UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

A Dynamically Reconfigurable Hard-Real-Time Communication Protocol for Embedded Systems

Thesis submitted to the
Faculty of Computer Science, Electrical Engineering and Mathematics
of the
University of Paderborn
in partial fulfillment of the requirements for the
degree of *Dr. rer. nat.*

by

Eng. André Luiz de Freitas Francisco

April 2012

Acknowledgments

This thesis is a result of my research activities at the *International Graduate School of Dynamic Intelligent Systems* of the University of Paderborn in Germany. I am grateful to Prof. Dr. Franz Rammig, the supervisor of this thesis, who not only gave me the chance to pursue my Ph.D. but also provided valuable feedback and suggestions during this work. I thank Prof. Dr. Ansgar Trächtler for the time I spent with his working group *Control Engineering and Mechatronics* and for his comments on my research. I would like to dedicate this thesis to the memory of Prof. Dr. Joachim Lückel, who will be remembered for his exemplarily enthusiastic and passionate attitude towards life and his work.

I am thankful to Siemens AG for granting me the fellowship for my Ph.D. studies and to Mr. Jürgen Bermann for his technical support. I also would like to thank the staff of the *International Graduate School* for their continuous assistance with the necessary formalities. I appreciate the support from Dr. Karl-Peter Jäker that made possible the implementation of the prototypes of this work and also the precious secretarial assistance of Mrs. Annette Bökamp-Gros. I thank Mauro Zanella for all the support and for encouraging me to accomplish this work.

This work was carried out in an interdisciplinary environment so I am deeply thankful for the many discussions with colleagues of the working groups *Control Engineering and Mechatronics*, *Design of Distributed Embedded Systems* and also *Power Electronics and Electrical Drives.* Moreover, I appreciate the help of all students that actively participated in this project.

Last, but not least I just cannot express enough gratitude to my parents for their unconditional support during this work and also to my wife for her lovely assistance and for encouraging me at all times.

Abstract

Real-time communication is a basic requirement for many distributed embedded systems. However, for an emerging new class of applications not only real-time behavior but also flexibility and adaptability will become necessary system attributes. Although the time-triggered paradigm has been successfully employed with a variety of real-time applications, it imposes some challenges with respect to dynamic reconfiguration, since the modification of certain communication parameters, such as transmission cycles, may potentially require the complete rearrangement of a global scheduling plan.

In order to increase the flexibility of real-time communication systems a new protocol called TrailCable was designed. It takes advantage of the properties of *Earliest Deadline First* (EDF) scheduling, which include optimal utilization bounds and the possibility to cope with heterogeneous task sets. A communication network is built with full-duplex, point-to-point links, and nodes can route packets to allow multi-hop message delivery. The so-called virtual real-time channels are unidirectional logical paths mapped on the physical network topology that transport data within specified latency times. This work introduces methods for automatically mapping real-time channels on a given network directly from communication requirement specifications.

The activation of real-time channels in the network is permitted only after a successful schedulability analysis, which can be executed automatically by a tool that checks XML-based network configuration models. At run-time, the characteristics of all incoming packets are checked against their specification by an admission control technique called bandwidth guardian, which is used to ensure that occasional faults will not impair the timeliness of other real-time channels.

Time-critical functions of the communication protocol, such as scheduling, admission control, packet routing, and clock synchronization, are implemented by means of dedicated hardware. A fully operational FPGA-based prototype was built and used in different measurement experiments to validate the real-time behavior of the protocol under real conditions.

Zusammenfassung

Echtzeitkommunikation ist eine Grundanforderung für viele verteilte eingebettete Systeme. Für eine neue Klasse von Anwendungen sind jedoch nicht nur Echtzeitfähigkeit, sondern auch Flexibilität und Anpassungsfähigkeit notwendige System-Attribute. Obwohl zeitgesteuerte Kommunikation bereits in einer Vielfalt von Echtzeitsystemen erfolgreich eingesetzt wurde, kann dadurch die Aufgabe, ein dynamisches System zu rekonfigurieren, erschwert werden. Der Grund dafür ist, dass unter Umständen eine Modifikation bestimmter Parameter wie z. B. des Sendezyklus dazu führen kann, dass die gesamte Kommunikationsplanung angepasst werden muss.

Um die Flexibilität zu erhöhen, wurde in dieser Arbeit ein neues Kommunikationsprotokoll namens TrailCable konzipiert. Es profitiert von den Eigenschaften des *Earliest Deadline First* Scheduling-Verfahrens, wie z. B. der optimalen Ausnutzung von Ressourcen und der Unterstützung von heterogenen Tasks. Ein Kommunikationsnetzwerk wird aufgebaut mit Hilfe von voll-Duplex-, Punkt-zu-Punkt-Verbindungen, wobei die Knoten Datenpakete weiterleiten können, um eine Multi-hop Übertragung zu gewährleisten. Die sogenannten virtuellen Echtzeit-Kanäle sind unidirektionale, logische Pfade, die auf die Netzwerk-Topologie abgebildet werden, um Daten innerhalb der vorgegebenen Latenzzeiten zu übermitteln. Es werden Methoden vorgestellt, die es erlauben, automatisch die Kommunikationsanforderungen erfüllende Echtzeit-Kanäle auf das Netzwerk abzubilden.

Echtzeit-Kanäle können nur dann aktiviert werden, wenn im Voraus ein Akzeptanztest erfolgreich durchgeführt wurde. Solch eine Prüfung kann mittels eines Tools automatisch erfolgen. Alle dafür notwendigen Netzwerkinformationen werden aus XML-Dateien eingelesen. Zur Laufzeit prüft ein Mechanismus, der Bandbreitenwächter genannt wird, ob die eingelesenen Pakete mit ihrer Spezifikation übereinstimmen, damit Fehler die Echzeitfähigkeit anderer Kanäle nicht beeinträchtigen können.

Zeitkritische Funktionen des Kommunikationsprotokolls, wie Scheduling, Bandbreitenwächter, Routing und Uhrsynchronisation, sind mittels dedizierter Hardware implementiert. Ein voll funktionsfähiger FPGA-basierter Prototyp wurde aufgebaut und in zahlreichen Tests evaluiert, um das Echtzeit-Verhalten des Protokolls unter realen Bedingungen zu testen und zu analysieren.

À minha família

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded system technology is nowadays a significant innovation driver. The increasing importance of embedded systems is reflected in the statistics provided by the ARTEMIS website [15]:

- About 98 % of computing devices are now embedded
- Global market is worth €60 billion with annual growth rates of 14 %
- More than 16 billion embedded devices predicted by 2010 and over 40 billion by 2020
- More than 35 % of the value of a modern car is due to embedded electronics

Not only the number of embedded devices is increasing, but also their connectivity. Distributed embedded systems are present in a great variety of applications, such as cars, aircraft, industrial production plants, medical equipments, etc. A common requirement on these systems is real-time behavior, because their embedded computers are usually used to control physical processes.

The ever increasing complexity of distributed real-time systems, including their communication infrastructure, imposes many different design challenges. Requirements on communication protocols are becoming more stringent with respect to dependability and security, but at the same time higher flexibility and dynamic behavior are also needed.

This thesis will focus on wired real-time communication for embedded systems. In the context of this work, a new protocol aimed at providing higher flexibility was designed, implemented, and verified. Typical applications considered for this study are, for example, distributed mechatronic systems.

1.1 Research Motivation

The requirements on real-time communication systems are continually changing over time. Let us consider the example of the automotive industry. In the mid-1980s, CAN-bus [45] was developed at Robert Bosch GmbH out of the need to exchange information among distributed electronic control units (ECUs) for reducing the amount of cable harnesses. This enabling technology has then rapidly become a *de-facto* communication standard for cars. However, the ever increasing complexity and criticality of new applications (e.g., *drive-by-wire* systems) imposed new requirements on communication so that new data buses were needed. The solution came by means of the time-triggered paradigm [32] and corresponding protocol implementations such as FlexRay [35]. With the current development pace, in which software gets about 10 times bigger from one car generation to the next [20], it is possible to predict new challenges for the design of future communication infrastructures. With the integration of more and more functions into ECUs, communication protocols will have to cope with different, and maybe conflicting, application demands such as real-time and best-effort data transmission.

Another challenge for the design of real-time communication protocols is the emerging need for dynamic reconfiguration support. Generally, real-time systems are designed in a static fashion, but an increasing number of applications will be required to adapt themselves at run-time. Examples are self-optimizing mechatronic systems such as those addressed in the *Collaborative Research Center 614* [9] at the University of Paderborn.

Even common assumptions taken into account for the design of real-time communication systems can be revisited for new protocols. One example is the assumption that processes are always activated in equally spaced time intervals. Although control loops generally rely on this property to operate correctly, this behavior can also be restrictive in some other applications such as radars, where the round-trip time of a transmitted pulse varies according to the target distance. In fact, real-time platforms with higher flexibility can be employed with a larger variety of applications.

Although great versatility is expected from real-time communication systems, costs must be kept as low as possible and therefore resource-efficient hardware architectures are crucial. In many embedded systems, communication controllers must be small enough to fit in a small die area. Data transmission bandwidth is another limited resource that must be efficiently utilized. To achieve that, low communication overhead and good scheduling policies are important factors to be considered.

The objective of this work is to address the challenges mentioned and to provide solutions that can be integrated into a new communication protocol. Another goal of this thesis is to provide an abstraction layer for the communication infrastructure in such a manner that independent, concurrent applications can also be independently designed and mapped on a given network. With the time-triggered paradigm, one of the first steps to build a distributed system is to find a suitable dispatching plan for all functions, which is not always a simple task because, for example, a compromise must be found to establish the basic communication period. The alternative approach investigated in this thesis is to first determine the communication requirements for each functionality alone and then, by means of an automatic process, to search for a feasible real-time solution that meets the individual, original requisites. Another advantage of the proposed approach is that modification of the requirements of a certain function can be made transparent to others as long as feasibility holds. The latter approach implies that the communication infrastructure must adapt itself to application demands and not the other way around.

1.2 Chapter Outline

This thesis is organized as follows:

Chapter 2, **Background and Related Work** presents paradigms of communication systems and introduces selected commercially available protocols. Moreover, a list of desired requirements for a new embedded real-time communication protocol is proposed.

Chapter 3, **The TrailCable Communication Protocol** begins with the concept of sporadically triggered systems. The TrailCable communication protocol is then explained in detail, including the real-time feasibility tests with a given configuration. A run-time mechanism for fault-tolerance is also presented.

Chapter 4, **The TrailCable Verifier Tool** deals with the modeling of the TrailCable network communication system. Network configurations are described by means of XML files, which can be checked by a tool for real-time schedulability.

Chapter 5, **Dynamic Reconfiguration** discusses the framework that allows a TrailCable network to be reconfigured at run-time. Moreover, this chapter also presents methods for automatically mapping real-time communication channels in a given network.

Chapter 6, **The Communication Engine Hardware** gives an overview of the hardware-based communication engine of the TrailCable protocol. The characteristics of the hardware implementation are analyzed in a design space exploration study.

Chapter 7, **Experimental Results** presents measurement results gained by practical experiments that verify the real-time and fault-tolerance capabilities of the TrailCable protocol.

Chapter 8, **An Application Example: The RailCab Test Track** is a case study that describes the employment of the TrailCable protocol with a real application, namely the RailCab test track.

Chapter 9, **Conclusion** closes this thesis with a review of the TrailCable protocol features and gives an outlook to future research directions.

Chapter 2

Background and Related Work

This chapter introduces some of the paradigms, trade-offs, and implementation aspects of real-time communication systems. A survey of selected communication protocols used in distributed control systems is also presented. The objective is not to provide an exhaustive list, but rather to present some of the approaches employed to build up real-time communication protocols. Different requirements lead to different designs and therefore one particular real-time communication protocol can be well suited for a given application while not for others.

2.1 Paradigms Overview

According to Kopetz [57], data communication protocols for embedded systems can be divided into two main categories: event- and time-triggered. Many times, this distinction goes beyond the communication layer and is also used to characterize embedded systems as a whole, including the corresponding operating system and application software. The peculiarities of the two approaches will be outlined in the following.

2.1.1 Event-Triggered

Event-triggering was the first paradigm used for data communication. As the name suggests, at any time an event can trigger data transmission. Such events can be generated by internal timers, external interrupts, or other mechanisms. The term event-triggered is usually used to indicate that triggering actions are not entirely pre-coordinated or foreseen. As a consequence, real-time feasibility analysis can be very difficult or even impossible to perform.

2.1.2 Time-Triggered

Determinism is achieved in a time-triggered system by pre-planning the execution of processes or communication tasks and their precise instant of activation. Time-triggered communication systems usually use the time-division multiple access (TDMA) approach to share the transmission medium between different nodes in a static, pre-defined manner. In order to allow a consistent execution of a TDMA schedule in a distributed system, global clock synchronization becomes necessary to make all participating nodes able to operate in a coordinated manner.

2.1.3 Discussion

In practical applications of distributed embedded systems, especially in the domain of mechatronics, it turns out that control algorithm processes are executed in cycles activated periodically. The periodic activation of a determined set of packets of fixed and pre-defined sizes along with a known medium access scheme provide rules that restrict the uncertainty in the event-triggered approach and can allow checking whether real-time behavior can be achieved under a given configuration. This being the case, an advantage of the event-triggered paradigm over the time-triggered one is that nodes may start transmission as soon as data is available, which may contribute to the reduction of communication latency. However, such an advantage exists only when the communication infrastructure has a low utilization factor. When transmission capacity is high, not only the maximum latencies are maximized, but also the communication jitter. Such effects basically depend on the communication protocol used, mainly in the available link bandwidth, and in the approach used for medium access (arbitration) and packet transmission. Time-triggered protocols allow a deterministic behavior of the communication system, with possibly higher, though constant, response times.

When it comes to control applications, there is a trade-off in selecting the appropriate paradigm. For a variety of control systems, communication latencies can be taken into account when designing plant controllers, so that the additional delay caused by data processing can be compensated for. This is, however, only possible when latencies are rather constant, justifying the use of time-triggered architectures. On the other hand, if certain control processes in a distributed system require lower communication latency times or lower transmission periods than others, it may become a complex task to accommodate the necessary data traffic efficiently with a TDMA schedule. A more detailed study involving the trade-offs of both event- and time-triggered approaches for control systems is presented in [12].

The comparison between both paradigms however, goes well beyond temporal requirements. One of the most important reasons for employing time-triggered communication systems is the reliability level that can be offered by protocols based on this approach. The so-called *by-wire* applications are good examples where safety plays a key role and dependable technologies, including communication protocols, are indispensable.

2.2 Network Architectures

Another aspect to be taken into account in the classification of communication systems is the physical infrastructure, including characteristics of the data transmission medium, type of supported network topologies, and hardware costs.

The first implementations of communication protocols for embedded systems were based on shared buses, i.e., the transmission of a given node is broadcast to all others via a common medium interconnect. Such an arrangement is called bus topology, its main advantage being the simplicity of the necessary hardware and thus reduced component costs. However, some aspects must be considered when bus topologies are used. Firstly, the electrical bus characteristics restrict the maximum bus length, bandwidth, and number of nodes. Also, a single point of failure, such as a faulty node or a cable disruption or short-circuit, can lead to a general failure. Furthermore, good arbitration techniques are required in order to allow bus access to all nodes in a fair and efficient manner.

To overcome the safety restrictions imposed by a bus topology, usually a star interconnect is employed. The reason is that single link or node failures can be isolated from the remaining network. The fault-tolerance that can be achieved with a star topology justifies its use in applications that require higher safety levels. The work by Ademaj *et al.* [10] studies the increased safety of the star interconnect when compared to the shared bus. Another characteristic of the star interconnect is that the point-to-point links from the nodes to the central hub allow the use of electrical standards with higher transmission rates and even the use of optical connections. On the other hand, a star interconnect normally requires higher cabling effort and an extra hub, resulting in higher costs when compared to the bus interconnect.

With the advent of large scale production of Ethernet-based components, new technologies based on this standard are extending the original application focus, which is the classical home and office network, to real-time communication systems. The main hurdle in this respect is to change the medium-access approaches, which are based in collision detection and probabilistic contention solving mechanisms, to deterministic procedures. In order to guarantee real-time behavior, extensions to the original Ethernet standard are made, such as the introduction of clock synchronization mechanisms and policies for coordinated medium accesses. The use of Ethernet-based real-time solutions was originally aimed at the networking of industrial plants, which can benefit from the usage of the same communication standard for real-time machine control, supervision systems, and plant management. Despite the fact that Ethernet controllers are usually more complex and require more hardware resources than the controllers of some communication protocols for embedded systems, they allow for the creation of flexible connection arrangements such as ring, star or mixed topologies with relatively high transmission rates.

2.3 Overview of Protocols

This section introduces some of the communication protocols that are used in real-time embedded systems. Although there are many more available solutions, the presented ones allow a comparison study of different design approaches and compromises that must be made when designing communication systems. Other communication protocols under research in academia are also referenced throughout this thesis.

CAN bus

The Controller Area Network (CAN) developed by Bosch in 1988, has become the most widely used communication bus in the automotive sector and can be also found in other domains such as factory automation. According to the CAN in automation (CiA) organization [24], more than 2 billion CAN nodes have already been sold. Moreover, CiA presents surveys by market research institutes [23] stating that more than 400 million CAN controllers were sold in 2005 with an estimative of twice as much in 2010.

CAN is a low-cost protocol based on a bus topology. Participating nodes access the transmission medium via a non-destructive and priority-based arbitration procedure [44]. The priority is determined by an ID field at the beginning of a message. When two or more nodes attempt a transmission simultaneously, the message with the highest priority (lowest ID number) gains access to the bus and is able to transmit the remaining part of the packet. Such arbitration is supported by an electrical mechanism at the physical layer that uses the concept of recessive (logically 1) and dominant (logically 0) bits. Nodes concurring in the arbitration process are able to drive each bit of the ID field simultaneously onto the bus and read back the current state. If the bus state does not correspond to the bit being transmitted, this means that the recessive bit of a certain message was overwritten by a dominant bit of another. In this case the message with the recessive bit backs out of the arbitration process and prepares itself to receive the upcoming data from another node. The arbitration procedure continues until only one message is left in the process, which happens during the last ID bit.

The CAN physical layer consists of a differential transmission pair, which provides good noise immunity. The maximum supported bandwidth is 1 Mbps for buses up to 40 meters

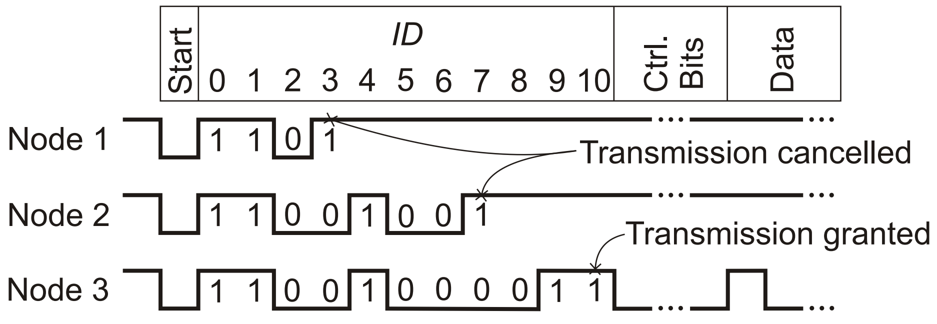


Figure 2.1: CAN arbitration example

and is reduced with the increase in length. Each CAN message transports up to 8 payload bytes.

Since the CAN standard specifies only the basic data-transfer mechanism among data buffers in different nodes, the implementation of higher level protocols to support the required communication, management and monitoring features is left to the application software. In order to allow interoperability between embedded devices from different manufacturers, higher level communication standards were also introduced. Examples are the CANopen, maintained by the CiA organization, and the J1939, which is a standard of the Society of American Engineers (SAE) institute.

Although largely used today, the CAN bus has some limitations to be employed in some new generation automotive networks. The main reasons are safety, performance and determinism requirements imposed on emerging architectures. To overcome such restrictions, time-triggered protocols were developed and have already achieved operational status. Even the CAN bus has gained its own time-triggered specification, the TTCAN [45]. Although the TTCAN allows deterministic data communication, bandwidth and safety gains are better exploited in the native time-triggered protocols presented below.

TTP/C

Developed by the working group of Prof. Herman Kopetz at Vienna Technical University, the Time-Triggered Protocol (TTP/C) [86, 84] is a deterministic communication protocol that meets the requirements of the now superseded SAE class C specification [77]. The structure of a time-triggered distributed computing cluster consists of fault-tolerant units (FTUs), each one representing a single node in the network, which are interconnected via the communication system.

Access to the transmission medium in TTP/C is based on broadcasts according to the time-division-multiple-access (TDMA) approach, which uses a statically pre-defined scheduling table and requires a common notion of time in all participating nodes. Advantages of this method are predictability and low jitter, but at the price of a lack of flexibility.

The communication network interface (CNI) of TTP/C is an autonomous component, responsible for managing all communication functions independently of a host microcontroller. Due to the TDMA approach, data packets have low overhead, since it is possible to coordinate the communication traffic based on global time references. In order to achieve this, all relevant information concerning the protocol operation is stored in the CNIs of all nodes in the so-called MEDL (Message descriptor list).

TTP/C was designed for applications with rigorous safety requirements. For this purpose, fault-tolerant clock synchronization, redundancy management, bus guardians and membership agreement are some of the services and mechanisms provided by the protocol. Such TTP/C capabilities, combined with a certified development process, have made it possible to employ the protocol, for example, in aerospace and rail-signaling applications.

When it comes to network topologies, TTP/C supports both bus and star arrangements. For increased safety, the star is preferred to the bus topology [10]. TTP/C has been tested with different physical layers, such as RS-485 and Ethernet. Based on the experience gained with the TTP/C protocol, an alternative development, the TT-Ethernet [58] concept, was introduced. TT-Ethernet is compatible with the IEEE 802.3 Ethernet standard [43] and can be implemented with commercial-off-the-shelf (COTS) components. Both software-only and dedicated hardware implementations for TT-Ethernet are possible, the first being a low-cost solution and the second a high-performance alternative. As opposed to TTP/C, TT-Ethernet allows both event- and time-triggered communication, with the latter always having preference over non-deterministic operation.

FlexRay

FlexRay [35] is the result of the efforts of car and chip manufacturers to develop a dependable and deterministic, yet flexible, data communication system to interconnect ECUs in automotive systems. It is becoming the *de-facto* standard in this application field, with more and more companies supporting the initiative. Like TTP/C, FlexRay is a time-triggered protocol that uses the TDMA bus-access approach for data transmission, according to a static schedule, in order to guarantee determinism. One of the main differences, however, is that FlexRay reserves part of the TDMA communication cycle for the so-called dynamic segment (Figure 2.2), where a priority-based scheme (mini-slotting technique) is used to grant the nodes access to the bus. This is specially useful for the non-critical functions in a distributed system that are better served by the event-triggered communication paradigm.

When it comes to interconnections, FlexRay networks can be built using the following topologies: passive bus, active star, or a combination of both. Also, it is possible to

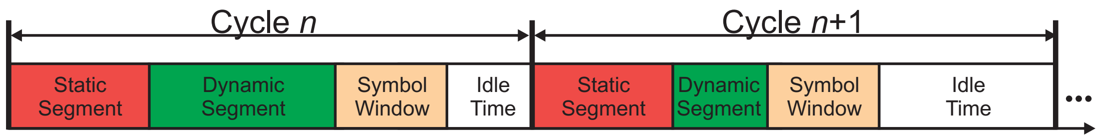


Figure 2.2: FlexRay communication cycle

use redundant, doubled channels when higher reliability or bandwidth is required, or still employ single links when cost and simplicity are a major concern.

Unlike TTP/C, FlexRay relies on its own physical layer standard [34], specially developed for this purpose. The nominal data transmission rate of FlexRay is 10 Mbps with the net throughput being about the half of this value.

AFDX

The Avionics Full-Duplex Switched Network (AFDX) [25], formalized by the ARINC 664 specification, is a safety critical communication protocol designed for interconnecting aircraft computers. This standard is based on IEEE 802.3 Ethernet, which contributes to an overall cost reduction (due to the use of COTS components) and an increase in bandwidth as compared to older avionics data buses such as ARINC 429 [14]. However, in order to guarantee determinism and reliability, the operation of AFDX requires additional mechanisms that are not standard in commercial Ethernet networks.

As the name indicates, all links in AFDX are full-duplex. This is a requirement to avoid collisions in the transmission medium, which lead to non-deterministic behavior. The main types of components in an AFDX network are end-systems and switches. An end-system represents the interface between the AFDX network and an avionics subsystem. End-systems are always connected to AFDX switches, which can, in turn, be connected to other switches to expand the network (Figure 2.3). To cope with communication failures, AFDX is able to employ a pair of independent networks for data transmission. In this kind of operation, the destination end-system identifies and verifies the replicas arriving via the two redundant paths and passes a single instance of the message to higher-level protocols.

A basic difference between AFDX and other Ethernet-based system is the concept of Virtual Links, which are logical data flows from one source to one or more destinations, in a scheme that resembles the multi-drop characteristic of ARINC 429. Moreover, instead of using the destination address to route data packets, AFDX employs 16-bit message identifiers that are used by the switches to determine the way packets are forwarded in the network. Multiple Virtual Links are able to share the same Ethernet link. Therefore, in order to guarantee deterministic behavior, each Virtual Link must have a pre-defined bandwidth, which is a function of its transmission rate and payload. The Bandwidth

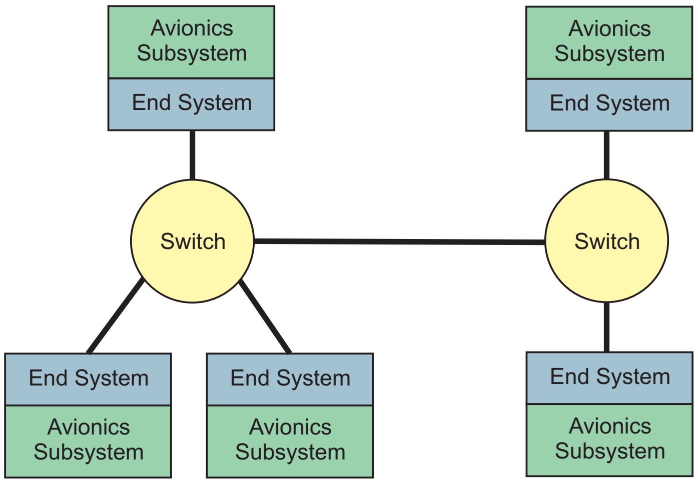


Figure 2.3: An example of an AFDX network

Allocation Gap (BAG) is the minimum allowed interval between two consecutive packets and ranges, in power of 2, from 1 to 128 milliseconds. The AFDX employs the UDP/IP protocol rather than TCP/IP, thus making possible to limit the traffic of a given Virtual Link to a single data packet per BAG. Virtual Link Schedulers are responsible for ensuring the bandwidth specification, selecting a given frame for transmission, and performing bandwidth regulation to guarantee a deterministic Quality of Service (QoS) of AFDX. The schedulers select a frame from the input buffers for transmission according to the round-robin algorithm.

PROFINET

PROFINET [74] is an Ethernet-based communication protocol aimed at industrial automation applications. It was initiated by PROFIBUS International with development support from Siemens AG, which has selected PROFINET as one of the main data communication standards for its automation product line. Since PROFINET is an open standard, different vendors now offer a variety of compatible hardware and software solutions.

There are two so-called perspectives in the communication standard [81], namely PROFINET CBA (Component Based Automation) and PROFINET IO (Input/Output) that are complementary and may coexist. The focus of the former is on the realization of modular applications and machine-machine communication. PROFINET CBA facilitates the management of complex automation plants as autonomous components can be used to abstract many aspects of an equipment, including electrical, mechanical, and software characteristics. PROFINET IO is employed to interconnect controllers and distributed peripherals and follows the consumer/provider model.

PROFINET defines three classes of communication requirements to meet the needs of higher-level non-real-time applications ranging to low-latency, real-time distributed control devices. The first of them employs conventional TCP/IP communication and achieves typical cycle times of 100 ms. It is a so-called class A solution since no modification in the TCP/IP stack nor special hardware is required.

The second class, called PROFINET RT (**R**eal-**T**ime), provides a real-time cyclic communication that is implemented by a special driver that bypasses the ordinary communication stack, resulting in lower cycle times, in the order of 10 ms. In order to prevent normal TCP/IP traffic from impairing the correct functionality of PROFINET RT, the packets of the latter are assigned a higher priority according to the IEEE802.1q standard. PROFINET RT is a class B solution because special software drivers are required, but it works with COTS Ethernet hardware that comply with IEEE802.1q.

PROFINET IRT (**I**sochronous **R**eal-**T**ime) provides the highest possible performance of all three classes. In this mode bandwidth is reserved for IRT packets at the beginning of each communication cycle, so that even with a high utilization of TCP/IP traffic the real-time behavior can be guaranteed. For this mode, a precise clock synchronization of the network equipment is implemented in order to meet the precision requirements on motion control applications. This characterizes a class C concept that employs special software and hardware. With PROFINET IRT it is possible to reach cycle times of 250 μs with a jitter below 1 μs.

EtherCAT

EtherCAT, originally developed by Beckhoff [47], is also an Ethernet-based communication standard for industrial applications. The EtherCAT concept consists of one master and several slaves that make up a logical ring, although line, star and tree topologies are physically permitted (Figure 2.4). A characteristic of EtherCAT is that the master, as opposed to the slaves, needs no special hardware for its operation, just a standard Ethernet port. Once the master triggers the transmission of an EtherCAT packet to the ring, the slaves, by means of dedicated hardware, are able to read and alter the contents of the packet as it passes by. As the delay caused by the slaves is minimal, very high performance can be achieved, with cycle times in the range of some microseconds, and jitter well below 1 μs. As with PROFINET, EtherCAT also employs clock synchronization. The Precision Time Protocol, IEEE1588 Standard [42], is used for this purpose.

Since devices are daisy-chained, a fault in the network, either in a node or in a connection, may isolate several slaves. In order to overcome this kind of problem, EtherCAT also specifies a redundancy scheme. It consists in adding a second Ethernet port to the master node in order to close a ring in the network. Normally, only one of the Ethernet ports at the master node is operational (the second is on stand-by). If a disruption occurs in

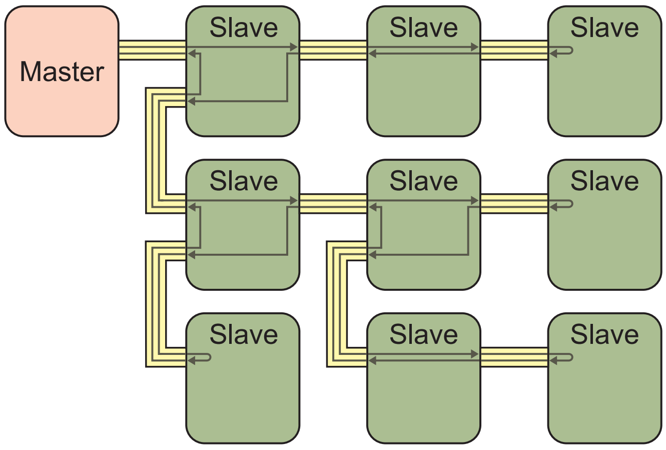


Figure 2.4: An example of an EtherCAT network

the network and the ring is broken, the Ethernet ports at the master node are then used to initiate transmission in both ring directions. If the fault was caused by a connection problem, all slaves will be able to rejoin the network.

SpaceWire

As the name suggests, SpaceWire is a data-communication standard that is used in satellites and other spacecraft. It is based on the superseded IEEE1355-1995 standard [41], which was derived from the T9000 Transputer [26] asynchronous serial connections, and employs LVDS (TIA/EIA-644) [83] as the physical layer. Today SpaceWire is backed by ESA (European Space Agency), which has issued an official standard for it [31].

SpaceWire is based on full-duplex, point-to-point serial links with signaling rates ranging from 2 Mbps to 400 Mbps. The full-duplex links comprise a total of four differential pairs (two for each direction) and utilize a data-strobe encoding technique that allows simple recovery of the transmission clock. Nodes in a SpaceWire network can be either directly connected to others or via switches that rely on wormhole routing to minimize latency times.

Different types of addressing are defined for SpaceWire. With path addressing, when a packet arrives at a switch, the first byte of payload is taken to determine the destination port and then deleted. With logical addressing, the destination address is the first packet byte and is kept by the switches. Another possibility is region addressing, a combination of the two types above that facilitates multilevel routing and reduces the size of routing tables.

In SpaceWire a flow-control mechanism is used to avoid the overflow of receiver buffers. The rationale is that transmission is only allowed if the buffers have capacity to accept incoming packets. This feature is implemented by sending special control packets from the receiving node to the transmitting to indicate the status of the buffer.

Another feature of SpaceWire is the so-called *Group Adaptive Routing* that permits a dynamic selection of parallel links for transmission and allows fault-tolerant operation because if one of the links has a fault, alternative paths are available. Additionally, SpaceWire defines packets called *Time Codes* that are used for synchronizing the network with respect to a time master.

The features described make SpaceWire a fault-tolerant, scalable and high-performance communication protocol. Still, one of its limitations is the lack of a formal approach to real-time operation. Although different authors have proposed means to guarantee real-time behavior [67, 69], this is not covered in the original standard.

2.4 Discussion of Real-Time Communication Systems

Emerging applications are likely to impose ambitious requirements on real-time communication protocols in embedded systems. Among the many challenges one may have to deal with, it is possible to list, for instance, adaptivity, efficiency, complexity management, and mixed criticality. Meeting all these stringent demands is a non-trivial problem, especially because peculiarities of applications may render some solutions inappropriate. On the other hand, it is possible to ponder the characteristics of a versatile communication scheme to cover many aspects of the requirements presented. A list of desired protocol characteristics and respective justifications is proposed below:

Mixed Hard-Real-Time and Best-Effort Behavior Protocols that support either one or the other approach, but not both, may have limited applicability. While hard-real-time behavior is the basic requirement on many embedded systems, especially with control applications, non-real-time communication is useful for other functions such as monitoring, logging, updates, configuration, and LAN connection. As embedded systems get more and more complex and heterogeneous, the need for communication protocols supporting both approaches concurrently may be more common. This functionality is explicitly supported, for example, by FlexRay, PROFINET, and TTEthernet. On the other hand, TTP/C is an example of a protocol that was designed mainly for hard-real-time communication (although mapping best-effort communication in the reserved time slots is possible). CAN and SpaceWire belong to the class of protocols used in real-time applications that requires, however, a careful design of the applications in order to guarantee deterministic behavior.

Dynamic Reconfiguration Hard-real-time systems are commonly designed to perform pre-defined functions that will not be altered during the life-cycle of a product. In this kind of applications all real-time communication and processing needs are defined and analyzed during the design phase (static approach). However, new applications such as flexible industrial plants or transportation systems (see Chapter 8) can profit from, or even require, a dynamic reconfiguration of the real-time communication. Commercially available hard-real-time communication systems usually do not allow for dynamic reconfiguration, but academic projects are proposing ways of doing so, such as the Flexible Time-Triggered approach [70].

Performance and Efficiency Low-latency and efficient bandwidth utilization allow using a communication protocol in a greater variety of applications. Some distributed control applications, for instance, require sampling rates of more than 1 kHz and latency times in the range of microseconds. Efficiency is also an issue since bandwidth in embedded systems is normally constrained. Therefore, communication overhead must be minimized and medium access schemes should avoid unnecessary idle times.

Multi-master Capability Some protocols (e.g., EtherCAT) rely on a master/slave scheme to control communication. Although this approach simplifies the design, one drawback is that a faulty master potentially disrupts communication. For safety-critical applications such as *X-by-wire* and flight control systems where modules are replicated and a limited number of faulty nodes must be tolerated, a multi-master communication approach may become a necessity. TTP/C, FlexRay, SpaceWire, CAN, and AFDX are examples of multi-master protocols.

Distributed Clock Synchronization A common notion of global time is a requirement for many embedded systems. It allows, for instance, coordination of different tasks and determination of the order of events. Moreover, it has been proved [85] that consensus in a distributed system can be reached if, for example, it is synchronous and the communication delay is bounded. Consensus is an important service in distributed systems since it allows nodes to agree on the same data and actions.

Fault-Tolerant Operation The resilience against faults is an important characteristic of communication protocols. Redundancy, for instance, is a common approach to fault-tolerance since by meticulous design the availability of a system can be extended by means of replication. Another way of implementing fault-tolerance is by using a filtering technique. An example are the bus guardians in TTP/C that protect the data bus against the so-called *babbling idiot* failures [57], which occur when a node sends messages indiscriminately. The CAN bus, for example, has no such mechanism and a *malicious* node can jeopardize all data communication. Distributed high-integrity systems are usually designed to cope with arbitrary (or Byzantine [59]) failures, which require at least $3.k+1$ nodes to tolerate k faults.

Support for Composability and Scalability According to Kopetz's definition [57], *"An architecture is said to be composable with respect to a specified property if the system integration will not invalidate this property once the property has been established at the subsystem level"*. In practical terms, this concept means that the properties of smaller components such as timeliness will persist after being integrated into larger subsystems. Without composability, building complex systems becomes a hard or even impossible task. Scalability refers to the capability of increasing load, size, bandwidth, and other parameters of a functional unit without disproportionately increasing the effort to cope with the higher demand.

Flexible Topologies Using communication protocols that support hybrid topologies can potentially facilitate the design of distributed systems. Hybrid topologies give designers more options to accommodate data links, especially in applications with tight physical constraints. SpaceWire is an example of a flexible protocol when it comes to topologies since it employs point-to-point connections so that different arrangements can be built.

Resource Efficient Implementation Given that the scope of this discussion is the embedded domain, available resources such as memory, processing power, chip area, and bandwidth are normally scarce. Therefore an important characteristic of a communication protocol tailored to this area is an efficient and resource-saving utilization. In automotive applications, for instance, communication controllers are usually integrated into the microcontroller die.

The following chapters of this thesis will present and evaluate a new communication protocol that was designed bearing in mind the above requirements. With the freedom to build up a new protocol from scratch it was possible to take decisions about implementation trade-offs and define an architecture to meet the envisioned needs.

Chapter 3

The TrailCable Communication Protocol

This chapter presents the TrailCable hard-real-time data communication protocol which is aimed at building reliable and flexible distributed embedded systems. The motivation to develop a new communication protocol is due to the fact that although buses and networks for reliable hard real-time applications exist, they usually have limited flexibility and restricted support for dynamical reconfiguration, if at all. To overcome these limitations, the rationale of the TrailCable protocol is based on the sporadic-triggering paradigm, which is introduced in the following.

3.1 The Sporadic-Triggering Paradigm

Event- and time-triggered paradigms are usually considered as two opposite approaches for building computing systems. It can be assumed, however, that time-triggered systems are actually a proper subset of the more general event-triggered approach. In event-triggered systems ($\mathcal{ET}$) there is no rule whatsoever for the events to occur and tasks can be released at any time. On the other hand, time-triggered systems ($\mathcal{TT}$) are characterized by a strict rule: all actions must be triggered at globally known times. This rule implies that $\mathcal{TT} \subsetneq \mathcal{ET}$ and in practice it imposes a huge gap between event- and time-triggered implementations, since global time synchronization and statically defined scheduling tables are required for the latter.

While the characteristic of the time-triggered paradigm may facilitate building up reliable and deterministic systems, one of its drawbacks is the lack of flexibility. In her book *Real-Time Systems* [61], Jane Liu lists some of the disadvantages of the clock-driven approach. The following text is an excerpt of her book:

1. *The release times of all jobs must be fixed. In contrast, priority-driven algorithms do not require fixed release times. (...) we can guarantee the timely completion of every job in a priority-driven system as long as the inter-release times of all jobs in each periodic task are never less than the period of the task. This relaxation of the release-time jitter requirement often eliminates the need for global clock synchronization and permits more design choices.*

2. *In a clock-driven system, all combinations of periodic tasks that might execute at the same time must be known a priori so a schedule for the combination can be precomputed. This restriction is clearly not acceptable for applications which must be reconfigurable on-line and the mix of periodic tasks cannot be predicted in advance. (...) a priority-driven system does not have this restriction. We can vary the number and parameters of periodic tasks in the system provided we subject each new periodic task to an on-line acceptance test.*

3. *The pure clock-driven approach is not suitable for many systems that contain both hard and soft real-time applications. (...)* [p.112]

This thesis presents and relies on the so-called *sporadic triggering* ($\mathcal{ST}$) concept that can be used as an alternative to overcome $\mathcal{TT}$ limitations and reduce the gap between event- and time-triggered systems. In sporadically triggered systems two consecutive activations of a given job must be separated by a minimum time interval, the task period [1].

In many applications, the sporadic-triggering approach is a good compromise between event- and time-triggered systems. The reason for this is that the sporadically-triggered paradigm, as opposed to the event-triggered approach, makes it possible to ensure hard-real-time behavior while offering a greater flexibility and independence between tasks than do time-triggered implementations. However, in order to fully exploit the capabilities of the sporadically-triggered paradigm, a dynamic scheduling algorithm becomes necessary to assign active tasks to a given resource. This is more complex than the use of static scheduling tables in the time-triggered approach.

Since all of their activations are periodic, time-triggered systems are also sporadically-triggered. However, not all sporadically-triggered systems are also time-triggered due to the fact that the former do not require a synchronized operation and their job activations can happen any time as long as the minimum time interval between them is larger or equal to the specified period. Thus, the relation between the three described paradigms is represented by Figure 3.1 and can be expressed as: $\mathcal{TT} \subsetneq \mathcal{ST} \subsetneq \mathcal{ET}$.

[1] Within the real-time research community, the word *sporadic* is used to describe tasks with a minimum, predefined inter-arrival time. This is the interpretation throughout this thesis, but care must be taken because such a term can be misleading for a broader audience, since it can be inferred that a certain event happens only in an isolated or unruly manner.

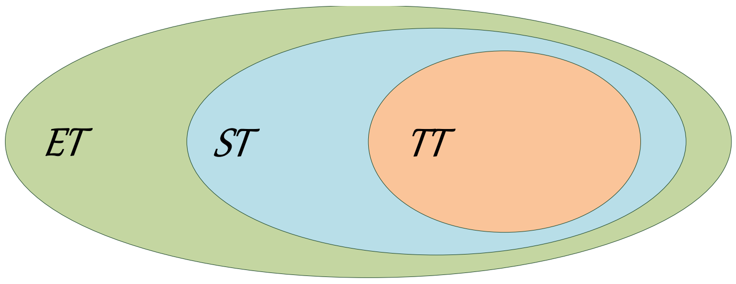


Figure 3.1: Triggering paradigms

The increased flexibility of the $\mathcal{ST}$ paradigm does not impose a limitation to building dependable systems. A completely synchronous operation is not required for this purpose as long as the maximum time intervals for processes to complete are bounded. The *Timed Asynchronous Distributed System Model* by Cristian and Fetzer [28] shows that if all processes have a defined time interval to respond and nodes have access to hardware clocks, it is possible to implement distributed services such as clock synchronization, membership, atomic broadcast and consensus.

The TrailCable protocol to be presented in this chapter is based on the $\mathcal{ST}$ paradigm and is therefore not only suitable for hard-real-time use but also flexible enough to allow, for example, dynamic reconfiguration of communication tasks.

3.2 TrailCable Protocol Rationale

The infrastructure of a TrailCable network consists of point-to-point, wired communication links. In order to reduce cabling complexity, cost, size, and weight, nodes are able to act as data switches, forwarding data between neighbors. Although wireless networks have a lot of benefits, wired links can offer an overall better communication reliability since its transmission medium is less prone to external interferences when compared to radio counterparts. Moreover, the performance characteristics of a wired link do not deviate much, while wireless communication is heavily dependent on spectrum utilization, distance between nodes, etc.

A typical node in a TrailCable network is depicted in Figure 3.2. It consists of a host processor with I/Os to interact with physical processes and the communication layer. The latter is divided into the communication engine, which is the autonomous hardware component that handles packet switching and real-time scheduling, and the communication interface, which acts as an abstraction of the communication services provided by the protocol for the host processor.

Real-time communication is established by means of virtual channels that are mapped on the network infrastructure. To guarantee real-time behavior, data transmission in

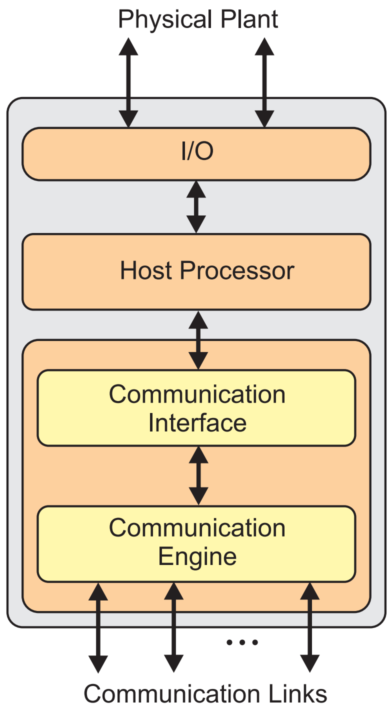


Figure 3.2: The TrailCable node

each link direction is controlled by a scheduler which is responsible for assigning the transmission medium to a given channel. The scheduler algorithm of the TrailCable is *Earliest Deadline First* (EDF) [60] and was chosen because it allows high utilization of the communication resources and is well suited for coping with dynamic configuration. For the schedulers, virtual real-time channels are considered as communication tasks. A communication task (in this thesis also simply referred to as task) represents the process of transmitting to the communication link one complete data packet of a given real-time channel within a specified deadline.

The point-to-point links are resources with limited capacity, and therefore a feasible schedule is necessary to dynamically assign the available bandwidth to different, concurrent communication tasks. In order to prove feasibility, for each channel the following parameters must be provided: minimum inter-arrival time, packet size, and channel deadline. The arrival rate and the packet size parameters are the same for all point-to-point communication links that make up a channel.

To better exploit bandwidth, the protocol also supports non-real-time communication. The scheduling algorithms are able to guarantee that non-real-time traffic will not interfere with the hard-real-time communication. This communication concept has some similarities with the work by Kandlur, Shin and Ferrari [51, 50]. We extend that approach by adding packet preemption, a bandwidth guardian for fault-tolerance, and a new clock synchronization technique, all of them implemented in hardware at the communication-protocol level. Moreover, the TrailCable approach has a different channel establishment and scheduling methodologies. By using packet preemption, the restriction set by [51], which implies that only small data packets can be used for real-time traffic, can be relaxed. This can decrease the transmission overhead and simplifies the implementation as the messages do not have to be segmented into small packets prior to transmission. Implementing fault tolerance and clock synchronization mechanisms at the communication level also reduces the complexity of the application. In addition, fault-tolerance mechanisms that work independently of the host software can be more robust and are able to reduce communication latency when forwarding data packets. It has been shown (e.g. [56]) that clock synchronization implemented in hardware allows a more accurate time reference.

3.3 Virtual Real-Time Communication Channels

The TrailCable network infrastructure can be described as an undirected graph $G := (V, E)$, where V is the set of vertices and E the set of edges that connect those vertices [29]. In our case the vertices are computing nodes and the edges are bidirectional electrical or optical communication links. The degree of a node corresponds to the number of its active links. The maximum degree of a node therefore is limited by the number of communication ports a node has.

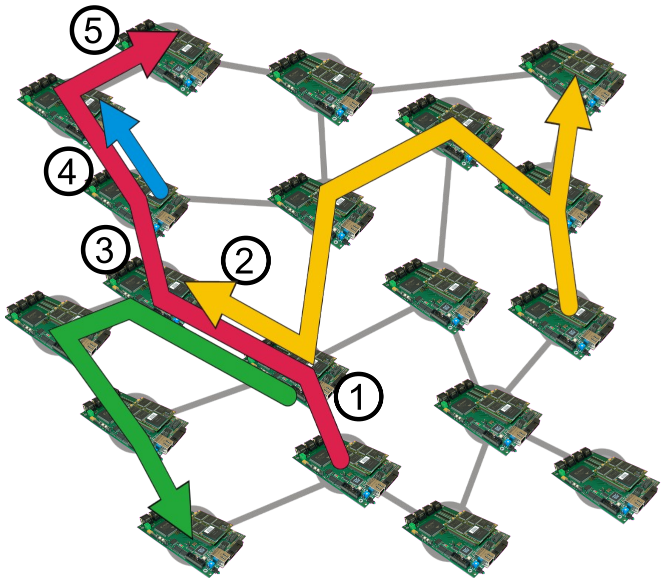


Figure 3.3: Network infrastructure

Once a network infrastructure is defined, it is possible to map real-time channels on it. A real-time channel is a logical tree (over which data is periodically transmitted from one origin to one or more destination nodes in the network) with the corresponding packet and time characteristics. Figure 3.3 shows an example where four real-time channels are mapped on a communication infrastructure.

The end-to-end connections between the origin and one destination are called paths or routes and are defined by $P := (V, E)$, where $V = x_0, x_1, ..., x_k$ and $E = x_0x_1, x_1x_2, ..., x_{k-1}x_k$ for distinct $x_n{}'s$. The source node is represented by x_0 and the path sink by x_k.

As long as the real-time constraints can be met, a single data link can be shared by concurrent real-time channels. In order to identify which real-time channel a data packet belongs to, the packet header contains a unique identifier (ID field). However, it is important to note that the same real-time channel can have different IDs for different links. This is possible because the intermediate routing nodes are able to alter the IDs of an incoming packet to another predefined one. This procedure can make it easier, for example, to reconfigure the system at run-time.

Because the system is able to route multiple incoming real-time channels (that come from either one or multiple input links) to the same output link, a communication scheduler for each output port is indispensable. As mentioned, the scheduling strategy chosen for the TrailCable protocol is EDF. One of the reasons is that the maximum bounds of the data-link utilization can be improved as compared to other scheduling algorithms, such as rate-monotonic [22]. Moreover, it is shown in this chapter that EDF can be employed by all nodes of a certain real-time channel without the need for explicit synchronization between nodes. The information a node requires for assigning new absolute deadlines for the scheduling of data packets is obtained by analyzing the traffic characteristics of previous packets. The communication engine of the TrailCable is also able to cope with preemptions, which are required by the EDF algorithm, both at the low-level data transmission layer and at the scheduler and dispatcher components. Thus, instead of using the common approach to EDF-based communication, which implies a segmentation of large packets into smaller ones that are individually scheduled, this work relies on packet preemption.

Since we are dealing with hard real-time communication, timing requirements and route characteristics for all channels must be known a priori in order to assure the feasibility of a given configuration. The required schedulability analysis must be executed before any real-time channel is altered or added to the system. During run-time, the characteristics of each task are also needed by the communication engine and stored in data structures that we call scheduling tables. Real-time channels can be added, removed, and reconfigured on-line in a dynamic manner, but on condition that acceptance tests are successful.

