in-vehicle communication. A further requirement was to reduce not only the weight but also the impending increase in the complexity of the wiring harness in the car [Lawrenz, 2013].

Since the type of data as well as the time criticality differs from application to application, modern cars use more than one communication system to fulfil bandwidth requirements and reduce costs at the same time. For this reason, the Society of Automotive Engineers (SAE) defined a classification for automotive communication protocols in 1994. Four automotive network categories were defined, namely Class A, Class B, Class C, and Class D, which can be seen in Table 2.1 [Navet and Simonot-Lion, 2013].

In 1998, Motorola reported that replacing wiring harnesses with LANs in the four doors of a BMW reduced the weight by 15 kilograms while functionality could be enhanced [Leen and Heffernan, 2002].

In consideration of the circumstances that there are different classes of applications, different bus protocols have emerged, such as the Controller Area Network CAN, Local Interconnected Network (LIN), Media Oriented Systems Transport (MOST) and FlexRay [Lawrenz, 2013]. In 2008, BMW released the BMW 7 series, which implements four CAN buses, a FlexRay bus, a MOST bus, several LIN buses, an Ethernet bus and also wireless interfaces [Navet and Simonot-Lion, 2013].

As it can been seen in Table 2.1, Class A networks have a data rate lower than 10 kbit/s and are used to transmit simple control data, such as trunk release, seat control, door lock or lightning. LIN is one of the most common representative of Class A networks [Navet and Simonot-Lion, 2013], [Lawrenz, 2013].

Table 2.1: Classification of the four automotive networks with their applicability [Leen et al., 1999].

Network classification	Speed	Application
Class A	<10 kbit/s low speed	convenience features, e.g. trunk release, electric mirror adjustment
Class B	10-125 kbit/s medium speed	general information transfer, e.g. instruments, power windows
Class C	125 kbit/s -1 Mbit/s high speed	real-time control, e.g. power train, vehicle dynamics
Class D	>1 Mbit/s	multimedia applications, e.g. Internet, digital TV hard real-time critical functions, e.g. X-by-wire applications

Class B networks operate at medium speed, which is specified with a data rate between 10 kbit/s and 125 kbit/s. The main purpose of Class B networks is to reduce the number of sensors by supporting data exchange between ECUs. The most common used network for this classification is the low-speed CAN bus [Navet and Simonot-Lion, 2013], [Leen and Heffernan, 2002].

Class C networks, that operate at a data rate between 125 kbit/s and 1 Mbit/s, are used for real-time control applications, such as power train control and vehicle dynamics. Data transmitted on a Class C network, has high demands concerning real-time and failure resistance. Mostly a high-speed CAN bus is used to fulfil the requirements of a Class C network [Tuohy et al., 2015].

For the Class D networks, the SAE has not yet defined specific standards. However, networks that exceed a data rate of 1 Mbit/s are classified as Class D networks [Leen et al., 1999]. This classification is most relevant for

multimedia applications and hard real-time critical functions. For this type of application the MOST protocol, with a bandwidth of up to 150 Mbit/s, is primarily used [Tuohy et al., 2015].

The most common automotive networks are outlined in Table 2.2.

Table 2.2: Current automotive network technologies with corresponding maximal bit rate [Tuohy et al., 2015].

Protocol	**Maximal bitrate**	**Medium**	**Protocol**
LIN	19.2 kbit/s	Single wire	Master/Slave
CAN	1 Mbit/s	Twisted pair	CSMA/CR
FlexRay	20 Mbit/s	Twisted pair/Optical fibre	TDMA
MOST	150 Mbit/s	Optical fibre	TDMA
LVDS	655 Mbit/s	Twisted pair	Serial/Parallel

All functions embedded within a vehicle differ in terms of performance, timeliness but also in safety demands. Therefore, different QoSs, such as response time, jitter, bandwidth, redundant communication channels for tolerating transmission errors, and efficiency of the error detection mechanism, are required. For this reason, the embedded in-vehicle system is commonly divided into several functional domains that correspond to different features, constraints and demands [Navet et al., 2005].

- **Powertrain**: Through micro-controllers with high computing power and several complex control laws with sampling rates of the order of milliseconds, the powertrain domain is responsible for controlling the engine. Due to the high rotation speed of the engine, strict timing constraints are imposed on the scheduling of critical tasks. Furthermore, frequent data exchanges with other vehicle domains are required. For example data exchange with the *chassis* domain to enable ABS and ESP functionality and also data exchange with the *body* domain to provide climate control and dashboard information [Navet and Simonot-Lion, 2013], [Navet et al., 2005].
- **Chassis**: The main function of the chassis domain is to gather information concerning steering/braking solicitations and driving conditions (ground surface, wind, etc.) to provide functionalities such as ABS, ESP, ASC (Automatic Stability Control), or 4WD (4 Wheel Drive). Since

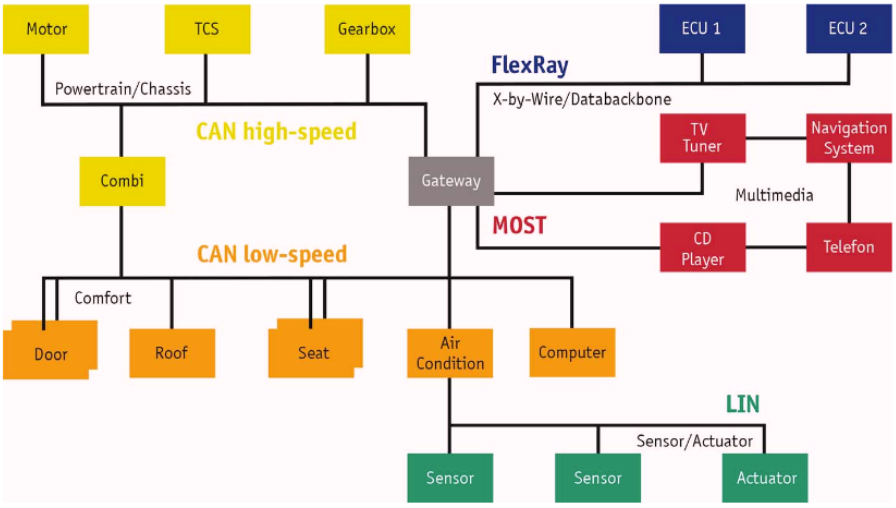


Figure 2.2: Electronic architecture of a vehicle that is divided into different functional domains. Each domain, such as Powertrain, Chassis, Body, and Multimedia, consist of different ECUs and network protocols [Mayer, 2006].

the chassis domain has a strong impact on the vehicle's stability, agility and dynamics, the communication of the chassis functions are more critical from a safety standpoint.
Furthermore, the chassis domain introduces the "X-by-wire" technology, which refers to the replacement of mechanical or hydraulic systems by fully electrical/electronic ones. The powertrain and chassis functions operate mainly as closed-loop control systems with a time-triggered implementation to ensure deterministic real-time behaviour of the system [Navet and Simonot-Lion, 2013], [Navet et al., 2005].

- **Body**: The body domain consists of software-based systems that control dashboard, wipers, lights, doors, windows, seats, mirrors, and climate control. Functions within the body domain require many exchanges of small pieces of information among themselves. Therefore, not all nodes require a large bandwidth, which led to the introduction of low-cost networks such as Local Interconnect Network (LIN). The activation of body functions is mainly event-triggered, for example driver/passenger opens a door [Navet and Simonot-Lion, 2013], [Navet et al., 2005].
- **Telematics**: Telematics functions provide functionality such as in-car

navigation system or remote vehicle diagnostics. In general, telematics functions, such as vehicle monitoring, wireless communication, and location devices, have a high demand on deadline constraints, since applications engaged in this domain, rely on real-time data [Navet and Simonot-Lion, 2013], [Navet et al., 2005].

- **Multimedia**: In comparison to the telematics domain, applications of the multimedia domain do not require real-time information exchange. The multimedia domain is responsible for rear seat entertainment, CD, DVD, and hands-free phone control. Furthermore, this domain integrates human-machine interface (HMI) and defines multimedia QoS, wherever preserving the integrity and confidentiality of information is crucial [Navet and Simonot-Lion, 2013], [Navet et al., 2005].
- **Safety**: An emerging domain, usually referred to as "active and passive safety" domain, focuses on functions ensuring the safety of the occupants. Electronic-based systems, such as Tire Pressure Monitoring System (TPMS), Adaptive Cruise Control (ACC), deployment of airbags, or impact and roll-over sensors, are increasingly embedded within vehicles [Navet and Simonot-Lion, 2013], [Navet et al., 2005].

Today's vehicles commonly consist of an electronic architectures that includes four different types of networks interconnected by gateways. Such an electronic architecture as well as the different domains can be seen in Figure 2.2. Within the Volvo XC90 for example, up to 40 ECUs are interconnected by a LIN bus, a MOST bus, a low-speed CAN, and a high-speed CAN [Navet and Simonot-Lion, 2013]. Most likely, a bus dedicated to occupant safety systems such as the "safe-by-wire plus" will be added in the near future [Navet and Simonot-Lion, 2008].

Since the practical part of this thesis was mainly developed for the CAN bus, the concept of the Controller Area Network is discussed in more detail.

2.3 Controller Area Network

The Controller Area Network (CAN) is an automotive-specific multicast-based communication protocol developed by Robert Bosch GmbH, to provide a cost-effective communications bus for automotive applications [Tuohy

et al., 2015], [Di Natale et al., 2012]. Typically, CAN is used to exchange control traffic between ECUs within a vehicle. However, it is also used in factory and plant controls, in robotics, medical devices, and also in some avionics systems [Johansson and Martin, 2005].

2.3.1 History and Standardization

In 1986, the Controller Area Network was officially released by Robert Bosch GmbH. Although the first CAN chip was offered to car manufacturers in the late 1980s by Intel, the eventual standardization of CAN was in November 1993 [Lawrenz, 2013]. CAN on a twisted pair of copper wires became an ISO standard and is now a de-facto standard in Europe for data transmission in automotive applications [Navet et al., 2005].

The ISO family 11898 describes the architecture of CAN in terms of the layers of the International Organization for Standardization/Open System Interconnection (ISO/OSI) model. While ISO 11898-1 specifies both parts of the physical layer and parts of the data link layer for transmission rates up to 1 Mbit/s, ISO 11898-2 focuses on high-speed medium access unit (MAU) and some medium dependent interface (MDI) features. Further extensions of the ISO 11898 family describe features such as low-speed, faul-tolerant, time-triggered communication, or power saving mode. CAN is also specified by the Society of Automotive Engineers (SAE) by the standard J2284-1 to -3.

In 1992, CAN was used first in the Mercedes S-Class and served there as a high-speed network for communication between engine control, transmission control and dashboard. Simultaneously, a low-speed CAN bus was used for distributed climate control. Shortly after, various manufacturers such as BMW, Porsche, and Jaguar, put CAN into series-production. Since 1994-1995, CAN is the most widely used communication protocol for automotive applications [Lawrenz, 2013].

The latest version of the CAN protocol is the version 2.0, which is further on divided into 2.0a and 2.0b. Both specifications are identical, except for the length of the message identifier (ID). Specification 2.0a defines a 11-bit message identifier that is known as the standard frame format. The second

specification 2.0b defines the extended frame format which uses a 29-bit message ID [Navet et al., 2005].

Each CAN message is labelled by an ID, that is transmitted within the message and is unique to the whole system. This serves two purposes, on the one hand the identifier can be used for giving priority for transmission, since the lower the numerical value of the identifier, the greater is the priority of the message. On the other hand unique IDs enable message filtering upon reception [Navet et al., 2005].

With regard to this thesis and also in general, both CAN versions, specification 2.0a and specification 2.0b, can be used simultaneously.

2.3.2 Characteristics

CAN is a contention-based multi-master network that efficiently handles data which needs to be transmitted periodically, aperiodically, or on demand. Due to CAN's collision resolving algorithm, a high schedulable utilization and guaranteed bus access latency of less than 150 μs for the highest-priority message on a 1 Mbit/s bus, can be achieved [Zuberi and Shin, 1995], [Navet et al., 2005]. However, for lower priority messages, a bus access latency of less than 150 μs cannot be guaranteed.

In the CAN specification in its version 2.0 by Bosch R. [Bosch, 1991] in 1991, the following properties are being stated:

- Prioritization of messages
- Guarantee of latency times
- Configuration flexibility
- Multicast reception with time synchronization
- System wide data consistency
- Multimaster
- Error detection and signalling
- Automatic retransmission of corrupted messages as soon as the bus is idle again
- Distinction between temporary errors and permanent failures of nodes and autonomous switching-off of defect nodes

Further characteristics of CAN are its high reliability in noisy environments through CRC checks and bit-stuffing, as well as its reconfiguration flexibility [Zuberi and Shin, 1995].

2.3.3 Arbitration

The fact, that any network node can start to transmit a message once the CAN bus is free, raises the problem of possible conflicts. The CAN protocol follows the *Carrier Sense Multiple Access* with *Collision Detection* and *Arbitration on Message Priority* (CSMA/CD+AMP) approach. All nodes have to observe the bus (*Carrier Sense*) and wait until it is in idle state before writing onto the bus. Whereas writing onto the bus can be initiated by multiple nodes simultaneously (*Multiple Access*). Conflicts of simultaneously transmitting units are detected (*Collision Detection*) and resolved by bit-wise arbitration (*Arbitration on Message Priority*) using the identifier of each unit [Johansson and Martin, 2005], [Lawrenz, 2013].

The arbitration on the CAN bus relies on the fact that a sending node monitors the bus while transmitting. Therefore, the signal must be able to propagate to the most remote node and return back before the bit value is decided. This, by implication, requires the bit time to be at least twice as long as the propagation delay that influences the data rate [Navet and Simonot-Lion, 2013].

Hence, a dependency between the bit time and the signal propagation delay clearly exists. Meaning that the maximum achievable bit rate depends on the length of the CAN bus [Di Natale et al., 2012].

For example, at a bus length of 40 meters, 1 Mbit/s is feasible. Whereas, lowering the data rate to 250 kbit/s, the bus length can be extended up to 250 meters [Navet and Simonot-Lion, 2013]. In Table 2.3 a more detailed listing can be observed.

In the idle state, the CAN bus has a voltage of 2.5 V, which is also called the recessive state of the CAN bus and is represented by a logical ”1”. In the dominant state, the idle voltage of 2.5 V is changed by 1 V. Meaning that the dominant state of a high-speed CAN (CAN_H) is indicated by the voltage

Table 2.3: Typical bit rates in respect to bit time and CAN bus length [Di Natale et al., 2012].

Bit rate	Bit time	Bus length
1 Mb/s	1 μs	25m
800 kb/s	1.25 μs	50m
500 kb/s	2 μs	100m
250 kb/s	4 μs	250m
125 kb/s	8 μs	500m
62.5 kb/s	16 μs	1000m
20 kb/s	50 μs	2500m
10 kb/s	100 μs	5000m

of 3.5 V, and for a low-speed CAN (CAN_L) the voltage is at 1.5 V. For both versions, the dominant state is represented by a logical ”0” [Mischo et al., 2015], [Di Natale et al., 2012]. This specification can be seen in Figure 2.3.

Every time a ”0” bit level was sent by one of the nodes, the bus is in the dominant state regardless if other nodes transmitted the ”1” bit level. Therefore, ”0” is the dominant bit value, while ”1” is the recessive bit value [Navet and Simonot-Lion, 2013], [Zuberi and Shin, 1995].

Whenever a node transmits a recessive bit onto the bus while another node transmits a dominant bit, the resulting bus level is dominant due to the logical ”and” operation realized by the physical layer. Hence, every node sending a recessive bit but detects a dominant bit on the bus, immediately stops transmitting [Navet and Simonot-Lion, 2013].

Identifiers are located at the beginning of each message and transmitted ”*most significant bit first*”. Thus, the node with the numerically lowest identifier will gain bus access. Nodes that have lost arbitration wait with the retransmits until the CAN bus is free again [Navet and Simonot-Lion, 2013].

The CAN transceiver, on the receiver side, provides the recessive signal level and protects the controller chip input comparator against excessive voltages on the bus lines [Di Natale et al., 2012].

In Figure 2.4, an example arbitration can be seen. The bit streams *S1-S3*, which are outlined in Table 2.4, are simultaneously competing for the CAN

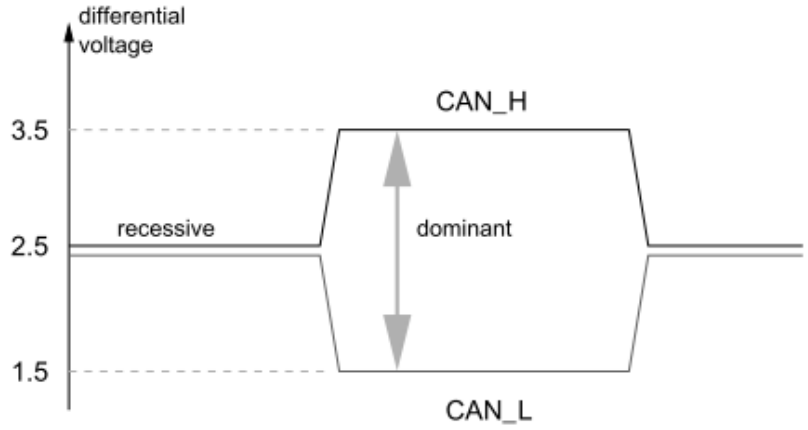


Figure 2.3: Voltage levels of the CAN bus [Di Natale et al., 2012].

bus, so that a bit-wise arbitration based on the identifier is performed.

Table 2.4: Concrete bits of the three bit streams *S1-S3* [Johansson and Martin, 2005].

Bit stream	**Bits**
S1 (Node 1)	11001101010
S2 (Node 2)	11001011011
S3 (Node 3)	11001011001

The bus acquisition algorithm works in the manner, that messages with their associated ID are transferred to the bus interface chip, which waits until the bus is idle. Once the bus is in idle state, the bus interface chip writes the ID onto the bus, one bit at a time, starting with the most significant bit. After each written bit, all interface chips on the bus wait long enough for signals to propagate along the network, before reading the bus. If the writing chip had written a recessive bit but reads a dominant bit, which means that there is another node on the bus with a message that is of higher priority, the chip drops out of contention [Zuberi and Shin, 1995].

A major advantage of CAN's arbitration mechanism is, that neither address data nor extra bus control information needs to be sent. Hence, every node picks up all traffic from the bus and filters out the relevant messages.

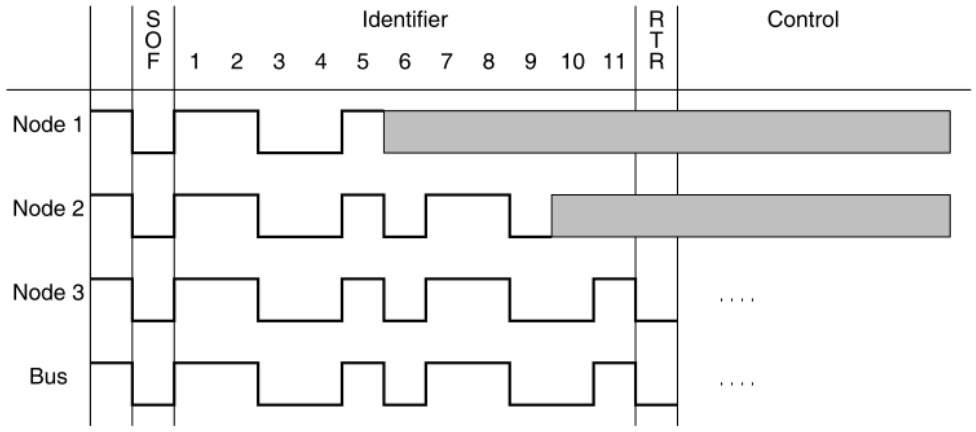


Figure 2.4: Arbitration example of three bit streams (Nodes) competing for transmission onto the CAN bus. Node 3 manages to transmit onto the CAN bus after succeeding in the arbitration due to exceeding dominant bits "0" [Johansson and Martin, 2005].

Consequently, units with a low priority may experience large latency if high-priority units are very active. That represents a crucial drawback of the CAN arbitration mechanism [Johansson and Martin, 2005].

2.3.4 Bit Coding

As for the bit coding, the CAN protocol makes use of the Non Return to Zero (NRZ) method to encode the bit stream on the bus line. This method has the advantage that only a minimum bandwidth for signal transmission is required. However, the NRZ encoding, where the bit level is constant during the bit time, contains no information about the bit clock, which may cause problems with synchronization and thus lead to erroneous bit detection [Di Natale et al., 2012].

The bit time, is the time between the emission of two successive bits of the same frame. Nodes need to resynchronize periodically to not lose the bit time. Hence, long sequences without bit transitions should be prevented to avoid drifts in the node clocks [Navet et al., 2005], [Di Natale et al., 2012].

For this reason, the *bit-stuffing* mechanism was introduced. The stuffing method requires the transmitter, after having sent five consecutive bits of identical value, to insert ("*stuff*") an additional bit of inverse value into the bit stream. A sequence of five consecutive bits of identical value is then recognized by the receiving side, so that the following bit, the *stuff bit*, can be removed before processing the content of the frame. Thus, the maximal distance between signal edges in a bit stream is five bits, which is a compromise between the lengthening of the frames and the tolerance of synchronization and drifts [Lawrenz, 2013], [Di Natale et al., 2012].

2.3.5 Frames Types

The CAN protocol uses four different types of frames: Data Frame, Error Frame, Remote Frame, and Overload Frame. While the data frame is the only frame that actually transports message data, all other frames are for fault containment, triggering and synchronization [Lawrenz, 2013], [Di Natale et al., 2012].

- **Data Frame**: Carries data from a transmitter to possibly multiple receivers.
- **Remote Frame**: Is transmitted by a bus node to request the transmission of the Data Frame with the same identifier.
- **Error Frame**: Is transmitted by any node on detecting a bus error.
- **Overload Frame**: Is used to provide an extra delay between the preceding and the succeeding Data or Remote Frames [Bosch, 1991], [Di Natale et al., 2012].

Since the *Data Frame* is the most relevant frame type within this thesis, it is discussed in more detail while the other frame types can be largely neglected.

Data Frame

Data Frames are used to carry data from a transmitter to one or more receivers. According to the specification by Bosch R. in 1991, a Data Frame

is composed of seven different bit fields: Start of Frame, Arbitration Field, Control Field, Data Field, CRC Field, ACK Field, and End of Frame. With reference to Figure 2.5, the Data Field can also be of length zero [Bosch, 1991].

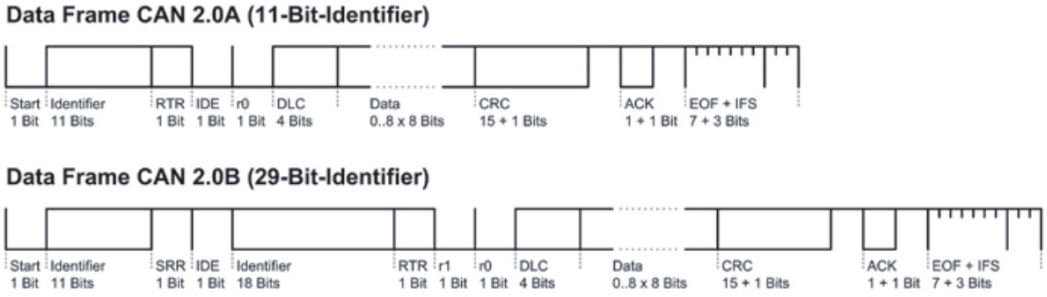


Figure 2.5: Detailed illustration of data frames with different formats, the standard CAN frame format and the extended CAN frame format. Additionally all relevant fields including their length can be seen [Lawrenz, 2013].

- **Start of Frame (SOF)**: This single bit represents the very beginning of every CAN message. It is always set to ”0” (dominant) so that the bus idle state of ”1” (recessive) is overruled [Lawrenz, 2013].
- **Arbitration Field**: The Arbitration Field consists of the Identifier Field and the RTR (Remote Transmission Request) bit. However, there are two different formats of CAN messages, as already outlined in Section 2.3.1. CAN messages with the standard frame format use an Identifier Field with a length of 11 bits plus 1 additional bit. The Identifier Field of extended CAN frames is 29 bits plus 3 additional bits long. Both, the standard frame and the extended frame, start with the 11 bits (the most significant bits in chase of the extended format) of the identifier, followed by the RTR bit in the standard format and by the SRR (Substitute Remote Request) in the extended format. The RTR bit is used to distinguish Data Frames from Remote Request Frames. In case of Data Frames it is set to ”0” (dominant), and conversely it is set to ”1” (recessive) in case of Remote Request Frames. The SRR in the extended frame format is always set to ”1” (recessive) and solely a placeholder for guaranteeing the deterministic resolution of the arbitration between standard and extended frames. The extended frame continues with a single IDE (IDentifier Extension) bit, which

is always recessive, followed by the remaining 18 least significant identifier bits and the RTR bit.

- **Control Field**: With regard to the standard frame format, the IDE bit is part of the Control Field and always dominant. Thus, the IDE bit can be used for the distinction between the standard frame format and the extended frame format [Di Natale et al., 2012]. Furthermore, the IDE bit indicates that the identifier is completed. While the Control Field of the extended format contains two reserved bits (r1, $r0$), the standard format has only one reserved bit ($r0$). The last four bits of the Control Field are dedicated to the DLC (Data Length Code), that defines the length of the following Data Field.
- **Data Field**: The Data Field contains the actual data of the message, whereas the length may vary between 0 and 8 byte depending on the Data Length Code [Bosch, 1991].
- **Cyclic Redundancy Check (CRC) Field**: This field contains the checksum for the preceding bits of the frame, such as SOF, Arbitration Field, Control Field, Data Field (if present). Hence, every transmitted frame provides a 15-bits-long checksum (CRC Sequence) that can be used to verify the correctness of the frame. This mechanism, however, is only used for fault detection, not for error correction. The calculation of the CRC polynomial enables the detection of up to five single-bit errors in one message. Moreover, so called burst-errors can be recognized up to the length of the CRC Sequence [Lawrenz, 2013]. A more detailed explanation of this polynomial calculation can be found in the CAN specification by Bosch R. [Bosch, 1991].
  Each receiving node independently recalculates the CRC checksum, based on the received message, and compares it to the CRC Sequence provided in the message. If an error is detected, an automatic and instant retransmission of the incorrect message is performed. This error detection mechanism leads to a high data integrity and a short error recovery time [Johansson and Martin, 2005]. The CRC Delimiter, which is set to recessive, completes the CRC Field.
- **Acknowledge Field**: The Acknowledge Field is 2 bits long, whereas the first bit represents the actual acknowledgement (ACK Slot) and the second bit is always set to "1" (recessive) and serves as a delimiter. The reception of a syntactically correct message on the CAN bus is acknowledged by all nodes by sending a dominant value in the ACK

Slot. The transmitting node sends out the message with a recessive value in the ACK Slot and expects that this recessive level is overwritten by a dominant value. A missing dominant value in the ACK Slot is considered to be an acknowledgement error by the transmitter of the message. A successful detection of the dominant value in the ACK Slot does not mean that the frame was received by all nodes, however, it guarantees that at least one node received it [Lawrenz, 2013].

- **End of Frame (EOF)**: The frame is concluded with 7 recessive bits representing the End of the Frame. As already mentioned before, bit-stuffing is not applied in this section, so that the end of the frame can be recognized [Lawrenz, 2013].
- **Inter Frame Space (IFS)**: The Inter Frame Space consists of 3 bits which are used to separate the current frame from the following frame. This period is also used for transferring a correctly received message from protocol controller into the receive buffer, or for transferring a message from transmit buffer to the protocol controller [Davis et al., 2007], [Lawrenz, 2013].

In consideration of Figure 2.5, *bit-stuffing* is applied to the fields from *Start of Frame* to the *CRC* (Cyclic Redundancy Check) Field, with reference to Data and Remote Frames. Bit-stuffing is not applied in any other field and also not in Error or Overload Frames [Lawrenz, 2013].

2.4 Real-Time Systems and Environment

To create a common understanding, the terms real-time system as well as real-time environment are briefly discussed.

A real-time system is a computerized system with explicit deterministic or probabilistic timing requirements [Sha et al., 2004], [Risat, 2010]. Real-time computer systems are required to react to events within time intervals dictated by its environment. This instant, at which the event or result has to be processed, is called *deadline*. The deadline, which is a timing information, is a common characteristic of many real-time systems. If a real-time system consists of a set of nodes that are interconnected by a real-time

Table 2.5: Hard real-time versus soft real-time systems [Kopetz, 1997].

Characterisitic	**Hard real-time**	**Soft real-time**
Response Time	hard-required	soft-desired
Peak-load Performance	predictable	degraded
Control of Pace	environment	computer
Safety	often critical	non-critical
Size of Data Files	small/medium	large

communication network, it is called a distributed real-time system[1] [Kopetz, 1997]. Furthermore, real-time systems can be classified as *hard* real-time systems or *soft* real-time systems, which also applies for the classification of deadlines.

In hard real-time systems, a set of concurrent real-time tasks must be processed in such a way, that their specified deadlines are met. In contrast to this, in a soft real-time system deadlines should be met but can also be violated without a catastrophic result [Audsley and Burns, 1990].

From the design perspective, hard real-time systems and soft real-time systems distinguish themselves fundamentally. While the design of hard real-time systems has to consider guaranteed temporal behaviour for all specified load and fault conditions, for a soft real-time system it is acceptable to miss a deadline occasionally [Kopetz, 1997].

A brief summary of the differences between hard and soft real-time systems is outlined in Table 2.5.

Response Time: Applications designated for hard real-time systems often demand a response time in the order of milliseconds or less. Thus, a hard real-time system must be highly autonomous to ensure a flawless processing without human interaction. In contrast, the response time requirements of soft real-time systems are in the order of seconds, so that human interaction is not precluded. Consequently, if a deadline of a soft real-time application is missed, the system is not compromised in this way.

[1] INtime for instance can be operated as a distributed real-time system (more details in Section 2.6).

Peak-load Performance: While in soft real-time systems degraded operations in rarely occurring peak load scenarios are tolerated for economic reasons, hard real-time systems must guarantee by design that the computer system meets the specified deadline in all situations. For soft real-time systems the average performance is considered, whereas for hard real-time systems the performance in all possible peak-load scenarios must be predictable.

Control of Pace: Hard real-time systems are paced by the state changes occurring in the environment and thus remain synchronous with the environment (controlled object and human operator). Soft real-time systems on the other hand can influence the pace of the environment in case of overload situations. For example, a reservation system that cannot keep up with the demands of the operators, simply extends the response times and forces the operators to slow down.

Safety: While safety violations in a hard real-time system may lead to catastrophic results, soft real-time systems tolerate violations due to non-critical consequences.

Size of Data Files: Since the temporal accuracy of information is invalidated by the flow of time, the key concern in hard real-time systems is on the *short-term* temporal accuracy of rather small data sets. This is in contrast to soft real-time systems, where the long-term integrity of large data sets is the key issue [Kopetz, 1997].

2.5 Real-Time Scheduling

The term "real-time scheduling" refers to the process of calculating a schedule for a set of competitors that share the same limited resource in a real-time system. Mostly a scheduler component is in charge for this calculation by providing an algorithm or policy for ordering [Audsley and Burns, 1990].

In the literature, many articles dealing with real-time scheduling can be found. Most of them focus on process scheduling, in which the processor (-time) is the limited shared resource and the competing processes are the competitors.

However, within this thesis we focus on message scheduling, since the shared resource is a network and the competing competitors are data streams [Natale and Meschi, 2001], [Zuberi and Shin, 1995], [Tindell and Hansson, 1994]. Whereas data streams are understood as active connections of an application.

With regard to the literature, all concepts and principles described in the following sections, are focused on process (task) scheduling, however, are also valid for message scheduling unless stated elsewise. Figure 2.6 illustrates the classifications of real-time scheduling.

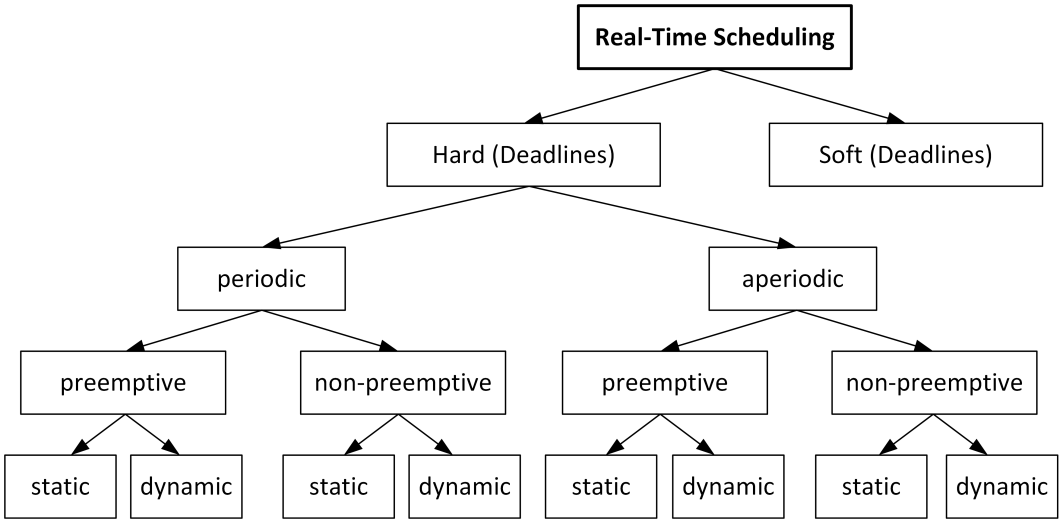
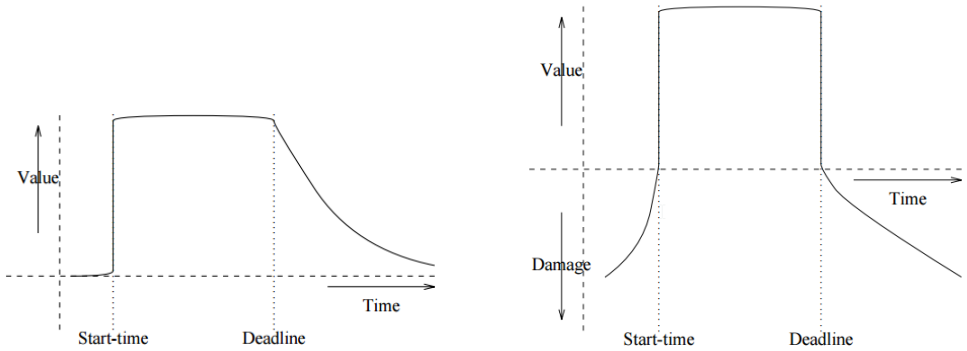


Figure 2.6: Classification of real-time scheduling [Kopetz, 1997].