To describe the timing characteristics of a channel, a real-time communication task is assigned to each node along this channel. The main parameters (Figure 3.4) that describe a real-time communication task i for a given node are the following:

- *Release time ($r_{i,j}$)* - the latest possible instant in which the data packet of the j-th instance is ready to be transmitted

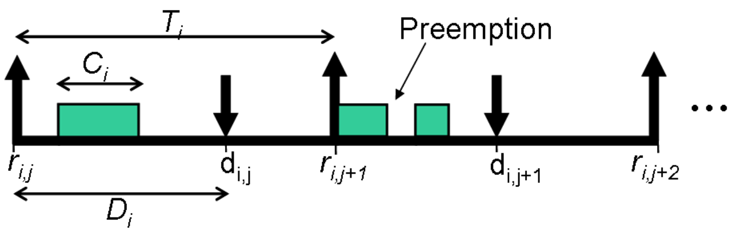


Figure 3.4: Parameters of a real-time task

- *Period (T_i)* - minimum time interval between two consecutive release times at the source node
- *Relative deadline (D_i)* - the interval after the release time in which the data transmission must be completed. The absolute deadline is different for each task instance and represented by $d_{i,j}$
- *Computation (or transmission) time (C_i)* - time needed to transmit a data packet. The value of C_i depends on the amount of data bytes to be transferred and on the transmission rate. The parameters T_i and C_i are equal in all nodes that build a real-time channel whereas D_i can be different for each link.

In order to reduce the communication latency and thus the minimum achievable channel deadlines, the virtual cut-through method was chosen to forward data packets. So, after receiving a packet header and retrieving the ID (which allows retrieving the corresponding routing information contained in the scheduling tables) the packet can be handed over to the scheduler immediately, without the need for waiting for the whole packet to be received. So, if we do not consider the propagation time of the communication links and the intrinsic processing delays of the routing nodes, the release times from the second hop onwards can be expressed as a function of the absolute deadline of the previous communication link as follows:

$$r^n_{i,j} = d^{n-1}_{i,j} - (C_i - \alpha) \tag{3.1}$$

where $d^{n-1}_{i,j}$ is the absolute deadline of the previous communication link and α the time to transmit the packet header.

The timing diagram of Figure 3.5 shows one scheduling example based on the network infrastructure of Figure 3.3. The numbers on the left-hand side of Figure 3.5 associate the time diagrams with the links of Figure 3.3. It can be noted that the release times at the intermediate and final nodes (2 to 5) are $(C_i - \alpha)$ time units advanced with respect to the previous absolute deadline. As also shown in Figure 3.5, a packet transmission, however, can start even before the actual release times when the communication link is idle. This reduces the average channel latencies and is also a requirement for the run-time admission control presented in Section 3.6.

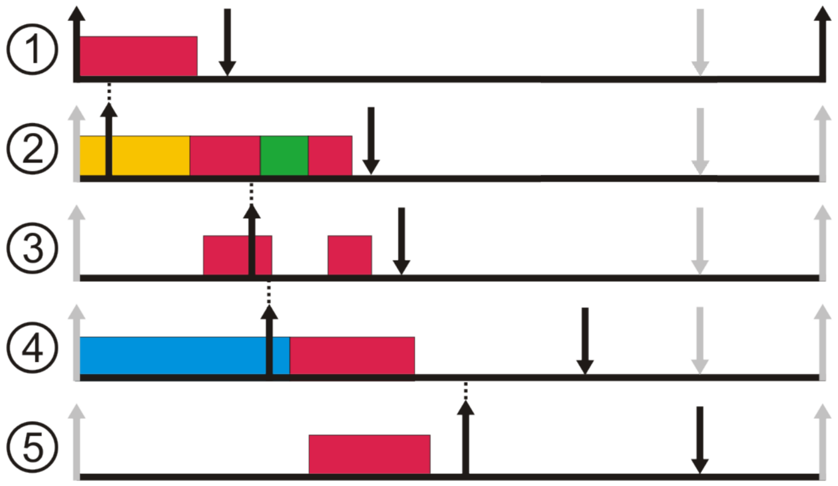


Figure 3.5: Scheduling example

In Section 3.5 we will present the methodology we use to check whether a set of communication tasks for a given communication port is feasible. Even if the data communication at every single communication port can be proven feasible by that methodology, it is still necessary to check whether the end-to-end channel deadlines are met as well. To do this, we first need to introduce two new parameters (Figure 3.6) that must be known for such an analysis: *prop* and fw.

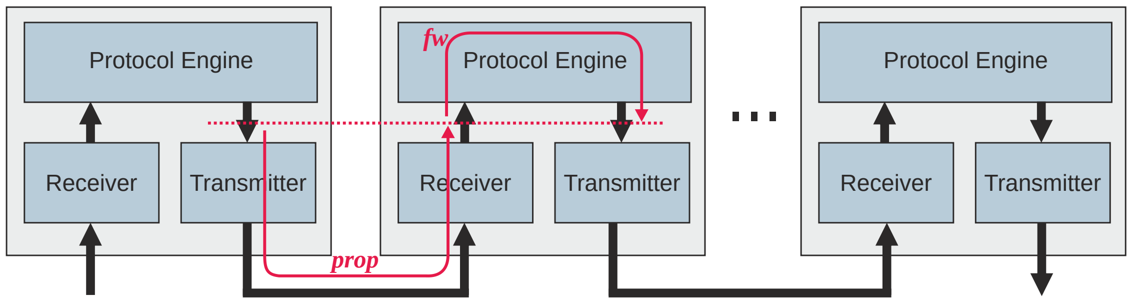


Figure 3.6: Packet-forwarding timing parameters

The term $prop^n$ is the line-propagation delay from the point in time where a single byte is written into the transmitter of a node n to the point where the same byte can be read at the receiver of a neighboring node $n+1$. Although the time necessary to transmit a packet is fixed, $prop^n$ can still vary slightly, depending, for example, on cable lengths. The next term, fw^n, represents the worst-case duration for forwarding a single byte from the receiver of a node n to one of its transmitters, under the assumption that no other tasks be active in the forwarding node. The parameter fw_n is fixed for a given implementation of the protocol. The term, D_i^n, is the relative deadline in the n-th node of a real-time channel i. Different tasks of a channel i can have different relative deadlines.

The total end-to-end channel latency is determined by summing up the latencies added by each involved node, which consists of its *prop* and fw delays and the relative deadline

D_i. However, thanks to the use of the virtual cut-through forwarding method, it is still possible to subtract, for each intermediate node, $(C - \alpha)$ from the total channel latency. It follows that in order to guarantee that the deadline of a channel i (δ_i) be met, the following condition must be satisfied:

$$\delta_i \leq \sum_{n=0}^{k-1} (prop^n + fw^n + D_i^n) - (k-1)(C_i - \alpha) \tag{3.2}$$

where k is the number of nodes in the channel route from the source to a destination, including all intermediate hops. Meeting this condition for all paths of all communication channels is a necessary step to assure the feasibility of a TrailCable communication network.

3.4 Preemption Mechanism

Besides being a requirement for the classical EDF scheduling, packet preemption has recently been studied by researchers looking for a way to improve time-sensitive communications, especially in the Internet domain. But there have not been many initiatives to adopt such mechanisms with hard real-time communication networks for embedded applications, such as in data buses for mechatronic systems.

The TrailCable packet-preemption mechanism relies on comma-control words to indicate whether the packet being received is starting or being resumed. A comma-control word consists of a unique sequence of bits at the data-link layer that, once detected, can be used for alignment purposes. Commas are supported by some encoding schemes such as the 8B/10B [87]. As the name suggests, the 8B/10B coding method converts 8 data bits into 10 coded ones for transmission and implies, therefore, a 25 % overhead. The benefit of this coding scheme, however, is that it provides a manner to align the receiver with the transmitter, guarantees a maximum of 5 consecutive equal bits in the encoded data stream, eliminating the DC-level at the link layer (which facilitates the hardware design) and provides control characters that can be sent instead of data symbols. Two different control characters called K28.5 and K28.1 are commas in 8B/10B and make it possible for the receiver of the TrailCable protocol to recognize *START* and *RESUME* headers. The functions of these headers are the following:

- *START* - used to indicate the beginning of a new packet. When the communication link is idle, a sequence of *START* commas is also constantly transmitted to allow the receiver to be kept synchronized with the transmitter.

- *RESUME* - indicates that a previously preempted packet is being resumed.

Each real-time packet is composed, at least, by a *START* header and a data section with variable length. The headers described contain, besides the comma, the packet ID (by which the routing and scheduling information can be retrieved from the scheduling table) and finally the CRC of the packet ID. The reason for protecting the ID by a CRC is that a transmission fault could otherwise alter the correct ID, which would lead to potentially wrong scheduling and routing decisions. The data section is composed by a payload, its CRC, a timestamp (used for clock synchronization), and also the respective CRC (Figure 3.7). During transmission of the data packet over a real-time channel the payload CRC must be kept unchanged. On the other hand, the ID can be altered during the forwarding process and consequently the ID-CRC field as well. The same occurs for the timestamp section. The CRC polynomials were selected on the basis of the guidelines presented in [55].

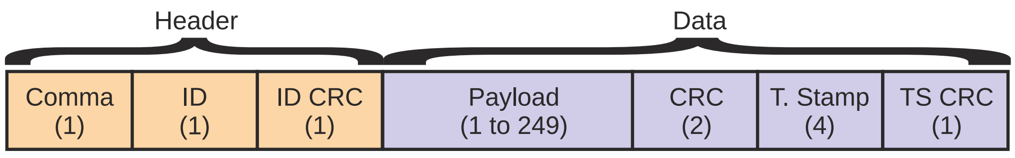


Figure 3.7: Format of a real-time packet

A header must always be sent atomically, but the data section can be preempted at the end of each single byte. When a preemption occurs, the context of the previous task is saved and the new task starts. As soon as the task with the highest priority is transmitted, a preempted task can be resumed by sending a RESUME header. This method allows the same packet to be preempted more than once in different instants of time and multiple packets to be preempted in a row. A preemption example can be seen in Figure 3.8.

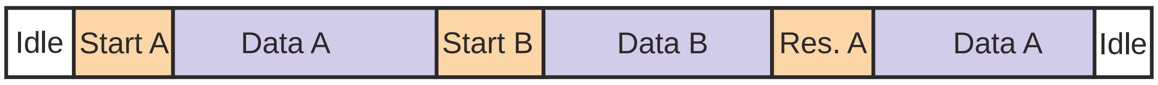


Figure 3.8: Packet preemption

One further advantage of using the 8B/10B coding scheme in the TrailCable protocol is that even if the receiver loses synchronization with the transmitter at a given time due to transient communication errors, detecting a new comma is sufficient to reestablish the synchronization. Therefore, faulty packet segments do not impair any subsequent, correct ones, thus enhancing communication robustness and reducing masquerading failures.

The scheme presented imposes no additional overhead for the high-priority tasks that preempt lower-priority ones. This is due to the fact that the amount of bytes sent by a high-priority task will be the same regardless if it started when the link was idle or by preempting another one. On the other hand, when tasks resume after being preempted by another one it is necessary to resend a header thus increasing overhead. Despite of this, using preemption allows a reduction of channel latency times and has, in average, less overhead than scheduling smaller, segmented data packets since each segment also has its own header.

For sensitive applications where hardware area and memory size are a major concern, disabling preemption can save some resources. In this case it would not be possible to use the original EDF scheduling, but there is a variant of this algorithm called npEDF (non-preemptive EDF) that may become an alternative. Although not an optimal algorithm as the EDF, npEDF also has some of the benefits of the former. Short [80] gives details about npEDF and shows that it is well suited for resource-constrained applications. In our work, however, we will concentrate on the classical EDF and benefit from preemptions since the amount of hardware resources required for this function is low thanks to an optimized architecture (described in Chapter 6) and because communication overhead is also kept low since only three bytes are required in a *RESUME* header.

3.5 Schedulability Analysis

This section presents a method for performing the schedulability analysis of each communication port scheduler presented in the network. The objective is to assure that by means of the EDF algorithm all schedulers will be able to deliver the data packets that reach a given communication port within the specified deadlines.

The clock drift between two consecutive nodes is an important effect to be reckoned with when scheduling tasks in a distributed system. If a certain set of communication channels uses the entire bandwidth available, a problem occurs if the oscillator at the sender is slightly faster than the one at the receiver. In this situation the sender can occasionally produce more bits than the receiver is able to handle correctly. Therefore it is necessary to scale the value of the period and deadlines that describe the behavior of the real-time tasks. For this purpose we use the deviation Δ as a factor that corresponds to the ratio between the lowest and highest clock oscillators used for the TrailCable engine in a network. As an instance, using 50 ppm clock oscillators, two nodes are expected to deviate a maximum of 100 ppm from each other and therefore the Δ factor would be 0.99990. The values of T_i and D_i employed for both the acceptance tests and scheduling tables must be then scaled from the nominal ones as follows: $T_i = T_{i(nom)} \times \Delta$ and $D_i = D_{i(nom)} \times \Delta$.

At first sight, the effect of Δ would seem negligible, but consider for example a task relative deadline $D_{i(nom)} = 50ms$, $\Delta = 0.99990$ and a transmission rate of 32 Mbps. In this scenario, the difference between the nominal and corrected deadlines corresponds to the time required to transmit 20 bytes. This could potentially cause deadlines to be missed if the network operates under heavy load. Moreover, since TrailCable is hard-real-time, all details that could possibly impair schedulability must be taken into account and mitigated.

The transmission time C_i introduced in Section 3.3 is necessary for the schedulability analysis and calculated by adding to the payload 10 bytes (3 for the START header, 2 for the payload CRC and 5 for the timestamp) and multiplying them by the transmission time of a single byte:

$$C_i = (\text{payload} + 10) \times \frac{1}{\left(\frac{\text{baud rate}}{8}\right)} \tag{3.3}$$

Another necessary compensation is due to the overhead caused by the *RESUME* headers. Each time a task is preempted, a new *RESUME* header must be sent thus increasing the actual packet transmission time C_i. For this reason, before executing the schedulability analysis, we need to calculate the maximum transmission time $Cmax_i$ for each task i in a certain communication port by adding the respective overhead costs to C_i, as expressed by Eq. (3.4):

$$Cmax_i = C_i + \alpha \times P_i \tag{3.4}$$

The constant α is the transmission time for a *RESUME* header and the variable P_i represents the maximum number of times a task i can be preempted by others.

The classical EDF scheduler needs to sort absolute deadlines to determine the highest-priority task. Absolute deadlines are calculated at the arrival of each new task by adding its relative deadline value to the current time [2]. So, if a task i is active in the scheduler, a task m arriving immediately thereafter can only preempt i if $D_m < D_i$. Otherwise, d_i will be always smaller than d_m. Moreover, under these circumstances a task i can be preempted by m only once during the period T_m. It follows that a task m can preempt i no more than D_i/T_m times if $D_m < D_i$. The variable P_i is the total number of times i can be preempted with a set of t tasks. P_i is a conservative estimate, i.e., the number of preemptions can actually be lower than the calculated amount.

$$\mathcal{M} = \{D_m | D_i > D_m, \quad 1 \leq m \leq t\} \qquad P_i = \sum_{\forall x \in \mathcal{M}} \left\lceil \frac{D_i}{T_x} \right\rceil \tag{3.5}$$

With the clock and preemption compensations performed, it is possible to use standard approaches for performing the schedulability analysis of an EDF task set. More specifically, since we are dealing with the special case where deadlines are less than periods, we can apply the processor demand criterion by Baruah, Rosier, and Howell [17], which is described by the following theorem presented in [21]:

[2]With TrailCable, the method is slightly different in order to cope with the jitter introduced by previous schedulers. This method is described in Section 3.6.

Theorem. *If* $\mathcal{D} = \{d_{i,k} | d_{i,k} = kT_i + D_i, \quad d_{i,k} \leq \min(Bp, Hp), \quad 1 \leq i \leq n, \quad k \geq 0\}$, *then a set of periodic tasks with deadlines less than periods is schedulable by EDF if and only if:*

$$\forall L \in \mathcal{D} \qquad L \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i \tag{3.6}$$

This theorem consists in checking for all tasks and all instances whether the absolute deadlines can be met in the available processing time (or available bandwidth in our case) up to that point. The time interval for which these checks must be performed is bounded by the hyper period H_p, or by the busy period B_p, of the task set. The busy period of an interval $[0, L]$ is the minimum time necessary to complete the execution of all released tasks, which is defined as a quantity $W(L)$ and given by [21]:

$$W(L) = \sum_{i=1}^{n} \left\lceil \frac{L}{T_i} \right\rceil Cmax_i \tag{3.7}$$

The busy period of an interval $[0, L]$ must be equal to $W(L)$ and is therefore defined as $B_p = \min\{L \mid W(L) = L\}$. The busy period is calculated by Algorithm 1 also presented in [21].

Algorithm 1 busy_period

```
1: $L \leftarrow \sum_{i=1}^{n} Cmax_i$
2: $L' \leftarrow W(L)$
3: $H_p \leftarrow lcm(T_1, ..., T_n)$
4:
5: while $(L' \neq L)$ and $(L' \leq H_p)$ do
6:     $L \leftarrow L'$
7:     $L' \leftarrow W(L)$
8: end while
9:
10: if $(L' \leq H_p)$ then
11:     $B_p \leftarrow L$
12: else
13:     $B_p \leftarrow \infty$
14: end if
```

If the inequality (3.6) holds for $\forall L \in \mathcal{D}$, the task set is feasible for an EDF scheduler and the schedulability analysis for the given node is completed. The drawback of this method, however, is its complexity, since it is exponential. Nevertheless, in practice the schedulability analysis can be executed in reasonable time for many task sets, especially when the network is not working near its maximum capacity.

More recently, Albers and Slomka [11] presented improved algorithms for EDF schedulability analysis for deadlines different from periods. The methods are still exact and have pseudo-polynomial complexity, but outperform previous test algorithms.

Another possibility to cope with the complexity of schedulability analysis for EDF with deadlines less than periods is to sacrifice exactness in favor of lower complexity. Masrur, Drössler, and Färber [64] presented two sufficient algorithms that achieve polynomial complexity, namely $O(n^2)$ and $O(n\ log\ n)$.

3.6 Run-Time Admission Control

The admission control for the communication engine presented in this section is a key component for guaranteeing the hard-real-time constraints during run-time. It is responsible for checking whether the incoming packets in the nodes conform to the specifications employed for the schedulability analysis. As a consequence, the run-time admission control acts like a bandwidth guardian that can filter traffic anomalies. Such procedure aims at increasing the robustness and fault-tolerance of the real-time scheduling. If, for example, a faulty node sends a task with shorter periods or with increased payload size, the assumptions made initially for the schedulability analysis will not hold anymore. This undesired situation could make not only the mentioned task, but potentially all others miss their deadlines. To solve this problem the built-in bandwidth guardian individually filters possible overload conditions for each task, thus acting as a “temporal firewall” that guarantees that all correct tasks be correctly scheduled even in the presence of faults.

Additionally, the run-time admission control is also responsible for triggering the schedulers with the appropriate absolute deadline for each new task instance. This is required since one of the goals of the communication engine is to achieve a distributed scheduling without the need for global clock synchronization. Indeed, the method we use even dispenses with the transmission of explicit control information to synchronize the schedulers. In order to cope with such requirements, the admission control block analyses the incoming packets and, for every new task instance, calculates the absolute deadline that is then used by the scheduler(s) of the defined output port(s).

As will be shown below, the procedures for implementing the bandwidth guardian and calculating the deadlines have a lot in common. For this reason, both functionalities were integrated into the run-time admission control module.

In order to set up the bandwidth guardian for a communication port, it is necessary to define time intervals where data transmission is not allowed to occur. To do this, the earliest and latest possible instants of the release times must be known. Without loss of generality, let us assume that the transmission of a task at the source node be triggered periodically at a fixed time interval between any two instances. In this situation the release times at the originating node ($r^0_{i,j}$) will have no phase variation. On the other hand, from the second node of the route onwards, phase variations eventually occur due to the behavior of the scheduler. In the best case scenario, a data packet will immediately be transmitted from one node to the next if there are no higher-priority packets waiting for transmission. Due to virtual cut-through routing, the forwarding of packets can be accomplished as soon as the *START* header is received. This is illustrated in Figure 3.9 by the dotted up arrows. On the other hand, the worst-case scenario occurs if tasks are completed exactly at their deadlines, when the release times are given by Eq. (3.1). It can be seen that there are three different areas in the figure 3.9. The green area represents the sector where data packets are received before the worst-case release time. Then, the yellow sector represents the time after the worst-case release time, within which packet transmissions must be completed to meet their deadlines. Finally, the red sector indicates where packets of a certain task are not expected to be transmitted.

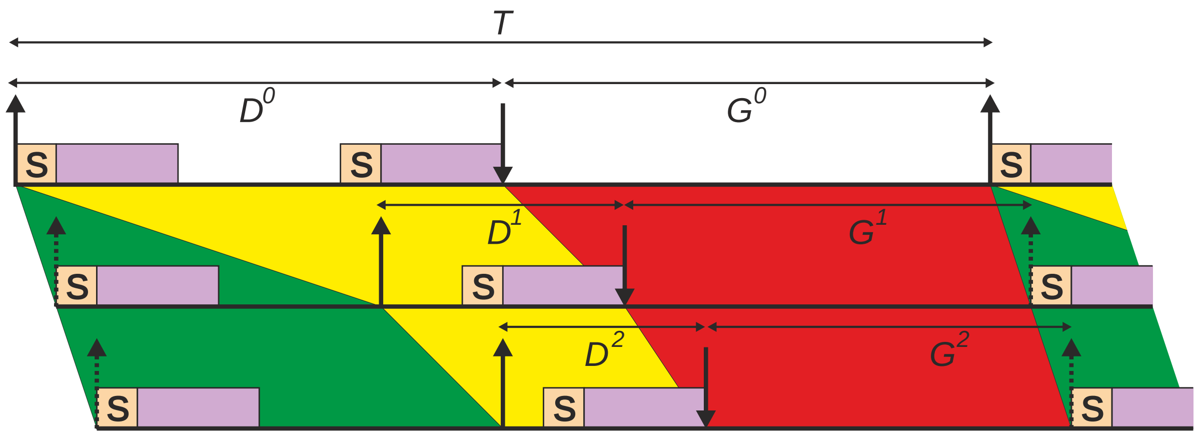


Figure 3.9: Guard intervals

The release times shown in Figure 3.9 follow a pattern that can analytically be defined as:

$$
\begin{array}{ll}
r^0_{i,j_{min}} = r^0_{i,j_{max}} = \Phi_i + j.T_i & \\
r^1_{i,j_{min}} = r^0_{i,j} + \alpha & r^1_{i,j_{max}} = r^0_{i,j} + D^0_i - (Cmax_i - \alpha) \\
r^2_{i,j_{min}} = r^0_{i,j} + 2.\alpha & r^2_{i,j_{max}} = r^0_{i,j} + D^0_i + D^1_i - 2.(Cmax_i - \alpha)
\end{array}
$$

The next step is the definition of a guard zone, defined by G^n_i (Figure 3.9), which is the minimum interval between completion of a data packet transmission and start of the transmission of the next instance. The constant G^n_i is expressed by:

$$G^n_i = T_i - D^n_i - (r^n_{i,j_{max}} - r^n_{i,j_{min}}) \tag{3.8}$$

In a generic form, it follows that the guard value throughout a path is given by:

$$\begin{aligned}
G_i^0 &= T_i - D_i^0 \\
G_i^1 &= T_i - D_i^1 - (D_i^0 - Cmax_i) \\
G_i^2 &= T_i - D_i^2 - (D_i^0 + D_i^1 - 2.Cmax_i) \\
&\vdots
\end{aligned}$$

It can be seen that the values of G_i^n are dependent on the characteristics of the previous nodes, namely their relative deadlines. On the other hand, the remaining parameters are not dependent on previous nodes but it is worth recalling that T_i and $Cmax_i$ are equal in the entire real-time channel.

In the TrailCable protocol there is no explicit synchronization among schedulers. The information a node requires to calculate the absolute deadlines for each new task instance is gained entirely from the incoming traffic. By this each node acts as a fault-containment region with respect to scheduling errors. This means that anomalies that could potentially cause scheduling problems are detected early enough and filtered out. Moreover, this approach has the benefit that no communication overhead is spent for controlling the schedulers and also that no global clock synchronization is required for the schedulers.

The analysis of the incoming packets to determine if they are timely correct is performed with the task parameters used for the schedulability analysis and also with the guard values G_i^n. In the source nodes there is no need for a bandwidth guardian since there is only outgoing traffic of a given communication task. In the intermediate nodes of a channel, the bandwidth guardian becomes necessary and the absolute deadlines for the local schedulers must also be calculated. In the destination nodes, it suffices to execute only the bandwidth guardian (and no absolute deadline calculation) because data packets will not be forwarded to a neighbor.

Before going on in the explanation of the admission control method, we have to introduce the additional parameters needed by the process. Besides T_i, $Cmax_i$, D_i^n and $d_{i,j}^n$, $r_{i,j}^n$ and G_i^n, the following parameters of the j-th instance of a task i at a node n are also needed:

- $s_{i,j}^n$ *(start time)* - time when the reception of a *START* header begins,
- $res_{i,j}^n$ *(resume time)* - time when the reception of a *RESUME* header is completed,
- $f_{i,j}^n$ *(finish time)* - time when the reception of every single task byte is finished (not to be confounded with the time the whole transmission is completed).
- $cs_{i,j}^n$ *(corrected $s_{i,j}^n$)* - the start time added by the intervals where a packet was not transmitted due to preemption (Figure 3.10).

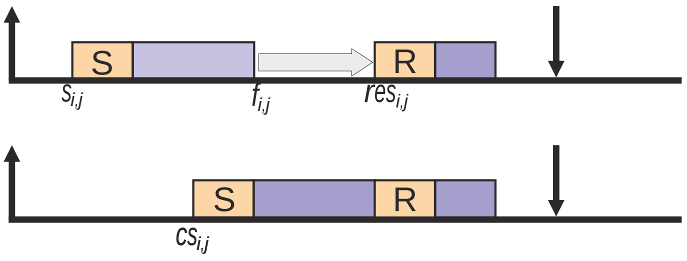


Figure 3.10: Admission control with preemption

The $cs^n_{i,j}$ parameter is updated when RESUME headers are received as follows:

$$cs^n_{i,j} = cs^n_{i,j} + res^n_{i,j} - f^n_{i,j} \tag{3.9}$$

Figure 3.11 illustrates the admission control method in the first intermediate node of a given route. When this node receives the first packet ($j = 0$) of a communication task i or the last instance was completed for a sufficiently long time (more than T_i time units from the current packet), it is possible to assume that the *START* header of the packet arrived at $\Phi_i = r_{i,0_{max}}$ and the absolute deadline for the output port scheduler is then calculated by adding D^1 to that instant.

The calculated deadline can be met since a prior schedulability analysis was performed and because $r_{i,j_{max}}$ will, in the worst-case, coincide with the critical instance of the task. The critical instant is the moment when the release of task causes the longest response time. With the preemptive EDF algorithm, the release time phases Φ of different tasks are not relevant for all deadlines to be met.

The second task instance is received at $\Phi_i + T_i + \Delta_1$. By this, it can now be assumed that the previous task was not received at $r_{i,0_{max}}$, but Δ_1 time units before it. Then, $r_{i,1_{max}}$ is taken as the moment when the *START* header of the second packet instance was received. Shifting the time references with respect to the previous instance will not impair the schedulability of the task set at the output port. This can be easily proven because if the separation of two task instances of i is greater than T_i, and the processor demand condition (3.6) be satisfied for T_i, it will also be for $T_i + \Delta$.

At the third instance we have the opposite situation, i.e., the interval between two instances of i is smaller than T_i. This is an indication that the current instance was received before $r_{i,2_{max}}$. If we continue with the same procedure of the last two instances it would not be possible to guarantee schedulability since in practice the task period would become smaller than T_i, which was used for the feasibility analysis. To solve this problem, we use a variable called hold, represented as H^n_i. It is used for two purposes. Firstly, it increases the guard interval in which no packets are allowed to be received. Secondly, it is employed to calculate the absolute deadline for the scheduler with respect to the time at which the *START* header was received from the previous node (cf. Figure 3.11).

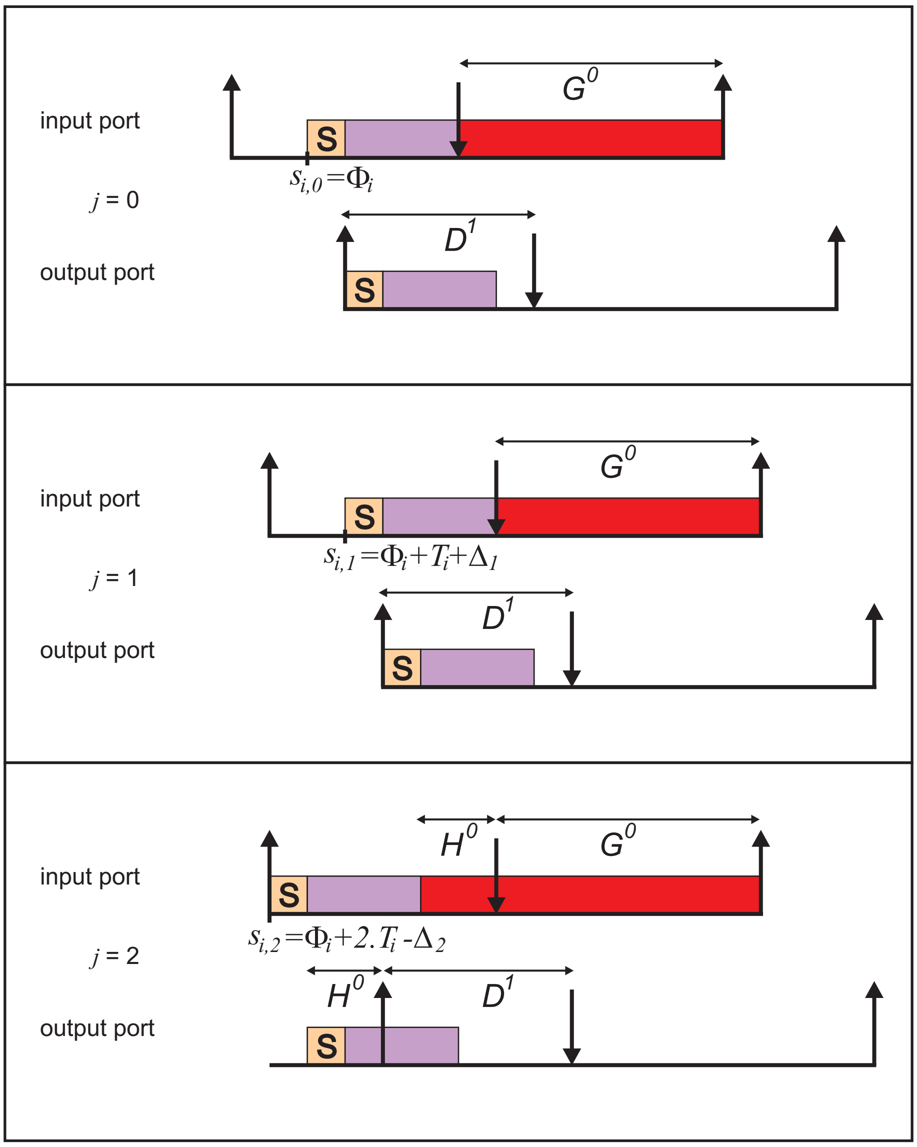


Figure 3.11: Hold Variable

The hold variable is only updated at the reception of *START* or *RESUME* headers accepted by the bandwidth guardian. The H_i^n variable is updated when a *START* header is received by:

$$H_i^n = H_i^n + T_i - (s_{i,j}^n - cs_{i,j-1}^n) \tag{3.10}$$

and also after a RESUME header by:

$$H_i^n = H_i^n - (cs_{i,j}^n - cs_{i,j-1}^n) \tag{3.11}$$

If the equations yield negative values for H_i^n, the latter is set to zero. The upper bound of H_i^n is determined by $(T_i - Cmax_i - G_i^{n-1})$, since exceeding this value would mean that a packet was initiated before the end of the last instance.

The bandwidth guardian performs two types of acceptance tests for each incoming header and data byte. The first one is performed when the *START* header of a new data packet arrives (3.12). This check is used to verify whether the incoming packet is received after the extended guard area (which is marked in red in Figure 3.11). The right term of the inequality (3.12) denotes the earliest time when a new task is able to start after the last instance.

$$s_{i,j}^n > f_{i,j}^n + G_i^{n-1} + H_i^n \tag{3.12}$$

Further acceptance tests are also performed upon reception of *RESUME* headers or normal data bytes. The check rationale, however, is different from the previous one. Here it is verified whether all bytes of the packet are received within the green and yellow areas in Figure 3.9.

$$t \leq s_{i,j}^n + (T_i - G_i^{n-1}) \tag{3.13}$$

If errors occur, either due to scheduling in previous nodes or corrupted data transmission, the system will be able to recover easily. In such a situation the values of H_i^n or $cs_{i,j}^n$ will only be updated when a correct header arrives. The consequence is a decrease in H_i^n, possibly down to zero. When H_i^n is zero, the packet can immediately be scheduled (recovering the system from the mentioned fault), because the time that has elapsed since the last correct task instance is more than or equal to T_i.

A new absolute deadline for each new task instance forwarded by a node n is calculated and sent to the scheduler when a *START* header is received. If a *RESUME* header is received, the absolute deadline will be updated and passed to the scheduler, which in turn will replace it internally. The absolute deadline is calculated by means of the following equation:

$$d_{i,j}^n = cs_{i,j}^n + D_i^n + H_i^n \tag{3.14}$$

Summarizing the admission control method, we can state that actions will be taken whenever headers or data bytes arrive at a node. When data bytes of a given task are received, it will be checked whether they arrive within a defined time interval after the corresponding *START* header. It is also assured that the maximum number of received data bytes will not exceed the packet size. On the other hand, when headers arrive at a receiver, the admission control block of the respective port will execute the following five steps:

1. Check if the header is correct, with a valid ID and correct CRC,
2. execute an on-line acceptance check,
3. if the acceptance check passes, update the H_i^n variable, otherwise abort reception,
4. calculate, if the packet is forwarded, the absolute deadline of the task and submit it to the scheduler.

The complete admission control flow is detailed in Figure 3.12.

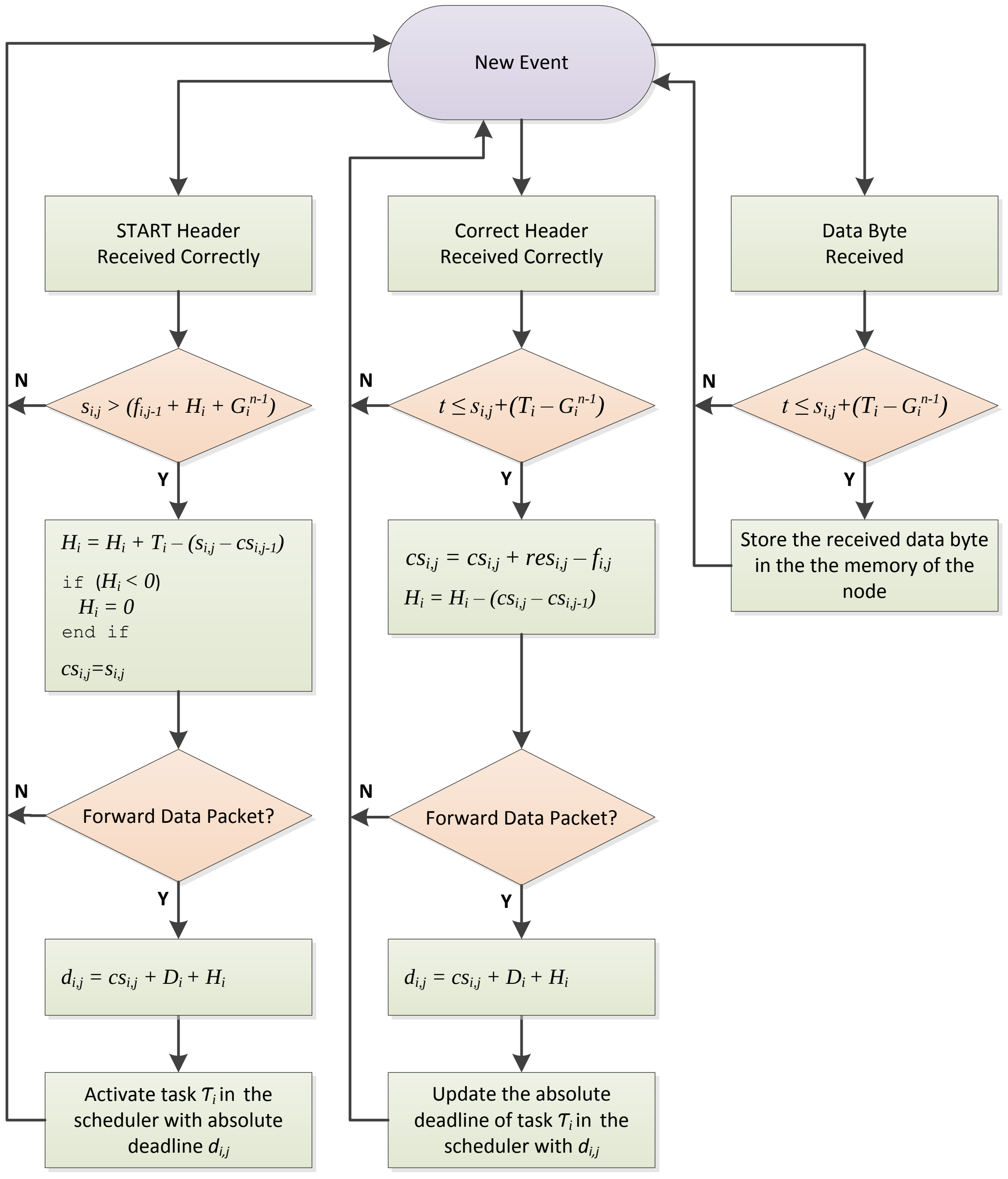

New Event
START Header Received Correctly
Correct Header Received Correctly
Data Byte Received
N
$s_{i,j} > (f_{i,j-1} + H_i + G_i^{n-1})$
N
$t \leq s_{i,j}+(T_i - G_i^{n-1})$
N
$t \leq s_{i,j}+(T_i - G_i^{n-1})$
Y
Y
Y
$H_i = H_i + T_i - (s_{i,j} - cs_{i,j-1})$
if ($H_i < 0$)
$H_i = 0$
end if
$cs_{i,j}=s_{i,j}$
$cs_{i,j} = cs_{i,j} + res_{i,j} - f_{i,j}$
$H_i = H_i - (cs_{i,j} - cs_{i,j-1})$
Store the received data byte in the the memory of the node
N
Forward Data Packet?
N
Forward Data Packet?
Y
Y
$d_{i,j} = cs_{i,j} + D_i + H_i$
$d_{i,j} = cs_{i,j} + D_i + H_i$
Activate task T_i in the scheduler with absolute deadline $d_{i,j}$
Update the absolute deadline of task T_i in the scheduler with $d_{i,j}$

Figure 3.12: Admission control flow

In the TrailCable protocol the admission control method described in this section is entirely executed in hardware for performance reasons. It is worth mentioning that the approach presented requires relatively few hardware resources since all calculations are performed either by simple additions or subtractions and no multiplication or division is used. Chapter 7 presents measurements that prove the functionality of the admission control method in a real hardware implementation of the TrailCable protocol.

3.7 Clock Synchronization

Clock synchronization plays an important role in real-time distributed systems. The TrailCable protocol supports this service by generating globally synchronized time interrupts to the host microcontrollers. Implementation of the clock synchronization algorithm is based on timestamps appended to the end of normal frames. These timestamps represent the sum of the accumulated line and routing delays along a path since the previous global time tick in the origin node. Each node on a channel path reads the timestamp (ts) of the previous node and adds the local routing delay (t_{route}) and the propagation time of the next link (t_{prop}). For better synchronization results, at run-time the TrailCable protocol measures the propagation time of the communication links to increase the precision of timestamps; this can easily be performed by means of a timed "pinging" procedure.

One of the most widely known clock synchronization protocols for point-to-point architectures is the IEEE 1588 [42], but it is based on a master-slave approach that has some drawbacks as far as safety is concerned. Therefore a distributed algorithm is used with TrailCable. In order to perform clock synchronization, a set of nodes must be initially defined. To reach a Byzantine agreement a minimum of $3 \times f + 1$ nodes are required to tolerate f faults, but this condition can be relaxed for less critical applications without fault tolerance, where the synchronization can be achieved with only two nodes. The algorithm used by TrailCable is based on the algorithm by Lundelius and Lynch [63], which is fully distributed. In each communication round, the nodes participating in the synchronization initially sort the time differences between the local clock and the clock of the remaining nodes (retrieved via the timestamps). Then, the f highest and f lowest values are eliminated and an average calculation is executed using the remaining values. The result is a correction factor that is used to increase or decrease the local communication period so that it converges to the global time. The TrailCable protocol employs the median value as the averaging function since it yields good results and requires a simple hardware implementation.

When compared to the clock synchronization procedures in time-triggered protocols such as TTP/C and FlexRay, TrailCable requires a higher packet overhead to explicitly send timestamps via the communication network. This happens because in the former protocols a broadcast bus is used, thus allowing all nodes to determine the exact moment a packet was sent by others. On the other hand the TrailCable approach has also some advantages. First of all, a set of nodes which is not synchronized or is still establishing the initial synchronization is able to transmit all packets to the intended destination, which is not possible with a TDMA access over a broadcast bus. Additionally, another interesting feature the TrailCable protocol supports is the definition of multiple clock synchronization domains, as exemplified in Figure 3.13. As can be seen in the example, clock regions can

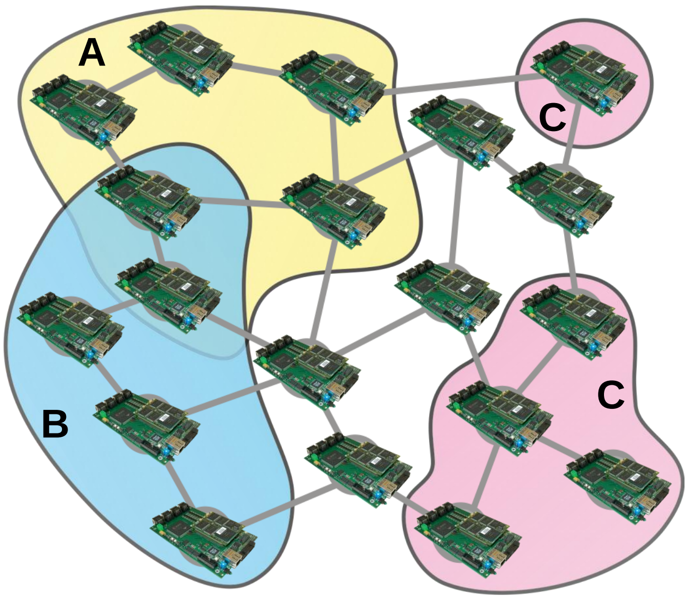


Figure 3.13: Multiple clock domains

be disjointed and even overlapping. This is possible because of the dynamic scheduling of packets using the EDF algorithm, which does not require a synchronized operation.

When it comes to scheduling feasibility, the effect of clock synchronization must be taken into account. Eventually, the period of a certain communication task will have to be shortened to achieve synchronization. If this is ignored for schedulability analysis, there is the possibility that deadlines will be missed in heavy traffic conditions. In order to avoid this, the amount of time a period can be shortened by is limited and known in advance. Additionally, this effect can be dealt with transparently by means of a further decrease in the value of the Δ factor (cf. Section 3.5) that is used for the feasibility tests. Nevertheless, since clock drift is generally very small over time, the utilization factor of the data links is not expected to be perceptibly impaired due to the clock synchronization service.

3.8 Non-Real-Time Communication

The TrailCable protocol is able to share bandwidth between hard-real-time and non-real-time communication. The former has always priority, but as soon as the ready queue of an EDF scheduler becomes idle, pending non-real-time packets can utilize the transmission medium. If a non-real-time packet is being transmitted and a new hard-real-time task arrives, then the ongoing transmission can be simply interrupted by the preemption mechanism to serve the higher-priority demand. As compared to TDMAs approaches where a time window is reserved for non-real-time traffic, such as in FlexRay and PROFINET, the TrailCable technique has the advantage that non-real-time traffic can fully utilize the available capacity when hard-real-time communication is idle.

The mechanism described is known as “background scheduling” [21] and, though simple, allows good separation of hard-real-time and non-real-time messages. On the other hand, dynamic servers [82] are not only able to guarantee that deadlines of real-time packets be met, but also improve response times of aperiodic, non-real-time traffic. However, the drawback of dynamic servers as compared to background scheduling is a higher complexity of the scheduling algorithms. In order to keep the hardware circuits of the packet schedulers simple, the initial TrailCable implementation employs the background technique. Indeed, in many applications the gains in performance of dynamic servers as compared to background scheduling present no considerable advantage. It is worth mentioning that although response times of non-real-time packets can be improved by the use of dynamic servers, average data throughput remains the same as with the background scheduling.

In order to differentiate real-time from non-real-time packets, the latter are assigned a special ID. Once the input ports identify such packets, these are passed to a non-real-time switch that routes the packet to the appropriate host or communication ports. Routing can be simplified by using techniques such as path and logical addressing employed with the SpaceWire protocol. This thesis will focus on hard-real-time communication, but given the loose coupling between time-critical and best-effort traffic provided by the TrailCable architecture, non-real-time functionalities can be added on without impairing the timeliness of the protocol.

Chapter 4

The TrailCable Verifier Tool

In order to facilitate the deployment of the TrailCable protocol in practical applications, a tool was developed to perform an automated schedulability analysis for a given network configuration. The tool, called TrailCable Verifier, reads a TrailCable network formal model, checks the feasibility of this model, and finishes on either of the two following outcomes: if the network configuration is feasible, C-code containing the configuration of each node in the network will automatically be generated. Otherwise, an error message will indicate where the network configuration feasibility check has failed. Although there are different tools for the analysis of real-time systems such as Symtas [39], the development of the TrailCable Verifier is justified by the fact that it performs a variety of tasks that are specific to the TrailCable protocol.

4.1 Communication System Modeling

A TrailCable network is modeled by means of a set of four XML files, each describing a different aspect of a given configuration: hardware properties, network topology, real-time tasks, and routing. Partitioning the network model into different files makes it modular and any changes in the configuration can be limited to the affected file only. Each of these four files has a corresponding Document Type Definition (DTD) that contains the grammar rules used to parse the XML description. The DTD files are listed in Appendix A.

To demonstrate the way a TrailCable network can be modeled, a simplified example of a brake-by-wire system is used. In such systems there are no mechanical links between the brake pedal and the brake actuators located in the wheels of the cars: control information is exchanged between different modules via a data bus that must meet stringent safety requirements. Figure 4.1 shows an example of a TrailCable network for an automotive brake-by-wire system. The central module is responsible for sensing the brake pedal

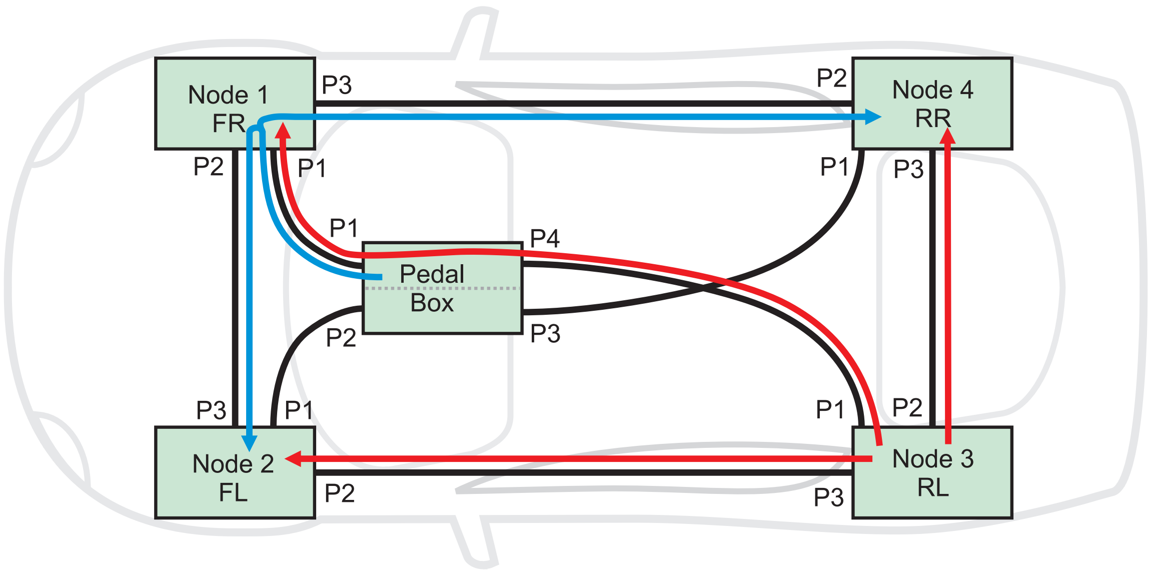


Figure 4.1: A brake-by-wire network

excursion and the hand-brake status and sends the information to the wheel nodes. This central node is two-fold redundant: it has two channels for reading the pedal excursion, with each channel connected to two opposite brake units. Each brake unit is made up of an ECU, connected to the network, that controls an electromechanical brake actuator (EMB). Additional external sensor data such as throttle, steering, and acceleration, that may be required in a brake-by-wire system, are also transmitted via the network but are omitted in this example for the sake of simplicity.

The design rationale for the proposed brake-by-wire system assumes the braking decision to be distributed: all four wheel nodes must reach an agreement (based on the data of a variety of sensors) on which braking forces will be applied to the wheels. Moreover, it is assumed that a wheel node will not take a different braking action than that agreed on by the other members. To cope with the potential safety risks that can arise due to malfunctioning nodes, a Byzantine agreement [59] can be employed in this brake-by-wire example. The Byzantine agreement is characterized by the fact that a small number of faulty nodes cannot impair the correct operation of others. In order to reach a Byzantine agreement, $3m + 1$ nodes are required to tolerate m faults. Therefore, if one wheel node fails in the given example, although the mechanical braking capacity of the system is compromised, the remaining three wheel nodes will be able to detect the fault and compensate for the braking forces to bring the car to a safe stop.