With regard to Figure 2.6, the first distinction is based on the timing information. Time-critical competitors are required to provide timing information, in the form of a deadline, to meet real-time constraints. To ensure a proper functioning of the scheduler, the competitors, regardless if tasks/processes or messages, must have a start-time as well as a deadline. The start-time is the instant of the occurrence of an event and is mostly represented as an absolute value (timestamp). The deadline is dated in the future and can either be an absolute value (timestamp) or a relative value. Irrespectively, whether relative or absolute value, the deadline is always in relation to the start-time. Not only real-time systems but also deadlines can be divided

into hard real-time deadlines and soft real-time deadlines [Audsley and Burns, 1990], [Kopetz, 1997].



(a) Soft Deadline

(b) Hard Deadline in a saftey critical system

Figure 2.7: Comparison between soft and hard deadlines with regard to damage in case of violating deadlines [Audsley and Burns, 1990].

2.5.1 Hard Deadlines

For a competitor with a hard deadline, it is a necessity that the processing is completed before its respective deadline [Kopetz, 1997]. In case of not meeting its hard deadline, the consequences have the potential to be catastrophic, as outlined by Figure 2.7. As already outlined in Section 2.4, a hard real-time system is one, in which the consequences of violating the hard deadline are much greater than any benefits provided by the service being delivered in time [Audsley and Burns, 1990].

Thus, the following assumption applies for hard real-time events [Audsley and Burns, 1990]:

$$C \leq D \tag{2.1}$$

C : *computation-time*
D : *deadline*

Which means, that the *computation-time* must not be longer than the *deadline*, given that both values are relative to the same start-time. Examples for such systems, which use a hard deadline, are air-traffic controls, heart pacemaker as well as engine controllers in cars. Further on, more and more embedded systems can be considered as hard real-time systems [Kopetz, 1997], [Audsley and Burns, 1990].

2.5.2 Soft Deadlines

Even if a system is considered to be a hard real-time system, not all computational events will be hard or critical. Soft, or also called non-critical, real-time deadlines can be missed without compromising the integrity of the system [Audsley and Burns, 1990].

Often soft and hard real-time deadlines are coexistent in real-time systems. While for time critical events, such as engine control, a hard deadline is necessary, for non-critical events, for example house-keeping tasks, a soft deadline is sufficient [Sprunt, 1990], [Audsley and Burns, 1990], [Spuri and Buttazzo, 1994].

With reference to our research, we consider a system in which both, hard and soft deadlines, are being used. Consequently, our system can be seen as a *mixed real-time system*. Further details about the communications model can be found in Section 4.2.2.

2.5.3 Workload Characteristics

In the automotive domain as well as in Computer-Integrated Manufacturing (CIM), computing devices such as controllers, actuators and sensors are being used. Due to this variety of devices, several message types must be considered. While some devices exchange messages periodically, others are more event-driven, such as smart sensors. Additionally, status information is usually exchanged without timing constraints, which leads to the classification of three message types [Zuberi and Shin, 1995]:

1. Hard deadline periodic messages

2. Hard deadline aperiodic messages
3. Non-real-time (best effort) aperiodic messages

The next distinction in the classification Figure 2.6, concerning real-time scheduling, is periodic or aperiodic.

Periodic Messages

In the context of real-time scheduling, an event is called periodic when it occurs repeatedly at regular time intervals to the extend of a certain tolerance. Usually, the period of the event is well known, thus it can be scheduled in advance [Risat, 2010]. Periodic events can be considered as time-critical, in the sense that the system cannot function without punctual completion. In the automotive application, if the antilock braking system is not activated within a short time interval after a wheel is locked, the vehicle is likely to become uncontrollable [Shin and Ramanathan, 1994].

Periodic messages are defined as information which is periodically exchanged. Since the functioning of one or more listening devices depends on the timely transmission of the message, the deadline of the message is crucial. Hence, for time critical functions a hard deadline is used. This kind of message type is usually used for sensor-based systems in the automotive and robotic domain [Zuberi and Shin, 1995].

With reference to this thesis, all messages are generally considered to be periodic messages unless stated elsewise. Further information can be found in Section 4.2.2.

Aperiodic Messages

Aperiodic, also known as sporadic, messages are characterized by the fact, that these kind of messages occur regardless of time. Often sporadic messages are also referred to as event-driven messages, since their occurrence is mostly coupled to events. For this type of message, smart sensors are most suitable for detecting such events [Zuberi and Shin, 1995].

Within this thesis, such aperiodic messages are used to simulate additional workload and sporadic bursts. Furthermore, such messages are used to prove the proper functioning of the dynamic priority scheduling algorithm, which is discussed in Section 4.5.

Non-real-time Messages

Messages of the non-real-time message type have no timing constraints and can therefore be accommodated by any communication protocol. Usually, status information or operational data is exchange with this type of message [Zuberi and Shin, 1995].

Non-real-time messages do not find application within this thesis. However, periodic as well as aperiodic messages can turn into non-real-time messages in case of a network bottleneck which causes deadline violations.

2.5.4 Static Scheduling

The next distinction in the classification Figure 2.6, concerning real-time scheduling, is preemptive or non-preemptive.

Most of the literature focuses on preemptive scheduling algorithms, which means that a running task or outgoing message can be interrupted by the request of a higher-priority task or higher-priority message. This behaviour in turn, creates the possibility for priority inversion and also deadlocks [Natale and Meschi, 2001], [Zuberi and Shin, 1995], [Tindell and Hansson, 1994].

In this thesis we use a real-time scheduling algorithm which follows the non-preemptive approach and therefore limits the risk of priority inversion, see Section 4.6.6.

Scheduling algorithms for message interlocking were introduced to increase the utilization of a shared communication channel. Process scheduling was an important issue back in times, where multiprocessing architectures for uniprocessor systems arose. For both applications, message and process

scheduling, two different approaches with regard to real-time scheduling are known [Natale and Meschi, 2001].

- Static Scheduling
- Dynamic Scheduling

Static as well as dynamic scheduling are commonly used, since both approaches come with advantages and disadvantages. A static scheduling algorithm is also called *fixed priority scheduling* algorithm, since priorities[2] are assigned to tasks once and for all [Liu and Layland, 1973].

Deadline Monotonic (DM) is a simple and effective solution, that uses a static priority ordering to solve the contention for the channel. According to the findings of Liu C. [Liu and Layland, 1973], the rate-monotonic (RM) priority assignment is optimum in the sense of processor utilization. RM priority assignment means that priorities are assigned to tasks according to their request rates, so that tasks with higher request rates will have higher priorities.

However, static scheduling algorithms cannot adapt to changes of the parameter settings. For this reason, a more dynamic approach is required whenever the scheduler shall adapt to environment changes. Within this thesis, we mainly focus on dynamic scheduling, since our Communications Model, in Section 4.2.2, requires a rather flexible scheduling approach.

2.5.5 Dynamic Scheduling

The main difference between fixed priority scheduling and dynamic priority scheduling is that static priority scheduling algorithms rely on the premise that the behaviour as well as the importance of the task or message is known in advance. Dynamic priority (*on-line*) scheduling, on the other hand, does not rely on a priori knowledge [Baruah et al., 2010]. This puts on-line scheduling strategies in the advantageous position to dynamically adapt the priority of the process or message at runtime. One of the most common

[2] As for the context of this thesis, the priority of a task cannot be compared to the priority of a message.

representatives of dynamic priority scheduling, is the earliest deadline first (EDF) algorithm.

While EDF scheduling is the basis of many real-time process scheduling algorithms, it is rarely used to schedule real-time messages for networks [Natale and Meschi, 2001]. According to Zuberi K. and Shin K. [Zuberi and Shin, 1995], EDF is impractical for real-time message scheduling, since absolute deadlines become larger and larger as time progresses. A large number of bits for an efficient encoding would be required in the end. If the granularity of a deadline representation is of the order of a microsecond, more then 20 bits would be required to represent deadlines for several seconds. With reference to CAN, this could still be realised with the extended 29-bits ID format. However, this means that 20-30% bandwidth will be wasted because of using the extended ID format [Zuberi and Shin, 1995].

Even relative deadlines have the drawback, that the value requires to be updated at each arbitration round. Moreover, the wide range of relative deadlines of a typical communication workload in a manufacturing environment, ranging from the fraction of millisecond to a few seconds, is difficult to handle [Natale and Meschi, 2001].

Further approaches, concerning static and dynamic real-time scheduling, are discussed in the Related Work section 3.

2.6 INtime

INtime® is a Real Time Operating System (RTOS) developed by the TenAsys Corporation. In the context of this thesis, INtime is used as Operating System (OS) for V&V systems on non-embedded hardware. Additionally, TenAsys provides an embedded Virtual Machine Manager (VMM). The VMM facilitates Inter-Process Communication (IPC), which can be used to extend system functionality alongside Microsoft Windows or another General-Purpose Operating System (GPOS). With the use of the VMM, features such as low-interrupt latency, direct access to I/O and guaranteed ownership of a CPU core are realized. Furthermore, by using the virtualization hardware for partitioning a multi-core system, it is possible to share one platform

between multiple OSs, without a significant impact on determinism [Main, 2010].

TenAsys INtime software runs on single-core, hyper-threaded and multi-core x86 PC platforms from Intel and AMD. Moreover, it can be used in two configurations:

- INtime for Windows, where the INtime RTOS runs alongside Microsoft Windows
- INtime Distributed RTOS, where INtime runs as a stand-alone RTOS [TenAsys, 2009]

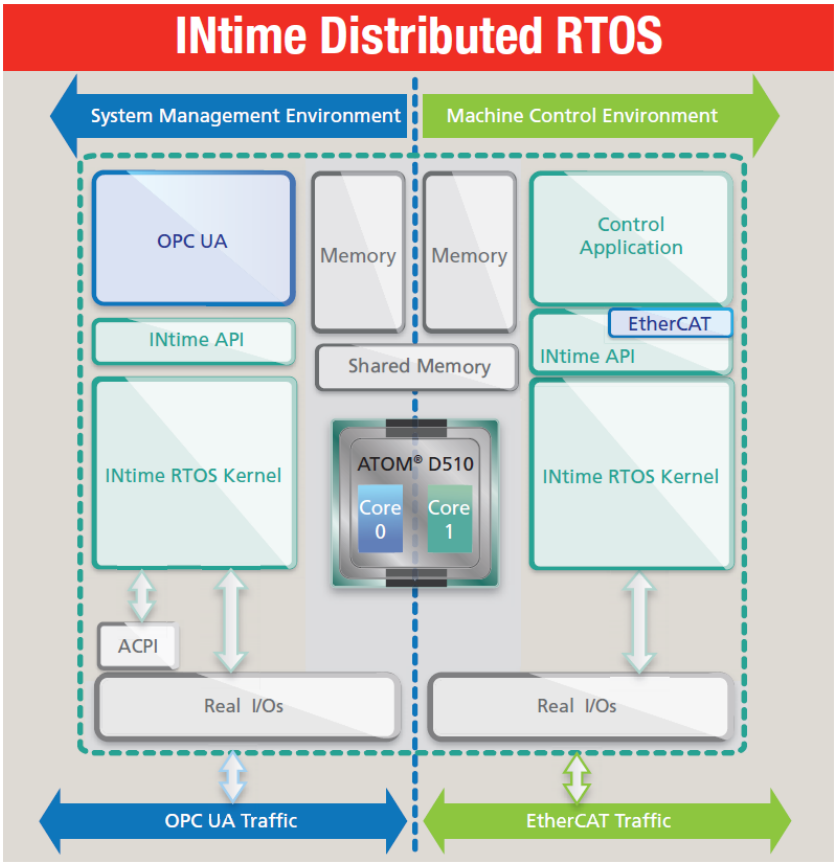


Figure 2.8: Architecture of the INtime distributed RTOS configuration [Grujon, 2011].

Figure 2.8 illustrates the concept of the INtime distributed real-time oper-

ating system. As it can be seen, it enables multiple INtime RTOS kernels to intercommunicate via shared memory and mailboxes in a deterministic way.

2.6.1 INtime Terminology

To establish a common understanding throughout this thesis, the most important INtime-specific terms are declared [TenAsys, 2009].

- **Host:** A host consists of one or more processing elements, such as cores or hardware threads.
- **Node:** A node is an instance of the INtime real-time operating system.
- **Windows node:** Is an instance of the Windows operating system, whether running on a single hardware thread or on multiple hardware threads.
- **Remote node:** A node other than the node where the current process is running.
- **Location:** A handle which uniquely identifies a node.
- **RT kernel:** Provides deterministic scheduling and execution of RT threads within RT processes.
- **Real-time application, C, and C++ libraries:** Provide direct access to the RT kernel services for RT threads.
- **NTX library:** Allows Windows threads to communicate and exchange data with RT threads within the application by providing RT interface extensions for the Win32 API.
- **Global Descriptor Table:** The GDT is a memory segment that contains descriptors for code, data and descriptor table segments.
- **Object:** An object is an instance of a data structure that occupies memory. Each object type has a specific set of attributes or characteristics. After creation, a handle that identifies the object is returned by the RT kernel.
- **High-level object:** A high-level object does not only consume memory, but also gets assigned a slot in the system GDT.
- **Low-level object:** Low-level objects consume memory without a GDT slot assignment. The amount of system memory controls how many low-level objects can be present at a given time.

2.6.2 Processes

The processes in a INtime system form a process tree, whereas each process is an RT kernel object that contains threads and all necessary resources. RT kernel processes have the following characteristics:

- Are passive and thus cannot make system calls,
- Include one or more threads,
- Isolate resources for the nested threads, particularly for dynamically allocated memory. Threads of one process compete for the associated memory of the process.
- Provide error boundaries, in the meaning that errors within one process do not corrupt other processes, since they reside in separate virtual address spaces.
- Objects associated with a process get deleted, once the process is deleted.

In figure 2.9, the process tree of a regular INtime instance can be seen. The root process is on top and builds the foundation, while each application process obtains resources from the root process.

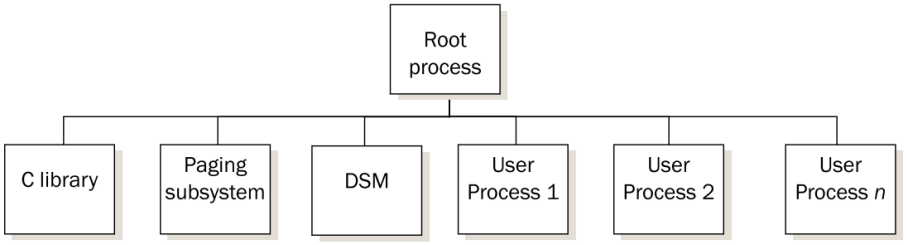


Figure 2.9: INtime process tree [TenAsys, 2009].

2.6.3 Memory Management

Part of the INtime node initialization is the reservation of system memory for the exclusive use by INtime applications and the RT kernel. The RTOS allocates memory which is either removed from the non-paged memory pool available for Windows applications, or allocated from memory that has been excluded from Windows use.

Memory Protection

The INtime RT kernel provides several protection levels for RT memory:

- **32-bit segmentation:** By keeping Windows and each RT process in a separate address space, INtime isolates and protects addresses not only between complex RT processes but also between RT processes and Windows processes.
- **Paging:** Since code, data and stack are automatically placed in non-contiguous areas of the virtual memory of an application, memory overruns are trapped as page faults. Furthermore, demand paging is not implemented although the RT kernel uses the processor's paging mode for virtual address translation. Each RT process loads into its own virtual address space, which is defined by a 32-bit virtual segment.
- **Virtual addressing:** Due to the fact, that for each RT process a separate memory space (defined by a virtual segment) is created by the RT Application Loader, RT processes cannot address beyond the virtual segment. Every RT process is partitioned into its own address space.

Furthermore, the RT kernel enables successful execution of RT threads even in the event of a total Windows failure. Advantageously, even if Windows stops operating, RT threads continue to run, unaffected by the failure, and can execute an orderly shutdown of the hardware [TenAsys, 2009].

Virtual Memory

Every process is in possession of a VSEG (Virtual Segment), which has the same size as the amount of Virtual Memory available to the process. However, the VSEG size must be large enough to contain all the memory dynamically allocated by the threads within the process.

Memory Pool

A memory pool consists of an certain amount of memory with a specified minimum and maximum. Every process has an associated memory pool, which is allocated to the process. The minimum size of memory can be

understood as a contiguous memory region. The memory needed for threads to create objects, comes from the memory pool of the process. If there is not enough contiguous memory available up to the maximum size of the memory pool of the process, the RT kernel tries to borrow memory from the root process. The total memory requirement of the system is always the sum of the memory requirements of each process. Although static memory allocation uses more memory than dynamic allocation, it is considered to be safer.

2.6.4 Thread Scheduling

With a priority-based scheduling policy the RT kernel ensures that the processor always executes the thread with the highest priority. The scheduling policy is enforced on every interrupt or system call so that the kernel holds an accurate execution state and priority for each thread [TenAsys, 2009]. An integer value from 0 (highest priority) to 255 is used as priority. For interrupts that cannot wait, such as serial input, a higher-priority (numerically lower) level shall be assigned. Cached input, on the other side, can be masked with a lower-priority (numerically higher) level.

Table 2.6: Priority range of INtime threads [TenAsys, 2009].

Range	Usage
0-127	Used by the OS for servicing external interrupts. Creating a thread that handles internal events here masks numerically higher interrupt.
128-130	Used for some system threads.
131-252	Used for application threads.

The different priority classes of INtime threads can be seen in Table 2.6. Notably, the priority levels 253-255 are reserved for a Windows-specific services.

Due to the scheduling policy of the RT kernel, the highest priority ready thread is always the running thread. If a thread is not in the *running* execution state, it is either *ready*, *asleep*, *suspended* or *asleep-suspended*. The transitions of a thread's execution state can be seen in Figure 2.10.

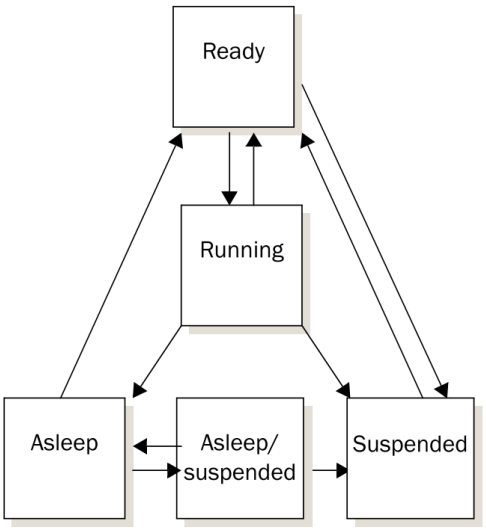


Figure 2.10: State transitions of a INtime thread [TenAsys, 2009].

A thread which is in the running execution state stays in this state until:

- It removes itself from the ready state by making a blocking system call,
- the round-robin time slice of the thread expires (in case of equal priorities),
- it gets preempted[3] by a high-priority thread which has become ready.

Threads are always created in the ready execution state. However, a thread can either put itself to sleep, or suspend or might indirectly put to sleep by the RT kernel due to a blocking call of the thread. An example for such a blocking call is waiting at a mailbox until a message[4] arrives. Once the message arrives, the kernel puts the thread into the ready state.

In case that threads have the same priority level, the round-robin scheduling is applied. This scheduling method takes care that equal-priority threads are run alternating. If a thread is still in the running execution state when

[3]Preemptive in the context of INtime threads, does not conflict with our non-preemptive dynamic priority scheduling approach for messages.

[4]The term *message* in the context of INtime cannot be compared to CAN messages in the context of scheduling.

its time slice expires, the thread is moved to the end of a circular queue for that respective priority level. Then the thread waits until all threads ahead of it use up their time slices so that its own time slice is next in line. This procedure is illustrated in Figure 2.11. However, a thread with a higher priority than the priority-level of the round-robin queue, still preempts any running thread regardless of the amount of time left in its time slice.

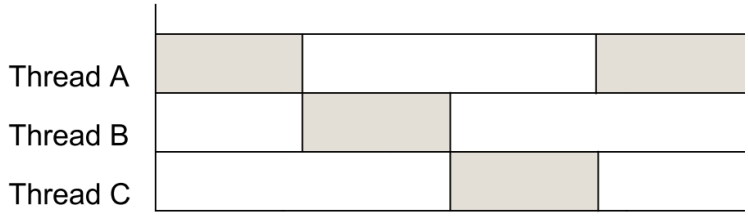


Figure 2.11: INtime round-robin thread execution [TenAsys, 2009].

2.6.5 Mailboxes

A mailbox is an RT kernel object, which enables inter-process communication as well as communication between threads of the same process. Moreover, mailboxes can be used for synchronization, since threads may have to wait for arrival of a message before executing. INtime provides two types of mailboxes:

- **Object Mailboxes:** Object mailboxes are used to pass object handles to another thread, which either runs in the same or in a different process. The most common objects to send or to receive are RT handles for other exchanges, such as semaphore handles or handles to a shared memory section.
- **Data Mailboxes:** With the use of data mailboxes, messages with up to 128 bytes of data can be exchanged between threads within the same process or a different process. Data mailboxes are simple to use and consume less memory [TenAsys, 2009].

Whenever a mailbox is created, the INtime kernel allocates the required resources from the process under which the thread is running. Every mailbox

is in possession of two queues. While one queue is dedicated to hold messages, the other queue holds threads. The *message-queue* holds messages waiting for threads to receive them. The *thread-queue* holds threads waiting for messages to arrive. The INtime kernel ensures that waiting threads receive messages as soon as they arrive, so that at any given time, at least one queue is empty. While the message-queue is always FIFO-based, the thread-queue can either be FIFO- or priority-based. Object mailboxes and data mailboxes handle queues differently. The default queue size of a data mailbox is three messages, whereas each message can be up to 128 bytes large. An object mailbox has a default queue size of eight messages, whereas the size of one message depends on the object type. Whenever a queue cannot accept incoming messages/objects anymore, because it holds already too many items, the INtime kernel automatically handles the overflow. The kernel creates a temporary overflow-queue that holds up to four objects, in case of an object mailbox, and up to 400 bytes, in case of a data mailbox. The overflow-queue is not deleted until each item is processed [TenAsys, 2009].

2.6.6 Semaphores

A semaphore is an RT kernel object, which can be used to synchronize threads and therefore lock shared resources. The concrete implementation of a semaphore is a counter that takes a positive integer value. By sending units to and receiving units from the semaphore, threads can be synchronised. However, semaphores do not enforce but enable synchronization. For a functioning synchronization, threads have to request and obtain units from the semaphore and return the units if no longer needed. Otherwise, synchronization is not achieved properly or threads can be permanently prevented from running. A semaphore with just one unit (single-unit) can be used as a mutual exclusion, also known as *mutex*, for data or shared resources. Semaphores make use of a queue, which holds threads waiting for units. The queue can either be FIFO- or priority-based [TenAsys, 2009].

Priority Bottleneck and Blocking

However, the usage of binary semaphores for mutual exclusion of data or shared resources may result in one of the following bottlenecks:

- The first bottleneck occurs when a low-priority *running* thread blocks a high-priority *ready* thread. Regardless of priority, the running thread controls the resource until it releases the units back to the semaphore.
- Another disadvantage that arises due to the usage of binary semaphores, is *priority inversion*. Priority inversion occurs when a low-priority thread obtains the required units to access a shared resource. Then this thread is preempted by a medium-priority thread, which is in turn preempted by a high-priority thread that needs to access the resource. The high-priority thread cannot access the data while the low-priority thread holds the units. Whereas the low-priority thread cannot complete its operation and return the units, since it is preempted by the medium-priority thread.
- The third bottleneck defines a blocking situation with regard to the shared resource. If the thread, which holds the semaphore unit, is suspended or deleted, no other thread can gain access to the shared resource. In case of suspension, only after the suspended thread is resumed and releases the semaphore unit, other threads can obtain a unit of the binary semaphore. If the thread is deleted, the semaphore prevents any other threads from ever using the shared resource [TenAsys, 2009].

3 Related Work

The underlying work of this thesis, is the heavily discussed topic of real-time scheduling with focus on communication multiplexing. Although a vast amount of related literature is available, only the most influential work can be considered within this thesis.

Commonly, real-time systems are characterized by their need for functional correctness and time-related operating principle. With reference to the literature, real-time systems can be classified on the basis of various aspects, as already outlined in Section 2.5 [Mohammadi and Akl, 2005], [Kopetz, 1997], [Audsley and Burns, 1990].

Most of the literature focuses either on real-time process[1] scheduling (an extensive survey by Sha et al. [Sha et al., 2004] and also [Liu and Layland, 1973], [Saez et al., 1999], [Sprunt, 1990], [Audsley et al., 1993], [Lehoczky, 1990]) or on real-time message scheduling [Zuberi and Shin, 1995], [Natale and Meschi, 2001], [Meschi et al., 1996b], [Tindell et al., 1995]. The scheduling approach of our demonstrator, however, cannot be categorized either way. Since our system schedules data streams, neither process nor message scheduling completely applies.

In 1973, Liu C. and Layland J. [Liu and Layland, 1973] laid the foundation for nowadays sophisticated scheduling algorithms. With their analysis on periodic task scheduling on uniprocessor systems, the optimal fixed priority and dynamic priority scheduling algorithms were determined. Moreover, they showed that the rate monotonic priority ordering, where tasks are assigned priorities in the order of their periods, is the optimal priority assignment policy for such task sets [Davis, 2014]. While fixed/static (off-line) priority scheduling algorithms rely on the premise that the behaviour

[1] In this context the terms *process* and *task* can be used interchangeably.

of the task is known beforehand, dynamic priority (on-line) scheduling algorithms have no a priori knowledge [Baruah et al., 2010]. However, this puts on-line scheduling strategies in the advantageous position to dynamically adapt to environment changes, as scheduling decisions are made at each instant.

In this thesis, we focus solely on on-line scheduling performed by a single processing unit. This central scheduling approach corresponds to our System Model demands (S3).

A well-known dynamic scheduling algorithm is the Earliest Deadline First (EDF) algorithm. In the EDF approach, tasks with the nearest deadline[2] will be assigned the highest priority, and tasks with the furthest deadline will be assigned the lowest priority [Liu and Layland, 1973], [Meschi et al., 1996b], [Meschi et al., 1996a]. Which means that the priority of a task is assigned inversely proportional to its deadline.

With the help of one of the major findings of [Liu and Layland, 1973], that dynamic scheduling approaches are proved to be more efficient than their static counterparts, Meschi et al. [Meschi et al., 1996a] investigated the performance of EDF for message scheduling with limited priority. Meschi et al. demonstrate that the schedulability test of Baker T. [Baker, 1990] for a uniprocessor environment can be applied to the network scheduling problem[3], when the EDF algorithm is used to schedule messages on a single shared channel.

Yet, unlike our approach, this work also considers the preemption of messages and thus respects priority inversion, which is both not intended by our models (S2) and (R3).

Audsley et al. [Audsley et al., 1993] and Burns et al. [Burns et al., 1994] show how the analysis of Joseph M. and Pandya P. [Joseph and Pandya, 1986] can be updated to include blocking factors introduced by periods of non-pre-emption, release jitter, and accurately take account of a task being non-pre-emptive for an interval before termination [Tindell et al., 1994].

[2]Deadline is the time-span or instant, depending whether relative or absolute deadline, by which execution must be done.

[3]The problem that a network access protocol is required that allows all deadlines of all periodic communications to be met.

Further research executed by Lehoczky J. and Sha L. [Lehoczky and Sha, 1986] outlines three important issues that distinguish the network scheduling problem from the processor scheduling problem: *task preemption*, *priority level granularity*, and *buffering*. The first issue, task preemption, conflicts with our System Model (S2) and can therefore be neglected. Since the scheduler is implemented within our system, and thus the priority levels as well as the buffers can be defined appropriately, the two other issues do not find application either.

A more similar approach was conducted by Zuberi K. and Shin K. [Zuberi and Shin, 1995]. In their work they present an approach that is capable of carrying periodic, sporadic, and non-real-time messages through a CAN bus. This aim corresponds well with our Communications and System Model, (C1), (C2), (S2)[4], and (S3). One of their early findings is that the non-preemptive EDF scheduling is impractical for CAN, for the simple reason that the deadline encoding would consume too many bits of the 11-bit standard CAN ID field. With the use of the 29-bit extended CAN ID format, the deadline encoding would be possible but would also results in 20-30% more bandwidth consumption compared to the standard format.

Irrespectively, this disclosure does not preclude the EDF for our demonstrator concept, since the message header must not be modified in any case.

The dilemma that the deadline monotonic[5] (DM) is easy to implement but gives low utilization, and that the EDF admittedly improves utilization but wastes an enormous amount of bandwidth because of the increased message length, Zuberi K. and Shin K. developed a Mixed Traffic Scheduler (MTS). The MTS uses EDF for high-speed messages and DM for low-speed messages, and hence improves CAN bus utilization compared to the sole DM use. Notably, the EDF part of the MTS approach requires the modification of CAN message IDs, which is obviously not reconcilable with Requirement (R1) and makes this approach less relevant.

However, Baruah S. and Haritsa J. [Baruah and Haritsa, 1997] point out by referring to Jensen et al. [Jensen et al., 1985], that the EDF algorithm is solely

[4]Message transmission is not preempted in order to avoid retransmits.
[5]DM is a static priority scheduling approach.

optimal as long as normal (non-overloaded) conditions are present. Whenever the system is overloaded, in the sense that deadlines cannot be met, the performance of EDF decreases drastically. In fact, even *random* scheduling has a better performance at this point, which makes the continuous use of EDF in an emergency an fatal flaw.

With regard to Requirement (R2), to manage also scenarios in which the system is in an overload condition, a more adaptive approach is required.

Best-Effort algorithms make use of a rejection policy for overloaded systems based on removing tasks. As long as the system is underloaded, this algorithm behaves as EDF, and once the system is overloaded it chooses the subset of tasks that maximize the value [Locke, 1986]. Similar to the Best-Effort algorithm is the Robust Earliest Deadline (RED), proposed by Buttazzo G. and Stankovic J. [Buttazzo and Stankovic, 1993].

The RED algorithm is highly sophisticated and rich in features. For instance, RED is capable of detecting overloads, and consequently graceful degradation by rejecting the least value task. Whereas there is also a recovery routine implemented that tries to reaccept prior rejected tasks. The concept of the RED algorithm is actually quite similar to our approach, although instead of tasks, data streams are being rejected or degraded due to their criticality.

4 Connectivity Manager (CM)

Historically, the communication among embedded systems and automotive test systems was designed for single core architectures, with the consequence that each task on the test system needs to run on a separate core. This resulted in a 1:1 relationship between the ECU on the embedded system and the test-task running on the automotive test system (V&V system). To make use of multiple cores, which became available recently on both the automotive ECU controller side as well as in industrial PC's for the test system side, we designed a **Connectivity Manager (CM)** for the V&V system that is in charge of multiplexing several data streams across a shared network.

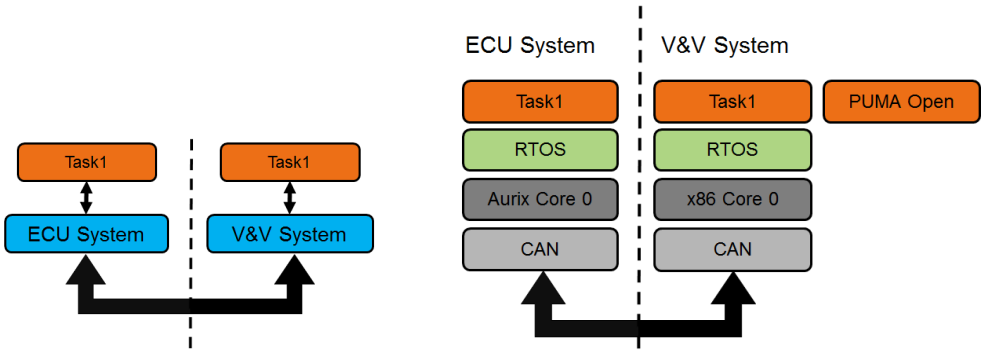
In the following sections we give an evaluation of the *status quo system* and thereof formulate the main goals for our *target system*. Furthermore, we define a *notation* and outline *communications model*, *system model* as well as *requirements* of our system. Moreover, we discuss our concept in detail and explain the architecture as well as the design of our system. Equally important for this research is the *dynamic priority communication scheduling* mechanism that is implemented as part of the Connectivity Manager. At the end of this chapter we reflect on the implementation of the demonstrator and point out limitations of the implementation.