In the brake-by-wire system presented not only node, but also communication faults must be considered. In order to reach a Byzantine agreement in a network, there must be $2m+1$ disjoint paths between the sender and the receivers [30]. Therefore the proposed brake-by-wire architecture requires each node to broadcast its information to the others via three disjoint routes (the way it is achieved is described in the following sections). It is worth

```
< Implementation

  timeResolution              = ".025"
  defaultForwardingDelay      = "1.250"
  defaultLinkPropagationDelay = "2.000"
  maximumTasks                = "64"
  deviation                   = "0.99"
  maxPeriodXdeadlineRatio     = "2"
  maximumPacketSize           = "256"
  maximumPayloadSize          = "249"
  maximumPeriod               = "25000000"
  preemptionHeaderSize        = "3"
  transmissionRate            = "32000000"
  maxHardwareMemory           = "4096"
  maxHostAccessedMemory       = "1023"
>
```

Figure 4.2: Example of a communication engine properties file

mentioning that the automotive network shown in Figure 4.1 could be further extended to support a variety of additional systems such as steering, power-train, etc., sharing the same communication links, thus reducing cabling efforts.

4.1.1 Communication Engine Properties

The communication engine properties file (Figure 4.2) contains different parameters that depend on a given hardware implementation of the TrailCable protocol. This file describes, among others characteristics, node and link performance, memory capacity, and maximum packet size.

For the brake-by-wire example, the parameters are set according to the communication engine hardware presented in Chapter 6. It has a time resolution of 25 ns, is able to handle 64 simultaneous real-time tasks per communication, has 4 kB internal memory and a data throughput of 32 Mbps. Moreover, the implementation accepts real-time tasks with a period of up to 25 seconds. Some of the parameters described in this file were already introduced in Chapter 3, such as forwarding delay, link propagation delay, and deviation.

4.1.2 Network Topology

The network topology file (Figure 4.3) describes the way nodes are physically interconnected to form the network. The number of nodes in the network and the maximum

```
<Graph numNodes="5" maxPorts="8" >

<Connection node1="0"  node2="1"  port1="1" port2="1" />
<Connection node1="0"  node2="2"  port1="2" port2="1" />
<Connection node1="0"  node2="3"  port1="3" port2="1" />
<Connection node1="0"  node2="4"  port1="4" port2="1"
                                  linkPropagationDelay="3.5" />

<Connection node1="1"  node2="2"  port1="2" port2="3" />
<Connection node1="2"  node2="3"  port1="2" port2="3" />
<Connection node1="3"  node2="4"  port1="2" port2="3" />
<Connection node1="4"  node2="1"  port1="2" port2="3" />

<NodeInformation node="0" forwardingDelay="1.5" />

<Host name="pedal_box" node="0"   port="0" />
<Host name="FR_wheel"  node="1"   port="0" />
<Host name="FL_wheel"  node="2"   port="0" />
<Host name="RL_wheel"  node="3"   port="0" />
<Host name="RR_wheel"  node="4"   port="0" />

</Graph>
```

Figure 4.3: Example of a network topology file

number of ports for each of them are the first information required for the network topology XML file, as they set the constraints for the creation of a given communication infrastructure.

Each communication link is instantiated in the network topology XML file by a tag called Connection. After the tag, the two nodes that are connected with their respective ports are indicated. The propagation delay of all links can be configured independently, should they be larger (to avoid impairing the channel deadline checks) or significantly smaller (to avoid a too pessimistic scenario) than the default value inserted in the hardware properties file. The propagation delay depends basically on cable lengths, therefore for a given application different communication links are likely to have different latencies.

In the same way that the propagation delay can be set independently for each link, the forwarding delay of all nodes can be defined. Nevertheless, if all nodes of a given application are implemented with the same TrailCable communication hardware engine, the forwarding delay holds for the whole network.

Each host is assigned an alias to make the model more intuitive and to facilitate the creation of real-time channels. The port representing the host interface is explicitly indicated as well.

```
<ChannelList graph="Graph.xml" StaticRoute="StaticRoute.xml"
 DynamicRoute="DynamicRoute.xml" communicationHWproperties="CommHWprop.xml">

  <!-- Pedal Box outgoing routes -->
  <Channel id="04" sourceHost="pedal_box" period="1000" payloadSize="64" >
    <TargetHost host="FR_wheel" deadline="500" />
    <TargetHost host="FL_wheel" deadline="500" />
    <TargetHost host="RR_wheel" deadline="500" />
  </Channel>
  (...)

  <!-- FR Wheel outgoing routes -->
  (...)

  <!-- FR Wheel outgoing routes -->
  (...)

  <!-- RL Wheel outgoing routes -->
  <Channel id="30" sourceHost="RL_wheel" period="1000" payloadSize="64" >
    <TargetHost host="pedal_box" deadline="500" />
    <TargetHost host="FR_wheel" deadline="500" />
  </Channel>

  <Channel id="32" sourceHost="RL_wheel" period="1000" payloadSize="64" >
    <TargetHost host="FL_wheel" deadline="500" />
  </Channel>

  <Channel id="34" sourceHost="RL_wheel" period="1000" payloadSize="64" >
    <TargetHost host="RR_wheel" deadline="500" />
  </Channel>

  <!-- RR Wheel outgoing routes -->
  (...)

</ChannelList>
```

Figure 4.4: Example of a real-time tasks file

4.1.3 Real-Time Channels

The XML file that models the real-time channels is the main input for the TrailCable verifier tool. It includes references (Figure 4.4) to the remaining model files that contain the network topology, routing information, and communication engine properties.

Real-time channels are instantiated by entering their basic characteristics: source host, destination(s) host(s) with respective deadline(s), payload, and ID. A different deadline can be assigned to each destination host of a real-time communication task. The routing information for each real-time task is defined in a distinct XML file since there are potentially many different possible routes between the source node and the destination node(s) that even could be dynamically reconfigured.

In the brake-by-wire example all nodes send information to all others. The central pedal box implements such a logical broadcast employing four real-time channels (each targeting a set of three wheels) while the wheels accomplish the same task with three channels. As a result, a total of 16 real-time channels are instantiated in the presented system. All real-time channels were assigned a period of 1 ms, a channel deadline of 500 μs and a payload of 64 bytes. For clarity reasons, only one (shown in blue) of the four real-time channels of the pedal box and only the channels (shown in red) of one of the four wheels are represented in the file represented in Figure 4.4.

4.1.4 Routing

An explicit and consistent channel route must be assigned to each real-time channel in the network. Routes are created by a set of transmitter-receiver pairs that, when ordered, define a path between the source and all destination hosts. In special situations, when two nodes are interconnected via two or more links, it is also required to define which transmitter port is to be used.

A route is associated to the corresponding real-time channel by means of a common ID. An appropriate relative deadline can be assigned to each link segment in the route. Optionally, the ID of a real-time channel can also be altered in each hop of the network. The latter functionality is achieved by specifying a new destination ID for every route segment.

Figure 4.5 presents the routing definition XML file for the brake-by-wire example. Firstly, it shows the route of one of the four real-time channels that originate at the central pedal box. This route leads initially to the front-right wheel, where it bifurcates and reaches both the rear-right and front-left wheels. The remaining three routes from the pedal box (omitted in Figure 4.5) follow the same pattern: each of them exits the central node via a different communication link, reaches the first wheel node, is then split and ends in the two neighboring wheel nodes.

The routing XML file also defines the routes that initiate in the four wheel nodes. Figure 4.5 shows the three routes from the rear-left wheel node. The first one reaches the opposite wheel node via the pedal box node. The two other routes lead to the immediate wheel node neighbors. This is also the pattern employed for the three remaining wheel nodes.

```
<RouteList>

  <!-- Pedal Box outgoing routes -->
  <ChannelRoute channelID="04" defaultRelativeDeadline="200"
                               defaultDestinationTaskID="04" >
    <Path from="0" to="1" />
    <Path from="1" to="2" relativeDeadline="60" sourcePort="2"
                          destinationTaskID="04" />
    <Path from="1" to="4" relativeDeadline="60" sourcePort="3"
                          destinationTaskID="04" />
  </ChannelRoute>
  (...)

  <!-- FR Wheel outgoing routes -->
  (...)

  <!-- FL Wheel outgoing routes -->
  (...)

  <!-- RL Wheel outgoing routes -->
  <ChannelRoute channelID="30" defaultRelativeDeadline="200"
                               defaultDestinationTaskID="30" >
    <Path from="3" to="0" />
    <Path from="0" to="1" />
  </ChannelRoute>

  <ChannelRoute channelID="32" defaultRelativeDeadline="200"
                               defaultDestinationTaskID="32" >
    <Path from="3" to="2" />
  </ChannelRoute>

  <ChannelRoute channelID="34" defaultRelativeDeadline="200"
                               defaultDestinationTaskID="34" >
    <Path from="3" to="4" />
  </ChannelRoute>

  <!-- RR Wheel outgoing routes -->
  (...)

</RouteList>
```

Figure 4.5: Example of a routing file

4.2 Tool Flow

As soon as the network XML-based model is available, the TrailCable Verifier tool can be called on to run the schedulability analysis on the given configuration. The tool executes

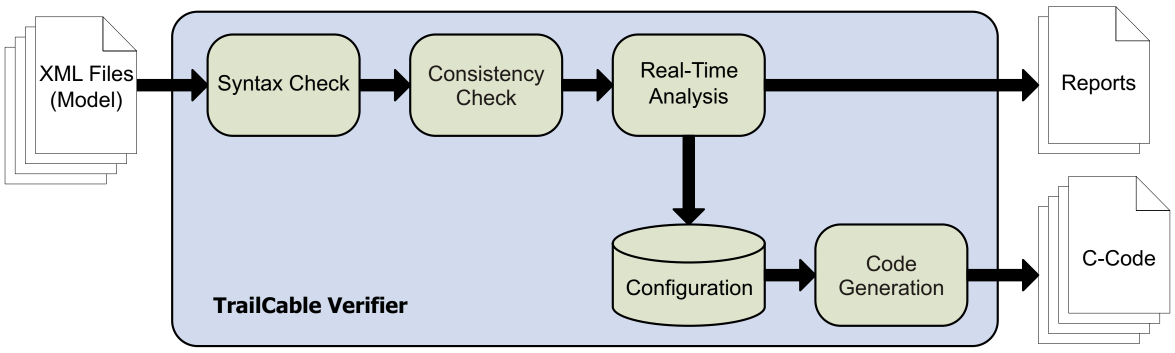


Figure 4.6: The TrailCable Verifier Flow

the following steps when started:

1. **XML syntax check** - reads the XML files and verifies whether the syntax is correct on the basis of the DTD definitions,

2. **Model consistency check** - in this step the tool checks whether the network topology complies with a set of rules, such as: only one connection is allowed for each communication port, the maximum number of nodes is respected, etc. It is also verified if there are valid routes for all real-time channels,

3. **Real-Time Analysis** - this component executes two tasks. Firstly, given the relative deadlines, propagation and forwarding delays of all possible links of a real-time channel, it is verified whether the channel deadline can be met by means of the procedure described in Section 3.3. If the test fails, the TrailCable Verifier application quits after showing the reason why a certain channel is not feasible. Afterwards, for each communication port in the network, the tool lists all of its real-time tasks with the respective deadlines and checks whether the set is schedulable, using the demand criterion [60], as described in Section 3.5. Should the test fail, the TrailCable Verifier application will quit after indicating the affected communication port and the fault reason,

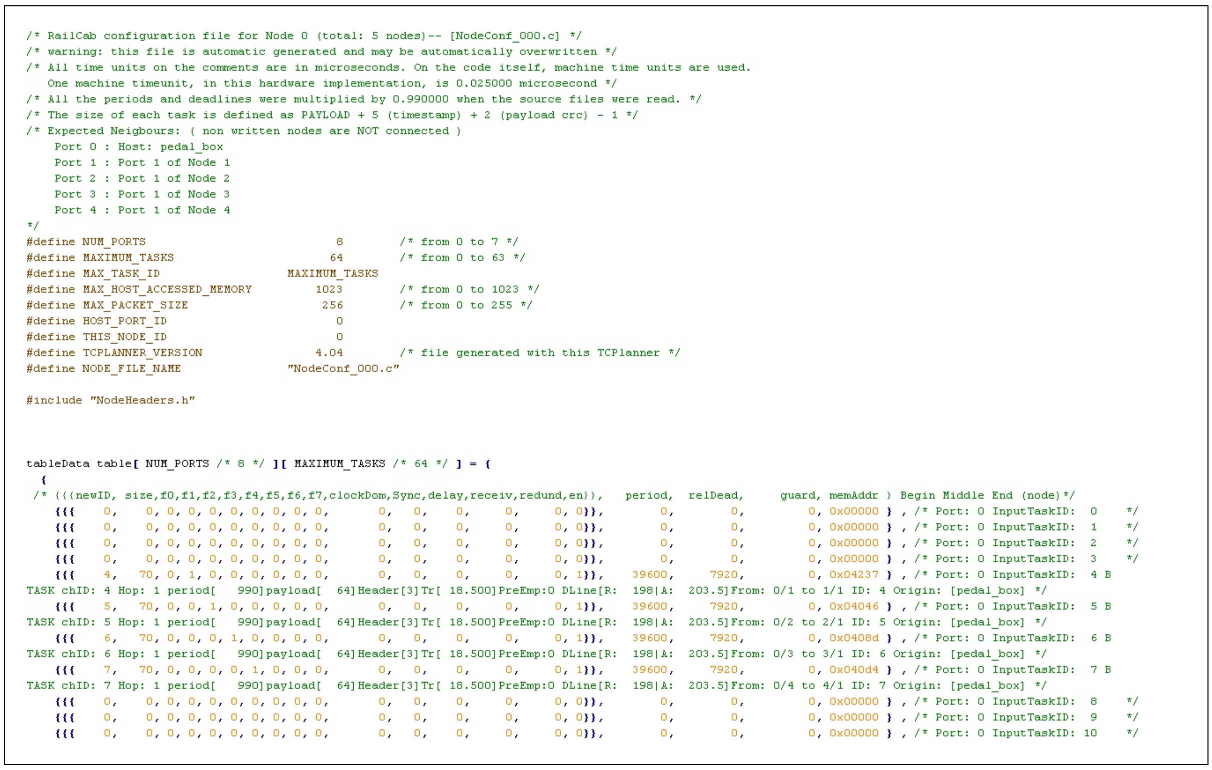
4. **Network Configuration** - if all tests are passed successfully, the application generates configuration data for each node in the network,

5. **Code generation** - using the network configuration data the application generates C-code for all nodes in the network.

In addition to C-code, the TrailCable Verifier tool creates two basic reports. The first is a list of all real-time channels, including their routes and worst-case channel latencies. The second is a list of all communication ports and the real-time channels that pass through them.

4.3 Configuration Data and Code Generation

In order to make sure that the network nodes are, at run-time, consistent with the model used for the schedulability analysis, a file containing the protocol configuration is automatically generated by the TrailCable Verifier tool. Other advantages of working with automatic code generation from a higher-level model include the seamless transition to the target hardware and the possibility to cope with complex networks.



```
/* RailCab configuration file for Node 0 (total: 5 nodes)-- [NodeConf_000.c] */
/* warning: this file is automatic generated and may be automatically overwritten */
/* All time units on the comments are in microseconds. On the code itself, machine time units are used.
   One machine timeunit, in this hardware implementation, is 0.025000 microsecond */
/* All the periods and deadlines were multiplied by 0.990000 when the source files were read. */
/* The size of each task is defined as PAYLOAD + 5 (timestamp) + 2 (payload crc) - 1 */
/* Expected Neigbours: ( non written nodes are NOT connected )
    Port 0 : Host: pedal_box
    Port 1 : Port 1 of Node 1
    Port 2 : Port 1 of Node 2
    Port 3 : Port 1 of Node 3
    Port 4 : Port 1 of Node 4
*/
#define NUM_PORTS                        8        /* from 0 to 7 */
#define MAXIMUM_TASKS                   64        /* from 0 to 63 */
#define MAX_TASK_ID                MAXIMUM_TASKS
#define MAX_HOST_ACCESSED_MEMORY      1023        /* from 0 to 1023 */
#define MAX_PACKET_SIZE                256        /* from 0 to 255 */
#define HOST_PORT_ID                     0
#define THIS_NODE_ID                     0
#define TCPLANNER_VERSION             4.04        /* file generated with this TCPlanner */
#define NODE_FILE_NAME             "NodeConf_000.c"

#include "NodeHeaders.h"



tableData table[ NUM_PORTS /* 8 */ ][ MAXIMUM_TASKS /* 64 */ ] = {
  {
 /* {{{newID, size,f0,f1,f2,f3,f4,f5,f6,f7,clockDom,Sync,delay,receiv,redund,en}},   period,  relDead,     guard, memAddr } Begin Middle End (node)*/
   {{{    0,    0, 0, 0, 0, 0, 0, 0, 0, 0,       0,    0,     0,      0,     0, 0}},       0,        0,         0, 0x00000 } , /* Port: 0 InputTaskID:  0    */
   {{{    0,    0, 0, 0, 0, 0, 0, 0, 0, 0,       0,    0,     0,      0,     0, 0}},       0,        0,         0, 0x00000 } , /* Port: 0 InputTaskID:  1    */
   {{{    0,    0, 0, 0, 0, 0, 0, 0, 0, 0,       0,    0,     0,      0,     0, 0}},       0,        0,         0, 0x00000 } , /* Port: 0 InputTaskID:  2    */
   {{{    0,    0, 0, 0, 0, 0, 0, 0, 0, 0,       0,    0,     0,      0,     0, 0}},       0,        0,         0, 0x00000 } , /* Port: 0 InputTaskID:  3    */
   {{{    4,   70, 0, 1, 0, 0, 0, 0, 0, 0,       0,    0,     0,      0,     0, 1}},   39600,     7920,         0, 0x04237 } , /* Port: 0 InputTaskID:  4 B
TASK chID: 4 Hop: 1 period[   990]payload[  64]Header[3]Tr[ 18.500]PreEmp:0 DLine[R:  198|A:  203.5]From: 0/1 to 1/1 ID: 4 Origin: [pedal_box] */
   {{{    5,   70, 0, 0, 1, 0, 0, 0, 0, 0,       0,    0,     0,      0,     0, 1}},   39600,     7920,         0, 0x04046 } , /* Port: 0 InputTaskID:  5 B
TASK chID: 5 Hop: 1 period[   990]payload[  64]Header[3]Tr[ 18.500]PreEmp:0 DLine[R:  198|A:  203.5]From: 0/2 to 2/1 ID: 5 Origin: [pedal_box] */
   {{{    6,   70, 0, 0, 0, 1, 0, 0, 0, 0,       0,    0,     0,      0,     0, 1}},   39600,     7920,         0, 0x0408d } , /* Port: 0 InputTaskID:  6 B
TASK chID: 6 Hop: 1 period[   990]payload[  64]Header[3]Tr[ 18.500]PreEmp:0 DLine[R:  198|A:  203.5]From: 0/3 to 3/1 ID: 6 Origin: [pedal_box] */
   {{{    7,   70, 0, 0, 0, 0, 1, 0, 0, 0,       0,    0,     0,      0,     0, 1}},   39600,     7920,         0, 0x040d4 } , /* Port: 0 InputTaskID:  7 B
TASK chID: 7 Hop: 1 period[   990]payload[  64]Header[3]Tr[ 18.500]PreEmp:0 DLine[R:  198|A:  203.5]From: 0/4 to 4/1 ID: 7 Origin: [pedal_box] */
   {{{    0,    0, 0, 0, 0, 0, 0, 0, 0, 0,       0,    0,     0,      0,     0, 0}},       0,        0,         0, 0x00000 } , /* Port: 0 InputTaskID:  8    */
   {{{    0,    0, 0, 0, 0, 0, 0, 0, 0, 0,       0,    0,     0,      0,     0, 0}},       0,        0,         0, 0x00000 } , /* Port: 0 InputTaskID:  9    */
   {{{    0,    0, 0, 0, 0, 0, 0, 0, 0, 0,       0,    0,     0,      0,     0, 0}},       0,        0,         0, 0x00000 } , /* Port: 0 InputTaskID: 10    */
```

Figure 4.7: Section of automatically generated C-code

For each node in the network a different C-code file is generated by the tool. This file includes data tables with all variables required to set up the TrailCable communication hardware. This C-code file is then compiled together with TrailCable drivers and application code in order to create the executable binary file for the network node.

The generated C-code file (Figure 4.7) instantiates a configuration table for each communication port in a network node. These tables contain entries that specify the routing, scheduling, and addressing parameters for all communication tasks. The fields of such entries are: ID, size, packet forwarding information, period, relative deadline, guard time, memory address, and configuration flags. The format of these entries is compatible with the hardware implementation of the TrailCable protocol (Section 6.3.3), so that a simple memory copy operation of the data tables is sufficient to program the network node.

Chapter 5

Dynamic Reconfiguration

The majority of distributed embedded systems, especially in the domain of mechatronics, are characterized by the fact that the required real-time data communication layer is designed and configured during the design phase. Hardware and software in such systems are normally implemented for a pre-specified function that cannot be altered during run-time. For this reason, the development of most wired real-time communication protocols currently available was not targeted at supporting dynamic reconfiguration.

With the advent of new research activities in emerging fields such as self-optimizing and self-organizing systems [9], a higher flexibility of the real-time data communication infrastructures is desired. Reconfiguring both the application functions and the underlying data communication layer yields brand new possibilities to continually adapt a system and make it better suited to perform the necessary functions at a given time.

Due to the EDF-scheduling mechanism of the TrailCable protocol that dispenses with a global clock synchronization, reconfiguration of portions of a running network is facilitated since all necessary changes can be limited to the affected components only. Moreover, reconfiguring one communication channel in a given node does not require any other concurrent channels to be altered in the process.

Since the focus of this work is on the hard-real-time behavior of the communication system, a feasibility analysis must be carried out prior to any on-line reconfiguration to guarantee predictability. Therefore, the fact that the communication system can be dynamically reconfigured does not imply that any chosen configuration is necessarily valid. Buttazzo [21] presented the basic scheme of the guarantee mechanism used in generic dynamic hard-real-time systems. This basic scheme can be adapted to fit the TrailCable communication characteristics, as shown in Figure 5.1.

This chapter is organized as follows: we initially present a basic scheme for classifying different types of reconfiguration processes. Then, the framework that permits the recon-

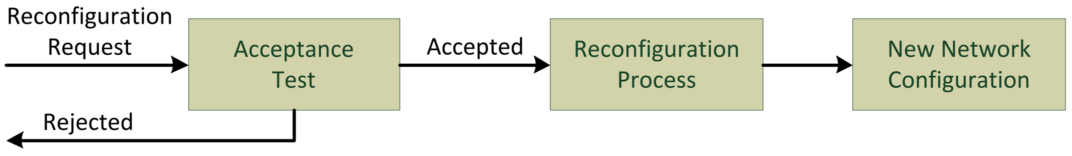


Figure 5.1: Dynamic reconfiguration scheme

figuration in the TrailCable network will be introduced. Finally, we deal with methods to allow the creation of new configurations at run-time.

5.1 Classification

In order to distinguish different types of reconfiguration processes, it is useful to classify them with respect to their duration and data integrity. Figure 5.2 shows the classification scheme employed.

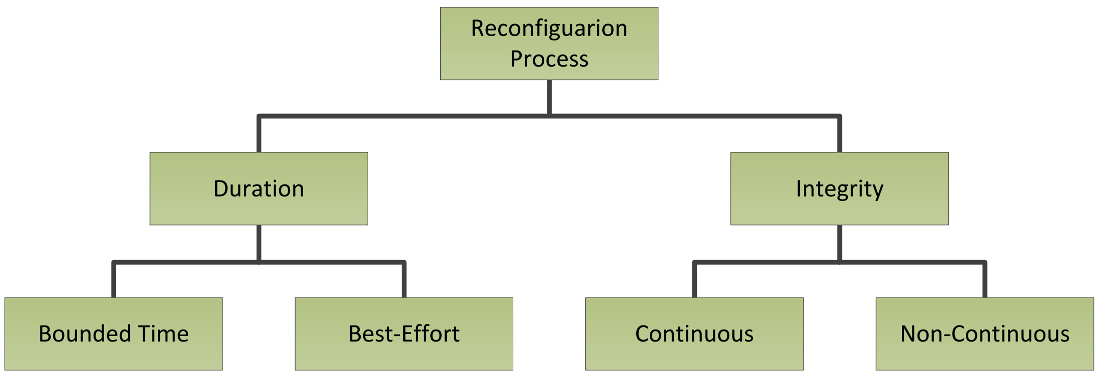


Figure 5.2: Classification of reconfiguration processes

Bounded time reconfiguration The reconfiguration process must be completed within a bounded time interval. In order to accomplish this, two requirements must be met. Firstly, it must be assured that the new network configuration is feasible, i.e., allocating all required channels and meeting their deadlines is possible. This first requirement is met if a schedulability analysis with a positive result is performed prior to configuring the network. The second requirement is that enough communication bandwidth be available for exchanging data among the participating nodes during the reconfiguration phase. Real-time communication channels can be reserved in order to meet this requirement.

Best-effort reconfiguration No guarantee is given that a new reconfiguration request will be accepted or that it can be executed within a specified time window. The advantage, however, is that this approach is more flexible, allowing, for example, the network to continually search for an optimal configuration during run-time to better suit application needs. Since there is no hard deadline for the process, non-real-time channels can be used for coordinating the reconfiguration.

Continuous reconfiguration No data packets are lost during the reconfiguration process. Such a reconfiguration can be guaranteed, for example, if the requirements of a given real-time channel are to be alleviated, that is, the transmission period or the deadlines are increased or the payload of the data packets is reduced.

Non-continuous reconfiguration The reconfiguration process leads to a transient and short interruption of the data packet transmissions in one or more channels. This happens, for example, when an existing channel must be temporarily excluded for the rearrangement of its communication path.

5.2 Reconfiguration Framework

Valid TrailCable network configurations can be generated either off-line by means of the TrailCable Verifier tool presented in Chapter 4 or at run-time. The framework described in this section allows interchanging different configurations at run-time and provides an abstraction layer that makes the reconfiguration process transparent to the requesting host.

The role of this framework is to provide a mechanism, but not a reconfiguration policy. By mechanism it is meant the technical realization that makes it possible to accomplish a certain task (the network reconfiguration, in this case). On the other hand, a policy specifies which entity has the rights to perform reconfiguration. Such a distinction facilitates the development of reconfigurable systems since both aspects can be regarded independently. A similar approach is used, for example, in the Linux operating system [27].

While the reconfiguration mechanism clearly belongs to the TrailCable protocol, reconfiguration policies may differ from one application to another and should be defined on a case-to-case basis. Therefore, when building up applications with dynamic reconfiguration capabilities, the system designer should stipulate a set of rules in order to define a policy. A policy must answer questions like these: Which node(s) may start a reconfiguration? Is there an arbitration to avoid conflicting reconfiguration requests? Are there any restrictions to issuing reconfiguration requests? These questions are neither exhaustive nor necessary but illustrate a guideline example to build a policy for a given application.

In this work we concentrate on the reconfiguration mechanism, which is implemented by means of the framework presented in Figure 5.3. It consists of the following com-

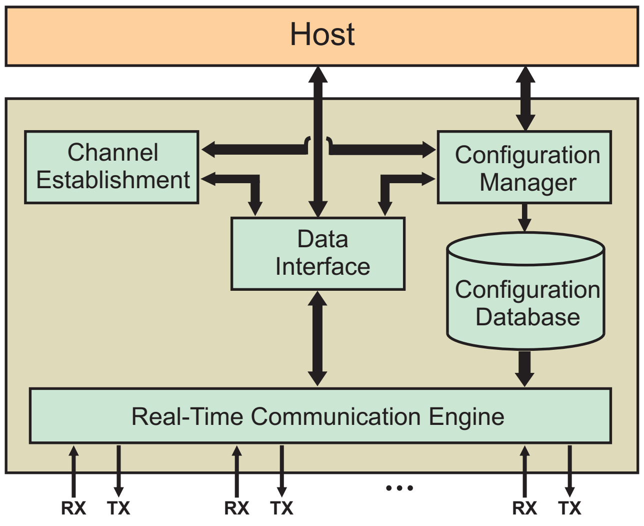


Figure 5.3: The reconfiguration framework

ponents: Data Interface, Configuration Database, Configuration Manager, and Channel Establishment. They will be detailed below.

5.2.1 Data Interface

The Data Interface component is a device driver for sending and receiving data packets. It abstracts the communication hardware so that other software modules can access the physical communication layer by means of simple system calls.

When the TrailCable protocol is used without its reconfiguration capabilities, the Data Interface component is used exclusively to exchange application data among different computing nodes. On the contrary, nodes are required to share information regarding the reconfiguration process. For this reason both the Channel Establishment and the Configuration Manager blocks are internally connected to the Data Interface component.

The decision whether real-time or non-real-time communication channels are used for reconfiguration depends on application demands. The trade-off here is that reserving real-time channels for this purpose allows the reconfiguration to be performed in a bounded time. On the other hand, the use of non-real-time channels dispenses with the necessity of allocating potentially under-utilized communication bandwidth (that could be otherwise exploited by running applications), but reconfiguration may take longer to complete.

5.2.2 Configuration Database

The role of the Configuration Database is to store multiple *channel* and *port configurations* that can be activated at run-time.

The *channel configuration* stores relevant information about a real-time channel, including the respective source node, destination node(s), minimum period, packet size, and channel deadline. Moreover, the channel configuration includes one or more mapping options that define the employed route and port configurations for all nodes of such channel. A global key is used to identify each channel configuration unequivocally.

The *port configuration* is a schedulable set of real-time tasks that use the same communication port for data transmission. Each real-time task in a given configuration is represented by its ID, a period (T_i), a relative deadline (D_i), a guard time (G_i), and the packet size (used to calculate the transmission time C_i).

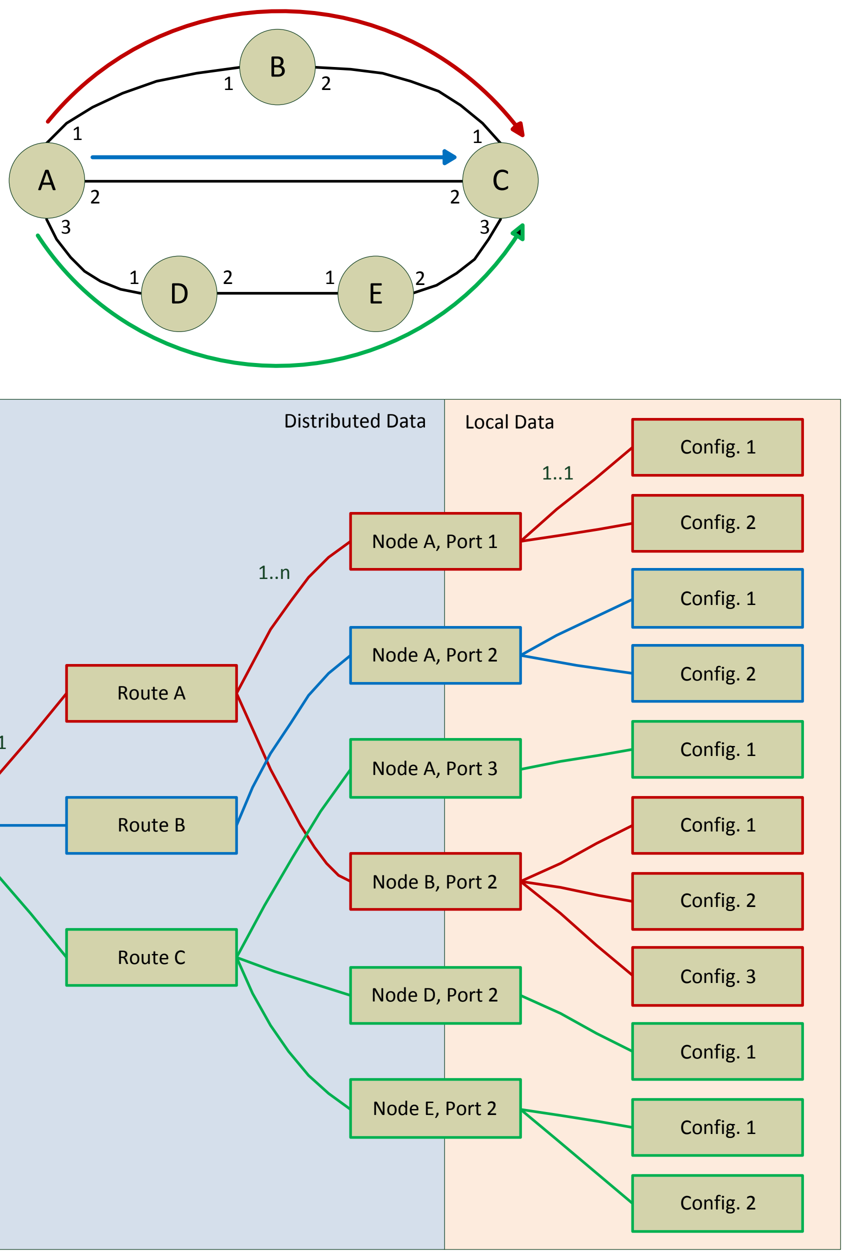


Figure 5.4: Configuration database example

Figure 5.4 illustrates an example of a data structure that stores the configuration of one real-time channel. The channel has three different and mutually exclusive mapping routes, indicated by the colors red, blue and green. Each of these routes is characterized by the set of communication ports involved in the transmission of the packets.

5.2.3 Configuration Manager

Reconfiguration requests are issued by the host to the Configuration Manager, which is then responsible for activating the new configuration. The activation is a distributed process where all nodes affected by the reconfiguration must participate. In this process there are one proactive node and one or more reactive ones. The proactive node receives the reconfiguration request from the host and is in charge of informing the reactive node(s) which configuration will be activated. This is done by sending the respective key. Then, all nodes will program the communication engine hardware.

If a new task is added, the programming process is straightforward: it is sufficient to write all parameters of that task to the communication engine and then enable it. Reprogramming an existing task can be done either in a continuous or a non-continuous manner. The latter alternative is simpler since the procedure does not need to be synchronized with the actual packet transmission. On the other hand, to guarantee a continuous reconfiguration it must be assured that the reprogramming a real-time communication task takes place exactly between two consecutive data packets. Therefore, while the former packet is routed and scheduled with the old configuration, the latter utilizes the new one. Reprogramming the communication engine exactly within two packets of a real-time task can be very challenging because the processor must be informed exactly when a packet transmission ends. Moreover, depending on the communication period, the time available for the reconfiguration can be very limited, demanding a high performance processor.

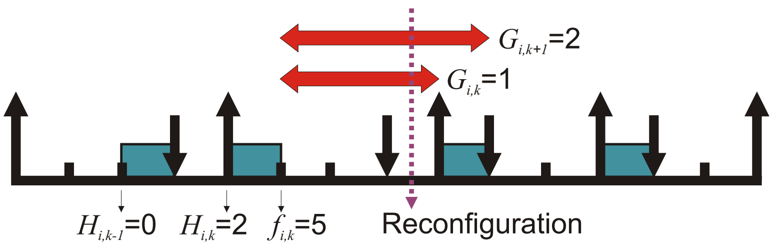


Figure 5.5: Effect of increasing the *guard* parameter

Another challenge to deal with during a continuous reconfiguration is an increase in the Guard parameter, G. Figure 5.5 shows such a reconfiguration scenario. In this example, the task parameters before reconfiguration are: $T = 4$, $C = 1$, $D = 3$, and $G = 1$. In the new configuration both the period and the relative deadline are decreased to $T = 3$ and $D = 1$. Therefore the new guard value is increased to $G = 2$. The communication time

C is kept constant. It can be seen that if the acceptance condition from Eq. (3.12) is evaluated with the new value of G, the first packet after reconfiguration will be discarded. A simple solution to this problem would be to calculate the right-hand term of Eq. (3.12) for each data byte received, so that after reconfiguration the acceptance condition can be checked correctly if the last calculated value is used. This solution, however, implies a higher degree of parallelization of the TrailCable hardware, since it must execute not only the mentioned calculation, but also the acceptance check of the received byte itself, which is given by Eq. (3.13).

In order to solve these two problems for continuous reconfiguration, namely reprogramming a task exactly between two packets and coping with an increase in G, one can add appropriate hardware support in the communication engine. The proposed hardware support is based on double-buffering the scheduling tables. By this, while one scheduling table holds the parameters for the active configuration of a certain task, new parameters can be written by the host into a second one. As soon as all parameters are written into the inactive table, their roles are automatically switched at the right moment by the communication engine, which happens just after the acceptance condition of Eq. (3.12). In the proposed mechanism, it must also be guaranteed that the buffer holding the parameters of a particular real-time task can be switched independently of all other tasks.

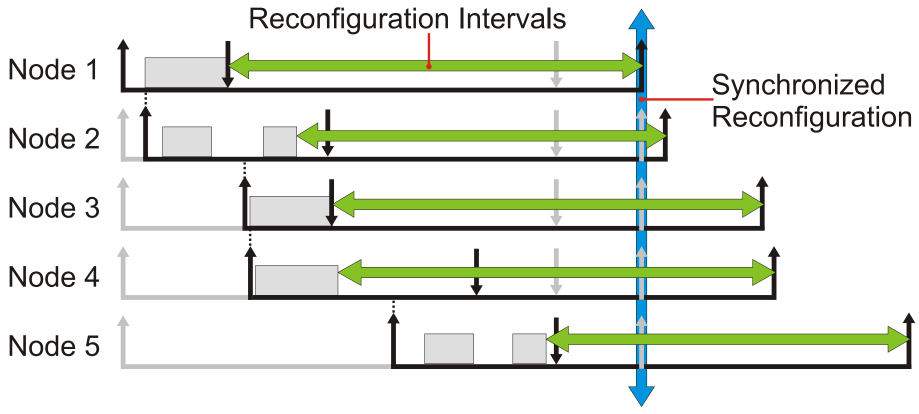


Figure 5.6: Continuous reconfiguration intervals

In a continuous reconfiguration all nodes involved in the reconfiguration of a given real-time channel must perform the reprogramming procedure between the same two consecutive task instances (Figure 5.6). One possibility of informing the nodes about the next reconfiguration interval is to use the clock synchronization mechanism of the TrailCable protocol. In this approach a future time instant for triggering the reconfiguration is selected by the proactive node and broadcast to the reactive ones. Another simple alternative is to set the source of a real-time channel as the proactive node and use a specific field in the data packet contents to inform the intermediate nodes of the moment the reconfiguration in that channel must be carried out.

5.2.4 Channel Establishment

For some applications it can be rather simple to create all the required configurations at the design phase and simply activate them at run-time. However, such an approach is not always practical. One of the reasons for this is that a TrailCable network can be used to interconnect nodes running applications that are completely unrelated, independent and even possibly developed by distinct groups.

To overcome such limitation, the TrailCable protocol allows the creation of new configurations at run-time, which is supported by the Channel Establishment component. In order to create a new configuration, the first step is to gather information about the network, such as the current topology (since modifications can occur due to failures or new cabling) and the list of the real time channels with the correspondent routes and relative deadlines. The process of gathering information from other nodes is realized via the Data Interface component.

Once all the required network information is available, the actual creation of a new configuration can take place. Due to the complexity of this task, the process description is presented in Section 5.3. This process is normally executed in a best-effort manner since it is not possible to guarantee that an appropriate configuration will be found nor how long the process will take to complete. When a valid configuration is generated, it must be distributed to all participating nodes to permit activation.

In analogy to the Configuration Manager component, it is possible to classify the nodes participating in the channel establishment process as either proactive or reactive. The method described so far is conducted by a proactive node, as it is responsible for executing all three steps: information gathering, configuration generation, and distribution. The remaining nodes are reactive: upon arrival of requests from the proactive node they either send the local information or store new configurations in the local database. In order to guarantee hard-real-time behavior even in the case of faults, the configuration received by the reactive nodes may be submitted to a schedulability analysis before being accepted.

5.3 Network Configurations

The TrailCable protocol allows a very efficient utilization of the available communication bandwidth thanks to the EDF scheduling strategy. However, in order to exploit this potential it is necessary to map all real-time channels to a network accordingly. By choosing the best routes and the best relative deadlines for scheduling the data packets, the number of real-time channels that can be mapped to the network can be maximized. In this section we will deal with this problem and present methods to find mappings for real-time channels that are both necessary and sufficient. Configurations generated off-line can benefit from more efficient mappings at the price of higher processing performance, whereas configurations generated on-line usually require simpler algorithms but with an adverse effect on optimality.

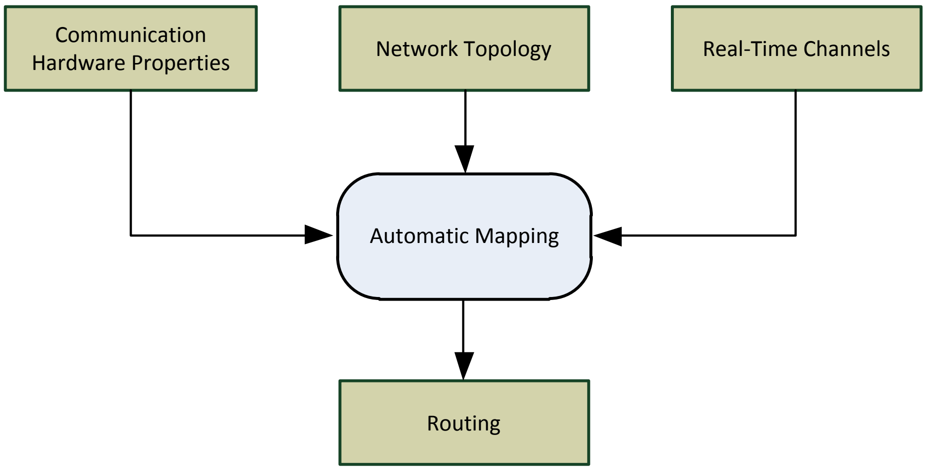


Figure 5.7: Generation of network mappings

The flow employed in the automatic mapping of real-time channels is depicted in Figure 5.7. The three necessary inputs correspond to the information contained in the XML files described in Section 4.1: *Communication Hardware Properties*, *Network Topology*, and *Real-Time Channels*. The mapping process is then responsible for automatically generating the information related to the *Routing* XML File. At run-time, however, the use of XML files is less practical and they can be substituted by equivalent data structures that are stored in the memories of the nodes.

Before showing how the mapping process works, we will introduce a method that models all possible combinations of relative deadlines that maintain a task set feasible. Then, we extend this approach to support end-to-end real-time guarantees of all channels in a network.

5.3.1 EDF Deadline Assignment

Different authors have presented methods for the assignment of deadlines with EDF scheduling. For example, in [19] both exact and approximate techniques were presented that allow finding feasible regions of deadlines. In [54], the assignment is done by means of linear programming with the objective of reducing jitter.

Other than reducing jitter, the deadline assignment in the current work has for main objective a maximization the number of real-time tasks that can be mapped in a communication port.

The method presented in this section is based on the integer linear programming (ILP) technique. One advantage of this approach is that many different solvers are available.

These solvers bring many advances from the field of optimization and operational research, which are transparent from the point of view of the application. Examples of solvers include, for example, the open source `lp_solve` [65] and even FPGA-based ones such as [18], which aims at increasing performance. An ILP problem consists in finding the optimal value of a cost function whose variables are constrained by linear equations or inequalities. ILP problems are formalized as follows:

$$\begin{array}{ll} \text{maximize} & c^\top x \\ \text{subject to} & Ax \leq b \end{array}$$

In this section, we consider a set Γ of n real-time tasks under EDF. Each task is characterized by a period T_i and a computation time C_i. The objective is to model the relationship among the relative deadlines D by means of linear relationships. With this model, one can employ an ILP solver, for example, to minimize the sum of all deadlines while keeping the task set feasible.

The work by Kim et al. [54] introduced a similar approach, but imposed the constraint that a relative deadline D_i can be larger than D_j if and only if C_i is larger than C_j:

$$\forall i, \forall j, \quad D_i \leq D_j \iff C_i \leq C_j$$

In this thesis the constraint above is relaxed. Relative deadlines are allowed to be in the range $C_i \leq D_i \leq T_i$, as long as the task set is feasible. The upper bound T_i prevents that a new task instance is activated before completion of its predecessor.

$$\forall i: \; 1 \leq i \leq n \qquad D_i \leq T_i \tag{5.1}$$

If a given task set with deadlines less than periods is feasible, then any increase in the relative deadlines up to the period value will not impair feasibility. On the other hand, the actual lower bounds of relative deadlines depend on the parameters of the whole set and cannot be analyzed independently. To formulate the permitted lower bounds of relative deadlines, the processor demand criterion [17], represented by the inequality (5.2), is used.

$$L \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i \tag{5.2}$$

In a task set Γ with deadlines less than periods, it is sufficient to apply the processor demand criterion to the absolute deadlines $d_{i,k}$ of each task instance. So, since $L = d_{i,k}$ and $d_{i,k} = k_i T_i + D_i$, it follows that $L - D_i = k_i T_i$. Moreover, the number of instances of the task i can be restricted to either the busy period B_p or the hyper period H_p of Γ. It is worth mentioning that relative deadlines are irrelevant to calculate both B_p and H_p (see Section 3.5). Then, we can rewrite Eq. (5.2) without the L variable:

$$\forall i: \; 1 \leq i \leq n \qquad \forall k_i: \; 0 \leq k_i \leq \left\lceil \frac{\min(H_p, B_p)}{T_i} \right\rceil$$

$$k_iT_i + D_i \geq \sum_{j=1}^{n} \left(\left\lfloor \frac{k_iT_i + D_i - Dj}{T_j} \right\rfloor + 1 \right) C_j \tag{5.3}$$

Equation (5.3) is, however, not suitable to be utilized as an ILP constraint because the floor function is not linear. To solve this problem, we need first to define mutually exclusive intervals of $(D_i - Dj)$ in such a manner that each of these intervals leads to one different result of $\left\lfloor \frac{k_iT_i+D_i-Dj}{T_j} \right\rfloor$.

The approach to finding such intervals consists in checking how task j interferes with task i. To do this, the first step is to determine the lower and upper bounds of $(D_i - Dj)$. The lower bound is characterized by the minimum possible value of D_i, which is C_i, and the maximum possible value of D_j, which is T_j. Thus, the lower bound of $(D_i - Dj)$ is $(C_i - T_j)$. Analogously, its upper bound is $(T_i - C_j)$. In a general form, the lower $(\alpha_{i,j})$ and upper $(\beta_{i,j})$ bounds of $(k_iT_i + D_i - Dj)$ are defined as:

$$\alpha_{i,j} = k_iT_i + C_i - T_j \qquad \beta_{i,j} = k_iT_i + T_i - C_j$$

If the division of a point in the interval $[\alpha_{i,j}, \beta_{i,j}]$ by T_j results in an integer value, such a point is called a *boundary* due to the fact the value of $\left\lfloor \frac{k_iT_i+D_i-Dj}{T_j} \right\rfloor$ changes at this position. For a given k_i, the set $R_{i,j}$ consists of all results of this floor function in the interval $[\alpha_{i,j}, \beta_{i,j}]$ and is expressed as follows:

$$R_{i,j} = \left\{ x \in \mathbb{N}_0 : \left\lceil \frac{\alpha_{i,j}}{T_j} \right\rceil \leq x \ \wedge \ x \leq \left\lfloor \frac{\beta_{i,j}}{T_j} \right\rfloor \right\}$$

The boundaries of $(k_iT_i + D_i - Dj)$ can be then determined by multiplying each element of $R_{i,j}$ by T_j. Further on, if k_iT_i is subtracted from each multiplication result, we will get the boundaries of $(D_i - Dj)$ for a given k_i. Repeating the process for all values of k_i we finally obtain the set $B_{i,j}$ of the boundaries of $(D_i - Dj)$:

$$B_{i,j} = \bigcup_{\forall k_i} \bigcup_{\forall x \in R_{i,j}} \{xT_j - k_iT_i\} \qquad \text{with:} \quad 0 \leq k_i \leq \left\lceil \frac{\min(H_p, B_p)}{T_i} \right\rceil$$

It is worth mentioning that the function $(D_i - Dj)$ is skew-symmetric, since the property $f(y,x) = -f(x,y)$ holds. So, the set $B_{j,i}$ representing the boundaries of the intervals of $(D_j - Di)$ can be directly derived from the set $B_{i,j}$:

$$B_{j,i} = \bigcup_{\forall x \in B_{i,j}} \{-x\}$$

Alternatively, it is possible to calculate the set $B_{j,i}$ by means of the same procedure described above for $B_{i,j}$:

$$\alpha_{j,i} = k_jT_j + C_j - T_i \qquad \beta_{j,i} = k_jT_j + T_j - C_i$$

$$R_{j,i} = \left\{ x \in \mathbb{N}_0 : \left\lceil \frac{\alpha_{j,i}}{T_i} \right\rceil \leq x \ \wedge \ x \leq \left\lfloor \frac{\beta_{j,i}}{T_i} \right\rfloor \right\}$$

$$B_{j,i} = \bigcup_{\forall k_j} \bigcup_{\forall x \in R_{j,i}} \{xT_i - k_jT_j\} \qquad \text{with:} \quad 0 \leq k_j \leq \left\lceil \frac{\min(H_p, B_p)}{T_j} \right\rceil$$

From the algorithmic perspective, the process of finding $B_{i,j}$ or $B_{j,i}$ can be done more efficiently if the lowest value between k_i and k_j is selected to find one of the sets and then apply the symmetry property to find the reciprocal one. Since one set can be directly derived from another, it actually suffices to consider only one of the sets of boundaries for a given task pair (i, j). As a convention, we store the set $B_{i,j}$, with $i < j$. For a set Γ with n tasks, the total number of (i, j) pairs is then $\frac{n!}{(n-2)! \times 2}$. The Algorithm 2 presents the procedure of calculating the boundary set $B_{i,j}$ of the possible task pairs (i, j). The elements of each set $B_{i,j}$ are sorted in increasing order to facilitate the next steps.