Notably, our concept and architectural design is held as generic as possible to consider both, the ECU system and the V&V system. However, within this thesis we mainly focused on the V&V system and decided to use CAN as shared communication network. For this reason, some of the following sections are influenced by these perimeters. Furthermore, in the upcoming chapters and sections the terms *task* and *application*, as well as *connection* and *data stream* can be used interchangeable.

Since this thesis was conducted in cooperation with the AVL List GmbH, the status quo system represents an AVL-specific automotive system that relies on the PUMA/PUMA Open framework.

4.1 Status Quo System

These AVL-specific automotive systems were designed for a single core architecture, which results in a 1:1 relationship between ECU system and V&V system. Due to this inflexible relationship, the utilization of shared resources, such as memory and network interfaces, is limited. To enhance and improve the automotive test system, we started with an analysis of the existing test system.



(a) Relation of the two systems.

(b) Communication stack between the two systems.

Figure 4.1: According to the status quo system design a 1:1 relationship between *ECU system* and *V&V system* is prevalent.

In Figure 4.1 this 1:1 relationship is illustrated. While Figure 4.1a outlines the simple relation between *Task1* on the *ECU system* and *Task1* on the*V&V system*, 4.1b describes the whole communication stack between the aforementioned tasks. Due to the inflexibility of the 1:1 relationship, the above described architecture entails the following disadvantages:

- **Limited Scalability**: Since each operating task needs to be in possession of a communications adapter, the number of simultaneously

running tasks is limited by the number of available CAN interfaces. Thus, no *n:m relationship* between tasks among the two systems can be realized and highly limits the scalability of the design. A further limitation emerges due to the fact, that one core can only host one application. Meaning that every additional task on either side requires an additional core at that respective system, which consequently results in higher costs.

- **Inefficient Resource Sharing**: Furthermore, neither system is capable of sharing operating system resources, such as memory with its accompanying memory management, CPU kernel time, communications adapter, as well as a global wall-clock time. Especially the latter is of high importance whenever tasks need to operate synchronously.
- **Limited Communication Capability**: Due to the "one task, one core" premise, communication between cores of the same system, so called inter-core communication (ICC), is not possible. This implies that tasks within the same system but on different cores are not able to communicate with each other.
- **No Parallelism**: With the 1:1 system design, equal procedures are unnecessarily performed manifold. For example, the initialization of the communications adapter is performed by each core, although the network could efficiently be shared among all cores within the same system. The same redundancy applies for memory management and time synchronization.

Furthermore, in Figure 4.1b it can be seen which layers are involved in the communication flow between the task on the automotive embedded system (*ECU system*) and the task on the automotive test system (*V&V system*). Starting on top of the V&V system, *Task1*, which might in practice be a PUMA Open[1] application, runs within a real-time operating system (RTOS). In the past, AVL decided to use TenAsys INtime® as a real-time operating system, due to its robustness and stability [TenAsys, 2007]. In further consequence, the RTOS runs on a computer that is in possession of the physical communication hardware. At the V&V system side, the computer is a regular workstation that runs INtime besides Windows®. The software driver of the physical communication hardware is loaded into

[1]PUMA Open is a test bed software by AVL for testing engines, transmissions, and powertrains. Furthermore, it is the platform for all automation tasks.

the real-time operating system, with the result that real-time applications can access the underlying communication network. At AVL, most test bed systems rely on a controller area network (CAN) as communication network. Therefore, also our implementation (see Section 4.6) is targeting CAN. Following alongside the CAN bus to the ECU system, the next step in the communication flow is the communications adapter of an ECU. As already stated earlier in Section 2.1.1, an ECU is a control unit that usually contains an embedded microcontroller (MCU). To be more precisely, among others, AVL uses AURIX^{TM} MCUs from Infineon. Powered by the microcontroller, the ECU runs its own real-time operating system, for instance FreeRTOS^{TM}. As a last step of the communication flow, *Task1* of the ECU system is reached and thus the data arrived at the receiver.

Concluding, each test bed is controlled by automotive test systems (V&V systems), which are, amongst other things, responsible for providing the embedded in-vehicle system (ECU system) with simulation data. Additionally, the embedded in-vehicle system returns diagnosis as well as analysis data with the result of a prevalent bidirectional communication.

4.2 Notation and Model

For a better understanding of the Connectivity Manager approach, a model definition as well as a notation declaration is helpful. Furthermore, we define requirements that our *communications model* as well as the *system model* needs to fulfil.

Notably, the communication within our system works as follows. Tasks want to transmit data in pre-defined time intervals, wherefore data streams are established. The data, which is dedicated to be transmitted within one period, is packed to a *data package*. However, a data package is too large to fit into one network message. Hence, the data package is segmented into smaller parts and then divided among multiple network messages. Meaning that one data package usually consists of several network messages. All network messages within the same data package have an equal deadline, which is the end of the respective period.

4.2.1 Notation

The most important and relevant notations that are defined below are additionally illustrated by Figures 4.2 and 4.3.

- B_{NET} overall bandwidth of the communication network;
- B_{CM} overall bandwidth dedicated to the Connectivity Manager of a V&V system;
- S_n the n-th data stream;
- B_n worst-case bandwidth estimation of S_n;
- C_n criticality of S_n;
- V_n data volume of S_n;
- λ_n cycle-time of S_n;
- Q_n send-queue of S_n;
- $P_{l,n}$ the l-th data package of S_n;
- $T_{\text{CREATE,l,n}}$ creation-time of P_l of S_n;
- $M_{i,l,n}$ the i-th message of P_l of S_n;
- $D_{l,n}$ the l-th absolute deadline of $M_{i,l,n}$;
- $T_{\text{START,n}}$ start-time of S_n;

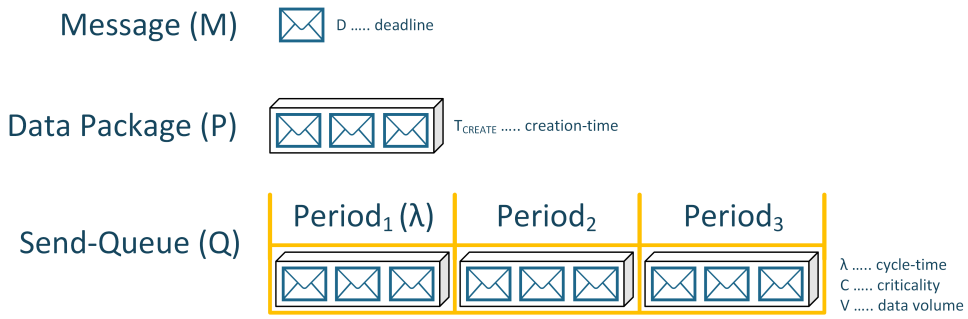


Figure 4.2: Visualization of relevant items.

As it can be seen in Figure 4.2 each message M holds a value that represents the deadline of the message. In our approach, the *deadline* of a message is an absolute value and represents the instant by which the processing of the message must be done. Further, each message is part of a data package, whereas the number of messages within a data package is defined by the

data volume V_n of the data stream S_n. For simplicity reasons, in this figure, the data volume is illustrated in the form of message icons, unlike as in the implementation where the unit of the data volume is stated in byte. Considering Figure 4.2, the data volume is simplified to three messages . Moreover, each data package brings along a creation-time $T_{\text{CREATE,l}}$ which is an absolute value that represents the instant at which this data package was created. This brings us to the first conclusion, that the data of each data stream is fragmented into data packages that contain the actual data in the form of network messages. The cycle-time λ_n of the data stream defines at which intervals data packages are being sent. The number of intervals is measured in periods. Each period is characterized by a start and an end, whereas the end of a period is simultaneously the start of the following period. While the cycle-time is stated in milliseconds (ms), the criticality is indicated by a integer value ranging from 1 (low) to 99 (high).

Table 4.1: Units of the input parameters.

Input Parameter	**Symbol**	**Type/Unit**
cycle-time	λ	ms
data volume	V	byte (messages)/period
criticality	C	uint [1..99]
deadline	D	absolute time value
creation-time	T_{CREATE}	absolute time value
start-time	T_{START}	absolute time value
bandwidth	B	bit/s

Considering Figure 4.3, a more concrete example can be seen. Here we have a similar scenario with just one send-queue, but already with given values. For readability reasons, the cycle-time is set to 1 second and the data volume to 3 messages. Furthermore, the criticality is defined with the integer value 7 and all data packages are labelled accordingly. The messages are also denoted with the corresponding index. Special attention should be paid to the creation-times ($T_{\text{CREATE,l}}$) of the data packages (P_l). As it can be seen, the creation-time is neither synchronized to the start of the period nor is there a constant interval to the following creation-time. This behaviour should illustrate, that the data package creation may drift within its period. This is attributable to the operating task which might use a clock with a

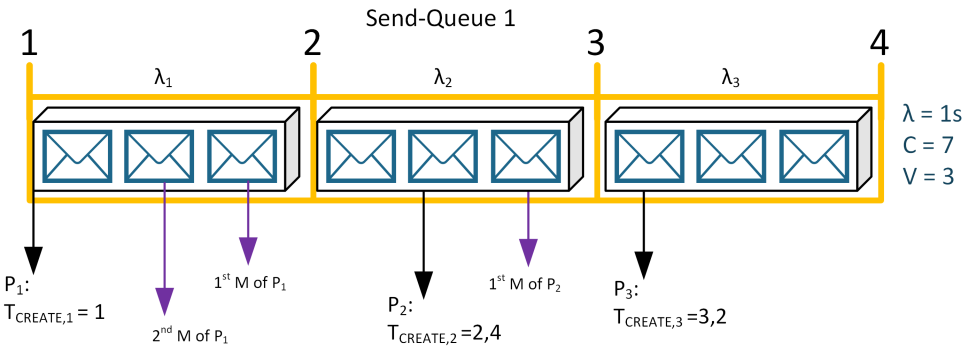


Figure 4.3: Concrete example of a send-queue.

poor resolution. Further details concerning this topic are outlined in 4.6.6.

4.2.2 Communications Model

Our communications model consists of independent connections S_n which represent data streams[2] initiated by multiple applications. As already outlined in Figure 4.2, every S_n is characterized by three attributes: V_n data volume, λ_n cycle-time, and C_n criticality. Those three attributes are parametrized upfront and remain constant as long as the data stream is active.

As mentioned above, the criticality is an input parameter and can therefore be seen as a static attribute of a data stream. Furthermore, the following chapters and sections make use of the term *priority*. Whereas priority in the context of our system always indicates the importance of a message that was issued by a data stream. Hence, a high-priority message is more important to the system and therefore processed earlier. Consequently, we use the term priority to refer to the importance of a message, whereas the importance is evaluated by our system.

The difference between criticality and priority is, that the criticality is a static attribute of a data stream and the priority is an evaluated importance of a message, that is solely used within our system. However, the criticality

[2]Within this thesis, the terms data stream and connection can be used interchangeably.

can influence the priority of a message, since the priority of a message is either based on the deadline or on the criticality. This is further outlined by (S3) and also Section 4.5. Notably, the term priority should not mistakenly be considered to be the priority of an INtime process nor of a CAN message outside our system, unless stated elsewise. Priority is a factor of importance which we use to order messages within our system.

The following enumeration describes aspects of the communication model our system has to deal with.

(C1) **Periodic**: Applications attempt to read or write data at intervals depending on their prior parametrization. The time between the read and write intervals is defined by the cycle-time of the data stream. For this reason, concrete periods, starting with the very first attempt, can be determined. However, due to clock inaccuracies the read or write attempt of the task might drift, with the result that the cycle-time is not strictly adhered.

(C2) **Sporadic**: However, in addition to (C1), tasks might also act non-deterministic. Meaning that a task might skip a period or attempts to send a different amount of messages than defined by the data volume. Consequently, to make the system more robust, our approach does not take it for granted that applications fully adhere to the cycle-time and therefore also sporadic read and write attempts are being considered.

(C3) **Hard Real-Time**: Most automotive applications can be seen as real-time applications with a hard deadline. The deadline of such applications must not be violated since its timeliness might be necessary for a proper functioning of the system.

(C4) **Soft Real-Time**: Additionally to (C3), real-time applications might not be able to meet their hard deadlines due to external changes, such as network faults. Since such unpredictable faults cannot be precluded, hard real-time applications are being degraded to soft real-time applications. This degradation allows applications to violate their deadlines without compromising the system.

(C5) **Mixed-Criticality Data**: Each application is defined by a criticality attribute C_n which represents the importance of the application. Applications with a high criticality value will be prioritized higher while low-critical applications are proportionally lower prioritized. There-

fore, within our system both, higher and lower criticality data streams, are involved wherefore our system must be capable of dealing with mixed-criticality data. The criticality is taken into account when deciding which data stream is being degraded first. Further details concerning mixed-criticality data can be found in Section 4.3.2.

4.2.3 System Model

Since a scheduler component is an important part of the Connectivity Manager system, the following scheduling as well as design characteristics are being considered for our system model:

(S1) **Multi-Core**: Mostly for economic reasons[3] automotive (test) systems are being redesigned to be multi-core capable. In other words, running applications can be spread across multiple cores.

(S2) **Non-Preemptive Scheduling**: In the literature, preemptive scheduling is described as an approach in which the scheduler is allowed to interrupt the processing of the scheduled item. In case of network scheduling, the data transmission of a node might be interrupted by a concurring node. In terms of process scheduling the execution of a process might be suspended due to a higher critical process demanding the CPU. Since our scheduler is neither directly attached to the network, and therefore not affected by network contention, nor in charge of scheduling concurrent processes, our system is considered a non-preemptive system. According to Section 2.5 and Figure 2.6, our scheduling approach can be characterized as non-preemptive scheduling.

(S3) **Dynamic Priority Communication Scheduling**: The priority of a message is either based on $D_{l,n}$ or C_n, whereas the relevance of this priority basis might change during run-time. Therefore, a dynamic priority communication scheduling approach is necessary. Further information about dynamic priority scheduling can be found in Section 4.5.

[3] `https://www.artemis-emc2.eu/project_overview/`

4.2.4 Requirements

The Connectivity Manager system was designed in consideration of the following requirements:

(R1) **Data Integrity**: Due to compatibility reasons, the data, which goes through our system, has to remain consistent and must not be altered. Meaning that message headers, for instance the CAN ID, must not be modified.

(R2) **Overload Adaptive**: The system must be capable of managing situations in which it is overloaded, caused by external changes such as network bottlenecks, simultaneous arrival of events, or faults of peripheral devices. This requirement is in close contact with (C4), which characterizes the communication model to be allowed to violate deadlines in case of an overloaded system state. In such an event, the criticality is of highest relevance for the prioritization. Further details regarding overload system states can be found in Section 4.5.1.

(R3) **Limited Priority Inversion:** Since our system follows a dynamic priority scheduling approach, there is a legit chance that priority inversion might occur. Therefore, the target system shall limit any occurrence of priority inversion. With regard to (S2), priority inversion is largely limited as messages are transmitted one after another. So the picking and forwarding of a message is an atomic process, due to mutual exclusion. The only occurrence of priority inversion might happen in the stack of the network driver, which is beyond the area of responsibility of the Connectivity Manager. In Section 4.6.6 more details about priority inversion can be found.

4.3 Concept

In consideration of the above described models as well as requirements, we came up with a concept that extends the status quo system towards an improved *Target System*. In the next section, our idea of the target system as well as our goals are described.

4.3.1 Target System

Our aim is to upgrade the old system to a multi-core architecture while keeping it as compatible as possible, in terms of application programming interfaces (API). Figure 4.4 illustrates the rough idea of our concept and pictures the target system we strive for. This, however, should not be seen as a final architecture, that one will be discussed later in Section 4.4.1.

The three major goals of our concept as well as of this whole thesis are as follows:

1. **Shared communication link:** The underlying communication network should be shareable among all tasks. Meaning that only one physical link as well as one communication adapter per system is required to enable data exchange. A shared communication link aims for the large advantage of lowering costs due to wiring and interface reduction. However, this goal is accompanied with the drawback that not only the communication link is shared among all tasks, but also the bandwidth of the communication link. Hence, a central scheduler is necessary to take over the multiplexing of the data.
2. **Shared resources among multiple cores:** The second major goal of our concept is to upgrade the system from a single core architecture to a multi-core architecture. The impact of this goal is twofold. On the one hand side multiple cores should be running within the same system and on the other hand side multiple tasks should be hosted by one core. Furthermore, with this core alignment, several resources such as memory management, CPU kernel time, or a global wall-clock time can be shared.
3. **Handle mixed-criticality data:** Due to major goal 1 and 2, the consideration of mixed-criticality data becomes a necessity. Since the communication link is shared among multiple cores and tasks, a flexible scheduling approach that bundles and multiplexes all data streams is required. Further on, the data streams may vary in terms of criticality, which is why our scheduler must be capable of handling mixed-criticality data.

4.3.2 Data Distinction

Additionally to major goal 3, the target system will further be in charge of dealing with data that varies in periodicity. Meaning that within our system data streams exist, that differ not only in terms of criticality but also vary in terms of cycle-time. An example of this data distinction can be found in Table 4.2 and Table 4.3.

Table 4.2: Data streams varying in periodicity (cycle-time) and criticality.

purpose	**f [Hz]**	λ**[ms]**	**criticality**
real-time control	200	5	80
diagnostic	100	10	30
housekeeping	10	100	50

Table 4.3: Data streams varying in criticality.

purpose	**f [Hz]**	λ**[ms]**	**criticality**
wheel speed front	200	5	80
wheel speed back	200	5	70

Considering Table 4.2, an example for different types of data can be seen. Some data streams might operate on a real-time level with short cycle-times while other data streams might be in charge of diagnostics with a less frequent periodicity. Furthermore, a distinction in terms of criticality is observable. For example, diagnostic data streams might require a smaller cycle-time due to data accurateness, however housekeeping data streams might be more critical to the functioning of the system.

With reference to Table 4.3, we can see an example in which the data streams are both operating on the same frequency but with different criticalities. Wheel speed front and wheel speed back are quite crucial for ABS or power steering, wherefore both data streams operate on a real-time level. However, the data of wheel speed front might be of higher criticality because of its importance for transmission and gearbox, assuming a front-wheel drive configuration. Comparing our new concept to the old system, Figure 4.1, it obviously increased in the number of utilized cores. With the number of cores (layer **2**) also the number of independent real-time operating systems

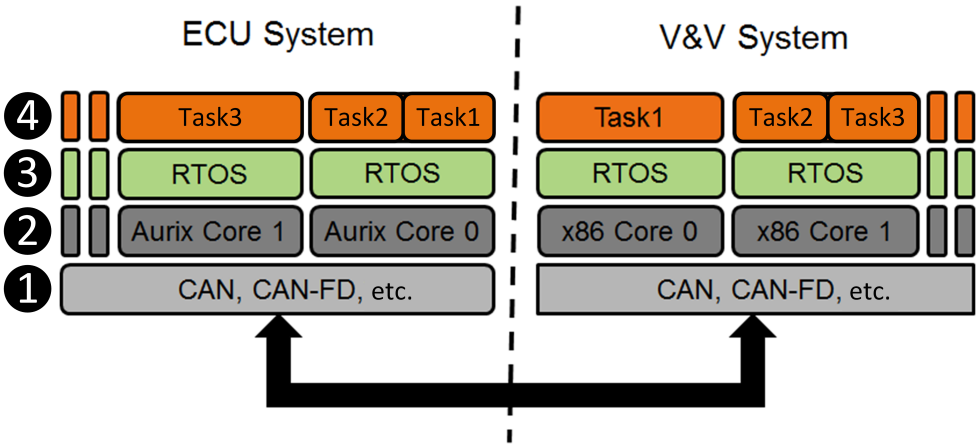


Figure 4.4: Basic concept of the target system.

(layer **3**) increased. This new design enables multiple cores to share the same communication network which is labelled with layer **1** in Figure 4.4. As for the concept, the communication network is assumed to be as generic as possible to support various transport technologies such as CAN, CAN-FD or any future technology.

4.3.3 Characteristics of Target System

In consideration of the System Model in Section 4.2.3 and the goals of the target system, we decided to introduce a new layer that includes a central component called Connectivity Manager. The Connectivity Manger represents the main processing component that can be reached from all cores and tasks. Still, the Connectivity Manger exists only once per system and is responsible for the following functionality:

- Sessions management (establish, open, reject, close)
- Perform read/write operations to the communication network
- Manage shared memory area
- Network overload behaviour management
- Message prioritization

In the following section, the design as well as the architecture of our target system, especially of the Connectivity Manager, is outlined.

4.4 Architecture and Design

Although our design is valid for the whole automotive communication system, our main focus lies on the automotive test system, which is also called validation and verification (V&V) system.

4.4.1 Architecture

As it can be seen in Figure 4.5, the V&V system is based on a x86 processor architecture, while the automotive (ECU) system relies on embedded microcontrollers, in our approach for example Infineon AURIX™ multi-core microcontrollers. The dotted circle indicates the area of responsibility within this thesis as well as for the demonstrator.

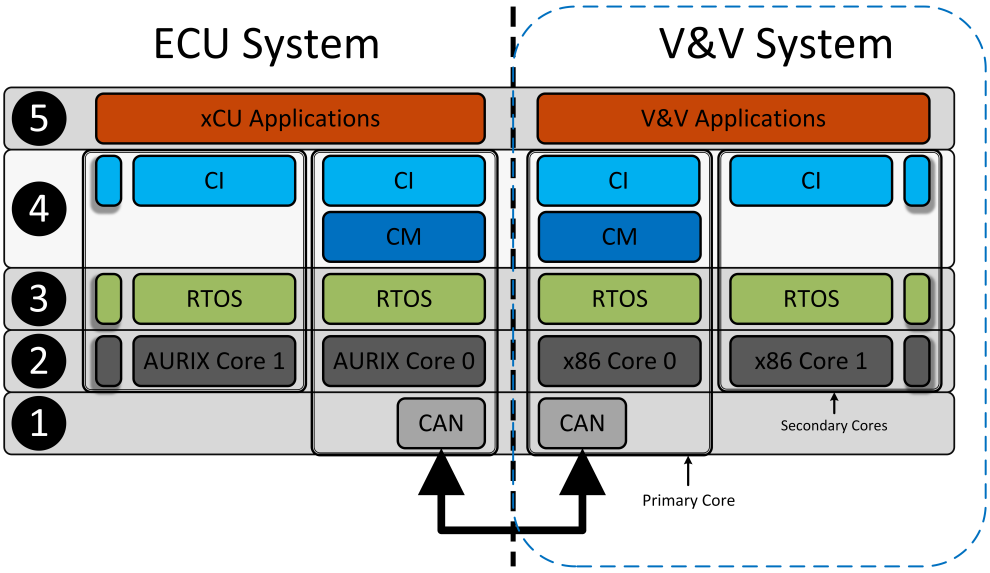


Figure 4.5: Architecture of the demonstrator.

(1): This layer employs the network protocol that is used to enable a bidirectional communication between the ECU system and the V&V system. Since CAN is still the most common communication standard in the automotive industry, also our design and demonstrator is dedicated to CAN. However, we designed our system as generic as possible so that other communication standards, such as CAN-FD or automotive Ethernet, are not precluded.

(2): The *second* layer represents the processor architectures and also illustrates the multi-core design. Both systems are considered to provide multiple cores, whereas only one core is in possession of the physical communication network interface. In our approach, this core takes the role of the *primary core* whereas all other cores are *secondary cores*.

(3): Based on the processor architecture, an instance of the respective real-time operating system is deployed on each core. In case of our V&V system, INtime®[4] is being used. The ECU system runs FreeRTOS instances.

(4): One of our major changes to pursue the Connectivity Manager approach, was the interposition of the *fourth* layer. By inserting a new layer we were able to detach the application (*fifth*) layer from the network driver that is running on the real-time operating system (*third*) layer. The decoupling of the two layers increases the flexibility as well as the scalability of the whole system, in the sense that applications no longer communicate directly with the underlying network driver. Instead, the Connectivity Manager takes over this communication and acts as a multiplexer. This, however, raises the problem of a more complex information exchange among the cores. Therefore, an additional component, namely the Connectivity Interface (CI), is necessary to enable inter-core communication in our multi-core environment (further details in Section 4.4.3). From the application perspective, the communication network interface seems to be directly attached. In fact, this is achieved through the Connectivity Manager by a kind of virtualization of the underlying communication network.

(5): The *fifth* layer represents the application layer, which is now detached from the network driver. Due to the network virtualization by the

[4] As already outlined in Section 2.6, INtime® is a real-time operating system by TenAsys Corporation.

Connectivity Manager, one or more automotive applications can be executed on each core while sharing the same physical network. In other words, each core can host multiple applications, while each application can again conduct multiple V&V tasks. Moreover, the network virtualization enables the exchange of messages between cores within the same V&V system.

In the redesign of the V&V system, we introduced a new layer, called Connectivity Manager/Connectivity Interface, to decouple the application layer from the network driver. The effect of this decoupling is twofold. On the one hand, the system becomes multi-core capable and provides a virtualization of the underlying communication network. On the other hand, an additional Connectivity Interface is required to manage communications across multiple cores.

4.4.2 Design

The Connectivity Manager becomes the main, central processing component and exists only once per system, necessarily running on the primary core. As it can further be seen in Figure 4.6, the Connectivity Interface (CI), on the contrary, exists on each core and is responsible for redirecting data from applications to the sole Connectivity Manager instance and vice versa. Since the Connectivity Manager has exclusive access to the physical communication network interface, it acts as a multiplexer by bundling all incoming data streams. A built-in scheduler takes care of prioritizing the data with respect to deadline or criticality (further discussion in Section 4.5).

The main design principle is, that exactly one Connectivity Manager is responsible for multiple Connectivity Interfaces. Whereas one Connectivity Interface is responsible for multiple *tasks* on the same core. In turn, one task is allowed to request multiple connections to the communication network. The very first initialization of each interface is conducted via a configuration request initiated by the task. The actual data exchange is then performed via shared memory, which is discussed in more detail in the following Section 4.4.3.

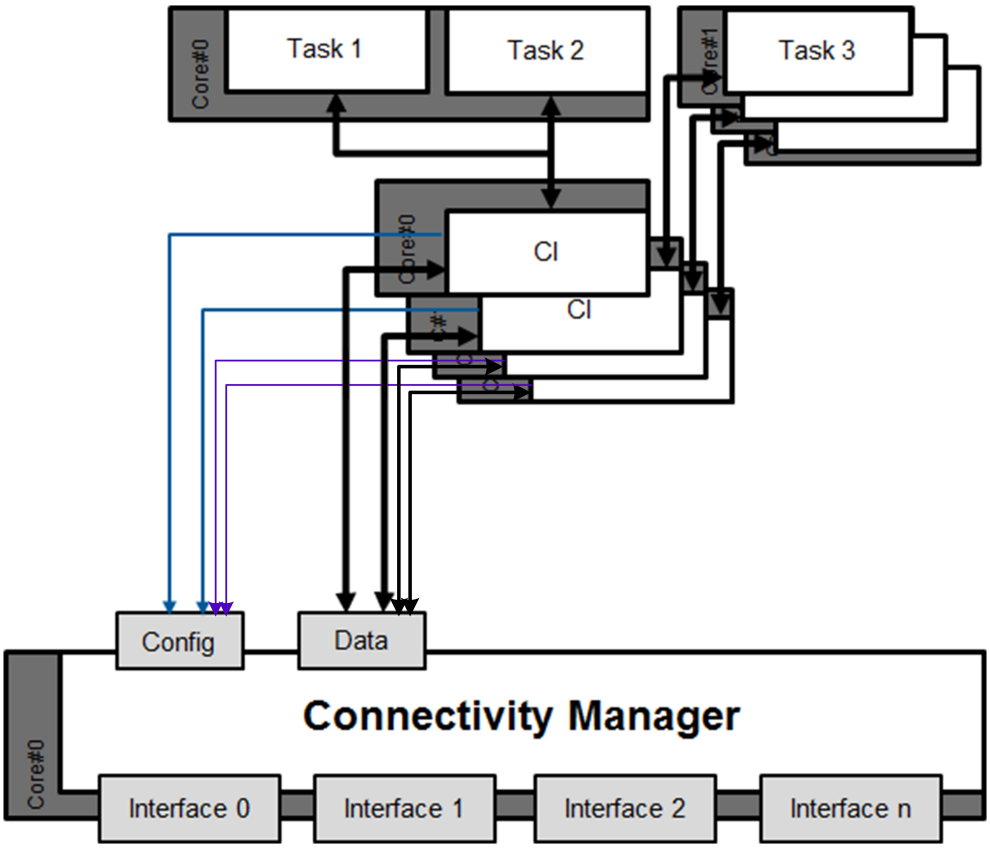


Figure 4.6: Design of Connectivity Manager. Multiple configuration and data channels.

From the design perspective, the Connectivity Manager is conceived to manage multiple interfaces, so that each connection is coupled with the optimal interface, such as CAN, CAN-FD or Ethernet respectively with varying bit rates. However, for our practical demonstrator we focused on only one interface, in particular a CAN interface.

Concluding, the following design considerations influenced the design of our system:

- Generic design
- Support various communication technologies
- Compatible with the old system
- Extensible and scalable

- Adaptable for ECU system

4.4.3 Inter-Core Communication

One of the most challenging tasks when creating our design, was the realization of the inter-core communication (ICC) on data stream level. Since most applications operate in real-time, a fast and reliable communication among cores is required. Further, Synchronization has to be considered, as threads are simultaneously accessing data. Nevertheless, to retain a lean and quick-response system, the synchronization (locking) overhead must be held at a minimum. In Figure 4.7, the inter-core communication scheme, including the most relevant objects, is illustrated.

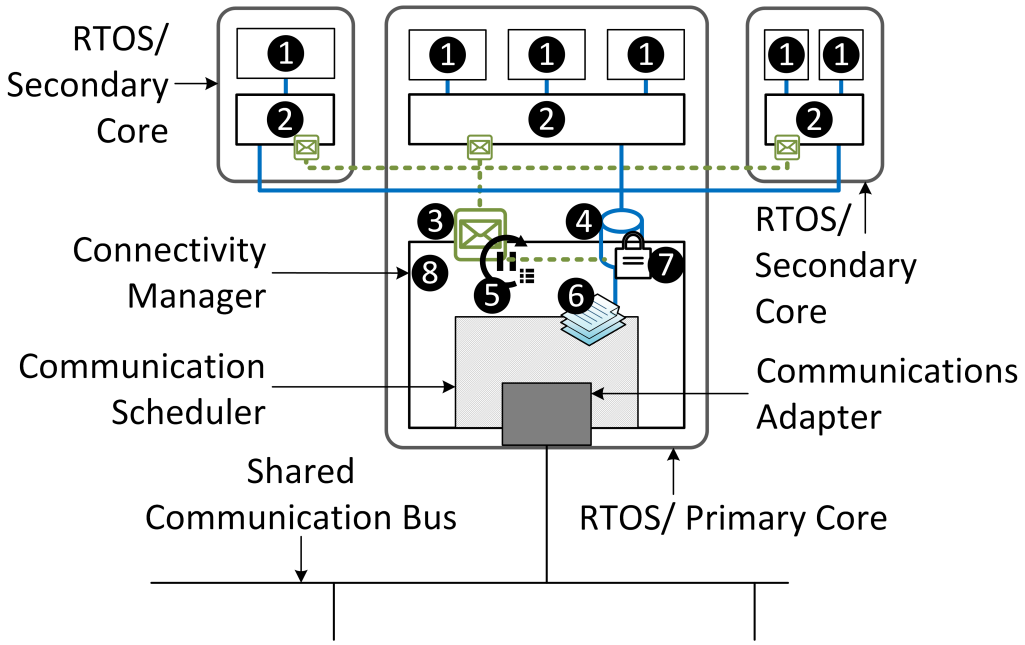


Figure 4.7: Inter-core communication scheme.

(1) Application: As Figure 4.7 emphasises, on one core several V&V applications can be executed simultaneously. Two or more applications can coexist in the V&V system while being hosted on different cores. Whenever an application wishes to conduct a test task, a new data

stream must be established first. This is initiated by approaching the Connectivity Interface and disclosing information (C_n, V_n, λ) about the desired connection. A so called Connection Descriptor object holds values for cycle-time, data volume and criticality of the task. In case of a successful registration of the connection, a valid handle for this specific connection is returned. Every further action of the task, such as reading or writing data, requires the declaration of a valid handle. In the event of reading or writing data, the pointer to a buffer has to be provided as well.

(2) Connectivity Interface: On each core, an independent instance of the CI is required to forward the data from applications to the CM. One CI instance can be responsible for multiple applications, whereas each application can again demand several connections to the communication network.

Connections are precisely distinguishable by their system-wide unique connection id, which is simultaneously used as a handle. Whenever a new connection needs to be established, the CI signals the CM with a mailbox message, that includes the connection information as well as a worst case bandwidth estimation based on cycle-time and data volume. On the basis of the provided information, the CM performs a bandwidth-check whether the shared communication network can still be utilized with an additional connection (see Section 4.5.2). Once a connection successfully registered at the Connectivity Manager, a FIFO send-queue on which all outgoing messages are being queued is created in the Connectivity Manager.