Once the boundaries of the task pair (i, j) are known, it is then possible to find the intervals of $(D_i - D_j)$. The basic idea is to start out from the interval $[(C_i - T_j),\ (T_i - C_j)]$, which represents all possible values of $(D_i - D_j)$ and then, for each element in $B_{i,j}$, break the interval that contains the respective boundary as depicted by Fig. 5.8. Algorithm 3 is employed for this purpose. The result is a set $\chi_{i,j}$ in which each element represents one interval:

$$\chi_{i,j} = \{[B_{i,j_1}, B_{i,j_1}],\ ]B_{i,j_1}, B_{i,j_2}[,\ [B_{i,j_2}, B_{i,j_2}],\ \ldots\ ,]B_{i,j_{w-1}}, B_{i,j_w}[,\ [B_{i,j_w}, B_{i,j_w}]\}$$

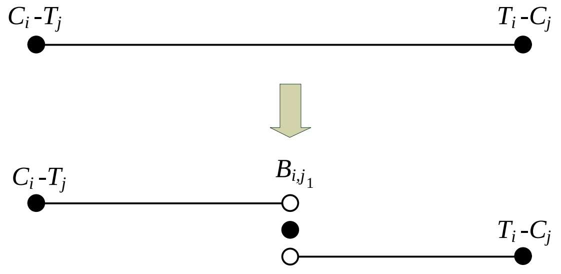


Figure 5.8: Breaking intervals

Due to the symmetry property for the boundaries, the set of intervals $\chi_{j,i}$ can also be directly found:

$$\chi_{j,i} = \{[-B_{i,j_1}, -B_{i,j_1}],\ ]-B_{i,j_1}, -B_{i,j_2}[,\ [-B_{i,j_2}, -B_{i,j_2}],\ \ldots\ ,]-B_{i,j_{w-1}}, -B_{i,j_w}[,\ [-B_{i,j_w}, -B_{i,j_w}]\}$$

All intervals of $\chi_{i,j}$ are mutually exclusive since the value of $(D_i - D_j)$ can be in only one of them. To represent this constraint, we introduce the function $u(x)$, where x is an interval of $\chi_{i,j}$. The function $u(x)$ returns 1 if the result of $(D_i - D_j)$ is in the interval x, or 0 otherwise. The exclusivity constraint for the ILP is given below:

$$\sum_{y=1}^{w} u\left(\chi_{i,j_y}\right) = 1 \qquad \text{where} \quad w = \text{size}(\chi_{i,j}) \tag{5.4}$$

Algorithm 2 Boundaries of Deadlines

```
1: for $i = 1$ to $n$ do
2:    for $j = i + 1$ to $n$ do
3:       $max\_k_i \leftarrow \left\lceil \frac{\min(H_p, B_p)}{T_i} \right\rceil$
4:       $max\_k_j \leftarrow \left\lceil \frac{\min(H_p, B_p)}{T_j} \right\rceil$
5:       $B_{i,j} \leftarrow \{\emptyset\}$
6:       if $max\_k_i \leq max\_k_j$ then
7:          for $k = 0$ to $max\_k_i$ do
8:             $\alpha = kT_i + C_i - T_j$
9:             $\beta = kT_i + T_i - C_j$
10:            $R_{i,j} = \left\{ x \in \mathbb{N}_0 : \left\lceil \frac{\alpha}{T_j} \right\rceil \leq x \,\wedge\, x \leq \left\lfloor \frac{\beta}{T_j} \right\rfloor \right\}$
11:            for all $x$ in $M$ do
12:               $B_{i,j} \leftarrow B_{i,j} \cup \{xT_j - kT_i\}$
13:            end for
14:         end for
15:      else
16:         for $k = 0$ to $max\_k_j$ do
17:            $\alpha = kT_j + C_j - T_i$
18:            $\beta = kT_j + T_j - C_i$
19:            $R_{j,i} = \left\{ x \in \mathbb{N}_0 : \left\lceil \frac{\alpha}{T_i} \right\rceil \leq x \,\wedge\, x \leq \left\lfloor \frac{\beta}{T_i} \right\rfloor \right\}$
20:            for all $x$ in $M$ do
21:               $B_{i,j} \leftarrow B_{i,j} \cup \{-xT_i + kT_j\}$
22:            end for
23:         end for
24:      end if
25:      sort($B_{i,j}$)
26:   end for
27: end for
```

With all intervals and the exclusivity constraint defined it is possible to reformulate the floor function in equation (5.3) as follows:

$$\left\lfloor \frac{k_i T_i + D_i - Dj}{T_j} \right\rfloor \Longrightarrow \frac{k_i T_i + \sum_{y=1}^{w} u\left(\chi_{i,j_y}\right) \chi_{i,j_y}}{T_j} \Longrightarrow \sum_{y=1}^{w} u\left(\chi_{i,j_y}\right) \gamma_{i,j_y}$$

Algorithm 3 Intervals of Deadlines

```
Require: $B_{i,j}$ sorted in increasing order
1: if $B_{i,j} = \{\emptyset\}$ then
2:     $\chi_{i,j} \leftarrow \{\ [C_i - T_j,\ T_i - C_j]\ \}$
3: else
4:     $\chi_{i,j} \leftarrow \{\emptyset\}$
5:     if first$(B_{i,j}) \neq C_i - T_j$ then
6:         $\chi_{i,j} \leftarrow \chi_{i,j} \cup \{\ [C_i - T_j,\ \text{first}(B_{i,j})[\ \}$
7:     end if
8:     for all $x \in B_{i,j}$ do
9:         $\chi_{i,j} \leftarrow \chi_{i,j} \cup \{\ [x, x]\ \}$
10:        if $x \neq$ last$(B_{i,j})$ then
11:            $\chi_{i,j} \leftarrow \chi_{i,j} \cup \{\ ]x, \text{next}(x)[\ \}$
12:        else if $x \neq T_i - C_j$ then
13:            $\chi_{i,j} \leftarrow \chi_{i,j} \cup \{\ ]x,\ T_i - C_j]\ \}$
14:        end if
15:    end for
16: end if
```

For simplicity purposes, the term γ_{i,j_y} is used to represent the value of the floor function for the interval χ_{i,j_y}. It is now possible to rewrite equation (5.3) without the floor function:

$$k_i T_i + D_i \geq (k_i + 1)C_i + \sum_{\substack{1 \leq j \leq n, \\ i \neq j}} \sum_{y=1}^{w} u\left(\chi_{i,j_y}\right)\left(\gamma_{i,j_y} + 1\right) C_j \tag{5.5}$$

The method presented eliminates the floor function by checking the interaction of each task pair (i, j). For sets with three or more tasks it is still necessary to establish a relation among the relative deadlines of all tasks. To accomplish this, we explicitly define the upper and lower bounds of each interval $\chi_{i,j}$ represented by equations (5.6) and (5.7) respectively.

$$\forall y:\ 1 \leq y \leq w \qquad\qquad w = \text{size}(\chi_{i,j})$$

$$D_i - D_j \begin{cases} > C_i - T_j - 1 + \ u\left(\chi_{i,j_y}\right)\left(\text{lb\_val}(\chi_{i,j_y}) - C_i + T_j + 1\right) \\ \qquad\qquad\qquad\qquad \textbf{if } \text{lb\_endpt}(\chi_{i,j_y}) = \text{“]”} \\ \\ \geq C_i - T_j - 1 + \ u\left(\chi_{i,j_y}\right)\left(\text{lb\_val}(\chi_{i,j_y}) - C_i + T_j + 1\right) \\ \qquad\qquad\qquad\qquad \textbf{if } \text{lb\_endpt}(\chi_{i,j_y}) = \text{“[”} \end{cases} \tag{5.6}$$

$$D_i - D_j \begin{cases} \leq T_i - C_j + 1 + \ u\left(\chi_{i,j_y}\right)\left(\text{ub\_val}(\chi_{i,j_y}) - T_i + C_j - 1\right) \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{if } \text{ub\_endpt}(\chi_{i,j_y}) = \text{``]''} \\ < T_i - C_j + 1 + \ u\left(\chi_{i,j_y}\right)\left(\text{ub\_val}(\chi_{i,j_y}) - T_i + C_j - 1\right) \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{if } \text{ub\_endpt}(\chi_{i,j_y}) = \text{``[''} \end{cases} \tag{5.7}$$

The functions lb_val(x) and ub_val(x) return the lower and upper bound values of the interval x respectively. The functions lb_endpt(x) and ub_endpt(x) return the type of the interval endpoints (either open or closed). The ILP constraints created by means (5.6) and (5.7) take into account whether the considered endpoints are open or closed in order to reflect the interval characteristic.

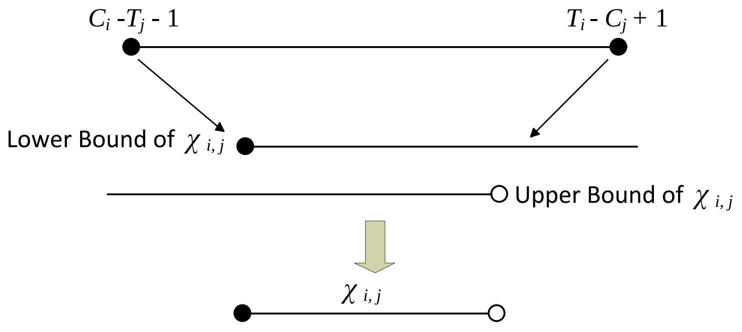


Figure 5.9: Constraining intervals

Figure 5.9 shows in a graphical manner how the $\chi_{i,j}$ intervals are constrained. If the result of function $u(\chi_{i,j_y})$ is 1, then $(D_i - D_j)$ is constrained to that respective interval by increasing the lower bound and decreasing the upper bound. All remaining intervals do not represent any kind of constraint since their lower bounds become lower than $(C_i - T_j)$ and their upper bounds higher than $(T_i - C_j)$.

The ILP constraints for the deadline assignment problem are given by (5.1), (5.4), (5.5), (5.6), and (5.7). For a problem that consists in finding, for example, the lowest possible sum of relative deadlines the cost function is $\sum_{i=1}^{n} D_i$. Alternative cost functions can also be utilized, for instance, to minimize the jitter of the real-time tasks [54]. In the context of this work, however, the main objective is to find a feasible mapping of all real-time channels in a network.

5.3.2 Mapping of Real-Time Channels

The deadline assignment method presented above can be adapted and further extended to support the task of mapping real-time channels in a TrailCable network. To accomplish this, additional constraints are necessary to describe possible routes and the maximum latencies of the channels.

The problem of finding routes in a network can be formulated with ILP by means of the flow conservation method. Its basic idea is that the flow entering a node equals the flow leaving it. In a real-time channel, the flow is generated by the source node and reaches the sink nodes via the communication links. For a given channel, the flow in a communication link (graph edges) is expressed by $e_c^{s,t}$, where c is the channel ID, s and t are pairs of the type (v,p) that represent the origin and destination vertices and ports of that edge. It is worth remembering that since two nodes can be interconnected by more than one communication link, the respective ports are also required to unequivocally represent the chosen link.

Given a network topology, the definition of the route-finding problem for a channel c using the flow conservation method consists in applying equation (5.8) to the source, all sink nodes, and all potential intermediate vertices:

$$f(c) = \sum_{\forall (s,t):\, v \in s} e_c^{s,t} - \sum_{\forall (s,t):\, v \in t} e_c^{s,t} \tag{5.8}$$

where,

$$f(c) = \begin{cases} \text{Number of sinks} & \textbf{when } v \text{ is the source node of channel } c \\ 0 & \textbf{when } v \text{ is an intermediate node of channel } c \\ -1 & \textbf{when } v \text{ is a sink node of channel } c \end{cases}$$

In order to reduce the number of routing constraints, especially for large networks, it is convenient to restrict the set of nodes that can be used to build a real-time channel. One possible criterion is the maximum channel span, i.e., the maximum number of intermediate hops between source and sink nodes. Additionally, application-specific knowledge may be also utilized for the purpose of finding appropriate channel routes.

For the further steps a new variable $bin\_e_c^{s,t}$ must be introduced. As the name suggests, it is a binary representation of $e_c^{s,t}$ meaning whether that edge is being used or not:

$$bin\_e_c^{s,t} = \mathrm{bin}(e_c^{s,t}) \qquad \text{with} \quad bin\_e_c^{s,t} = \{0|1\}$$

The function bin(x) with $x \in \mathbb{Z}$ returns 1 if $x \geq 1$ and 0 if $x \leq 0$. The variable conversion from integer ($e_c^{s,t}$) to binary ($bin\_e_c^{s,t}$) can be done with ILP using the following two constraints:

$$0.5 < e_c^{s,t} + \mathrm{sinks}\,(1 - bin\_e_c^{s,t}); \tag{5.9}$$

$$0.5 > e_c^{s,t} - \mathrm{sinks} \times bin\_e_c^{s,t}; \tag{5.10}$$

With these binary variables it is possible to explicitly impose the condition that a real-time channel can enter a node only via a single communication link:

$$\forall v : \qquad \sum_{\forall (s,t):\, v \in t} bin\_e_c^{s,t} \leq 1 \tag{5.11}$$

Any established route must meet the channel deadline requirement. Thus, the channel deadline equation (3.2) must hold for all sink nodes. In our ILP problem definition, the following inequalities are used to express the channel deadline constraint:

$$\forall (s,t) \quad and \quad \forall p \in s :$$

$$\delta_c^{v \in t} \geq \delta_c^{v \in s} + D_c^{s,t} + (prop_{s,t} + fw_{v \in s} - C_c + \alpha)\, bin\_e_c^{s,t} - N(1 - bin\_e_c^{s,t}) \tag{5.12}$$

The basic idea is that the channel latency at a node equals the latency at the last hop plus the relative deadlines and intrinsic delays of the predecessor node. The constant N is a sufficiently large number that is used to relax the channel deadline constraint when a communication link is not used.

Except for the relative deadline $D_c^{s,t}$, all other parameters in the inequality (5.12) are known. To determine the relative deadlines, we use the method described in Section 5.3.1 with minor modifications. If a communication link is not used by a certain real-time channel, then all relative deadlines $D_c^{s,t}$ are set to zero and the upper bound constraint for the relative deadlines becomes:

$$D_i^{s,t} \leq T_i \times bin\_e_i^{s,t} \tag{5.13}$$

If the channel utilizes that link, then the upper bound of the relative deadline is the period T_i. Moreover, it is also necessary to adapt the inequality (5.5), so that it takes into account just the active real-time channels in a given communication link for the deadlines assignment. To accomplish this, the first thing to do is to introduce the variables $act_{i,j}^{s,t}$ that indicate whether both real-time channels i and j are active in the communication link (s,t):

$$act_{i,j}^{s,t} = bin\_e_i^{s,t} \wedge bin\_e_j^{s,t}$$

Since the operator $\wedge$ (disjunction) is not directly supported in an ILP problem, the following two constraints are employed to obtain the variable $act_{i,j}^{s,t}$ from $bin\_e_i^{s,t}$ and $bin\_e_j^{s,t}$:

$$2\, act_{i,j}^{s,t} < bin\_e_i^{s,t} + bin\_e_j^{s,t} + 0.5; \tag{5.14}$$

$$2\, act_{i,j}^{s,t} > bin\_e_i^{s,t} + bin\_e_j^{s,t} - 1.5; \tag{5.15}$$

The exclusivity constraint (5.4) is used to indicate that only one of the possible intervals of $(D_i - D_j)$ can be selected for a valid solution. Such a constraint only makes sense if both tasks i and j are active and is therefore rewritten as:

$$\sum_{y=1}^{w} u\left(\chi_{i,j_y}^{s,t}\right) = act_{i,j}^{s,t} \tag{5.16}$$

Adapting the constraint (5.5), the lower bound of the relative deadline of channel c in the communication link (s,t) becomes:

$$D_i^{s,t} \geq [(k_i+1)C_i - k_iT_i]\, bin\_e_i^{s,t} + \sum_{\substack{1\leq j\leq n,\\ i\neq j}} \sum_{y=1}^{w} u\left(\chi_{i,j_y}^{s,t}\right)\left(\gamma_{i,j_y}^{s,t}+1\right) C_j \tag{5.17}$$

It should be noted that if a communication link (s,t) is not used by the real-time channel c, then the relative deadline $D_c^{s,t}$ is set to zero by means of the constraints (5.13) and (5.17).

Finally, we alter the constraints (5.6) and (5.7) so that they are valid only when when both tasks i and j are active. Otherwise, the restrictions are relaxed.

$$\forall y:\ 1\leq y\leq w \qquad\qquad w = \text{size}(\chi_{i,j})$$

$$\begin{aligned} &D_i^{s,t} - D_j^{s,t} + u\left(\chi_{i,j_y}^{s,t}\right)\left(C_i - 2T_j - 1 - \text{lb\_val}(\chi_{i,j_y}^{s,t})\right) \\ &\qquad -(C_i - T_j)act_{i,j}^{s,t} + (T_j+1)bin\_e_j^{s,t} \begin{cases} > 0 & \textbf{if } \text{lb\_endpt}(\chi_{i,j_y}^{s,t}) = \text{``]''} \\ \geq 0 & \textbf{if } \text{lb\_endpt}(\chi_{i,j_y}^{s,t}) = \text{``[''} \end{cases} \end{aligned} \tag{5.18}$$

$$\begin{aligned} &D_i^{s,t} - D_j^{s,t} + u\left(\chi_{i,j_y}^{s,t}\right)\left(2T_i - C_j + 1 - \text{ub\_val}(\chi_{i,j_y}^{s,t})\right) \\ &\qquad -(T_i - C_j)act_{i,j}^{s,t} - (T_i+1)bin\_e_i^{s,t} \begin{cases} \leq 0 & \textbf{if } \text{ub\_endpt}(\chi_{i,j_y}^{s,t}) = \text{``]''} \\ < 0 & \textbf{if } \text{ub\_endpt}(\chi_{i,j_y}^{s,t}) = \text{``[''} \end{cases} \end{aligned} \tag{5.19}$$

Optionally, it is possible to include a further constraint to cope with fault tolerance by assuring that routes of specific real-time channels are disjoint. The group of such real-time channels is called disjoint path group (DPG) and the constraint is expressed as:

$$\forall v:\quad \sum_{\substack{\forall (s,t):\, v\in t,\\ \forall c:\, c\in DPG}} bin\_e_c^{s,t} \leq 1 \tag{5.20}$$

Additional constraints can be used to represent the limitations of the TrailCable communication engine hardware. For example, it can be checked whether the amount of available memory is sufficient to buffer all real-time tasks in a certain node. Moreover, the number of real-time tasks in a communication link is limited by the scheduler capacity. The larger the number of tasks that can be handled simultaneously, the higher the number of available packet IDs:

$$\sum_{\forall c} bin\_e_c^{s,t} \leq \text{IDs} \tag{5.21}$$

One possible objective function for this ILP problem consists in minimizing the sum of deadlines of all real-time channels:

$$\text{min:} \quad \sum_{\forall v \, \in \, V} \delta_c^v \tag{5.22}$$

The outcome of this objective function is that the channel routes will be as direct as possible and all relative deadlines will be set to the lowest possible values. In order to facilitate the inclusion of further real-time channels during run-time without the need to alter the network configuration, relative deadlines can be increased as long as the resulting channel deadlines remain lower than the requirement of the application.

For the sake of simplicity, preemption overhead was not considered in the presented model. One manner to cope with preemption is to add the respective overhead in the channel deadlines after all communication channels are mapped in a network. Although this method may not guarantee mapping optimality, it is simple and can be performed efficiently.

5.3.3 The *Fit Minimum Laxity First* Algorithm

The method based on ILP for finding valid mappings of real-time channels presented in the previous section has the advantage of finding possible solutions even for a high utilization factor of the communication bandwidth. If preemption overhead can be neglected, the ILP model would be optimal in the sense that if there is a valid configuration, then it can be found. The main drawback of the method, however, is the required processing time, especially for complex configurations. In order to reduce the complexity, one may impose some restrictions on the requirements of the real-time channels. One example is to apply a rule that all periods of the real-time channels must be harmonic, i.e., $T_c = kT_h$, where T_c is the period of a channel, T_h the harmonic period, and k an integer constant.

To overcome the limitations of the ILP method, in this section we introduce a new algorithm called *Fit Minimum Laxity First* or FMLF. Although this algorithm is not intended to yield an optimal mapping configuration of a TrailCable network, its run-time complexity is considerably lower, which makes it appropriate for many systems requiring dynamic reconfiguration.

In the real-time scheduling theory, laxity X_i, or slack time, is a parameter that indicates the time a task can be delayed by others and still meet its deadline; it is given by $X_i = D_i - C_i$. In the scope of this work, however, we will utilize the term laxity to express the time difference between the real-time channel deadline required by an application δ_{req} and the time needed to transmit the packet in the best case δ_{best}, i.e., when no other channels exist in the network and the most direct route is taken.

The FMLF algorithm is executed in 5 steps that are listed below:

1. **Calculate the laxity of the real-time channels.** Clearly, the FMLF algorithm starts out with calculating the laxity of all real-time channels that need to be mapped in a given network. If a channel has multiple destinations, the laxity of each sink node is calculated individually. In order to calculate the laxity, the first step is to find the shortest routes to the sink nodes of all channels. This can be done by means of the breadth-first search (BFS) algorithm. The complexity of BFS in the worst case is $O(|V| + |E|)$, where $|V|$ represents the number of nodes (vertices) and $|E|$ the number of communication links (edges) of the network part considered. The BFS algorithm must run for each source node of a real-time channel and therefore the overall runtime complexity to find all shortest paths can be limited, without any further optimizations, to $O(\ |S|\ (|V| + |E|)\ )$, where S is the set of source nodes of all real-time channels.

   Once the shortest routes are known, the lowest achievable channel deadlines δ_{best} can be calculated by means of equation (3.2). The laxities of each sink node v of a channel c are then available: $X_c^v = \delta_{req}{}_c^v - \delta_{best}{}_c^v$.

2. **Sort the real-time channels in the non-decreasing order of laxity.** The heuristic of the FMLF algorithm consists in mapping real-time channels with the most stringent deadline requirements first. The basic idea is that mapping the "harder" channels first will make it easier to map the remaining ones. The sorting process has a runtime complexity of $O(n\ log\ n)$ and its output is a linked list containing the sorted laxities with the respective channel identifiers.

3. **Select the real-time channel with the lowest laxity and perform incremental schedulability tests.** This step in the algorithm checks which communication links still have enough capacity to include the selected real-time channel. If a communication link can be used, then the lowest possible relative deadline for the corresponding EDF scheduler is also calculated by means of a similar process as used in the ILP problem definition of Section 5.3.1. The basic difference, however, is that once a real-time channel is mapped, its relative deadlines are fixed and therefore the complexity of the ILP problem is considerably reduced because only one new relative deadline will have to be assigned during each iteration.

   This step of the algorithm can be distributed among the nodes of a network. In this case, a master node broadcasts to all potential intermediate hop nodes the period T_i and communication time C_i of the real-time channel. The slave nodes then check whether the real-time channel can be admitted in their communication ports and, if so, determine the lowest possible relative deadline for each of the schedulers. Finally, the slave nodes this information back to the master.

4. **Find the route for the real-time channel.** Once the feasible communication links and their respective relative deadlines are known, it can be checked whether

a route is available. This can be done by means of the A* algorithm [38], with the relative deadlines representing the edge weights of the network graph. This step finds the route with the least channel deadline, if there be any, and uses it. If two or more routes have the same channel deadline the route with less communication links will be chosen. A route, however, is only established if the achievable channel deadline is equal or lower than the required one. If a route to another sink node of the same channel is already mapped, the existing path or a part of it can be used.

In order to increase the probability of success when mapping the next channels, the relative deadlines that will be used for the EDF schedulers along the new route are increased as much as possible (by a factor Δ) without the channel deadline being missed.

$$\sum_{\forall v \in Route} (D_c^v + \Delta) = \delta_{req_c} \tag{5.23}$$

The new relative deadlines are then sent to the respective nodes, so that the real-time task can be activated.

5. **Select the next real-time channel, if any, and return to step 2.** The previous steps are repeated until all real-time channels in the list are processed.

5.4 Chapter Summary

This chapter dealt with the dynamic reconfiguration of the TrailCable protocol. In the first part, a reconfiguration framework was presented. Such a structure allows the use of the TrailCable protocol with dynamic applications and makes it possible to integrate other domain-specific structures such as OCM [9].

The second part of this chapter was dedicated to methods that find valid TrailCable configurations. The first method was based on integer linear programming (ILP) and is aimed at finding optimal mappings of the real-time channels, i.e., find feasible configurations if they exist. This is especially useful when the utilization factor of the communication links are rather high. Due to the runtime complexity of the ILP method, an alternative algorithm was introduced, which sacrifices optimality in order to find possible mappings in less time.

The use of dynamic reconfiguration has many potential advantages in an embedded system. For example, it makes possible to dynamically change the data rate of the communication links according to its demands to save power. Another use of dynamic reconfiguration is to increase the tolerance to faults, since once faulty communication links or nodes are detected, the routing of the real-time channels can be automatically altered if alternative paths are available.

Chapter 6

The Communication Engine Hardware

Due to the approach of the TrailCable protocol that relies on hardware scheduling and low-level optimizations to achieve its performance and determinism goals, a new communication controller was developed from scratch, rather than employing commercial off-the-shelf (COTS) components. Reasons for this decision were, among others, the following protocol features:

Scalable Scheduler In order to allow handling multiple communication tasks simultaneously, a hardware-based EDF scheduler is necessary to deliver constant performance, regardless of the number of tasks.

Packet Preemption To fully profit from the EDF scheduling strategy, a mechanism allowing preemption of the data communication packets at the physical layer is of great importance.

Admission Control Upon the reception of a data packet header, some of the task characteristics are checked to assure that they are consistent with the specifications assumed for the schedulability analysis. This procedure must be executed as fast as possible so that by the time the packet header transmission is completed, a decision on acceptance is immediately available. Since this operation takes only a few clock cycles and is carried out simultaneously in different communication ports, a hardware solution is required.

Low Overhead Since the main application area of the TrailCable communication protocol is the domain of embedded systems, a concise frame format is used, reducing the size of headers and control fields. The price of a relatively simple header used in the

TrailCable protocol is additional processing required to fetch the local scheduling tables, which are spread among small RAMs in order to increase the total memory bandwidth.

Clock Synchronization Even though not necessary for the TrailCable protocol to achieve real-time behavior, global clock synchronization can be required for supporting higher-level functionalities. For this reason a time-unit was integrated into the hardware design to allow precise time synchronization of a set of network nodes. Since the time-stamps are very precise (with a one clock cycle resolution), hardware support is once again indispensable.

The functions listed above are implemented by means of a dedicated hardware architecture called *Communication Engine* (CE), which encapsulates the digital circuits required for an autonomous operation of the TrailCable protocol. This chapter introduces the details of the communication engine architecture, discusses its actual implementation in FPGA devices and presents a design space exploration study to demonstrate the impact of different hardware configurations on the required amount of hardware resources.

6.1 Communication Engine Architecture

The communication engine is built up with three main components: communication port(s), a host port, and an interconnect bus (Figure 6.1). The communication ports are responsible for receiving and checking incoming data from the links as well as scheduling and sending data packets to the physical medium. Besides implementing an interface to the host microprocessor, the host port manages the memory storage and retrieval of all real-time data packets traversing, leaving, or entering the node. Another function of the host port is to carry out clock synchronization by means of a time-unit. Finally, the interconnect bus is in charge of transferring data among several communication and host ports. For this purpose, three independent buses are used: a control bus that transfers incoming data packets and the corresponding control information from one port to the real-time module and other ports, a real-time data bus that transfers real-time data from storage memories to the communication ports, and one that is intended to distribute non-real-time data for transmission.

Due to the fact that the architecture is built up in a modular manner, with the help of the VHDL language constructs it was possible to parameterize some properties of the implementation. Therefore, the system designer is able, for example, to determine the number of communication ports needed for different nodes. For low-cost and simple applications where redundancy is not necessary, it is possible to define a communication engine with only a single communication port, which requires few hardware resources and fits into small FPGAs (which potentially also contain application-specific logic). On the

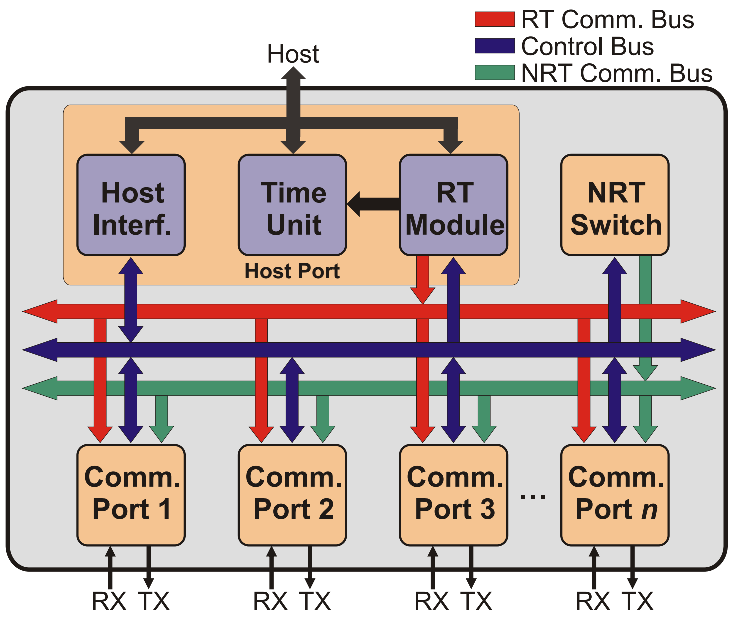


Figure 6.1: Communication Engine Architecture

other hand, multiple communication ports can be used, which allows the construction of complex network topologies.

The following sections give a detailed description of the three parts of the communication engine: the host port, the communication port, and the interconnect.

6.2 Host Port

The host port is responsible for the following functions: implementing an interface to allow data exchange with a host microcontroller (Host Interface), providing a clock synchronization service and time-related functions (Time Unit), and managing internal memory banks where data packets are stored (Real-Time Module). The three modules are described below.

6.2.1 Host Interface

The data transmission procedure initiated by the host processor is divided into two steps: writing data packet contents to the internal memory of the data communication engine and then triggering the transmission. There are two different modes of triggering the transmission: one is a commanded triggering by the host controller itself. In this type of operation, the packet is immediately handed over to the scheduler and then transmitted according to its priority. The second mode of starting a transmission is by means of time-triggering. In this case, periodic signals generated by the time unit are used to trigger the transmission of pre-selected packets. When clock synchronization is available, such periodic signals are also synchronized in all participating nodes, allowing a coordinated data communication.

When it comes to data reception, the communication engine is able to issue interrupts to the host processor either upon reception of pre-defined data packets or at specific times (synchronized operation). These modes allow both time- and event-triggered data communication to occur in a concurrent manner, thus contributing to a high flexibility in a distributed system based on the TrailCable protocol.

Another function of the Host Interface is to initiate the process that measures the propagation delay of the communication links. The basic scheme of this functionality is depicted by Figure 6.2. The process starts when a host issues a delay measurement request. Once the neighboring node receives this request, it acknowledges the source host with another data packet. When this packet arrives at the source node its timestamp contains the time spent in both source and destination communication engines. Since the source node keeps the exact time the request packet was triggered and also the time when the acknowledgment arrives, it is possible to determine the propagation time of the links.

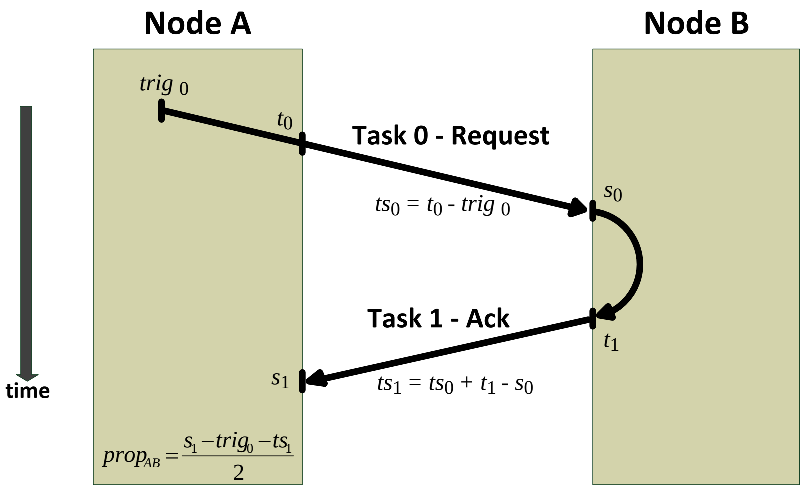


Figure 6.2: Propagation delay set-up

Table 6.1 presents the flow of the link propagation retrieval process. Within a communication port, the task IDs 0 and 1, respectively, are reserved for the request and acknowledgment of data packets. This is necessary because for the transmission of these packets the timestamps cannot be updated with the link propagation delays.

6.2.2 Time Unit

The main function of the time unit is to establish a fault-tolerant clock synchronization for a set of nodes in the network (See Section 3.7). Timestamps are used to exchange timing

Table 6.1: Measurement of propagation delay

Node A	**Node B**
1) Trigger task 0 transmission	
2) Save the triggering time of task 0 ($trig_0$)	
3) Save transmission start time of task 0 (t_0)	
4) Set the time stamp of task 0: $ts_0 = t_0 - trig_0$	
	5) Receive task 0 and save its reception start time (s_0)
	6) Acknowledge task 0 reception by sending task 1
	7) Save transmission start time of task 1 (t_1)
	8) Set the time stamp of task 1: $ts_1 = ts_0 + t_1 - s_0$
9) Receive task 1 and save its reception start time (s_1)	
10) Calculate the link propagation delay from node A to node B: $$prop_{ab} = \frac{s_1 - trig_0 - ts_1}{2}$$	

information among nodes. The timestamps used for clock synchronization are updated with both the propagation delay of the links they pass by and with the time spent in each communication engine. Thus, the timestamps represent the elapsed time between the transmission triggering at the source node and the final reception at the destination node(s).

In a synchronized system, the transmissions of data packets are triggered at globally known times. Therefore, by means of the timestamps each node participating in the clock synchronization is able to determine the difference between the local and remote time references with high precision.

As regards hardware, it is simpler to employ the median rather than the mean value of the time differences to determine the offset correction factor for the local node. The reason is that calculating the mean requires two steps: sorting the time differences (in order to eliminate the highest and lowest deviations) and then accumulate all values and divide them by the number of entries. On the other hand, finding the median dispenses with the second step, as it suffices to take the mid value of the sorted list. In practice, although the mean could yield better clock synchronization results, the difference to the median alternative is potentially negligible in many applications. Hence, the decision to use a simpler and faster hardware component can be justified.

The Time-Unit is also responsible for generating interrupts to the host processor at predefined times in order to support a time-triggered operation of the system when the clock synchronization service is employed.

6.2.3 Real-Time Module

The payload data of incoming real-time packets are stored in internal memories located in the Real-Time Module. Besides storing data, the Real-Time Module is responsible for checking whether the CRC of received payload data is correct. If not, the packet will be marked as faulty and its transmission will be canceled immediately in order to prevent failures propagation. The Real-Time Module also creates the CRC for the payload data of the packets whose origin is the local host. Moreover, another function of the Real-Time Module is to retrieve the timestamps from the incoming data packets and hand them over to the Time-Unit. Finally, it is also the role of the Real-Time Module to update timestamps of packets being transmitted with the propagation delay of the addressed output link.

The implementation of the Real-Time Module is fully pipelined to allow multiple communication ports to write and fetch data in a concurrent manner.

6.3 Communication Port

For each link, the communication ports (Figure 6.3) handle both data reception and transmission. In order to perform these tasks, the communication ports are divided into different layers. On the receiving side, incoming data streams are first decoded and parallelized. The receiver passes each single control or data byte to the next layer. It is then a task of the admission control block to interpret such incoming information and reconstruct the packets considering possible preemptions in the data stream. Moreover, upon detection of *START* or *RESUME* headers, the admission control block performs acceptance tests to verify whether incoming packets are syntactically correct and have the expected temporal behavior. When an acceptance test is successful, the data packet will be allowed into the internal buses, otherwise it will be discarded.

When it comes to data transmission, a scheduler selects the highest-priority task for transmission. The dispatcher has then to fetch the data contents from the Real-Time

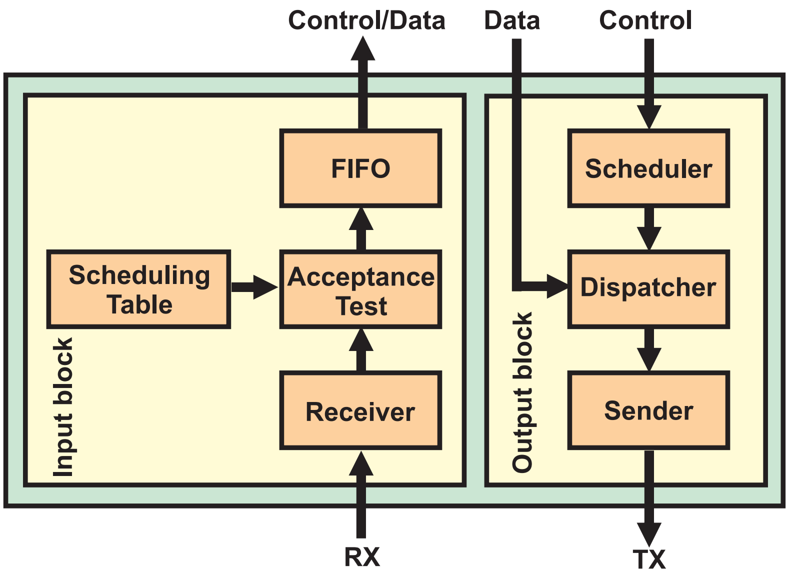


Figure 6.3: Communication port

Module, create the packets, and pass them over to the sender unit, where data is coded and serialized for transmission. At any given time, if a higher-priority task is selected by the scheduler, the dispatcher needs to start the transmission of the new packet by sending the appropriate header. A detailed description of the components of the communication ports will be presented in the following sections.

6.3.1 Physical Layer

TrailCable was designed to allow the use of a variety of standards at the physical layer as long as they support a coding scheme like 8B/10B. The data links are required, however, to be bidirectional, point-to-point and full-duplex. Among different alternatives, the Low Voltage Differential Signaling (LVDS) was chosen for the first implementations because of the following benefits: high data rates, noise immunity, low power consumption, and increasing adoption by manufacturers of integrated circuits.

LVDS is defined by the ANSI/TIA/EIA-644-A standard [83] and is becoming more and more popular for high-speed data communication applications. The low voltage swing (approximately 350 mV) specified for the LVDS standard brings some benefits. The first is low-power operation, with a dissipation slightly higher than 1 mW for a transmission pair. For comparison, the RS-422 standard, which is also differential, draws about 70 times more power than LVDS. Moreover, a low-voltage swing is appropriate for higher data rates since it is possible to change states faster when compared to technologies with higher voltage excursions. Additionally, for given a signaling rate, the lower the voltage swing, the lower the necessary slew rate, which in turn helps to reduce the amount of dissipated electromagnetic energy and therefore to enhance EMC. Since LVDS relies on differential data transmission, interferences that appear simultaneously in both wires are suppressed, which makes this standard perform well with environmental noise.

There is currently a trend for the major FPGA suppliers to support the LVDS standard in their devices. The TrailCable protocol profits from this scenario, since in many cases the only integrated circuit required for its implementation can be an FPGA that holds both the protocol engine and the LVDS drivers.

Although there are many advantages for using the LVDS standard, it imposes a limitation on the maximal cable length, which depends on the transmission rate. Typically, a 100 Mbps LVDS link can be no more than about 30 meters long. For longer link distances, the TrailCable can also use optical transmission. Another advantage of using fiber optics as a medium is noise immunity, allowing the employment of TrailCable in very noisy electrical conditions, such as high-voltage installations or switching, high-power electronics equipment like servo devices.

Since the 8B/10B coding scheme is DC-balanced and guarantees a sufficient amount of level transitions, it is also appropriate for optical transmissions and no modifications are required for the TrailCable logic to accommodate either LVDS or a fiber optic communication link. The only differences are at the hardware level, with the respective transceivers for each technology. This feature allows opting for the transmission medium which better suits the application needs.

6.3.2 Receiver

After passing by the physical layer circuitry, the serial data stream is transformed into a medium-independent sequence of bits, which is the input of the receiver logic implemented in the FPGA.

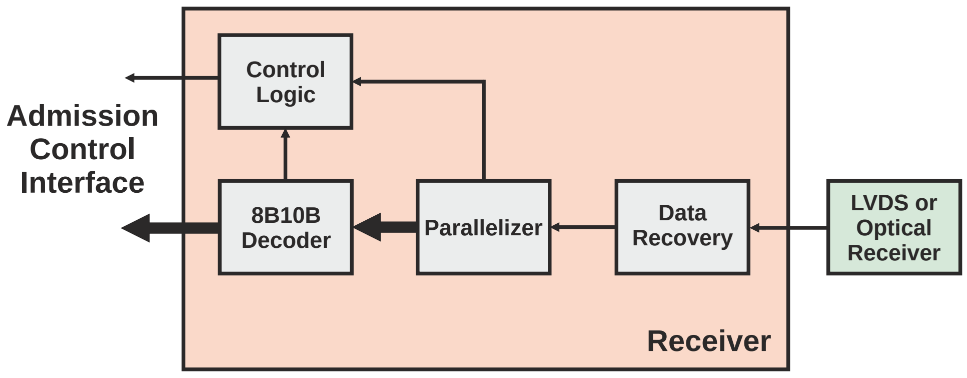


Figure 6.4: Receiver unit architecture

The nominal data transmission rate is equal to the receiver clock frequency. In order to cope with phase differences and frequency drift, the receiver synchronizes the data stream with the local clock domain. This is performed by the data recovery block, whose implementation is based on the Xilinx Application Note 224 [78]. By means of a 4-times oversampling of the input data stream, the data recovery block is able to detect the signal transitions and sample each bit at the correct time. At each local clock cycle, the data recovery block outputs either one bit (normal case), two bits (if the transmission rate is slightly higher than the receiver clock) or no bit at all (if the transmission rate is slightly lower than the receiver clock).

The subsequent receiver stage is the parallelizer, which feeds a 8B/10B decoder. The parallelizer searches and detects 8B/10B comma control bit-sequences in the synchronized data handed over by the recovery block. By this, it is possible to determine exactly where every 10-bit-coded byte starts. Successive transmission errors may lead to a loss of synchronization that prevents the parallelizer from picking up the correct 10 bits of an encoded byte. However, it is possible to recover from this situation as soon as a new comma control byte is detected.

The last stage in the receiver is the 8B/10B decoder, that in addition to decoding the 10-bit word to the equivalent byte, informs the following modules whether the decoded word represents data or a control comma. For the latter case, the 8B/10B decoder also indicates if the control word is a *START* or a *RESUME* comma.

6.3.3 Admission Control

The admission control component implements the behavior described in Section 3.6. This block analyses the incoming data from the receiver in order to locate where packets begin, terminate, preempt or resume, according to the TrailCable protocol semantics. Afterwards, the admission check itself takes place. The design rationale for the implementation of the admission control logic was to permit all the required processing to be executed immediately, i.e., before the first payload byte is received. This was made possible by an optimized hardware architecture specifically designed for this purpose. As a result, there are no possible combinations of received data that could lead to a processing overload, thus impairing the real-time behavior of the communication system.

The hardware architecture that performs admission control consists of two main blocks: A scheduling table and an admission control unit. For each communication task, the scheduling table holds specification parameters and run-time variables.

As the name suggests, the specification parameters define the characteristics of the real-time tasks used for the schedulability analysis. These parameters are read-only for the communication engine hardware. Only the host is allowed to write these parameters. The specification parameters are: output ID, frame size, forwarding table, operation flags, T_i (period), D_i (relative deadline), and G_i (guard).

Run-time variables are updated by the communication engine hardware and used to store information about the current status of the real-time tasks. The run-time variables are: H_i (hold), $s_{i,j}$ (start time), $cs_{i,j}$ (corrected $s_{i,j}$), $f_{i,j}$ (finish time), and CRC.

6.3.4 Scheduler

The scheduler is responsible for selecting for transmission, within a set of active real-time tasks, the one with the highest priority. As described in Section 3.5, the EDF algorithm was chosen for this purpose. The main functionality of the scheduler is therefore to order real-time tasks with respect to their absolute deadlines so that at any given time the highest-priority task is known.

The scheduler has an interface to allow insertion or removal of active tasks. New insertion requests come from other communication or host ports while removal requests are initiated by the dispatcher whenever the transmission of a frame ends.

One requirement in the design of the TrailCable schedulers was to limit their response time (i.e., the interval between changing the scheduler task set and finishing the priority ordering process) to the transmission time of one packet header. By this, it is assured that when the dispatcher finishes the transmission of a header, it will be able, if necessary, to start the transmission of a new one immediately. The scheduler response time accounts for a substantial portion of the forwarding delay of the node, fw_n. Therefore, the higher the response time, the higher also the latencies of the real-time channels. In the TrailCable communication engine implementation presented in this chapter, the transmission time of a header takes 750 ns (or 30 clock cycles), and thus the scheduler response time must be kept under this value at all times. As a consequence, the forwarding delay fw_n is very small, about 2 μs for a 32 Mbps bit rate. As a matter of comparison, this is less than that of many commercial Ethernet switches operating at 100 Mbps [48, 49].

The scheduler response time is also crucial for maintaining consistency between the hardware implementation and the assumptions made for the EDF schedulability analysis. The feasibility tests do consider the extra time spent for preemptions but do not take into account large scheduler response times. In fact, for the TrailCable schedulers the response is immediate whenever a newly arriving task has the highest priority, since a simple comparison between the deadlines of both the current and the new task is sufficient to determine the highest-priority task.

Summarizing the requirements of the scheduler response time for the TrailCable protocol, it follows that:

- Response time if the new task has the highest priority one: immediate.
- Response time for all other cases: $<$ 30 clock cycles.

The next sections give a deeper insight into the TrailCable scheduler hardware, which was specifically designed to meet the above requirements.

6.3.4.1 Priority Queue

Consider the task of inserting a new entry into a list of ordered values. This new entry must be placed in such a position that the list order be maintained. For a list with n elements, a software implementation of this function using the classical binary search algorithm has a complexity $O(log\ n)$. However, even for small values of n the execution time of this algorithm is rather high to handle multiple (and possibly simultaneous) incoming scheduling requests arriving at a given communication port. Additionally, the fact that the execution time of such algorithm depends on n is an inconvenient when it comes to scalability: the larger the number of tasks supported by the scheduler, the higher are its response times.

In order to overcome these limitations, the TrailCable protocol utilizes a hardware-based sorting mechanism for the priorities of the tasks. Some of the possible alternatives for the implementation of this component, often called priority queue, are presented in [46, 53, 75, 66], which were mainly developed for high-speed networks. For the TrailCable protocol, however, the aim is at reducing the amount of resources needed to construct the queues while offering a scalable architecture with good performance.

Priority Queue Architecture

The TrailCable priority queue architecture consists of a control logic block and multiple queue modules (Figure 6.5). The queue modules have an internal memory where the priority values are stored. Modules can be daisy-chained in order to increase the priority queue capacity.

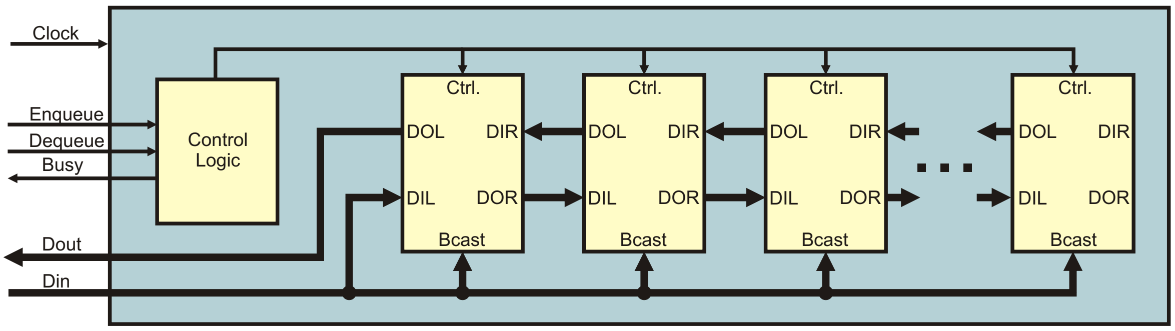


Figure 6.5: Priority queue architecture

The execution times of the two basic priority queue operations, enqueue (which inserts a new entry and reorders the queue) and dequeue (which removes the highest-priority entry from the queue) have, thanks to the employed hardware architecture, run-time complexity $O(1)$ (constant) regardless of the number of modules. As a matter of fact, the execution time is proportional only to the size of the individual modules. Therefore, the presented architecture is extremely scalable. The price paid for the scalability is an increase in hardware resources for extending the priority queue, which is linear with respect to its size.

Priority Queue Control Block

The control logic block is the component responsible for receiving the priority queue interface commands (enqueue or dequeue) and generating the appropriate control signals to queue modules. An interesting fact about the control block is that it does not have to be modified at all if queue modules are added to or removed from the architecture. This is due to the fact that the operations performed by all queue modules are always synchronized and identical as they use exactly the same signals from the control logic block. Keeping one single centralized control logic block for the entire priority queue also contributes to reducing the required area on the chip and improving the overall architecture resource efficiency.

The interface of the priority queue consists of the following signals: *Enqueue*, *Dequeue*, *Busy*, *Din*, and *Dout*. *Enqueue* or *Dequeue* signalize the beginning of the respective operation. *Busy* is set by the control logic during execution of the specified operation. *Din* is used to insert the value in an enqueue operation and *Dout* returns, in a dequeue operation, the highest priority value stored in the queue.

In order to show how the enqueue and dequeue operations are executed in each queue module, the respective algorithms 4 and 5 are presented. They are implemented by means of a state machine in the control logic block.

For the enqueue operation, the first phase consists in checking in each queue module whether the new input value (*Bcast*) is larger than the highest value stored in the module. If so, the module outputs its lowest value to its right-hand neighbor via the *DOR* output. Otherwise, the value passed to the neighbor is the input value (*Bcast*). In the next step, all modules read the value generated in the first phase by their left-hand neighbor and store it in a local register. The last phase consists in the sorting process itself, which is performed by successively comparing the value of the register with each entry in the queue modules. When the value of the register is greater than the current queue entry, both are swapped. The process continues until all the values of the local queue are compared.

Algorithm 4 Enqueue

```
1: integer  REG,  queue[1..n]
2: input    BCAST,  DIL
3: output   DOR
4:
5: if (BCAST  >  queue[1]) then
6:    DOR  ←  queue[n]
7: else
8:    DOR  ←  BCAST
9: end if
10: REG  ←  DIL
11: for i  = 1 to n do
12:    if  (REG  >  queue[i])  then
13:       swap(queue[i], REG)
14:    end if
15: end for
```

Algorithm 5 Dequeue

```
1: integer  REG,  queue[1..n]
2: input    DIR
3: output   DOL
4:
5: DOL  ←  queue[1]
6: REG  ←  DIR
7: queue[i..n-1]  ←  queue[2..n]
8: queue[n]  ←  REG
```

Priority Queue Module

The designed queue module (Figure 6.6) has storage elements (one memory and one register), one comparator, one multiplexer, and one *Write Enable Logic* block, which consists of a simple combinational circuit for generating the write enable signals for both the memory and the register.