To receive return values and acknowledgements, the CI is in possession of a mailbox. In case of a reading or writing task, a previously attached shared memory is used to accomplish the data handover between application and CM.

(3) Mailbox: A mailbox is a real-time (RT) object provided by INtime (see Section 2.6.5) which can be used to receive mailbox messages with a size of maximum 128 Bytes. With this mailbox principle, components, such as CI and CM, can signal each other and exchange small amounts of data. The CM holds a thread that is in charge of constantly listening for new mailbox messages and performing the according action.

(4) Shared Memory: INtime also provides a RT object to share large-scale memory among cores. A shared memory (see Section 2.6.3) can be

mapped into the address space of multiple RT processes.

(5) Mailbox Thread: RT threads (see Section 2.6.4) can be assigned to wait until a certain event, such as the arrival of a mailbox message, occurs. The Mailbox Thread listens for incoming messages at the CM mailbox and performs the according actions before it starts listening again.

(6) Processing Thread: The Processing Thread is suspended as long as the message[5] FIFO send-queue is empty. Whenever a new message is copied onto the message send-queue, the Processing Thread is resumed and is suspended again once all messages were processed.

(7) Semaphore: As soon as data can possibly be accessed by two or more concurrent threads, synchronization becomes an issue. In our implementation, a semaphore is used to enable synchronization between the Mailbox Thread and the Processing Thread. With the help of a semaphore with the permission count of 1, mutual exclusion can be achieved.

(8) Connectivity Manager: As already mentioned, the Connectivity Manager is the central, main processing component. System-wide, only one CM instance is admitted, necessarily running on the primary core. The system-wide sole CM instance is responsible for multiple CI instances and is in charge of multiplexing the forwarded messages onto a shared communication bus. As part of the CM component, a scheduler is implemented that takes care of prioritizing and ordering messages for transmission (in Section 4.5, the prioritization is discussed in more detail). Furthermore, the Connectivity Manager holds two threads, Mailbox and Processing Thread . While the Processing Thread is constantly scheduling and transferring messages from the FIFO message send-queues to the communication network, the Mailbox Thread is listening for further instructions in form of mailbox messages. For the actual data exchange, we used a shared memory that is mapped into the process address space of the CM and each CI. Whereas every connection is associated with a unique *offset* within the shared memory. A semaphore provides the necessary locking functionality.

In Figure 4.7 the dotted line represents the communications paths of the mailboxes. Notably, each Connectivity Interface as well as the sole Connec-

[5] In this context a message is a CAN frame intended for the communication network, and not a mailbox message.

tivity Manager is in possession of a Mailbox **(3)**. Since all mailboxes are global RT objects, each mailbox can contact any other mailbox via mailbox message among multiple cores. However, it is not designated that Connectivity Interfaces **(2)** communicate among themselves. Furthermore, the blue straight line illustrates the data flow beginning at the Application **(1)** and ending at the Shared Memory **(4)** of the Connectivity Manager **(8)**. In contrast to the mailboxes, the shared memory exists only once and is created as well as managed by the Connectivity Manager. Every Connectivity Interfaces maps this shared memory to its own address space so that a rapid data exchange among cores is given.

Any operation of an application results in a chain of actions, beginning at the Connectivity Interface then to the Connectivity Manager next to the communications adapter and alongside the chain back to the application. The initiation of any operation, whether registering a new connection, closing an existing connection, reading or writing data, is propagated via mailboxes. While the actual data exchange is performed via shared memory, the data location is provided via a mailbox message. This communication flow, concerning the operations open, write, read and close, is further discussed in the following Section 4.4.4.

Moreover, in Figure 4.7 it is observable that the aforementioned communication scheduler is an important part of the Connectivity Manager. With the help of the Processing Thread (6), the scheduler prioritizes and processes the outgoing messages of all FIFO send-queues and forwards messages after message onto the shared network via the communications adapter. As this figure further illustrates, the communications adapter is directly attached to the Connectivity Manager, whereas it is originally hosted by the RTOS and also connected to the shared communication bus.

4.4.4 Communication Flow

In this section we outline the particular communication steps that are necessary to achieve opening and closing connections as well as reading and writing data. Since some operations are very complex and involve many communication steps, only the most important activities are illustrated in

the following sub-sections. A more detailed illustration can be found in Appendix 6.

Open

In the status quo system, a task was directly attached to the communications adapter (see Section 4.1). To stay compatible with the given interfaces, the Connectivity Interface required only a minor change in the call function for opening a new connection. We extended the parameter list with an additional connection descriptor, that holds values for cycle-time, data volume and criticality of the attempting connection.

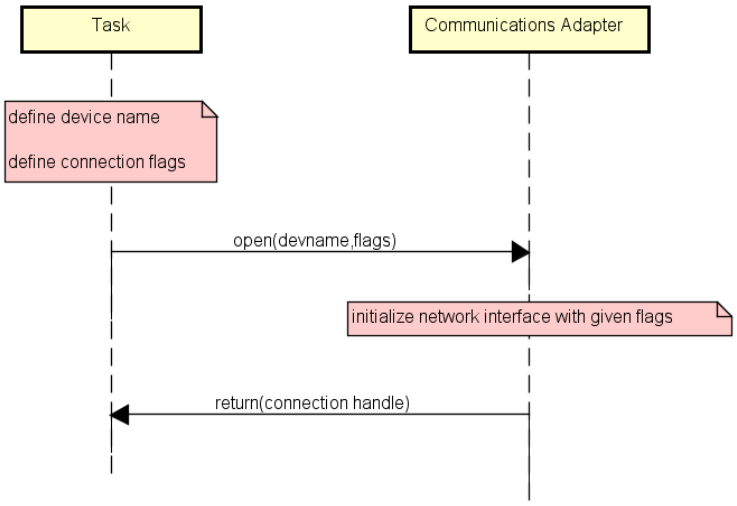


Figure 4.8: Communication flow of the old open sequence.

In Figure 4.8 the opening sequence of the status quo system can be seen. As already mentioned, the *task* communicates directly with the communications adapter and receives a connection handle in case of successful connection registration.

Since our approach relies on a pre-registration bandwidth-check, we introduced the extended function call *openExtended(devname,flags,descriptor)* for establishing a new connection. With the first two parameters (*devname*

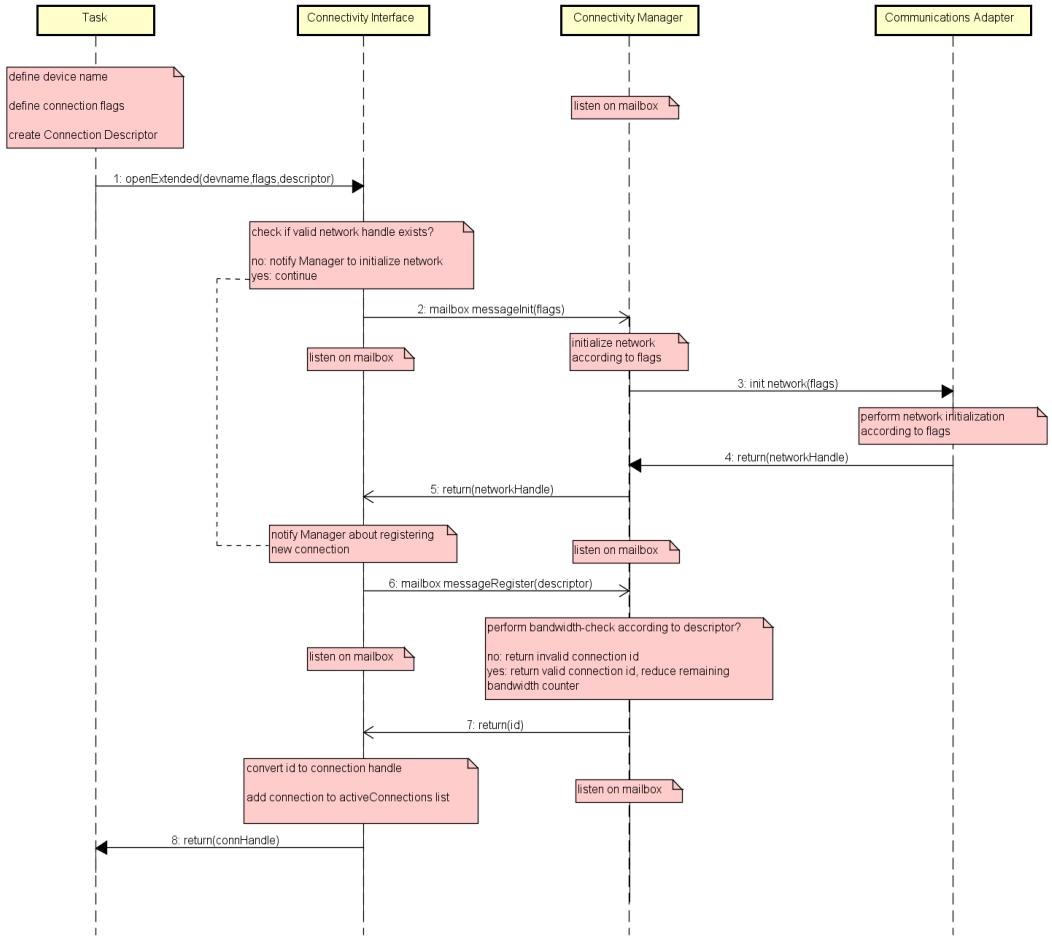


Figure 4.9: Communication flow of the extended open sequence.

and *flags*), which are used to initialize the CAN bus, we remain compatible to the old interface, while the *descriptor* parameter is used to provide additional information about the connection. As it can be seen in Figure 4.9, the *Task* performs the opening function call, including the required parameters, on the Connectivity Interface. Then, the Connectivity Interface checks if there is a valid network handle already present. If no valid network handle exists, the Connectivity Interface forwards the initialization configuration information (devname, flags) to the Connectivity Manager via mailbox message. With the help of these parameter, the Connectivity

Manager initializes the network through the communications adapter. After successful initialization, a valid network handle is fed back to the Connectivity Interface. After receiving a valid network handle, the Connectivity Interface contacts the Connectivity Manager again to register a new connection. With the aid of the supplied descriptor, the Connectivity Manager is able to perform a pre-registration bandwidth-check. A detailed discussion concerning the bandwidth-check can be found in Section 4.5.2. Basically, this pre-registration bandwidth-check examines if the attempting connection can be utilized by the shared network in terms of bandwidth demand. If there is not enough bandwidth available, the registration of the attempting connection is rejected and a invalid (negative) connection ID is returned to the Connectivity Interface. In case of a successful registration, the next free valid connection ID is returned. This ID is then converted to a connection handle (*connHandle*) that is necessary to perform any other operation, such as reading or writing data, or closing the connection. Notably, it can be seen that the Connectivity Manager as well as the Connectivity Interface are listening on their mailbox whenever a request or response is awaiting. Moreover, filled arrows represent synchronous function calls, while unfilled arrows indicate asynchronous communication via mailbox messages.

Write

In Figure 4.10, a simplified communication flow of the write operation can be seen. Once again, the Task initiates the data transmission by calling the function *write(connHandle, data)* of the Connectivity Interface. The Connectivity Interface copies the *data* buffer into the shared memory at an index *i*, that indicates the area within the shared memory dedicated to that task. Afterwards, the Connectivity Interface contacts the Connectivity Manager via mailbox message that includes the index *i*. Then, the Connectivity Manager takes this data, that is stored in the shared memory at index *i*, and segments it to network messages, in particular to CAN messages. The CAN messages are then copied onto a FIFO send-queue that was specifically created for that connection.

An additional thread picks message after message from this send-queue and forwards them one by one to the communications adapter. The communica-

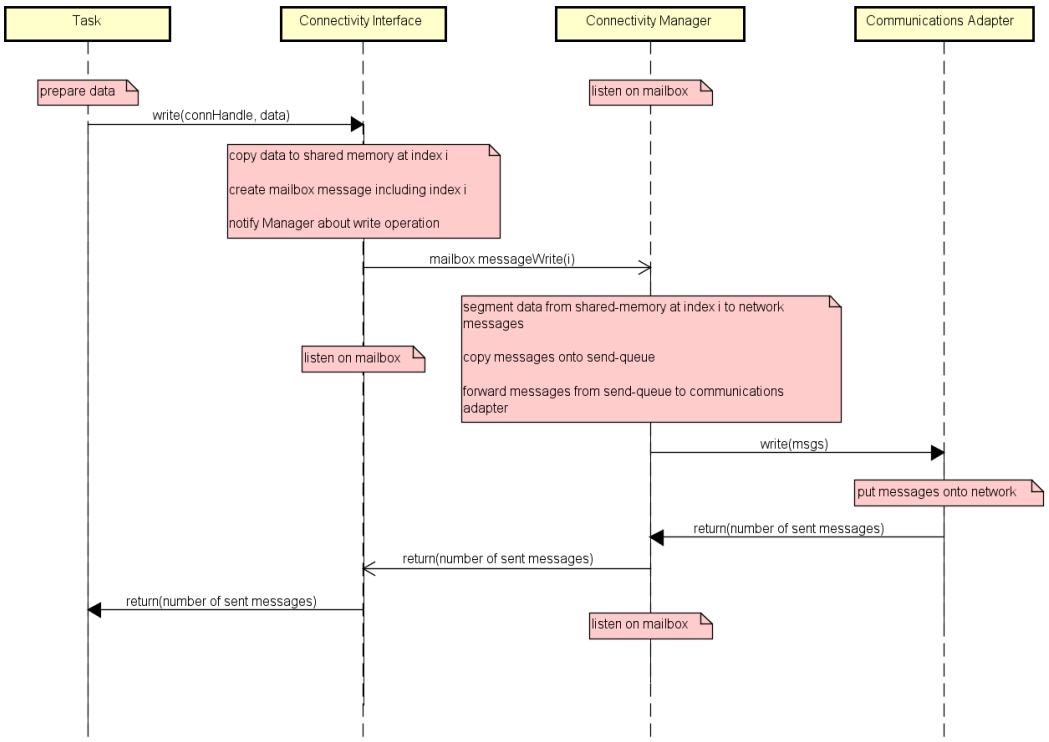


Figure 4.10: Communication flow of the write sequence.

tions adapter puts the messages onto the shared network and returns the number of messages that were transmitted. This number of sent messages is then returned alongside the communication chain to the task. As already mentioned before, this figure illustrates the simplified communication sequence for writing data. A more detailed illustration with all intermediate steps and involved threads can be found in Appendix 6.

Read

The communication flow of the read operation is similar to the write operation. The Task initiates the reading by calling the function *read(connHandle,buf, number)*, whereas this time an empty buffer as well as a pre-defined number of messages to be read is provided. The Connectivity Interface notifies the

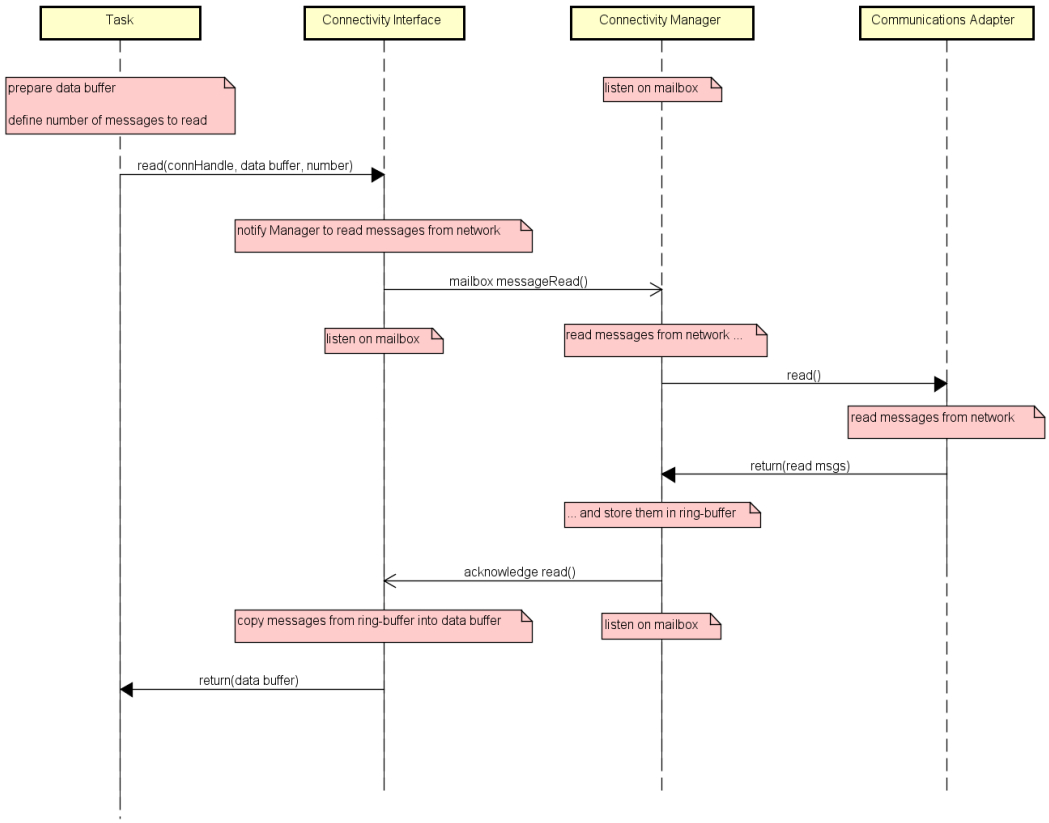


Figure 4.11: Communication flow of the read sequence.

Connectivity Manager, that further prompts the communications adapter to read all messages from the CAN bus that arrived in the meantime. After the new messages were returned to the Connectivity Manager, they are stored in a ring-buffer, as it is outlined in Figure 4.11. According to the respective read pointer of the task, relevant messages are copied from the ring-buffer into the data buffer provided by the task.

Close

In Figure 4.12, a simplified communication flow of the close operation can be seen. The task initiates the closing by calling the function *close(connHandle)* of the Connectivity Interface. The Connectivity Interface notifies the Connectivity Manager by sending a de-registration mailbox message including the connection handle. The Connectivity Manager invalidates the connection ID corresponding to this connection handle and deletes the respective send-queue.

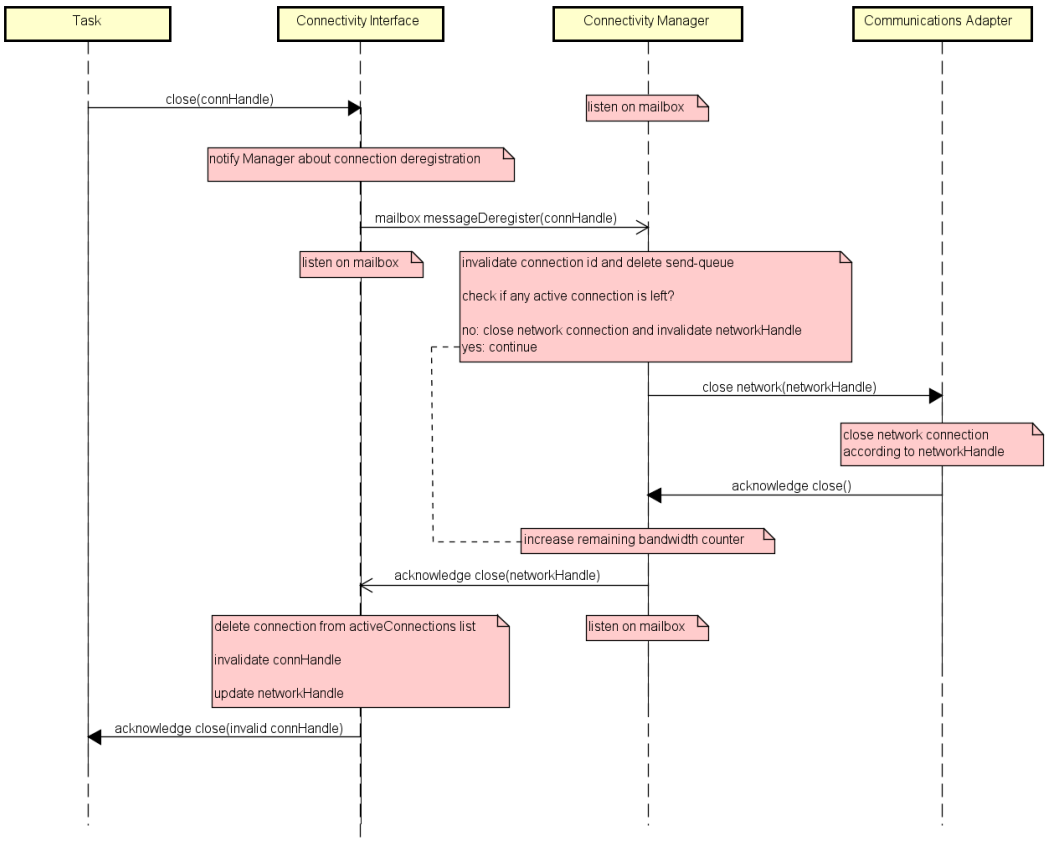


Figure 4.12: Communication flow of the close sequence.

Furthermore, it checks if there are any active connections left. In case of no remaining connections, the Connectivity Manager tells the communications

adapter to close the connection to the communication network. Thus, also the network handle of the Connectivity Manager becomes invalid. Afterwards, the counter for the remaining bandwidth is updated. As a next step, the Connectivity Manager acknowledges the Connectivity Interface. The Connectivity Interface removes the respective connection from the active connections list and invalidates the connection handle which is then returned to the calling task. Moreover, the Connectivity Interface updates the network handle accordingly.

4.5 Dynamic Priority Scheduling

In this section we describe our communication scheduling approach and emphasise the innovative contribution. In consideration of the model definition in Section 4.2, we decided to follow a dynamic priority scheduling approach. Our algorithm functions as a EDF scheduler while the system is in a non-overloaded condition, and switches to best-effort principle whenever the system becomes overloaded. Detailed information regarding the system states can be found in the following Section 4.5.1. One scenario in which the system can possibly become overloaded is, when a bottleneck on the shared communication network emerges. Meaning that the traffic on the shared communication network increased so that the dedicated bandwidth, available to the CM system, decreased. This reduced bandwidth is probably not enough to fulfil the bandwidth demands of all connections. At this point, the algorithm switches to the best-effort policy and our novel prioritization approach comes in.

4.5.1 System States

Basically, at a time, our system can only be in one out of two system states, namely overloaded and non-overloaded system state. The purpose of the following sub-sections is to provide a clear distinction between non-overloaded and overloaded system state.

Non-overloaded

The non-overloaded system state is the regular operating mode of our system. This state is the default system state and is prevalent from system start-up. Moreover, this system state is mainly characterized by the fact that it is not overloaded, meaning that the dedicated bandwidth is sufficient and messages are being processed before their respective deadlines. The scheduling of the messages is based on their deadline, also called earliest deadline first (EDF) scheduling. Therefore, as long as the system is in the

non-overloaded state, it is guaranteed that messages are being processed on time and their hard deadlines are not being violated.

Concluding, this system state is characterized by:

- EDF scheduling,
- Message order (prioritization) is based on their deadline D,
- Dedicated bandwidth is sufficient (Assumption 1 (Section 4.5.2) is true),
- Punctual transmission is guaranteed,
- Hard deadline of messages.

Overloaded

However, at every time the system might become overloaded due to network errors that reduce the actual bandwidth of the network. Consequently, the bandwidth of the network might become smaller than the bandwidth demand of the Connectivity Manager system, which is the accumulated bandwidth demand of all active connections.

The system might become overloaded due to errors on the network, or additional traffic caused by a concurrent network node. However, not only network bottlenecks but also the processing time of the Connectivity Manager can cause overload scenarios. In consequence of short cycle-times and timing drifts, the system can become overloaded as well.

Concluding, this system state is characterized by:

- Best effort scheduling,
- Message order (prioritization) is based on the criticality,
- Dedicated bandwidth is insufficient (Assumption 1 (Section 4.5.2) is false),
- Best effort transmission is performed - punctual transmission is no longer guaranteed,
- Hard deadline of messages might be degraded to soft deadlines.

4.5.2 Schedulability Test

In the literature, a schedulability test commonly refers to task scheduling on processors. In our approach, however, we seek to schedule data streams on communication networks. The Connectivity Manager holds a value that represents the total amount of bandwidth available to the CM system. This brings us to the first assumption:

Assumption 1 *The available bandwidth dedicated to the CM B_{CM} must be smaller than or equal to the overall remaining bandwidth of the communication network B_{NET}*

$$B_{CM} \leq B_{NET} \tag{4.1}$$

Assumption 1 ensures that all data streams registered at the Connectivity Manager, can make use of the underlying network.

Whenever an application attempts to register a new connection, a worst case bandwidth estimation, based on the provided cycle-time λ_n and data volume V_n, is calculated. With the help of this worst-case bandwidth-estimation and the available bandwidth of the Connectivity Manager, the second assumption can be defined.

Assumption 2 *Connection S_{NEW} with bandwidth B_{NEW} is accepted, if:*

$$B_{NEW} + \sum_{j=1}^{n} B_j \leq B_{CM} \tag{4.2}$$

Assumption 2 shows the previously mentioned bandwidth-check performed by the Connectivity Manager. Before a new connection is accepted, the Connectivity Manager checks the utilized bandwidth, by summing up the worst-case bandwidth-estimations of all registered connections. If the bandwidth of the attempting connection plus the utilized bandwidth is smaller than or equal to the dedicated available bandwidth of the Connectivity Manager, the attempting connection is accepted. This bandwidth-check assures that the specified data volume of a data stream can be transferred within the respective cycle-time. In other words, the bandwidth-check guarantees that

even peak load scenarios, where connections attempt to send data at the same time, can be scheduled and handled by the underlying communication network. As long as all connections adhere to their periodical data volume, the maximum utilized bandwidth at a given moment follows Assumption 2. However, due to our dynamic approach, unexpected sporadic bursts are being handled as additional connections and have no major impact on the system, assuming that the underlying communication network retains enough capacity.

Additionally to the bandwidth-check, also the processing time of a message needs to be considered to ensure the schedulability of a message. Within this thesis, this message processing time is defined as worst-case execution time (WCET). The WCET consists of the time that is necessary to copy all messages of the data package onto the respective send-queue, and the time that is needed to pick one message from the send-queue and forward it to the communications adapter. Although these proceedings are constant in time, the executing thread might get interrupted. Thus, we are talking about a worst-case execution time instead of a processing time. Moreover, the WCET of the first message of the data package is shorter than of the last message of the data package, since the last message has to wait longer for transmission. Furthermore, also the number of concurrent connections influences the WCET of a message. For example, the message of *Connection A* might be favoured over the message of *Connection B* due to higher criticality or smaller deadline (in terms of closer).

For simplicity reasons, in the following Assumption 3 we consider only one connection and only the first message of a period. Meaning that thread interruptions are not possible and thus a seamless processing is enabled.

Assumption 3 *Let n' be the number of messages in a period. Thus, the worst-case execution time of the first message consists of the copy-time of all messages plus the picking-time and transmission-time of the respective message. To guarantee punctual transmission of the messages, the cycle-time of the data stream must be larger than the WCET.*

$$WCET = \left(\sum_{j=1}^{n'} copyTime \right) + pickingTime + transmissionTime$$

$$WCET \leq \lambda \tag{4.3}$$

The formula to calculate the WCET for a connection that sends more than one message in each cycle, is as follows:

$$WCET = \left(\sum_{j=1}^{n'} copyTime \right) + \left((pickingTime + transmissionTime) * n' \right) \tag{4.4}$$

So if there are 3 outgoing messages in a period, the WCET is composed of the copy-time of 3 messages plus the picking-time and transmission-time of a message times 3. Whenever the WCET of all outgoing messages is smaller or equal than the cycle-time, then it is guaranteed that all messages are schedulable within a period.

Since connections usually send more than one message per period, Formula 4.4 can be considered as the generally valid WCET calculation formula. Further information concerning WCET can be found in the analysis Section 5.4.1.

4.5.3 Deadline Calculation

Each message, regardless from which data stream, holds a value that represents the *deadline*. Messages within the same data package hold the very same deadline. As already mentioned above, each data package is marked with a value that represents the *creation-time* of this data package, and its included messages. This creation-time is essential for the deadline calculation of the messages. Considering the creation-time more precisely, the corresponding period in relation to the start-time can be determined. Being aware of the corresponding period, the deadline of the messages on herein before mentioned data package is exactly the end of this corresponding period. While the cycle-time λ is a constant time value, the deadline, start-time T_{START} and the creation-time T_{CREATE} are absolute timestamps with the accuracy of 100 nanoseconds.

Formula 1 *With the use of the start-time T_{START}, the creation-time T_{CREATE}, and the cycle-time λ (assuming $\lambda > 0$), the corresponding period can be determined with the following formula:*

$$period = \left\lceil \frac{T_{CREATE} - T_{START}}{\lambda} \right\rceil \tag{4.5}$$

Formula 2 *The actual deadline D can then be calculated by the following formula while respecting the result of Formula 1:*

$$D = \begin{cases} T_{START} + \lambda, & \text{if } period = 0 \\ (period * \lambda) + T_{START}, & \text{if } period > 0 \end{cases} \tag{4.6}$$

Whereas the case "if $period = 0$" in Formula 2, is for the special use case, in which the connection sends data for the very first time. Consequently, the creation-time is equal to the start-time ($T_{\mathrm{CREATE}} = T_{\mathrm{START}}$), so that the result of the period is 0.

Example 1 *In this example the cycle-time is defined to be 10 milliseconds. Since all other time indications are in the order of 100 nanoseconds, the 10 ms are multiplied by 10000 to be compatible.*

$$period = \left\lceil \frac{6361870838255\underline{5}6071 - 6361870838252\underline{8}3603}{\boldsymbol{10} * 10000} \right\rceil$$

$$period = \lceil 2.72468 \rceil$$

$$period = 3$$

$$D = (3 * 10 * 10000) + 6361870838252\underline{8}3603$$

$$D = 6361870838255\underline{8}3603$$

The digits that are underlined in Example 1, represent the significance ranging from 10 milliseconds to 100 nanoseconds. The digits that are bold represent the significance of 10 and 1 millisecond, which is of most relevance. Notably, only the bold digits change in this calculation due to the cycle-time of 10 milliseconds. Thus, everything behind the bold digits remains unchanged. As it can further be seen in this example, the creation-time falls into period 3, indicated by the value 2.72. Consequently, the deadline is determined to be the end of period 3. In the following Table 4.4, the start and end times are outlined, whereas we assume that the start-time T_{START} is also the beginning of the first period.

Table 4.4: Detailed information concerning periods.

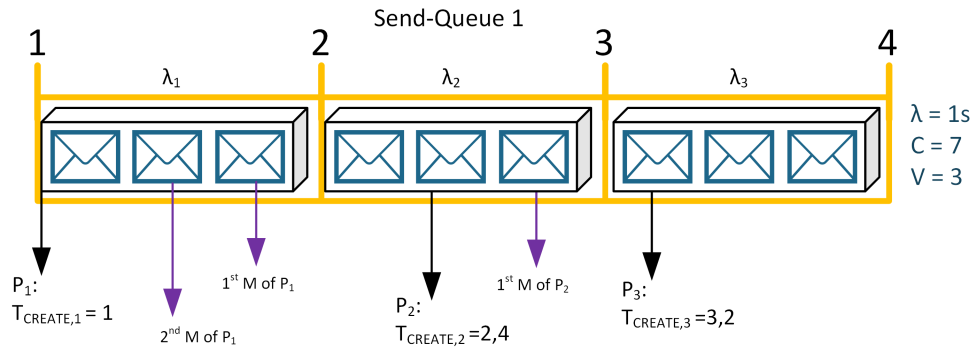
Period	Start	End
1	...5**2**83603	...5**3**83603
2	...5**3**83603	...5**4**83603
3	...5**4**83603	...5**5**83603

In Table 4.4 it can be clearly seen, that the deadline (D) of Example 1 corresponds to the end of period 3. For readability reasons, the unchanged digits in front of the 100 milliseconds significance are being replaced by dots.

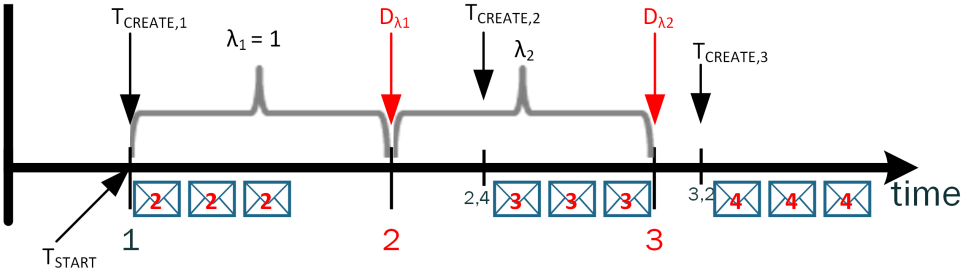
4.5.4 Functioning of the Scheduler

In this section we want to illustrate the functioning of the scheduler by going through a simple scenario. In Figure 4.13a we outline a scenario with the same preconditions as already used in Section 4.2. This scenario consists of only one data stream (S_1) and therefore also only one send-queue (Q_1). Whereas, for simplicity, the unit of the data volume is defined as messages per period instead of byte per period. Since only one data stream is being considered, the criticality can be neglected.

In Figure 4.13b the functioning of the scheduler is illustrated. The processing of the scheduler can be imagined as an abstract timeline on which all upcoming events are scheduled. Thus, events, such as instant of start, instant



(a) Parameter visualization of the data stream.



(b) Timeline that illustrates the scheduling of the simple send example.

Figure 4.13: A simplified example of a data stream sending messages.

of creation, period beginning and ending as well as the number of messages within a period, can be observed. The numbers in red indicate the respective deadline of the message which is the end of the period. Furthermore, the cycle-time of the data stream is set to 1 second. Notably, data packages are not created in constant intervals due to poor clock resolution and drifts of the task. That effect is indicated by the different creation-times in Figure 4.13a.

The functioning of the scheduler can further be outlined with the help of a flow diagram, that can be seen in Figure 4.14 and Figure 4.15.

In Figure 4.14, the main contributing classes and how they are related and cascaded are observable. Moreover, the interaction of the two major threads, namely *mailbox thread* and *processing thread*, can be seen. The mailbox thread is constantly listening for new instructions in the form of mailbox messages. Whenever data is intended to be written, the mailbox thread copies the new data onto the *send-queue* before it wakes up the processing thread, that is awaiting data to be processed.

As a first action, the processing thread checks if the *send-queue* at the first position of the *processing list* is empty. Due to our sorting algorithm, an empty send-queue at the first position of the processing list implies that all following send-queues are also empty. In the case that the first send-queue is empty, the processing thread knows that there are no messages of neither send-queue to process and puts itself to sleep again. If the first send-queue is not empty, the thread starts processing the messages by initially locking the processing list and the send-queues by acquiring a *semaphore* unit.

After that, the scheduler class is invoked by the function call *pickMessage()*. The communication scheduler stores the current delay state of the system to the variable *old delay*, that will be used later on in the *clean up* procedure. Then the send-queue at the first position is called, since this queue contains the message that will be sent next. For plausibility reasons, it is checked that this send-queue contains at least one message, if not, an invalid message (error message) is returned. Otherwise, the first message is popped from its queue and is set to be the *return message m*. Moreover, the message count is decreased due to the removal of the first message. Next, it is checked, whether the system-time is greater than the respective deadline of the message, which would mean that the message is being processed after

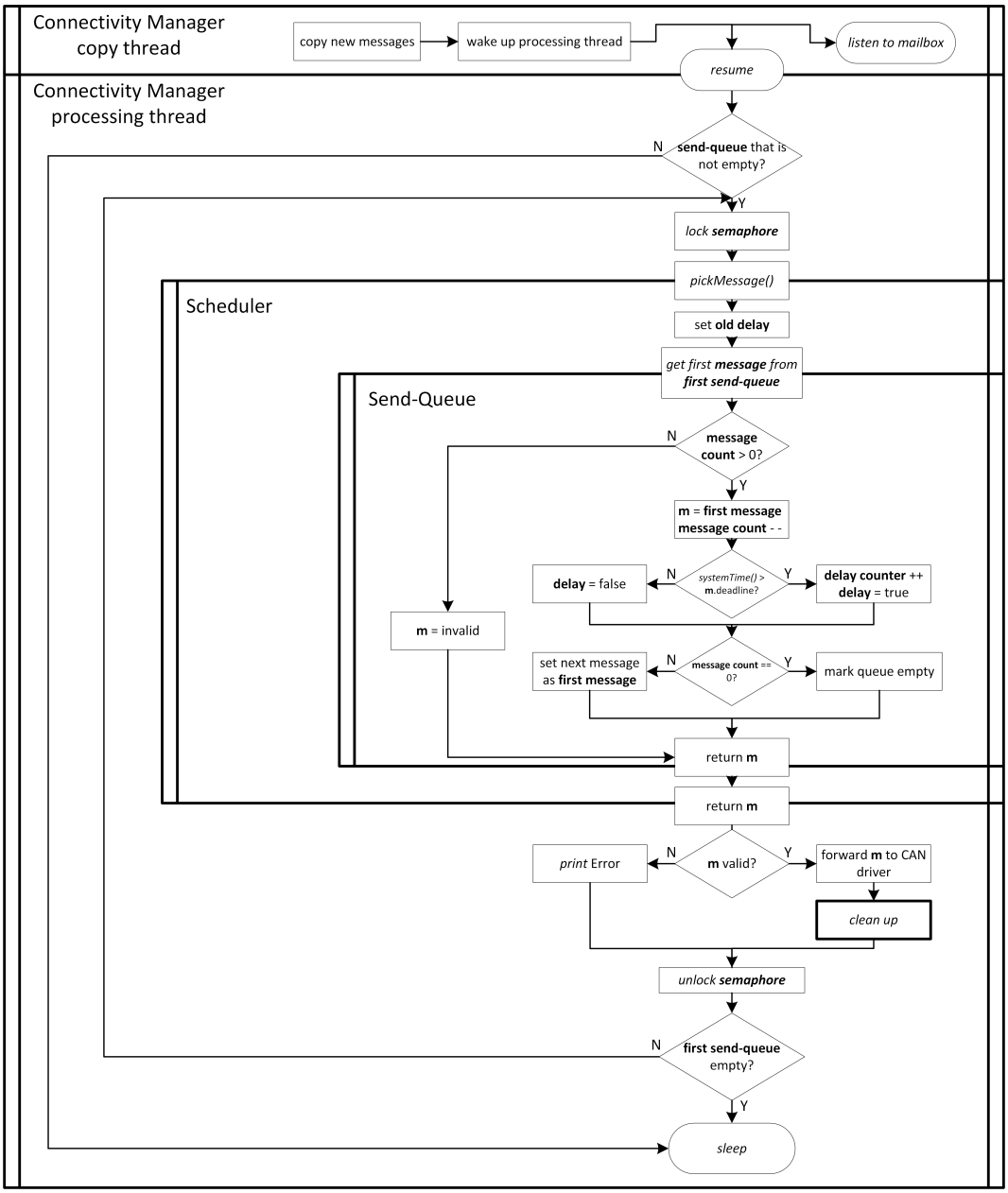


Figure 4.14: Flow diagram of the message picking procedure.

its deadline. In that case the deadline was violated and a delay occurred. Therefore, the delay counter is increased and also the delay state of the system is set to true.

As a last step of the send-queue class, it marks itself as empty in the case that no further messages exist. Otherwise, the next message is set to be the *first message* of the queue. Eventually, the return message *m* is returned through the scheduler class back to the Connectivity Manager. There, the message is checked for validity and if valid it is forwarded to the CAN driver. Afterwards a *clean up* procedure is performed before the *semaphore* is released again. This whole routine is performed as long as the send-queue at the first position contains messages and is therefore not marked as empty. Otherwise, the processing thread puts itself to sleep again.

In the following Figure 4.15, the aforementioned *clean up* procedure, which is no less important, is explained.

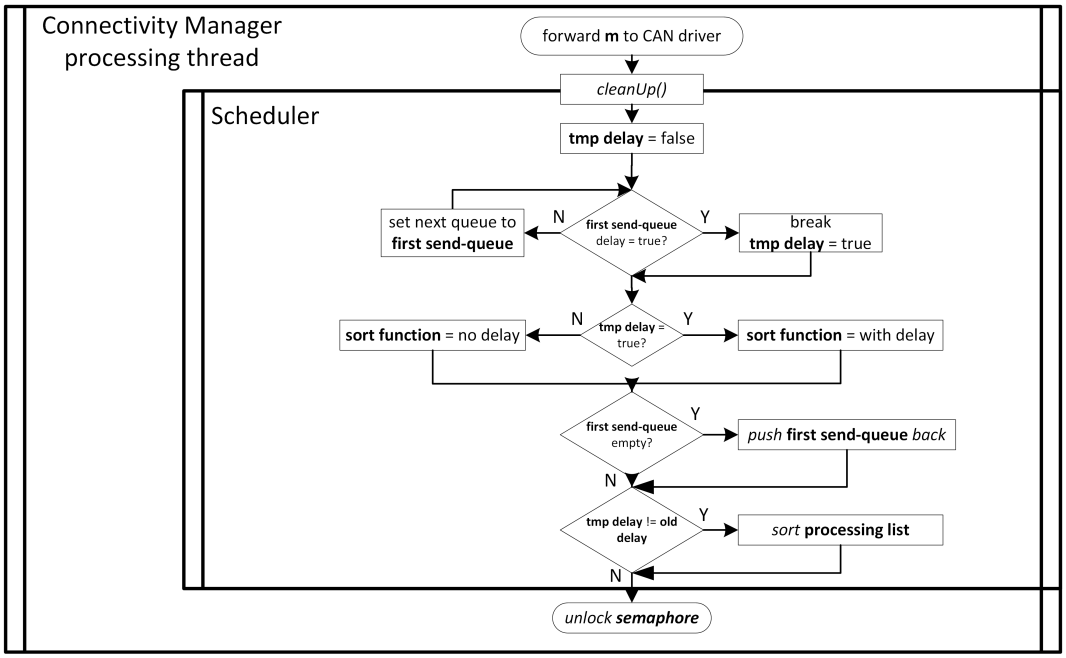


Figure 4.15: Flow diagram of the clean up procedure.

While the initiation of the clean up is done by the Connectivity Manager class, the conduction of the procedure is handled by the scheduler class. As

a first step a *temporary delay* variable, that is later taken for comparison, is set to false. Afterwards, a foreach loop iterates through all send-queues and checks whether the queue is marked as *delayed*. If so, the temporary delay variable is set to *true* and the loop is cancelled.

Next, the sort function is assigned according to the value of the temporary delay variable. With the help of a function pointer the sort function is set to either sort with regard to delay or without any delay considerations. Sorting with regard to delay means that the criticality is of highest priority, while not considering delays means that the deadline is the most important sorting criteria. Furthermore, the send-queue at the first position is checked whether it is empty, and if so it is pushed back to the last position.

Additionally, the value of the temporary delay variable is compared to the value of the old delay variable, that was set in the previous picking procedure. Especially the last two activities of the scheduler are responsible that it is guaranteed that if the send-queue at the first position is marked as empty all other send-queues are empty as well. Finally the clean up is finished and the thread returns from the scheduler class to the Connectivity Manager.

To explain the functioning of the scheduler by a more complex example, we consider the following Figures 4.16 and 4.17.

The preconditions of the following explanatory example are as outlined in the following Table 4.5:

Table 4.5: Preconditions of the send example.

	S1	S2	S3
data volume V [msgs]	3	2	1
cycle-time λ [sec]	1	0,7	0,7
start-time T_{START} [time]	1	2	1
send-queue (Q)	Q1	Q2	Q3

As it can be seen, the example consists of three data streams (S1, S2, and S3) and therefore three send-queues (Q1, Q2, and Q3). For a better understanding of the visualization of the example, the data volume was again simplified to *messages* (instead of byte) and the cycle-time is defined in the

order of *seconds*. The start-time is also declared in the order of seconds, however it refers to a point in time on the visualized timeline of Figure 4.16 and 4.17. In the first step **(1.)** of the complex sending example, it can be seen that each send-queue consists of three data packages, donated by P. The creation-time indicates at which moment of time the respective data package was created. The processing list shows the sorting of the send-queues by deadline ascendantly, with the consequence that the send-queue with the closest deadline is on top of this list and will be processed next. In the timeline visualization on the right side, all events, such as the deadline (D) of a message, the cycle-time (λ) of a data stream, the start-time (T_{START}) of each data stream as well as the processing of the messages is illustrated. On the very right side, the message that is sent within this processing step is shown. The blue dotted line represents the system-time with respect to the timeline. In the first processing step **(1.)** the single message of data package 1 of send-queue (Q3) is sent, since Q3 with its closest deadline (1.7) is listed on top of the processing list.

Notably, in the second processing step **(2.)** it can be seen that the processing list changed and also the system-time on the timeline advanced. The labels of send-queue 3 (Q3) were also updated. However the creation-time (1.9) of Q3 is not yet reached, that is why the deadline of Q3 is ∞ in the processing list. Due to this work around the send-queue is marked as empty and no longer considered for processing (but also not deleted for performance reasons). Furthermore, the envelope icon is displayed only after the data package was created. Consequently, Q1 is the only send-queue that is being processed in the **steps 2.-4**.

In the fifth processing step **(5.)**, the second data package (P2) of Q3 was created, hence send-queue 3 is listed in the processing list again. Since its deadline (D2) is the closest, the message of data package 2 of send-queue 3 is sent in this processing step.

In the next processing step **(6.)**, also send-queue 2 (Q2) becomes involved due to the creation of its first data package. Since Q2 is sorted to the top of the processing list, the first message of data package 1 (P1) of send-queue 2 (Q2) is sent in this processing step.

In the last processing step **(7.)** the circumstances are very similar to the previous processing step, hence the second message of data package 1

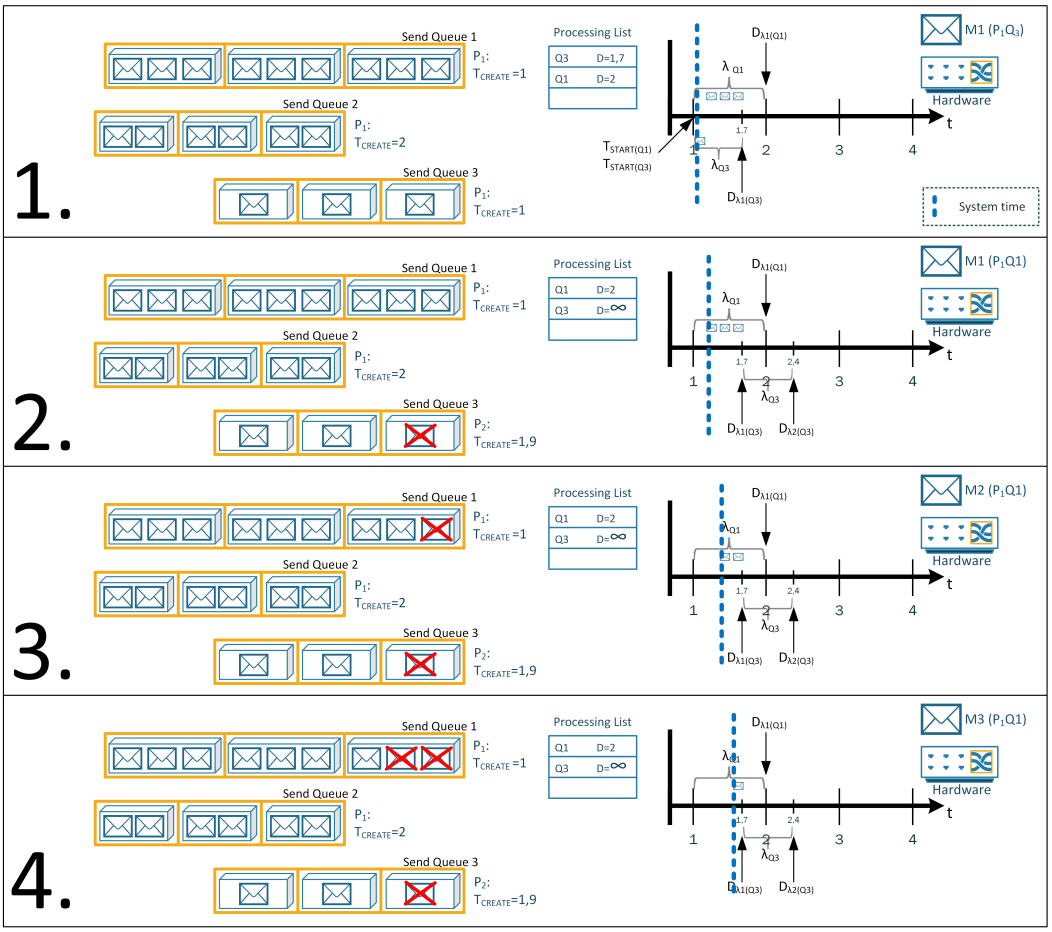


Figure 4.16: A more complex example of the send procedure.

(P1) of send-queue 2 (Q2) is sent. Notably, send-queue 1 is listed in the processing list again since the second data package was created.

4.5.5 EDF Scheduling (non-overloaded)

As mentioned before, for each registered connection the Connectivity Manager creates a FIFO message send-queue. By the time a connection attempts

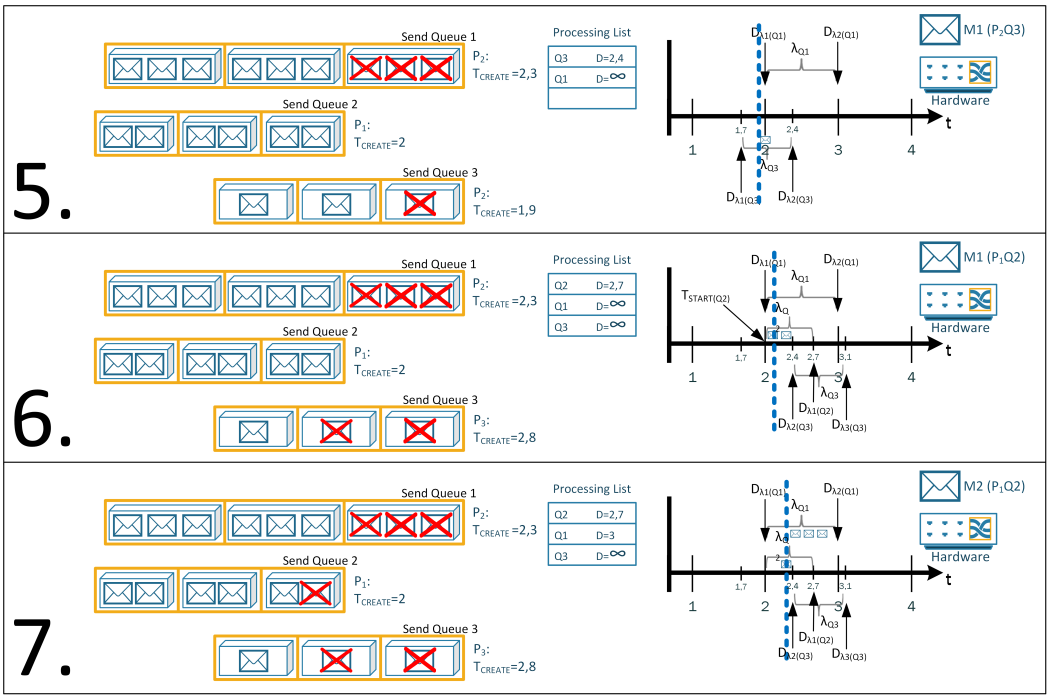


Figure 4.17: A more complex example of the send procedure (cont.).

to write, the according data is split into CAN messages which in turn are copied onto the send-queue. The number of resulting frames depends on their data volume, for example:

Example 2

$$\frac{V_n\ [Byte]}{msgSize\ [Byte]} = \frac{100}{20} = 5\ CAN\ Messages$$

Whereas 20 bytes indicate the worst-case estimation for the size of a CAN message. To follow up Example 2, in each period 5 CAN messages are copied onto the send-queue. Every copy process includes a *creation-time* which is a timestamp denoting when the *data package* P_l was created. At the very first write attempt, this creation-time is also used to define the beginning of the first period. The creation-time in combination with the

cycle-time λ_n is taken to determine the deadline of each message. Whereas messages which fall into the same period have an equal $D_{l,n}$, in fact the end of the period.

The main advantage of a creation-time is twofold. On the one side, period as well as message counters, that could be used to determine the respective period as well as the corresponding deadline, become unnecessary what furthermore limits the occurrence of overflow errors. On the other side, not relying on counters makes the system more flexible in the sense that connections can send a varying amount of data without confusing the scheduler. Thus, connections can skip periods or send more data than specified by the data volume, which enables us to consider sporadic tasks as well. Thus, due to a creation-time the system is independent of internal counters and thus makes it more flexible.

The scheduler is active as long as there is a message that has not yet been picked from its send-queue. Hence, in EDF mode the scheduler processes messages as fast as possible in the order that the message with the earliest deadline is picked next. This behaviour has the advantage, that data can even be processed in advance.

In a non-overloaded system, every single deadline is met and no delays occur. Whenever a delay occurs, which means that a message is processed only after its respective deadline, the scheduler switches to best-effort mode (see Figure 4.18).

4.5.6 Best Effort Scheduling (overloaded)

In the event of a deadline violation, the whole send-queue gets marked as delayed and the best-effort scheduling mode is applied. Whilst at least one send-queue is marked as delayed the whole system remains in the best-effort scheduling mode. Since the initial bandwidth is not fully available anymore, the bandwidth demand of all connections can no longer be fulfilled. One of the main characteristics of the best-effort approach is, that it performs a degradation of less important connections/data streams. Concretely, the scheduler resorts the message order in a way that connections with a high criticality have the highest priority. As a result, connections that registered

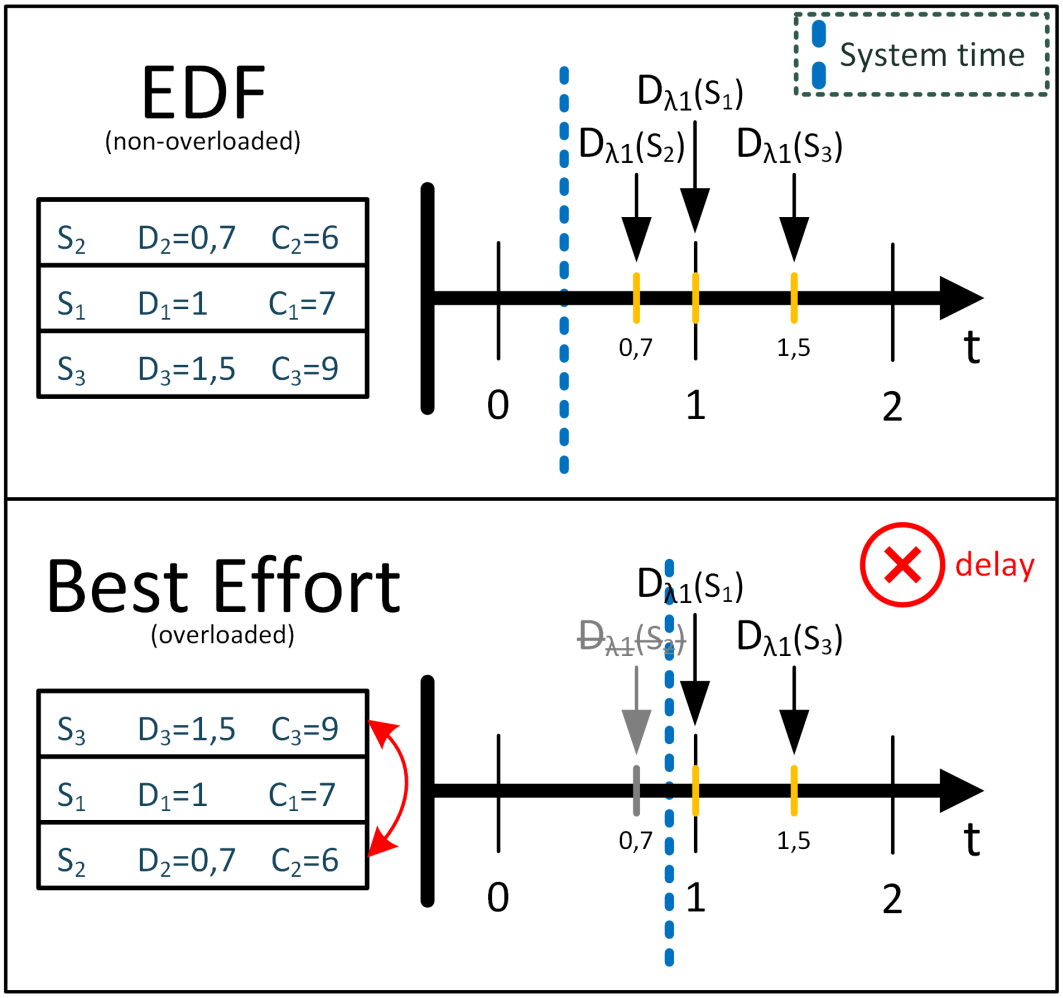


Figure 4.18: Reordering of the processing queue in the event of a deadline violation, occurring whenever the system-time overtakes a valid deadline. The processing queue is then sorted by criticality C_n descendingly, so that the connection with the highest criticality is on top and will be processed next.

with a high criticality are favoured in the case of an overloaded system. With this approach we achieve that highly critical applications can continue to function in a real-time manner, while less critical applications are degraded in the sense of timeliness. Hence, degraded messages might get queued onto their respective send-queue. However, messages are only queued as long as a certain memory threshold is not reached. Further information about this threshold, known as watermark, can be found in Section 4.6.3.

As soon as the bottleneck on the network disappears, our system is capable of catching up on the delayed connections. Since our approach processes messages as fast as possible, the demonstrator is able to consume the available bandwidth of the whole network. This effect is discussed in Section 5.3.2. The scheduler switches back to EDF mode when all queued messages have been processed and every send-queue is marked as *not delayed*.

Concluding, our algorithm performs according to the earliest deadline first (EDF) principle and switches to best-effort approach once the system becomes overloaded. In the best-effort approach, low critical data streams are degraded to assure real-time processing of high critical data streams. Meaning that deadlines of low critical messages might get violated to adhere the deadline of high critical messages.

4.6 Implementation

In this section we want to outline further information regarding implementation details. The implementation of the demonstrator was done in the programming language C++ in the version C99. Due to the real-time requirement we used the real-time operating system RTOS INtime 4.20 in the 32-bit version. With regard to real-time capability, we made use of the real-time third-party library Boost[6], which we used as a basis for some few components such as the ring-buffer (further details in Section 4.6.5). Since this demonstrator was mainly developed for the AVL List GmbH, it was a primary requirement to integrate and utilize their self-developed real-time framework called ARTE (AVL real-time environment). As for the

[6] `http://www.boost.org/`

communication network, we decided to use the Controller Area Network CAN, since it is the most common and utmost widespread in-vehicle communication bus. However, as already mentioned earlier, neither our design nor the implementation precludes a future adoption to additional communication technologies, such as CAN-FD or Automotive Ethernet. Yet, our implementation was optimized to CAN which is discussed in more detail in Section 2.3. This section is also dedicated to present the concrete structure of the Connectivity Interface and Connectivity Manager class. Eventually, we also address the limitations of our demonstrator.

4.6.1 Connectivity Manager

In Figure 4.19, a class diagram illustrates the structure of the Connectivity Manager class. Notably, the Connectivity Manager class holds several dependencies to other classes and structures (C++ structs). The classes *Mailbox* and *SharedMemory* are reference classes from the ARTE framework, which extends the functionality of the actual INtime RT objects. Except for the RThandle which is an INtime real-time object, all other classes and structs were designed specifically for our needs. As it can be seen, the Connectivity Manger is in possession of an *scheduler* object which in turn is in possession of all send-queues[7]. As outlined in Figure 4.19, a send-queue has a start-time, a deadline (which is the deadline of the first message in the queue), and also a flag that indicates if the queue is delayed. Furthermore, the Connectivity Manager is in possession of a *BusHandle*, which is specified by the according *BusType* such as CAN. At this point, we tried to keep the design and also the implementation as generic as possible to potentially support multiple network technologies. Additional attributes of the Connectivity Manager are total size of the shared memory (*shdMemorySize*), the next free shared memory offset (*shdMemoryOffset*), as well as the next free ID (*nextFreeId*). Since every connection obtains a unique identifier (connectionId), a specific mapping between connection and its reserved data slot in the shared memory(*shdMemoryIndex*) can be accomplished. The unique connectionId is also used to track active connection (*activeConnections*) with reference to the respective *connectionDescriptor*.

[7]In the program code theses send-queues are being named ”ConnectionSendLists”.

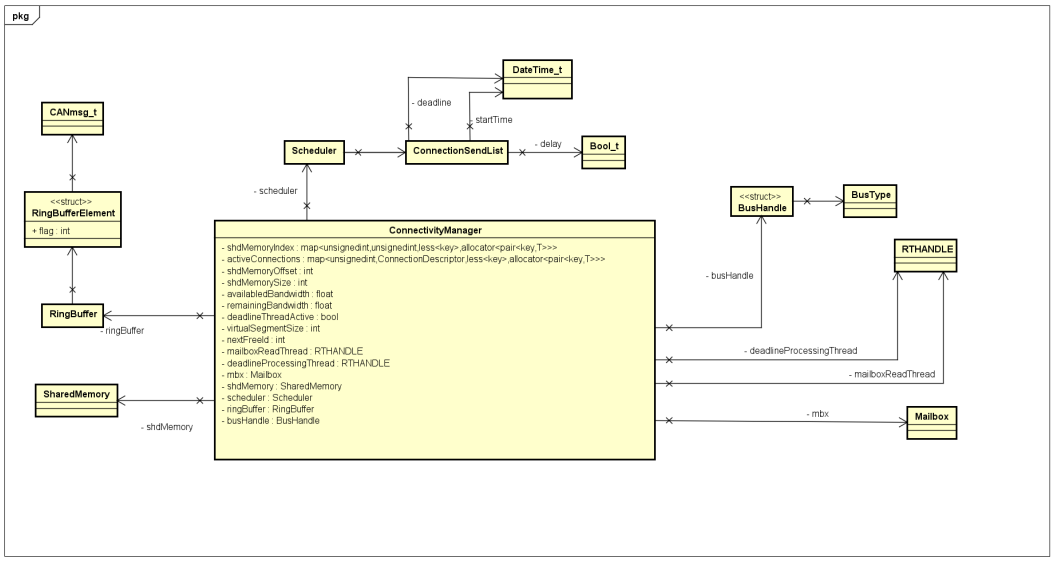


Figure 4.19: Connectivity Manager - class diagram.

Moreover, attributes regarding the bandwidth and the virtual segment size are available to the Connectivity Manager. The oft-quoted processing thread (*deadlineProcessingThread*) and mailbox thread (*mailboxReadThread*) can also be seen in this class diagram. An equal important attribute of the Connectivity Manger is the *ringBuffer* object that is derived from the *RingBuffer* class which in turn is based on the CircularBuffer class of the real-time third party library Boost. Notably, the ring-buffer is being filled by objects of the *RingBufferElement* struct. Furthermore, at this point we tried to keep the design as well as the implementation generic so that multiple message formats are potentially supported.

A more detailed version of this class diagram including additional attributes and methods can be found in the Appendix 6.

4.6.2 Connectivity Interface

In Figure 4.20, the structure of the Connectivity Interface class is outlined in the form of a class diagram. As it can be seen, the Connectivity In-

terface holds several dependencies to other classes such as Mailbox and SharedMemory, which are both classes provided by the ARTE framework. Moreover, the Connectivity Interface holds a handle to the process of the Connectivity Manager as well as to its mailbox. Additional member attributes are a *busHandle*, a map of active connections (*activeConnections*) as well as a mapping of connections to their slot in the shared memory (*slotIndex*). As already mentioned at the Connectivity Manager class, the connectionId is unique so that a distinguishable mapping can be accomplished. Furthermore, the attribute *nodeId* represents the ID of the INtime node on which this particular Connectivity Interface is running. Since multiple instances of the Connectivity Interface are permitted, the nodeId is used as an naming extension to uniquely distinguish between instances of the Connectivity Interface.

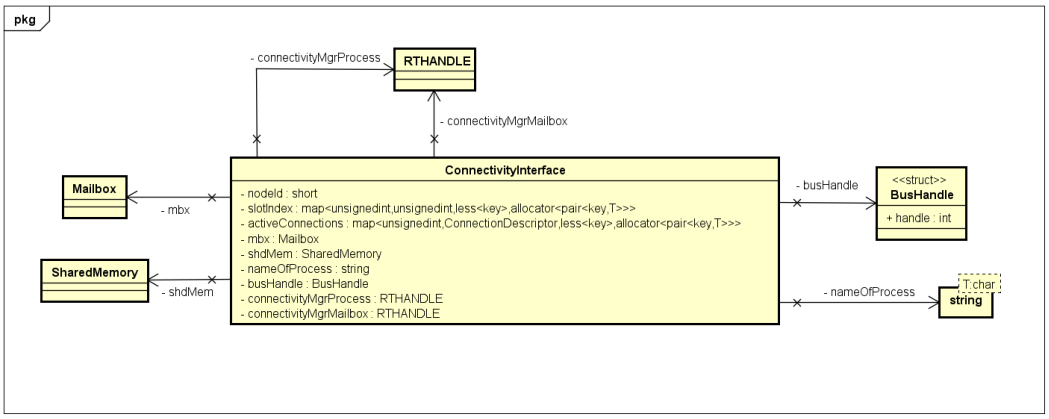


Figure 4.20: Connectivity Interface - class diagram.

The attribute *nameOfProcess* indicates the name of this Connectivity Interface process that follows the naming convention of the string *"ConnInf"* concatenated with the aforementioned nodeId. Due to this unique process name, this instance of the Connectivity Interface can be looked up across nodes/cores.

A more detailed version of this class diagram including additional attributes and methods can be found in the Appendix 6.

4.6.3 Inter-Core Communication

Since the concept of the inter-core communication was extensively discussed in the prior Section 4.4.3, we only emphasise on the implementation details in this section. For the implementation of the mailboxes as well as the shared memory we used the service of the ARTE framework that extends the functionality of basic RT objects of the INtime RTOS. As previously mentioned, any communication between Connectivity Interface and Connectivity Manager is propagated via mailboxes, whereas the data exchange, regardless of reading or writing data, is performed with the help of the shared memory. This procedure is illustrated in Figure 4.21.