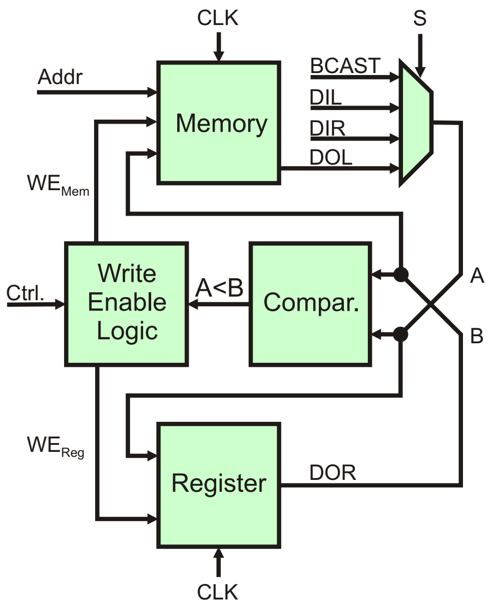


Figure 6.6: Queue module

The module has three data inputs (DIL, DIR, and *Bcast*) and two data outputs (DOL and DOR). The *Bcast*, DIL, and DOR data interfaces are used during the enqueue operation, whereas DOL and DIR are used for the dequeue operation. *Addr*, S, and *Ctrl* are signals generated by the control block. Their functions are the following:

- *Addr* - current memory address,
- S - channel selection signal for the multiplexers,
- *Ctrl* - control signals used to indicate whether the write enable signals for the memory and register are forced to specific values or obtained by taking into account the current comparison results.

The architecture of the queue module supports the enqueue algorithm introduced in algorithm 4 as follows: First of all, the lowest-priority value stored in the memory is transferred to the register by addressing the memory, setting the multiplexer input channel to DOL, and generating a write enable signal to the register. In the next step, the multiplexer must be set to output the *Bcast* value so that the latter can be compared with the register contents. If the priority of *Bcast* is higher than the current register content, the former will be copied to the register, overwriting the previous lowest priority. At this point, the register contains either the lowest priority of the queue module or the *Bcast* input, whichever has the highest priority. This procedure corresponds to lines 5 to 9 of the enqueue algorithm. The third phase consists in writing the DIL input unconditionally to the register (line 10 of the enqueue algorithm). The following steps execute the sorting process (line 11 to 15 of the enqueue algorithm) by comparing the register value to the memory contents successively. If the priority stored in the register is higher, the *Write Enable Logic* block generates "write" commands to both memory and register simultaneously, which causes the contents to be swapped between them. This operation is repeated until the last position of the memory (which contains the lowest priority stored in the module) is reached. Each comparison step can be accomplished in just one clock cycle. As a result, the total number of clock cycles necessary to sort the queue is 4 (initialization steps) plus the depth of the memory used. In the current TrailCable implementation the enqueue operation always takes 20 clock cycles since it employs a 16-address memory. So, if additional queue modules are daisy-chained, the priority queue capacity will increase in steps of 16, but the duration of the enqueue operation remains the same.

The dequeue operation (Alg. 5) is considerably simpler. Each module outputs to the left-hand neighbor the highest-priority entry stored locally (via DOL), simultaneously reads from the right-hand neighbor (via DIR) the new lowest-priority entry for the local module, and stores it into the register. The shifting operation shown in the dequeue algorithm (line 7) can be implemented by simply incrementing a pointer to the memory address of the highest-priority entry. Finally the register value is copied to the lowest priority memory address. The dequeue operation takes only 4 clock cycles to execute.

Commanding simultaneous enqueue and dequeue operations is also possible. Removal of the entry from the priority queue is done prior to the insertion. Since this a combination of both enqueue and dequeue operations, it takes 7 clock cycles for initialization plus one extra clock cycle for each memory address. Table 6.2 summarizes the execution latency of all operations.

Table 6.2: Priority queue execution cycles

Operation	**Execution Cycles**	**TrailCable Scheduler Implementation**
Enqueue	4 + memory depth	20
Dequeue	4	4
Both Simultaneous	7 + memory depth	23

The memory depth of 16 chosen for the TrailCable implementation is a trade-off between latency and hardware area. As regards latency, Table 6.2 shows that the operations take at most 23 cycles to complete, which is quite reasonable for the application considering that the transmission of a packet header takes 30 cycles. As regards hardware resources, memories with a depth of 16 can be very efficiently implemented in FPGAs. The slices of the Spartan-3 Xilinx FPGAs [91], for instance, have two main components: a look-up table and a flip-flop. The so-called *Distributed Memory* feature allows the use of these look-up-tables for implementing a 1-bit-wide RAM memory with a depth of 16. As a result, it is possible to build an n-bit x 16 memory with only n FPGA slices, instead of $n \times 16$. Newer FPGA families, such as the Xilinx Spartan-6 [92], allow using one look-up table to build either one 1-bit x 64, or two 1-bit x 32 memories.

Priority Queues for Deadline-Based Schedulers

When employing the EDF scheduling algorithm, priority queues are used to sort the deadlines of all active tasks in ascending order. Since the absolute value of deadlines get higher and higher over time and because the priority queues entries have a restricted size, data overflows will eventually occur.

The first straightforward solution for coping with overflows would be to dimension the priority queue entries so that no overflows can happen during the expected operation lifetime of a given system. Nevertheless, this strategy has the disadvantage of considerably increasing the required hardware resources. Another possibility to deal with overflows, which is used in the TrailCable communication engine, is to adapt the priority queue hardware architecture in such a way that overflows cannot impair the correct operation of the sorting function. In the latter approach, the bit-width of the priority queue entries is dependent on the maximum relative deadline of the scheduler, which is many orders of magnitude smaller than the operation lifetime of system.

In order to cope with overflows, the maximum relative deadline supported by the scheduler must be restricted. The necessary condition is that a relative deadline, D, cannot be larger than half of the range of the time base used by the system:

$$\max(D_i) = \frac{\text{range}(\textit{time base})}{2} \tag{6.1}$$

The time base is a counter that is incremented periodically and represents the absolute time of the scheduler. In the TrailCable implementation presented in this chapter, the time base is incremented at each clock cycle and has a resolution of 25 ns.

Another assumption required to handle overflows correctly is that if a deadline expires (its value reaches the current time), it will be removed immediately from the priority queue. This situation is not likely to occur with the TrailCable protocol since schedulability analyses are performed in advance for each scheduler. If, however, this condition do occur due to some kind of fault, the hardware must be able to recover from this unwanted state as soon as possible.

To exemplify the method of handling overflows, we consider a simple system having a time base implemented with a register of 3 bits, counting repeatedly from 0 to 7. As a

Table 6.3: Example: Handling overflows

Time Base			
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

$D_i = 1$			
c_2	b_2	b_1	b_0
0	0	0	1
0	0	1	0
0	0	1	1
1	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0

$D_i = 2$			
c_2	b_2	b_1	b_0
0	0	1	0
0	0	1	1
1	1	0	0
1	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1

...

$D_i = 4$			
c_2	b_2	b_1	b_0
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1

result, the range of the time base is 8. Applying Eq. (6.1), it follows that the maximum relative deadline allowed by the system is 4.

Table 6.3 shows an example of this scenario. The time base is shown on the left-hand side. The possible absolute deadlines stored in the priority queue are presented in the remaining tables. For example, when the time base is 0, all values in the queue must range between 1 and 4 (represented by bits b_2, b_1, and b_0).

From the tables and based on the previous assumptions, it can be seen that during the time base interval from 0 to 3, no value in the queue can overflow. Therefore, no special mechanism for handling overflows is needed in this first half of the table.

On the other hand, the first overflows can occur from the time base position 4 onwards. It is possible to note that from this position either the MSB, b_2, is one (no overflow occurred), or zero (overflow occurred). For this reason, if the bit b_2 is inverted during the time-base positions from 4 to 7, the "chronological" relative order of the absolute deadlines is reestablished and comparisons can be realized even with overflows.

It follows that in order to handle overflows, the MSBs of the inputs of a comparator, c_n, must then be: a) unchanged in the first half of the time-base range; b) inverted in the second half of the time-base range. This logic is equivalent to: If TB_n (MSB bit for the time-base) is '0', $c_n = b_n$, else $c_n = \neg b_n$. Such a logic can be easily implemented in hardware with exclusive-or gates. Figure 6.7 shows the digital circuit for handling overflows.

It should be noted that the bit transformation shown in Figure 6.7 need only be done for the comparison operations. The values stored in the priority queues must keep their original format.

Now that the mechanism for handling overflows has been explained, it is possible to define a general method to find out the necessary bit-width for the deadline entries of any given deadline-based scheduler. This method is the following:

1. Define the time-base resolution of the system, tb_{res}.

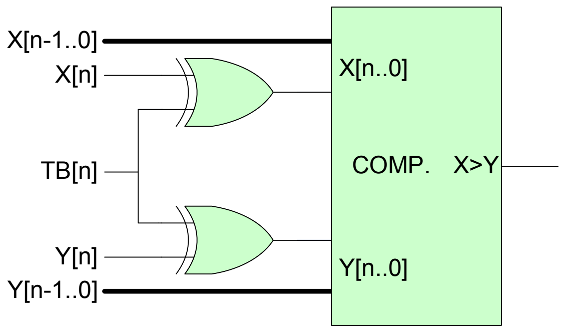


Figure 6.7: Modified comparator

2. Define the maximum relative deadline for the real-time tasks, D_i.
3. Divide the maximum relative deadline by the time base resolution. This yields as a result the number of time-base units for the maximum relative deadline, tb_{units}.
4. Calculate the amount of bits necessary to express tb_{units} using $\log_2(tb_{units})$.
5. Increment $\log_2(tb_{units})$ to satisfy equation (6.1).

Summarizing the previous steps in one formula we have:

$$bit\_width = \left\lceil \log_2 \left(\frac{\max(D_i)}{tb_{res}} \right) \right\rceil + 1 \tag{6.2}$$

For the TrailCable protocol implementation described in this chapter, the time-base resolution is 25 ns and the maximum relative deadline (D_i) is 50 s. Thus, we find that the deadline entries must be at least 32-bit long.

6.3.4.2 Balanced Priority Queues

For many applications, a single-priority queue can be enough to construct a real-time scheduler. However, given the low latency requirements for the TrailCable protocol, priority queues must be instantiated in parallel so that the scheduler be able to process simultaneous operations in a timely fashion and satisfy the response time requirement.

As demonstrated in Section 6.3.4, a priority queue operation may take up to 30 clock cycles. For this reason it is necessary to have one priority queue for each possible scheduling request within this interval. Since all other communication ports may generate scheduling requests simultaneously it follows that the number of priority queues required for a certain scheduler is equal to the number of communication and host ports in the node minus one:

$$PQs = ports - 1 \tag{6.3}$$

Additionally, all priority queues together must be able to store, in the worst case, an entry for each real-time task of a communication port. Hence, dividing the maximum number tasks of a communication port by the number of priority queues yields the minimum capacity (number of entries) required for the priority queues:

$$entries = \left\lceil \frac{max.\ tasks}{PQs} \right\rceil \tag{6.4}$$

However, a problem arises when parallel priority queues are used: if one of them is full and at least another one still has the capacity to store two or more entries, the assumption that the scheduler is able to handle simultaneous scheduling requests from all others ports does not hold anymore. Consider, as an example, a system with 1 host and 3 communication ports where the schedulers can handle up to 12 tasks simultaneously. Applying (6.3) and (6.4) we get the arrangement shown in Figure 6.8. Note that for the given example PQ_1 has only one stored entry while PQ_2 and PQ_3 have reached their maximum capacity. In this scenario, with three simultaneous scheduling requests it would not be possible to meet the requirement of response time, since all requests would have to be serialized to PQ_1. In this undesirable situation the priority queues are said to be **unbalanced**.

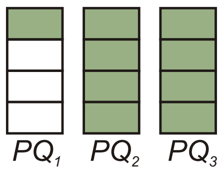


Figure 6.8: Unbalanced priority queues

In order to solve this problem, a mechanism is required to balance the number of entries in the priority queues. The priority queues are said to be **balanced** if the maximum difference in the number of entries between any two of them is one. Keeping priority queues balanced when inserting new entries is a rather simple task: it is sufficient to choose one of the queues with less entries. Contrary to this, the procedure for removing the entries from the queues requires a more meticulous method.

Figure 6.9 is an example of how to keep the priority queues balanced when entries are removed one after another. Initially at $t = 0\ \mu s$, PQ_1 stores 3 entries while PQ_2 and PQ_3 store 2 entries each. At this point, an entry is removed from PQ_1 and all priority queues will contain the same number of elements. They are therefore balanced. At $t = 15\ \mu s$, another entry is removed from PQ_1. The queues are still balanced since the maximum difference between them is only one entry. At $t = 30\ \mu s$ one more entry is removed from PQ_1. Now, if no action is taken, the priority queues will become unbalanced. To keep them balanced, the first entry from either PQ_2 or PQ_3 must be transferred to PQ_1. Finally, at $t = 45\ \mu s$ once more an entry is removed from PQ_1. To keep the priority queues balanced at this point, the first entry of PQ_3 is transferred to PQ_1.

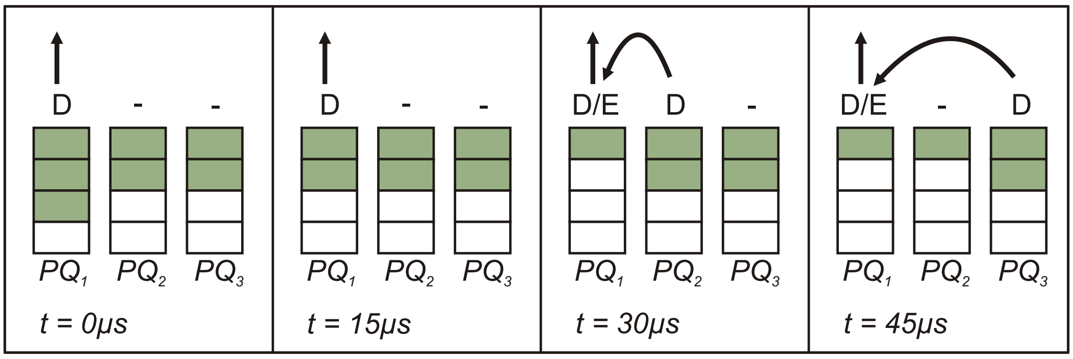


Figure 6.9: Balanced priority queues

6.3.4.3 Scheduler Hardware Architecture

Balanced priority queues are employed by the TrailCable schedulers in order to allow high scalability and low latency times in the process of sorting absolute deadlines. Priority queues are kept balanced by a block called *Balanced Queues Control* (Figure 6.10), which manages all enqueue and dequeue operations. For dequeue operations this block is able, if necessary, to automatically transfer an entry from one priority queue to another to keep the queues balanced. For enqueue operations the control block determines the priority queue in which the new entry must be inserted.

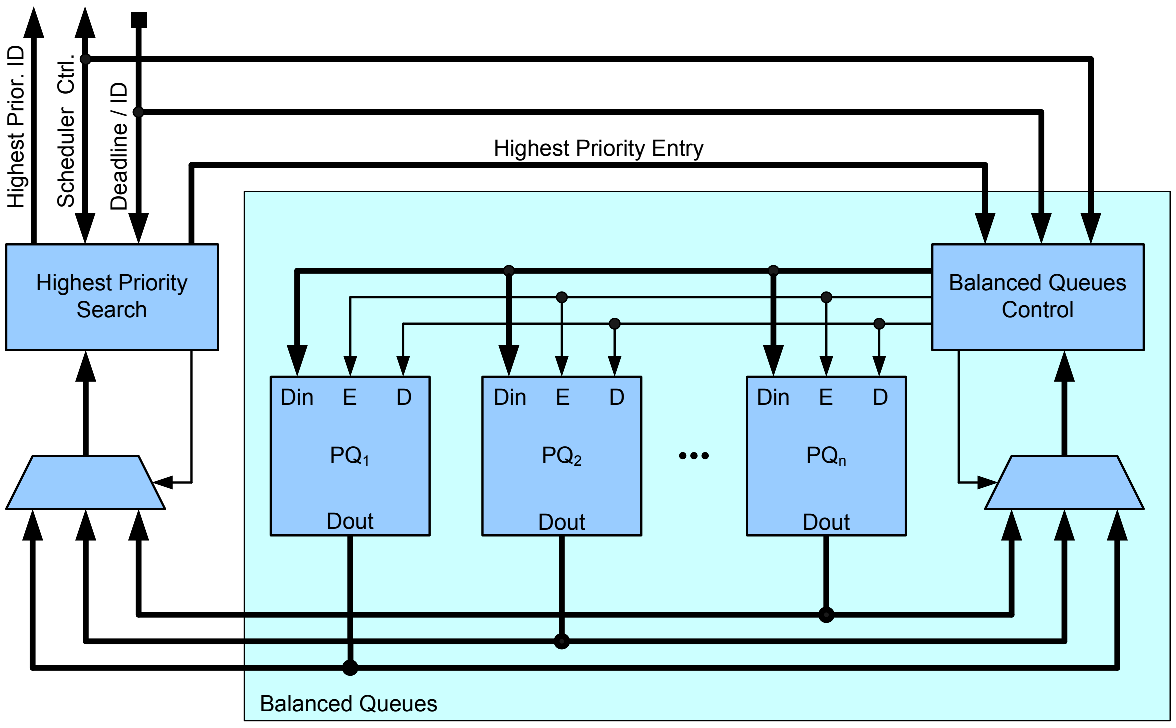


Figure 6.10: Scheduler hardware architecture

Another component present in the scheduler hardware architecture is a highest-priority search block that keeps the highest-priority entry of the scheduler. So, whenever a new entry with a higher priority than the current one arrives, it will be possible to inform the dispatcher immediately. Another function of this block is realized when the transmission of a packet is completed: it searches and removes from the priority queues the new highest-priority entry of the scheduler.

In the TrailCable protocol, the entries stored in the priority queues consist of both an absolute deadline and an ID. The deadlines are used for the sorting process in the EDF scheduling, whereas the ID allows the scheduler to associate a deadline with the corresponding real-time task. In fact, the relevant information to the dispatcher is only the ID of the highest-priority task, which is an output of the scheduler.

6.3.5 Dispatcher

The dispatcher is responsible for organizing data for transmission, according to the task selected by the scheduler. To accomplish this, the dispatcher performs different functions such as creating packets, fetching memory contents, updating timestamps, and controlling the sender unit. In the following sections these functionalities will be described in detail.

6.3.5.1 Packet Construction

Packets are created by the dispatcher whenever the scheduler informs it that there is an active task awaiting transmission. The dispatcher then acknowledges the scheduler and starts the creation of the packet header, which consists of a comma (either *START* or *RESUME*), ID, and the correspondent CRC.

In order to determine whether the current transmission is starting or resuming (after a previous preemption), the dispatcher monitors all incoming scheduling requests for the respective communication port. The first transmission of a task initiated after the corresponding request will be assigned a *START* header whereas all subsequent ones will be assigned a *RESUME* header.

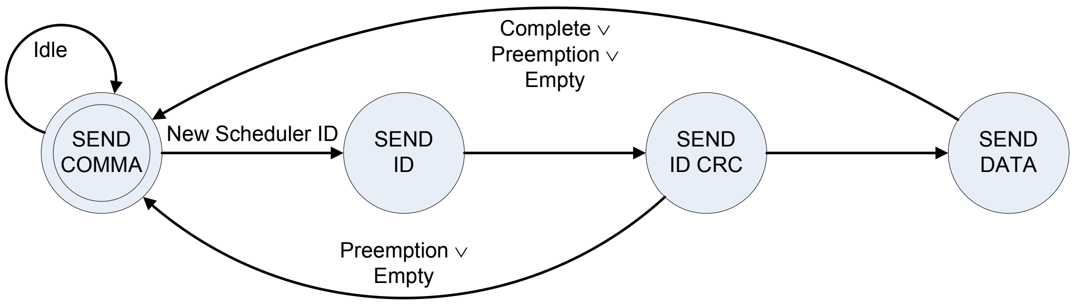


Figure 6.11: Dispatcher state machine

The state machine in Figure 6.11 shows the way the dispatcher constructs packets for transmission. At first it is in the *Idle* state. If there are no tasks available for transmission the scheduler keeps sending *START* commas. As soon as the scheduler informs an ID, the dispatcher first sends this ID, followed by its CRC. From this moment on, the dispatcher will be able to promptly preempt a task being transmitted, should the scheduler issue such a command. In this case, a new *START* comma is sent and the new packet transmission begins. When the transmission of a certain packet is completed, i.e., all bytes including the complete timestamp is sent, the dispatcher returns to its initial state. At this point it starts the transmission of the pending packets, if there are any. The transmission of a task may also stop if there are no more data to be sent, i.e., the memory buffer gets empty. This situation may occur when a packet being received by a node is preempted.

6.3.5.2 Fetching Transmission Data

During transmission of the packet header, the dispatcher begins fetching data from the *Real-Time Memory* to be transmitted. The dispatcher addresses the contents of this memory by providing both a *pointer* and an *offset* value (Figure 6.12). The *pointer* consists of both the task ID and the respective source port. The *offset* value is initialized upon the start of a packet transmission and then updated for each byte sent.

The fetching mechanism of the TrailCable protocol allows various communication ports and hence various dispatchers to access the *Real-Time Memory* contents concurrently by means of a pipelined circuit, shown in Figure 6.12. The dispatchers access the memory in a round-robin fashion. Once a dispatcher is granted access to the memory, it drives the outputs of both *pointer* and *offset* multiplexers. Then, two clock cycles thereafter the requested data content will be available, if both the *pointer* and the *offset* values are valid.

The memory map and the real-time memory in Figure 6.12 belong to the host port, the multiplexers and demultiplexers belong to the real-time interconnect and the dispatchers are located in the communication ports. The fetching pipeline has three stages. In the first one, the pointer is used to address the memory map. Simultaneously, the offset is stored in a register. In the second stage, the outputs of the memory map and the offset register are combined to create the address for the real-time memory. In the last stage, the data byte for transmission is available from the real-time memory. Also in this stage, a bit indicates to the appropriate dispatcher whether the data is valid or not.

In order to be considered valid, both the accessed memory address and the offset must be correct. The memory address for the access is considered valid if the memory map was initialized with a correct value. The offset is valid if the corresponding byte was already written into the memory by the input port.

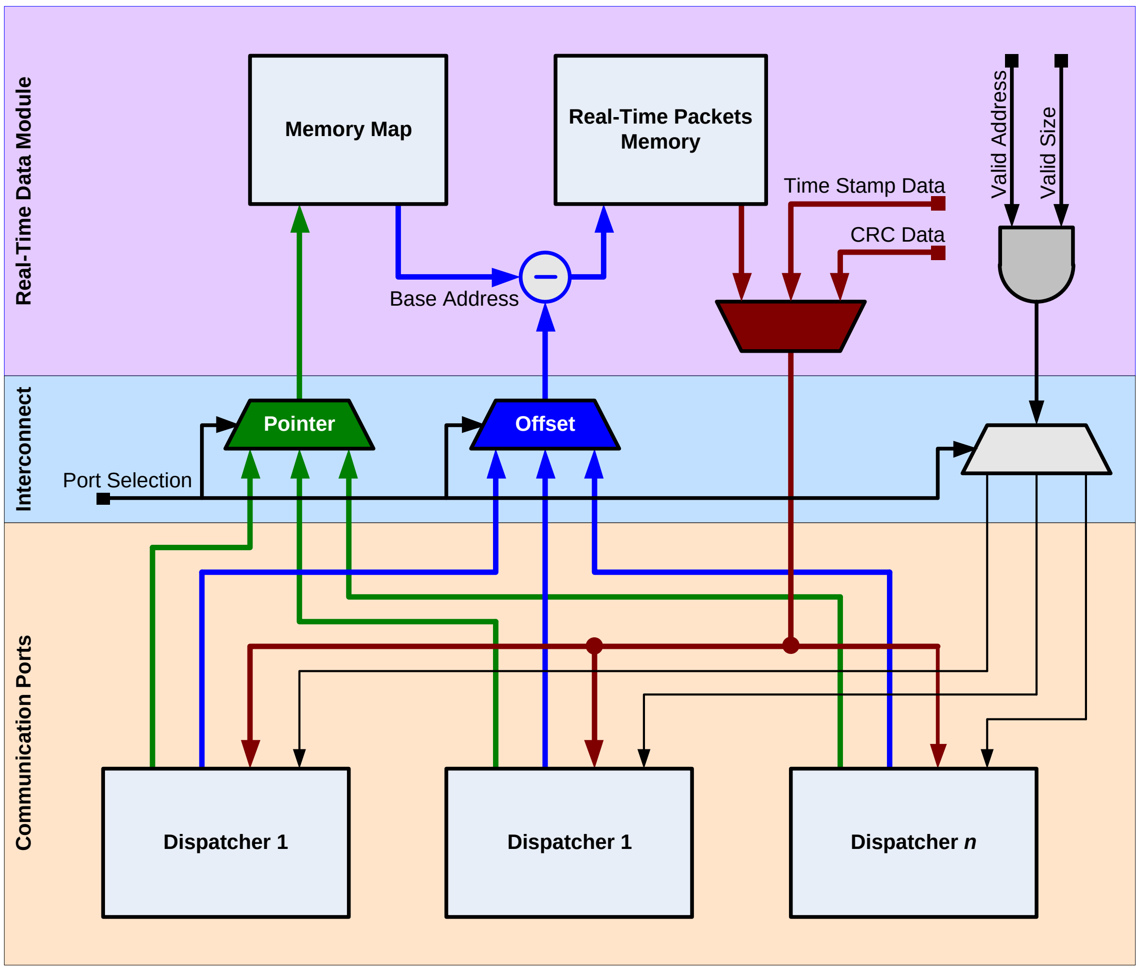


Figure 6.12: Fetching of packet data

Since the dispatcher gets the response two cycles after its request and due to the fact that the dispatchers request a new data every 10 clock cycles, the pipelined architecture presented is able to handle up to 8 communication ports.

Besides packet payload, the respective CRC and the timestamp are also retrieved by the dispatcher from the host port by the presented mechanisms. For the packets originating in the same node, the payload CRC is calculated by the Host Port. For the packets that come from other nodes, the payload CRC is not altered and is therefore fetched like normal payload data.

6.3.5.3 Updating Timestamps

In every hop, packets have their timestamps automatically updated by the TrailCable communication engine. This is done by adding both the measured propagation delay of the next link and the elapsed time since reception of the *START* header (or activation if it originates in the same node) till the exact time of transmission start.

The host port is responsible for adding the propagation delay for the next link to the timestamp, while the role of the dispatcher is to add the elapsed time in the node. To do so, the dispatcher utilizes the start time $s_{i,j}$, the transmission time $t_{i,j}$, and the current timestamp, $ts^k_{i,j}$. $s_{i,j}$ is determined by the receiver unit whereas $t_{i,j}$ is obtained by the dispatcher itself when it commands the transmission of a new *START* header. The timestamp update realized by the communication engine is represented by Eq. (6.5) that also shows the factors that are updated by the dispatcher and the Real-Time Module (Section 6.2.3).

$$ts^{k+1}_{i,j} = ts^k_{i,j} + \underbrace{(t_{i,j} - s_{i,j})}_{\substack{\text{Updated by the}\\ \text{Dispatcher}}} + \underbrace{prop\ _{k,k+1}}_{\substack{\text{Updated by the}\\ \text{Real-Time Module}}} \tag{6.5}$$

Timestamps have a resolution of 25 ns, which is the same resolution of the measurements of both $s_{i,j}$ and $t_{i,j}$. In order store the maximum latencies in a TrailCable network, 4 bytes are needed to represent the timestamp. An additional timestamp CRC is also appended to the end of the packet. The need for an extra CRC is justified by the fact that while the payload CRC must not be changed in intermediate communication hops, timestamps need to be updated frequently. A CRC for the timestamp is required in order to detect any communication faults that could impair the clock synchronization or the retrieval of the measured transmission latency of packets.

6.3.6 Sender

The sender unit receives commands and data from the dispatcher and generates the transmission bit-stream for the physical layer. The dispatcher indicates whether a data or comma transmission should take place. In the latter case, the comma type (*START* or *RESUME*) is also informed by the dispatcher. Albeit the commands come from the dispatcher, the timing of these operations are provided by the sender unit. The reason is that only the sender unit is able to determine the exact instant when each new byte for transmission must be available.

The hardware architecture of the sender unit is basically reciprocal to the receiver unit. Data provided by the dispatcher go initially to a 8B/10B decoder. This component then generates a 10-bit code that is passed on to a serializer. The output of the serializer is the transmission bit-stream for the physical layer. Therefore, it takes 10 system clock cycles for each data or comma control word to be transmitted. So, this limits the allowed latency of the dispatcher, which must be able to prepare a new byte for transmission within this interval.

Even when there are no packets for transmission, the sender unit continuously generates *START* commas, indicating that the link is idle. Besides being necessary for optical communication, this behavior also permits the receiver unit to keep itself synchronized with the sender and to check whether a connection is established.

6.4 Design Space Exploration

One of the aspects that must be taken into account when designing an embedded communication protocol is the amount of hardware resources required. Embedded systems in general are characterized by their relative small silicon area and memory footprints. The communication protocol employed should therefore be no exception. This section presents results on the resource utilization for FPGA-based TrailCable protocol implementations with different configurations. The architecture presented in Figure 6.1 was employed in this design space exploration, except for the fact that the non-real-time switch was not included. The total memory available for packet buffering in the Real-Time module was kept constant at 16 kB for all configurations.

Since the number of communication ports in the hardware architecture is customizable, we decided to vary their number in order to check the impact on the amount of required resources. Moreover, the number of real-time tasks a single communication port is able to handle also reflects the required FPGA area. Indeed, a TrailCable FPGA implementation can be tailored to specific application needs by specifying the number of communication ports and the maximum simultaneous real-time tasks.

In order to observe how many hardware resources are needed for an FPGA-based TrailCable protocol implementation, the number of ports was varied from 2 to 8, one of them being always a host port and the remaining ones, communication ports. Additionally, for each of these configurations the maximum number of allowed simultaneous real-time tasks was set to 8, 16, 32, 64, 128, and 256. Therefore, a total of 42 different implementations of the TrailCable communication hardware were explored. Given the number of ports and real-time tasks in the system, it is possible to dimension the priority queues accordingly by means of the equations (6.3) and (6.4). The first equation can be seen as a performance requirement while the second as a capacity requirement. If we apply the two equations to the whole exploration space we get as a result the required capacity for each priority queue, as shown in Table 6.4.

Since the priority queue modules in the TrailCable hardware store 16 entries, the actual capacity must be a multiple of this number (Table 6.5). The performance requirement represented by equation (6.3) imposes the necessity for parallel priority queues, even if capacity could be achieved by fewer queues. This means that the communication ports will be able to sort an even higher number of real-time tasks. Table 6.6 presents the actual number of real-time tasks that can be handled by each communication port. It is calculated by multiplying the values in Table 6.5 by the number of instantiated priority

Table 6.4: Required PQ Capacity

Ports	**Tasks**					
	8	**16**	**32**	**64**	**128**	**256**
2	8	16	32	64	128	256
3	4	8	16	32	64	128
4	3	6	11	22	43	86
5	2	4	8	16	32	64
6	2	4	7	13	26	52
7	2	3	6	11	22	43
8	2	3	5	10	19	37

Table 6.5: Actual PQ Capacity

Ports	**Tasks**					
	8	**16**	**32**	**64**	**128**	**256**
2	16	16	32	64	128	266
3	16	16	16	32	64	128
4	16	16	16	32	48	96
5	16	16	16	16	32	64
6	16	16	16	16	32	64
7	16	16	16	16	32	48
8	16	16	16	16	32	48

queues. It can be seen that one communication port alone in the TrailCable communication engine is able to sort up to 336 priorities simultaneously. In this scenario, all communication ports combined would be able to sort up to 2352 priorities. Nevertheless, any extra sorting capacity of the communication ports cannot be always fully exploited since all remaining hardware structures are laid out for a pre-defined number of real-time tasks, 256 being the maximum for the current implementation. As a result, the maximum number of real-time tasks the current TrailCable protocol implementation can schedule simultaneously is limited to 1792.

Table 6.6: Actual communication port capacities

Ports	**Tasks**					
	8	**16**	**32**	**64**	**128**	**256**
2	16	16	32	64	128	256
3	32	32	32	64	128	256
4	48	48	48	96	144	288
5	64	64	64	64	128	256
6	80	80	80	80	160	320
7	96	96	96	96	192	288
8	112	112	112	112	224	336

In the design space exploration performed, the VHDL code for the TrailCable communication engine was parameterized for all 42 possible combinations of the numbers of ports and real-time tasks. Each configuration was synthesized and mapped using the Xilinx ISE 12.3 tool [89] for a Xilinx Spartan-6 FPGA, model XC6SLX75T-3FGG376 [92]. The results can be seen in Figures 6.13, 6.14, and 6.15. The figures represent, respectively, the slice registers, slice LUTs, and block memory usage.

Figure 6.13 shows that the TrailCable hardware requirements on slice registers increase in a linear fashion with respect to the number of ports and tasks. Also, it can be observed that the number of ports has a much larger influence on resource utilization when compared to the number of real-time tasks. This behavior is advantageous since in complex

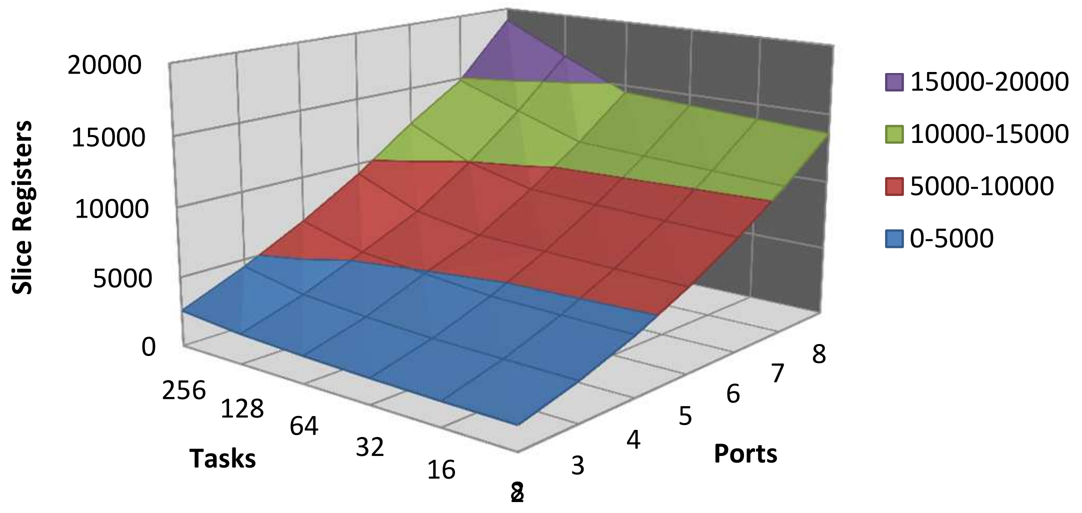


Figure 6.13: Usage of slice registers

networks the demand on real-time tasks and available bandwidth usually tends to be a more limiting factor than the number of available communication routes.

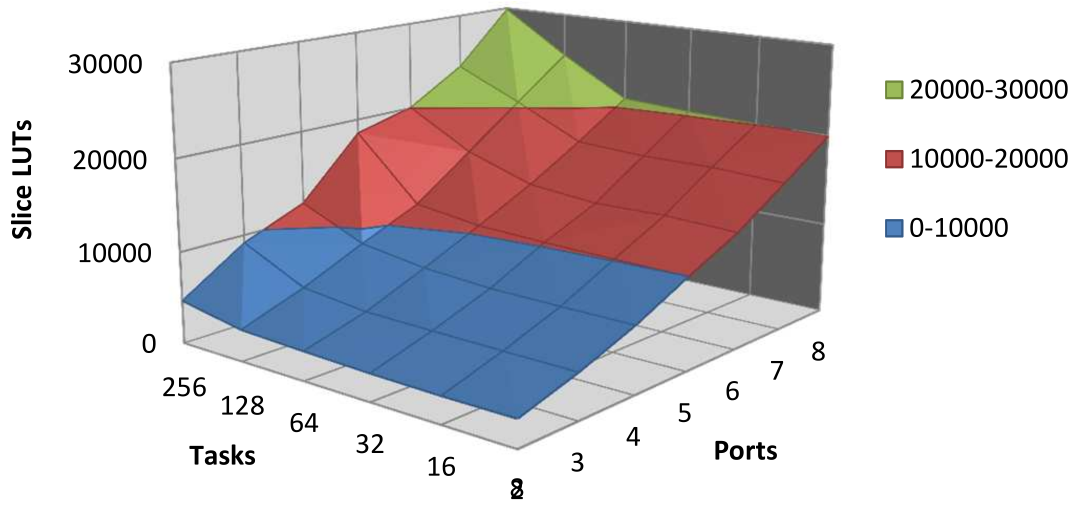


Figure 6.14: Usage of slice LUTs

Figure 6.14 indicates that the requirements on the slice LUTs are similar to the slice registers. The figure shows, however, that the number of LUTs increases slightly more for configurations with 128 and 256 real-time tasks. This can also be noticed in Figure 6.13, but to a lower extent. The reason for this effect can be explained by the architecture of the Spartan-6 FPGA family, which is based on 6-input LUTs. Therefore distributed memories with a depth of 64 and a combinational logic with up to 6-bit inputs can be efficiently constructed with these units.

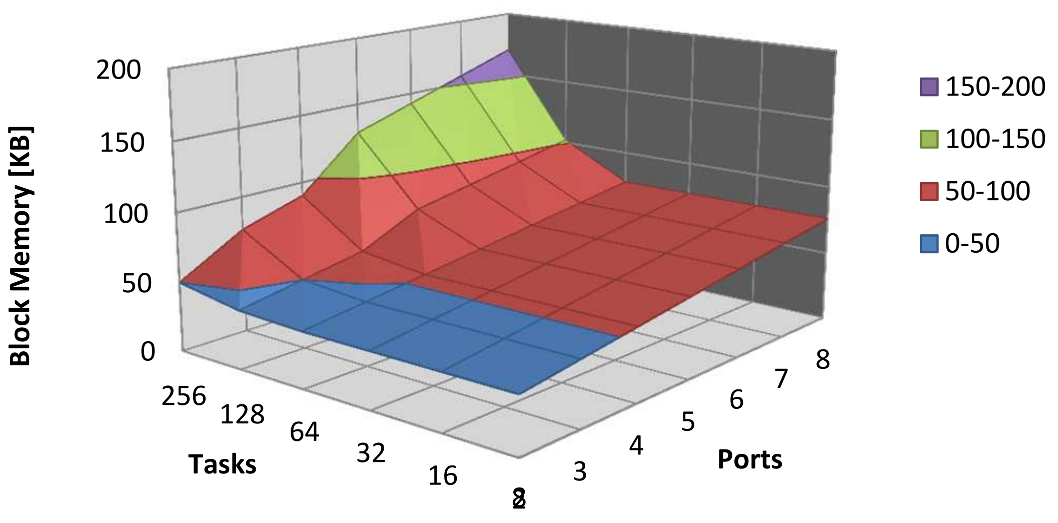


Figure 6.15: Usage of block memory

Besides normal logic slices, the TrailCable communication engine implementation also utilizes block-RAM memories. Usage of the latter resources is presented in Figure 6.15. It can be seen that for the configurations of up to 64 real-time tasks the number of utilized block memories is kept constant for a given number of ports. This is justified by the fact that the decision whether to use block or distributed RAMs for the internal data tables was taken for a 64 tasks scenario. Therefore, if the design is dimensioned for fewer tasks, the block memories will not be fully exploited. However, the design could be easily modified to use distributed instead of block memories in these situations, leading to less memory usage.

From this design space exploration it becomes clear that with an increasing number of instantiated ports, the FPGA area required for the communication ports increases as well. Figure 6.16 shows how the number of occupied slices of a single communication port grows in relation to the host port and real-time module. The results indicate that the FPGA area required by the communication ports increases linearly, whereas the remaining components consume just some additional slices as the total number of ports gets larger. This behavior also confirms the scaling capability of the TrailCable communication engine.

A comparison of the required FPGA resources of the TrailCable protocol with other commercially available communication protocols cannot always be done in a straightforward manner, since the rationale of the designs may differ in many aspects. Nevertheless, comparisons can be useful to classify communication protocols with respect to their overall resource usage. Since the TrailCable protocol is aimed at hard-real-time embedded sys-

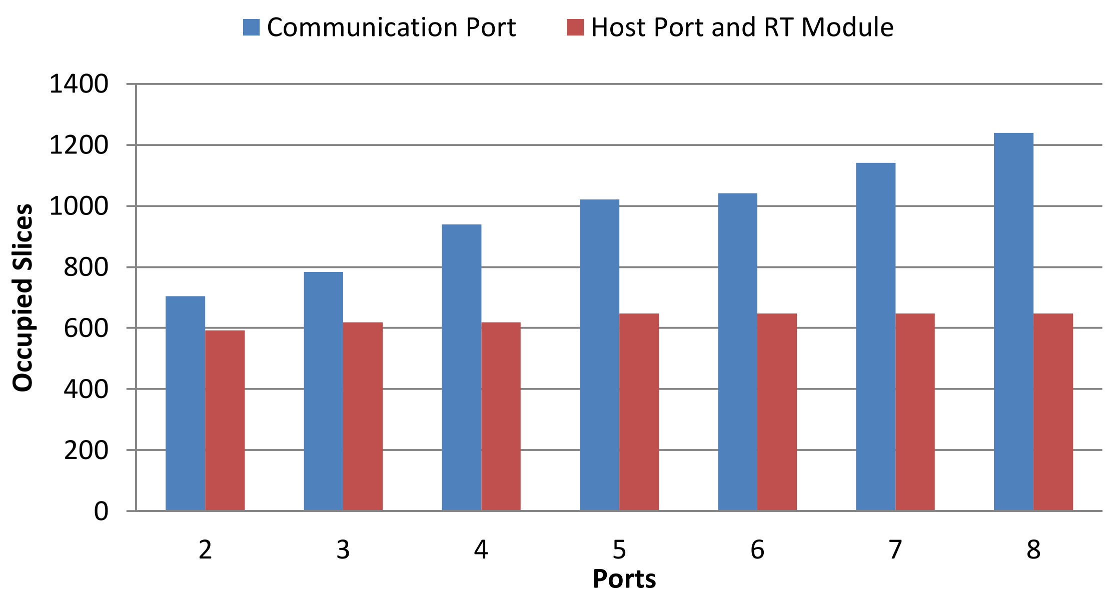


Figure 6.16: Component resources (128 tasks per communication port)

tems, it is worth checking whether its area requirements are close to that of other protocols used in this domain.

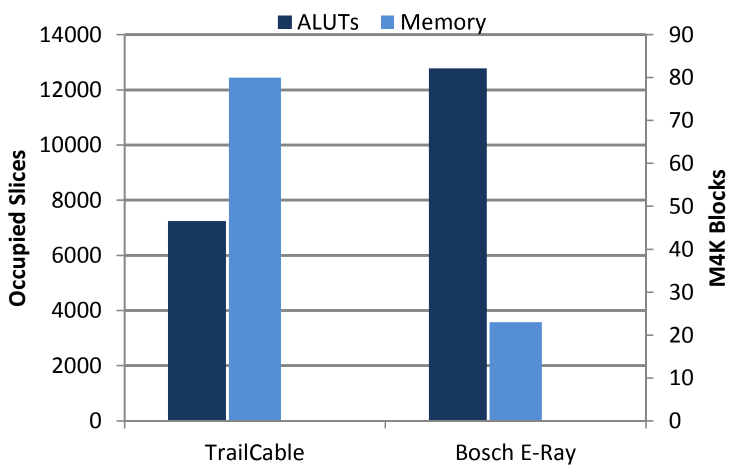


Figure 6.17: Comparison of TrailCable and Bosch E-Ray resources

Figure 6.17 shows a hardware resource comparison between TrailCable and a FlexRay IP called E-Ray [76], developed by Bosch. The TrailCable hardware was configured with two communication ports with up to 256 real-time tasks each. E-Ray has 128 message buffers and is able to handle messages up to 254 bytes long. It can be seen that the TrailCable implementation requires less logic resources but more memory. The lower requirement on ALUTs of TrailCable reflects its efficient architecture (activating non-real-time communication, however, would lead to an increase in the required area). The higher memory requirement is explained by the fact that the incoming packets in the TrailCable protocol must be internally buffered since packet transmission is postponed if higher-priority tasks are using the same output link. Moreover, although the TrailCable communication engine was originally optimized for Xilinx FPGAs, the results presented in the figure are for an Altera FPGA implementation, due to the fact that the available E-Ray data are also from this supplier.

The comparison highlights the fact that TrailCable is a practicable alternative to a communication protocol because it not only provides a variety of features that assure hard-real-time behavior but can also be implemented efficiently.

6.5 Chapter Summary

The TrailCable communication engine presented is a fully operational design and can be employed in a variety of applications. Thanks to its flexible architecture, it can be adapted for meeting specific application requirements and allowing the construction of simple, single-channel communication controllers up to multi-port configurations with hundreds of real-time tasks. However, it is still possible to ponder new features for further improving the TrailCable communication engine capabilities. Some of the possibilities are listed below:

Higher Throughput The TrailCable protocol was originally designed to provide communication services for embedded systems, with typical communication rates under 100 Mbps. Limiting communication bandwidth allows executing different instructions in serial steps, thus reducing the FPGA area requirements. On the other hand, if the data processing is carried out by parallel circuits, higher performance can be achieved and consequently higher communication rates. This design compromise and the utilization of high-speed transceivers available in current FPGAs are expected to boost the TrailCable performance and allow the utilization of data links with more than 1 Gbps bit rates.

Robustness against Transient Faults The TrailCable communication engine has different data tables and buffers distributed in its architecture. For highly dependable systems, transient faults in these memories become a concern. To cope with this problem, it is possible to use error correcting codes (ECC) in all storage elements. Moreover, application-independent solutions such as FPGA hardware synthesis with triple modular redundancy are readily available [52] and also contribute to increasing the design robustness.

Dynamic Hardware Partial Reconfiguration FPGA partial reconfiguration has already been used to increase the flexibility of networking hardware [37]. With TrailCable, dynamic reconfiguration can also potentially extend the capabilities of the protocol. By means of FPGA partial reconfiguration, it would be possible to add, remove or alter communication ports during operation. As an example, under-utilized links can have their traffic deviated to alternative ones and the FPGA area for the respective communication port can be made available to other services. In another scenario, partial reconfiguration could be employed to dynamically adapt the size of the communication ports to their current traffic demand, so that the lower the number of real-time tasks they handle, the lower also the occupied FPGA area. Figure 6.18 shows the FPGA placement of a TrailCable engine with three communication ports and the reserved area for an additional one.

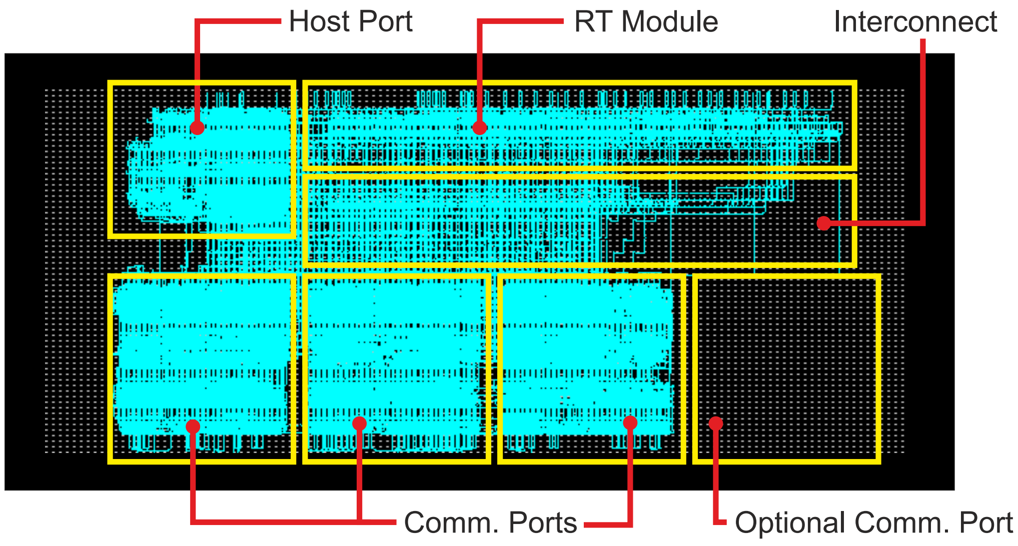


Figure 6.18: Communication engine placement on an FPGA

Chapter 7

Experimental Results

This chapter presents quality and performance measurements of the TrailCable communication engine hardware introduced in Chapter 6. The influence of different design decisions can be clearly seen in the measurements, whose parameters range from bandwidth, latency to precision characteristics. The results are grouped in three different sections. In the first one, an extensive approach is employed to identify the upper and lower bounds of latency times. In the second section, a method to evaluate jitter and reaction times is used. The last section deals with the clock synchronization mechanisms and discusses parameters which can have influence on the achievable precision.

7.1 Latency Times of Virtual Real-Time Channels

When implementing a real-time data communication protocol, a basic requirement is that the transmissions of data packets occur within defined latency bounds. The rationale of the TrailCable protocol is based on the specification of deadlines for each virtual real-time channel for analyzing the schedulability of a given configuration. The channel deadlines can be seen as the maximum allowed latency time for a given channel. In this section, the ability of the TrailCable protocol to meet deadlines is put to the test. In order to make an extensive analysis, not only are we going to check whether deadlines can be met, but also what is the exact latency time in packet transmissions.

The system setup for the experiments carried out during the latency analysis is depicted in Figure 7.1. The hardware boards of the three nodes are described in detail in Chapter 8, but at this time it is sufficient to know that each board comprises a discrete microcontroller (running the host functions) connected via its data bus to an FPGA (with the communication engine hardware implementation). In addition, an extra component was integrated into the FPGA design to allow measuring the transmission latency time with high precision. This component starts a counter when the source host triggers a data

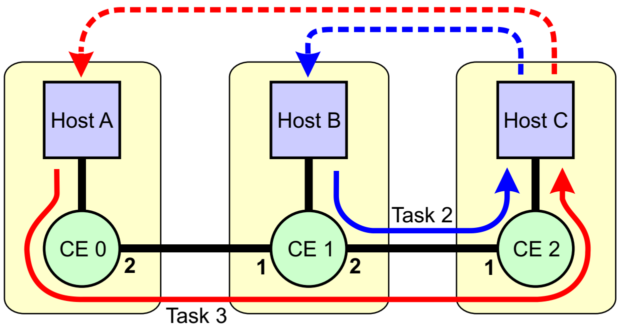


Figure 7.1: Latency measurement setup

Table 7.1: Single task configuration

Task ID	2
Period	196μs
Source Host (Relative Deadline)	B (65μs)
Destination Host	C
Payload	249 bytes
Required Channel Deadline	68μs
Actual Channel Deadline*	67.35μs
Transmission Time*	64.75μs
Minimum Achievable Latency	65.5μs

*Calculated by the TrailCable Verifier tool

transmission. Upon complete reception of the transmitted packet at the remote communication engine, a pulse is sent back to the source node via an additional wire (indicated by the dashed lines in Figure 7.1). This pulse, when detected at the source node, stops the counter, which will then contain the amount of clock cycles representing the total transmission latency time.