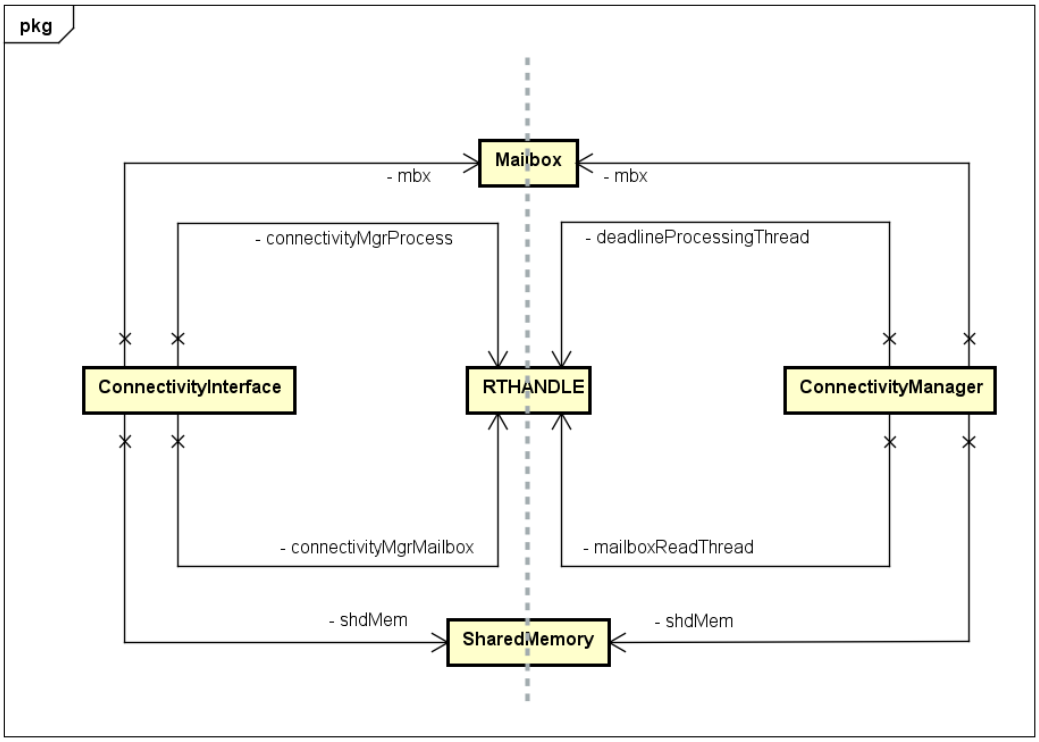


Figure 4.21: Communication between Connectivity Manager class and Connectivity Interface class.

As it can be seen in Figure 4.22, both the Connectivity Manager and the Connectivity Interface hold a reference to the shared memory that was

initially created by the Connectivity Manager and afterwards mapped by the Connectivity Interface. Likewise, both classes create their own mailbox to enable communication among cores. However, the Connectivity Interface has also a RT handle that references to the mailbox of the Connectivity Manager. At the Connectivity Interface there is also a RT handle referencing to the process of the Connectivity Manager.

The Connectivity Manager on the other side holds RT handles to its processing thread and mailbox thread. The dotted line in this figure symbolises the separation line between those two components.

In the upper area of Figure 4.22, the enumeration *MessageType* can be seen with all different types of use cases a mailbox message can be used for. For example to tell the Connectivity Manager to initialize the bus, the Connectivity Interface sends the mailbox message MailboxMessageInitializeBus that is marked with the *MessageType Bus_init*. All message types on the left hand side of the dotted line are used by the Connectivity Interface to initiated the according action. Remarkably, the NodeId is part of every initiation message so that the Connectivity Manager can respond to the right Connectivity Interface. The Connectivity Manager responds with the respective acknowledgement, that is illustrated on the right hand side of the dotted line. With varying combinations of positive and negative values for *connectionId* and *value* the Connectivity Manager informs whether the action was successful or not. Notably, the MessageType is simply an indication within the mailbox message and is not a mailbox message type distinction as between Object Mailbox and Data Mailbox (outlined in Section 2.6.5).

Watermark

As mentioned earlier in the Best Effort Scheduling section 4.5.6, messages are queued onto the send-queue once the system is in a overloaded state. However, the queuing is only performed until a certain threshold is reached. This threshold is called watermark and is implemented to avoid errors such as buffer overflows and out-of-memory exceptions. In particular, the watermark is a value that is calculated for every send-queue. Since, the Connectivity Manager knows about the total amount of memory (virtual segment - VSEG) dedicated to itself, a memory distribution among the

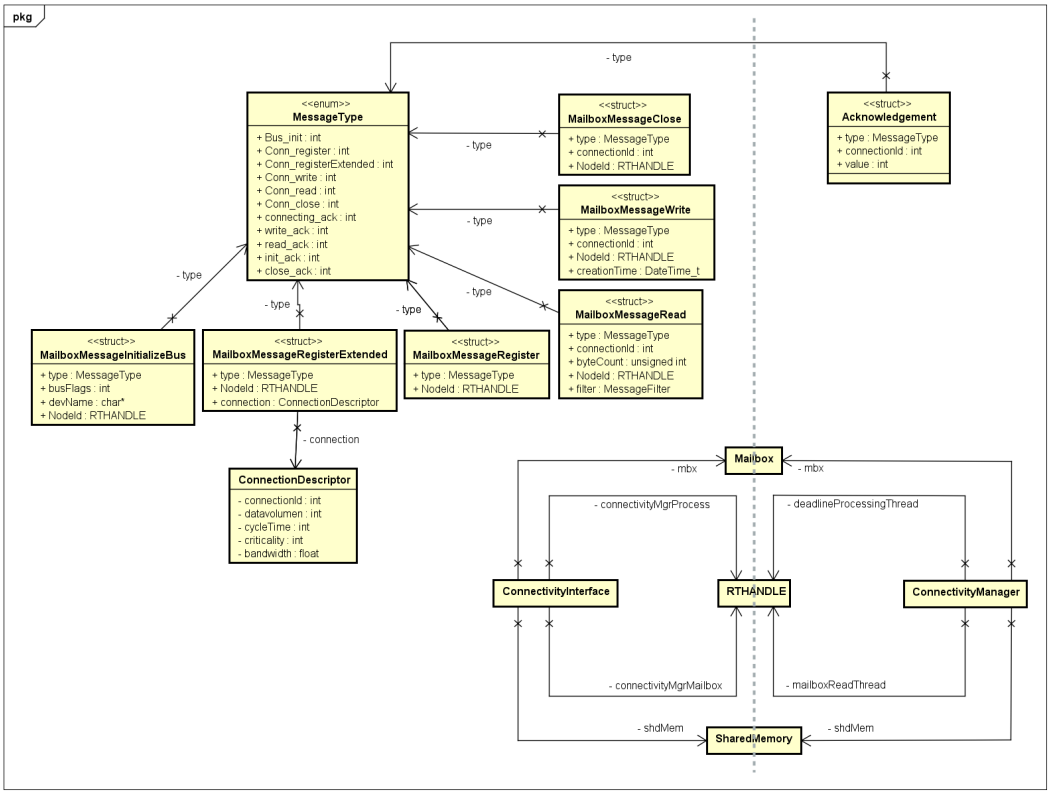


Figure 4.22: Communication between Connectivity Manager class and Connectivity Interface class including message types.

connections can be achieved. However, only 80% of the total amount of memory (VSEG size) is taken for queuing messages onto send-queues. Thus, 20% of the VSEG size remains as a buffer for the Connectivity Manager process, for its threads, as well as for heaps and stacks. Meaning that the sum of all watermarks of all connections is maximally 80% of the VSEG size. The watermark of a connection is proportionally to the bandwidth demand of the respective connection. Consequently, if a connection demands 15% of the available bandwidth, the watermark limits the memory consumption of the respective connection to 15% of the memory dedicated for queuing messages, which is 80% of the total amount of memory (VSEG size).

4.6.4 Scheduler

For the implementation of the scheduler, we decided to use a list, called processing list, that contains entries of send-queues. Since send-queues are implemented as FIFO lists, the processing list can be seen as a list of lists/queues. Due to this approach, the length of the processing list is of size n, whereas n devotes the number of registered connections. Hence, the sorting of the processing list can be accomplished in $O(n \log n)$ time [8], instead of $O(m \log m)$ time, whereas m indicates the number of all messages which is mostly greater than n. A function pointer is used to enable two different sorting approaches. One for sorting when the system is in the non-overloaded state *sortFunctionNoDelay()* and one for sorting as long as the system is overloaded *sortFunctionWithDelay()*. Thus, the criteria in relation to their importance with regard to the respective system state is as follows:

$$\begin{aligned} non-overloaded &: deadline > criticality > ID \\ overloaded &: criticality > deadline > ID \end{aligned}$$

The reordering of the processing list is performed in two cases:

- after new messages were copied onto a send-queue
- after a message was picked from a send-queue

However, in the second case a reordering of the list is only applied if the state of the system changed, such as from *delayed* to *not delayed* or vice versa (see Figure 4.15).

4.6.5 Ring-Buffer

Since send-queues are solely used for storing outgoing messages, we introduced a second storage area for both, incoming as well as outgoing messages. This second storage area is a ring-buffer that is based on the circular-buffer component of the Boost third party real-time library. Whenever messages are read from the CAN bus, they are stored in the ring-buffer. Outgoing messages that were forwarded to the CAN bus are also copied

[8] `http://www.cplusplus.com/reference/list/list/sort/`

onto the ring-buffer. From a connection's perspective, there is no difference to the status quo system, it still thinks that it is directly attached to the CAN bus. However, due to our approach it is a kind of virtualization of the CAN bus, which brings the advantage that connections on the same V&V system can also exchange messages.

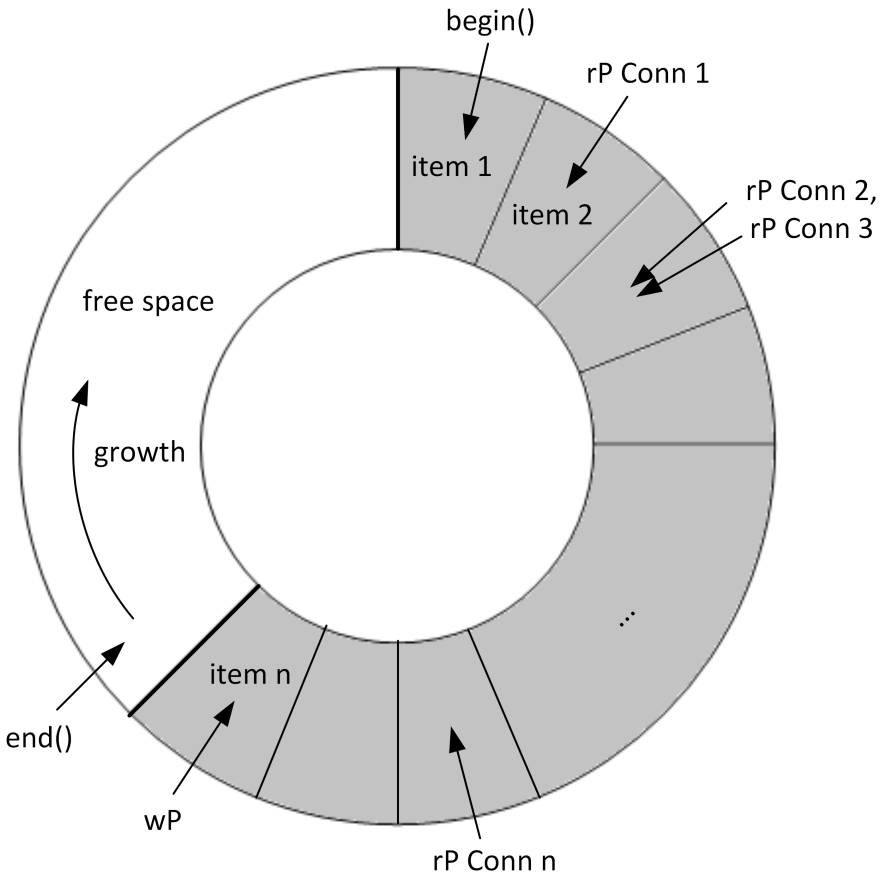


Figure 4.23: Ring-buffer that is based on the Boost circular-buffer. For each connection a read pointer (rP) indicates the last slot that was read by the respective connection. The single write pointer (wP) indicates the last slot that was written to.

One single write pointer (wP) is used to indicate the last slot of the ring-buffer that was written to, which is illustrated in Figure 4.23. Furthermore,

for each connection a read pointer (rP) indicate the slot that was last read by the respective connection. Hence, whenever a connection performs the *read* operation, firstly new messages from the CAN bus are stored in the ring-buffer, and secondly all messages, beginning from the read pointer of the connection to the sole writer pointer of the ring-buffer, are returned to the connection. As an additional feature, we implemented a message filter that filters out messages that are relevant for this connections. This filtering is based on the CAN ID and can be defined for a single ID or for range of IDs. The main advantage of this storage concept is, that it can not overflow since it is a closed circle. However, this also entails the disadvantage that when no free slots are available anymore, this storage design begins to overwrite itself, beginning with the very first slot. This means that a data loss is possible if the size of the ring-buffer is small or connections do not read their messages regularly. Nevertheless, since our demonstrator is designed to run on the V&V system, which is based on a regular x86 architecture, memory space is not really an issues so that the size of the ring-buffer can be defined generously.

In theory, the ring-buffer is also placed in the shared memory so that not only the Connectivity Manager, but also the Connectivity Interface can access the messages and copy them directly into the passed buffer of the application. However, in practice the ring-buffer needs a custom allocator to be created within the shared memory, which was out of scope of this thesis.

4.6.6 Further Implementation Challenges

In this section we want to outline further challenges we faced during the implementation of our demonstrator.

Poor Clock Resolution (Drifts)

In Figure 4.24, the problematic of poor clock resolution is illustrated. The black lines on the timeline represent the boundaries of the periods. The red lines indicate a theoretical write initiation by the application, that can

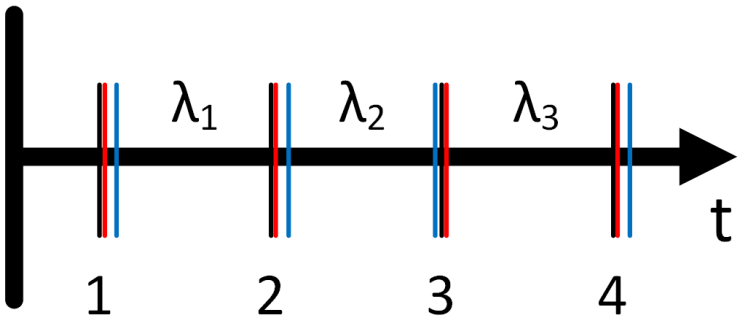


Figure 4.24: Initiation (creation-time) falls into the wrong period due to drifts caused by poor clock resolution. Red lines indicate the optimal initiation instant, while blue lines denote the actual initiation instant.

be considered as optimal since it occurs on a regular basis. However, in practice, due to the resolution of the clock our system makes use of, the initiation of the write operation occurs irregularly. As it can be seen when focusing on the blue line, that represents the actual instants of the write initiation, that the intervals between the specific blue lines is not consistent. Considering Figure 4.24, in period 1 the actual write attempt (blue line) is initiated close to the optimal write instant (red line), however it can be seen as a bit delayed since it is not exactly at the same instant. The same situation appears in period 2, where the actual write operation is a bit delayed in comparison to the optimal instant. Due to this delay the system tries to compensate this lateness by initiating the write operation earlier as previously. Hence, the write operation which is dedicated to be in period 3, is already initiated in period 2. Consequently, the system assumes that this write operation belongs to period 2. However, the processing time for these messages is not sufficient enough since the deadline (end of period 2) is very close. Thus, the system recognises delays, although the bandwidth capacity was not modified.

Although we tried different time and sleep functions from several libraries, including Boost, ARTE and native C++, the result remained the same. Due to a poor clock resolution, the system was inaccurate from a timing perspective, with the consequence that delays were detected which were not caused by a network bottleneck. For this reason, we applied a work-around that

artificially pushes the initiation of the write operation to the middle of the period, so that minor irregularities of the intervals are balanced and the initiation occurs in the correct period. However, this work-around is only sufficient for connections with a cycle-time greater than 3 ms. For shorter cycle-times the drifts are too great so that initiations might fall into the wrong periods again.

Priority Inversion and Data Race

Whenever multiple data streams are being multiplexed, priority inversion might occur. However, in our implementation the priority inversion of messages is limited since its only occurrence might happen in the memory stack of the CAN driver, which is out of scope for this thesis and thus for this implementation. Since the procedure of picking and forwarding one message after another is synchronized with the help of a semaphore, it can be seen as an atomic process. Hence, this guarantees, that the message with the highest priority at this moment, is forwarded to the CAN driver and no message priority shifting can occur once a message was picked.

4.7 Limitations

Due to the limited amount of time to deliver a functioning demonstrator, we focused on the most important functionality and accepted disadvantages in terms of limitations and optimizations.

- In the final release of the demonstrator the ring-buffer was not included in the shared memory since this would have required a custom allocator. The implementation of the custom allocator was not a trivial task and was therefore neglected due to time issues. However, the read functionally was completely implemented and tested, since in scenarios with only one node, the ring-buffer does not necessarily be allocated in the shared memory.
- A minor limitation of our implementation is, that the Connectivity Manager process (ConnMgr.rta) must run within the same INtime

instance, and therefore on the same node, as the A2te.CoreSvr.rta, which is the process of the AVL ARTE framework.

- The final release of our demonstrator used the CAN driver in blocking mode, which means that the whole processing thread of the Connectivity Manager is blocked until the driver returns. This becomes an issues when the driver does not accept messages anymore due to a network bottleneck. Due to the blocking of the driver, also the processing thread stands still and outgoing messages must be queued.

5 Evaluation and Analysis

This section is dedicated to the evaluation as well as the analysis of the demonstrator. Moreover, we outline the hardware and software specification of our test environment and expose the conducted test cases. Consequently, we discuss the impact of the demonstrator with regard to memory and CPU consumption. Finally, a comparison to other approaches is made and the real-world applicability of our demonstrator is shortly reviewed. Since the major research contribution of this thesis is the dynamic priority communication scheduling, also the evaluation and analysis section focuses mainly on the demonstrator sending messages rather than reading messages.

5.1 Hardware and Software Specification

For reasons of transparency and reconstructability of the following evaluation and analysis, we want to outline the hardware and software specification of the test environment of our V&V system first.

5.1.1 Hardware

As a hardware platform, we used a regular Windows workstation and installed INtime 4.20 (32-bit) alongside Windows 7 (64-bit). Due to INtime software licence limitations, our demonstrator system consisted of two cores only. Consequently, two cores were occupied by the two independent INtime instances A and B, while all remaining cores were left to Windows. Each INtime node owns the following hardware:

- **CPU**: Intel(R) Xeon(R) CPU E5607 @ 2.27 GHz

- **INtime Kernel Memory**: 320 MB
- **Process Memory Pool**: 48 MB
- **Memory Allocation**: Non-paged Windows Pool
- **Kernel Clock Rate**: 50 μs

Additionally, the primary core has also access to a Janz-Tec *CAN-PCI board*[1] with two isolated, non-intelligent, low cost CAN controller. The two CAN 2.0 b interfaces with SJA1000 CAN controllers are compatible to ISO/DIS 11898-2. Furthermore, the CAN controllers operate at a maximum bus clock frequency of 33 MHz. In favour of testing our demonstrator in an isolated manner, we attached an additional external device to the very same communications network. This additional device is a Windows 7 (64-bit) Notebook that was connected to the CAN bus via a PEAK-System PCAN-USB adapter.

5.1.2 Software

On the Notebook, the software PCAN-View was installed for performing both, listening passively on the CAN bus while capturing the whole traffic, and also for transmitting self-defined CAN messages onto the CAN bus.

Further software, that we used to analyse the traffic on the CAN bus and hence verify the performance of our demonstrator, was the open source application *Busmaster* developed by a collaboration between Robert Bosch Engineering and Business Solutions Private Limited and ETAS. Moreover, we used the software *CANalyser* by Vector to obtain the highest possible accurate results.

To analyse the RT kernel CPU utilization of our demonstrator, we used the Windows-specific *Performance Monitor* that can be attached to the INtime kernels. For memory observations the INtime-specific resource monitoring tool was used.

A custom script came handy when observing the behaviour of our system over a long period of time and even over night. The custom script

[1] `https://www.janztec.com/produkte/embedded-computing/cancanopen/can-pcil/`

took screenshots at pre-defined intervals, so that no manual interaction was required. Those screenshots were used to trace the operating of the demonstrator. Moreover, the custom script gave an indication of the number of sent messages between screen-shots.

The implementation of the rather small-scale scenario is sufficient to demonstrate the multi-core ability, regardless of how many cores[2] are available in total.

5.2 Usage of the Demonstrator

Basically, the demonstrator software consists of two INtime applications, the ConnectivityManager.rta and the ConnectivityInterface.rta. RTA stands for Real-Time Application and is the file extension for real-time processes within the INtime RTOS. The Connectivity Manager application is started first, since it is responsible for creating and initializing memory and locking resources. After a successful start-up, the INtime process ConnectivityManager.rta is listed as *running* within the process directory of the INtime-Explorer of the respective INtime node. Additionally, the INtime process maps itself to a Windows console dialogue to enable keyboard input. In Table 5.1 the input parameters, that are gathered by the Connectivity Manager, can be seen.

After defining the initial parameters, the Connectivity Manager is ready to be contacted by the Connectivity Interfaces. As already mentioned earlier, the *ID*, the Connectivity Manager tracks, is an internal ID and not the CAN message ID[3]. Continuing, a Connectivity Interface application is ready to be lunched either on the same node or on a different one. Regardless on which node the ConnectivityInterface.rta process is started, a second Windows console dialogue is opened and the process is listed as *running* within the INtime-Explorer of the respective node. Furthermore, also the Connectivity Interface performs a start-up routine, in which among other things it searches for a ConnectivityManager.rta process throughout all nodes. In case

[2]In the following sections the terms core and node can be used interchangeably.
[3]The CAN message ID is specified in advance outside of our system.

Table 5.1: Input parameters for the Connectivity Manager.

Input	Description	Type/Unit
First ID	Defines the first internal ID that is available to a connection.	uint [1..99]
VSEG handle	Specifies the handle to acquire the total size of the virtual segment of the INtime node. If the entered RThandle turns out to be invalid, the VSEG size can be specified manually (see next row).	RThandle
VSEG size	Manual definition of the total size of the virtual segment.	MByte
Shared memory size	Defines the total size of the shared memory.	KByte
Ring-buffer size	Defines the maximum number of elements (CAN-messages) the ring-buffer can hold before overwriting.	uint

of a successful retrieval, the Connectivity Interface is ready for keyboard input. The input, the Connectivity Interface asks for, can be seen in Table 5.2. In Figure 5.1 the RT I/O Console can be seen with the aforementioned keyboard inputs. It is also observable that the Connectivity Manager was started on Node B, since it is listed in the process directory. With focus on the lower part of the illustration, it can further be seen that the CAN bus was successfully initialized with 500 kbit/s and that the first connection was correctly established with ID=1 and memory offset=10, which was both assigned by the Connectivity Manager. Any further proceeding is done via keyboard input, for example to start a connection the ID of the connection needs to be entered. It is also possible to start all connections simultaneously. The test-run ends automatically, after the specified test duration. Once the test-run ended, important information, such as possible deadline violations, delays, remaining messages and if any messages were queued, is displayed in the console window of the Connectivity Manager.

Table 5.2: Input parameters for Connectivity Interface.

Input	Purpose	Unit
Test duration	Defines the duration of the test run.	seconds
Bit rate	Specifies the bit rate that should be used to initialize the CAN bus).	kbit/s
Additional Time	Defines how much time should be added to the start of each period to avoid drifting. This parameter was mainly used for debugging.	microseconds
Number of connections	Specifies how many data streams/connections are being created within this test run.	uint
Criticality	Defines the criticality of this data stream.	uint
Data volume	Defines the data volume of this data stream.	Byte
Cycle-time	Defines the cycle-time of this data stream.	milliseconds

In Figure 5.2 these important information can be seen. Since no delays occurred, no further information about deadline violations or message queuing is shown. It is also noticeable, that the Connectivity Manager was started on Node A, as it can be seen in the top section of this figure. That implies, that the communication in this scenario was performed among multiple cores.

5.3 Results

Based on our major goals, stated in Section 4.3.1, and with reference to our model definition and requirements 4.2.1, we focused on proofing the proper functioning of our concept. In this Results section we outline the

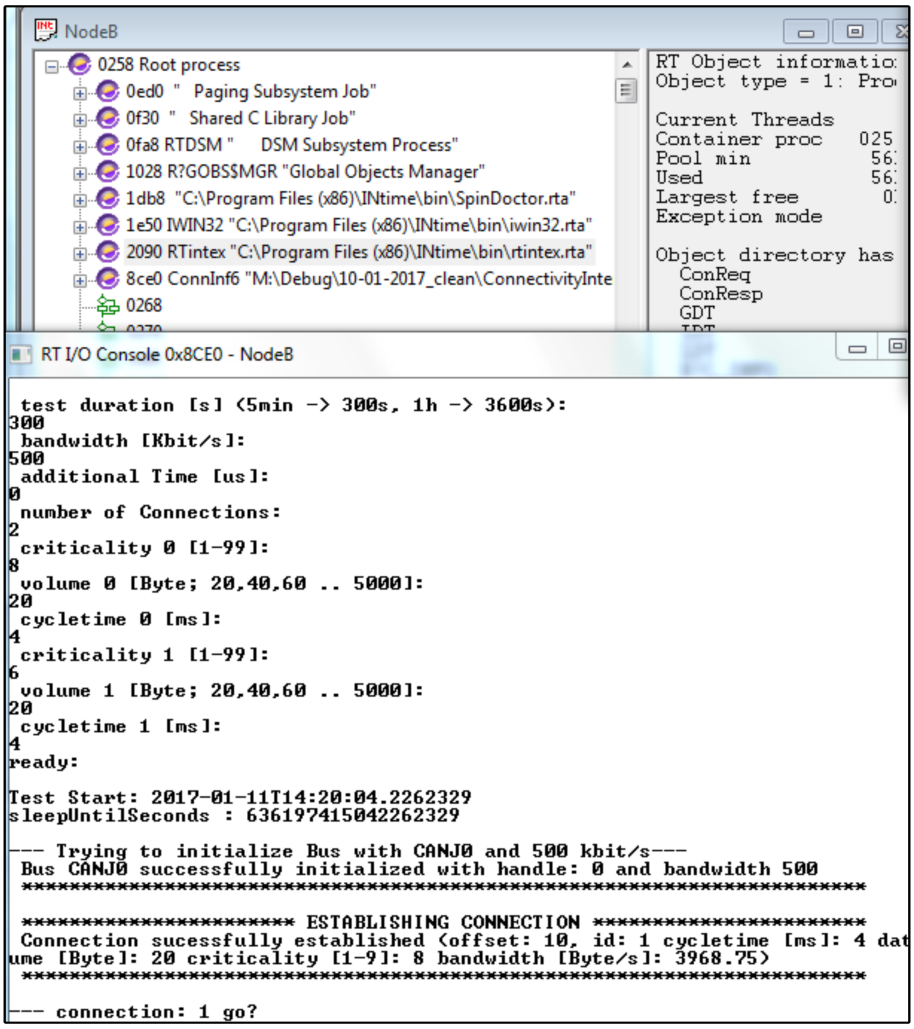


Figure 5.1: Connectivity Interface on Node B - input parameters and connection establishing.

findings of our demonstrator after being tested on our defined test cases (see Section 5.3.1) and asserted on maximum CAN utilization, long term capability, minimum cycle-time, and dynamic priority schedulability.

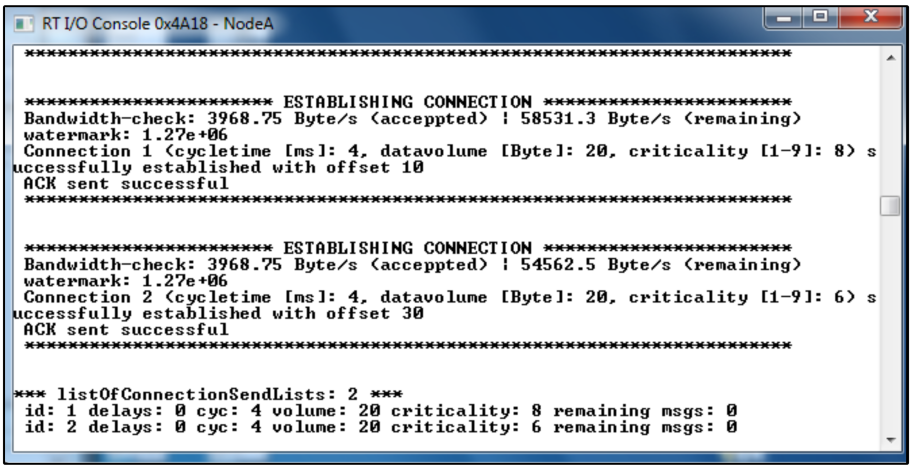


Figure 5.2: Connectivity Manager on Node A - important information of test-run.

5.3.1 Test Cases

We defined the following test cases, that each release of our demonstrator had to pass successfully. To assure the validity of our results we rerun most of our test cases with varying bit rates, such as 50, 100, 250 and 500 kbit/s.

- The number of sent messages per minute must correspond to our Excel formula (Section 5.3.1) and CANalyser measurements.
- The total number of sent messages must correspond to our Excel formula.
- The total number of sent messages of two independent data streams on the same node must be equal.
- The total number of sent messages of two independent data streams on different nodes must be equal.
- Up to 12 connections can be processed simultaneously.
- The minimum cycle-time of 4 ms can be assured without causing delays.
- A Bandwidth utilization of 98% can be reached without causing delays.
- Proper functioning of the dynamic priority scheduling, regardless to which bit rate the CAN bus was initialized.
- The system is capable of persisting in long term test-runs.

- Connections are allowed to differ in terms of data volume without compromising the system.

The most influential test cases are outlined and discussed in the upcoming sections.

Test Duration

With respect to the duration of our test-runs, it is important to note, that the actual duration is always longer than its pre-defined duration. Even with different approaches and programmatic functions, the deviation could not be eliminated. This effect is caused by clock inaccuracies which was already discussed in Section 4.6.6. At a duration of 300 seconds the deviation is approximately 1 second. Furthermore, also the CAN bus initialization, which is initiated by the very first connection, consumes approximately 2 seconds of the test-run duration. Since connections might send up to 2000 messages per second, every additional second might bias the results. Thus, we considered all that aspects in the upcoming evaluation and analysis of our demonstrator.

Excel Formula

To verify the results, we compare the values that were measured in practice to the pre-calculated values. With the help of the software MS Excel we created formulas to calculate the following:

- Accurate bandwidth demand of multiple connections.
- Accurate bandwidth demand of additional traffic.
- Messages per second.
- Nominal count of sent messages.

Formula 3 *In consideration of the input parameters cycle-time λ_n, data volume V_n, and an estimation of the size of a CAN message (msgSize), an accurate bandwidth demand (B) can be determined with the following formula:*

$$msgSize[byte] = \frac{111 + stuffBits}{8} \tag{5.1}$$

$$B[byte/s] = \frac{V_n[byte]}{20} * msgSize[byte] * \frac{1000}{\lambda_n[ms]} \tag{5.2}$$

Formula 5.1 is used to calculate the size of a CAN message, whereas 111 bit[4] are assumed as basis of a 8 data byte message with standard ID format. Since we are certain about the used CAN ID as well as the data being sent, we can accurately determine the number of stuff bits. According to the calculation in Section 2.3.4, we estimated 2 stuff bits for all of our test cases. To receive the message size in byte, the total number of bits is divided by 8, with the result of *14.125* byte per message.

The message size (*msgSize*) is then used in Formula 5.2, to calculate the accurate bandwidth demand (B). Noteworthy, this calculation is different from the worst-case bandwidth estimation in Section 4.5.2, since the accurate message length is used rather than worst-case values. At the first fraction, the data volume is divided by 20, because the software treats messages as 20 bytes large objects. Meaning that the software splits up the outgoing data into 20-byte objects, regardless of the actual size or content. Notably, the data that the connection wants to transfer consists already of completed messages, including all header information. Therefore, it is necessary that the defined data volume is a multiple of 20, since any remainder is cropped. After determination of how many messages the data volume yields, we multiply that number with the pre-calculated size of a CAN message (14.125 byte). The second fraction considers the cycle-time, which is always in milliseconds, and converts the result to a more readable unit of byte per second. To determine the bandwidth utilization, the bandwidth is put in relation to the bit rate of the CAN bus. The value of the data volume is naturally capped by the bit rate of the network and the performed bandwidth-check. This is further explained by Example 3.

Example 3 *In this example the bandwidth utilization of a 250 kbit/s CAN bus is calculated with the help of Formula 5.2.*

$$B[byte/s] = \frac{220}{20} * 14.125 * \frac{1000}{5} \textbf{= 31075 [byte/s]}$$

[4]According to `http://www.esacademy.com/en/library/calculators/can-best-and-worst-case-calculator.html`, a 8 data byte standard CAN message consists of 111 bit, stuff bits not included.

$$utilization[\%] = \frac{B[byte/s] * 100}{B_{NET}[byte/s]} = \frac{31075 * 100}{31250} \textbf{= 99.44 [\%]}$$

As it can be seen in Example 3, the data volume of a connection with a cycle-time of 5 milliseconds, is naturally limited to 220 bytes, assuming that the CAN bus is operating at 250 kbit/s. The value of 31250 bytes per second is the converted bit rate of 250 kbit/s.

Formula 4 and 5 outline how we calculate the number of *messages per second* and the *nominal count* of sent messages.

Formula 4 *In consideration of the input parameters cycle-time λ_n, data volume V_n, and an estimation of the size of a CAN message (msgSize), the number of messages per second (mps) can be determined with the following formula:*

$$mps = \frac{V_n[byte]}{20} * \frac{1000}{\lambda_n[ms]} \tag{5.3}$$

Formula 5 *To determine the nominal count of sent messages (nc) the number of messages per second (mps) is multiplied by the number of seconds the test-run lasts (duration).*

$$nc = mps * duration \tag{5.4}$$

5.3.2 Dynamic Priority Communication Scheduling

One of the main criteria of our demonstrator was the proper functioning of the dynamic priority scheduling of data streams. To prove this requirement, we defined the following test scenario as outlined in Table 5.3. The test scenario consists of 6 connections (data streams) that are evenly distributed between the two INtime nodes to simulate a real world test environment. Each connection has the same data volume[5] V_n as well as the same cycle-time λ_n, which leads to the same amount of messages per second and an accumulated bandwidth load of *84.75%* of the CAN bus, assuming that $B_{\mathrm{CM}} = B_{\mathrm{NET}} = 500$ kbit/s. These 6 data streams solely differ in terms of

[5]The standard 11-bit CAN ID format is used.

Table 5.3: Test Scenario - 6 connections.

	S_1	S_2	S_3	S_4	S_5	S_6
INtime Node	A	B	A	B	A	B
V_n **[Byte]**	500					
λ_n **[ms]**	40					
C_n	4	6	8	2	7	5
Messages/s	625					
Bandwidth [Byte/s]	8828.12					
Bandwidth usage [%]	14.12					

criticality C_n, which is arbitrarily assigned. A high criticality value represents a greater importance, such as Connection 3. While the first four rows of Table 5.3 represent pre-defined settings, the values of the other three rows can be calculated. Since we precisely know about the data being sent, we are able to calculate the accurate bandwidth, including protocol overhead and stuff bits. The overall duration of this test-run was 31 minutes, while every 60 seconds the number of successfully transmitted CAN messages was captured by our custom script. For the capturing we used the additional external CAN hardware[6] which was attached to the same CAN bus via USB adapter. The emulation of an additional node on the shared bus was also achieved by the use of this PCAN-USB hardware. Moreover, with the help of the CAN bus monitoring tool, PCAN-View, we were able to independently analyse the CAN traffic via an external device.

In this scenario all connections were started at the same time. For a better understanding, Figure 5.3 can be divided into seven stages.

Table 5.4: Additional load.

	Bandwidth [byte/s]	**Bandwidth [%]**
single	9500	15.20
double	19000	30.40
triple	28500	45.60

[6]PCAN-USB by PEAK System.

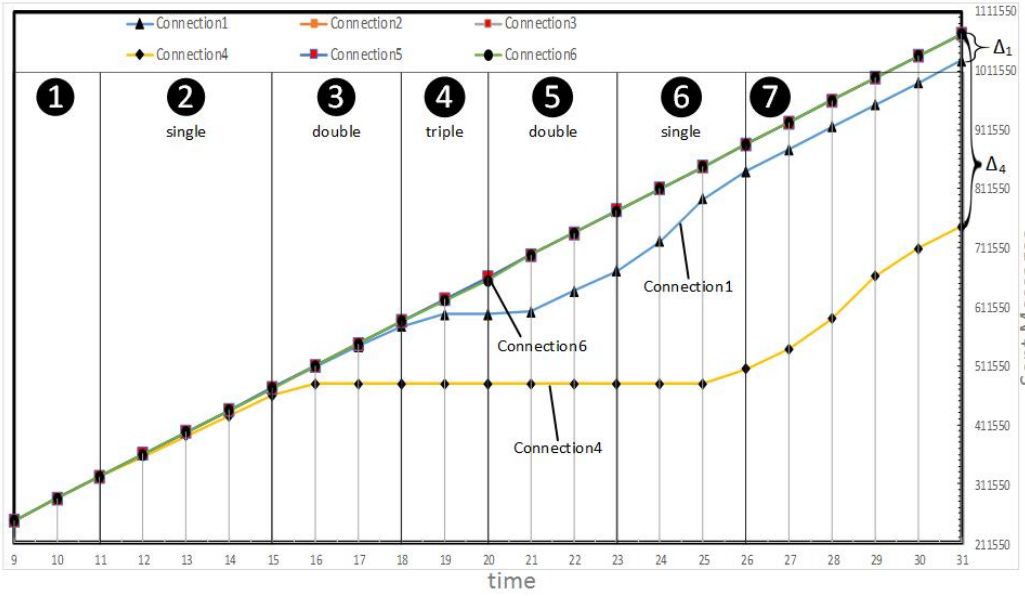


Figure 5.3: Graph of the prioritization example. 6 equal connections demand the CAN bus in the same volume. However, the three least-critical connections are degraded once the CAN bus is demanded by single, double, and triple additional load.

(1) **0′-11′**: Until minute 11, the Connectivity Manager was the only node using the CAN bus by sending data of the 6 connections. Meaning that the system was neither overloaded nor delays occurred and thus a normal operation, in the sense of EDF scheduling, was given.

(2) **11′-15′**: At 11′ we started to simulate a bottleneck by generating additional load on the shared bus via the external PCAN-View monitoring tool. Consequently, from 11′ until 15′ one additional data stream, with the characteristic of 1 CAN message in the extended 29-bit CAN ID format every 2 milliseconds, is also using the shared bus. As it can be seen in Table 5.4 the *single* additional load demands as much bandwidth that the CAN bus utilization rises up to *99.95%* (84.75% of CM + 15.20% of single additional load). This high utilization already causes delays within our system. Notably, the single additional load consumes more bandwidth than one of the registered CM connections (15.20% > 14.12%). As a result, exactly one CM connection, precisely the connection with the lowest criticality value (Connection 4), is *de-*

graded and gets delayed. At the very moment when any connection violates a deadline for the first time, the scheduler switches from EDF to best-effort principle as a delay occurred. Consequently, messages of Connection 4 are not being processed as regularly as those of higher critical connections, which can be seen in Figure 5.3, where the graph of Connection 4 is slightly decreasing its slope in comparison to the other graphs. Therefore, transmission is delayed and messages of Connection 4 are being queued constantly. However, each connection queue can only store messages until a certain threshold (see Section 4.6.3) is reached. In such an event, messages are no longer accepted from this specific connection, leading to data loss.

(3) **15′-18′**: In this stage, the *double* amount of additional load was generated, resulting in a bandwidth demand of *30.40%*. Altogether a theoretical bandwidth utilization of 115.15% is demanded. The consequences are twofold, on the one side the additional bandwidth demand is large enough that messages of Connection 4 are not longer sent but only queued. This queue, however, reaches its threshold at 16′, which means that new messages from Connection 4 are no longer accepted and therefore lost. Δ_4 denotes the difference between nominal and actual count sent messages of Connection 4. On the other side, also Connection 1, the second lowest critical connection, is being delayed due to the overloaded system condition.

(4) **18′-20′**: Stage 4 causes the same effects as stage 3. Tripling the additional load puts the CM system in such an overloaded condition that three connections get delayed. Since the *triple* additional load affects three CM connections, also Connection 6, in addition to Connection 4 and Connection 1, is delayed. Moreover, messages of Connection 1 are also neither sent nor accepted anymore, which leads to further message loss indicated by Δ_1.

(5) **20′-23′**: In this stage, the additional load is reduced back to *double*, so that only two CM connections are affected. For this reason, Connection 6 recovers from its bandwidth violation, and can catch up on the non-delayed connections due to the best-effort approach. Afterwards, also messages of Connection 1 get processed again and new messages are accepted. Furthermore, it tries to catch up, which can be observed by the steeper slope. However, the additional load is still too demanding so that the original slope cannot be restored yet.

(6) **$23'$-$26'$**: Stage 6 aims to reduce the additional load back to *single* and enables Connection 1 to catch up with the non-delayed connections. Since messages of Connection 1 were rejected in stage 4, the gap Δ_1 remains. The slope of the graph of Connection 1 can be restored and also messages of Connection 4 are accepted and processed again.

(7) **$26'$-$31'$**: In this stage, all additional load is removed from the bus and the CM system is again the only active node. Thus, the entire CAN bus bandwidth can be used so that the least critical and longest delayed Connection 4 can catch up. At $29'$ the delay has been made up and the scheduler switches from best-effort back to EDF principle. However, also on this connection, rejected messages cannot be made up so that Δ_4 and also Δ_1 remain.

Table 5.5: Test results.

	S_1	S_2	S_3	S_4	S_5	S_6
C_n	4	6	8	2	7	5
nominal count (nc)	1074175					
total msg loss Δ	45444	0	0	326775	0	0
actual count	1028731	nc	nc	747400	nc	nc
delayed msgs	351669	381	0	359494	0	93443
%	36.96	0.03	0	63.88	0	8.69

Considering the values in Table 5.5, it can be clearly seen, that higher critical data streams are favoured over less critical data streams in the event of changing bandwidths on the shared communication network. Moreover, this test scenario demonstrates how our scheduling approach adapts to overload conditions and seeks for a maximum network utilization to catch up on delays. It should be noted at this point, that % in Table 5.5 represents the percentage of messages that were either lost or delayed in respect to the nominal count.

5.3.3 Maximum CAN Utilization

Another test scenario aimed to find the maximal bandwidth utilization without overloading the system nor causing delays. A bandwidth utilization of 98% was reached before any message of the 4 concurrent data streams was delayed. The accurate value of 98.39% was determined by slightly increasing the external additional load. The test scenario was as follows:

Table 5.6: Maximum CAN utilization - data stream definition.

	ID length [bit]	λ_n **[ms]**	V_n **[byte]**	**mps**	**byte/s**	**%**
S_1	11	4	60	750	10593.75	16.95
S_2						
S_3						
S_4						
sum S_n				**3000**	**42375**	**67.8**
S_a	29	1	20	1000	18750	30
total	**-**	**-**	**-**	**4000**	**61125**	**97.8**

As it can be seen in Table 5.6, the maximum utilization scenario consists of 4 connections from the CM system, and one additional data stream (S_a) from the external device. The 4 CM connections ($S_1 - S_4$) use the standard CAN ID format of 11-bit and come with an equal cycle-time (λ_n) and data volume (V_n). Due to this connection parameters, the values for messages per second (mps), bandwidth demand (byte/s), and percentage of bandwidth demand (%) can be calculated. For the bandwidth calculation we used our customized Excel formula which was already discussed in Section 5.3.1. This brings us to an intermediate amount of 3000 messages per second and a bandwidth demand of 67.8%, given that the CAN bus was initialized to a bit rate of 500 kbit/s. Furthermore, we used additional load (S_a) from an external device to push the CAN bus utilization as high as possible without overloading the system. The additional data stream uses the extended CAN ID format of 29-bit and comes with a data volume of 20 byte every millisecond. This results in 1000 messages per second and a bandwidth demand of 30%. According to our calculation, a total of 4000 messages per

second and a bandwidth utilization of 97.8% could be reached before any delays occurred.

CAN Channel: CAN 2

Statistic	Current / Last	Min	Max	Avg
Busload [%]	98.39	67.93	98.46	69.90
Min. Send Dist. [ms]	0.000	n/a	n/a	n/a
Burst Time [ms]	0.754	0.528	2.352	0.764
Bursts [total]	174147	n/a	n/a	n/a
Frames per Burst	3	2	9	3
Std. Data [fr/s]	2999	434	3009	2976
Std. Data [total]	9612424	n/a	n/a	n/a
Ext. Data [fr/s]	1000	0	1001	56
Ext. Data [total]	181551	n/a	n/a	n/a

Figure 5.4: Maximum CAN utilization.

Our results are quite congruent in comparison to the live measurement that was captured with the CANalyser software and is outlined in Figure 5.4. The CANalyser detected a maximum bandwidth utilization (busload) of 98.46%, which is close to our calculation of 97.8%. Moreover, it measured 3000 messages per second (fr/s) of standard CAN messages and 1000 messages per second (fr/s) of extended CAN messages. Notably, even the calculated bandwidth utilization of 67.8% without the additional load corresponds to the measured minimum busload of the CANalyser (67.93%). This is possible, due to the fact, that the additional load was engaged when the measurement was already in progress.

The results of this test case prove the correct timeliness as well as the proper functioning of our demonstrator. Since the additional load demands almost the whole bandwidth and the remaining bandwidth capacity is approximately 1.5%, any skip of a period or lateness of the demonstrator would result in delays. This test case proves further, that the theoretical prediction can be assured by the practical measurement.

5.3.4 Long Term

In a different test scenario the *long-term* stability of the demonstrator was verified. This long-term test-run consisted of *12 equal connections* that constantly used up 67.80% of the 500 kbit/s CAN bus. To simulate also other nodes on the bus, a minor additional traffic was applied as well. At an overall bus utilization of 77.08% more than *952 million CAN messages* were successfully transmitted within **88 hours**. For the whole time the system persisted stable without any noteworthy delays. Only two CAN messages were delayed due to other reasons. This test-run demonstrates the real world applicability of our system, as in live systems the CAN bus is usually limited to a utilization of 40% to avoid bottlenecks [Davis et al., 2007].

The resulting output of this test-run can be seen in Figure 5.5a. Particularly, the two delayed messages, which did not originate from the least critical connection but from a random one, can be observed. The three top lines of Figure 5.5a indicate a warning that a delay might occur. This warning is issued whenever the copying of the new messages onto the send-queue just finished to close to the respective deadline of this period. In particular, when the remaining time to the deadline is smaller than 1% of the cycle-time.

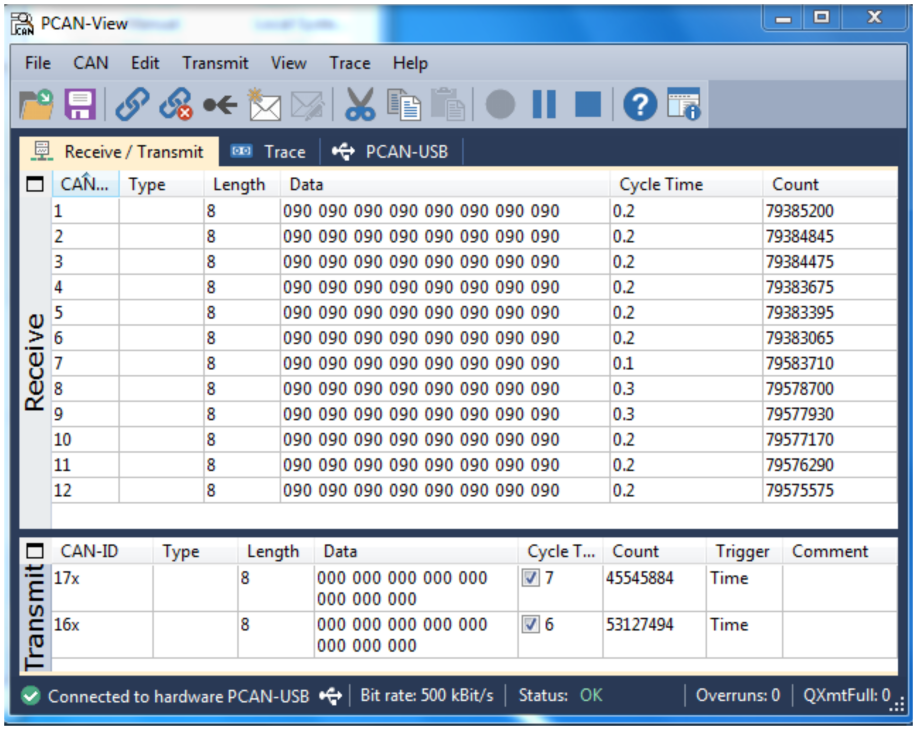
ID: 2		x	y	z
creation-time C:	63618838947	740	634	9
deadline D:	63618838947	740	752	4
ID: 8		x	y	z
creation-time C:	63618838947	740	635	6
deadline D:	63618838947	740	698	4

In the paragraph above, z indicates the 100 nano seconds significance, y labels microseconds, and x represents milliseconds. As it can be seen, the creation-time of connection ID=2 is very close to its deadline, only 117500 nano seconds away. The same applies to connection ID=8, however the remaining time is even less. Notably, the creation-time of ID=2 is fairly close to ID=8, and also the deadline is not far apart. That let us conclude that the

```
id: 10 close deadline: 1477 D: 636187083824922612 C: 636187083824921135
id: 2 close deadline: 1175 D: 636188389477407524 C: 636188389477406349
id: 8 close deadline: 628 D: 636188389477406984 C: 636188389477406356

id: 1 delays: 0 criticality: 9 remaining msgs: 0
id: 2 delays: 0 criticality: 97 remaining msgs: 0
id: 3 delays: 0 criticality: 83 remaining msgs: 0
id: 4 delays: 2 criticality: 38 remaining msgs: 0
id: 5 delays: 0 criticality: 57 remaining msgs: 0
id: 6 delays: 0 criticality: 75 remaining msgs: 0
id: 7 delays: 0 criticality: 23 remaining msgs: 0
id: 8 delays: 0 criticality: 65 remaining msgs: 0
id: 10 delays: 0 criticality: 34 remaining msgs: 0
id: 11 delays: 0 criticality: 83 remaining msgs: 0
id: 12 delays: 0 criticality: 30 remaining msgs: 0
id: 9 delays: 0 criticality: 69 remaining msgs: 0
```

(a) Summary of the test-run.



(b) CAN messages captured by PCAN-View.

Figure 5.5: Results of the long term test-run over 88 hours. 952 million CAN messages were successfully transmitted.

2 delayed messages are caused by this poorly timed copy process and the following warning in the I/O console, that also slows down our system.

In the *Receive* section of Figure 5.5b the 12 connections of the demonstrator can be seen. Each connection operating at a cycle-time of 20 ms and a data volume of 100 byte. The number of messages in the *Count* column is slightly different due to the fact, that the connections were started sequentially. In the *Transmit* section the additional connections with the cycle-time 6 and 7 can be noticed. Notably, the data section is filled with '0's, resulting in the maximum amount of stuff bits. Moreover, it can be seen that the CAN bus was initialized to 500 kbit/s.

5.3.5 Minimum Cycle-Time

With reference to Section 4.3.2, real-time data, such as wheel speed, needs to be exchanged in very short cycles. Therefore, the minimum cycle-time of the demonstrator, that does not cause delays, should also be as short as possible. Through simulations we reached a minimum cycle-time of **4 milliseconds**, whereas it depends on the data volume if a delay occurs. If the amount of data is rather little, such as 20 byte, the Connectivity Manager is fast enough to process the data before its deadline. However, with a larger data volume the system consumes more time for copying and converting the data into messages, so that deadlines might get violated. Further, every cycle-time smaller than 4 milliseconds is too short, so that the processing time is greater than the cycle-time. This is not an issue of schedulability but of worst-case execution time (WCET), which we focus on in the upcoming Section 5.4.1.

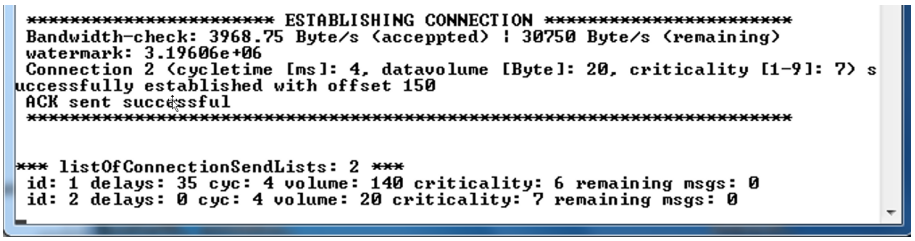


Figure 5.6: Two connections with same cycle-time but different data volume.

Figure 5.6 shows the resulting output of the Connectivity Manager after two connections with the same cycle-time of *4 milliseconds* were run successively for approximately 5 minutes[7]. Meaning that the second connection with ID=2, was only started after the first connection with ID=1 ended. Thus, we assured that the two connections were not influencing each other. Consequently, the criticality of the connections was irrelevant, since there was no other active connection, that could have been degraded by the scheduler, anyway. However, the size of the data volume made the difference if the connection got delayed or not. In this scenario the connection with a data volume of *140 byte* got delayed while the connection with just 20 byte of data volume remained without any delay. The value 35 indicates the number of messages that were delayed, meaning that the processing of those messages did not complete before their deadline. The CAN bus was set to 500 kbit/s in this scenario.

At a cycle-time of *5 milliseconds* we could not find any occurrence of delays regardless of which data volume was used. We even tested our assumption with different bit rate configurations of the network. For example with a set-up of two parallel connections with 5 milliseconds cycle-time and a data volume of 200 byte, we were not able to detect any delayed message on the CAN bus, that was initialized to 250 kbit/s. Notably, a connection with 200 Byte every 5 millisecond generates a bandwidth demand of 90.4% of the prior mentioned bus. Consequently, we are confident to say, that with a cycle-time of 5 milliseconds, every amount of data can be processed without delays. Noteworthy, a further raise of the data volume would have been senseless, since the maximum amount of data is limited by the bit rate of the bus (see Section 5.3.1). Meaning that a connection with a huge bandwidth demand would anyway be rejected by the Connectivity Manager due to the worst case bandwidth estimation.

Furthermore, our demonstrator can also handle connections with a cycle-time smaller than 4 milliseconds, however it cannot be guaranteed that messages will not get delayed. For example in Figure 5.7, connections with 1 millisecond cycle-time originating from different nodes were tested.

[7]The runtime was not exactly 5 minutes long since the connections were started manually via keyboard input and the first connection additionally had to initialize the bus (see Section 5.3.1)

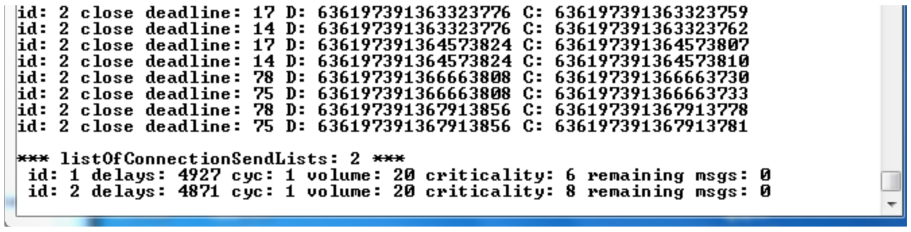


Figure 5.7: Two connections with the same cycle-time of 1 millisecond got delayed to the same extent.

As it can be seen, the connections were run successively, so that the criticality was again irrelevant for this test scenario. Both connections were delayed to the same extent, numerically an average of 1.8% of all sent messages got delayed. In consideration of Table 5.7, the concrete values of the prior

Table 5.7: Comparison of connections from different nodes with a cycle-time of 1 millisecond.

connection	sent messages	delayed messages	delayed %	nominal count	runtime [sec]
1	262521	4927	1.87	294000	294
2	270130	4871	1.80	299000	299

mentioned test scenario can be seen. With special focus to the column "*nominal count*", it is notably that the connections were not able to sent as many messages as they theoretically should. We calculated the nominal count with our Excel formula with respect to the runtime of the respective connection. The slightly different runtime values can be explained so that Connection 1 was in charge of initializing the CAN bus before any message could be transmitted. For both test-runs the duration was set to 5 minutes (300 seconds). Due to the very short cycle-time, the connections were not able to transmit as many messages as expected within the given time. However, the rather small average value of 1.8% of delayed messages let us come to the conclusion, that the connections simply skipped a period when they were highly delayed. This would also explain the difference, which is denoted by Δ_{loss}, between *sent messages* and *nominal count*. For Connection 1 the percentage of Δ_{loss} is 10.71% and for Connection 2 it is 9.66%.

The values outlined above indicate, that the performance of both connections

is alike, regardless on which node the connection was started. Furthermore, this implies that the inter-core communication was implemented efficiently, since the Connectivity Manager is always hosted on the other node for one of the two connections. Obviously this longer communication flow does not influence the processing of the connections. Further test cases with cycle-times of 2 and 3 milliseconds show similar results, whereas the Δ_{loss} decreased the higher the cycle-time was defined.

Concluding, the minimum cycle-time, that does not necessarily cause delays, is 4 milliseconds. Smaller cycle-times can also be handled, since only a minor percentage of sent messages is delayed. However, the data loss Δ_{loss} must not be neglected.

5.4 Analysis

In the upcoming sections a discussion concerning the herein before mentioned worst-case execution time is conducted, followed by an analysis with regard to CPU and memory usage. Concluding, our demonstrator is compared to other approaches as well as assessed in terms of real-world applicability.

5.4.1 Worst-Case Execution Time

Our understanding of the term *worst-case execution time* (WCET) within this thesis is, the time between the instant, in which the connection hands over the message to the Connectivity Interface, and the instant in which the message is picked from the send-queue of the Connectivity Manager and forwarded to the CAN driver. Due to thread scheduling of the INtime RT kerneland mutual exclusion, this time is not solely constant but can result in a varying execution time. An illustration of this definition is given by Figure 5.8.

In other words, the WCET is the maximal time a message requires to enter our system and to end up on the network. For simplification we assume that only one message is intended to be sent. In Figure 5.8, it can be seen how

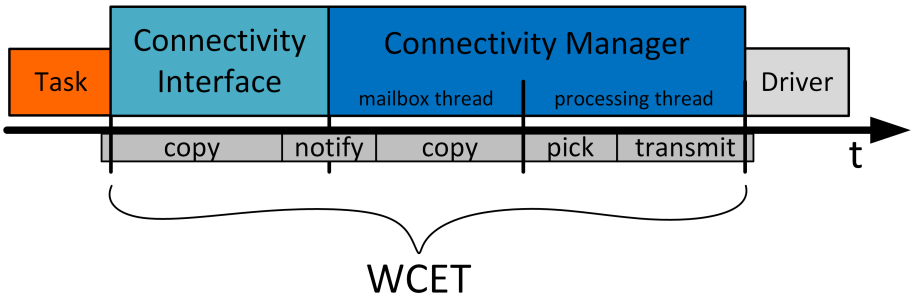
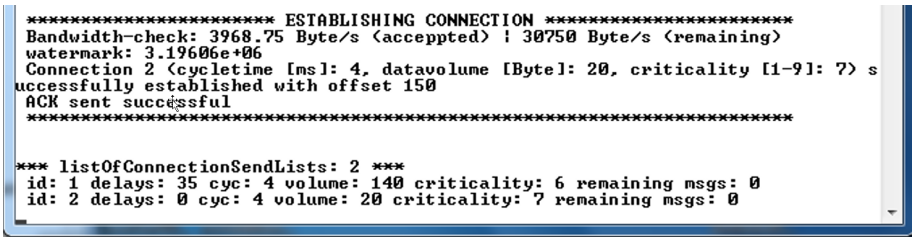


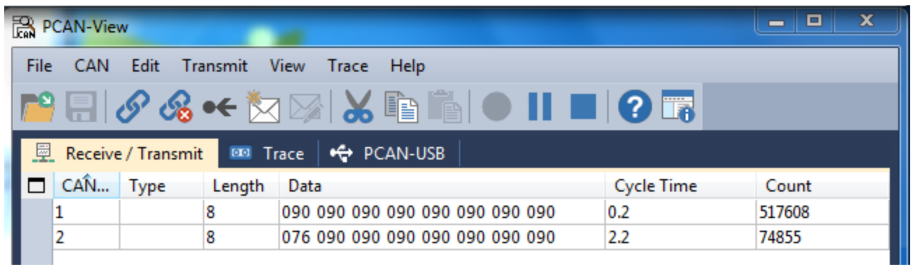
Figure 5.8: Composition of the worst-case execution time.

this maximal time is composed. First, the *Task* initiates the message transfer by contacting the Connectivity Interface. The Connectivity Interface copies the message from the buffer of the *Task* to the shared memory. Afterwards, the Connectivity Manager is notified and takes over the message by copying it from the shared memory to the corresponding send-queue. Once the message approached the send-queue, the processing thread, that has just been woken up, picks the message from the queue and forwards it to the driver of the network. Those five activities, which are illustrated as grey boxes in Figure 5.8, are the main components of the WCET. Each of these activities can be interrupted due to thread scheduling or locking. As mentioned above, this scenario considered only one message. However, the WCET strongly depends on the data volume, since the copy-time increases with the number of messages that shall be sent.

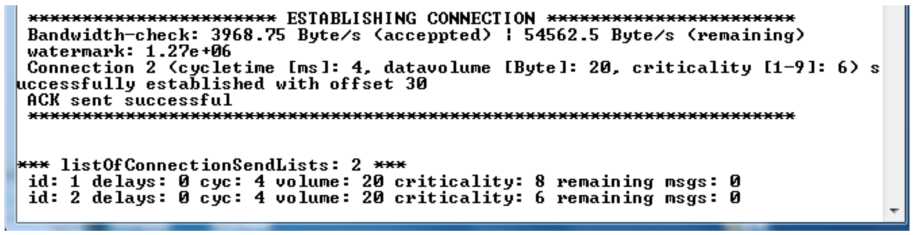
In the analysis of our demonstrator, we figured out that the worst-case execution time of our system is approximately 3 milliseconds. Therefore, as discussed previously, the minimal cycle-time must not be smaller than 4 milliseconds, whereas it also depends on the data volume. Any lower deviation of the minimal cycle-time implies message delays, because the copying of the messages takes to long that the adherence of deadlines cannot be guaranteed. In Figure 5.9 we tested exactly this assumption. Figure 5.9a pictures the same scenario as already discussed in Section 5.3.5. Two connections with equal cycle-time but with different data volume. Both connections have a cycle-time greater than the WCET, however the data volume of the first connection is too large so that the copying of the messages takes too long and messages are being delayed. Nevertheless, 35



(a) Two connections with same cycle-time but different data volume.



(b) Message count via PCAN-View of scenario (a)



(c) Two connections with same cycle-time and equal data volume.

Figure 5.9: Example of worst-case execution time (WCET).

delayed messages out of 517608 sent messages is still a presentable result that can be seen in Figure 5.9b. Notably, the ratio between data volume of Connection 1 and Connection 2 is approximately the same, in particular 7 times, as between sent messages of Connection 1 and Connection 2.

Additionally, the WCET is also in dependency on the effects of drifts and a poor clock resolution, which is both discussed in 4.6.6. The appearance of drifts might shorten the actual available time within a period.

5.4.2 CPU Usage

While our system was tested on the aforementioned test cases, we focused, among other things, on the RT kernel CPU usage. With the help of the Windows specific Performance Monitor, that can be attached to the INtime RTOS to monitor the RT kernel CPU performance, we were able to record the CPU utilization in the following situations:

- **Initialization phase**: Creation and initialization of various resources.
- **Regular operating**: Reading and writing messages while the system is in an non-overloaded state.
- **Message queuing**: Queuing of outgoing messages onto send-queue due to overloaded system state.
- **Deinitialization phase**: Shut-down of resources.

The *initialization phase* includes the following activities:

1. Initialization of the AVL ARTE core framework.
2. The CAN driver CANj.rsl is loaded into the address space of the process.
3. The third party real-time library BOOST is loaded.
4. Own process is registered in the process directory.
5. Ring-buffer, that is based on the BOOST circular-buffer, is created and initialized.
6. Creation and initialization of shared memory, semaphore and mailbox.
7. The time server of the ARTE core framework is initialized and tested.
8. Creation of mailbox and processing thread.

At this point it is important to mention, that the initialization phase as it is described above, only applies for the Connectivity Manager application, since it is the main processing component. The Connectivity Interface simply functions as an interface and is started after the Connectivity Manager. Thus, all required resources have already been created and initialized, so that the Connectivity Interface only needs to refer to them. Solely its own mailbox is created and initialized. Consequently, the initialization phase of the Connectivity Manager is much more CPU consuming than of the Connectivity Interface. This effect can be seen in Figure 5.10, that illustrates a scenario in which both INtime cores were used. The Connectivity Manager

was hosted on *node A* (blue line), while the Connectivity Interface was started on *node B* (red line). The first peak of *node A* indicates the initialization of the AVL ARTE core framework. The second peak is caused by the initialization of the time server and the third peak can be ascribed to the initialization of the CAN bus via can_j driver. Furthermore, it can be seen that the initialization phase of the Connectivity Interface on *node B* (red line) is not as CPU consuming as the initialization phase of the Connectivity Manager. This scenario solely focuses on the start-up of the applications, no data streams were being processed.