Three different experiments were set up to obtain the behavior of the system under different circumstances. The results of this procedure allow a better understanding of some of the TrailCable protocol characteristics. The experiments are detailed in the following.

7.1.1 Single Task Experiment

The first and simplest experiment consists in periodically transmitting a data packet from host B to host C (Task 2). Host A is not used in this experiment and is kept inactive. The objective of this experiment is to find out how much latency time jitter is caused by the communication engines. The parameters for the communication task configuration are presented in Table 7.1.

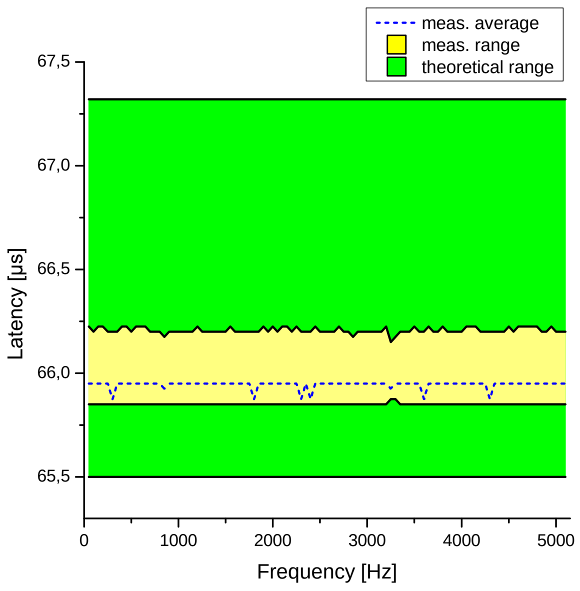


Figure 7.2: Latency measurement

In order to check the feasibility of the system, the TrailCable Verifier tool was executed with the requested task parameters. The maximum latency time of 67.35 μs was calculated by the tool for the virtual real-time channel. Besides the maximum latency, another value of interest at this point is the minimum achievable latency time. It can be found by adding link propagation delays and best-case forwarding delays (opposed to the worst-case forwarding delays for the channel deadline calculation) to the packet transmission time. In the current experiment the minimum achievable latency time would be about 65.5 μs. Thus, all latency times in this particular scenario must be in the tolerance range between 65.5 μs and 67.35 μs. The expected jitter of the current experiment is therefore about 1.85 μs.

The results of the latency time measurement from host B to host C are shown as a frequency response in Figure 7.2. The frequency represents the transmission periodicity of the data packets. For every 50 Hz frequency incrementation step, one thousand transmissions were initiated and their latency times were logged. Of special interest are the minimum and maximum latency times, which define the borders of the measured latency range. The dashed line in the figure shows also the average latency time, obtained from all one thousand transmissions carried out for each frequency.

It can be clearly seen that the measured latency times lie entirely in the expected theoretical range. The system responds for all frequencies up to 5.1 kHz, which corresponds to a transmission cycle of approximately 196 μs. This value corresponds to the period parameter used to configure the communication hardware. For frequencies higher than 5.1 kHz, packets are eventually discarded by the bandwidth guardian mechanism. The difference between the expected range and measured one results mostly from the clock deviation factor, numerical tolerances, and the forwarding delays, which were estimated with an error margin.

In regards to quantitative aspects, it turned out that the jitter (the difference between the maximum and minimum measured latency times) for the current scenario was only 375 ns. Since the clock oscillator operating frequency is 40 MHz for the implemented communication engine, the jitter represents only 15 clock cycles. For comparison, it takes 10 clock cycles to transmit a single byte or control word. This explains the main source of jitter, which is caused by the necessity to wait for the ongoing transmission of a data byte or control word to be completed before starting the packet transmission triggered by the host.

From these results, it follows that the jitter caused by the communication engine hardware accounts only for a small and possibly negligible part of the latency time deviation. The relevant portion of the latency time deviation is caused by a real-time scheduler handling multiple tasks. The latter effect is nevertheless already well known and taken into account in the schedulability analysis of the real-time communication.

7.1.2 Multiple Nodes Experiment

Once the system behavior with a single task is known, the next step is to check the system response with concurrent communication tasks. For the sake of simplicity the experiments were carried out with just two communication tasks, but the system behavior with a larger number of tasks can be derived from this simple case. One of the two communication tasks is similar to the single task experiment (task 2), but with different deadlines. The additional communication task is originated at host A and has host C as a destination (task 3). The link between communication engines 1 and 2 (Figure 7.1) is shared by both communication tasks and some real-time packet scheduling characteristics can therefore be analyzed. The parameters of the current experiment are shown in table 7.2.

Task 3 has smaller relative and channel deadlines and is transmitted more frequently than task 2. The outcome of this configuration is that in order to meet its deadline, task 3 requires task 2, if active, to be preempted. This is the reason why the deadline of task 2 had to be increased in order to maintain the feasibility of the real-time communication. The focus in this experiment continues to be the latency time measurement of task 2, but now with the influence of the additional task 3, whose transmission period was kept constant at 94 μs. The results can be seen in Figure 7.3.

As expected, the maximum latency time deviation increases. The measured difference between the maximum and minimum latency values of 28.625 μs, reflects exactly the time needed to transmit one packet of task 3 (27.5 μs), plus the time of a resume header

Table 7.2: Multiple-task configuration

Task ID	2	3
Period	196 μs	94 μs
Source Host (Relative Deadline)	B (94 μs)	A (28 μs)
Intermediate Host (Relative Deadline)	-	B (28 μs)
Destination Host	C	C
Payload	249 bytes	100 bytes
Required Channel Deadline	97 μs	34 μs
Actual Channel Deadline*	96.0 3μs	33.66 μs
Worst Case Transmission Time*	65.5 μs	27.5 μs
Minimum Achievable Latency	65.5 μs	29 μs

*Calculated by the TrailCable Verifier tool

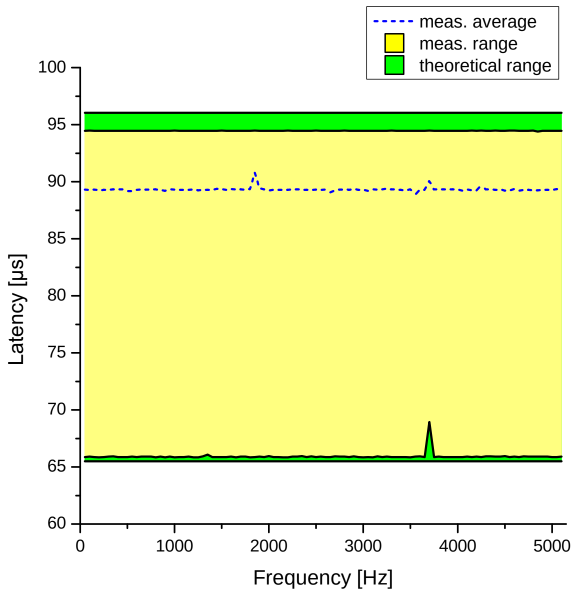


Figure 7.3: Effect of a higher priority task

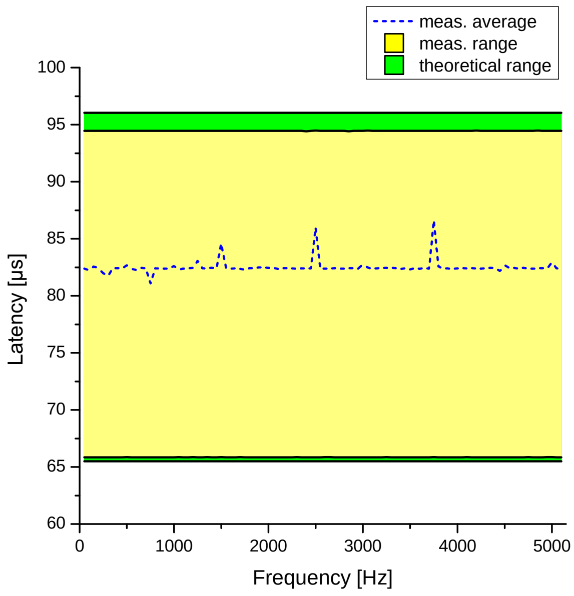


Figure 7.4: Effect of a disturbance task

of task 2 (750 ns) and the jitter, which could be determined in the single task experiment (375 ns).

Up to now, it has been assumed that the real-time communication was operated with the parameters that were validated by the TrailCable Verifier tool. Deadlines therefore could be met, because all communication tasks were configured according to the specification. An important issue, however, arises when faults occur and the actual operation differs from a valid configuration, potentially leading to deadline missing, among other anomalies. One mechanism to avoid these kind of problems is the bandwidth guardian. The second part of the current experiment is intended to check the efficacy of this mechanism. In order to inject a deliberate fault into the system, the configuration of the communication engine 0 and host A (Figure 7.1) was modified by reducing the transmission period of task 3 to 66.67 μs. Without the guardian mechanism, the alteration mentioned would lead task 2 to preempt twice and as a consequence, its deadline would be missed. The configuration of the 2 remaining nodes was left intact, so that this part of network is able to operate normally. The measurements results under these conditions are shown in Figure 7.4. The results show that the task 2 latency range is the same to the normal operation mode (without disturbances). However, a discrepancy can be identified. The average latency time decreases in comparison to the normal case. The explanation is that the bandwidth guardian of the communication engine 1 drops packets of task 3 that exceed the correct specification. In fact, every second packet had to be dropped. Therefore, the traffic of task 3 is reduced substantially and the average latency time of task 2 decreases as well. The result of this experiment proves the benefit of the bandwidth guardian mechanism, which is an important component when it comes to increasing the overall system fault-tolerance and robustness.

The nodes in all experiments were not synchronized with each other so that transmissions of tasks 2 and 3 can be triggered at any time. As expected, this does not impair the real-time behavior of the communication system.

7.1.3 Preemption Experiment

The objective of this experiment is to analyze the effect of the preemption mechanism. The network is initially configured like the multiple task experiment, i.e., two tasks (tasks 2 and 3) generated at hosts B and A, respectively, are transmitted to host C. The basic difference is that the relevant information in this particular case is the latency time of task 3. The transmission period of task 2 was now kept constant and that of task 3 varied. Under the original operation conditions, with the configuration of Table 7.2, the latency time measurement results are shown in Figure 7.5. Also in this case the latency times are within the expected range. The cut frequency is now about 10.65 kHz and corresponds, with a small tolerance, to the transmission period of 94 μs.

According to the measurements, the latency time of task 3 varies from to 29.95 μs to 31.15 μs. Of the total of this 1.2 μs difference, 750 ns are due to the start header of task 2, which can potentially be sent by the communication engine 1 just before a scheduling request for task 3 arrives. If this is the case, the header leads to an extra delay as its transmission is atomic, i.e., it cannot be preempted. The remaining jitter of about 450 ns is caused by the hardware implementation of all three nodes and is therefore slightly higher than the jitter of the task 2 latency measurements, which involves only two network nodes.

The results so far have shown that with preemption, the two latency times of tasks 2 and 3 were kept within the expected ranges. The next step of this experiment is to check the

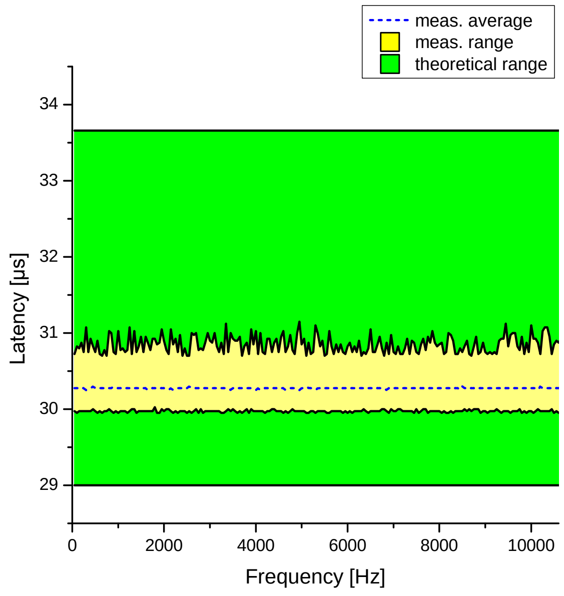


Figure 7.5: Preemption enabled

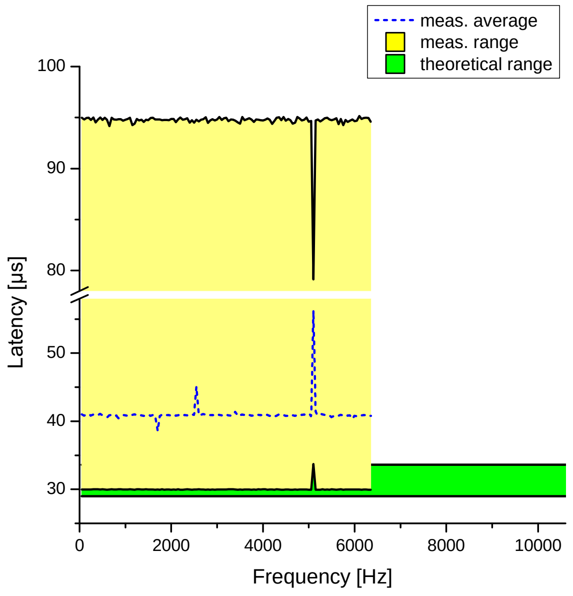


Figure 7.6: Preemption inhibited

consequences of inhibiting, in the communication engine 1, the preemption mechanism that has been used so far. The measurements taken under such conditions are presented in Figure 7.6. The natural consequence is that deadlines cannot be met anymore. Without preemption, task 3 may have to wait for an entire packet of task 2 to be transmitted before starting. The maximum latency time of task 3 which was 31.15 μs increases therefore to 95.15 μs. Moreover, the cut frequency of about 10.65 kHz is reduced to only 6.35 kHz due to the fact that higher latency times also decrease the maximum achievable transmission period.

This experiment confirms the benefit of using preemption for data communication networks. Without preemption, the deadlines that were assigned to the communication tasks cannot be met by any scheduling algorithm.

7.2 Distinctness of Reaction

The method called *Distinctness of Reaction* was proposed by Wolter, Albert, and Gerth [88, 13]. The main objective is to evaluate a computing system, taking into account reaction time and latency characteristics. The approach consists in generating external asynchronous stimuli and measuring the behavior of the system under test. The Distinctness of Reaction method is particularly suited to compare properties of communication systems (the focus of the current work), operating systems, microprocessors, among others.

The measurements are supported by an additional hardware unit (introduced in [88]), which was implemented as an FPGA component. This component substitutes the latency

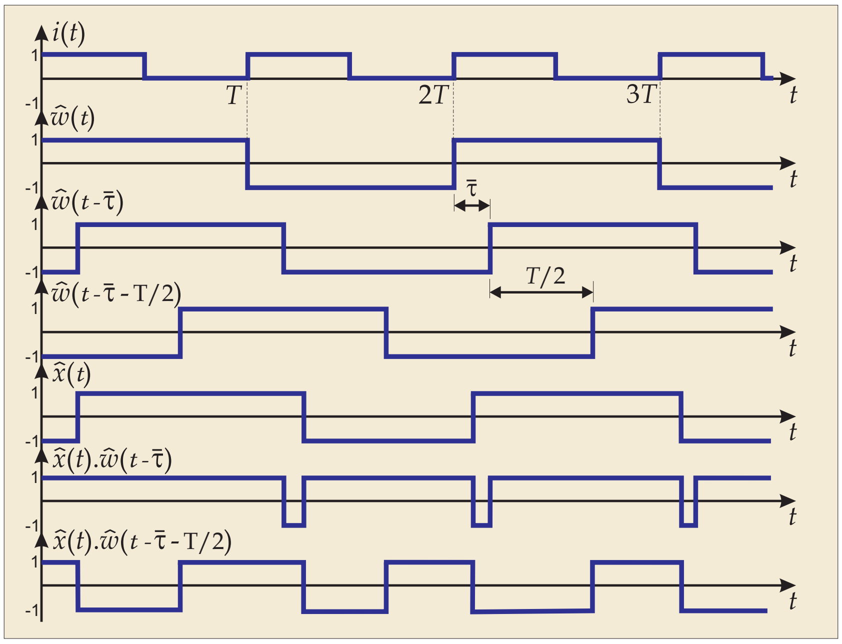


Figure 7.7: DoR signals

time measurement unit used in the previous experiments. The Distinctness of Reaction (DoR) measurement is based on the excitation signal $i(t)$, with period T and duty cycle of 50 % (Figure 7.7). The task of the system under test is to toggle an output signal, $x(t)$, each time a rising edge of the signal $i(t)$ is detected. The signal $x(t)$ is then compared to the signal $w(t-\overline{\tau})$, which is the ideal response $w(t)$ when toggling at each rising edge of $i(t)$, shifted by a delay constant $\overline{\tau}$. The delay constant $\overline{\tau}$ is the average of the time a system takes to process and toggle the output signal $x(t)$. Under ideal conditions $x(t)$ would be equal to $w(t-\overline{\tau})$, but in practical applications the reaction times are likely to be disseminated within a certain range. The larger the disparity between $x(t)$ and $w(t-\overline{\tau})$, the larger therefore the jitter caused by the system under test.

The functions $x(t)$ and $w(t)$ are digital signals and can assume the values 0 or 1. It is helpful, however, to transform these functions for further developments so that they have the values -1 or 1, as follows:

$$\hat{x} = \begin{cases} 1 : x(t) = 1 \\ -1 : x(t) = 0 \end{cases} \quad ; \quad \hat{w} = \begin{cases} 1 : w(t) = 1 \\ -1 : w(t) = 0 \end{cases}$$

When the product $\hat{x}(t)\hat{w}(t-\overline{\tau})$ is 1, both signals are equal, otherwise the result is -1. It follows that with the integral

$$\text{DoR} = \lim_{n\to\infty} \frac{1}{nT} \int_0^{nT} \hat{x}(t)\hat{w}(t-\overline{\tau})dt \tag{7.1}$$

the average jitter over a sufficient large number of cycles can be determined. Nevertheless, the Eq. 7.1 is not solvable because $\overline{\tau}$ is still unknown. Finding the average latency $\overline{\tau}$ is also an objective when applying the DoR approach. In order to cope with this problem, an orthogonal correlation method is employed. The latency $\overline{\tau}$ is initially estimated to be t_D. The function $\hat{x}(t)$ is then not only correlated with $\hat{w}(t-t_D)$ but also with $\hat{w}(t-t_D-T/2)$. These correlations can be interpreted as the real and imaginary parts of a frequency response diagram and are defined below:

$$\mathrm{Re} = \lim_{n\to\infty} \frac{1}{nT} \int_0^{nT} \hat{x}(t)\hat{w}_R(t-t_D)dt \tag{7.2}$$

$$\mathrm{Im} = \lim_{n\to\infty} \frac{1}{nT} \int_0^{nT} \hat{x}(t)\hat{w}_I(t-t_D-T/2)dt \tag{7.3}$$

The value of DoR is the real part Re when t_D is equal to $\overline{\tau}$. To find the wanted t_D, this variable will be adjusted until the imaginary component value Im reaches its minimum. Due to the orthogonal correlation, when Im is minimum Re reaches its maximum, which is the DoR value that is being looked for. At this point, the parameters DoR and $\overline{\tau}$ are available and the measurement for a determined frequency is concluded. In order to plot a frequency response that resembles the classical style with amplitude and phase along a frequency axis, the measurement procedure will be described repeated continuously from a low frequency up to the frequency where the system fails to respond. The phase (or skew, from now on) is given as a value from 0 to -100 % and is calculated with the equation below:

$$\mathrm{Skew} = -\frac{\overline{\tau}}{T} \tag{7.4}$$

That means that the maximum skew of 100 % corresponds to the point where the system reacts after an interval equivalent to the nominal period of the input signal. If the reaction time gets any longer the system will not be able to react after each rising edge of $i(t)$ and the value of DoR will therefore be zero. The Distinctness of Reaction method was briefly described only to support the upcoming experiments with the TrailCable protocol. For more detailed information about DoR, refer to the original work ([88, 13]).

7.2.1 Single Node Experiment

With a basic background on the DoR method it is possible to describe the first experiment. The objective is to evaluate the performance limits of the TrailCable protocol when operating with tasks transmitting two application bytes. The setup is shown in

Figure 7.8. An interrupt request to the host microcontroller is generated at each rising edge of the signal $i(t)$, which is generated by the DoR measurement unit located in the FPGA. The interrupt service routine triggers the transmission of a packet (task 2 - sent via the output communication port 2). Via an external loopback cable, task 2 is forwarded to the input port 1. A polling loop detects the reception of task 2 and acknowledges it with the transmission of task 3, which is transmitted in the opposite direction. Finally, upon the reception of task 3, the microcontroller toggles the output signal $x(t)$.

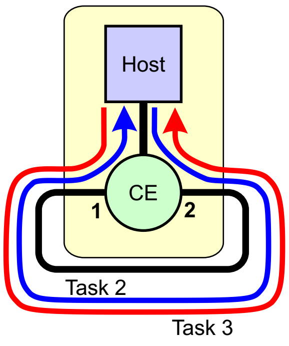


Figure 7.8: DoR measurement setup with one node

The frequency response of the given system was identified by means of an automated process. For each frequency, the DoR and Skew values were captured taking a time span of one thousand cycles of $i(t)$. The frequency response plots are depicted in Figure 7.9 and 7.10.

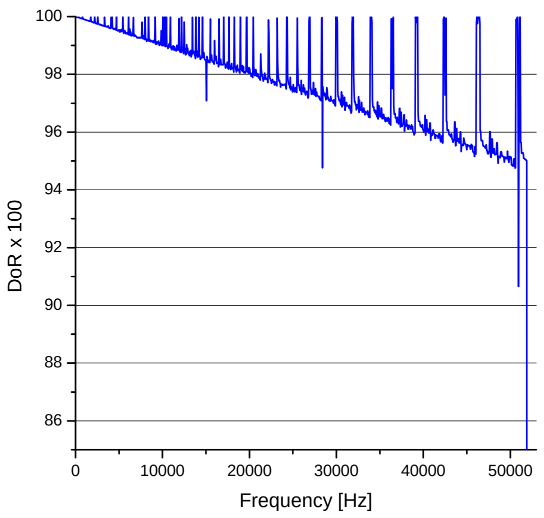


Figure 7.9: DoR - single node

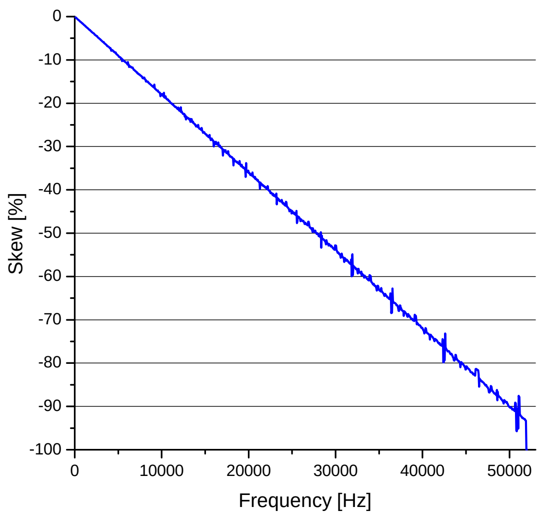


Figure 7.10: Skew - single node

From the experiment results, it can be observed that the system responds up to a frequency of 51.9 kHz. The maximum achievable excitation frequency depends both on the microcontroller and the TrailCable protocol performance. The skew drops linearly with the frequency, whereas the DoR plotting shows multiple peaks close to 1. Such peaks could be reproduced at the same frequencies in different measurement procedures. The reason is that the source of such peaks are resonance appearances, which occur when the timing of the host software functions gets synchronized with the data communication. The reproducibility stems also from the fact that the clock oscillator used by the microcontroller is the same that drives the DoR measurement unit. A second clock oscillator is used for the TrailCable protocol.

Although the main objective of this experiment was to evaluate the TrailCable protocol, the microcontroller characteristics affect the results significantly, especially at higher frequencies. For comparison, the transmission latency time of the two tasks combined is below 10 μs. This value can be even exceeded by the processing time of the host software functions involved in the process, which can also account for more jitter than the communication layer itself. The results, however, allow a good estimation of the capabilities of the TrailCable protocol. In [13] a similar procedure was executed to find the frequency response of the CAN bus. The described system used exactly the same microcontroller that was employed for the current experiments, so that only the software architectures differ one another to some extent. The frequency response of the CAN bus interface showed a cut frequency of 1291 Hz, more than 40 times lower than 51.9 kHz, the result achieved with the TrailCable protocol. This can be explained by the higher transfer rate and lower overhead of the latter.

7.2.2 Multiple Nodes Experiment

The current experiment is similar to the latency measurement setup for multiple nodes of Section 7.1.2. The basic difference is that when tasks 2 or 3 arrive at host C, the latter commands the transmission of another packet that will be sent back to the respective origin. The new setup is presented in Figure 7.11. The configuration of tasks 2 and 3 in both directions follow the parameters of Table 7.2, with the origin and destination hosts swapped for the acknowledgment tasks. The frequency response obtained with the DoR method is used to evaluate the behavior of task 2 and is based on the following process: the DoR measurement unit connected with host B generates the excitation signal $i(t)$. At each rising edge of this signal, an interrupt is invoked and leads host B to send a packet of task 2. When the response packet created by host C arrives, the output signal $x(t)$ is toggled.

The frequency response of task 2 was taken in three different scenarios. In the first one (single task), task 3 was completely disabled so that no packets were transmitted at all. For the second run (two tasks at limit), both tasks 2 and 3 were active and the packets of task 3 were transmitted at the minimum possible transmission period (94 μs). The last measurement (two tasks with disturbance) was executed by further decreasing the

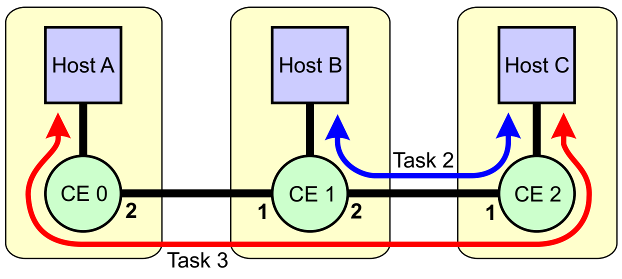


Figure 7.11: DoR measurement setup with three nodes

transmission period of task 3 to an invalid value. Figures 7.12 and 7.13 depict the frequency response for all three scenarios.

It can be seen in the figure that the theoretically valid ranges of the DoR and skew values are also indicated. To find these regions two parameters are sufficient: the best and worst-case latency times for task 2 to reach host C and return. The best-case latency is therefore twice the transmission time of task 2 plus the link propagation delay. The worst-case latency is twice the channel deadline of task 2. To increase the precision of the two parameters, one can consider the processing time needed by host C to trigger the acknowledgment task. For the current example, the best (bc_{lat}) and worst-case (wc_{lat}) latency times are 132 μs and 203 μs, respectively. With both parameters known, the valid range is determined for each frequency f as follows:

$$1 - f(wc_{lat} - bc_{lat}) \leq \text{DoR} \leq 1 \tag{7.5}$$

$$-100.f.bc_{lat} \leq \text{Skew} \leq -100.f.wc_{lat} \tag{7.6}$$

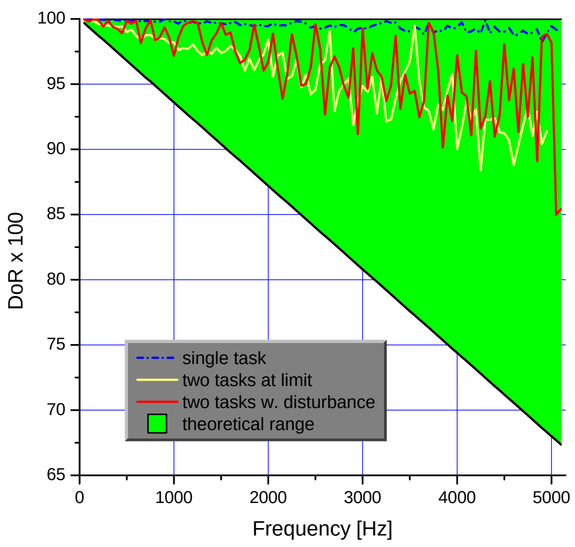


Figure 7.12: DoR - multiple nodes

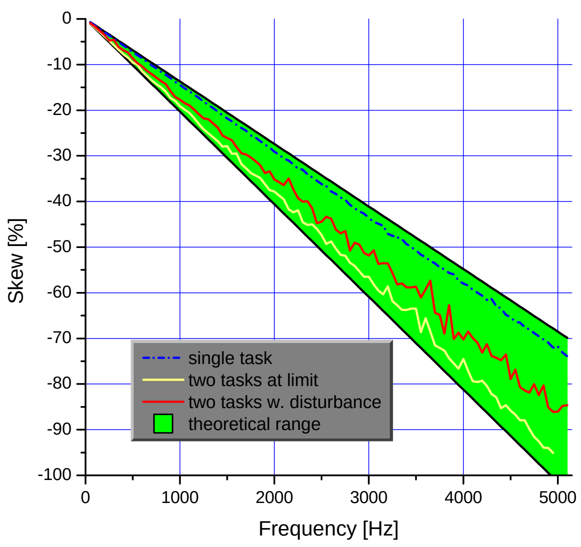


Figure 7.13: Skew - multiple nodes

At first sight it can be noted that all frequency response plots are within the valid region. Yet some peculiarities can be identified. Firstly, the DoR value for the single task scenario is more constant and higher than the others. The skew is lower and rather constant for the entire frequency range too. The reason is that task 2 is not preempted due to the inexistence of task 3, so that the latency times are kept within a small tolerance. Furthermore, with task 3 active, the DoR value suffers more variation because of the increased latency time deviation time caused by the preemption. Also, an interesting effect can be observed in the skew results, as the three plots can clearly show what happens with the latency time in each scenario. As opposed to the single task scenario, the two task scenario with channel deadlines set to the limit of feasibility leads to a higher skew. But, although in the third scenario the activation rate of task 3 is even faster, the skew does not get any higher, it decreases instead. This is once again caused by the bandwidth guardian mechanism, which has to discard some out-of-specification packets of task 3 in order to maintain the correctness of the timing characteristics of task 2. Without the bandwidth guardian, the skew of task 2 was most likely to overshoot the valid region.

[13] presents a frequency response for a time-triggered communication protocol, the TTCAN. Both DoR and skew values decrease almost linearly with the frequency. The reason is that the data transmission can only be triggered at pre-defined time slots. At a cycle time of 1 ms, for example, the rising edge of $i(t)$ can occur at any moment of this cycle. As a result, the reaction time of the sender is 500 $\mu s \pm 500 \mu s$. The outcome is that although predictable, the improvement of the reaction times in time-triggered protocols is limited by the global cycle time. One advantage of the TrailCable protocol paradigm is that, especially for networks working with a low or medium capacity, considerably better reaction times can be reached. This can be partially explained by the fact that packet transmissions can be triggered at any time, as long as the minimum time interval between two instances is respected. Such characteristic is also reflected by the measurements of this section. For a single task, both DoR and Skew values were better than with a concurrent task. Even in the case that multiple tasks were running simultaneously, there was a good margin between the measured DoR and the expected worst-case DoR values.

Although the DoR method represents a good way of comparing different real-time systems and analyzing their behavior, it is important to bear in mind that this approach should not be used as the only means for measuring worst-case jitter or latency. The reason for this is that the method integrates the DoR variable along many cycles, leading to extraordinary occurrences being filtered out. Furthermore, the latency time is obtained indirectly from the measurement with the lowest jitter. On the other hand, the DoR method is an intuitive manner for control engineers to analyze the behavior of the communication protocol. With the DoR frequency responses, the task of analyzing and simulating a distributed system with respect to control characteristics can be facilitated to some extent. Finally, another

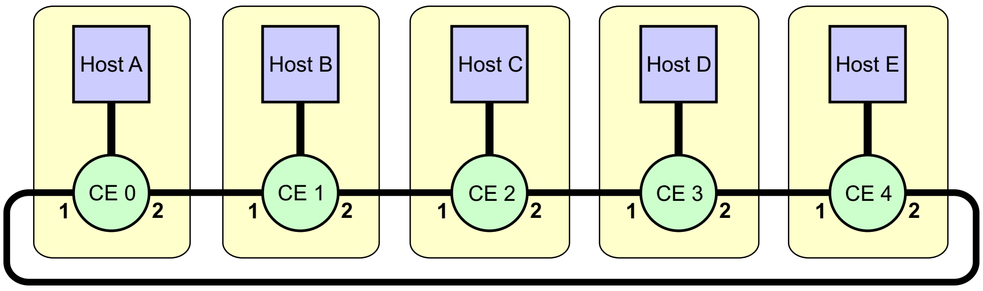


Figure 7.14: Network for the clock synchronization precision measurement

advantage of the DoR method is that the frequency responses can be used to support the specification of necessary channel deadlines. With a given frequency response including the required DoR and Skews values, one can easily define, by means of equations 7.5 and 7.6, the best and worst-case latency times and then specify a suitable channel deadline.

7.3 Clock Synchronization

Although clock synchronization is not required for real-time communication with the TrailCable protocol, different applications can profit from this extra service. In this case the quality of the clock synchronization may become an important technical aspect to be taken into account. In this section, measurements regarding the clock synchronization are presented. The focus is on the maximum clock deviation, i.e., the maximum time difference between two synchronized events.

7.3.1 Measurement Setup

Before proceeding with the interpretation of the measurement results it is necessary to introduce the network employed. Due to the capacity of the FPGA used in the hardware system, which allows only two communication ports and one host port, the chosen topology is a ring with 5 nodes (Figure 7.14) with optical data links. This limitation, however, does not impair the quality of the measured results since the process of synchronizing clocks relies basically on timestamps, which work according to the same principle, regardless of the topology.

With respect to the virtual real-time channels, redundancy was used. At the beginning of each communication cycle, all hosts transmit one synchronization packet through the two communication ports. These packets are successively forwarded by the neighbors until the information reaches all participating nodes. For example, host C sends its synchronization packet clockwise via its communication engine port 2, which is forwarded by the remaining boards until the packet reaches host B. The same applies for the anti-clockwise transmission which begins in host C and ends in host D.

In this configuration, each host receives up to eight synchronization packets (two for each other node in the network). However, only the first arriving data packet from a certain source node is considered in the synchronization process.

7.3.2 Interpretation of the Results

When the presented network is synchronized, all nodes transmit data packets periodically and approximately at the same time. It was possible to achieve synchronization cycles as low as 100 μs with the given architecture. Although the synchronization cycles depends mainly on the amount of data sent and on the deadlines of the virtual channels, such a small cycle reaffirms the performance of the TrailCable protocol thanks to its dedicated hardware structure.

However, not only the shortest possible cycle is an important aspect, but also how precise the synchronization is. In order to measure such characteristic the maximum deviation parameter was employed. The maximum time deviation is obtained in a synchronized system by measuring, from the point of view of one of the hosts, the maximum time difference between the local event and the corresponding synchronized events that are generated in the other hosts. The deviation can be positive or negative, but in this study only the absolute values are considered.

Three different setups were used to measure the maximum deviations. They are detailed as follows:

- Normal Scenario - the system is configured in the standard manner, which means that when traversing a communication engine timestamps are updated by means of accumulating the time interval needed from the beginning of reception until the beginning of the transmission to the next node. Moreover, the pre-calculated link delay time of the output port in use is also added to the timestamps.

- Artificial Delay - in this configuration the setup is similar to the normal scenario. The basic difference is that an additional propagation delay was artificially added to each of the data links in order to simulate longer cables. With this extra propagation delay, packets take 1.25 μs longer to travel through each link.

- Inhibited Delay Retrieval - derived from the Artificial Delay scenario, in this particular case the timestamps are only updated with the time spent passing through the communication engine. The link delay values are not added to the timestamps in contrast to the two previous scenarios.

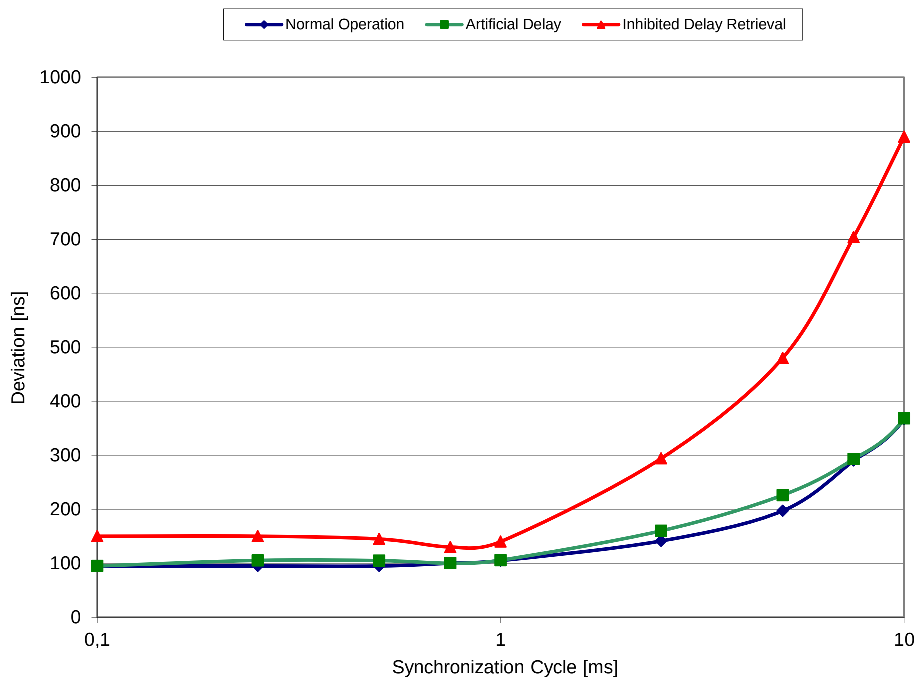


Figure 7.15: Maximum clock synchronization deviation

Measurements were made for communication cycles in the range of 100 μs to 10 ms for all three scenarios. The results are presented in Figure 7.15. The deviation with communication cycles of up to 1 ms are kept approximately constant. From this point on, they increase rather linearly (considering a non-logarithmic scale). The reason for this is that during short cycles the drifting of the clock oscillators is relative small and does not impact significantly the synchronization. From the 1 ms cycles onwards, the effect of the clocks drifting becomes apparent and, as expected, the longer the communication cycles, the greater the corrections needed to keep the system synchronized, which in turn reflects on greater deviations.

An interesting conclusion that can be made from the presented results is that for achieving clock synchronization in the presented scenarios, timestamps are indispensable. The plots of the Normal and the Artificial Delay scenarios are very close to each other, despite the extra propagation delay. This is possible only because the actual propagation delays on the cables are accurately measured and therefore timestamps are very precise. If the propagation delay retrieval process is inactive and the correspondent information is not available, the maximum deviation will increase rapidly beyond a certain point. This effect can be observed by the plotting of the Inhibited Delay Retrieval scenario. It is important to emphasize that in the Inhibited Delay Retrieval scenario, only the link propagation delay part was not added to time stamps. This portion accounts only for a minor part of the entire time stamp.

Even for synchronization cycles of 10 ms, the maximum measured deviation of about only 380 ns is still quantitatively very good for most practical applications in the embedded system domain, especially mechatronic systems. Therefore no measures were taken to further minimize the maximum deviation. A simple technique for this purpose would be to use rate correction in addition to the implemented offset correction. An explanation and discussion of offset and rate correction techniques can be found in the chapter *Global Time* of the Real-Time Systems book by Kopetz [57].

Chapter 8

An Application Example: The RailCab Test Track

This chapter shows, by means of an example, how to employ the TrailCable protocol with a real world application. The application chosen is the test track network of the RailCab, a novel train system being developed at the University of Paderborn. The TrailCable technology is especially well suited for the real-time data communication requirements of this application, which is characterized by high flexibility, high throughput, and low latency. A complete hardware and software solution tailored to the RailCab test track was realized and will be presented in this chapter, followed by an outlook on new possibilities of extending the test track functionalities with the support of the TrailCable protocol.

8.1 The RailCab Train System

The history of rail transport dates back to the early 1820's with the introduction of steam locomotives for passenger and freight transport. Since then, uncountable technological improvements have been made to increase safety, comfort, speed, and capacity. Nevertheless, the basic concept has remained the same: rail transport in its classical arrangement consists of monolithic carriages that are made up of one or more locomotives coupled to transport wagons. With an ever increasing demand for decentralized transport demand with complex logistic requirements, new paradigms of rail transport are urgently needed to overcome the limitations imposed by the current model. One contribution in this direction is the RailCab project that aims at increasing the flexibility of rail transport.

The RailCab is an innovative train system that originated as a joint research project of six different institutes at the University of Paderborn [68]. The system is based on independent, small, and autonomous vehicles that are called shuttles. An important advantage of this configuration is the possibility to build up a very flexible transport network where the shuttles are assigned to travel on demand basis rather than on a static schedule, used by the majority of the conventional trains. Moreover, with the RailCab technology shuttles are able to travel non-stop from origin to destination thanks to a computer-based traffic control system. This potentially reduces the total traveling time between cities on secondary routes as changing trains and intermediate stops are no longer required.

In order to increase energy efficiency and track throughput, individual shuttles are able, while running, to come closer and build up convoys and also to travel apart. When running in convoys, shuttles are separated from each other by only a few centimeters, which reduces the drag on the rear vehicles. To add agility to the convoy operations, conventional active rail switches are replaced by passive ones. It becomes therefore a task of the shuttles to go into the intended direction over a rail switch. This is achieved by means of an innovative steering system that is also a key-feature of the RailCab system [33]. Furthermore, shuttles are also equipped with an active suspension and tilt module. This allows for higher comfort and also increased safety, especially when on curves [40].

Traction and braking of the shuttles are based on doubly-fed linear motors, which account for different benefits when employed with the RailCab concept. The application example described in this chapter is closely related to the RailCab driving module. In order to provide a basic background, the linear motor operation is presented below.

8.1.1 Linear Drive

The doubly-fed linear motors [72] consist of a primary and a secondary element. The primary is installed along the rail track, the secondary is located under the shuttles (Figure 8.1) and both elements are built with 3-phase windings. There are two important advantages for the doubly-fed linear motors. Firstly, the asynchronous operation of the two elements allows relative movements between shuttles traveling over the same primary segment. Building up, splitting up, and maintaining convoys are therefore supported by the proposed motor technology. Secondly, there is the possibility to transfer energy from the track to the shuttles, even while the latter are moving. This eliminates the necessity of overhead wires or contact rails along the tracks and dispenses with the necessity to stop shuttles frequently in order to recharge their batteries.

Other advantages of the linear motors include the capability to climb steeper slopes than do conventional trains. The reason is that the traction is achieved by means of electromagnetic forces and does not depend on the contact of the wheels with the track, which can be even contaminated by snow, water or other elements that reduce the required friction. In addition, the lack of movable parts within the motors allows for good reliability and reduced maintenance.

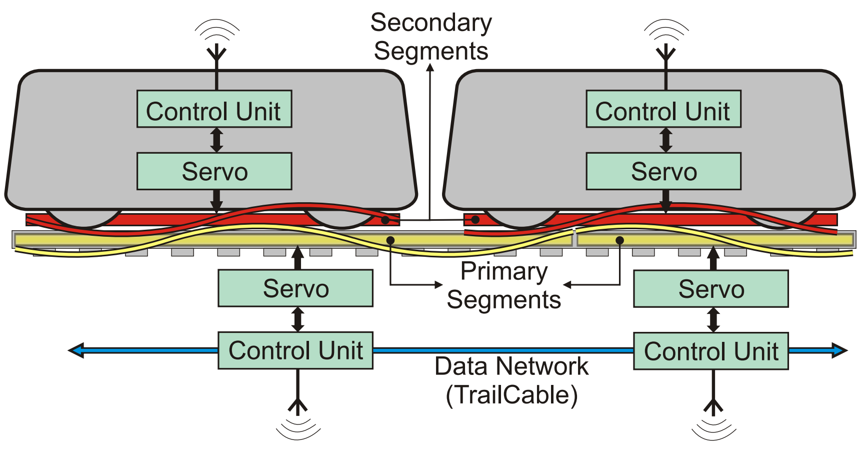


Figure 8.1: Arrangement of shuttles and motors

The shuttles are responsible for adjusting not only their own secondary element, but also the operation of the primary motor element they are running over. This is achieved by means of wireless communication between the shuttles and the rail-track. The required information to be transmitted by the shuttles consists of the electrical current amplitude and frequency set-points for the primary element. However, it is also required to establish a synchronization between the shuttles and the rail-track. With the synchronization mentioned, servo equipments installed at the shuttles and track are able to output consistent values to primary and secondary motor elements at the right time. If the signal phases of the two segments are not set accordingly, efficiency losses as well as incorrect traction forces are likely to be the consequence. A promising synchronization solution is the use of Hall sensors, described in [79]. In this approach, Hall sensors are installed at the front and rear ends of the shuttle to measure the magnetic field of the primary element. With the information from the sensors, a phase-locked loop (PLL) is tuned to the primary magnetic field frequency and provides a reference that allows a synchronized setting of the signals for the secondary element.

Moreover, in order to guarantee a smooth operation of the shuttles, consecutive primary elements must also be synchronized with each other. Only by means of such a synchronization is it possible to generate an electromagnetic field that is continuous in the junction of two primary elements. The synchronization of the primary elements can be achieved by means of a data communication system that triggers simultaneously the reference values for all involved segments.

It is evident from the operation principle of the doubly-fed linear motors that a data communication network is a relevant aspect and must be carefully designed. The following section presents different communication paradigms for the RailCab test-track and shows that an appropriate system architecture is important to assure high scalability and robustness of the desired operation.

8.2 Test Track Network Architecture

A 1:2.5 scale test-track (Figure 8.2) of the RailCab system was built at the University of Paderborn to support the development process of a variety of sub-systems within the project. With a total length of 530 meters, the rail track consists of a straight and an oval section. The primary linear motor elements are installed along the whole track, with the exception of the switch yard and the first meters of the straight section, which are equipped with reaction plates. The primary segments are, on average, six meters long. Each of them is individually connected to a servo equipment, which consists basically of a current controller and the associated power electronics stage. The whole test track facility employs a total of 83 servo devices, which are installed in four power stations along the test track.

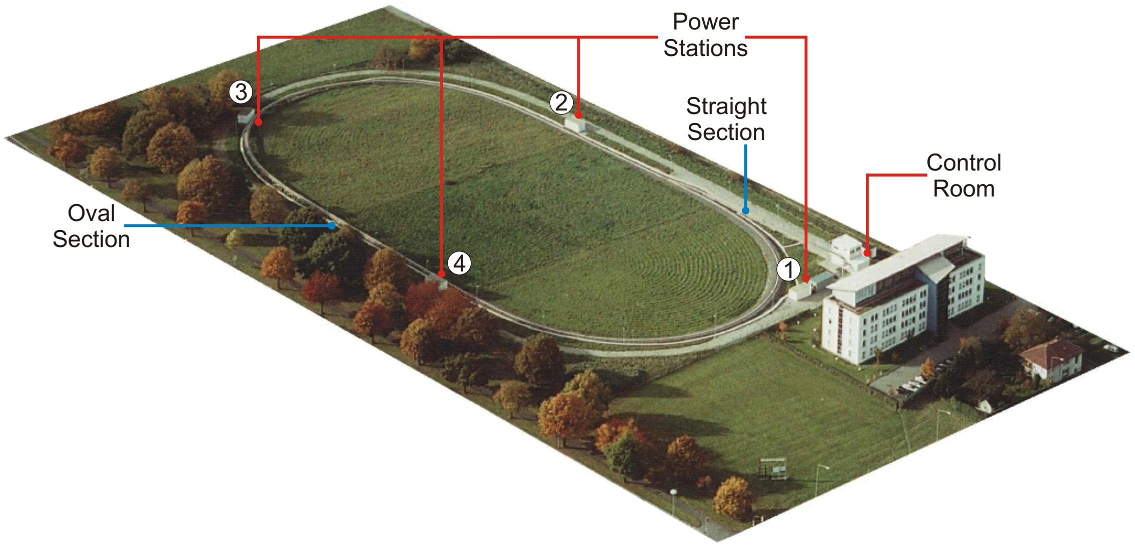


Figure 8.2: Aerial view of the RailCab test-track facility

Another part of the RailCab facility is a control room, where the operation of different systems of the shuttles and the track can be monitored. From this position it is also possible to remotely set parameters and command different actions. In the following sections we will present two network architectures that can be used to interconnect all servo devices, all required electronic control units, and the computers used for the human-machine interface. The first architecture illustrates a centralized approach while the second introduces a distributed solution which is based on the TrailCable real-time communication technology.

8.2.1 Centralized Control Architecture

The centralized architecture was the first to be implemented at the test track. Rather than to evaluate different data communication techniques for the RailCab project, the objective of this initial implementation was to setup an infrastructure that allows experiments related to the linear motors and provides a system for shuttles to run on the test-track in a satisfactory manner. Besides, this was a valuable step to gather experience and information regarding data communication for the current work. A detailed description of the implemented system is presented in [73]. Of special interest here is the architecture (Figure 8.3) and the operation paradigm.

The main component of the centralized control architecture is a network master node. It is responsible for establishing communication with the shuttles via radio interfaces, for processing motor control data and for setting the appropriate reference values of each primary element. The wireless communication is based on radio modems with a link bandwidth of 125 kBaud. Due to the point-to-point nature of the wireless link, the transmission delays are approximately uniform over the time. With this characteristic behavior it was possible to synchronize shuttles with a polling byte transmitted periodically by the network master. This basic synchronization mechanism allows frequency and current amplitudes for the primary and secondary motor elements to be altered simultaneously, this being a condition for a correct generation of the traction and braking forces.

The connection of the network master node to the servo devices is established via CAN bus interfaces. There are four independent buses, one for each power station. Thus, each bus has, including the network master, between 20 and 25 connected devices. Due to the length of the CAN bus wires at the test track, which can reach about 300 meters, and the spurious environment, the transmission bandwidth of the interfaces was reduced to 125 kBaud. Synchronization of all servo devices follows a principle similar to that used by the wireless interfaces. It is achieved by means of a CANopen PDO (process data object) that is also periodically sent via all four buses simultaneously. After the synchronization PDO, the network master node transmits control datagrams that activate the necessary primary segments with the required set-points and turn off segments that are no longer used.

One advantage of the centralized control approach is the simplicity of synchronizing all events, since they are all triggered by the network master node. Moreover, software implementation for the described system is facilitated since many relevant functions on the track side are executed on a single computer. However, a centralized solution is only feasible for a system like the test track that is limited in size. For larger track extensions, distributed approaches are essential and are the focus of the current work.