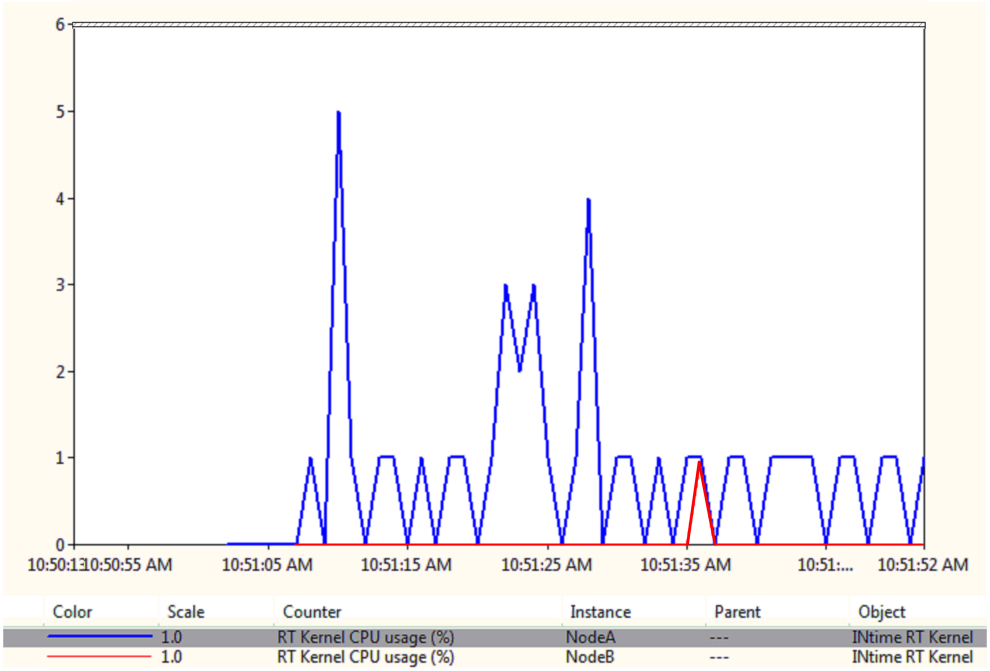


Figure 5.10: Difference of CPU usage of Connectivity Manager on node A (blue line) and Connectivity Interface on node B (red line) while initialization phase.

Furthermore, in terms of CPU usage the *message queuing* situation is not applicable to the Connectivity Interface, since the communication between Connectivity Interface and Connectivity Manager is asynchronous (see Section 4.6.3). Meaning that the Connectivity Interface is unaffected of the overload-state of the Connectivity Manager. In Figure 5.11 all aforementioned phases can be observed, except for the initialization phase that was

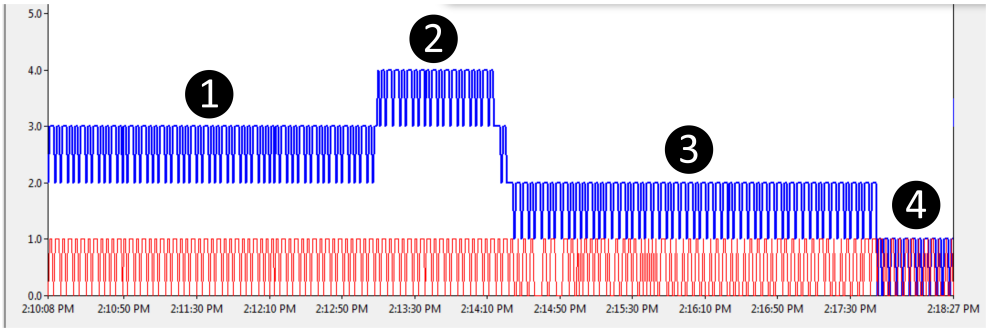


Figure 5.11: Different CPU load phases of the Connectivity Manager (blue line) in comparison to the Connectivity Interface (red line).

discussed previously. This test-run consisted of 6 equal connections that were distinguishable only by their criticality. Moreover, these connections were evenly distributed among two cores. As it can be seen by the steady CPU consumption, **(1)** indicates a regular operating phase at a time where both Connectivity Interfaces, with three actives connections each, were operating. Since *node A* (blue line) hosted not only the Connectivity Manager application but also one of the two Connectivity Interface applications, the RT kernel CPU utilization was at 3% in comparison to 1% of *node B* (red line) that only hosted the second Connectivity Interface application. Due to the generation of additional traffic, the Connectivity Manager was put into an overload-state with the result that the CPU usage rose to 4%, indicated by **(2)**. The raise of the CPU utilization can be affiliated to the additional CPU load that was required to queue all outgoing messages. After approximately one minute, the test-run of the Connectivity Interface application on node A (blue line) ended. Therefore, the Connectivity Manager was able to reach a non-overloaded state again because the queuing of messages was not necessary anymore. Thus, the CPU usage of node A (blue line) decreased and settled down at 2%, since the Connectivity Interface application on the same not node was no longer active but the test-run of the second Connectivity Interface application was still ongoing. This behaviour is indicated by **(3)** in Figure 5.11. Finally, also the Connectivity Interface application on node B ended its test-run, so that the Connectivity Manager had no more connections to process. Consequently, the Connectivity Manager de-initialized itself and the RT kernel CPU utilization decreased to 1%, which is indicated

by **(4)**. Notably, the RT kernel CPU usage of node B (red line) was once again not affected by the state changes of node A (blue line).

Concluding, we observed that the RT kernel CPU usage directly depends on the number of operating connections. The more active connections, the higher is the regular level of the CPU utilization. Additionally, also the state of the Connectivity Manager application is crucial to the CPU load. Meaning that whenever the system is overloaded, more processor resources are required for queuing outgoing messages, which increases the overall CPU utilization of the node. In comparison to a plain INtime node, the RT kernel CPU usage increased from 1% to 8% on the node that hosted the Connectivity Manager, a Connectivity Interface with 12 connections, the CAN driver, and a helper process. Especially, at the initialization phase and in stress-phases where many messages were queued, the CPU usage rose to 8%. However, this minor increase of RT kernel CPU usage can still be seen as an acceptable result.

5.4.3 Memory Usage

While the demonstrator was conducting various test-runs, we observed the memory consumption of both, the Connectivity Manager application and the Connectivity Interface application. With regard to memory consumption, the virtual segment size (VSEG) is of highest relevance. In the settings of the INtime RTOS the virtual segment size for user applications can be defined. For a detailed analysis of the demonstrator while being in the overload-state, we assigned a large VSEG size of 32 MB. This amount of memory is enough to queue outgoing messages for several minutes which in return gives us enough time to observe the behaviour of the demonstrator.

In Figure 5.12, the memory consumption of the Connectivity Manager process as well as of the Connectivity Interface process can be seen. At that moment, the system was in a regular operating state without additional message queuing of any of the 5 active connections. In a regular operating mode (non-overloaded) the Connectivity Manager process consumes 4 MB of the total virtual segment size of 32 MB. In comparison to that, the Connectivity Interface process uses only 2620 KB while operating. This supports our

Type	Name	PhysMem	VirtTotal	VirtUsed	VirtLargest
Process	Root process	318 M	256 M	56 M	194 M
Process	" Paging Subsystem Job"	32 K			
Process	" Shared C Library Job"	32 K			
Process	RTDSM " DSM Subsystem Process"	263 K			
Process	R?GOBS$MGR "Global Objects Manager"	527 K	4 M	260 K	3832 K
Process	"C:\Program Files (x86)\INtime\Bin\SpinDoctor.rta"	56 K			
Process	IWIN32 "C:\Program Files (x86)\INtime\bin\iwin3...	192 K	4 M	4 K	4092 K
Process	RTintex "C:\Program Files (x86)\INtime\bin\rtinte...	301 K			
Process	A2te.CoreSvr "C:\Program Files (x86)\AVL\Cobra...	768 K	32 M	2144 K	26 M
Process	ConnMgr "M:\Debug\ConnectivityManager.rta"	1485 K	32 M	4 M	27 M
Process	ConnInf7 "M:\Debug\ConnectivityInterface.rta"	457 K	32 M	2620 K	27 M

Figure 5.12: Detailed memory list of INtime real-time processes running on that node.

claim, stated in 5.4.2, that the Connectivity Interface simply acts as an interface and that most resources are being mapped instead of specially created. The number of 4 MB of the Connectivity Manager process is appropriate in consideration that this includes shared memory, ring-buffer, semaphore, mailbox, threads and also send-queues along with outgoing messages. Once the system becomes overloaded, the virtual segment size of the Connectivity Manager process increases drastically since the send-queues are getting more and more filled with outgoing messages. However, to avoid reaching the maximum of 32 MB of total VSEG size and a resulting programmatic stack-overflow, a threshold, called watermark, with a value of 80 % of the total VSEG size is being used. In Section 4.6.3, the watermark concept was already discussed in detail. As long as the threshold is exceeded, no new outgoing messages are being accepted from the Connectivity Manager. Thus, the total VSEG size cannot be reached in practice, which was also verified in long-term test-runs.

5.4.4 Comparison to Other Approaches

To the best of our knowledge, no comparable approaches were proven in practice, so that the performance of our demonstrator cannot be matched. Unfortunately, the remarkable comparison to the status quo cannot be drawn since the prior system cannot be easily adapted to our test environment and results of a live system are not yet available.

6 Outlook and Conclusion

Due to upcoming features, such as automatic parking, traffic sign recognition or advanced driver assistance systems (ADAS), vehicles are becoming increasingly sophisticated. However, with greater sophistication comes greater complexity of those in-vehicle systems, networks, but also automotive test systems. To keep up with the complexity of in-vehicle systems, automotive test systems (V&V systems) need to be enhanced as well.

With our work, a V&V system of AVL was upgraded to a multi-core architecture while supporting mixed-criticality data. Due to the increased complexity of our system, a more flexible approach was required. Our solution to this problem is a central managing component, called Connectivity Manager, that is responsible for providing shared resources as well as managing communications among multiple cores of the test system. To handle mixed-criticality data streams, we designed a dynamic priority scheduler that was implemented as part of the Connectivity Manager.

In this thesis we presented a dynamic priority communication scheduling approach that is capable of adapting to bandwidth changes on the shared CAN bus. Through simulations, we proved the proper functioning of our algorithm in close to real world scenarios with the result that higher critical data streams are favoured over less critical data streams. A further scenario focused on finding the maximum CAN bus utilization without causing failures. The maximum network utilization of 98.39% could be reached before any message of the four concurrent data streams was delayed.

Furthermore, we investigated on memory consumption as well as on CPU usage of our enhanced V&V system. During stress phases, in which messages of several data streams were queued, the RT kernel CPU usage increased from 1% to 8% in comparison to a plain core. The memory con-

sumption of our Connectivity Manager process was at the peak of 4 MB during regular operating.

An additional test case yielded for the minimum cycle-time that can be successfully handled by our system. Through simulations we reached a minimum cycle-time of 4 milliseconds without violating deadlines. In a long term test-run we prove the stability of our demonstrator, since our system persisted functioning throughout 88 hours with a CAN bus utilization of 67.80%. Via an external device we also applied additional load to the network, to simulate also other nodes on the bus. As a result, we were able to demonstrate the real world applicability of our system, as in live systems the CAN bus is usually limited to a utilization of 40% [Davis et al., 2007].

Eventually, we demonstrated how our system manages overload situations and seeks for maximum bandwidth utilization of the underlying CAN bus to catch up on delays. As a principal achievement, our system can be used to extend the utilization limit of the CAN bus.

One of our major goals for further improvements of the demonstrator is the support of multiple communication technologies, such as CAN-FD or Automotive Ethernet. Although the final release of the demonstrator is designed for the CAN bus and thus for the CAN message format, crucial design definitions were held as generic as possible. For example the entries of the ring-buffer can easily be adapted to any other message format. Furthermore, read and write operations are designed to hand over data buffers that are in the size of byte instead of specific messages. Hence, data blocks can be converted to any message type. A next demonstrator might be in possession of multiple network interfaces varying in bit rate and message type. With respect to bandwidth demands, the Connectivity Manager could independently decide during runtime which interface to use for which application.

A further design consideration that would enhance the demonstrator, is the distribution of tasks between multiple Connectivity Manager instances. This concept is called load balancing and is mainly used to uniformly distribute the load between multiple components so that a single component is never confronted with a peak load scenario. Moreover, each Connectivity Manager instance could be in possession of a different set of interfaces, with the result that each connection could be multiplexed in the best way. However, an

additional component would be necessary that takes over the coordination of the load balancing and the task distribution among the multiple Connectivity Manager instances. Consequently, an additional third component in the communication flow would also mean that the administrative expenses increase. On the other side, tasks could be dynamically swapped between Connectivity Managers.

As a last improvement of our demonstrator, we suggest to file the timestamp at which the last message of each connection was sent. With the help of this timestamp, the waiting-time of a connection, in case of a network bottleneck, could be determined and thus it would be possible to find a fair bus load distribution among all active connections. This approach would not only consider the deadline and the criticality of data streams, but also the amount of time data streams are delayed.

Appendix

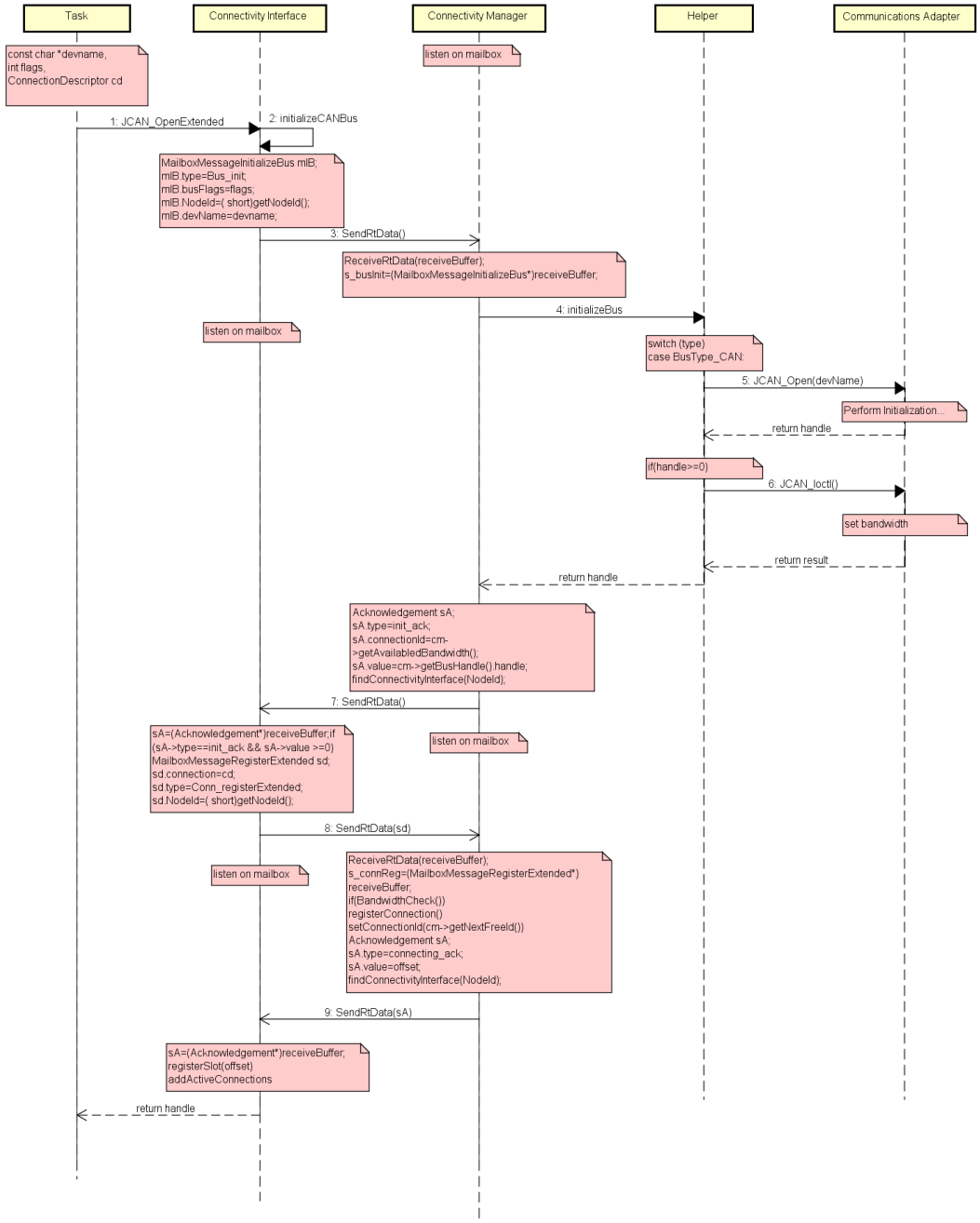


Figure A.1: Complex sequence diagram of the extended open operation.

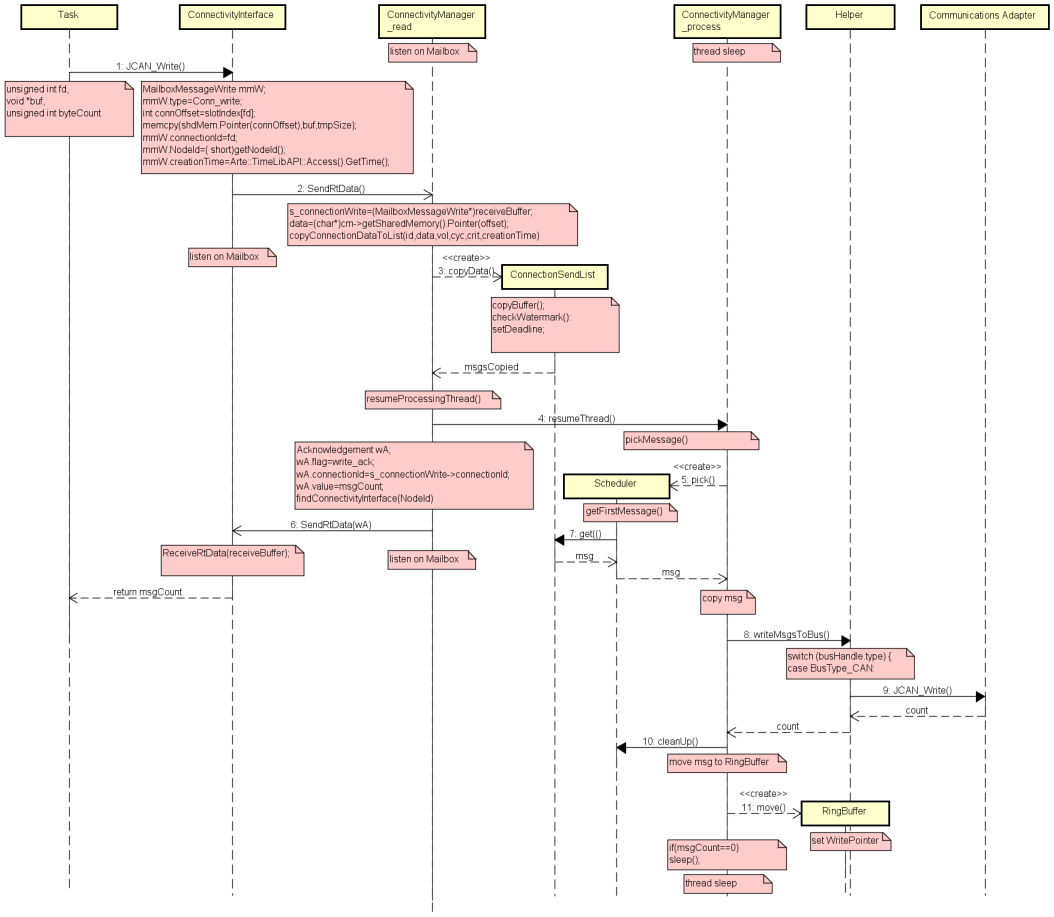


Figure A.2: Complex sequence diagram of the write operation.

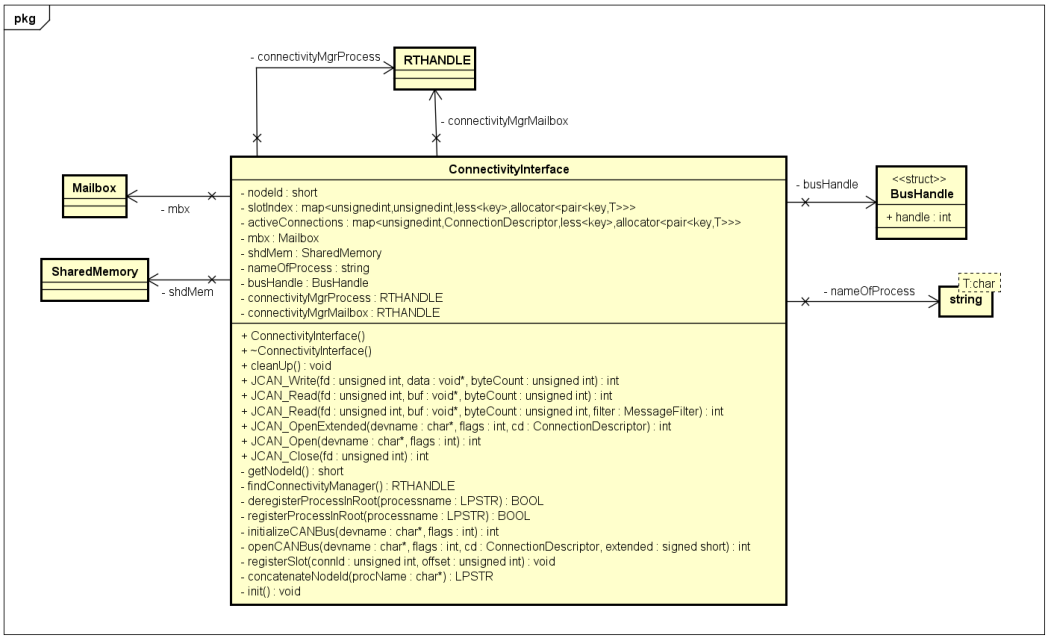


Figure A.3: Connectivity Interface - detailed class diagram.

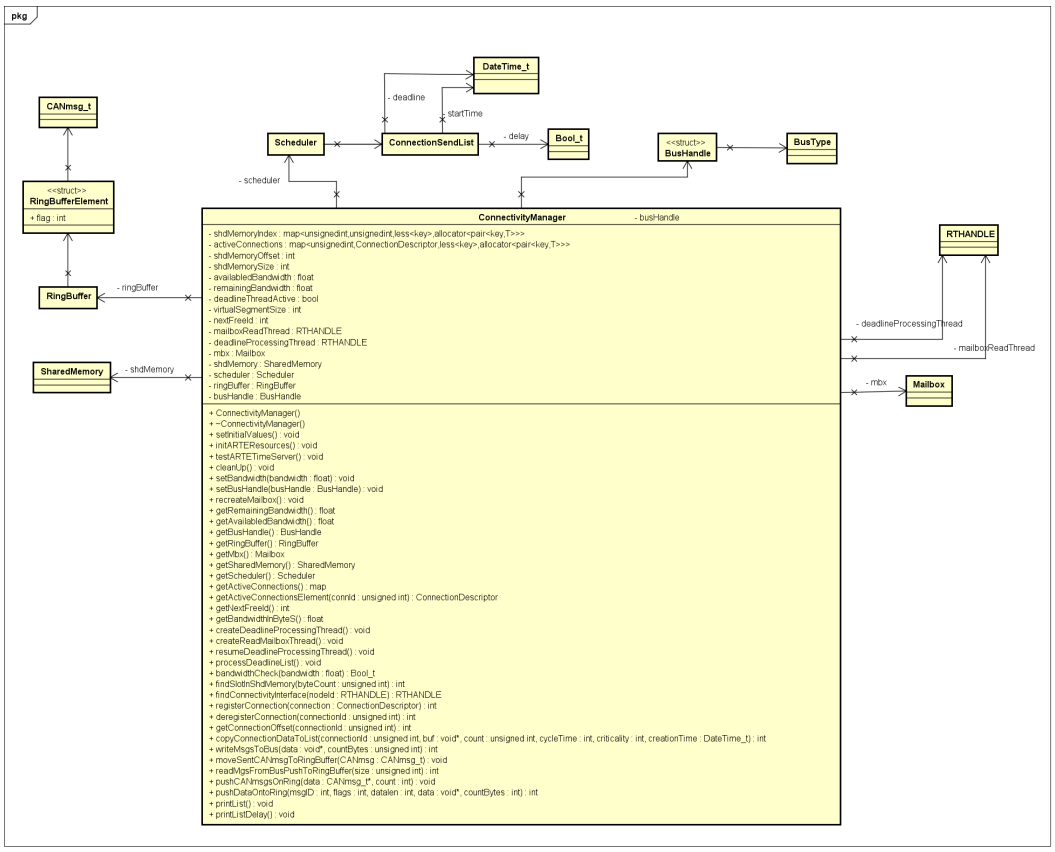


Figure A.4: Connectivity Manager - detailed class diagram.

Bibliography

[Albert, 2004] Albert, A. (2004). Comparison of event-triggered and time-triggered concepts with regard to distributed control systems. *Embedded World*, pages 235–252.

[Audsley and Burns, 1990] Audsley, N. and Burns, A. (1990). Real-Time System Scheduling. 2(3092).

[Audsley et al., 1993] Audsley, N., Burns, A., Richardson, M., Tindell, K., and Wellings, A. J. (1993). Applying New Scheduling Theory To Static Priority Preemptive Scheduling. *Software Engineering Journal*, (September):284–292.

[Baker, 1990] Baker, T. P. (1990). A Stack-Based Resourse Allocation Policy for Realtime Process. *Real-Time Systems Symposium*, pages 191–200.

[Baruah et al., 2010] Baruah, S., Li, H., and Stougie, L. (2010). Towards the design of certifiable mixed-criticality systems. *Real-Time Technology and Applications - Proceedings*, pages 13–22.

[Baruah and Haritsa, 1997] Baruah, S. K. and Haritsa, J. R. (1997). Scheduling for overload in real-time systems. *IEEE Transactions on Computers*, 46(9):1034–1039.

[Bosch, 1991] Bosch (1991). CAN Specification Version 2.0. page 72.

[Burns et al., 1994] Burns, A., Nicholson, M., Tindell, K., and Zhang, N. (1994). Allocating And Scheduling Hard Real-Time Tasks On A Point-To-Point Distributed System. (March).

[Buttazzo and Stankovic, 1993] Buttazzo, G. C. and Stankovic, J. A. (1993). RED: Robust Earliest Deadline Scheduling. pages 2–7.

[Charette, 2009] Charette, R. N. (2009). This Car Runs on Code.

[Davis, 2014] Davis, R. I. (2014). A review of fixed priority and EDF scheduling for hard real-time uniprocessor systems. *ACM SIGBED Review*, 11(1):8–19.

[Davis et al., 2007] Davis, R. I., Burns, A., Bril, R. J., and Lukkien, J. J. (2007). Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272.

[Di Natale et al., 2012] Di Natale, M., Zeng, H., Giusto, P., and Ghosal, A. (2012). Understanding and Using the Controller Area Network Communication Protocol: Theory and Practice. *inst. eecs. berkeley. edu/˜ ee249/fa08/Lectures/ . . . .*

[Grujon, 2011] Grujon, C. (2011). Flexible Embedded Systems Architectures for a Future of Change. pages 12–13.

[Jensen et al., 1985] Jensen, E. D., Locke, C. D., and Tokuda, H. (1985). A Time-Driven Scheduling Model for Real-Time Operating Systems. *6th IEEE Real-Time Systems Symposium RTSS' 85*, pages 112–122.

[Johansson and Martin, 2005] Johansson, K. H. and Martin, T. (2005). Vehicle Applications of Controller Area Network. *Sensors Peterborough NH*, VI:741–765.

[Joseph and Pandya, 1986] Joseph, M. and Pandya, P. (1986). Finding Response Times in a Real-Time System.

[Kopetz, 1997] Kopetz, H. (1997). *REAL-TIME SYSTEMS, Design Principles for Distributed Embedded Applications*.

[Koscher, 2014] Koscher, K. (2014). Securing Embedded Systems: Analyses of Modern Automotive Systems and Enabling Near-Real Time Dynamic Analysis. *Ph.D Dissertation*.

[Kraus et al., 2016] Kraus, D., Leitgeb, E., Plank, T., and Löschnigg, M. (2016). Replacement of the Controller Area Network (CAN) Protocol for Future Automotive Bus System Solutions by Substitution via Optical Networks. pages 1–8.

[Lawrenz, 2013] Lawrenz, W. (2013). *CAN System Engineering*, volume 53. Springer-Verlag London.

[Leen and Heffernan, 2002] Leen, G. and Heffernan, D. (2002). Expanding automotive electronic systems. *Computer*, 35(1):88–93.

[Leen et al., 1999] Leen, G., Heffernan, D., and Dunne, A. (1999). Digital networks in the automotive vehicle. *Growth (Lakeland)*, 10(6):257 – 266.

[Lehoczky, 1990] Lehoczky, J. P. (1990). Fixed priority scheduling of periodic task sets with arbitrary deadlines.

[Lehoczky and Sha, 1986] Lehoczky, J. P. and Sha, L. (1986). Performance of Real-time Bus Scheduling Algorithms.

[Li and Yao, 2003] Li, Q. and Yao, C. (2003). *Real-Time Concepts for Embedded Systems*.

[Liu and Layland, 1973] Liu, C. L. and Layland, J. W. (1973). Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.

[Locke, 1986] Locke, D. (1986). Best-Effort in Decision Making For Real-Time Scheduling.

[Main, 2010] Main, C. (2010). Virtualization on multicore for industrial real-time operating systems. *IEEE Industrial Electronics Magazine*, 4(3):4–6.

[Mayer, 2006] Mayer, E. (2006). Serial Bus Systems in the Automobile. *Vector Informatik GmbH*, pages 1–6.

[Meschi et al., 1996a] Meschi, A., Di Natale, M., and Spuri, M. (1996a). Earliest Deadline Message Scheduling with Limited Priority Inversion 1 Introduction Earliest Deadline Approach in Network Scheduling. pages 87–94.

[Meschi et al., 1996b] Meschi, A., Di Natale, M., and Spuri, M. (1996b). Priority inversion at the network adapter when scheduling messages with earliest deadline techniques. *Proceedings - Euromicro Conference on Real-Time Systems*, pages 243–248.

[Mischo et al., 2015] Mischo, S., Jorn Stuphorn, Rainer Constapel, P. H., Alexander Leonhardi, Heiko Holtkamp, N. L., and Ochel, S. P. (2015). Bus systems.

[Mishra and Gurumurthy, 2014] Mishra, G. and Gurumurthy, K. (2014). Dynamic Task Scheduling on Multicore Automotive ECUs. *International Journal of VLSI design & Communication Systems (VLSICS)*, 5(6):2–9.

[Mohammadi and Akl, 2005] Mohammadi, A. and Akl, S. G. (2005). Scheduling Algorithms for Real-Time Systems.

[Nager et al., 2017] Nager, M., Baunach, M., Priller, P., and Wurzinger, J. (2017). Real-Time Multiplexing of Mixed-Criticality Data Streams for Automotive Multi-Core Test Systems. Number Icc, pages 220–227.

[Natale and Meschi, 2001] Natale, M. D. I. and Meschi, A. (2001). Scheduling Messages with Earliest Deadline Techniques. (1993):255–285.

[Navet and Simonot-Lion, 2008] Navet, N. and Simonot-Lion, F. (2008). A Review of Embedded Automotive Protocols. *Automotive Embedded Systems Handbook, Industrial Information Technology Series*, pages 1–42.

[Navet and Simonot-Lion, 2013] Navet, N. and Simonot-Lion, F. (2013). In-vehicle communication networks-a historical perspective and review. *Industrial Communication Technology Handbook, Second Edition*, 96(June 2005):1204–1223.

[Navet et al., 2005] Navet, N., Song, Y., Simonot-Lion, F., and Wilwert, C. (2005). Trends in automotive communication systems. *Proceedings of the IEEE*, 93(6):1204–1222.

[Risat, 2010] Risat, M. P. (2010). Scheduling Algorithms For Fault-Tolerant Real-Time Systems. *Online*, 6(3):93–100.

[Saez et al., 1999] Saez, S., Vila, J., and Crespo, A. (1999). Soft aperiodic task scheduling on hard real-time multiprocessor systems. *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No.PR00306)*, pages 424–427.

[Sha et al., 2004] Sha, L., Abdelzaher, T., Årzén, K. E., Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J., and Mok, A. K. (2004). Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3 SPEC. ISS.):101–155.

[Shin and Ramanathan, 1994] Shin, K. G. and Ramanathan, P. (1994). Real-Time Computing: A New Discipline of Computer Science and Engineering. *Proceedings of the IEEE*, 82(1):6–24.

[Sprunt, 1990] Sprunt, B. (1990). Aperiodic Task Scheduling for Real-Time Systems. *Department of Electrical and Computer Engineering*, Doctor in:210.

[Spuri and Buttazzo, 1994] Spuri, M. and Buttazzo, G. C. (1994). Efficient aperiodic service under earliest deadline scheduling. *Proceedings - Real-Time Systems Symposium*, pages 2–11.

[TenAsys, 2007] TenAsys (2007). INtime Real-Time Operating System Helps.

[TenAsys, 2009] TenAsys (2009). INtime ® 4.0 Software 31001-6.

[Tindell et al., 1995] Tindell, K., Burns, A., and Wellings, A. (1995). Analysis of Hard Real-Time Communications. pages 1–26.

[Tindell and Hansson, 1994] Tindell, K. W. and Hansson, H. (1994). Analysing Real-Time Communications : Controller Area Network (CAN) *. (06).

[Tindell et al., 1994] Tindell, K. W., Hansson, H., and Wellings, A. J. (1994). Analysing real-time communications: Controller area network (CAN). *Proceedings - Real-Time Systems Symposium*, (06):259–263.

[Tuohy et al., 2015] Tuohy, S., Glavin, M., Hughes, C., Jones, E., Trivedi, M., and Kilmartin, L. (2015). Intra-Vehicle Networks: A Review. *IEEE Transactions on Intelligent Transportation Systems*, 16(2):534–545.

[Zuberi and Shin, 1995] Zuberi, K. and Shin, K. (1995). Non-preemptive scheduling of messages on controller area network for real-time control applications. *Real-Time Technology and Applications*, pages 240–249.