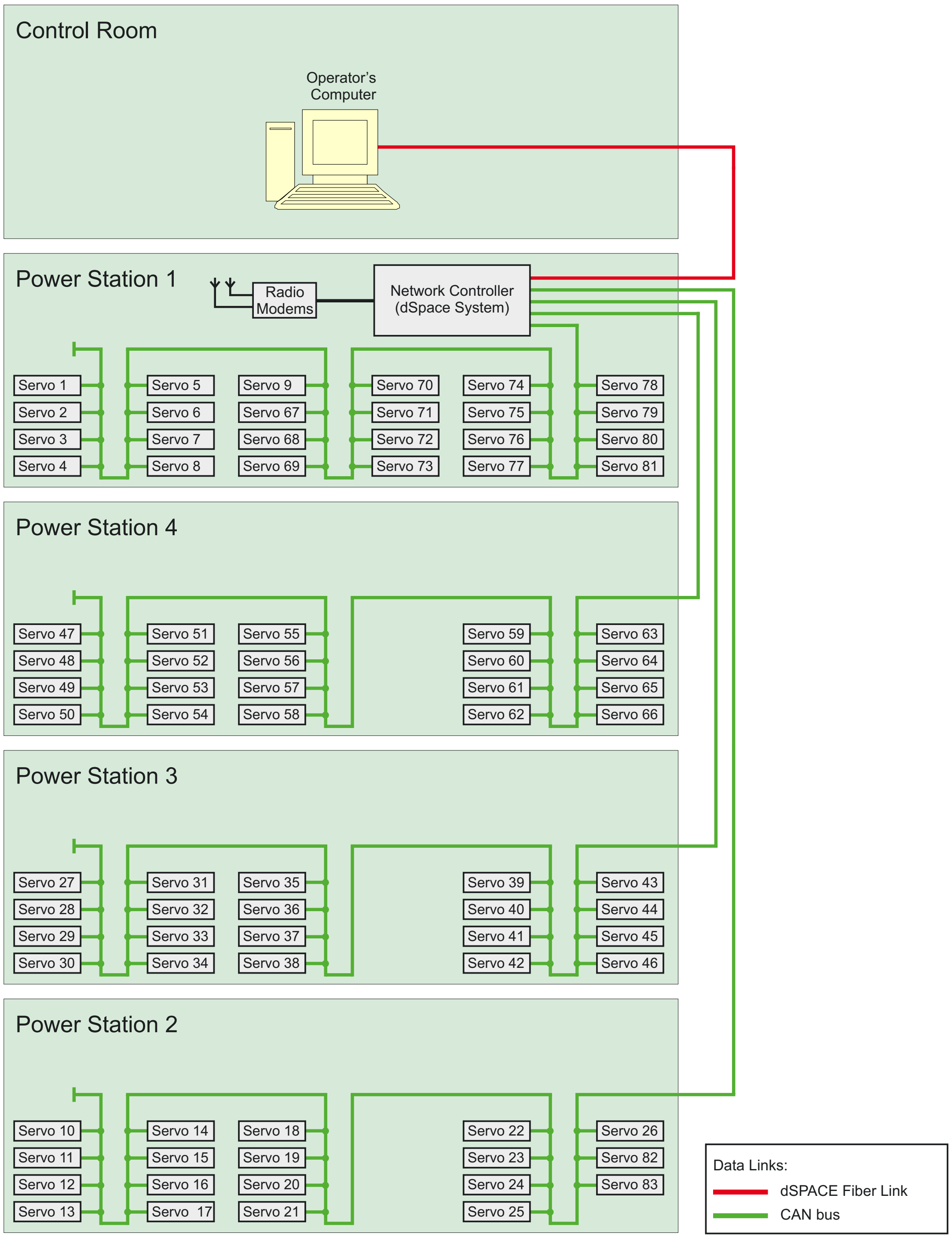


Figure 8.3: Centralized control architecture

8.2.2 Distributed Control Architecture

In order to manage the complexity of a commercial RailCab system, a modular design of the rail-track control network is of vital importance. The benefits of modularity include the possibility to improve system scalability, increase availability, build up fault-tolerance mechanisms by means of redundant components, and better distribute resources along the track (such as ECUs, radio modems, and networking devices). A basic requisite to building a system that is modular, not only with respect to its functionality, but also regarding its physical distribution, is an appropriate data communication system.

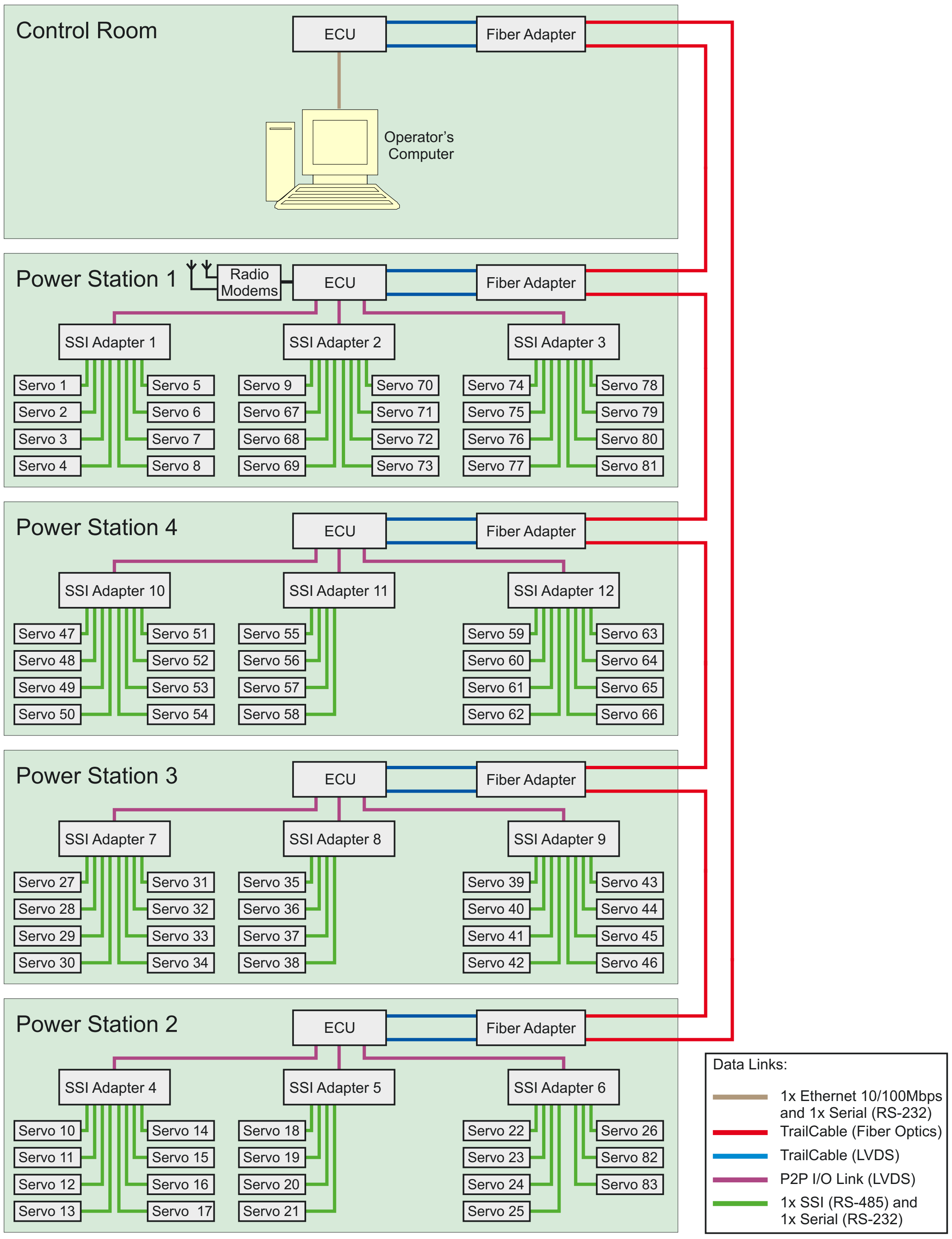


Figure 8.4: Distributed control architecture

Due to real-time behavior, performance and flexibility, among other requirements, the TrailCable protocol is a promising alternative when it comes to building up a control network for the RailCab application. In order to take a first step in applying the TrailCable technology for this purpose, an alternative data communication architecture was developed for the test-track facility. The main goal was to evaluate the TrailCable protocol under real operational conditions. The architecture employed is depicted in Figure 8.4.

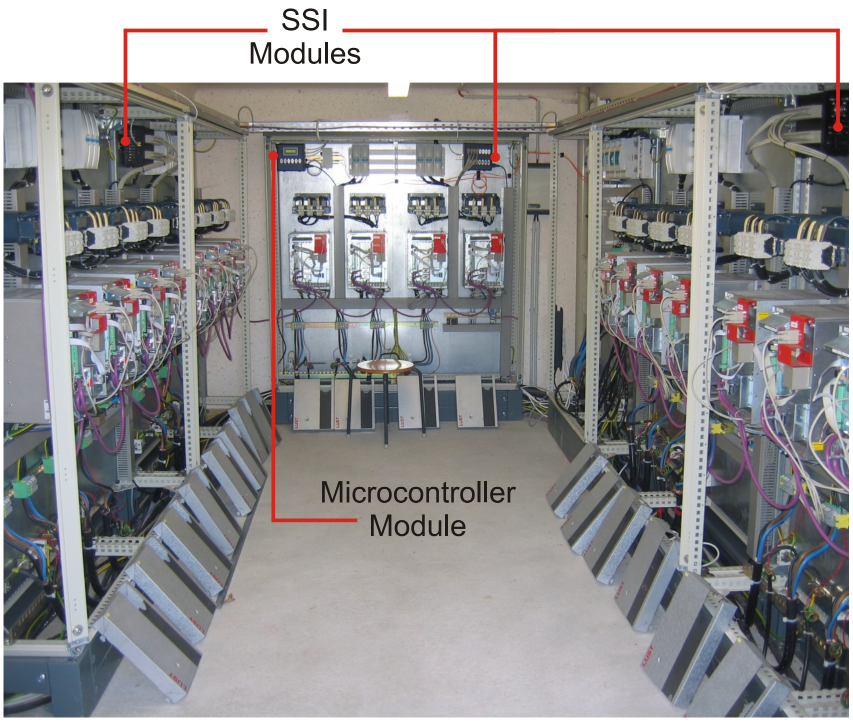


Figure 8.5: Power station

Contrary to the centralized control architecture where the network master is responsible for managing the entire system, this task is performed by five ECUs in the distributed control approach. Inside each power station (Figure 8.5) there is one ECU that controls up to 24 servo devices that are locally connected. In addition, a fifth ECU is located in the control room and used to gather data for monitoring purposes and to broadcast commands from the host-PC to the test track network. Communication between the control room ECU and the host-PC is implemented with the UDP/IP protocol over Ethernet. All ECUs are interconnected via the TrailCable protocol in a ring topology that follows the test track geometry. Fiber-optic cables are employed for the communication links, providing immunity against interferences caused mainly by the high currents of the linear motors. Additionally, for a given transmission bandwidth fiber-optic communication allows the possibility to reach longer link distances as compared to electrical wires.

With this basic configuration, it would be possible to connect radio modems to any ECU. However, in order to maintain physical compatibility with the centralized control architecture and because the wireless communication with the shuttles is not the focus of this work, it was chosen to keep the radio modems exclusively in the power station 1. Never-

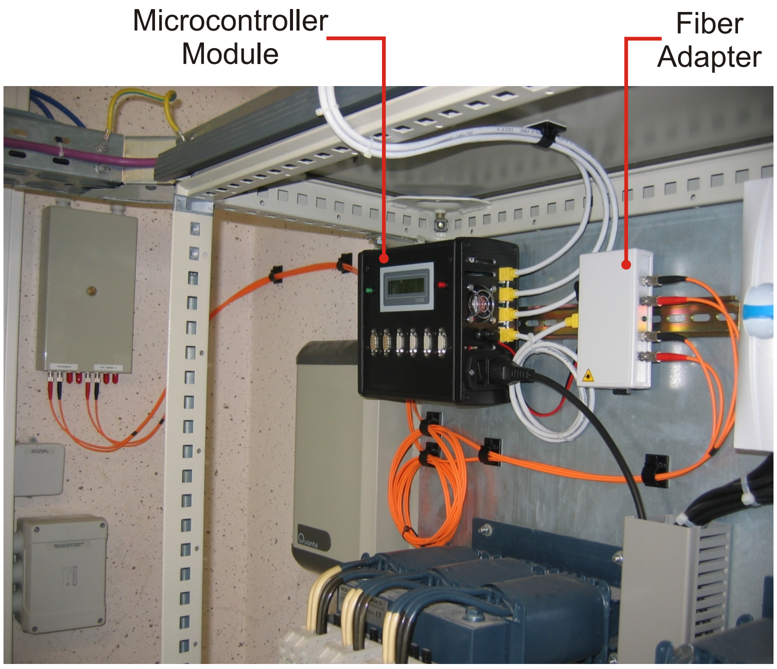


Figure 8.6: Microcontroller module and fiber adapter

theless, the TrailCable protocol can provide a good support for wireless communication in the RailCab system.

The servo devices are connected to the ECUs via three peripheral units. Instead of a CAN bus interface, the servos are connected to the peripheral units via a synchronous serial interface (SSI), based on the RS-485 electrical standard. This is why the peripheral units are also called SSI modules. Each SSI module has the capability to interface to up to eight servos. The main advantage of the SSI over the CAN interface is its higher transmission rate, which allows communication cycles of 8 kHz. This means that reference values can be set every so often, representing a 160-fold gain over the CAN bus for the given application. Moreover, because of the SSI point-to-point links, local communication errors affect only the data transmission of the associated servo device without impairing the remaining network components.

Besides the SSI link between the peripheral unit and the servo device, there is also an additional RS-232 serial interface, which is used for a remote monitoring of the servo equipments from the host-PC located at the control room. The servos were designed to be connected via a serial interface to a PC running a configuration and diagnostics tool provided by the equipment manufacturer. In order to accomplish the same functionality but in a remote manner, the host-PC is connected to the control room ECU with an RS-232 interface. Then, via the test track network data received by this ECU is forwarded transparently to the desired RS-232 interface of a given peripheral unit and reaches the intended servo. The same happens in the opposite direction. With this mechanism one servo at a time can be selected to be connected to the host-PC, allowing high flexibility

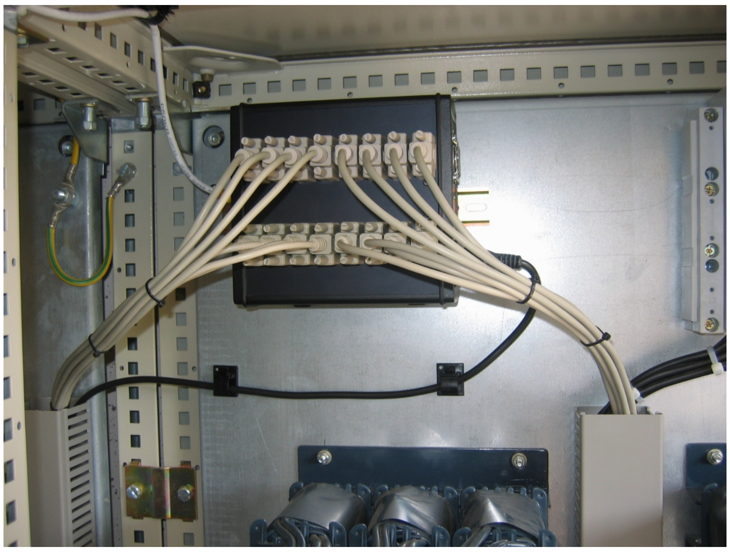


Figure 8.7: SSI module

for monitoring purposes.

Finally, the data links between the ECUs and the SSI modules are implemented by means of a proprietary point-to-point connection based on the LVDS standard. This proprietary interface is designed to carry application-specific data packets in a periodic manner. Employing this solution instead of the full-featured TrailCable protocol itself is motivated by the fact that the communication functionality is rather simple and the SSI modules have no microprocessor. Thus, less resources such as FPGA area and processing power are required.

After an introduction to the distributed control architecture, further details of the implemented system will be presented below. The next section presents the developed hardware and software components as well as the linear motor control strategy for the test track.

8.3 Test Track Implementation

A complete hardware and software solution was designed and implemented to evaluate the TrailCable protocol at the RailCab test track facility. One of the objectives was to create a system configuration that is based on components that are either limited in processing power (microcontrollers) or area (FPGAs) resources, but with an overall high performance. This could be achieved due to the FPGA-implemented communication functions, which contributed to reduce processor utilization, and due to a careful planning and scheduling of the software routines. In order to show how this strategy was pursued, the implemented system is described below. Firstly, the realized hardware is presented with both the electronic board features and the functionality of the FPGAs employed being explained. Then, software tools for the host PC, which were developed to support system operation and monitoring, are introduced. Finally, we will deal with the control software for the test track ECUs.

8.3.1 Hardware Components

There are three different types of hardware components that were developed to be used at the test track facility: the ECU board, the SSI module and the fiber-optic adapter. The distributed control architecture was created with the objective of limiting the number of hardware components to be installed. This reduced costs and system complexity, but still allowed evaluation of the developed components, especially the TrailCable protocol. For this reason, each power station is equipped with only one ECU, responsible for managing all local servo devices and executing communication routines. The SSI modules are used as a communication bridge between the ECU and the servos. Because all functions are processed in parallel in the FPGA of the SSI module, it was possible to add data interfaces for up to 8 servo equipments, which considerably reduces the number of required hardware elements. The three hardware component types are individually presented below.

8.3.1.1 ECU Module

The ECU board (Figure 8.8) is a new development derived from the Rabbit project [8] and consists of two main components: a Freescale PowerPC MPC555 microcontroller [36] and a Xilinx Virtex-400E FPGA [93], both interconnected and operating with a clock frequency of 40 MHz. The microcontroller is assembled with 4 MB RAM and 4 MB Flash memories in a piggy-back board provided by Phytec, a third-party company [71].

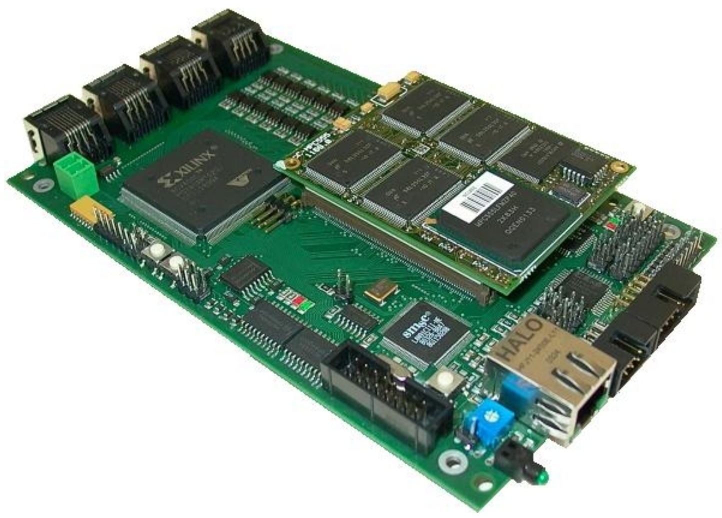


Figure 8.8: ECU board

Additional features of the developed ECU board include a 10/100 Mbps Ethernet interface, an LCD-display interface, a CPLD for glue logic, and FPGA configuration via the microcontroller, a 2 Mbit serial flash, four LVDS connectors (each one containing four transmission pairs), serial RS-232 interfaces from both the microcontroller and the FPGA, CAN bus interface, and a variety of LEDs and switches circuitry. The microcontroller and the FPGA are programmed via BDM and JTAG interfaces, respectively. The block diagram of the ECU board is shown in Figure 8.9.

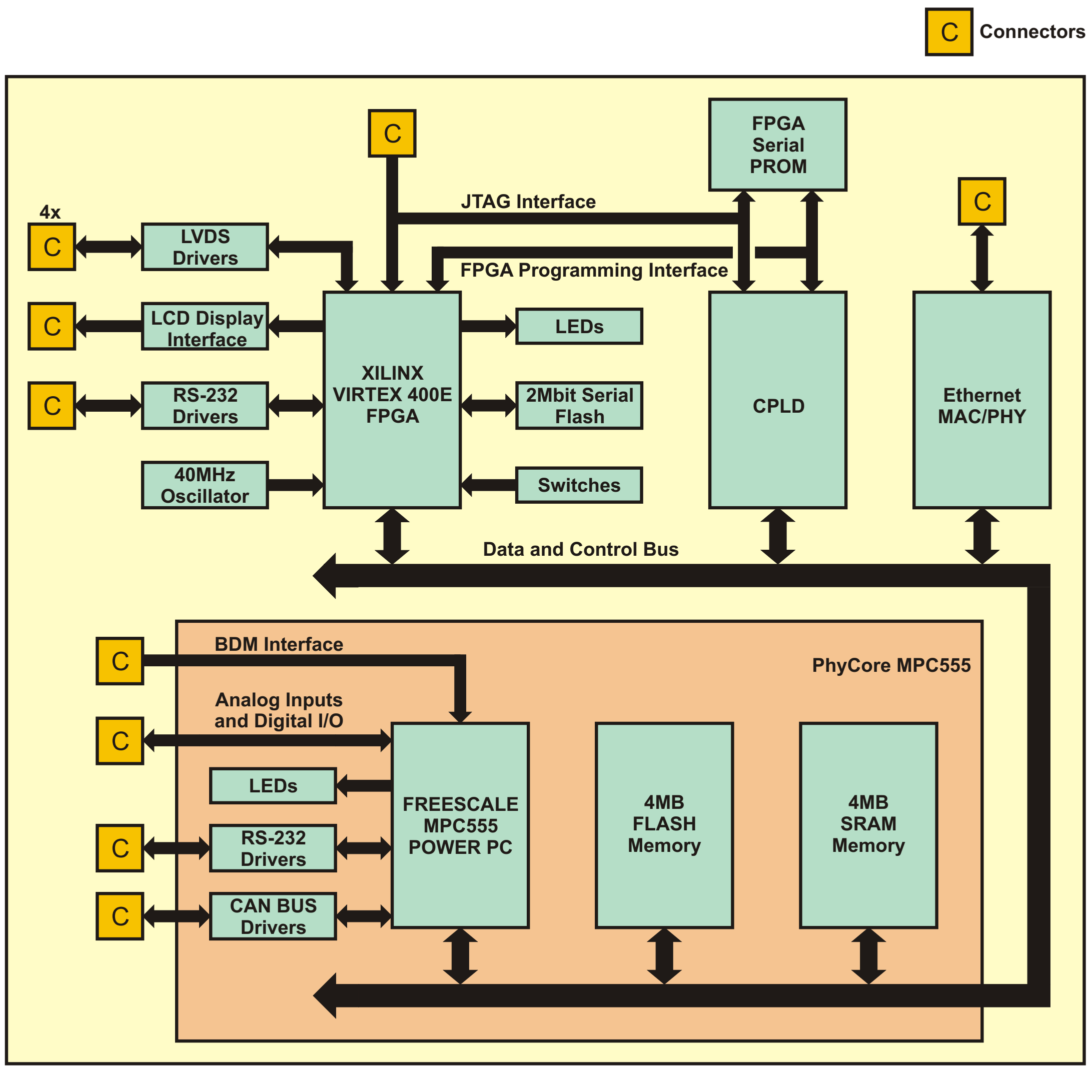


Figure 8.9: ECU board diagram

The FPGA is connected to the microcontroller via a 32-bit external bus interface so that the FPGA can be accessed like an external memory. The microcontroller acts as a master and is responsible for initiating read and write accesses to data in the FPGA memories. One advantage of this approach is a high achievable transfer rate, due to the low-latency and 32-bit data access. Additionally, interrupt (IRQ) lines are available from the FPGA to the microcontroller, allowing triggering interrupt routines upon programmed events in the reconfigurable logic device.

Apart from containing the TrailCable communication engine, the FPGA is used for other purposes as well. An optimized implementation of the components allowed the following additional functions to be placed in the FPGA: a communication interface for three SSI modules, an LCD-display driver, an RS-232 communication interface (UART), a watchdog, and an I/O component to read the state of jumpers and set digital outputs. The architecture of the entire FPGA design is shown in Figure 8.10.

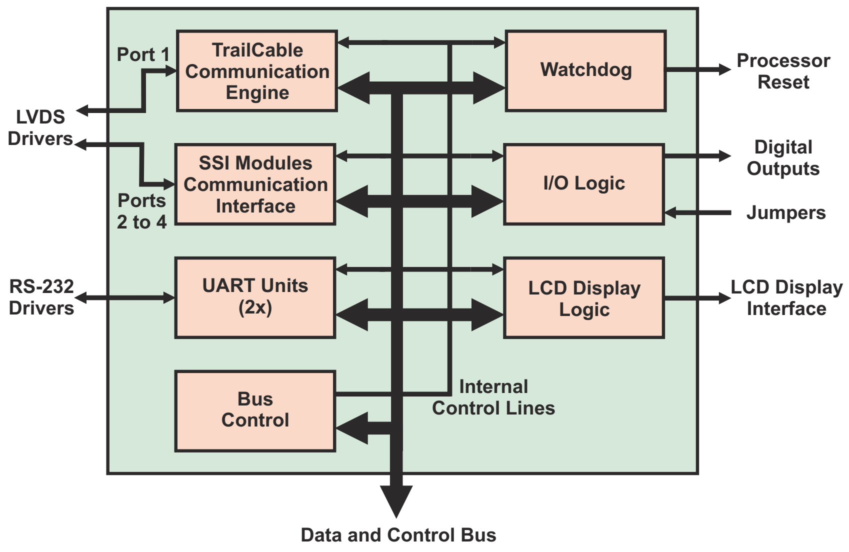


Figure 8.10: ECU FPGA diagram

The TrailCable communication engine module was built as described in Chapter 6. It consists of one host port and two communication ports. For the test track application, instead of using 32-bit registers a 16-bit architecture was chosen since it suffices for deadlines and periods in the range of a few microseconds. The 16-bit variant allows reduction of the required FPGA area.

Although the Freescale MPC555 microcontroller already provides RS-232 serial communication interfaces, adding a similar functionality to the FPGA at the ECU board proved beneficial. The UART developed requires only a small amount of resources when compared to other FPGA components and allows the use of a very simple software driver running in the MPC555 microcontroller. An FPGA-based UART was also originally required for the SSI module due to the lack of a microcontroller in that board so that only a small development effort was necessary to use the same component in the ECU board.

One of the main components of the FPGA architecture is a proprietary communication interface for the connection of up to three SSI modules. Each of the three available interfaces consists of two full-duplex data links running at 32 Mbps, allowing low communication latencies. Each full-duplex link is responsible for data exchange with 4 servos and thus a total of 8 of such devices can be addressed by a single interface.

When creating the communication interface for the SSI modules, special attention was paid to guarantee synchronization of all servo devices. Since the ECU boards are synchronized with each other by means of the TrailCable protocol, it is possible to use the ECUs as time masters for the servos. The communication logic with the SSI modules was designed to initiate simultaneous packet transmission in all interfaces when triggered by the TrailCable time-reference tick. In fact, packet transmission in all links starts at the same clock cycle. Since the SSI modules are synchronized with the ECUs by means of such packet transmissions, all servos installed in a given power station can thus be synchronized with each other with high precision. So, when it comes to the whole test track, synchronization among all servos depends primarily on the quality of the clock synchronization among all ECUs, which is also very precise (see Section 7.3). For this reason, all 83 servos at the test track can be triggered virtually simultaneously since the jitter is less than 1 μs.

The communication logic with the SSI modules has some characteristics inherited from the physical layer of the TrailCable protocol: both the same electrical standard (LVDS) and the coding scheme (8B/10B) are used. The packet format (Figure 8.11), however, is different and was designed to suit specific application needs. When no data is sent, idle control commas are continuously transmitted. A new packet is sent every 250 μs and consists of a control byte (from the ECU to the SSI modules) or a status byte (from the SSI modules to the ECU), 10 data bytes for each one of the four servos, a CRC byte (calculated from all 41 bytes sent previously) and up to 8 bytes for asynchronous communication. The transmission of an entire packet takes only 12.5 μs.

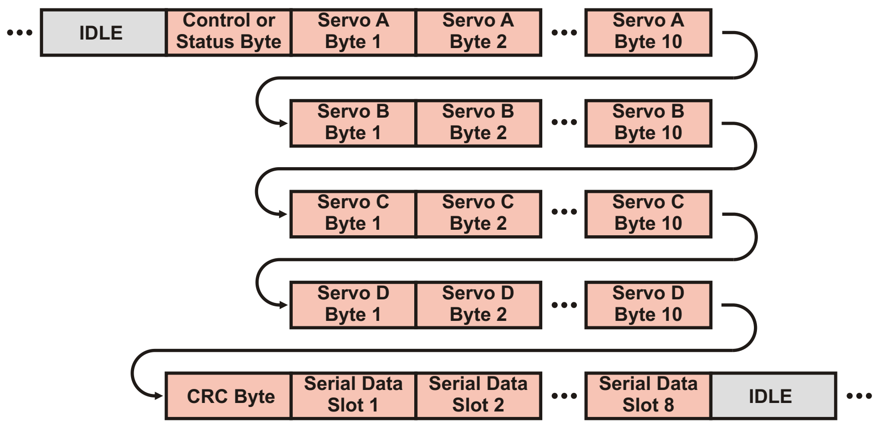


Figure 8.11: Packet format for communication with SSI modules

The status byte sent by the SSI modules to the ECU indicates whether the servos are properly connected to the peripheral units. Additionally, the ECU is also able to retrieve not only the condition of its own data links but also those of the SSI modules. This allows automatic detection of link failures of the entire test track communication network, regardless of the type of communication interface used.

A block diagram of the SSI communication component is shown in Figure 8.12. The architecture provides both synchronous and asynchronous communication features. The former is used to transfer set-point values to the servos and to retrieve operation status from them. For this purpose, at every communication cycle the contents of a dual-port RAM memory are read and transmitted in the packet section reserved to carry data to the servos. Using such a memory instead of an FIFO permits higher software performance, since only part of the memory contents need to be updated periodically from one packet to the other. On the other hand, the asynchronous communication obtains data to be transmitted from an FIFO, since the number of bytes to be sent is unknown and not constant. As long as the transmission FIFO is not empty, each byte will be put in the assigned serial slots at the end of the data packet. When the FIFO is empty, idle control commas are transmitted instead of data. The asynchronous communication is used for a transparent tunneling of the RS-232 serial interface between the control room ECU and one selected servo, as described in the distributed control architecture.

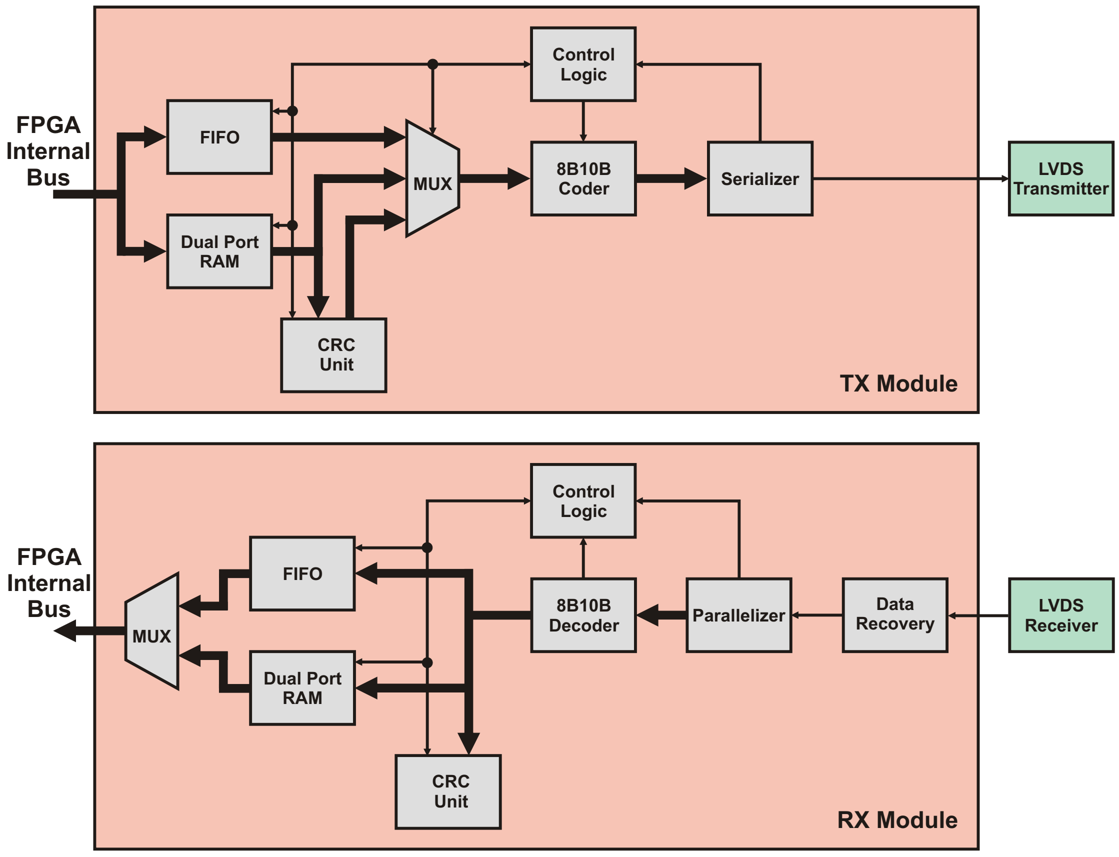


Figure 8.12: Communication interface of the SSI modules

In addition to the synchronization, it was also important to integrate safety mechanisms into the communication interfaces in the three SSI modules. Without such a mechanism, if the microcontroller fails to update the internal dual-port memory, the set-point values transferred to the servos would be kept constant, leading to potential errors. In order to cope with this problem, a watchdog was included in the FPGA design. If the microcontroller does not generate “live” signals within specified time intervals, an emergency state will be activated. When this happens, predefined safety packets instead of the normal memory contents are transmitted to the servos which are in turn deactivated to avoid incorrect operation. Moreover, when the watchdog goes into the emergency state, the microcontroller can automatically be reset to give it the opportunity to recover from transient failures.

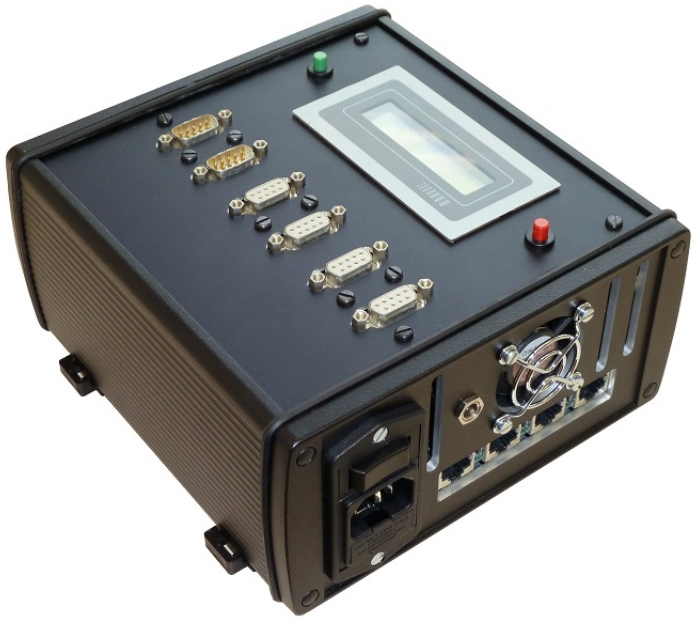


Figure 8.13: ECU module case

For installation at the test track, the ECU board was placed in an appropriate casing (Figure 8.13) to protect the electronic components and to facilitate cabling and fixation. The ECU module case incorporates an internal AC adapter for the power supply and a fan for heat dissipation. Additional components include an LCD display, connectors, and push-buttons and an auxiliary power output.

8.3.1.2 SSI Module

The SSI board (Figure 8.14) is employed as a bridge that has an LVDS proprietary connection to an ECU on the one side and eight servo interfaces on the other side, each containing both an SSI and an RS-232 link.

The core component of the board (Figure 8.15) is a Xilinx Spartan-2 200 FPGA [90] which is responsible for all data processing as no microcontroller is available in the SSI module. All communication interfaces are controlled individually, so that a simultaneous operation of all links is possible.

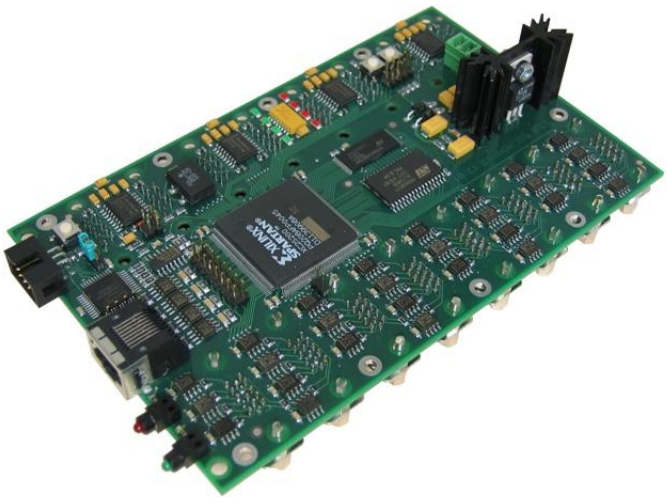


Figure 8.14: SSI board

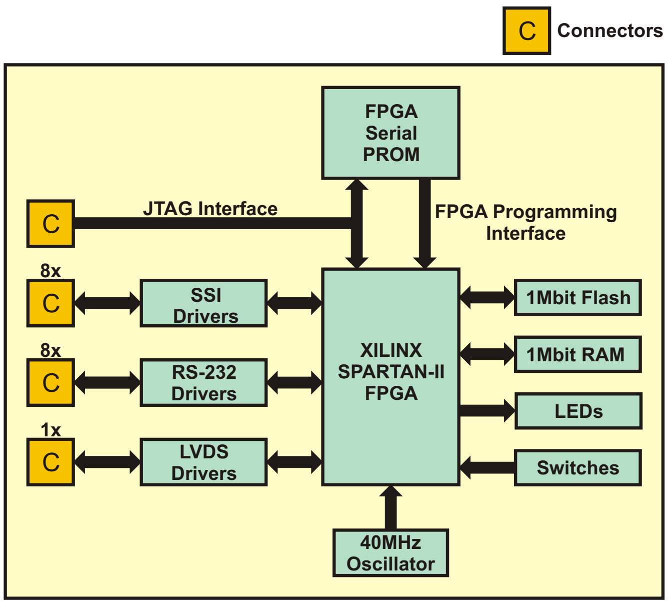


Figure 8.15: SSI board diagram

As already mentioned, the LVDS interface between ECU and an SSI module consists of two full-duplex data links. The wires required for both links are physically integrated in a single CAT5 cable to facilitate installation. Since each full-duplex link is assigned to the data transfer of 4 servos, we chose to build up two identical processing modules in the FPGA that work completely independently of one another (Figure 8.16).

When the SSI module receives a packet transmitted by the ECU, it is checked whether the correspondent CRC-byte matches. If so, data is read and the packet arrival time is taken into account for the synchronization of the corresponding SSI interfaces. Should no packet be received within a determined time interval, an emergency state will be activated, similarly to the ECU board. In this situation, the peripheral unit allows for a

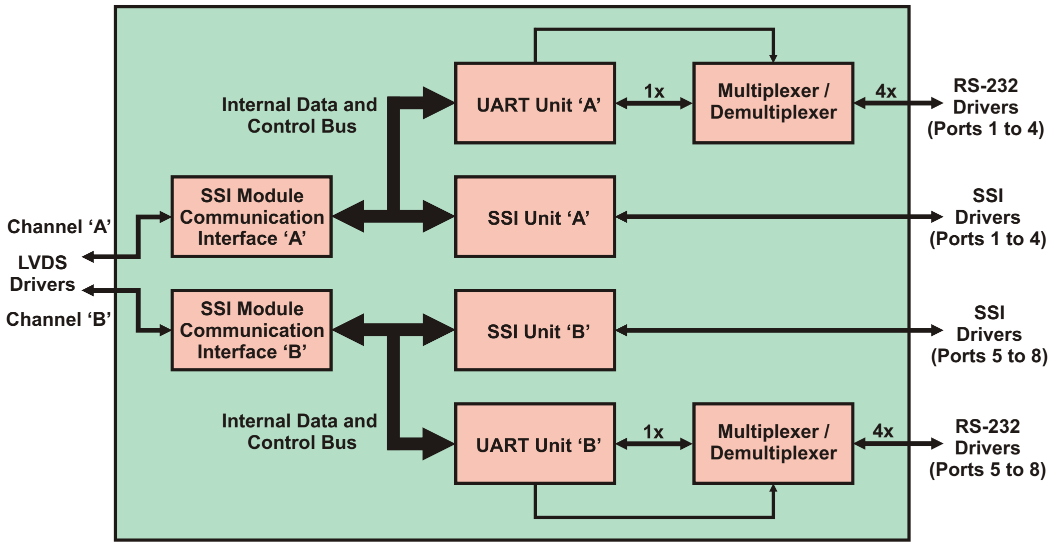

Internal Data and Control Bus
UART Unit 'A'
1x
Multiplexer / Demultiplexer
4x
RS-232 Drivers (Ports 1 to 4)
Channel 'A'
SSI Module Communication Interface 'A'
SSI Unit 'A'
SSI Drivers (Ports 1 to 4)
LVDS Drivers
Channel 'B'
SSI Module Communication Interface 'B'
SSI Unit 'B'
SSI Drivers (Ports 5 to 8)
Internal Data and Control Bus
UART Unit 'B'
1x
Multiplexer / Demultiplexer
4x
RS-232 Drivers (Ports 5 to 8)

Figure 8.16: SSI FPGA diagram

direct counter-measure as it can automatically deactivate the servos via the SSI interface. As a result, if an error occurs with either the ECU board or with the physical connection between the ECU and SSI modules, safety mechanisms will automatically shut down the servos to minimize the outcomes of operation disruption.

The signals of the SSI interface (Figure 8.17) between the peripheral units and the servo devices consist of a synchronization clock, serial data transmission (TXD) and reception (RXD) lines, each one with a clock reference (TXC and RXC, respectively). The SSI module takes the master role and generates the synchronization clock signal of 8 kHz (125 μs period) based on the time reference retrieved from the ECU module. After the rising edge of the synchronization clock signal, data transmission starts and 10 bytes are transferred from the SSI module to the servos and vice versa. Each data byte is preceded by a start bit and followed by a stop bit. The data transmission operates at 20 MBd, thus allowing low communication latency.

Figure 8.18 shows the timing characteristic for read and write data accesses with the servos. At each time-reference tick from the TrailCable communication engine, a packet is generated at the ECU module and is transmitted to the SSI module. When it arrives at the destination, the data transfer of the SSI interfaces is triggered. As soon as the data read from the servos via the SSI interface is available, it is sent back to the ECU module. The whole process takes less than 40 μs to complete. Such a performance is necessary for allowing a control cycle with all required processing and communication functions to run with a 250 μs period.

The synchronous data transmission capability of the communication interface between the ECU and the SSI modules is used to transfer reference values and status information to and from the servo devices. On the other hand, the part of the data packets reserved

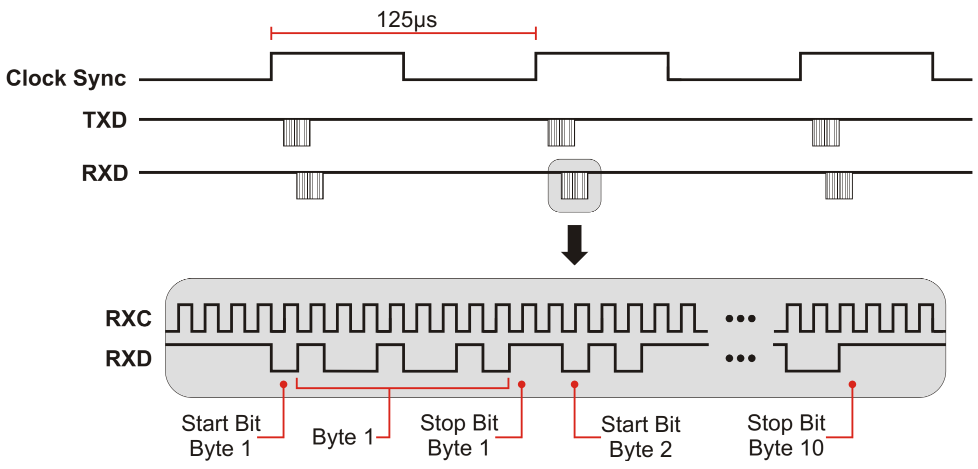


Figure 8.17: SSI interface

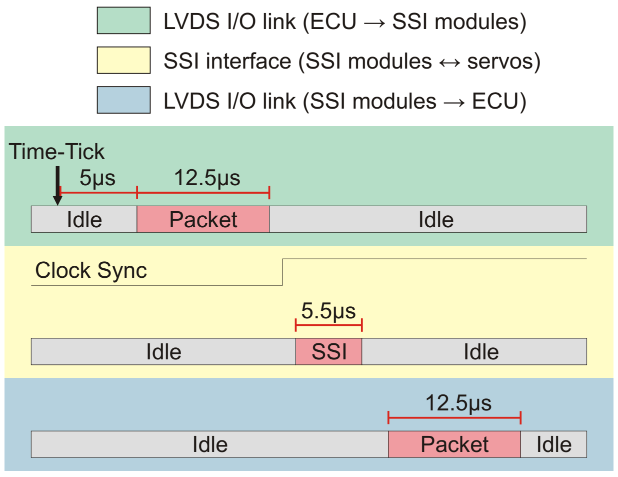

LVDS I/O link (ECU → SSI modules)
SSI interface (SSI modules ↔ servos)
LVDS I/O link (SSI modules → ECU)
Time-Tick
5µs
12.5µs
Idle
Packet
Idle
Clock Sync
5.5µs
Idle
SSI
Idle
12.5µs
Idle
Packet
Idle

Figure 8.18: Timing diagram of servo data accesses

to asynchronous transfers is employed to create a bridge mechanism between the ECU and the servos via an RS-232 serial interface. In order to establish a connection, the control byte (Figure 8.11) sent in the packet from the ECU to the SSI module selects one RS-232 interface to be activated at a given time. By this, whenever there is data in the asynchronous packet segment, the respective bytes are forwarded to the selected servo via the RS-232 interface. Data received by the SSI module in the RS-232 interface is also appended to the packet sent back to the ECU. This allows the ECU to establish a transparent and bidirectional connection with one of the RS-232 servo interfaces.

The casing of the SSI module (Figure 8.19) is similar to that of the ECU. It also contains an internal AC adapter, a fan for heat dissipation, and can be fixed in a DIN rail.

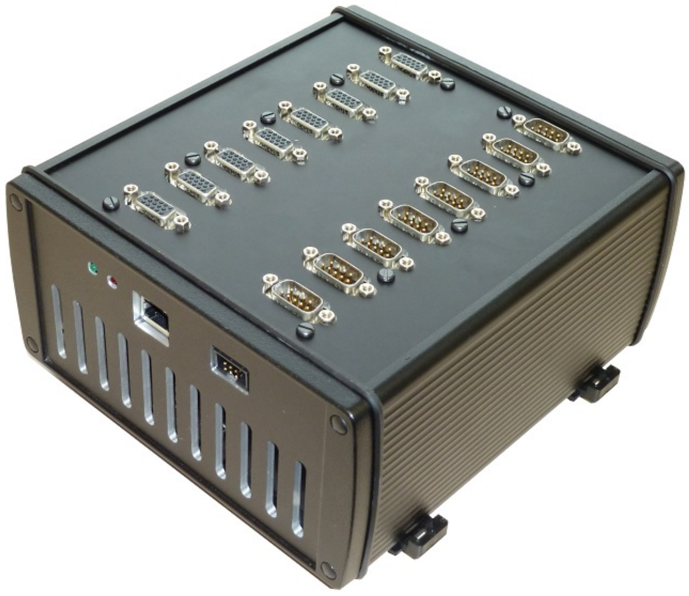


Figure 8.19: SSI module case

8.3.1.3 Fiber-Optic Adapter

The fiber-optic adapter (Figure 8.20) is the simplest and smallest of all 3 hardware boards. Thanks to the 8B/10B coding scheme used by the TrailCable protocol physical layer, which is appropriate for both electrical and optical transmissions, all the required functionality is performed exclusively by the communication drivers circuitry. Although a more complex design could be considered to reshape transmission signals or to implement additional functionalities, the chosen approach proved to be efficient for the intended use.

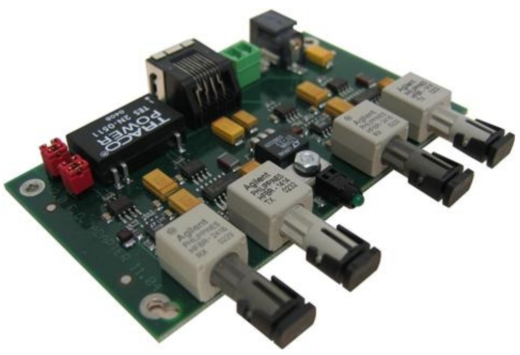


Figure 8.20: Fiber-optic adapter board

There are two bidirectional ports available at the adapter. Each one consists of an LVDS receiver connected to an optical transmitter and an optical receiver connected to an LVDS transmitter (Figure 8.21). Both LVDS and optical interfaces work with the same baud rate. The adapter ports therefore allow a completely transparent electrical/optical conversion of the data links, without the need for any configuration. Thus, by simply connecting the electrical cables of the TrailCable protocol to the adapter, it is possible to achieve optical links of up to 800 meters at 32 MBd with multi mode 62.5/125 μm fibers [16]. Longer cabling and the adapter itself have, however, an influence on the link propagation delay. Nevertheless, the TrailCable can cope with this issue since it is able to measure the propagation delay correctly by means of its native link control mechanism.

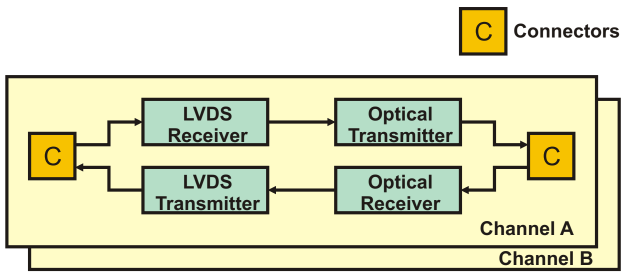


Figure 8.21: Fiber-optic board diagram

Like the ECU and SSI modules, the fiber-optic adapter board is placed in a DIN rail mounted case (Figure 8.22). The power supply is provided by an external + 5 V source, which can be either an AC adapter or the auxiliary output of the ECU module.

Figure 8.22: Fiber-optic adapter case

8.3.2 Software Tools

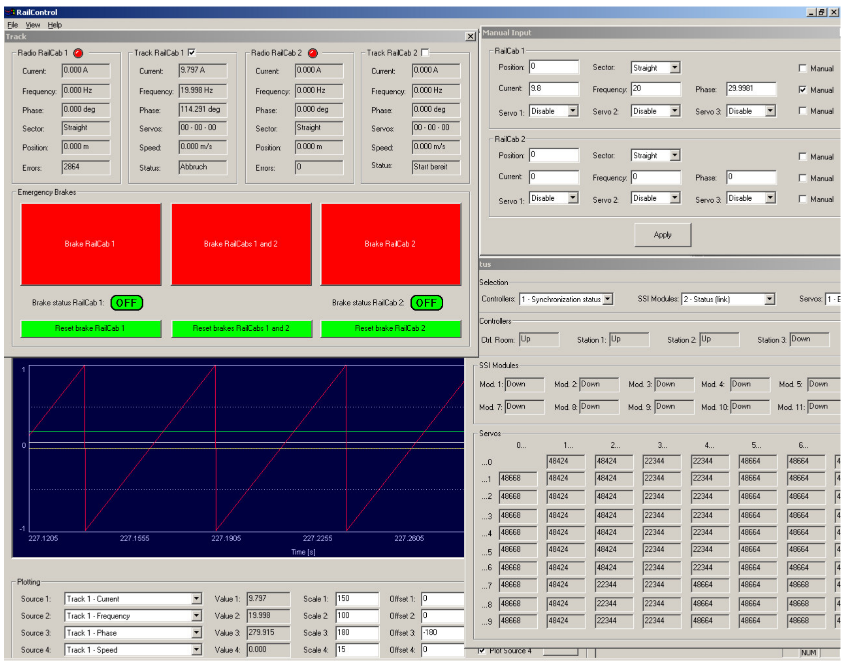
Due to the high complexity of the system, there was a need to develop software tools to support a variety of configuration and monitoring activities. There are two specifically designed software tools: the RailControl and the RailView programs. The RailControl is required for test track operation, whereas the RailView is an optional visualization tool. Since both tools are completely independent of one another, it is possible to execute them on a single PC or different ones.

Besides the developed tools, a third-party program, distributed by the servo manufacturer, can be also employed. By means of this tool it is possible to establish a connection between the PC and one servo device via an RS-232 interface. The tool, called *Drive Manager* [62], allows the parametrization and monitoring of the servo equipment. One of the features of the test track network is to allow a transparent tunneling between the PC running the *Drive Manager* software, which is located at the control room, and one of the 83 remote servos. The servo to be used in the tunneling process is selected with the RailControl tool and can be altered at run-time by the system operator.

In order to obtain a better understanding of the distributed control functions running in the ECUs, it is worth taking a deeper look at the features of the two developed tools first. The RailControl and the RailView utilities will therefore be introduced below.

8.3.2.1 RailControl Tool

The RailControl tool offers a variety of functions to support the test track operation. It is used to activate the servo devices, set emergency brakes of the shuttles, monitor and log the system state, among other features. In order to group the tool features and facilitate usability, different windows were used. These are:



Radio RailCab 1		Track RailCab 1 ✓		Radio RailCab 2		Track RailCab 2 ☐	
Current:	0.000 A	Current:	9.797 A	Current:	0.000 A	Current:	0.000 A
Frequency:	0.000 Hz	Frequency:	19.998 Hz	Frequency:	0.000 Hz	Frequency:	0.000 Hz
Phase:	0.000 deg	Phase:	114.291 deg	Phase:	0.000 deg	Phase:	0.000 deg
Sector:	Straight	Servos:	00 - 00 - 00	Sector:	Straight	Servos:	00 - 00 - 00
Position:	0.000 m	Speed:	0.000 m/s	Position:	0.000 m	Speed:	0.000 m/s
Errors:	2864	Status:	Abbruch	Errors:	0	Status:	Start bereit

Brake status RailCab 1: OFF
Brake status RailCab 2: OFF

Plotting

Source 1:	Track 1 - Current	Value 1:	9.797	Scale 1:	150	Offset 1:	0
Source 2:	Track 1 - Frequency	Value 2:	19.998	Scale 2:	100	Offset 2:	0
Source 3:	Track 1 - Phase	Value 3:	279.915	Scale 3:	180	Offset 3:	-180
Source 4:	Track 1 - Speed	Value 4:	0.000	Scale 4:	15	Offset 4:	0

Manual Input

RailCab 1

Position:	0	Sector:	Straight			☐ Manual
Current:	9.8	Frequency:	20	Phase:	29.9981	✓ Manual
Servo 1:	Disable	Servo 2:	Disable	Servo 3:	Disable	☐ Manual

RailCab 2

Position:	0	Sector:	Straight			☐ Manual
Current:	0	Frequency:	0	Phase:	0	☐ Manual
Servo 1:	Disable	Servo 2:	Disable	Servo 3:	Disable	☐ Manual

Selection

Controllers: 1 - Synchronization status SSI Modules: 2 - Status (link) Servos: 1 - E

Controllers

Ctrl. Room: Up Station 1: Up Station 2: Up Station 3: Down

SSI Modules

Mod. 1:	Down	Mod. 2:	Down	Mod. 3:	Down	Mod. 4:	Down	Mod. 5:	Down
Mod. 7:	Down	Mod. 8:	Down	Mod. 9:	Down	Mod. 10:	Down	Mod. 11:	Down

Servos

	0...	1...	2...	3...	4...	5...	6...
...0		48424	48424	22344	22344	48664	48664
...1	48668	48424	48424	22344	22344	48664	48664
...2	48668	48424	48424	22344	22344	48664	48664
...3	48668	48424	48424	22344	22344	48664	48664
...4	48668	48424	48424	22344	22344	48664	48664
...5	48668	48424	48424	22344	22344	48664	48664
...6	48668	48424	48424	22344	22344	48664	48664
...7	48668	48424	22344	22344	48664	48664	48668
...8	48668	48424	22344	22344	48664	48664	48668
...9	48668	48424	22344	22344	48664	48664	48668

✓ Plot Source 4

Figure 8.23: RailControl screenshot

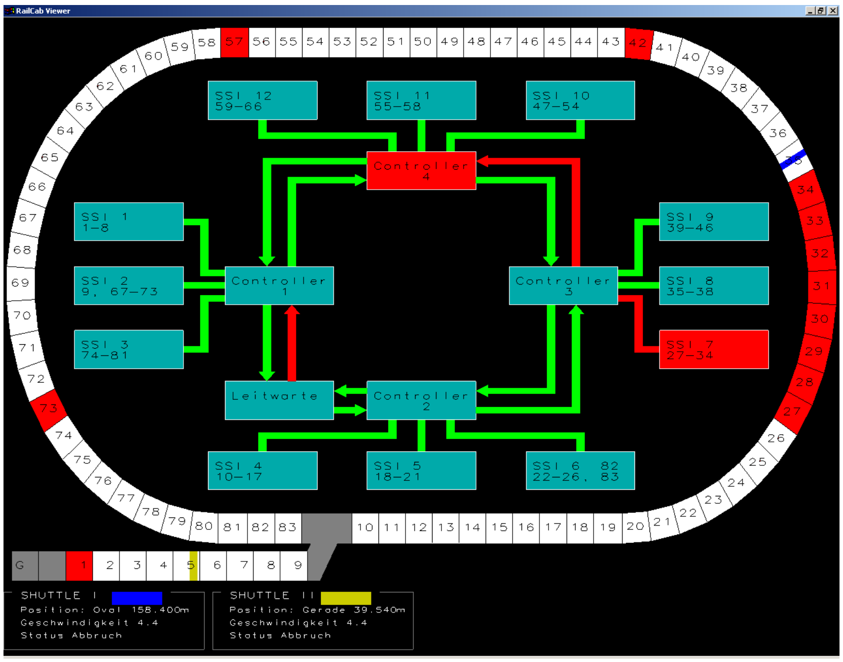
- Setup - used to manage the UDP/IP connection to the control room ECU, to manually activate safety flash lights at the test track and to select a servo device for the DriveManager utility tunneling mechanism.

- Track - is the main control window and allows the user to activate the test track, for the operation of one or two shuttles. It also presents radio link status, the reference values received from the shuttles via the wireless interface, the enabled servo devices, and their respective references values. Additionally, the operator has the possibility to set the emergency brakes of each shuttle independently.

- Status - as the name suggests, it is used to show a large number of the test track hardware parameters, such as communication error counters, synchronization information, number of detected peripheral units, servo operation condition, and others.
- Plotting - used to plot test track signals during system run-time. Also provides a logging capability in order to support off-line data analysis.
- Manual Input Control - allows the user to manually set the reference values for the servo equipments instead of processing of parameters transmitted by the shuttles.

8.3.2.2 RailView Tool

Opposed to RailControl, the RailView program is not strictly necessary for controlling the system, but plays an important role for monitoring the operations. The RailView utility is an OpenGL-based visualization tool that gives the operator the possibility to check system status at a glance. The test track is graphically shown with all 83 primary segments represented individually by means of a specified color code, which allows the status or malfunction of motors or servos to be promptly identified. Moreover, the position of the shuttles is also presented in the visual track model, allowing the operator to check whether the primary segments are being activated and deactivated consistently as the vehicles are passing by.



SHUTTLE I
Position: Oval 158.400m
Geschwindigkeit 4.4
Status Abbruch

SHUTTLE II
Position: Gerade 39.540m
Geschwindigkeit 4.4
Status Abbruch

Figure 8.24: RailView screenshot

The RailView tool shows the status of all ECUs, SSI modules, and the entire data network. For the main communication ring implemented with the TrailCable protocol, there is even the possibility to detect failures in each direction of transmission. In the unlikely event of electronic or link failures the faulty component is highlighted to allow the operator to take immediate counter-measures. The RailView therefore is a useful tool to reduce operator workload and support system operation.

8.3.3 System Software

This section introduces the system software of the ECUs used in the test track, which heavily depends on data communication due to the system-inherent distributed characteristic. The system software is responsible for three different groups of functions: motor control, status identification, and data tunneling.

All function groups are integrated in a common run-time framework, based on a TDMA scheduling. The basic cycle time of the scheduling plan is 4 ms, divided into 16 rounds of 250 μs (Figure 8.25). The necessity to design the schedule with 250 μs rounds arises from the fact that the servo devices get new reference values with a 4 kHz rate. Apart from the motor control routines that must run every 250 μs, the remaining monitoring and management functionalities are divided by means of a cyclic executive and build up a scheduling plan that spans over all 16 rounds. The start times of the 4 ms cycles of all ECUs are synchronized by means of the TrailCable protocol and as a consequence each one of the 16 rounds is also triggered at approximately the same time in different microcontrollers. Such a synchronization is a prerequisite for the distributed TDMA scheduling and allows a consistent operation of all ECUs.

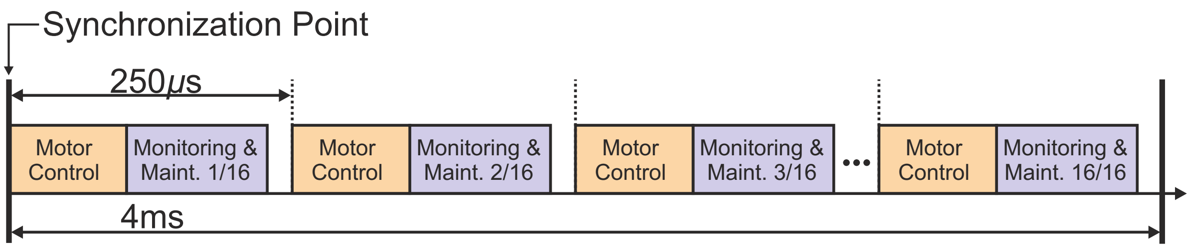


Figure 8.25: Communication cycle of the system software

A real-time operating system was not used for the described implementation. Because no real-time operating system is available, the software running in the ECUs is responsible for triggering the TDMA scheduling plan. This is supported and facilitated by the TrailCable communication engine, which generates periodic interrupt signals that invoke interrupt service routines (ISR) to execute real-time processes. After completion of each ISR, the program flow returns to a polling loop. Functions of non-real-time nature, such as the

UDP/IP communication executed by the ECU in the control room or the LCD management, are handled exclusively in the polling loop. The real-time processing triggered by the ISRs suspend the polling loop and has, therefore, higher priority. At each of the 16 rounds, predefined functions are called by the ISR, which acts like a task dispatcher.

This scheduling methodology is highly optimized and was necessary to meet the very strict timing requirements of this RailCab application. With a more powerful microcontroller, an alternative solution could be taken into consideration: instead of manually balancing the processing time in different rounds, a real-time operating system could be used for handling the scheduling of tasks with different activation periods (e.g., 250 μs for motor control, 4 ms for status retrieval, etc.). The two presented implementation alternatives, however, have one thing in common: they both profit from the flexibility of the TrailCable protocol. Whether TDMA or dynamic scheduling is used, the TrailCable protocol can be perfectly suited to the chosen approach since it can provide real-time communication regardless of software paradigms.

The three function groups: motor control, status identification, and data tunneling will be described in the following.

8.3.3.1 Motor Control

The main objective of the new communication infrastructure implemented at the RailCab test track is to control the primary motor segments in order to allow operation of the shuttles. In a normal scenario, the reference values for the primary segments are defined by shuttles moving over the corresponding track section. The packets sent by a shuttle to the test track via radio basically include the following information: shuttle position and required primary current and frequency. With the shuttle position it is possible to decide, at the track, which primary segments must be activated. Usually, up to three primary segments are activated simultaneously in order to guarantee at all times the presence of a magnetic field under a moving shuttle.

In addition to the primary current and frequency values obtained from the shuttles, the activated servo devices require also a phase reference value. In each 250 μs round, this reference phase value is updated, generating a sinusoidal signal at the primary segments. The phases of all servo devices must be synchronized with each other in order to ensure a smooth signal transition between two primary segments.

Data packets are sent by the shuttles via the wireless interface every 20 ms. The test track components acknowledge reception of the packets to the shuttles in order to allow error detection in the radio interface and also to inform the shuttles about the test track status.

In order to allow a simple transition from the centralized to the distributed control architecture it was chosen to keep the radio modems initially in the power station 1 and also to use the local ECU as a master for motor control functions. This ECU gathers data from the shuttles (via the wireless interface) and user commands (via the RailControl utility) to control the servo devices properly. The master ECU node broadcasts the information on which servos are active and the associated reference values to all others. Since clock synchronization is provided, all ECUs are able to simultaneously set the appropriate values to the servos, which as a consequence allows the magnetic field of the motors to be generated without abrupt discontinuities on different primary segments.

The given configuration requires the following communication among the ECUs (Figure 8.26): a connection between the ECUs of the control room and station 1 as well as connections between the ECU of station 1 and the remaining ones. The first one is used to transfer commands and configuration received from the RailControl tool and forward it to the ECU in station 1. The second one is responsible for broadcasting the set-point of the servo devices to all ECUs. In order to allow a fault tolerant communication of all channels, data is always sent clockwise and counter-clockwise into the test track fiber-optic ring. Should a long or permanent link communication failure occur, immediate actions, such as shutting the motors down, can be taken by the ECUs in order to avoid unsafe behavior. The same applies in the case clock synchronization is lost or the semantics of data packet contents are incorrect.

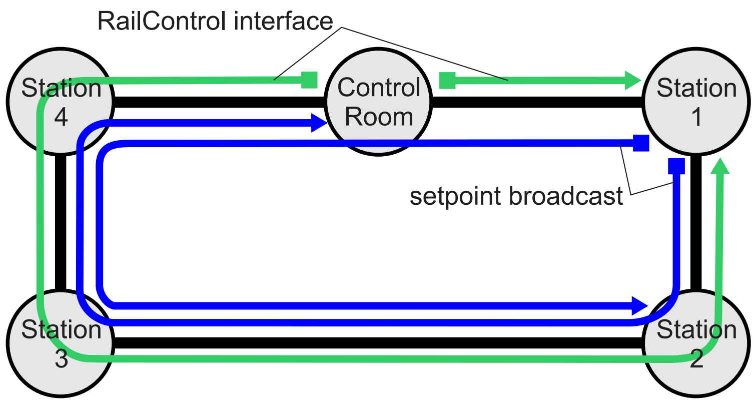


Figure 8.26: Virtual communication channels for motor control

The TrailCable data packets of the motor control functions have a payload of 41 bytes and are transmitted every 4 ms. The channel deadline is 250 μs, allowing a packet that is sent at the beginning of a round to be processed in the consecutive round at the destination ECU. During design, the TrailCable Verifier tool was employed to check the feasibility of all data packets transmitted via the TrailCable network, including the communication channels for motor control, status identification, and data tunneling functions. The latter two will be described below.

8.3.3.2 Status Identification

The status identification function group is responsible for retrieving information about the operational condition of a variety of system components. It is possible, for example, to monitor the reference values used by all active servos and to check whether communication links are working properly, the number of communication errors, synchronization status, link propagation delays, and even operational parameters of the servo devices. The ECUs gather information locally and transmits status data packets to the control room to be displayed in the PC running the RailControl and RailView tools.

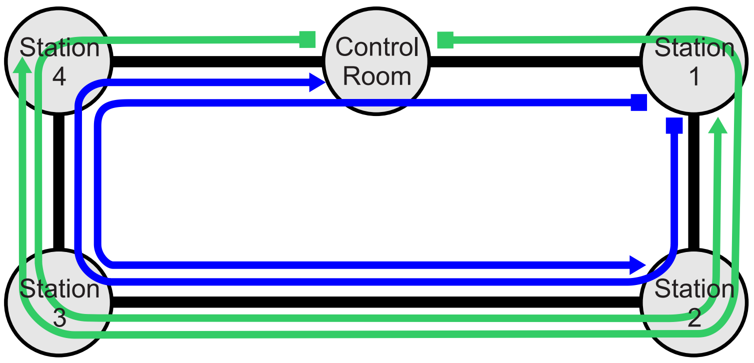


Figure 8.27: Virtual communication channels for status monitoring

In order to establish clock synchronization among all ECUs it is desirable that all nodes participate in this process. The use of the status data packets for this purpose turned out to be appropriate since most of the ECUs are required to transmit status data anyway. The only necessary extension was the configuration of all ECUs to transmit their status data packets not only to a single destination but also to all other nodes in the network. Because the clock synchronization mechanism is executed primarily by the TrailCable hardware, only a small overhead was added to the software running in the ECUs. As with the motor control packets, the status channels were built in a redundant manner: each ECU sends packets both clockwise and counter-clockwise to all other nodes. For the sake of simplicity only the channels generated at two nodes are presented in Figure 8.27, but all other remaining ones also transmit their data to the others via two channels. The clock synchronization process takes into account only the timing information of the first correctly received packet. The payload of the status data packets is 70 bytes, and as with the motor control, packets are transmitted with a period of 4 ms and have a channel deadline of 250 μs. Indeed, motor control and status packets are always sent simultaneously by the ECUs at the beginning of each 4 ms cycle.

8.3.3.3 Data Tunneling

The last function group is data tunneling. The goal here is to capture any bytes transmitted via the PC in the control room on its RS-232 interface and to forward them via the test track network to a selected servo, which is connected via another RS-232 interface. The reciprocal operation is also available, so that data from a servo can be transmitted back to the PC. Thus, the *Drive Manager* utility running in the PC can access a servo as if it were locally and directly connected via an RS-232 cable. In Section 8.3.1 we explained how an ECU establishes communication with a servo. The objective now is to demonstrate how the process takes place with the TrailCable-based communication ring.

The ECU in the control room acts as a master when it comes to the data tunneling functions. This ECU has the information from the servo selected by the user (via the RailControl tool) for the tunneling mechanism and is directly connected to the PC running the *Drive Manager* utility. The ECU is therefore able to create data packets containing both the selected servo and any data received from the PC and to broadcast it to the whole network. The ECU, by which the selected servo is controlled, then forwards the received data to the appropriate SSI module. The communication channels of the data tunneling function follows the same strategy as the other functions, i.e., packets are sent in a redundant manner via alternative routes.

Transmission from the remote ECUs to the master requires a different strategy. If individual communication channels were to be established between each of the remote ECUs and the master, the required bandwidth could even impair the feasibility of the real-time communication tasks. The solution was to profit from one of the TrailCable main features, namely dynamic reconfiguration. The idea is to establish only one communication channel at a time between one remote ECU and the master. All other ECUs remain in stand-by mode, waiting for a reconfiguration command.

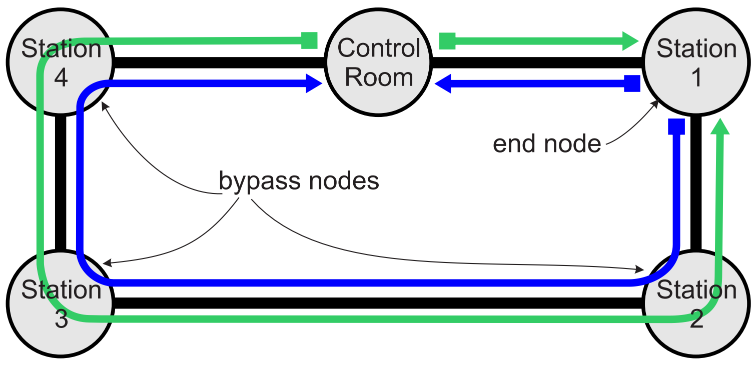


Figure 8.28: Virtual communication channels for data tunneling

The reconfiguration scheme (Figure 8.28) is based on two configurations for the data tunneling packets. In the first one, each remote ECU operates in a bypass mode, allowing a tunneling packet received in one of its ports to be forwarded to the opposite one.

In the second possible configuration, an ECU is set as the end node; it is responsible for generating tunneling packets (with data received from the servos) and sending them via its communication ports. During normal operation, the ECU corresponding to the selected servo works as a source and all other remote ECUs operate in the bypass mode. Should the selected servo change at run-time, the communication configuration of all remote ECUs would immediately be altered as well. This reconfiguration method allows therefore all ECUs to share common real-time channels and thus to reduce the amount of required communication resources.

Once again, redundant communication channels were employed for a fault-tolerant operation. The tunneling data packets carry 18 payload bytes and have also a 250 μs channel deadline. However, as opposed to the previously described data packets for the motor control and status identification function groups, the transmission period for the data tunneling packets is 2 ms. It was chosen to run the tunneling process twice in the 4 ms cycle, so that both the end-to-end latency time and the data packet size could be reduced.

8.4 Chapter Summary

This chapter has shown how the TrailCable protocol was employed with the RailCab test track facility. The use of such technology allowed low transmission latencies (below 250 μs), redundant communication and reconfiguration capabilities. These features represent a promising alternative for interconnecting RailCab track components, especially when larger systems are to be built. Scalability is a major concern in similar applications and imposes different design challenges for hard-real-time embedded networks. The TrailCable protocol supports the task of building up scalable and complex networks for different reasons. Firstly, it dispenses with the necessity of a global communication scheduling, which, however, can still be used for a small set of neighboring nodes. Furthermore, the communication channels of unrelated functions can be independently designed and integrated into the system. Also, due to its flexible topology, the task of cabling and creating distributed architectures can be easily supported. Further ideas of how to employ the TrailCable in larger networks are presented in [6].

Chapter 9

Conclusion

This thesis presented a new communication protocol named TrailCable, aimed at providing flexible and reliable hard-real-time communication for distributed embedded systems. TrailCable was designed bearing in mind safety aspects in order to broaden its utilization potential in applications such as critical distributed control systems. TrailCable is based on the sporadic triggering ($\mathcal{ST}$) paradigm, which closes the gap between event- and time-triggered approaches.

A TrailCable network consists of multiple nodes interconnected by full-duplex, point-to-point links. Nodes may have more than one communication port, thus acting as data switches. In order to guarantee real-time behavior and increase bandwidth utilization, packet scheduling is done with EDF scheduling, which has been proved to be optimal in the sense of feasibility. One contribution of this thesis is a mechanism that calculates the absolute deadlines for the EDF schedulers based on the characteristics of previous packets. This mechanism also integrates the so-called bandwidth guardian function that checks whether communication conforms to the parameters used for schedulability analysis. Additional services provided by the protocol include fault-tolerant clock synchronization and non-real-time communication.

A detailed model of a TrailCable network, described by means of XML files, is used by tools to perform an automated schedulability analysis of the proposed configuration. Should the acceptance tests pass, C-code containing the configuration of each node will automatically be generated.

One of the main features of the TrailCable protocol is the support for dynamic reconfiguration, which allows for real-time channels to be added, removed, or altered, at run-time. Dynamic reconfiguration takes advantage of the EDF algorithm for allocating packets to the transmission links. So, no global clock synchronization is required for coordinating medium accesses and parts of the network can be configured independently of others, thus reducing complexity.

This work introduced an ILP-based approach to optimally map real-time channels in a given network topology. Since EDF is optimal as well, the bandwidth utilization of a TrailCable network can therefore be very efficiently allocated to real-time channels. With the TrailCable communication model, virtual channels, provided that they are real-time feasible, can: 1) originate at any node, 2) reach destinations via multiple paths, 3) have activation rates, deadlines, and packet sizes independent of other channels. Taking this model into consideration, one can state that TrailCable is one of the most efficient protocols when it comes to bandwidth utilization.

Another algorithm for mapping real-time channels, the *Fit Minimum Laxity First*, was presented as an alternative to the ILP approach. Although not optimal, *FMLF* has much lower run-time complexity and is hence more appropriate for dynamic channel allocation.

With EDF scheduling and clock synchronization, the TrailCable protocol can be employed in different manners. It can be either used in such a way that packets that arrive earlier than their deadlines are immediately processed, but at the disadvantage of introducing jitter, or also be programmed to periodically process packets at globally known times, which on average will require a slightly higher transmission latency but reduces the jitter to the precision of the clock synchronization, typically below 1 μs. This latter approach allows a time-triggered operation of the application software while the communication infrastructure operates with sporadic triggering.

When it comes to scalability, one issue is the hardware implementation of EDF schedulers and the internal switch logic. The hardware architecture presented is able to handle up to 1792 (7 communication ports, 256 tasks each) simultaneous real-time packets, a number that is limited only by the number of IDs reserved in the initial protocol implementation. Not only is an increase in the number of IDs possible, but the schedulers are also scalable to cope with such an increased demand. The main components of the schedulers, the priority queues, were built up in such manner that insertion and removal operations have a run-time complexity of $O(1)$ and require a chip area proportional to the number of tasks.

Given the low overhead and the preemptive EDF scheduling, relatively low communication latency times can be achieved with TrailCable. In order to prove the theoretical bounds, Chapter 7 presented measurement experiments. For a data rate of 32 Mpbs, latency times below 10 μs could be achieved. The results also confirmed that the expected forwarding time of the TrailCable nodes operating at 32 Mbps is even less than that of some commercial Ethernet switches operating at 100 Mpbs. Moreover, experiments have shown the efficacy of the bandwidth guardian mechanism, which filters out non-conform packets in order to ensure the timely delivery of correct ones. Experiments based on the DoR method have shown how the TrailCable behaves with respect to jitter and communication skew.

Finally, a real application employing TrailCable was presented in Chapter 8: the RailCab test track. In this particular case the system software was built using a cyclic executive, which runs in a synchronous manner in all nodes. The maximum deviation of the clock synchronization in this example was measured in a setup similar to that in Chapter 7 and turned out to be lower than 500 ns. In this specific application, more than 80 servo drives could therefore be triggered with only a marginal phase difference. Additionally, the application benefited also from the dynamic reconfiguration feature of TrailCable. It was possible to alter, during run-time, real-time communication channels used for a tunneling service between the control room and the power stations.

9.1 Outlook

Hard-real-time communication is a topic with many open research directions. This thesis represents a contribution in this field and can be used as a basis for further investigations. Among the many possibilities, some areas of interest are listed below:

Membership Service. Since TrailCable ensures bounded message transmission delays and has a distributed clock synchronization service, it provides the necessary conditions for building up a membership service. Also, redundancy management can be rather simply implemented by checking, at the final receivers, redundant packets for correctness in order to pass a single instance of the message to the application layer. Membership service and redundancy management are the basis for building up a fault-tolerance abstraction layer to further reduce the complexity of the application software.

Extension of the $\mathcal{ST}$ Paradigm to Application Software Processes. Transmission with TrailCable is done by successively scheduling packets in one node after the other. At the last node it is possible, for example, for the communication protocol to trigger the processor task that receives the packet in a similar manner, given that the real-time operating system at the host also works with the EDF algorithm. The same can be applied to transmitting software process. This interaction between communication and processing systems may contribute to additional efficiency gains, especially with system-on-a-chip (SoC) implementations.

Real-Time Mapping Algorithms. In chapter 5 two approaches for mapping real-time channels on the network were presented: one optimal and the other with lower run-time complexity. There is a research potential in this field to find new techniques for allocating communication tasks in the network according to different kinds of constraints. It is possible to go even further and evaluate algorithms that also find a suitable network topology in an autonomous manner. If communication requirements and constraints are given, it can be possible to automatically generate a network configuration, including physical topology, channel mapping, and deadline assignments.

Mixed Criticality. The TrailCable architecture was designed in such a manner that although real-time and non-real-time packets are able to share the same transmission medium, they are managed independently in the nodes. This thesis has focused on the real-time part, which is the most critical part, and opens up the possibility to integrate different best-effort algorithms to handle non-real-time traffic in an efficient manner.

The current protocol features, the possibility to integrate other ones, and its suitability in a wide range of applications give TrailCable the credentials of a real “trailblazer” in the design of flexible hard-real-time communication systems. I hope that this work can inspire other advancements in the area and thus contribute to innovative application ideas.

Appendix A

Network-modeling DTD Files

This appendix presents the four DTD files used to parse the XML-based models of a TrailCable network configuration.

A.1 Communication Engine Properties

```
<!ELEMENT Implementation                            ANY>
<!ATTLIST Implementation timeResolution             NMTOKEN #REQUIRED>
<!ATTLIST Implementation defaultForwardingDelay     NMTOKEN #REQUIRED>
<!ATTLIST Implementation defaultLinkPropagationDelay NMTOKEN #REQUIRED>
<!ATTLIST Implementation maximumTasks               NMTOKEN #REQUIRED>
<!ATTLIST Implementation deviation                  NMTOKEN #REQUIRED>
<!ATTLIST Implementation maxPeriodXdeadlineRatio    NMTOKEN #REQUIRED>
<!ATTLIST Implementation maximumPeriod              NMTOKEN #REQUIRED>
<!ATTLIST Implementation maximumPayloadSize         NMTOKEN #REQUIRED>
<!ATTLIST Implementation maximumPacketSize          NMTOKEN #REQUIRED>
<!ATTLIST Implementation preemptionHeaderSize       NMTOKEN #REQUIRED>
<!ATTLIST Implementation transmissionRate           NMTOKEN #REQUIRED>
<!ATTLIST Implementation maxHardwareMemory          NMTOKEN #REQUIRED>
<!ATTLIST Implementation maxHostAccessedMemory      NMTOKEN #REQUIRED>
```

A.2 Network Topology

```
<!ELEMENT Graph           (Connection+, (NodeInformation | Host)*)>
<!ATTLIST Graph           numNodes               NMTOKEN #REQUIRED>
<!ATTLIST Graph           maxPorts               NMTOKEN #REQUIRED>

<!ELEMENT Connection                             EMPTY>
<!ATTLIST Connection      node1                  NMTOKEN #REQUIRED>
<!ATTLIST Connection      node2                  NMTOKEN #REQUIRED>
<!ATTLIST Connection      port1                  NMTOKEN #REQUIRED>
<!ATTLIST Connection      port2                  NMTOKEN #REQUIRED>
<!ATTLIST Connection      linkPropagationDelay   NMTOKEN #IMPLIED>

<!ELEMENT NodeInformation                        EMPTY>
<!ATTLIST NodeInformation node                   NMTOKEN #REQUIRED>
<!ATTLIST NodeInformation forwardingDelay        NMTOKEN #REQUIRED>

<!ELEMENT Host                                   EMPTY>
<!ATTLIST Host            name                   NMTOKEN #REQUIRED>
<!ATTLIST Host            node                   NMTOKEN #REQUIRED>
<!ATTLIST Host            port                   NMTOKEN #REQUIRED>
```

A.3 Real-Time Tasks

```
<!ELEMENT ChannelList                        (Channel+)>
<!ATTLIST ChannelList    graph               NMTOKEN #REQUIRED>
<!ATTLIST ChannelList    HWproperties        NMTOKEN #REQUIRED>
<!ATTLIST ChannelList    StaticRoute         NMTOKEN #REQUIRED>
<!ATTLIST ChannelList    DynamicRoute        NMTOKEN #IMPLIED>

<!ELEMENT Channel                            (TargetHost+)>
<!ATTLIST Channel        id                  NMTOKEN #REQUIRED>
<!ATTLIST Channel        sourceHost          NMTOKEN #REQUIRED>
<!ATTLIST Channel        period              NMTOKEN #REQUIRED>
<!ATTLIST Channel        payloadSize         NMTOKEN #REQUIRED>
<!ATTLIST Channel        redundancyMode      (0)         "0" >
<!ATTLIST Channel        clockDomain         (0|1|2|3)   "0" >
<!ATTLIST Channel        sync                (0|1)       "0" >
<!ATTLIST Channel        description         CDATA   #IMPLIED>

<!ELEMENT TargetHost                         EMPTY>
<!ATTLIST TargetHost     host                NMTOKEN #REQUIRED>
<!ATTLIST TargetHost     deadline            NMTOKEN #REQUIRED>
```

A.4 Routing

```
<!ELEMENT RouteList                                   (ChannelRoute+)>

<!ELEMENT ChannelRoute                                (Path+)>
<!ATTLIST ChannelRoute   channelID                    NMTOKEN #REQUIRED
                         defaultRelativeDeadline      NMTOKEN #REQUIRED
                         defaultDestinationTaskID     NMTOKEN #REQUIRED>

<!ELEMENT Path                                        EMPTY>
<!ATTLIST Path           from                         NMTOKEN #REQUIRED
                         to                           NMTOKEN #REQUIRED
                         sourcePort                   NMTOKEN #IMPLIED
                         relativeDeadline             NMTOKEN #IMPLIED
                         destinationTaskID            NMTOKEN #IMPLIED>
```

Author's Publications

[1] Francisco, A. L. d. F. A Hardware Implementation of an EDF-based Communication Protocol. In *Proc. of 10th Brazilian Workshop on Embedded and Real-Time Systems (WTR 2008).* Rio de Janeiro, Brazil, May 2008.

[2] Francisco, A. L. d. F. and Diez, M. An EDF-Based, Hard-Real-Time Communication Engine for Distributed Embedded Systems. In *IFIP International Conference on Network and Parallel Computing (NPC 2006).* Tokyo, Japan, October 2006.

[3] Francisco, A. L. d. F. Towards Dynamically Reconfigurable Hard-Real-Time Communication for Embedded Mechatronic Systems. In *36th Annual IEEE/IFIP International Conference on Dependable System and Networks - Student Forum.* Philadelphia, PA, USA, June 2006.

[4] Francisco, A. L. d. F. Resource-Efficient FPGA-Based Priority Queues. In *VLSI-SoC - PhD Forum.* Perth, Australia, October 2005.

[5] Francisco, A. L. d. F. and Rammig, F. J. Fault-Tolerant Hard-Real-Time Communication of Dynamically Reconfigurable, Distributed Embedded Systems. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing.* ISORC '05, Washington, DC, USA, pp. 275–283, IEEE Computer Society. 2005.

[6] Francisco, A. L. d. F., Schulz, B., and Henke, C. Towards a Real-Time Communication Network for Autonomous Rail Vehicles. In *From Specification to Embedded Systems Application* (Rettberg, A., Zanella, M. C., and Rammig, F. J., eds.). Vol. 184 of *IFIP International Federation for Information Processing.* pp. 245–254, Springer Boston. 2005.

[7] Francisco, A. L. d. F., Rettberg, A., and Hennig, A. Hardware Design and Protocol Specification for the Control and Communication within a Mechatronic System. In *Distributed and Parallel Embedded Systems.* DIPES '04, Toulouse, France, pp. 113–122, August 2004.

[8] Zanella, M. C., Robrecht, M., Francisco, A. L. d. F., Horst, A., Lehmann, T., and Gielow, R. S. RABBIT - A Modular Rapid Prototyping Platform for Distributed Mechatronic Systems. In *Proceedings of the 14th symposium on Integrated circuits and systems design.* SBCCI '01, IEEE Computer Society 2001.

Bibliography

[9] Adelt, P., Donoth, J., Geisler, J., Henkler, S., Kahl, S., Klöpper, B., Münch, E., Oberthür, S., Paiz, C., Podlogar, H., Porrmann, M., Radkowski, R., Romaus, C., Schmidt, A., Schulz, B., Voecking, H., Witkowski, U., and Witting, K. *Selbstoptimierende Systeme des Maschinenbaus – Definitionen, Anwendungen, Konzepte.* No. 234, HNI Verlagsschriftenreihe, Paderborn 2009.

[10] Ademaj, A., Bauer, G., Sivencrona, H., and Torin, J. Evaluation of Fault Handling of the Time-Triggered Architecture with Bus and Star Topology. In *Proc. of International Conference on Dependable Systems and Networks (DSN 2003).* pp. 123–132, 2003.

[11] Albers, K. and Slomka, F. Efficient Feasibility Analysis for Real-Time Systems with EDF Scheduling. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1.* DATE '05, Washington, DC, USA, pp. 492–497, IEEE Computer Society 2005.

[12] Albert, A. Comparison of Event-Triggered and Time-Triggered Concepts with Regard to Distributed Control Systems. *Embedded World.* pp. 235–252, 2004.

[13] Albert, A., Wolter, B., and Gerth, W. Distinctness of Reaction - Ein Messverfahren zur Beurteilung von Echtzeitsystemen (Teil 2). In *at - Automatisierungstechnik.* pp. 445–452, Oldenbourg Wissenschaftsverlag, October 2003.

[14] ARINC. *ARINC Specification 429P1-17 Mark 33 Digital Information Transfer System (DITS), Part 1, Functional Description, Electrical Interface, Label Assignments and Word Formats.* 2004.

[15] ARTEMIS Joint Undertaking. "*Embedded Systems.*"
On-line: http://www.artemis-ju.eu/embedded_systems. December 2011.

[16] Avago Technologies. "Inexpensive DC to 32 MBd Fiberoptic Solutions for Industrial, Medical, Telecom, and Proprietary Data Communication Applications. Application Note 1121."

[17] Baruah, S. K., Howell, R. R., and Rosier, L. E. Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor. In *Real-Time Systems Symposium.* Lake Buena Vista, FL, USA, December 1990.

[18] Bayliss, S., Bouganis, C.-S., Constantinides, G. A., and Luk, W. An FPGA Implementation of the Simplex Algorithm. *IEEE International Conference on Field-Programmable Technology.* 2006.

[19] Bini, E. and Buttazzo, G. The space of EDF deadlines: the exact region and a convex approximation. *Real-Time Systems.* vol. 41, pp. 27–51, January 2009.

[20] Broy, M. Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering.* ICSE '06, New York, NY, USA, pp. 33–42, ACM 2006.

[21] Buttazzo, G. C. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications.* Norwell, MA, USA: Kluwer Academic Publishers 2000.

[22] Buttazzo, G. C. Rate monotonic vs. EDF: judgment day. *Real-Time Systems,* vol. 29, no. 1, pp. 5–26, 2005.

[23] CAN in Automation e.V. "*CAN Newsletter special issue automotive.*"
On-line: http://www.can-cia.de/index.php?id=416. 2006.

[24] CAN in Automation e.V., H. "*25 years of CAN.*"
On-line: http://www.can-cia.de/index.php?id=1344.

[25] Condor Engineering Inc. *AFDX / ARINC 664 Tutorial* (1500-049). May 2005.

[26] Cook, B. M. and Walker, P. H. SpaceWire and IEEE 1355 Revisited. In *International SpaceWire Conference.* 2007.

[27] Corbet, J., Rubini, A., and Kroah-Hartman, G. *Linux Device Drivers, 3rd Edition.* O'Reilly Media, Inc. 2005.

[28] Cristian, F. and Fetzer, C. The Timed Asynchronous Distributed System Model. In *IEEE Transactions on Parallel and Distributed Systems.* pp. 603–618, June 1999.

[29] Diestel, R. *Graph Theory.* Springer-Verlag 2005.

[30] Dolev, D. The Byzantine Generals Strike Again. In *Journal of Algorithms.* Vol. 3 pp. 14–30, March 1982.

[31] ECSS. *SpaceWire - Links, nodes, routers and networks.* 2008.

[32] Elmenreich, W., Bauer, G., and Kopetz, H. The time-triggered paradigm. *Proceedings of the Workshop on Time-Triggered and Real-Time Communication.* 2003.

[33] Ettingshausen, C., Hestermeyer, T., and Otto, S. Aktive Spurführung und Lenkung von Schienenfahrzeugen. *6. Magdeburger Maschinenbautage, Intelligente technische Systeme und Prozesse - Grundlagen, Entwurf, Realisierung.* 2003.

[34] FlexRay Consortium. *FlexRay Electrical Physical Layer Specification v3.0.1.* 2010.

[35] FlexRay Consortium. *FlexRay Protocol Specification v3.0.1.* 2010.

[36] Freescale Semiconductor Inc. "*MPC555: 32-bit Power Architecture Microcontroller.*"
On-line: http://www.freescale.com/files/microcontrollers/doc/user_guide/MPC555UM.pdf. 2010.

[37] Griese, B. *Adaptive Echtzeitkommunikationsnetze.* PhD thesis . University of Paderborn. 2009.

[38] Hart, P. E., Nilsson, N. J., and Raphael, B. Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *SIGART Bull.* pp. 28–29, December 1972.

[39] Henia, R., Hamann, A., Jersak, M., Racu, R., and Kai Richter, R. E. *System Level Performance Analysis - the SymTA/S Approach.* ch. 2, pp. 29–72,. The Institution of Electrical Engineers, London, United Kingdom 2006.

[40] Hestermeyer, T. *Strukturierte Entwicklung der Informationsverarbeitung fr die aktive Federung eines Schienenfahrzeugs.* PhD thesis . University of Paderborn. 2006.

[41] IEEE Standards Association. *1355-1995 - IEEE Standard for Heterogeneous InterConnect (HIC), (Low-Cost, Low-Latency Scalable Serial Interconnect for Parallel System Construction).* 1995.

[42] IEEE Standards Association. *1588-2008 - IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems.* 2008.

[43] IEEE Standards Association. *IEEE 802.3-2008 Ethernet.* 2008.

[44] International Organization for Standardization. ISO 11898-1:2003 Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling. 2003.

[45] International Organization for Standardization. ISO 11898-4:2004 Road vehicles – Controller area network (CAN) – Part 4: Time-triggered communication. 2004.

[46] Ioannou, A., Manolis, and Katevenis, M. Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High-Speed Networks. In *In IEEE/ACM Transactions on Networking.* pp. 2043–2047, 2001.

[47] Janssen, D. and Büttner, H. EtherCAT Der Ethernet-Feldbus. *Elektronik.* vol. 23 2003.

[48] Jasperneite, J. and Gamper, S. Echtzeit-Betrieb im Ethernet - Industrial Ethernet Switches auf dem Prüfstand - Teil1. *Elektronik.* vol. 7, pp. 50–92, Apr 2007.

[49] Jasperneite, J. and Gamper, S. Echtzeit-Betrieb im Ethernet - Industrial Ethernet Switches auf dem Prüfstand - Teil2. *Elektronik.* vol. 16, pp. 60–65, Aug 2007.

[50] Kandlur, D. D. and Shin, K. G. Design of a Communication Subsystem for HARTS. *Tech. Rep.* 1991.

[51] Kandlur, D. D., Shin, K. G., and Ferrari, D. Real-Time Communication in Multi-Hop Networks. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS).* Washington, DC, pp. 300–307, IEEE Computer Society May 1991.

[52] Kastensmidt, F. L., Carro, L., and Reis, R. *Fault-Tolerance Techniques for SRAM-Based FPGAs.* Springer 2006.

[53] Kim, B. K. and Shin, K. G. Scalable hardware earliest-deadline-first scheduler for ATM switching networks. In *IEEE Real-Time Systems Symposium.* pp. 210–, 1997.

[54] Kim, T., Shin, H., and Chang, N. Deadline Assignment To Reduce Output Jitter Of Real-Time Tasks. In *IP Project, www.itpolicy.gsa.giv.* pp. 67–72, 2000.

[55] Koopman, P. and Chakravarty, T. Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04).* Washington, DC, USA, p. 145, IEEE Computer Society June 2004.

[56] Kopetz, H. and Ochsenreiter, W. Clock synchronization in distributed real-time systems. *IEEE Trans. Comput.* vol. 36, pp. 933–940, August 1987.

[57] Kopetz, H. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Norwell, MA, USA: Kluwer Academic Publishers. 1st ed. 1997.

[58] Kopetz, H., Ademaj, A., Grillinger, P., and Steinhammer, K. The Time-Triggered Ethernet (TTE) Design. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing.* ISORC '05, Washington, DC, USA, pp. 22–33, IEEE Computer Society 2005.

[59] Lamport, L., Shostak, R., and Pease, M. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems,* pp. 382–441, July 1982.

[60] Liu, C. L. and Layland, J. W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM,* vol. 20, no. 1, pp. 46–61, 1973.

[61] Liu, J. W. S. *Real-Time Systems.* Prentice Hall. 2000.

[62] LTi Drives GmbH. "*Drive Manager 3.*"
On-line: http://drives.lt-i.com/Produkte/PC-Bedienoberflaeche/
DRIVE-MANAGER-3/4123/ 2010.

[63] Lundelius, J. and Lynch, N. A new fault-tolerant algorithm for clock synchronization. In *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing.* New York, NY, USA, pp. 75–88, ACM Press. June 1984.

[64] Masrur, A., Drössler, S., and Färber, G. Improvements in Polynomial-Time Feasibility Testing for EDF. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE).* Munich, Germany, 2008.

[65] Michel Berkelaar, Kjell Eikland, P. N. Open source (mixed-integer) linear programming system - lp_solve. May 2004.

[66] Moore, S. W. and Graham, B. T. Tagged up/down sorter - A hardware priority queue. *The Computer Journal.* vol. 38, pp. 695–703, 1995.

[67] Murata, Y., Kogo, T., and Yamasaki, N. A SpaceWire Extension for Distributed Real-Time Systems. In *International SpaceWire Conference.* 2010.

[68] Neue Bahntechnik Paderborn. "RailCab." On-line: www.railcab.de. 2006.

[69] Parkes, S. and Ferrer, A. SpaceWire-RT. In *International SpaceWire Conference.* 2008.

[70] Pedreiras, P., Almeida, L., and Gai, P. The FTT-Ethernet Protocol: Merging Flexibility,Timeliness and Efficiency. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems.* Washington, DC, USA, pp. 152–, IEEE Computer Society 2002.

[71] PHYTEC Messtechnik GmbH. “*phyCORE-MPC555 System on Module.*” On-line: http://www.phytec.com/products/som/PowerPC/phyCORE-MPC555.html 2010.

[72] Pottharst, A., Henke, C., Schneider, T., Böcker, J., and Grotstollen, H. Drive Control and Position Measurement of RailCab Vehicles Driven by Linear Motors. *Int. Symposium on Instrumentation and Control Technology (ISICT), Beijing, China.* 2006.

[73] Pottharst, A. *Energietechnik und Leittechnik einer Anlage mit Linearmotor getriebenen Bahnfahrzeugen.* PhD thesis . University of Paderborn. 2005.

[74] PROFIBUS Nutzerorganisation e.V. “PROFINET Overview.” On-line: http://www.profibus.com/technology/profinet/overview/ 2010.

[75] Rexford, J. and Shin, K. G. Scalable hardware priority queue architectures for high-speed packet switches. In *IEEE Transactions on Computers.* pp. 1215–1227, IEEE 1997.

[76] Robert Bosch GmbH. “*FlexRay Communication Controller IP.*” On-line: http://www.semiconductors.bosch.de/en/ipmodules/flexray/flexray.asp. 2010.

[77] SAE International. Class C Application Requirement Considerations - J2056/1. 1993.

[78] Sawyer, N. Data Recovery. Application Note XAPP224. tech. rep. Xilinx, Inc 2005.

[79] Schneider, T., Schulz, B., Henke, C., and Böcker, J. Redundante Positionserfassung für ein spurgeführtes linearmotorgetriebenes Bahnfahrzeug. *Workshop Entwurf mechatronischer Systeme, Heinz-Nixdorf-Institut, Universitt Paderborn* 2006.

[80] Short, M. The Case For Non-preemptive, Deadline-driven Scheduling In Real-time Embedded Systems. *World Congress on Engineering.* 2010.

[81] Siemens AG. *PROFINET Systembeschreibung.* 2008.

[82] Spuri, M., Buttazzo, G., and Anna, S. S. S. Scheduling Aperiodic Tasks in Dynamic Priority Systems. *Real-Time Systems,* vol. 10, pp. 179–210, 1996.

[83] Telecommunications Industry Association. *Electrical Characteristics of Low Voltage Differential Signaling (LVDS) Interface Circuits.* February 2001.

[84] TTTech Computertechnik AG. TTP/C Protocol Specification. 2003.

[85] Turek, J. and Shasha, D. The Many Faces of Consensus in Distributed Systems. *Computer,* vol. 25, pp. 8–17, June 1992.

[86] Vienna University of Technology, Real-Time Systems Group. "TTP/C Project." On-line: http://www.vmars.tuwien.ac.at/projects/ttp/ttpc.html.

[87] Widmer, A. X. and Franaszek, P. A. A DC-Balanced, Partitioned-Block, 8B/10B Transmission Code. *IBM Journal of Research and Development,* vol. 27, no. 5, pp. 440–451, 1983.

[88] Wolter, B., Albert, A., and Gerth, W. Distinctness of Reaction - Ein Messverfahren zur Beurteilung von Echtzeitsystemen (Teil 1). In *at - Automatisierungstechnik.* pp. 396–403, Oldenbourg Wissenschaftsverlag September 2003.

[89] Xilinx Inc. "*ISE Design Suite Version 12.3.*"
On-line: http://www.xilinx.com/tools/designtools.htm. 2010.

[90] Xilinx Inc. "*Spartan-2 FPGA Family.*"
On-line: http://www.xilinx.com/support/documentation/spartan-ii.htm. 2010.

[91] Xilinx Inc. "*Spartan-3 FPGA Family.*"
On-line: http://www.xilinx.com/support/documentation/spartan-3.htm. 2010.

[92] Xilinx Inc. "*Spartan-6 FPGA Family.*"
On-line: http://www.xilinx.com/support/documentation/spartan-6.htm. 2010.

[93] Xilinx Inc. "*Virtex-E FPGA Family.*"
On-line: http://www.xilinx.com/support/documentation/virtex-e.htm. 2010.